Google Summer of Code 2021 Proposal : Boost.Real

Divyam Singal

April 13, 2021

Project - Boost.Real: Modifying internal representation of operation and improving math functions
Organisation - Boost C++
Mentors - Vikram Singh Chundawat, Laouen Belloli

1 Personal Details

- Name: Divyam Singal
- College/University: Indian Institute of Technology, Guwahati (IITG)
- Course/Major: Computer Science and Engineering (CSE)
- Degree Program: B.Tech
- Email: divyamsingal01@gmail.com, dsingal@iitg.ac.in
- Github Username: div5252
- Time Zone: Indian Standard Time (GMT +05:30)
- Availability:
  - How much time do you plan to spend on your GSoC?
    I plan to spend around 30 hours per week during the 10 weeks period of GSoC.
  - What are your intended start and end dates?
    I plan to start a bit early than the scheduled start date, around last week of May. I intend to finish the project by the first week of August so that we can have some buffer time.
  - What other factors affect your availability (exams, courses, moving, work, etc.)?
    My fifth semester will be starting from the beginning of August and the procedure for campus internships will also begin, so I will be busy during that period. That is why I intend to start early. Other than this, I would be able to give sufficient and required time for the project.

2 Background Information

2.1 Educational Background

I am a 2nd year Computer Science and Engineering student at Indian Institute of Technology, Guwahati. I have a minor in Robotics and Artificial Intelligence. I have a CPI (Cumulative Performance Index) of 9.70/10 in my major course and a CPI of 9.0/10 in my minor course, currently.

Some of the key courses taken are Data Structures and Algorithms, Design of Algorithms, System Software, Discrete Mathematics, Linear Algebra and Formal Languages, Automata Theory and Computation.

2.2 Programming Background

I am doing research internship under Prof. Deepanjan Kesh in the field of bitprobes, which is a set membership problem and is related to Data Structures and Algorithms. I have worked on two projects (OSS)-

1. Codeforces Crawler, a Web Development project using Django framework along with Beautiful Soup and chart.js.
2. Face detection system using YOLO algorithm, a Machine Learning project using Keras framework, OpenCV and Pillow library.

2.3 Programming Interests

I enjoy coding in C++ and regularly use it to participate in online competitive programming contests and to implement algorithms related to computer science and data structures. I started exploring Boost from December 2020. Concepts such as Template Meta Programming have really fascinated me.

I liked the idea of computable Real numbers and interval arithmetic and the Boost.Real library caught my eye. I have always been interested in Mathematics and find this as an opportunity wherein I'll be able to overlap my interests in programming and maths.

2.4 Plans beyond Summer of Code

Even after the GSoC period, I will continue to contribute to the Boost.Real library. There are some ideas which I proposed but can't be worked on in the GSoC timeline and I could work on these after the GSoC period.

2.5 Languages, Technologies and Tools

On the scale of 0-5, based on knowledge and my experience I would rate the following languages, technologies or tools as:

• C++ 98/03 (traditional C++):	3
• C++ 11/14 (modern C++):	4
• C++ Standard Library:	4
• Boost C++ Libraries:	3
• Git:	4

I am most familiar with Visual Studio and I use Windows Operating system. I can also work on Linux if needed, as I have Virtual Box and WSL installed. I also use VS Code as a text editor.

I am not familiar with Doxygen but I ensure that I will get familiar with Doxygen during/before the Community Bonding period.

