Multi-precision Implementation for FADBAD++

MULTI-PRECISION IMPLEMENTATION FOR FADBAD++

BY

ZHENG GU, B.ENG.

A PROJECT

SUBMITTED TO THE DEPARTMENT OF COMPUTING AND SOFTWARE

AND THE SCHOOL OF GRADUATE STUDIES

OF MCMASTER UNIVERSITY

IN PARTIAL FULFILMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF ENGINEERING

Master of Engineering (2017)	McMaster University
(Computer Science)	Hamilton, Ontario, Canada

TITLE:	Multi-precision Implementation for FADBAD++
AUTHOR:	Zheng Gu Bachelor of Engineering
SUPERVISOR:	Dr. Ned Nedialkov
NUMBER OF PAGES:	viii, 42

Dedicated to my family and true friends.

Abstract

Automatic differentiation (AD) is a set of techniques to evaluate numerically the derivative of a function specified by a computer program. The FADBAD++ package developed by Ole Stauning and Claus Bendtsen implements the forward, backward, and Taylor modes utilizing C++ templates and operator overloading. It enables users to differentiate functions that are implemented in built-in C++ arithmetic types (such as double) or other customized class types. This report describes a multi-precision extension of the forward and Taylor modes in FADBAD++ using the GNU MPFR library.

Acknowledgements

I would like to profusely thank Dr. Ned Nedialkov for his invaluable inculcation and great patience that helped me accomplish this project. I also want to show my appreciation to Gary Tan and Shawn Li, two honorable men who gave me a lot of help and encouragement during the project and in daily life. I would also like to show my gratitude to every professor, such as Dr. Franya Franek, Dr. George Karakostas, Dr. Ridha Khedri etc, in my graduation courses, who selflessly shared their knowledge and experience with me.

Contents

List of Figures

Chapter 1

Introduction

Automatic differentiation (AD) is a set of techniques to evaluate numerically the derivative of a function specified by a computer program. AD exploits the fact that every computer program, no matter how complicated, executes a sequence of elementary arithmetic operations (addition, subtraction, multiplication, division) and elementary functions (exp, log, sin, cos, etc.) [1]. By applying the chain rule repeatedly to these operations, derivatives of arbitrary order can be computed automatically, and accurately to the working precision [4].

The FADBAD++ package developed by Ole Stauning and Claus Bendtsen implements the forward, backward, and Taylor modes utilizing C++ templates and overloaded operators. It enables users to differentiate functions that are implemented in built-in C++ arithmetic types (such as double) or other customized class types. The package consists of four files: `fadbad.h` defines elementary functions inside, `fadiff.h` defines a template `F<U>` to implement the forward mode, `badiff.h` defines a template `B<U>` to implement the backward mode, and `tadiff.h` defines a template `T<U>` to implement the Taylor mode.

To extend to multiple precision the forward and Taylor modes in the original FADBAD++ package, we need a library capable of processing computation on multi-precision floating-point numbers. In our work, we use MPFR (Multiple Precision Floating-Point Reliable Library), a portable library written in C for arbitrary precision arithmetic on floating-point numbers. However, the FADBAD++ package utilizes templates which require built-in C++ types or customized class types. Therefore we resort to MPFR C++, a wrapper for the MPFR library that implements constructors, destructors, overloaded operators, and other C++ features.The class in the wrapper MPFR C++ is named `mpreal`.

Many elementary functions in `fadbad.h`, such as `mySin`, `myCos`, and `myExp`, use the corresponding functions `sin`, `cos`, and `exp`, defined in `math.h`. However, these functions in the C numerics library `math.h` only support built-in types in C++ such as float and double, but do not support `mpreal`. Provided we have overloaded these elementary functions in the specialized operation struct in `fadbad.h`, we could use the `mpreal` class directly into the forward, backward, and Taylor series templates in the FADBAD++ package without doing any changes. But if we did that, there would be another deficiency: many temporary objects will be created from that. We will discuss the details in Chapter 4.

This report consists of four parts.

1. Summary of the FADBAD++ package

   Chapter 2 briefly introduces the mechanism of the FADBAD++ package, including the header file `fadbad.h`, where universal elementary functions are defined, the header files `fadiff.h` and `tadiff.h`, where template classes for the forward and Taylor modes are defined.

2. Overview of MPFR

   Chapter 3 gives an overview of the GNU MPFR library to introduce how it can be used to deal with multi-precision computation. Furthermore, we will give an introduction to MPFR C++, a C++ wrapper for the MPFR library.

3. Multi-precision extension in the FADBAD++ package

   Chapter 4 explains how to extend FADBAD++ to multiple precision. We will discuss issues about temporary objects construction if we use the `mpreal` class directly in FADBAD++ templates. Then, we will show details regarding the extension step by step.

4. Examples

   In Chapter 5, we will give several examples as a guide to show how to use the multi-precision feature in the modified FADBAD++ package.

Chapter 2

Summary of FADBAD++

We give a brief introduction to FADBAD++ in 2.1. In 2.2, we introduce the static elementary functions and explain why it is necessary to specialize the struct and overload those functions for a user-defined data type. In 2.3, we introduce the mechanism of the implementation of the forward mode in the FADBAD++ package and give a short example to explain how to use that template class. In 2.4, we introduce the mechanism of the implementation of the Taylor mode in the FADBAD++ package and give a short example as well.

2.1 Introduction

The FADBAD++ package contains templates for performing automatic differentiation on functions implemented in C++ code. If the source code of a program, which is an implementation of a differentiable function, is available, then FADBAD++ can be applied to obtain the derivatives of this function [7].

To apply automatic differentiation in a program, the arithmetic type used in the program is changed to a customized type (F<U>, B<U> or T<U>). F<U> is the template class for the forward mode implementation, B<U> is the template class for the backward mode implementation, and T<U> is the template class for the Taylor mode implementation in the FADBAD++ package.

2.2 The fadbad.h file

The file fadbad.h implements for a C++ typename T a templated struct Op<T> that contains 29 static member functions. We call these 29 static member functions elementary functions in the following context.

We can call these functions outside the struct Op<T> to execute arithmetic operations for variables in C++ built-in types, such as float and double. For example, Op<double>::myCadd(x,y) executes x+=y for a double type variable x and a double type variable y.

If T is a built-in type, then all the elementary functions in Op<T>, such as mySin, myCos, and myExp, call their corresponding functions sin, cos, and exp in the C numerics library math.h, which only supports C++ built-in types. For example, Op::mySin is defined by

```
static T mySin(const T &x) { return ::sin(x); }
```

If x is of type float or double, then Op<T>::mySin(x) is equivalent to sin(x), where sin is implemented in math.h.

