RM-pkg
Reusable Mathematics Package

Akiel Aries

November 2022

Abstract

RM-pkg is a reusable mathematics library written in C++ originally inspired from undergraduate coursework at Northern Arizona University in mathematics, cybersecurity and computer science, Python based projects in CS 499 Contemporary Developments, Deep Learning, developments on vpaSTRM as well as my increasing curiosity of the all things pure and applied in the realm of mathematics
The goal is to make a reusable mathematics library similar to the use of math.h allowing users to call within their own projects. Starting with some implementations of basics of different mathematical topics like arithmetic, calculus, statistics, linear algebra, etc. in conjunction with more advanced algorithms seen in the blend of such topics, branches of Machine Learning, image processing and much more. Some of the modules operate in a circular dependant way, for example arithmetic operations seen in linear algebra vector operations that can be seen in algorithms implemented into the deep learning module. Many of these implementations were first prototypes in Wolfram Mathetmatica, then converted to C++ code for the package. Look in the drivers folder for examples on using these tools in your own project. This paper will highlight the modules in use, the mathematics/logic behind them, future developments, and more.

github repository: `https://github.com/akielaries/RM-pkg`

Contents

1 Tools and Dependencies

The goal of RM-pkg is to have as little dependencies as possible without re-inventing the wheel too much while performing speedy computations. Other than C and C++ standard libraries the 3rd-party dependencies that are used are deemed necessary for many of the packages functionalities.

1.1 Tools

- **gtest**: Used for unit testing. Within the projects development Makefile, there are option to run the tests for the modules in this package. This is done by compiling a driver file that runs the methods referenced in each module, the source module file itself, and then the gtest based test driver file for said module. Some modules are harder to test than others but for the most part simply checking the expected output against the computed output does all the checking that is needed.

- **AFL and AFL++**: AFL is a fuzzing tool built by Google but now deprecated. AFL++ is an open source community effort forked from AFL offering the same functionalities but much better. Advertised as better mutations, more speed, and better instrumentation. Fuzzing comes in handy for generating random inputs for your program and analyzing the outputs. Analyzing crashes, unique or not, proves valuable for implementing strong and secure code.

  The code below is an example on how testing on the arithmetic package was done.

```
#include <RM-pkg/arithmetic/arith.hpp>
#include <gtest/gtest.h>

namespace {
    arith ar;
    // test case, test name
    TEST(arith_test, add_positive) {
        EXPECT_EQ(46094, ar.rm_add(93, 106, 3551, 42344));
        EXPECT_EQ(6.85, ar.rm_add(1.25, 1.85, 2.75, 1));
    }
    // multiplication (product) testing
    TEST(arith_test, mult_positive) {
        EXPECT_EQ(240, ar.rm_mult(10, 8, 3));
        EXPECT_EQ(6.359375, ar.rm_mult(1.25, 1.85, 2.75, 1));
    }
    // subtraction
    TEST(arith_test, sub_positive) {
        EXPECT_EQ(5, ar.rm_sub(10, 8, 3));
        EXPECT_EQ(1, ar.rm_sub(1.25, 1.85, 2.75));
    }
}
```

1.2 Dependencies

RM-pkg makes use of a few open source cross-platform compatible packages and libraries in the cases of threading for performance, graphics libraries, and packages for testing and fuzzing.

- **openMP**: Open Multi-Processing is useful for simple threading when needed. For loops that don't make complex calls can be encosed in a `#pragma omp parallel for {}` declaration.
- **openCL**: Open Computing Language is another useful threading API that allows for more customized parallelization techniques.
- **openGL**: Open Graphics Library used for hardware-accelerated rendering. API makes use of interaction with a machines Graphics Processing Unit (GPU) allowing for quick rendering of modules that make use.
- **libXbgi**: Borland Graphics Interface reiteration. Provides a useful graphics API to get started with visualizing mathematical algorithms.

2 Modules

The purpose of this package is to also be as modular as possible. Simply calling a module in the package from within your C++ project with `#include <RM-pkg/arithmetic/arith.hpp>`, this allows a developer to utilize the basic operations implemented in the arithmetic module. Calling `#include <RM-pkg/number_theory/primes.hpp>` allows a developer to use the prime number based operations within the number theory module. As of now (Nov. 2022), there is no formal installation process or real way to use RM-pkg. The use of Makefiles + shell scripts is primarily for development purposes, users of the package will need to compile their own version of the library in their standard library directories or include the source and header files in their projects with some modifications. Perhaps in later implementations of RM-pkg, APIs for Python, Julia and possibly MATLAB depending on how easy the binding process is.