3 Project Proposal

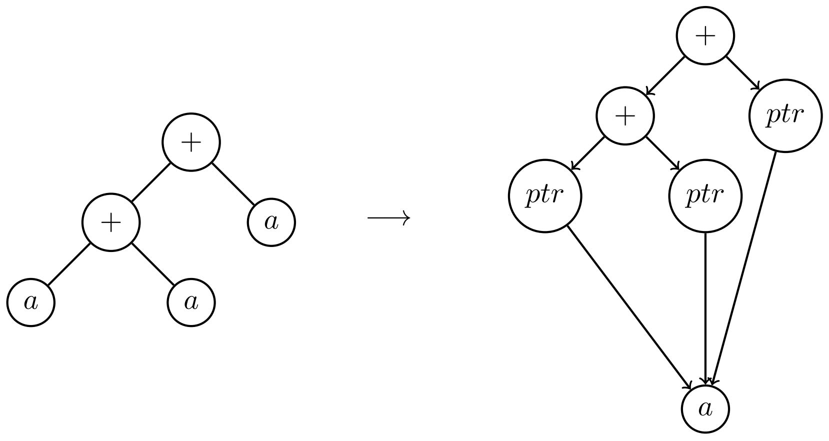
3.1 Expression Tree to DAG and Operation Optimizations

In our library, we have used an expression tree for Real operations. This structure is inefficient in terms of time and space. For example, consider the following snippet -

```
Real a(1);
for(int i = 0;i < 100; ++i)
    a = a + a;
```

This will create a tree for a with $2^{101} - 1$ nodes. A better implementation would be to use a Directed Acyclic Graph (DAG), which allows multiple references to the same node and avoid the unnecessary growth of size of structure.

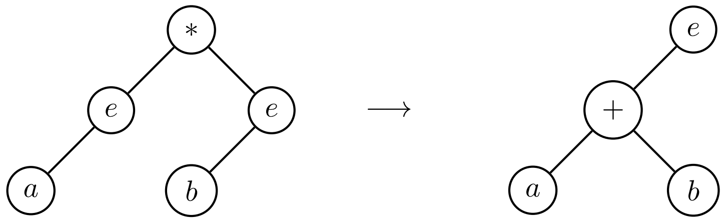
We will do so by making shared pointers to real a, which will reduce the memory. For example -



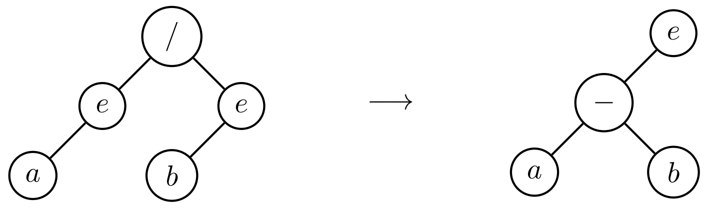
This implementation would still require the computation of a to a precision every time, leading to a large time complexity. This problem can be solved by careful caching of the approximations, making sure that the approximations to a node are asked with the same precision, and that the cached approximations are deleted when they are no longer needed. By counting their number and the number of requests already made, we can maintain efficient caching of all temporary results and delete them exactly when they are no longer needed.

We can make some arithmetic optimizations as follows -

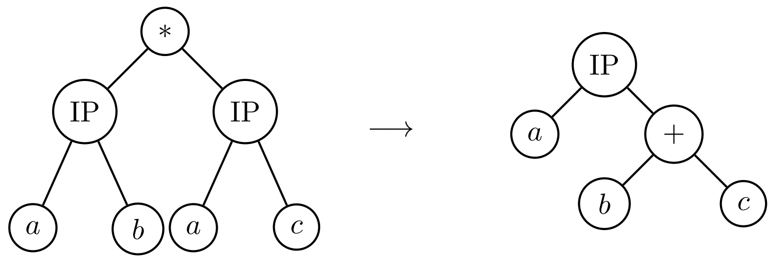
$$e^a e^b = e^{(a+b)}$$



$$e^a/e^b = e^{(a-b)}$$

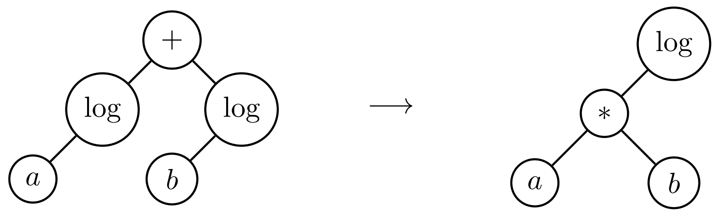


$$a^b a^c = a^{(b+c)}$$



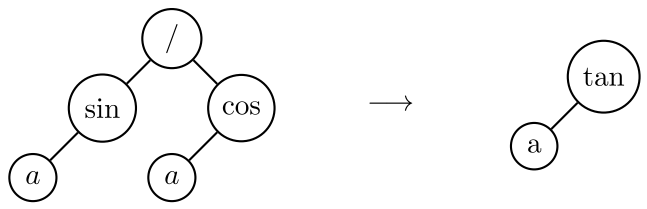
Similarly we can do optimization for $a^b/a^c = a^{(b-c)}$.

