Google Summer of Code 2020 Proposal

Project: Bringing Boost.Real to review-ready state
Organization: Boost C++
Mentors: Damian Vicino, Laouen Belloli

PERSONAL DETAILS:
Name: Vikram Singh Chundawat
Gender: Male
Age: 20 years
Time Zone: Indian Standard Time (GMT +05:30)
University: BITS PILANI, PILANI CAMPUS, RAJASTHAN, INDIA
Email: vikram2000b@gmail.com, **f20180128@pilani.bits-pilani.ac.in**
M. No.: +918003213447
Github Username: vikram2000b

Bio: I am a computer programming enthusiast currently pursuing engineering at BITS Pilani. I am well adept in c, c++ and python with a good knowledge of data structure and algorithms. I have more than 3 years of programming experience in which I have gained considerable experience in object-oriented programming, template metaprogramming and c++ STL.

Availability: I will be able to spend nearly 40-50 hours per week from 18 march till the final submission. I will have my final semester exams from 1 May to 15 May, so I will be less active in that period. I can start working from 18 May and would be able to work till August.

BOOST.REAL

About the project: The basic idea of the project was to deal with the rounding error generated in complex mathematical calculations using floating-point numbers. In c++, the numbers have to be rounded off to fit them in 'float' or 'double' or whatever data type we are using, we have limited memory to store the numbers. And in calculations using these numbers, the final error accumulated by rounding off these numbers can be too much, and also we cannot track back the major source of error. Boost.Real project tries to overcome these problems by representing numbers in ranges and the accuracy of those ranges can be increased with every iteration. The basic data structures and mathematical operations for these numbers were implemented in GsoC'18 by Lao Belloli, a detailed summary of initial work on this project is given on this blog. After that, in GsoC'19, a lot of improvements were done on this project by Kimberly Swanson (@universenox) and Sagnik Dey (@SagnikDey92) and the summary of these projects are work by Sagnik and work by Kimberly.

A lot of work was completed in these two years and a lot of problems and bugs were solved but still, there were some problems:

1. Problems of overflow in implementations of base INT_MAX and INT_MAX/2. We store the digits in a vector of integers and to use all the space available in a vector of integers, we need to store numbers in base INT_MAX, but serious problems of integer overflow occur in summation and multiplication operations.
   The base of INT_MAX was not implemented and INT_MAX/2 was implemented but still there were some problems of memory overflow and 1 bit of memory was not used for every integer in our vector, as the base was INT_MAX/2.
2. The computations are very slow in the new base (INT_MAX or INT_MAX/2).
3. A lot of redundant calculations and space is used in case of repeated calculations. In case of simple calculations like
   ```
   boost::real::real a("2");
   for(int i=0;i<100;i++)
   {
           a+=a;
   }
   ```
   A lot of extra space and unnecessary calculations are done due to tree structure of operations.
4. The constants like Pi were added in a hardcoded way due to the absence of power functions. The Pi can be calculated using the Chudnovsky Algorithm but that requires an efficient algorithm to calculate the power of real numbers.
5. No method to calculate the powers of real numbers.
6. There are no methods for mathematical functions like square root, exponent, logarithm, sin, cos, etc.

Ideas and Solutions:

A New Base (The base of 2^32):

The main problems behind implementations of new base were memory overflow and slow calculations. These were because earlier the method of multiplication was like this:

```
int sum = lhs[i]*rhs[j] + result[i_n1 – i_n2] + carry;
carry = sum/10;
result[i_n1 – i_n2] = sum%10;
i_n2++;
```

and for summation it was like:

```
 int digit = carry + lhs_digit + rhs_digit;
if (digit > 9) {
carry = 1;
digit -= 10;
```

```
} else {
carry = 0;
}
```

