Vectorisation and GPUs extensions of ROOT::Math routines

August 2015

Author:
Anca – Mihaela Popescu

Supervisors:
Lorenzo Moneta
Danilo Piparo

CERN openlab Summer Student Report 2015

Project Specification

This openlab summer project aims to provide a vectorised and accelerator/gpu ready implementation of the mathematical functions and statistical distributions offered by the ROOT::Math namespace together with a conspicuous number of tests stressing them with comparisons with their scalar counterparts and analytical values. The technologies involved in the work are autovectorisation capabilities offered by modern compilers, fast implementations of mathematical functions provided by the VDT mathematical library, explicit vectorisation via the VC library, OpenCL and, optionally, CUDA.

Abstract

The ROOT system provides a set of OO frameworks with all the functionality needed to handle and analyze large amounts of data in a very efficient way. Having the data defined as a set of objects, specialized storage methods are used to get direct access to the separate attributes of the selected objects, without having to touch the bulk of the data. Included are histograming methods in an arbitrary number of dimensions, curve fitting, function evaluation, minimization, graphics and visualization classes to allow the easy setup of an analysis system that can query and process the data interactively or in batch mode, as well as a general parallel processing framework, PROOF, that can considerably speed up an analysis.

In addition, ROOT offers an ensemble of advanced mathematical functions such as Bessel and Airy functions or distributions such as Landau, gammma, Cauchy or Breit-Wigner. These functions are relevant for a variety of performance critical applications, among which the statistical studies in HEP such as discoveries and exclusions. This kind of activities will be more and more important during the forthcoming 13 TeV collisions at the LHC.

Table of Contents

1 Introduction

ROOT is a framework for data processing, born at CERN, at the heart of the research on highenergy physics. Every day, thousands of physicists use ROOT applications to analyze their data or to perform simulations.

1.1 ROOT

The ROOT project was started in the context of the NA49 experiment at CERN. NA49 generates an impressive amount of data, about 10 Terabytes of raw data per run. This data rate is of the same order of magnitude as the rates expected to be recorded by the LHC experiments. Therefore, NA49 was the ideal environment to develop and test the next generation data analysis tools and to study the problems related to the organization and analysis of such large amounts of data.

ROOT provides a basic framework that offers a common set of features and tools for all domains of High Energy Physics computing. ROOT is an ideal environment to introduce physicists quickly to the world of Objects and C++. Thanks to the built-in CINT C++ interpreter the command language, the scripting, or macro, language and the programming language are all C++. The interpreter allows for fast prototypingof the macros since it removes the time consuming, compile/link cycle. It also provides a good
environment to learn C++. If more performance is needed the interactively developed macros can be compiled using a C++ compiler via a machine independent transparent compiler interface called ACliC.

The system has been designed in such a way that it can query its databases in parallel on clusters of workstations or many-core machines. ROOT is an open system that can be dynamically extended by linking external libraries. This makes ROOT a premier platform on which to build data acquisition, simulation and data analysis systems.
In 2002 ROOT became an official project within the Physics Department at CERN. Thanks to the manpower injected in the project by CERN and also by FNAL, ROOT could gradually be extended in many directions to become the cornerstone of most HEP software systems today, covering many areas of HEP computing, like analysis, simulation, reconstruction, event display and DAQ.

Besides in High Energy Physics ROOT is also widely used in many other scientific fields, like astronomy and biology but also in finance and medicine.

1.2 TFormula Class

- Class for evaluating mathematical functions provided as expression strings
- ROOT function class (TF1) derives from TFormula: uses TFormula constructs for making functions from string
- Replace old parser with the JIT provided by Cling, a real C++ interpreter

How Does it Work:
• TFormula creates a C/C++ functions which is passed to Cling
• The created function is now compiled on the fly using the JIT of Cling

Evaluation of TFormula:
• No need to have a dedicated parser to analyze and compile the code
• JIT compilation is done at initialization time, not when evaluating the expression
• The created function is evaluated using its function pointer, which can be retrieved via the ROOT interpreter interface

Since the main purpose of the project is to offer a gpu ready implementation of the mathematical functions from ROOT, the TFormula Class is not providing this feature, on account of its inability to compute vector types.