$$log(a) + log(b) = log(ab)$$



Similarly we can do optimization for $log(a) - log(b) = log(a/b)$.
The multiplication and division of trigonometric functions can also be optimised. For example -

$$\frac{sin(a)}{cos(a)} = tan(a)$$



The library implements some basic optimizations using the check_and_distribute function. In each of the above proposed optimizations we need to look at two levels at most. We can modify the check_and_distribute functions to include above proposed optimizations. For example, consider

$e^a e^b = e^{a+b}$. We first check if operation is multiplication and its operands are also real operation with exponent as its operation. If that is the case, we change the operation to exponent, and its lhs operand is also a real operation with addition as its operation.

3.2 Champernowne Irrational Improvement

While finding the n^{th} digit for of the Champernowne irrational number, what we are doing currently is to find all the previous digits before n, and so the time complexity is $O(n)$.

Instead of this, we can directly find the n^{th} digit without computing the previous digit as follows- We see that there are 1 1-digit binary numbers, 2 2-digit binary numbers, 4 3-digit binary numbers and so on. Using this we will be finding the number of digits of the number to which the n^{th} digit belongs to. Let the n^{th} digit be the part of binary number with $m+1$ digits.

So the digits covered till all m-digit numbers is -

$$\sum_{i=1}^{m} 2^{i-1} i = (m-1)2^m + 1 \leq n$$

$$(m-1)2^m \leq n-1$$

And here m can be found using binary search in $O(logn)$ time.

Having found m, we know that the n^{th} digit belongs to the $(m+1)$- digit binary number.
Digits left $= n - 1 - (m-1)2^m$. Then the binary number to which the n^{th} digit belongs is $2^{m+1} + \frac{\text{(Digits left)}}{m+1}$. And the position of n^{th} digit in the above number will be $m + 2 - \text{(Digits left)}\%(m+1)$. Finally the n^{th} digit can be found by number$\&(1 << \text{position})$.

So the time complexity of the algorithm is $O(logn)$ which is better than previous $O(n)$.

3.3 Adding e and golden ratio to irrationals

<u>e:</u> We will implement a function to get the n^{th} digit of e. We will not be using the Taylor series for the algorithm as the terms get smaller, the truncation error when they are added to the successive approximants of e will become larger. We will instead be using the following **Spigot** algorithm.

It is possible to reduce this error quite simply, by evaluating the series from the other end. This requires that the series be first truncated, for an infinite number of terms is unacceptable. This is a change of programming technique: first the required number of terms is computed, and then the series evaluated instead of an iterative loop with a testing exit.

We will rewrite e in a different way -

$$e = 1 + 1\left(1 + \frac{1}{2}\left(1 + \frac{1}{3}\left(1 + \frac{1}{4}\left(1 + \ldots\left(1 + \frac{1}{n}\right)\cdots\right)\right)\right)\right)$$

The method depends on the fact that except for the first two terms of the series, each successive term and their sum is less than 1. Suppose that a truncated portion of the series beyond the first two terms be multiplied by 10: it will then consist of an integer part and a fractional remainder series. The integer can be removed and the process is repeated, giving successively the fractional digits of e.