However, if T is a user-defined data type, say Interval, and an elementary function, like Op<Interval>::myExp, is not specialized for Interval, then it will go to the general template class Op<T>, and call the function Op<T>::myExp inside. But, we cannot simply call `Op<Interval>::myExp(x)` for a `Interval` type variable `x` because the corresponding arithmetic function `exp` defined in `math.h` does not support the type Interval. Similarly, since `Op<Interval>::myExp` is called in the function `fadbad::exp` for a data type `F<Interval>` in the forward mode, as a chain effect, we cannot simply call `fadbad::exp(xf)`, where xf is an `F<Interval>` variable. In Chapter 4, we show how to specialize with the data type `mpreal` to the general template class `Op<T>`.

To inform the user about such specializations, `fadbad.h` puts before the templated struct `Op<T>` the following message:

The following template allows the user to change the operations that are used in FADBAD++ for computing the derivatives. This is useful as an example for specializing with non-standard types such as interval arithmetic types.

2.3 Forward Method

The forward method of AD is implemented by a template class `F<U>`. An `F<U>` object contains a private member m_val of `U` type to store the value and an array of `U` type to store the gradient of the variable. Following the chain rule repeatedly, the gradient array inside the result `F<U>` object will contain all the partial derivatives with respect to independent variables in the function.

There are two different versions of the `F<U>` template class. The version where the gradient array is allocated statically is called stack version, while the other one where the gradient array is allocated dynamically is called heap version. Since the mechanism and usage of these two versions are identical, we will just take the heap version for instance in this report.

2.3.1 Overloaded elementary functions in fadiff.h

The arithmetic operators and the elementary functions are overloaded in `fadiff.h`. Here, as an example, we give the definition of the overloaded `sin` function.

```
template <typename T>
F<T> sin(const F<T>& a)
{
   F<T> c(Op<T>::mySin(a.val()));
   if (!a.depend())
   return c;
   T tmp(Op<T>::myCos(a.val()));
   c.setDepend(a);
   for (unsigned int i = 0; i < c.size(); ++i)
   c[ i ]            = a[ i ] * tmp;
   return c;
}
```

An `F<U>` object contains a value and a gradient. In those overloaded functions in `fadiff.h`, the elementary operation functions such as `mySin`, `myCos`, and `myExp` in the `fadbad.h` are called. For example, in the definition of the `sin` function above, static functions `mySin` and `myCos` are called to compute the value and the gradient array. Hence, by following the chain rule step by step, it will output a final result, an `F<U>` object with a value and a gradient of `U` type inside [6]. From the gradient array, all the partial derivatives could be obtained with respect to different variables.

2.3.2 An introductory example of using the forward mode template class

Suppose we have the function

$$f(x, y, z) = x + y \times \sin(z) \tag{2.1}$$

Then, we want to obtain partial derivatives with respect to x, y, and z, df/dx, df/dy, and df/dz. Now we are ready to differentiate this function by using the forward method template class `F<U>`.

If we work with doubles, all the input arguments should be of type `F<double>` as well as the returned variable.

```
F<double> func(const F<double>& x, const F<double>& y,  const
   F<double>& z)
{
        return x+y*sin(z);
}
```

Our function above is now prepared for computing derivatives. Before we call the function, we have to specify the variables we want to differentiate with respect to. After the call, we obtain the function value and the derivatives stored in the object f. This can be done with the following code

```
F<double> x,y,z,f;      // Declare variables x,y,z,f
x=1;                    // Initialize variable x with value 1
x.diff(0,3);            // Set x as an independent variable of
   index 0 of 3
y=2;                    // Initialize variable y with value 2
y.diff(1,3);            // Set y as an independent variable of
   index 1 of 3
z=3;                    // Initialize variable z with value 3
z.diff(2,3);            // Set z as an independent variable of
   index 2 of 3
f=func(x,y,z);          // Evaluate function and derivatives
double fval=f.x();      // Value of function
double dfdx=f.d(0);     // Value of df/dx (index 0 of 3)
double dfdy=f.d(1);     // Value of df/dy (index 1 of 3)
double dfdz=f.d(2);     // Value of df/dz (index 2 of 3)
```

The forward method is very natural and easy to implement as the flow of derivative information coincides with the order of evaluation [1]. Suppose we want to compute derivatives with respect to x, y, and z in the equation 2.1. If we apply the chain rule in the forward mode, here are the steps.

code list		expression	gradient
		x	$\nabla x = (1, 0, 0)$
		y	$\nabla y = (0, 1, 0)$
		z	$\nabla z = (0, 0, 1)$
v_1	$= \sin(z)$	$\sin(z)$	$\nabla v_1 = \cos(z) \cdot \nabla z$
v_2	$= y \times v_1$	$y \sin(z)$	$\nabla v_2 = y \cdot \nabla v_1 + v_1 \cdot \nabla y$
$f = v_3$	$= x + v_2$	$x + y \sin(z)$	$\nabla v_3 = \nabla x + \nabla v_2$

Table 2.1: Steps of the chain rule in the Forward mode

Correspondingly, we apply the chain rule to our corresponding function in C++, where the variables f, x, y and z are of `F<double>` type. In each step, we represent the temporary result with v of `F<double>` type comprising the value and the gradient array.

Here, we explain how to compute the gradients.

- First, using `x.diff(0,3), y.diff(1,3), z.diff(2,3)`, we set independent variables and now $\nabla x = (1, 0, 0)$, $\nabla y = (0, 1, 0)$, $\nabla z = (0, 0, 1)$.

- We call the function `sin` defined in `fadiff.h` to compute the returned variable (denoted as v_1) of `F<double>` type. v_1 comprises a value and a gradient array of type double, which are obtained by using the value and gradient array stored in the object z.

- Here we evaluate $y \times v1$. The program calls the overloaded operator “`*`” in `fadiff.h` and returns a temporary `F<double>` variable denoted as v_2. v_2 comprises a value and a gradient array of type double, obtained by using the values and gradient arrays of type double stored in object y and v_1.

- To evaluate v_1+v_2, where v_1, v_2 are `F<double>` type variables, the program calls the overloaded operator "`+`" in `fadiff.h` and returns a temporary `F<double>` variable, denoted as v_3.

- Finally, $f = v_3$ is processed. The program calls the overloaded operator "`=`" in `fadiff.h`. The value and gradient of type double in the object f to the working precision is obtained (because of using double type, the working precision here is 64-bit).

2.4 Taylor mode and the mechanism of implementation in tadiff.h

Taylor mode is implemented by the template class `T<U>` defined in the file `tadiff.h`. We can use the template class `T<U>` to compute the Taylor series of the result of the function.