2.1 Arithmetic

Basic arithmetic operations, addition, subtraction, multiplication, exponentiation, were implemented in a way to accept 'n' arguments of 'n' types. Meaning calling any of the respective functions allows numerous parameters to be passed in of various numeric data type. This acts as the basis for many of the basic functionalities of this package. In the repository for the project, see `/include/arithmetic/arith.hpp` for the implementation of this module. Since it makes use of C++ class templates, this module is header-only in development. Addition, subtraction, and multiplication were all implemented in an identical way, making use of templates and inline functionalites.

Within the RM-pkg repository, I had provided examples on how to use the modules as seen in the `/drivers` folder. Each driver file contains a `main()` function.

2.1.1 Addition

The arithmetic operation addition can sometimes be thought of the adding of two digits while a summation is the adding of a series of two or more digits. RM-pkg's add() function makes use of both. Since it is a template class allowing to add n numbers of many numeric data types, passing in just two numbers to add() qualifies as plain addition while passing in two or more variables qualifies as more simplified version of a summation.

Below is how the addition operation was implemented.

```
1 template<typename T>
2 inline T add(T t) {
3     return t;
4 }
5 template<typename T, typename... Ts>
6 inline auto add(T t, Ts... ts) {
7     return t + add(ts...);
8 }
```

Addition:

$$z = x + y$$

Where **x** and **y** are the addends and **z** is the sum

A typical summation may look something like this:
Summation:

$$f(x) = \sum_{i=1}^{n} x_1 + x_2 + ... + x_n$$

Where **i** is the index of the summation and **n** is the upper limit. For example the summation of one with a limit of three would look like:

$$6 = 1 + 2 + 3$$

RM-pkg's add() function operates differently for multiple digits as it does not take in an upper limit to calculate up to from the index. Instead, it takes in multiple indices like a list and adds them together. See the formula below:

$$f(x) = \sum_{i=a} a_1 + a_2 + ... + a_n$$

Where **a** is a series of digits input into the call to the function and **n** is the index of the list-like series passed in. See below:

```
int sum = ar.add(4, 8, 1, 2);
```

Translating the function call into the above formula would look like:

$15 = 4 + 8 + 1 + 2$

Below is a brief example on how to use the multiplication function in the arithmetic module.

```
#include <iostream>
#include <stdio.h>
#include <vector>
#include <cassert>
#include <RM-pkg/arithmetic/arith.hpp>

int main() {
    // declare arithmetic class object
    arith ar;
    int a = 10;
    int b = 8;
    int c = 3;
    double d = 1.25;
    float e = 1.85;
    float f = 2.75;
    long g = 1.35;
    float r0 = ar.add(a, b, c);
    int r1 = ar.add(d, e, f, g);
    float r2 = ar.add(d, e, f, g);

    printf("%d + %d + %d = %f\n",
            a, b, c, r0);
    printf("%f + %f + %f + %ld = %d\n",
                d, e, f, g, r1);
    printf("%f + %f + %f + %ld = %f\n\n",
                d, e, f, g, r2);
    return 0;
}
```

2.1.2 Subtraction

This operation works exactly like `add()` except it is performing subtraction instead of addition.

$z = x - y$

Where **x** is the *minuend* **y** is the *subtrahend* making **z** the *difference*.

Below is a brief example on how to use the subtraction function in the arithmetic module.

```cpp
#include <iostream>
#include <stdio.h>
#include <vector>
#include <cassert>
#include <RM-pkg/arithmetic/arith.hpp>

int main() {
    // declare arithmetic class object
    arith ar;
    int a = 10;
    int b = 8;
    int c = 3;
    double d = 1.25;
    float e = 1.85;
    float f = 2.75;
    long g = 1.35;
    int r3 = ar.sub(a, b, c);
    int r4 = ar.sub(d, e, f ,g);
    float r5 = ar.sub(d ,e, f, g);

    printf("%d - %d - %d = %d\n", a, b, c, r3);
    printf("%f - %f - %f - %d = %d\n", d, e, f, g, r4);
    printf("%f - %f - %f - %f = %f\n\n", d, e, f, g, r5);
    return 0;
}
```

2.1.3 Multiplication

The arithmetic operation multiplication can be conceptualized the same as addition. A product $\prod$ is the multiplication of a series of two or more numbers. Just like summation there are typically an index and an upper limit. Identical to the addition operation of this module, multiplication behaves like a modified product formula. Below is how the multiplication operation was implemented.

```cpp
template<typename W>
inline W mult(W w) {
    return w;
}
template<typename W, typename... Wv>
inline auto mult(W w, Wv... wv) {
    return w * mult(wv...);
}
```