2 TFormulaLite Class Implementation

The implementation needed is realised, in the first instance, by an intermediate approach with a new class: TFormulaLite Class. The TFormulaLite Class has a similar style to process, compile and create mathematical functions as the TFormula Class: the new class processes the formula by replacing the parameters with the corresponding values, TFormulaClass creates a C/C++ function which is passed to Cling and the new function is compiled using the JIT.

The TFormulaLite Class is a templated class implemented with the purpose to compute any type of formula with any type of values (vectors, PODs & others), from one dimension to four dimensions, parameterized or not.

```
template<class T>
class TFormulaLite
{
  …..
};
```

Moreover, the class provides template specializations, accomplished through the realization of another templated class, which is inside the TFormulaLite.

```
template<class T>
class TFormulaLite
{
      ......
      template <class T2>
      class Evaluator
      {
            ......
      };
};
```

How to use TFormulaLite Class:

- When creating the formula, the user has to specify the type of the calculation result wanted, the name of the function/formula and the formula

```
TFormulaLite<double> f1("f1", "x/2+10*x");

TFormulaLite<double> h1("h1", "p[0]*sin(x)+p[1]*x*x*x");
```

- When calculating the formula with a certain value, the user calls the Eval() method to compute any type of variable.

```
f1.Eval(30.);
```

```
std::vector<double> p={1.,10};
h1.Eval(30.,p);
```

- In case of several types, which are the most used types of variables in mathematical functions, there are several methods implemented to obtain a straight-forward evaluation. These cases are represented by:
  - double variables
  - float variables
  - Vc::double_v vectors
  - Vc::float_v vectors
  - std::vector<double>
  - std::vector<float>

Each of these methods, as well as the generic Eval() method, can be used with none or more parameters of double or float types. The Eval() method can also be used with one or more variables.

```
static T DoEvalVar(const T *x, const Int_t& nvar,
TFormulaLite<T3>& fct)
      {
            ......
            auto f = (T (*)(const T*))fct.fcnptr;
            return f(x);
      }
```

TFormulaLite Class implementation consists of:

- 2 constructors – one compound of the name and the formula and one compound of the name, the formula and the number of parameters in the formula
- Getters&Setters methods for the double and float parameters and for the name of the function
- Methods for obtaining the type of the value passed in the Eval() method
- Methods for processing the formula – replacing the parameters with the specific values a
- methods for pre-processing the formula – passing to Cling only a number of formulas as TString equal to the dimension of the variable in order to have vectorised functions in the case of std::vectors
- the generic Eval() method and the related methods for the popular types
- 1 method for obtaining the function pointer of the formula
- 2 templated classes inside: one for the specializations of the Eval() method and one for the specializations of the Eval() method in the case of multiple variables

3 Eval() Implementation

The Eval() method represents a method implemented with the purpose of computing any type of variable (ex.: double, float, int, unsigned long int etc.), more exactly the vector types. This method is actually the main difference between the TFormulaLite Class and the TFormula Class.
The algorithm of the Eval() method is:
- identify the type of the value passed by the user
- identify the name of the function created by the user
- inject in the gInterpreter (Cling) the name of the function and the type of the variable, obtaining the function pointer
- cast the function pointer with a T type
- return the result of the function computed with the value passed by the user

```
static T DoEval(const T& x, TFormulaLite<T> &fct)
{
        …
        auto f = (T (*)(const T&))fct.fcnptr;
        return f(x);
}

static T DoEval(const T& x, const std::vector<float> p,
TFormulaLite<T>& fct)
{
      ……
      auto f=(T (*)(const T&,  const std::vector<double>&))
      fct.fcnptr;
      return f(x,p);
}
```

In order to obtain a more efficient and an optimized way of calculation, the function pointer is determined only once, namely the name of the function and the type of the value is passed only once to Cling. Once the function is injected in the gInterpreter, the method will use the same function pointer for more computations.

```
static T DoEval(const T& x, TFormulaLite<T> &fct)
{
      if ( !fInjected )
            fcnptr = InitializationEvalVar(x, fInjected);
       ……
}

void* GetSpecialisedFcnPtr(const TString& funcName, const
TString& typeName)
{
      const TString textFunction = funcName+"<"+typeName+">";
      fInjected = true;
      return (void*)gInterpreter->ProcessLine(textFunction);
}
```