Let us understand this by an example -

$$\begin{aligned}
e &= 2 + \left[\frac{1}{2}\left(1+\frac{1}{3}\left(1+\frac{1}{4}\left(1+\frac{1}{5}\left(1+\frac{1}{6}(1+\dots)\right)\right)\right)\right)\right] \\
&= 2 + \frac{1}{10}\left[\frac{1}{2}\left(10+\frac{1}{3}\left(10+\frac{1}{4}\left(10+\frac{1}{5}\left(10+\frac{1}{6}(10+\dots)\right)\right)\right)\right)\right] \\
&= 2 + \frac{1}{10}\left[7+\frac{1}{2}\left(0+\frac{1}{3}\left(1+\frac{1}{4}\left(0+\frac{1}{5}\left(1+\frac{1}{6}(5+\dots)\right)\right)\right)\right)\right] \\
&= 2.7 + \frac{1}{100}\left[\frac{1}{2}\left(0+\frac{1}{3}\left(10+\frac{1}{4}\left(0+\frac{1}{5}\left(10+\frac{1}{6}(50+\dots)\right)\right)\right)\right)\right] \\
&= 2.7 + \frac{1}{100}\left[1+\frac{1}{2}\left(1+\frac{1}{3}\left(1+\frac{1}{4}\left(3+\frac{1}{5}\left(4+\frac{1}{6}(2+\dots)\right)\right)\right)\right)\right]
\end{aligned}$$

Note that we can use this method to calculate e to any desired base. A base of the form 2^n can lead to extremely simple multiplication by shifting. We can also perform the multiplication for decimal digits by 10^n, thus producing n digits at a time, increasing the efficiency.
This is the pseudo-code for the algorithm -

```
e_nth_digit(n):
    m = 4
    while(m * (ln(m) - 1) + 0.5 * ln(6.2831852 * m) <= (n + 1) * 2.30258509):
        m = m + 1
    for j from 2 until m
        coef[j] = 1
    d[0] = 2
    for i from 1 until n
        carry = 0
        for j from m to 2
            temp = coef[j] * 10 + carry
            carry = temp / j
            coef[j] = temp - carry * j
        d[i] = carry
    return d[n]
```

$\underline{\phi}$: Golden ratio, $\phi = \frac{1+\sqrt{5}}{2}$.
Now to compute the n^{th} digit, we can use the Newton's Method. The equation defining the Golden ratio is -

$$x^2 - x - 1 = 0 = x - 1 - \frac{1}{x}$$

Writing equation as the second form is slightly faster, in which case the Newton's iteration becomes

$$x_{n+1} = \frac{x_n^2 + 2x_n}{x_n^2 + 1}$$

with $x_0 = 1.6$.
These iterations all converge quadratically, that is each step roughly doubles the number of correct digits. The time needed to compute n digits of the golden ratio is proportional to the time needed to divide two n-digit numbers.

An easily programmed alternative using only integer arithmetic is to calculate two large consecutive

Fibonacci numbers and divide them. The ratio of Fibonacci numbers $F25001$ and $F25000$, each over 5000 digits, yields over 10000 significant digits of the golden ratio.

Using Newton's method, we can also calculate sqrt of numbers which are in regular use, such as $\sqrt{2}$, $\sqrt{3}$ or $\sqrt{5}$. This may be helpful to the users.

3.4 Adding more mathematical functions

Inverse Trigonometric Functions: We can find the asin and atan functions using the Taylor series. Rest four functions can be found using the following identities -

$$\begin{aligned} acos(x) &= \frac{\pi}{2} - asin(x) \\ asec(x) &= acos\left(\frac{1}{x}\right) \\ acsc(x) &= asin\left(\frac{1}{x}\right) \\ acot(x) &= \frac{\pi}{2} - atan(x) \end{aligned}$$

We will also implement the atan2 function as follows -

$$atan2(y,x) = \begin{cases} atan(\frac{y}{x}) & \text{if } x > 0 \\ \frac{\pi}{2} - atan(\frac{x}{y}) & \text{if } y > 0 \\ -\frac{\pi}{2} - atan(\frac{x}{y}) & \text{if } y < 0 \\ atan(\frac{y}{x}) \pm \pi & \text{if } x < 0 \\ \text{undefined} & \text{if } x = 0 \text{ and } y = 0 \end{cases}$$

For defining the OPERATION of the inverse trigonometric functions, we note that asin and atan are continuous increasing functions, while acos is a continuous decreasing function. So we can simply do like upper_bound = $acos$(lower_bound(x)). Note that we will have to check that the lower and upper bounds are in the respective domains of the function.

acot is a decreasing but non-continuous function. So we will be iterating for more precision until $acot$(lower_bound) > 0 or $acot$(upper_bound) < 0 as acot function is not defined at exact 0. Similarly we need to check for acsc and asec functions as they are non-continuous. For atan2 function, we need to iterate till either of lhs or rhs operand is not equal to exact 0.