The functions in `tadiff.h` like `sin`, `exp` will now "record" a directed acyclic graph (DAG) while computing the function value (which is the 0'th order Taylor-coefficient) [7]. This DAG can then be used to find the Taylor coefficients if the order of the Taylor expansion is given [3].

To obtain the whole DAG for all kinds of available arithmetic operation, an arithmetic operation is supposed to correspond to an unique class type. For example, if we call `sin(x)`, where x is a variable of type `T<double>`, it will create an object of the corresponding class `TTypeNameSin<U, N>` (`N` is the highest order of the coefficient in the Taylor series, which is 40 by default), meanwhile the 0'th coefficient of the Taylor series in that object is evaluated. Similarly, if we call `exp(x)`, where x is a variable of type `T<double>`, it will create an object of the corresponding class `TTypeNameEXP<U, N>`, meanwhile the 0'th coefficient of the Taylor series in that object is evaluated.

An unary arithmetic operation, say `sin`, corresponds to a class that has a pointer to the single operand, while a binary arithmetic operation, say `pow`, corresponds to a class that has two pointers to the two operands.

In each corresponded class, the virtual function `eval` is used to compute the resulting Taylor coefficient to the given order using the Taylor coefficients stored in the operand objects. In that function, relevant operation functions from `Op<T>` are called to obtain result values. For example, in the function `eval` in the class `TTypeNameSin<U, N>` which corresponds to the arithmetic operation `sin`, elementary functions from the templated struct `Op<T>`, such as `Op<T>::myCos`, `Op<T>::mySin`, `Op<T>::myCadd`, `Op<T>::myCdiv`, and `Op<T>::myCsub` are called inside.

2.5 An introductory example of using the Taylor mode template class

Suppose we have the function

$$f(x(t), y(t)) = x(t) + \sin(y(t)) \tag{2.2}$$

We want to obtain the Taylor coefficients of $f(t)$. Now we are ready to compute the Taylor series $f(t)$ by using the Taylor series template class `T<U>`.

First, we need to adapt our function 2.2 to a function in C++. All the input arguments should be of type `T<double>` as well as the returned variable.

```
T<double> func(const T<double>& x, const T<double>& y)
{
        return x+sin(y);
}
```

Before we call the function above, we have to specify the coefficients of the Taylor series in the input variables x and y. After the call, we obtain coefficients of the Taylor series stored in the object f. This can be done with the following code

```
T<double> x,y,f;      // Declare variables x,y,f
x=1;                  // Initialize 0'th coefficient of x
x[1]=1;               // Initialize 1'st coefficient of x
x[2]=2;               // Initialize 2'nd coefficient of x
y=2;                  // Initialize 0'th coefficient of y
y[1]=1;               // Initialize 1'st coefficient of y
f=func(x,y);          // Evaluate function and record DAG
double fval=f[0];     // Value of function which is also 0'th
    coefficiet of f
f.eval(10);           // Evaluate Taylor series f to order 10
// f[0]...f[10] now contains the Taylor-coefficients.
```

The recorded DAG for this statement is

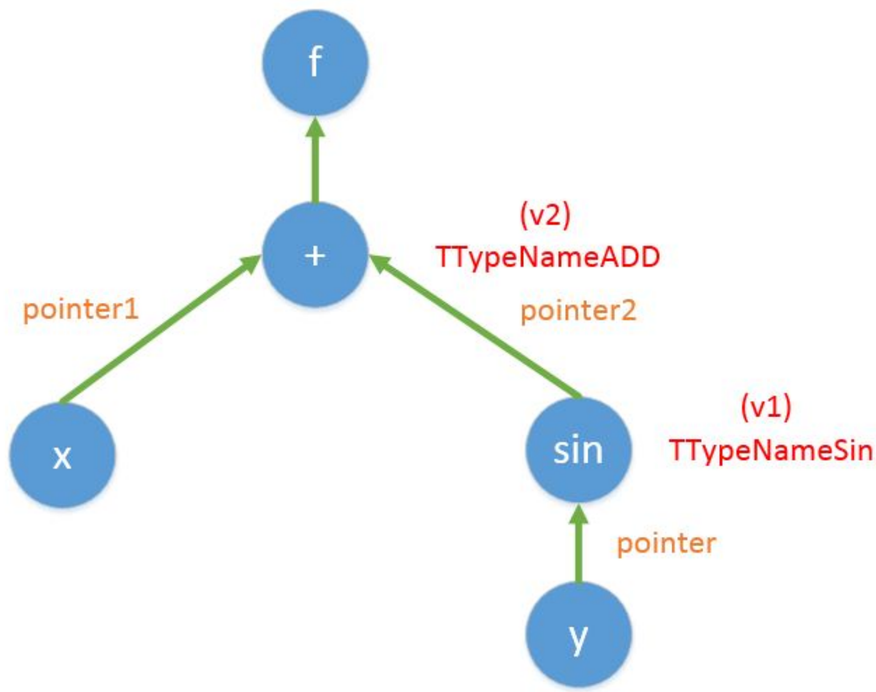

Figure 2.1: Directed Acyclic Graph Example

Here are the steps for the AD Taylor mode implementation in the FADBAD++ package.

- We initialize x and y of type `T<double>` and specify the coefficients of the Taylor series for x and y. Thus, the Taylor series of x is $x(t) = 1 + t + 2t^2$ and the Taylor series of y is $y(t) = 2 + t$.

- When processing `sin(y)`, an object of `TTypeNameSin<U, N>` type (denoted as v_1) will be constructed, which contains a pointer to the single operand y. The 0'th order Taylor-coefficient of v1 will be evaluated.

- When we evalute $x + v_1$, an object of `TTypeNameADD<U, N>` type (denoted as

v_2) will be constructed, which contains two pointers to operands x and v_1 (v_1 is of type `TTypeNameSin<U, N>`). The 0'th order Taylor-coefficient of v_2 will be evaluated.

- Finally, the result is assigned to variable f. The program calls the overloaded operator "`=`" in `fadiff.h`.

- If more orders of the Taylor coefficients in f are required, each object like v_1 and v_2 on the path in the recorded DAG will be re-evaluated to the given order from bottom to top. In the code above, each object in the recorded DAG is re-evaluated to order 10.

Chapter 3

Overview of MPFR

MPFR, short for Multiple Precision Floating-Point Reliable Library, is a portable library written in C for arbitrary precision arithmetic on floating-porint numbers. It is based on the GMP library, aiming to provide a class of floating-point numbers with precise semantics.

We cover a series of basic functions which we use in the multi-precision extension in the FADBAD++ package to be discussed in Chapter 4. For more details regarding the GNU MPFR library, please read the official GNU MPFR manual [2].