Multiplication:

$z = x \cdot y$

Where **x** is the *multiplier* **y** is the *multiplicand* making **z** the *product*.
A typical summation may look something like this:
Product:

$$f(x) = \prod_{i=1}^{n} x_1 \cdot x_2 \cdot ... \cdot x_n$$

Where **i** is the index of the product and **n** is the upper limit.
For example the product of one with a limit of three would look like:

$6 = 1 \cdot 2 \cdot 3$

Take this sample into account as well.

$$f(x) = \prod_{i=1}^{6} i^2$$

This results in:

$518,400 = 1 \cdot 4 \cdot 9 \cdot 16 \cdot 25 \cdot 36$

RM-pkg's mult() function operates differently for multiple digits as it does not take in an upper limit to calculate up to from the index. Instead, it takes in multiple indices like a list and multiplies them together. See the formula below:

$$f(x) = \prod_{i=a} a_1 \cdot a_2 \cdot ... \cdot a_n$$

Where **a** is a series of digits input into the call to the function and **n** is the index of the list-like series passed in. See below:

```
1 int sum = ar.mult(4, 8, 1, 2);
```

Translating the function call into the above formula would look like:

$64 = 4 \cdot 8 \cdot 1 \cdot 2$

Below is a brief example on how to use the multiplication function in the arithmetic module.

```
1 #include <iostream>
2 #include <stdio.h>
3 #include <vector>
4 #include <cassert>
5 #include <RM-pkg/arithmetic/arith.hpp>
```

```
6
7  int main() {
8      // declare arithmetic class object
9      arith ar;
10     int a = 10;
11     int b = 8;
12     int c = 3;
13     double d = 1.25;
14     float e = 1.85;
15     float f = 2.75;
16     long g = 1.35;
17     int r6 = ar.mult(a, b, c);
18     int r7 = ar.mult(d, e, f ,g);
19     double r8 = ar.mult(d ,e, f, g);
20
21     printf("%d * %d * %d = %d\n", a, b, c, r6);
22     printf("%f * %f * %f * %d = %d\n", d, e, f, g, r7);
23     printf("%f * %f * %f * %ld = %f\n\n", d, e, f, g, r8);
24     return 0;
25 }
```

2.1.4 Exponentiation

Exponentiation is also a basic arithmetic operation that also makes use of the multiplication operation. Multplying the base by itself continuously as specified by the power. STL provides the `pow()` function that is capable of performing similar operations, RM-pkg's `exp()` function operates in similar ways to the previously mentioned `add()`, `sub()` and `mult()` functions, taking in **n** arguments of **n** types. A basic exponentiation would include one base and one power, RM-pkg's exponentiation function allows for one base and **n** powers.

Exponentiation:

$$x^n = \underbrace{x \cdot x \cdot x \cdot ... \cdot x}_{\text{n times}}$$

Where **x** is the raised to the power of **n**.

Say we have an exponential equation like below:

$$x^{n^i}$$

Where **x** is our base, **n** is the power of our base, and **i** is the power of the result of our original exponential equation. Breaking it down as such:

$$x^{n^i} \Rightarrow \underbrace{x \cdot x \cdot ... \cdot x}_{\text{n times}} = y^i$$

$$y^i \Rightarrow \underbrace{y \cdot y \cdot ... \cdot y}_{\text{i times}} = z$$

Where **z** is our result.
Below is how the exponentiation operation was implemented.

```
template<typename Z>
inline W exp(Z z) {
    return z;
}
template<typename Z, typename... Zy>
inline auto exp(Z z, Zy... zy) {
    return z *= exp(zy...);
}
```

2.2 Calculus

Calculus is a branch of mathematics focused on continuous change. Split into two branches (see below), concerned with rates of change, slopes of a given curve, quanitity accumulations, areas between and below curves.

2.2.1 Differential

The derivative of a function of a real variable measures the sensitivity to change of the function value (output value) with respect to a change in its argument (input value). For example, the derivative of the position of a moving object with respect to time is the object's velocity: this measures how quickly the position of the object changes when time advances.

2.2.2 Integral

Coming Soon...

2.3 Linear Algebra

Branch of mathematics concerned with linear equations, maps and their purposes and value in vector and matrix spaces. The concept is foundational in other areas of mathematics, computing, and other realms of science and engineering.