Also we can add user-defined literals for angles, so that angle in trigonometric functions can be provided in both radians and degrees.

Hyperbolic Functions: We can compute the Hyperbolic functions using the Taylor series, but except for the computation of sinh and cosh, other series also require computation of Bernoulli (B_n) and Euler (E_n) numbers.

So rather than Taylor series, we can use the following identities -

$cosh(x) = \frac{e^x+e^{-x}}{2}$, $sinh(x) = \frac{e^x-e^{-x}}{2}$, $tanh(x) = \frac{e^x-e^{-x}}{e^x+e^{-x}}$
$sech(x) = \frac{1}{cosh(x)}$, $csch(x) = \frac{1}{sinh(x)}$, $coth(x) = \frac{1}{tanh(x)}$

For defining OPERATION of hyperbolic functions, we see that sinh and tanh are increasing and continuous functions. So it is simple to define the OPERATION for them. Whereas cosh decreases if $x < 0$ and increases if $x > 0$. So if lower(x) < 0 and upper(x) > 0, then $\text{lower}(\cosh(\text{x})) = 1$ and $\text{upper}(\cosh(\text{x})) = max(\cosh(\text{lower}(\text{x})), \cosh(\text{upper}(\text{x})))$.

Similarly for other three hyperbolic functions, we need to take care as they are either discontinuous or non-monotonic functions.

cbrt Function: As the power function is already implemented in library, it is easy to implement the cube root function. We can also add sqrt and cbrt to OPERATION. For OPERATION of cbrt, we need to take care as input can be negative.

$log_a b$ Function: In addition to the original log function in the library, we can also allow the function to be two input (using template metaprogramming), which calculates $log_a b$. We can do so by using the identity $log_a b = \frac{log_e b}{log_e a}$.

3.5 Library comparison and benchmarking with other similar libraries

RealLib: The RealLib package performs exact real number calculations. It provides two levels of access to real numbers. One of the levels operates on real numbers as complete entities but has poor performance when a multitude of simple operations is to be performed. For the latter case, their design provides an interface which operates on the level of approximations and is free of performance issues, but the objects operated on do not represent complete reals.

It uses some kind of type-2 approach using the TTE model of Weihrauch. The type-2 implementation of a binary operation computes an approximation based on approximations to its operands at higher precision, which is different from our implementation as we compute the operands at same precision. It does switch to dags to represent expression trees and does caching of approximations.

I do believe that to be sure about the accuracy of its result, we require higher accuracy from its arguments, as in RealLib. Though this is difficult to handle as the accuracy needed will grow by at least one bit in every iteration. For this RealLib, instead of a top-down approach, uses a bottom-up approach where the accuracy of the siblings determines the accuracy in a node, an approach which does not suffer from the above mentioned problem. I think it would be better if Boost.Real applies such an approach, for better precision of the result.

The bottom-up approach evaluates everything at a constant precision starting from the leaves of the tree, keeping track of the errors that are introduced at every step. If the end result of the computation on a tree is not accurate enough, the whole computation is restarted at higher precision until the process leads to a result which is accurate enough. The reiterations though don't add much to the complexity of the process as the time taken by the evaluation is dominated by the last iteration.