3.1 MPFR library functions

3.1.1 Initialization Functions

A `mpfr_t` object must be initialized before storing the first value in it.

```
void mpfr_init2 (mpfr_t x, mpfr_prec_t prec)
```

initializes x, sets its precision to the default precision, and sets its value to NaN.

`void mpfr_init (mpfr_t x)`

initializes x, sets its precision to be exactly `prec` bits and its value to NaN.

`void mpfr_clear (mpfr_t x)`

This function should be called when a `mpfr_t` variable is not used any more.

3.1.2 Arithmetic Functions

A series of arithmetic functions are given in the MPFR library to process corresponding arithmetic operations for `mpfr_t` type variables. In these arithmetic functions, a parameter needs to be set for the rounding mode. For example:

`int mpfr_add (mpfr_t rop, mpfr_t_op1, mpfr_t_op2, mpfr_rnd_t rnd)`

gets the addition of two `mpfr_t` variables and stores the result in the first parameter. Rounding mode needs to be set in the parameter rnd.

`int mpfr_sqr (mpfr_t_rop, mpfr_t_op, mpfr_rnd_t rnd)`

`int mpfr_sin (mpfr_t_rop, mpfr_t op, mpfr_rnd_t rnd)`

Some functions like these two above are used to get the square root or sin value of a `mpfr_t` variable.

3.1.3 Default Precision and Default Rounding Mode

Default Precision Setting

`void mpfr_set_default_prec (mpfr_prec_t prec)`

sets the default precision to be exactly `prec` bits, where `prec` can be any integer between `MPFR_PREC_MIN` and `MPFR_PREC_MAX`. The precision of a variable means the number of bits used to store its significand. The default precision is set to 53 bits initially.

Default Rounding Mode Setting

`void mpfr_set_default_rounding_mode (mpfr_rnd_t rnd)`

sets the default rounding mode to `rnd`, using one of the rounding mode: `MPFR_RNDN`, `MPFR_RNDZ`, `MPFR_RNDU`, `MPFR_RNDD`,and `MPFR_RNDA`. The default rounding mode is to nearest initially.

3.2 MPFR C++

Since MPFR library is written in C, it does not support the C++ features like class constructor, destructor and overloaded operators etc. To make it compatible with the AD templates in the FADBAD++ package, a C++ interface or a wrapper for MPFR is required.

MPFR C++ written by Pavel Holoborodko uses a modern C++ design with coverage of classes, templates and function objects. The class wrapping the GNU MPFR library is defined in the header file `mpreal.h` and named `mpreal`, which is to be used in the multiple-precision extension to the FADBAD++ package [5].

Chapter 4

Multi-precision extension in the FADBAD++ package

In this chapter, we will first discuss how to specialize the templated struct `Op<T>` in `fadbad.h` to enable us to use a user-defined class type `mpreal` instead of built-in C++ type like float and double in the AD templates in the FADBAD++ package. Then we will discuss the problem of temporary objects construction. Finally, we will show how we eliminate temporary objects in this multi-precision extension implementation.

4.1 Specialization of the templated struct `Op<T>` in fadbad.h

The template classes in the original FADBAD++ package enable users to differentiate functions that are implemented in built-in C++ types, such as float and double. However, if we want to implement multi-precision in FADBAD++, we need to use a user-defined class type, say `mpreal` which supports all the multi-precision operations in the GNU MPFR library and modern C++ features.

We need to specialize the template `Op<T>` with the class `mpreal`. The operators used in the general template `Op<T>` like "+", "+=", "*" are all overloaded in the the class `mpreal`.

For example, the overloaded operator "+=" is defined in the class `mpreal` as

```
inline mpreal& mpreal::operator+=(const mpreal& v)
{
        mpfr_add(mpfr_ptr(), mpfr_srcptr(), v.mpfr_srcptr(),
           mpreal::get_default_rnd());
        MPREAL_MSVC_DEBUGVIEW_CODE;
        return *this;
}
```

If we have a statement `x+=y`, where x and y are all `mpreal` variables, it will call the overloaded operator "+=" shown above. Since these operators are all overloaded, we could directly use them in the specialized `Op<mpreal>`.

As for elementary functions that have called corresponding elementary functions defined in the C numerics library `math.h` like

```
static T mySin(const T &x) { return ::sin(x); }
```

we could see that `sin` is also overloaded in the class `mpreal` as

```
const mpreal sin(const mpreal& v, mp_rnd_t rnd_mode);
```

we could directly use the function `mpreal::sin` to replace the function `::sin` defined in `math.h`.

```
Op<mpreal>::mysin(const mpreal &x) { return mpreal::sin(x); }
```

From the example above and based on the specialized templated struct `Op<mpreal>`, we could directly replace built-in C++ types like float and double with the user-defined class `mpreal` in the forward and Taylor mode template class as `F<mpreal>` and `T<mpreal>`, and accomplish the multi-precision extension in the FADBAD++ package.

4.2 Temporary objects issue

However, we come to another problem, the incessant construction and destruction of temporary objects by using overloaded operators like "+", "-", "*", and "/", and elementary functions like `mpreal::sin` in `mpreal` class. For example, consider the following code.

```
for(unsigned int i=0;i<10;++i)
{
        f = x / Op<T>::mySin(y);
}
```

If the data type of the variables f, x, and y is double, `Op<double>::mySin(y)` will be called to return a temporary result of type double and divide the double variable x for 10 times. However, if the data type of the variables f, x, and y is of type `mpreal`, circumstance seems to be more complicated, which involves construction and destruction of temporary objects.

From the `mpreal.h`, we could see that the declaration for the overloaded operator / is

```
const mpreal operator/(const mpreal& a, const mpreal& b);
```

`Op<mpreal>::mySin(y)` is called to return a temporary object of class `mpreal`. Then, this temporary object divides x and from the declaration above, the division will return another temporary object of class `mpreal` too, which will be assigned to the variable f. According to the C++ mechanism, a temporary object of a class will be constructed first and at the end of the statement, the temporary object will be destructed [9]. Although it is a very neat way, these construction and destruction procedures will be iteratively executed 10 times. Hence, we would like to come up with a way to avoid unnecessary temporary objects from being constructed especially those in the loops or nested loops, which will be a severe performance hit.

4.3 Temporary objects elimination

In this section, we discuss how to avoid temporary objects in the loops from being generated.

4.3.1 Improving the specialized templated struct `Op<mpreal>` in fadbad.h

In Section 4.1, we used overloaded elementary functions defined in `mpreal.h` to replace elementary functions defined in the C numerics library `math.h`. But they will return temporary objects of the type `mpreal`. Hence, we will discuss how to amend these functions to avoid returning temporary objects.