and numbers were stored in the base of 10, which limited our use of memory per integer. Thus a lot of memory was wasted per integer in our vector<int>. To solve this problem, the base of INT_MAX was tried, but that had a lot of problems with memory overflow. So, to avoid memory overflow, the base of INT_MAX/2 was implemented. The calculations in the new base were very slow, and calculations of base conversion were also slow. And in the base of INT_MAX/2, we were wasting 1 bit per integer in our vector<int>. The basic calculations were slow because a new set of methods to get (a*b)%mod and (a/b)/mod were needed to avoid memory overflow. Here is their implementation:

```
//Returns (a*b)%mod
T mulmod(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+a)%mod;
                //Multiply ‘a’ with 2
                a = (a*2)%mod;
                // Divide b by 2
                b /=2;
                }
        return res % mod;
}



//Returns (a*b)/mod
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) {
                        rem += a;
                        if( rem >= c ) {
                                rem -= c;
                                res++;
                        }
                }
                b /= 2;
                a *= 2;
                if (a >= c) {
                        a -= c;
                        res += b;
                }
        }
        return res;
}
```

These implementations made calculations in a new base very slow, and still after that we were not able to use all memory of integers.

Here, I propose to use the base of 2^32 instead of INT_MAX or INT_MAX/2. We have 32 bits in an integer, and I am planning to use all of them to store our number. I have completed the implementation of this base for string digits in the competency test. I am using bigger integer types like ‘long long’ to store numbers, where there is a chance of overflow. Here, I am not changing our system from vector<int> to vector<long long>. I am just changing our methods a little bit(It will be more clear when you will see the code below or code of my competency test). I am also planning to change our system from vector<int> to vector<unsigned int>, to avoid negative values when we use all 32 bits.

So, the new algorithms for addition and multiplication for base of 2^32 will be like,

```
// summation
long long val1,val2,sum,carry=0;
long long base = (1<<30);
base = (base<<2); // now our base if of 2^32
vector<unsigned int> res_vec(n,0);
for(int i=0;i<n;i++)
{
        sum = 0;
        sum+=carry;
        if(i<n1) val1 = vec1[i];
        if(i<n2) val2 = vec2[i];
        sum += val1+val2;
        carry = sum/base;
        res_vec[i] = sum%base;
        val1 = val2 = 0;
}
if(carry!=0) res_vec.push_back(carry);
return res_vec;
```

The complete method of summation and method of multiplication can be seen in file “digit.hpp” in files of competency tasks. The above code was just added to clear my

purpose of using ‘long long’. We can modify our system to use the base of 2^(sizeof(int)*8), for machine dependent sizes of integers.
Here are some benefits and drawbacks to this implementation.
Benefits:

- We will be able to use the new base without any problems of overflow.
- No waste of memory in using vector<unsigned int> to store digits.
- Faster calculations than the previous method(used in the base of INT_MAX/2) because there will no slower methods of multiplication (like there were for (a*b)%mod and (a*b)/mod).
- The calculations of base conversions will be faster, because it will only be an extension of binary conversion. We will do the binary conversion and put 32 digits of our binary number into one integer.

Drawbacks:

- For systems with optimized performance on 'long long' integer types, we won't be able to continue with vector<long long> and we will have to work with vector<int> or vector<long>. Thus we will not be able to take advantage of that optimization. And if we try to work with vector<long long>, we will either have to waste 1 bit or make calculations using packing-unpacking (using two 'long long' for calculations) and for multiplication. And that will also lead us to slower calculations.
- To store digits in vector<T>, 'T' should always be a smaller integer type than ‘long long’.

For systems with optimized performance on 'long long' integer types, if we use vector<'long long'> then we will have to use slower methods for summation and multiplication and we may need to waste 1 bit of memory per 'long long' and the slower methods for basic operations are more likely dominate the benefits of optimizations on 'long long'. So it will be better to use vector<unsigned int> to store digits. Because that will work faster and no loss of memory will be there.