2.3.1 Vector Addition

Two vectors of the same size can be added together by adding the corresponding elements, to form another vector of the same size, called the sum of the vectors. Vector addition is denoted by the symbol +. (Thus the symbol + is overloaded to mean scalar addition when scalars appear on its left- and right-hand sides, and vector addition when vectors appear on its left- and right-hand sides.)

2.3.2 Vector Subtraction

Coming Soon...

2.3.3 Vector Multiplication

Another operation is scalar multiplication or scalar-vector multiplication, in which a vector is multiplied by a scalar (i.e., number), which is done by multiplying every element of the vector by the scalar. Scalar multiplication is denoted by juxtaposition, typically with the scalar on the left.

2.4 Number Theory

A branch of pure mathematics with applications in cryptography, phyics, others areas of computation, etc. that primarily deals with the study of integers, integer valued functions, prime numbers, as well as the attributes and properties made from integers and their functions. This topic of mathematics is quite theoretical and is difficult to view the applications in a comprehensible sense. Beyond the basics of Number Theory, this module will also focus on encryption algorithms.

2.4.1 Caesar Cipher

Coming Soon...

2.4.2 Monoalphabetic Substitution Keyword Cipher

2.4.3 Rivest Cipher 2 (RC2) Encryption Algorithm

2.4.4 Rivest Cipher 4 (RC4) Encryption Algorithm

A stream cipher algorithm created by Ron Rivest (creator of RSA) previously declared insecure. RC4 generates a pseudorandom stream of bits (a keystream). As with any stream cipher, these can be used for encryption by combining it with the plaintext using bitwise exclusive or; decryption is performed the same way (since exclusive or with given data is an involution). This is similar to the one-time pad, except that generated pseudorandom bits, rather than a prepared stream, are used. To generate the keystream, the cipher makes use of a secret internal state which consists of two parts:

1. A permutation of all 256 possible bytes (denoted "S" below).

2. Two 8-bit index-pointers (denoted "i" and "j"). The permutation is initialized with a variable-length key, typically between 40 and 2048 bits, using the key-scheduling algorithm (KSA). Once this has been completed, the stream of bits is generated using the pseudo-random generation algorithm (PRGA).

2.4.5 Rivest Cipher 5 (RC5) Encryption Algorithm

Coming Soon...

2.4.6 Rivest Cipher 6 (RC6) Encryption Algorithm

Coming Soon...

2.5 Deep Learning

2.5.1 Linear Regression

A statistical algorithm with a linear approach for modeling the relationship between a scalar response and one or more explanatory variables (also known as dependent and independent variables). In linear regression, the relationships are modeled using linear predictor functions whose unknown model parameters are estimated from the data. Such models are called linear models.

2.5.2 Cross Validation

A resampling technique, the idea of this method is to train our model by utilizing the subsets of our data set then proceeding to evaluate + compare against the original. Essentially, use some part(s) of the data set for training, other part(s) for testing.

2.5.3 K-Fold Cross Validation

Split our data into a ‘k‘ number of subsets also called folds, and perform training/learning on the subsets leaving one ‘(k - 1)‘ for the final evaluation of the trained model. The method involves iterating ‘k‘ number of times using a different fold each time for testing against.

2.6 Complex

I had named this module accordingly due to its overall complexity and difficulty. This module name will be more clear as some topics will include the name. This module makes use of the openGL(Open Graphics Library) package for graphics purposes.

2.6.1 Topology

Topology is concerned with the properties of a geometric object that are preserved under continuous deformations, such as stretching, twisting, crumpling, and bending; that is, without closing holes, opening holes, tearing, gluing, or passing through itself.

Torus : A torus is a geometrical disc-shaped figure that is created by revolving a circle in a three-dimensional space about an axis that is coplanar with the cicle. Depending on the revolutions of the circle, different types of tori can be formed. Horn, spindle, and more types of torus are made when revolutions into a spherical disc are not met.

2.6.2 Dynamical Systems

Systems in which a function describes the time dependence of a point in an ambient space.

Fractals A geometric shape containing detailed structure at arbitrarily small scales, usually having a fractal dimension strictly exceeding the topological dimension.

- **Mandelbrot Fractal**:

- **Julia Fractal**:

2.6.3 Spline

A spline curve is a mathematical representation for which it is easy to build an interface that will allow a user to design and control the shape of complex curves and surfaces. The Complex module includes ways to create your own Non-Uniform Rational B-Splines (NURBS), Bezier Surfaces (as well as patches and curves)