Adding Operation functions to replace direct use of overloaded operators defined in mpreal.h

First, we consider the use of overloaded operators “`+`”, “`-`”, “`*`”, and “`/`”. Suppose we have the statement

$$x + y$$

where x and y are all of type `mpreal`. It will call the overloaded operator “`+`” defined in `mpreal.h`. The declaration of the overloaded operator “`+`” defined in `mpreal.h` is

```
const mpreal operator+(const mpreal& a, const mpreal& b)
```

It will always return a temporary variable of type `mpreal`. Noticing that the declaration of the add operation function defined in the MPFR library is

```
int mpfr_add (mpfr_t rop , mpfr_t op1 , mpfr_t op2 ,
   mpfr_rnd_t rnd )
```

The result is not returned as an object, but stored in the object the first parameter `rop` points to. Hence, this function does not return a temporary object of type `mpreal`. Using this kind of arithmetic operation functions defined in the MPFR library directly will be an alternative way to replace the use of overloaded operators “`+`”, “`-`”, “`*`”, and “`/`” defined in `mpreal.h`.

Here, we will add some functions for addition, subtraction, multiplication, and division in the specialized templated struct `Op<mpreal>`, which will be used to replace the use of overloaded operators “`+`”, “`-`”, “`*`”, and “`/`” defined in `mpreal.h`. For example, the definition for the addition function in `Op<mpreal>` to replace the use of overloaded operator “`+`” is

```
static void mpreal_add(mpreal &rop, const mpreal &op1, const
    double &op2, mpfr_rnd_t rnd = DEFAULT_RNDM)
{
        if (rop.get_prec() != DEFAULT_PREC)
        rop.set_prec(DEFAULT_PREC, DEFAULT_RNDM);
        mpfr_add_d(rop.mpfr_ptr(), op1.mpfr_ptr(), op2, rnd);
}
```

The result is returned through a pointer `rop` as the first parameter in this function instead of being returned as an object to avoid temporary objects from being generated. Inside the function, function `mpf_add_d` is directly called to process the addition. Similar to other overloaded operators “`-`”, “`*`”, and “`/`”, we will add functions into `Op<mpreal>` to process subtraction, multiplication and division without returning temporary objects.

Improving specialized elementary functions

We need to consider the use of the overloaded elementary functions defined in `mpreal.h`, which will return temporary objects of type `mpreal`. For example, the declaration of the overloaded `sin` function defined in `mpreal.h` is

```
const mpreal sin(const mpreal& v, mp_rnd_t rnd_mode);
```

It will return the result as a temporary object of type `mpreal`. Similar to the operator “`+`”, we could find a function in the MPFR library to have the same functionality.

```
int mpfr_sin (mpfr_t rop , mpfr_t op , mpfr_rnd_t rnd)
```

The result here is returned through the pointer `rop` as the first parameter. In this way does this function avoid temporary objects from being generated. Then, we replace all the overloaded elementary functions occurring in the specialized templated struct

`Op<mpreal>` with functions that have the same functionality defined in the MPFR library. For example,

```
Op<mpreal>::mysin(const mpreal &x) { return mpreal::sin(x); }
```

is amended to

```
static void mpreal_sin(mpreal &rop, const mpreal &x,
    mpfr_rnd_t rnd = DEFAULT_RNDM)
{
if (rop.get_prec() != DEFAULT_PREC)
rop.set_prec(DEFAULT_PREC, DEFAULT_RNDM);
mpfr_sin(rop.mpfr_ptr(), x.mpfr_ptr(), rnd);
}
```

Similarly, we replace the functions such as `cos`, `tan`, `exp`, `log`, using the functions defined in the MPFR library directly.

So far, all the functions in the specialized templated struct `Op<mpreal>` do not return results as temporary objects.

4.3.2 Temporary objects elimination method in fadiff.h

We modify every templated elementary function in `fadiff.h`.

- As for all the templated elementary functions in `fadiff.h`, we need to give a specialization for our user-defined data type `mpreal`. Hence, when calling the functions with parameters of type `mpreal`, the specialized elementary functions will be called instead of the general one. For example, the original declaration of the general templated `sin` function is

```
template <typename T>
inline FTypeName<T, 0> sin(const FTypeName<T, 0>& a)
```

We need to specialize this general template to our user-defined data type `mpreal`. Then, the declaration becomes as

```
inline FTypeName<mpreal, 0> sin(const FTypeName<mpreal,
   0>& a)
```

- We replace the arithmetic operators "+","-", "*", and "/" and functions defined in the struct `Op<T>` with elementary functions defined in the specialized struct `Op<mpreal>` step by step and keep the precedence of all the operands in the statements. For example, suppose we have

  `a+b*Op<T>::sin(c)`

  In this statement, two main points should be paied attention to.

  - Functions defined in `Op<T>` should be replaced with corresponding elementary functions defined in `Op<mpreal>`.
  - We should avoid the use of arithmetic operators "+","-", "*", and "/" and replace them with the corresponding elementary functions defined in the struct `Op<mpreal>`. Meanwhile, we need to focus on the right precedence for all the operands.

  After the modification, this statement is rewritten as

```
Op<mpreal>::mpreal_sin(TEMP_RESULT, c);
Op<mpreal>::mpreal_mul(TEMP_RESULT1, b, TEMP_RESULT);
OP<mpreal>::mpreal_add(TEMP_RESULT, a, TEMP_RESULT);
```

To store indispensable temporary results among the calculations, we need to create several static variables of type `mpreal` in advance. Empirically, two will be sufficient in our implementation. `TEMP_RESULT` and `TEMP_RESULT1` are two static variables of type `mpreal` defined in advance in the struct `Op<mpreal>`. By keeping the right precedence for all the operands and modifying the original statement step by step, `TEMP_RESULT` is the final result.

4.3.3 Temporary objects elimination method in tadiff.h

In tadiff.h, two main places will be modified.

1. We modify every arithmetic class in `tadiff.h` like `TTypeNameSIN` and `TTypeNameEXP`.

   - We specialize all the general arithmetic class templates like `TTypeNameEXP<T, N>` with our user-defined data type mpreal. For example, The original declaration of our general arithmetic class `TTypeNamEXP` is

     ```
     template <typename U, int N>
     struct TTypeNameEXP : public UnTTypeNameHV<U, N>
     ```

     Then we need to specialize this general template with our user-defined data type `mpreal`. After the modification, the declaration of the specialized form is

     ```
     template <int N>
     struct TTypeNameEXP : public UnTTypeNameHV<mpreal, N>
     ```

   - In the virtual function `eval` defined in every class like `TTypeNameMUL<mpreal, N>`, `TTypeNamePOW<mpreal,N>` etc, similar to the modification in `fadiff.h`, we replace operators such as $+, -, *, /$ and elementary functions defined in the struct `Op<T>` with corresponding functions from struct `Op<mpreal>`. For example, suppose we have this statement in the virtual function `eval` in the class `TTypeNameEXP<mpreal,N>`.

```
Op<U>::mycadd(a, (Op<U>::myOne() - Op<U>::myInteger(j)
   }/ Op<U>::myInteger(i)) * b * c);
```

Here is the precedence for all the operands in this statement.