RealLib does not store the history of a computation at every step. The operations are being executed with the order and locality that the programmer and compiler give. The library also tries to optimize the order of evaluation of the siblings of nodes with multiple arguments. These are some of the features which Boost.Real can try to incorporate.

iRRAM: The iRRAM is a very efficient C++ package for error-free real arithmetic based on the concept of a Real-RAM. Its capabilities range from ordinary arithmetic over trigonometric functions to linear algebra even with sparse matrices. iRRAM has had the reputation of the fastest exact real number library, using highly optimized GMP (Gnu Multiple Precision arithmetic library) for the higher precisions. In iRRAM too, the whole computation is repeated with a significantly better precision bound.

Programs written for the iRRAM operate only on the level of approximations, using a bottom-up evaluation scheme and modularity on the level of approximations. As such, they only simulate operations on complete objects, and at no point in time do they have access to the functions that define real numbers.

<u>XRC:</u> It is an implementation of lazy exact real arithmetic in C. It represents a real $\hat{x} \in \mathbb{R}$ by a function $x : \mathbb{Z}^+ \to \mathbb{Z}$ satisfying

$$|B^n \hat{x} - x(n)| < 1 \; \forall n \in \mathbb{Z}^+$$

Larger values of B cause information to grow at a greater rate as n increases. An important advantage of this representation is that less precise values may be efficiently computed from more precise ones by simple bit-shifting (if B is a power of two). This makes caching (memoizing) of the most precise known value of all intermediate quantities worthwhile; some requests are then satisfied from the cache. Comparison is also easy: if we can find a value of n such that $x(n)$ and $y(n)$ differ by more than one, then x and y must be unequal.

A rational $\hat{q}$ is correctly represented by $q(n) = \lfloor B^n \hat{q} \rfloor$. This allows us to initialize exact reals from exact integer or rational data. It builds a dependency graph (DAG) in which each node contains pointers to its arguments, a description of the operation, and a cache for the largest argument with which the function has been called and the corresponding return value.

Now let us have a **performance comparison** between the libraries -
We will use the logistic sequence example. We will compute the iteration $x_{i+1} = 3.75(1 - x_i)x_i$ with $x_0 = 0.5$.

```
real s("0.5");
real coeff("15/4", type::RATIONAL);
real one("1", type::INTEGER);
for (int i=1;i<=len;++i)
{
    s = coeff * (one - s) * s;
}
cout << s <<endl;
```

This is a scenario of heavily nested computations and our recursive evaluation mechanism failed for more than several hundred nested operations. Testing on my system (using a Intel Core i7 1.8GHz), I could compute for 20 iterations in 0.77s, but couldn't compute for more iterations. Timing on the other three exact real number systems, we get the following table -

iterations	RealLib	iRRAM	XRC
100	1ms	625ms	383ms
1000	150ms	12ms	143s
10000	48s	5.5s	-

Note that these times are recorded for calculating the first 6 digits of logistic sequence, whereas Boost.Real found around first 200 digits. Even though we can't compare directly, we can conclude that Boost.Real is relatively slow compared to the other real libraries.

Consider another example, where we ask for arbitrarily precise approximations to the value π -

```
real pi = boost::real::irrational::PI<int>;
pi.set_maximum_precision(40);
cout << pi << endl;
```

Boost.Real computed π to a 360 digit approximation in 46.0s, which is way more than the other three real libraries -

iterations	RealLib	iRRAM	XRC
10000	730ms	230ms	364s

This also calls for the need of some improvement in our approach to calculation of π, as we are doing irredundant calculations instead of caching the previous results.

3.6 Improvement in Newton Raphson Method

In the library, we have implemented binary search division for finding $\frac{48}{17}$ and $\frac{32}{17}$, which is used for the initial guess. Rather, we can pre-compute these fractions to some precision (maybe 10) and use them directly in the Newton Raphson algorithm, as they are used for the initial guess only.

4 Proposed Milestones and Schedule

Before June 7
I will familiarize myself with the project and understand the code base. Learn any programming concepts required with the help of my mentor. Do the appropriate changes as mentioned in Competency Task 1 and Task 2 (see below).

June 7 - June 13
Will work on conversion of expression tree to DAG.

June 14 - June 20
Continue working on the conversion to DAG.