A new method to store integers and rational numbers:
Here, I am proposing to add a new store 'integers' and 'rational numbers', because for integers and rational numbers there is no need to store them in a range. Storing them in range and iterating again and again for more accuracy is unnecessary. We also don't need an operation tree for operations between integers and rational numbers because their values will always remain the same no matter how many times we iterate for a more precise result. So, after having a different structure for integers and rational numbers we can make our calculations a bit more space-efficient and faster. e.g. we don't need to convert multiplication of two integers into an operation c = a*b where both a and b are integers so, the c will also be an integer, and instead of making it an 'operation' and storing copies of a and b in LHS and RHS pointers. We will just calculate the result of a*b and put it in values of c. The same will happen for multiplication. The same will happen for operation between two rational numbers and operation between one integer and one rational number. For the implementation of these numbers, we will not change the structure of our previous implementations of numbers. We will add two more types in KIND namespace. For integers we will store our values in upper_bound. For rational

numbers we will get/convert them in an 'a/b' form and will store 'a' in upper_bound and 'b' in lower bound. All operations will be modified accordingly to deal with these numbers.

Integer Power of real number numbers:
Here I am also proposing to implement as a new method to calculate integer powers of real numbers. This operation will have the LHS boundary as a real number and RHS boundary as a real_integer. It helps us in avoiding unnecessary recursion in case of multiplication of a number with itself. e.g. a = a*a*a, or for(int i=0; i<n;i++) a*=a; This method will not have any recursive method of multiplication, it will have repeated multiplication iteratively, thus reducing extra space used for making operation trees and avoiding redundant calculations. The power will be calculated in the same manner as they are calculated for normal floating-point numbers. Instead of direct multiplication, we will take into account the signs of ranges. We are having real^integer because it will help us in making a method for real^real.

Factorial of integers:
The method to calculate factorial of positive integers can also be implemented for real_intergers. It will also be done iteratively. The main purpose of having this is to calculate Taylor expansions.

Use of Taylor Expansions:
After having a successful implementation of the above points, a lot of other things can be added by using these methods. Like optimally calculating Taylor expansions, which will help us in making algorithms to logarithms, exponentials, trigonometric functions, and inverse trigonometric functions.

Real^Real:
After having implementations for all previous points. We can make algorithms to get real^real and square root of real numbers or fractional powers of real numbers.

result = a^b = exp(b* ln(a)) //ln stands for log of base e

For having the above implementations, we need to have implementations for exponential and logarithmic functions. **For that we will need Taylor expansions, thus a method for integer powers and factorials will be needed.**

Pi using Chudnovsky Algorithm:
Now having these implementations, Pi can be calculated using the Chudnovsky algorithm.

Scientific Constants:
Some constants for the scientific calculations can be added like the mass of an electron, the mass of a proton, the charge of an electron, Planck's constant, etc.

Some methods of optimization:
There are redundant calculations in operations like

```
		for(int i=0;i<100;i++)
		{
			a+=a;
		}
```
and a lot of extra space is used.
We can tackle this problem just by changing += a little bit.
Here the += operation will keep on making a tree of operations and that will create a very big tree and will do a lot of unnecessary calculations. We can add one extra condition that will change or modify the type of operation when the same number is added to itself.

```
real& operator+=(real& other){

	//if both pointers point to same number
	if(this == (&other)  &&  this->_kind==KIND::EXPLICIT)
	{
		this->_lhs_ptr = new real(*this);
		this->_lhs_ptr = new real("2.0");
		this->_lhs_ptr->_kind = KIND::INTEGER;
		this->_operation = OPERATION::MULTIPLY;
		return this;
	}
	if(this == (&other)  &&  this->_kind==KIND::MULTIPLY)
	{
		if(this->_lhs_ptr->_kind == KIND::INTEGER)
		{
			*(this->_lhs_ptr) *= 2;
			return this;
		}
		else if(this->_rhs_ptr->_kind == KIND::INTEGER)
		{
			*(this->_rhs_ptr) *= 2;
			return this;
		}
		// above values which are multiplied by 2 are integers
		// so there will be no extra tree or recursion will be created
		this->_lhs_ptr = new real(*this);
		this->_lhs_ptr = new real();
		this->_lhs_ptr->_kind = KIND::INTEGER;
		*(this->_lhs_ptr) = 2;
		this->_operation = OPERATION::MULTIPLY;
		return this;
	}
......}
```
same type of method will be used when we use *=, but instead of converting it to real_multiply, we will convert it to real_integer_power. These two changes can also

be applied to *,+ operator to make algorithms faster. For /= and -= we will simply return 1.0 and 0.0 without making it an operation and doing any calculations. These optimizations are only for very specific cases. I am trying to find more general optimizations like this with the help of my mentor.