- `temp_result1 = Op<U>::myInteger(j) / Op<U>::myInteger(i);`
- `temp_result2 = Op<U>::myOne() - temp_result1;`
- `temp_result3 = temp_result2 * b;`
- `temp_result4 = temp_result3 * c;`
- `a += tmep_result4;`

Then, We replace all the arithmetic operators and functions defined in `Op<U>` with elementary functions defined in the specialized `Op<mpreal>`. Here is the modification for each step.

- `TEMP_RESULT1 = (double)j/(double)i;`
- `Op<mpreal>::mpreal_sub(TEMP_RESULT2, 1, TEMP_RESULT1);`
- `Op<mpreal>::mpreal_mul(TEMP_RESUL3, TEMP_RESULT2, b);`
- `Op<mpreal>::mpreal_mul(TEMP_RESULT4, TEMP_RESUL3, c);`
- `Op<mpreal>::myCadd(a, TEMP_RESULT4);`

Since no more than two temporary results are used in the same function, we only need two static `mpreal` objects defined in advance in the struct `Op<mpreal>`. Thus, the final version of the modification is

```
TEMP_RESULT1 = (double)j/(double)i
Op<mpreal>::mpreal_sub(TEMP_RESULT, 1, TEMP_RESULT1)
Op<mpreal>::mpreal_mul(TEMP_RESUL1, TEMP_RESULT, b)
Op<mpreal>::mpreal_mul(TEMP_RESULT, TEMP_RESUL1, c)
Op<mpreal>::myCadd(a, TEMP_RESULT)
```

where `TEMP_RESULT` and `TEMP_RESULT1` are static variables of data type `mpreal` defined in advance in the templated struct `Op<mpreal>`.

2. We modify overloaded arithmetic operation functions (such as operator "+", "-", "*", "/" and `sin`, `cos`, `exp` and so on) in tadiff.h.

   - As for all the template elementary functions in `fadiff.h`, we need to give a specialization to our user-defined data type `mpreal`. Hence, when calling the functions with parameters of type `mpreal`, the specialized arithmetic operation functions will be called instead of the general ones. For example, the original declaration of the general arithmetic operation function template is

     ```
     template <typename U, int N>
     TTypeName<U, N> exp(const TTypeName<U, N>& val)
     ```

     Then, We specialize this general form to our user defined data type `mpreal` as

     ```
     template <int N>
     TTypeName<mpreal, N> exp(const TTypeName<mpreal, N>&
        val)
     ```

   - In these arithmetic operation functions in tadiff.h, any arithmetic operators or elementary functions defined within `Op<U>` used to evaluate the first item in Taylor expansions should be replaced with corresponding functions defined in `Op<mpreal>`. Temporary results from the evaluation should be stored first in the static `mpreal` objects defined in the specialized struct `Op<mpreal>` before passed into a constructor. For example, suppose we have the following statement in `exp`

```
new TTypeNameEXP<U, N>(Op<U>:: myExp(a));
```

where `a` is a variable of data type U.

This statement is modified in the specialized function for our use-defined data type `mpreal` as

```
Op<mpreal>::mpreal_exp(TEMP_RESULT, a);
new TTypeNameExp<mpreal, N>(TEMP_RESULT);
```

where `TEMP_RESULT` is a static variable of data type `mpreal` defined in advance in the templated struct `Op<mpreal>`.

Chapter 5

Examples

In this chapter, two examples are given to show how to use the multi-precision extension feature in the forward mode and Taylor mode templates in the modified FADBAD++ package. In each example, the max norm between the double datatype and the mpreal datatype is used to check the difference to verify the correctness of our modification of the original FADBAD++ package.

5.1 The heap-form template in the forward method

We modified ExampleFAD2.cpp from the distribution of FADBAD++ as follows.

```
#include <iostream>
#include "fadiff.h"
#define TERMS 2
using namespace std;
using namespace fadbad;
template <typename T> F<T> func(const F<T> *x_in, int n);
template <typename T> void show_result(F<T> &f_result, int n);
template <typename T, typename U>
void show_norms(F<T> &f_result1, F<U> &f_result2, int n);
int main()
{
  // double type
  // variables initiation
  F<double> f_double;                  // Declare variables f
  F<double> x_double[ TERMS ];         //  Declare 2 variables
  x_double[ 0 ] = 0.512;               // Initialize variable x
  x_double[ 1 ] = 2.141;               // Initialize variable y
  x_double[ 0 ].diff(0, TERMS);        // Differentiate with
     respect to x (index 0 of 2)
  x_double[ 1 ].diff(1, TERMS);        // Differentiate with
     respect to y (index 1 of 2)
  f_double = func(x_double, TERMS);    // Evaluate function and
     derivatives
  // output
  int output_prec = 15;
  cout.precision(output_prec);
  cout << "---------------------------------------------\n";
  cout << "Computed in double precision, \noutput in " <<
     output_prec << " digits" << endl;
  show_result(f_double, TERMS);
  // Settings for mpreal
  int prec = 128;
  /*Set the default working precision for the mpreal data type
     , the default working precision is 53
```

```
   * bit which is the same as double*/
  mpfr_set_default_prec(prec);
  // Stack-form template
  // variables initiation
  F<mpreal> f_mpreal;                    // Declare variables x,y,
     f
  F<mpreal> x_mpreal[ TERMS ];           // Declare two variables
  x_mpreal[ 0 ] = 0.512;                 // Initialize variable x
  x_mpreal[ 1 ] = 2.141;                 // Initialize variable y
  x_mpreal[ 0 ].diff(0, TERMS);          // Differentiate with
     respect to x (index 0 of 2)
  x_mpreal[ 1 ].diff(1, TERMS);          // Differentiate with
     respect to y (index 1 of 2)
  f_mpreal = func(x_mpreal, TERMS);  // Evaluate function and
     derivatives
  // output
  cout << "----------------------------------------------\n";
  cout << "Computed in MPFR precision " << prec << " digs" <<
     endl;
  cout << "output in " << output_prec << " digits" << endl;
  show_result(f_mpreal, TERMS);
  // show_norms
  int norm_output_prec = 5;
  cout.precision(norm_output_prec);
  cout << "----------------------------------------------\n";
  cout << "Errors between double and MPFR precision " << prec
     << "\n";
  show_norms(f_double, f_mpreal, TERMS);
  return 0;
}
template <typename T>
F<T> func(const F<T> *x_in, int n)
{
  F<T> x_out;
  x_out = atan(x_in[ 0 ]) * x_in[ 1 ];
  return x_out;
}
template <typename T>
void show_result(F<T> &f_result, int n)
{
  T  fval  = f_result.x();  // Value of function
  T *f_der = new T[ n ];
  for (int i = 0; i < n; i++)
```

```
    f_der[ i ] = f_result.d(i);  // get Value for each
        derivative
  // output
  cout << "f        = " << fval << endl;
  for (int i = 0; i < n; i++)
    cout << "df/d" << i << "th = " << f_der[ i ] << endl;  //
        output each derivative
  delete[] f_der;
}
template <typename T, typename U>
void show_norms(F<T> &f_result1, F<U> &f_result2, int n)
{
  // max-norm;
  cout << "fval    : " << fabs(f_result1.x() - f_result2.x())
      << endl;
  for (int i = 0; i < n; i++)
    cout << "df/d" << i << "th: " << fabs(f_result1.d(i) -
        f_result2.d(i)) << endl;
}
```