June 21 - June 27
Will work on caching of approximations to a node in DAG.

June 28 - July 4
Will work on various arithmetic optimizations on the DAG.

July 5 - July 11
Testing and fixing bugs for the newly constructed DAG.

July 12 - July 18
First evaluation of the project. Will work on Champernowne Irrational improvement, adding e and ϕ to irrationals and improving Newton Raphson method.

July 19 - July 25
Designing tests for newly added irrationals. Adding them to the documentation.

July 26 - August 1
Will start on adding inverse trigonometric and hyperbolic functions.

August 2 - August 8
Adding user defined literals for trigonometric functions. Adding more mathematical functions and defining OPERATIONS for them. Designing tests for these new functions.

August 9 - August 15
Fixing bugs and errors. Benchmarking and documentating the library and the work done in the project. Making the work ready for final evaluation of the project.

5 Programming Competency

All files of the competency test are uploaded in the repository: *Boost.Real-competency-tasks*.

1. Find and implement an optimal representation to store and work on numbers involving a large number of digits.

   It is implemented in the file *task1_add_subtract.cpp*, *task1_multiply.cpp*.

   I took most of the code from the Boost.Real library and made adjustments to it. I did not implement the base conversion portion as it would be exactly same as of our library. As in the library, I have used vector to store digits. Firstly, I replaced the insert function in add and subtract vectors by push_back function, as inserting at the beginning of vector will take linear time whereas push_back will take constant time. As we are only inserting in the beginning, so all we need to do is to reverse the vector with takes linear time after pushing back the digits. So overall the time complexity reduces from quadratic time to linear time.

   Similar issue is there in the normalize function, where we erase from the front, leading to more time complexity. Here also we can first reverse the vector and then pop_back, leading to a linear time complexity. Another change I did was to write base $= \lfloor \frac{\text{base}}{2} \rfloor + \lfloor \frac{\text{base}-1}{2} \rfloor + 1$. This is the correct way to handle for both even and odd bases in carryover case in add_vector function.

   We can even prevent overflow in mul_mod and mult_div functions like we did in the add_vector function. For example, consider the operation $res = (res + a)\%\text{mod}$. Here the sum $(res + a)$ might overflow, and we can use the method in add_vector to avoid this. These are the modified mul_mod and mult_div functions -

   ```
   //Returns (a*b)%mod
   template <typename T = int>
   static T mul_mod(T a, T b, T mod)
   {
       T res = 0; // Initialize result
       a = a % mod;
       while (b > 0)
       {
           // If b is odd, add 'a' to result
           if (b % 2 == 1) {
   ```

```
                res = res % mod;
                a = a % mod;
                if(res < (mod - a)) {
                    res = res + a;
                }
                else {
                    T min = std::min(res, a);
                    T max = std::max(res, a);
                    if (min <= (mod - 1)/2) {
                        T remaining = (mod - 1)/2 - min;
                        res = (max - mod/2) - remaining - 1;
                    } else {
                        res = (min - (mod - 1)/2) + (max - mod/2) - 1;
                    }
                }
            }

            // Multiply 'a' with 2
            a = a % mod;
            if(a < mod/2) {
                a = a * 2;
            }
            else {
                a = (a - mod/2) + (a - (mod - 1)/2) - 1;
            }

            // Divide b by 2
            b /= 2;
        }

        return res % mod;
    }

    //Returns (a*b)/mod
    template <typename T = int>
    static T mult_div(T a, T b, T c) {
        T rem = 0;
        T res = (a / c) * b;
        a = a % c;
        // invariant: a_orig * b_orig = (res * c + rem) + a * b
        // a < c, rem < c.
        while (b != 0) {
            if (b & 1) {
                if(rem < (c - a)) {
                    rem += a;
                }
                else {
                    T min = std::min(rem, a);
                    T max = std::max(rem, a);
                    if (min <= (c - 1)/2) {
```

```
                    T remaining = (c - 1)/2 - min;
                    rem = (max - c/2) - remaining - 1;
                } else {
                    rem = (min - (c - 1)/2) + (max - c/2) - 1;
                }
                res++;
            }
        }
        b /= 2;

        if(a < c/2) {
            a *= 2;
        }
        else {
            a = (a - c/2) + (a - (c - 1)/2) - 1;
            res += b;
        }
    }
    return res;
}
```