Proposal Timeline:

Before April 27:

- To familiarize myself with the project and understand the codebase.
- To learn any additional mathematical and programming concepts required with the help of the mentor.

April 27 – May 18

- I will be having my end semester examinations from March 1 – March 15, so I will be less active in this period.
- During this period I will understand the working of the community and familiarize myself with the community.

May 18 – May 25

- I will work to change the base and modify all the operations and test cases to work with a base of 2^(sizeof(int)*8).

May 25 – June 10

- Add integer and rational real numbers.
- Modify all operations to work efficiently for integers and rational numbers.
- Design test cases for integers and real numbers.
- Modify the documentation for users to work with these types.

June 10 – June 15

- Fixing bugs and making the work ready for evaluation.

June 15 Evaluation.

June 15 – June 20

- Implement a method to calculate factorial for integer type real numbers.
- Add it to documentation.

June 20 – June 25

- Implement the method to calculate the integer power of real numbers.
- Design all the test cases and modify the documentation.

June 25- July 6

- Modify the Pi from hard coded digits to the Chudnovsky algorithm.
- Adding user-defined literals for angles, so angle in trigonometric functions can be given in both "radians" and "degrees"

July 6 – July 13

- Fixing bugs and making the work ready for evaluation.

July 13 Evaluation.

July 13- July 30

- Add algorithms for exponential, logarithms and trigonometric functions using Taylor expansions.
- Make an algorithm for calculation of real^real
- Merge it with the previous method to calculate power, and make it run for all possibilities so that the power of (any)^(any) can be calculated. Here "any" means our number can be integer, rational, explicit, operations and algorithm.

- Build test cases for this algorithm.
- Document all the changes made.

July 30 – August 10

- Add some scientific constants to the library. (charge of an electron, the mass of electron, proton and neutron, planck's constant and mass of some other subatomic particles.)
- Further, refine tests and documentation for the whole project.
- Fixing bugs and errors.
- Making the work ready for final submission.

Final submission

I have considered nearly one week of buffer before each evaluation to fix bugs and deal with any unpredicted delays. I will try to add inverse trigonometric functions in case of earlier completion of scheduled works. I will discuss possible improvements in algorithms and structures of the library to solve the problem of redundant calculations and space used in repetitive calculations. The idea suggested above to avoid redundant calculation is only applicable to one specific case. I will try to find more generic optimizations and will try to implement them after the final submission.

Competency Test

All files of the competency test are uploaded in this repository:
https://github.com/vikram2000b/GSoC_tasks

To find an optimal representation for digits.
I have stored digits in vector<unsigned int>, and in a base of 2^(sizeof(int)*8) to avoid any waste of memory. This implementation is more efficient and faster than the previous ones that were used in this project.
I have added methods of base conversion in files ‘digit_helper.hpp’. All methods of storing and operations on digits are written in ‘digit.hpp’. The use of these methods is demonstrated in ‘digit.cpp’.

I was given three competency tasks by mentor Damian Vicino.

Task 1: 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).

There are two possible implementations of this task, one using float literal and other using string literal. The implementation with float literal can have problems of rounding error. I have implemented both of them. The implementation using float literal is written in ‘task1_using_float_literal.cpp’ and the other is written in ‘task1_using_string_literal.cpp’.

Task 2: 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’}}.

The methods of this implementation are written in ‘task2.hpp’ and they are used in ‘task2.cpp’.

Task3: 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.

The methods of this implementation are written in ‘task3.hpp’ and they are used in ‘task3.cpp’.