First of all, to use the templates defined in the namespace fadbad, we need to declare `using namespace fadbad`. Then, we need to set the default working precision and the default rounding mode for all the mpreal variables by using the two library functions in the GNU MPFR library below

`void mpfr_set_default_prec (mpfr_prec_t prec)`

`void mpfr_set_default_rounding_mode (mpfr_rnd_t rnd)`

In this version, we use the heap-form forward method template to define all the variables. We use mpreal whose working precision is 128 bits and double whose precision is 53 bits to calculate the final result in the function `func`. Finally, we check the difference using the max norm between mpreal and double to verify the correctness of the modification. The output is

```
-----------------------------------------------
Computed in double precision,
output in 15 digits
f       = 1.01312432251662
df/d0th = 1.69631991278333
df/d1th = 0.473201458438403
-----------------------------------------------
Computed in MPFR precision 128 digs
output in 15 digits
f       = 1.01312432251662
df/d0th = 1.69631991278333
df/d1th = 0.473201458438403
-----------------------------------------------
Errors between double and MPFR precision 128
fval    : 4.5637e-17
df/d0th: 8.5219e-17
df/d1th: 2.5721e-17
```

5.2 The Taylor-expansion method template

In ExampleTAD1.cpp

```
#include <iostream>
#include "tadiff.h"
#define TERMS 2
#define ORDER 10
using namespace std;
using namespace fadbad;
template <typename U> T<U> func(const T<U> *x_in, int n);
template <typename U> void show_result(T<U> &f_result);
template <typename U, typename G>
void get_max_norm(T<U> &f_result1, T<G> &f_result2);
int main()
{
  // double type
  // variables initiation
  T<double> f_double;            // Declare variables f
  T<double> x_double[ TERMS ];   //  Declare 2 variables
  x_double[ 0 ][ 0 ] = 1.121;
  x_double[ 0 ][ 1 ] = 1.353;   // Taylor-expand wrt. x (dx/dx
     =1)
  x_double[ 0 ][ 2 ] = 5.21234;
  x_double[ 1 ][ 0 ] = 2.221;
  x_double[ 1 ][ 1 ] = 2.253;
  x_double[ 1 ][ 2 ] = -12.123;
  f_double           = func(x_double, TERMS);  // Evaluate
     function and record DAG
  f_double.eval(ORDER);                        // Taylor-
     expand f to degree ORDER
  // f[0]...f[ORDER] now contains the Taylor-coefficients.
  // output
  int output_prec = 15;
  cout.precision(output_prec);
  cout << "-----------------------------------------------\n";
  cout << "Computed in double precision, \noutput in " <<
     output_prec << " digits" << endl;
  show_result(f_double);
  // Settings for mpreal
  int prec = 128;
```

```
/*Set the default working precision for the mpreal data type
   , the default working precision is 53
 * bit which is the same as double*/
mpfr_set_default_prec(prec);
/*Using one of the 5 rounding mode parameter:
MPFR_RNDN, MPFR_RNDZ, MPFR_RNDU, MPFR_RNDD, MPFR_RNDA
to set the default rounding mode, the default rounding mode
   is MPFR_RNDN.*/
mpfr_set_default_rounding_mode(MPFR_RNDN);
// mpreal type
// variables initiation
T<mpreal> f_mpreal;           // Declare variables f
T<mpreal> x_mpreal[ TERMS ];  //  Declare 2 variables
x_mpreal[ 0 ][ 0 ] = 1.121;
x_mpreal[ 0 ][ 1 ] = 1.353;  // Taylor-expand wrt. x (dx/dx
   =1)
x_mpreal[ 0 ][ 2 ] = 5.21234;
x_mpreal[ 1 ][ 0 ] = 2.221;
x_mpreal[ 1 ][ 1 ] = 2.253;
x_mpreal[ 1 ][ 2 ] = -12.123;
f_mpreal             = func(x_mpreal, TERMS);  // Evaluate
   function and record DAG
mpreal fval_mpreal;                            // Declare
   variables x,y,f
fval_mpreal = f_mpreal[ 0 ];                   // Value of
   function
f_mpreal.eval(ORDER);                          // Taylor-
   expand f to degree ORDER
// f[0]...f[ORDER] now contains the Taylor-coefficients.
// output
cout.precision(output_prec);
cout << "----------------------------------------------\n";
cout << "Computed in MPFR precision " << prec << " digs" <<
   endl;
cout << "output in " << output_prec << " digits" << endl;
show_result(f_mpreal);
// show max_norm
int norm_output_prec = 5;
cout.precision(norm_output_prec);
cout << "----------------------------------------------\n";
cout << "max_norm between double and MPFR precision " <<
   prec << "\n";
get_max_norm(f_double, f_mpreal);
```

```
  return 0;
}
template <typename U>
T<U> func(const T<U> *x_in, int n)
{
  T<U> x_out;
  x_out = sin(x_in[ 0 ] + x_in[ 1 ] / 3.2 - cos(5.263));
  return x_out;
}
template <typename U>
void show_result(T<U> &f_result)
{
  U fval = f_result[ 0 ];
  cout << "f(x,y)=" << fval << endl;
  for (int i = 0; i <= ORDER; i++)
  {
    U c = f_result[ i ];  // The i'th taylor coefficient
    cout << "(1/k!)*(d^" << i << "f/dx^" << i << ")=" << c <<
       endl;
  }
}
template <typename U, typename G>
void get_max_norm(T<U> &f_result1, T<G> &f_result2)
{
  // max-norm
  mpreal max_norm = 0;
  for (int i = 0; i <= ORDER; i++)
  {
    mpreal temp = 0;
    temp        = fabs(f_result1[ i ] - f_result2[ i ]);
    if (max_norm < temp)
    {
      max_norm = temp;
    }
  }
  cout << "max norm:\t" << max_norm << endl;
}
```