I also got the Karatsuba multiplication to work with INT_MAX base. We have equations of the form $z_2 = x_1y_1$, $z_1 = x_1y_0 + x_0y_1$, $z_0 = x_0y_0$. Now rather than writing $z_1 = (x_1 + x_0)(y_1 + y_0) - z_2 - z_0$, we can rewrite this is as $z_1 = (x_0 - x_1)(y_1 - y_0) + z_2 + z_0$. Doing this prevents overflow, and we also have to take care of the sign of the term $(x_0 - x_1)(y_1 - y_0)$ while using subtract_vector. Karatsuba did work faster than the standard multiplication, for in a 1000 by 1000 digit multiplication, Karatsuba ran in around 28 ms whereas standard multiplication ran in around 64 ms.

2. Explain how could you provide a "Small String Optimization"-like approach for Real numbers representation at compile time.

   Exact number in the library is being stored using vector.

   ```
   vector<Type> vect;
   ```

   will allocate the vector i.e. the header info on the stack, but the elements on the free store (heap). Operations on automatic variables (from the stack) are generally much faster than those involving the free store (the heap).

   Boost has a small_vector class, which is a vector-like container optimized for the case when it contains few elements. It contains some pre-allocated elements in-place, which allows it to avoid the use of dynamic storage allocation when the actual number of elements is below that pre-allocated threshold. small_vector is inspired by LLVM's SmallVector container. small_vector's capacity can grow beyond the initial pre-allocated capacity.

   Boost also has a static_vector class, which is an hybrid between vector and array: like vector, it's a sequence container with contiguous storage that can change in size, along with the static allocation, low overhead, and fixed capacity of array. The advantage in small_vector is that it will use the pre-allocated memory from the stack for its first N elements and only when the container needs to grow further, it will create a new storage block using an heap allocation.

An example representation of small_vector is given in the file *task2.cpp*. We can use the small vector in our library also, as many a times the number of digits in the number we use will be small (due to a very large base), and so the size of vector will be very small.

3. Provide a template function "get_vector", similar to std::get, that given the type X as a parameter, returns editable access to the vector of type X in a tuple of vectors.

   It is implemented in the file *task3.cpp*.

4. Provide a function receiving an integer (X) and a tuple (std::tuple⟨type1, type2, type3,...⟩) into a tuple of vectors (std::tuple⟨std::vector,std::vector, std::vector,...⟩ where each vector has X elements of each originally received in each tuple_element. E.g. for X=2 and the tuple {1, 1.0, 'a'} , the result type is std::tuple⟨std::vector, std::vector, std::vector⟩ and the values are: {{1, 1},{1.0, 1.0},{'a', 'a'}}.

   It is implemented in the file *task4.cpp*.

5. Define a user-defined literal assignable to float that fails compilation, when the value provided, cannot be expressed as a positive integer power of 0.5 (e.g. 0.5, 0.25, 0.125).

   It is implemented in the file *task5.cpp*. I have passed the values for testing as a constexpr, for otherwise giving value in runtime can't give compilation error.

6 Previous Work done and PRs

I have made around 5 and 11 pull requests in Boost.Real and Boost.Astronomy repositories respectively, some of which were merged and some are still to be reviewed.
Some of the important PRs in Boost.Real -

- Improved coding practises, like using *this* pointer, correct data type for precision (precision_t).
- Made some corrections in trigonometric functions, like correcting lower and upper bounds of the functions.
- Corrected the carry-over case in add_vector function.

Some of the important PRs in Boost.Astronomy -

- Implemented Julian date function and corrected time tests.
- Corrected the expression of positional angle.
- Implemented arithmetic functions like difference of two representations, multiplication of representation with a constant and matrix.

Here are the links to all my PRs in Boost - Real, Astronomy.