The equation to compute in this example is defined below:

$$f(x(t), y(t)) = \sin(x(t) + y(t)/3.2 - \cos(5.263))$$

in which all the variables are of `T<U>` type. The arithmetic operation functions will now "record" a directed acyclic graph (DAG) while computing the function value (which is the 0'th order Taylor-coefficient) [3].
The recorded DAG for this equation is

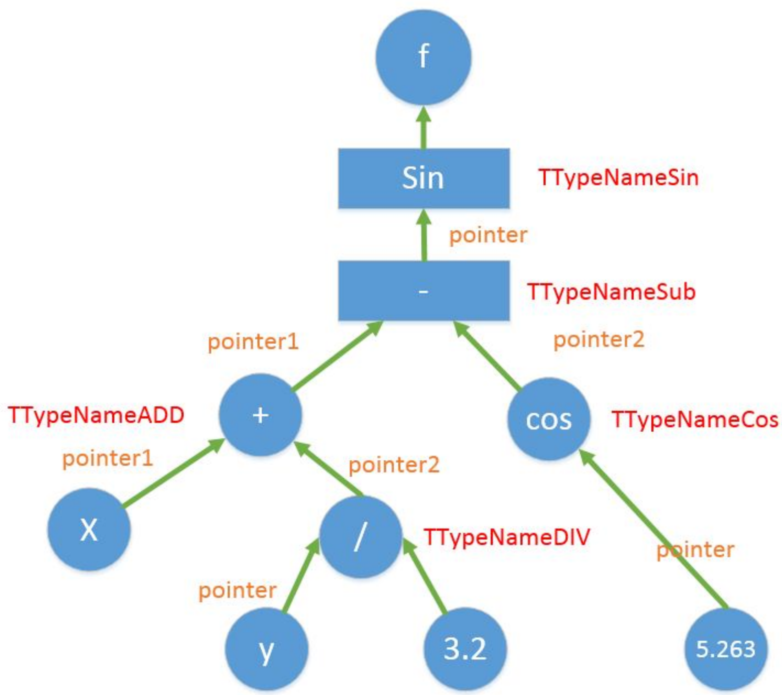

Figure 5.2: Directed Acyclic Graph in ExampleTAD1

Finally, we check the difference using the max norm between mpreal and double to verify the correctness of the modification. The output is

```
----------------------------------------------
Computed in double precision,
output in 15 digits
f(x,y)=0.961347321176936
(1/k!)*(d^0f/dx^0)=0.961347321176936
(1/k!)*(d^1f/dx^1)=0.566388646912616
(1/k!)*(d^2f/dx^2)=-1.64191826277923
(1/k!)*(d^3f/dx^3)=-3.21528673050749
(1/k!)*(d^4f/dx^4)=-1.08682624109917
(1/k!)*(d^5f/dx^5)=1.49621135884308
(1/k!)*(d^6f/dx^6)=2.12079159643671
(1/k!)*(d^7f/dx^7)=0.927780211679576
(1/k!)*(d^8f/dx^8)=-0.315745405477617
(1/k!)*(d^9f/dx^9)=-0.616890902418931
(1/k!)*(d^10f/dx^10)=-0.328477114662058
----------------------------------------------
Computed in MPFR precision 128 digs
output in 15 digits
f(x,y)=0.961347321176936
(1/k!)*(d^0f/dx^0)=0.961347321176936
(1/k!)*(d^1f/dx^1)=0.566388646912616
(1/k!)*(d^2f/dx^2)=-1.64191826277923
(1/k!)*(d^3f/dx^3)=-3.21528673050749
(1/k!)*(d^4f/dx^4)=-1.08682624109917
(1/k!)*(d^5f/dx^5)=1.49621135884308
(1/k!)*(d^6f/dx^6)=2.12079159643671
(1/k!)*(d^7f/dx^7)=0.927780211679576
(1/k!)*(d^8f/dx^8)=-0.315745405477617
(1/k!)*(d^9f/dx^9)=-0.616890902418931
(1/k!)*(d^10f/dx^10)=-0.328477114662058
----------------------------------------------
max_norm between double and MPFR precision 128
max norm:       4.8096e-16
```

5.3 Makefile

The overall makefile to make all the example executives is shown below:

```
CXX = g++
CXXFLAGS = -std=c++11 -I../include
LDFLAGS = -lmpfr -lgmp
EXEC = ExampleFAD2 ExampleTAD1 \
        ExampleTAD2
all: $(EXEC)
$(EXEC): % : %.o
        $(CXX) -o $@ $? $(LDFLAGS)
clean:
        rm -rf *.o *.txt $(EXEC)
result:
        rm ../output/*.txt
        @-echo ""
        @-echo "Generating output text to ../output"
        @-echo ""
        @-for i in $(EXEC); do \
                ./$$i >> ../output/$${i}_result.txt; \
        done
```

Bibliography

[1] *Automatic differentiation*. https://en.wikipedia.org/wiki/Automatic_differentiation. Url Updated: 2013-03-14.

[2] *GNU MPFR reference manual*. http://www.mpfr.org/mpfr-current/mpfr.pdf. Url Updated: September 2016.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, 3rd Edition*, The MIT Press, 2009.

[4] A. Griewank, *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation, Second Edition*, Society for Industrial and Applied Mathematic, September 26, 2008.

[5] P. Holoborodko, *MPFR C++*. http://www.holoborodko.com/pavel/mpfr/, 2008-2012.

[6] R. D. Neidinger, *Introduction to automatic differentiation and MATLAB object-oriented programming*, SIAM REVIEW, 52 (2010), pp. 545–563.

[7] C. B. Ole Stauning, *Flexible automatic differentiation using templates and operator overloading in C++*. http://www.fadbad.com/fadbad.html. Url Updated: 19-Apr-2012.

[8] S. Prata, *C++ Primer Plus (6th Edition)*, Addison-Wesley Professional, Oct. 18 2011.

[9] B. E. M. Stanley B. Lippman, Josée Lajoie, *C++ Primer (5th Edition)*, Addison-Wesley Professional, August 16, 2012.