Another improvement is represented by the implementation of the PreProcessFormula() method, which passes to the gInterpreter only a number of formulas, as a TString, equal to the dimension of the variable.
For example, for one dimension variable, it passes only one TString formula templated.

```
     fForm = "template<class T> T ";
     fForm += GetName();
     fForm += "(const T& x) { return ";
     fForm += fFormula;
     fForm += ";} ";
     gInterpreter->ProcessLine(fForm);
```

In the case of a std::vector<double>, the formula passed to Cling interpreter is vectorised directly when is sent to the gInterpreter.

```
void PreProcessFormula(const T& x, Int_t dimension)
{
     ……
     fForm = "template<class T> T ";
     fForm += GetName();
     fForm += "(const T& x) { T result = x; ";
     fForm += "for(Int_t i = 0; i < ";
     fForm += dimension;
     fForm += "; ++i) { result[i] = " ;
     fForm += fFormula;
     fForm += ";} ";
     fForm += "return result;}";
     gInterpreter->ProcessLine(fForm);
}
```

The template specializations made for the several types are implemented outside the main class:

- double

```
template <>
template <class T2> class TFormulaLite<double>::Evaluator { … };
```

- float

```
template <>
template <class T2> class TFormulaLite<float>::Evaluator { … };
```

- Vc::double_v

```
template <>
template <class T2> class TFormulaLite<Vc::double_v>::Evaluator {
… };
```

- Vc::float_v

```
template <>
template <class T2> class TFormulaLite<Vc::double_v>::Evaluator {
… };
```

- std::vector<double>

```
template <>
template<class T2>class TFormulaLite<std::vector<double>>::
                                                      Evaluator{…};
```

- Std::vector<float>

```
template <>
template<class T2>class TFormulaLite<std::vector<float>>::
                                                      Evaluator{…};
```

Furthermore, in these template specializations the type of the variable is not determined through the `typeid()` operator, but with another template specializations, implemented outside the TFormulaLite Class, including the generic method of determing the variable type.

```
template <class T>
class Initializer
{
	T value;
	TString type;
public:
	Initializer (T arg) { value = arg; }
	TString GetType()
	{
		TString typeName = TString(typeid(value));
		return typeName;
	}
};
template <>
class Initializer <double>
{
	double value;
public:
	Initializer (double arg) { value = arg; }
	TString GetTypeDouble()
	{
		return "double";
	}
};
template <>
class Initializer <float>
{
	float value;
public:
	Initializer (float arg) { value = arg; }
	TString GetTypeFloat()
	{
		return "float";
	}
};
```

```
template <>
class Initializer <Vc::double_v>
{
      Vc::double_v value;
public:
      Initializer (Vc::double_v arg) { value = arg; }
      TString GetTypeVDouble()
      {
            return "Vc::double_v";
      }
};
template <>
class Initializer <Vc::float_v>
{
      Vc::float_v value;
public:
      Initializer (Vc::float_v arg) { value = arg; }
      TString GetTypeVFloat()
      {
            return "Vc::float_v";
      }
};
template <>
class Initializer <std::vector<double>>
{
      std::vector<double> value;
public:
      Initializer (std::vector<double> arg) { value = arg; }
      TString GetTypeVD()
      {
            return "std::vector<double>";
      }
};
template <>
class Initializer <std::vector<float>>
{
      std::vector<float> value;
public:
      Initializer (std::vector<float> arg) { value = arg; }
      TString GetTypeVF()
      {
            return "std::vector<float>";
      }
};
```

4 Other optimisations

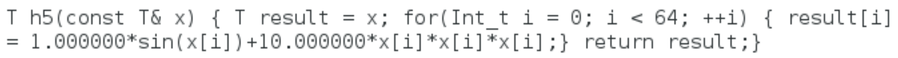
The main optimisations added to the TFormulaLite Class are represented by introducing the directly methods of computation for 6 types and by passing the TString formula to the gInterpreter (Cling) as a vectorised function in the case of the std::vectors.

The TFormulaLite Class has 6 template specializations included for the most used types in mathematical computations:

- Double
- Float
- Vc::double_v
- Vc::float_v
- std::vector<double>
- std::vector<float>

Using these template specializations by calling the Eval() method, the type of the value is determined faster in the mathematical functios implemented.

Furthermore, the vectorised formula passed to the gInterpreter is an useful way of creating C++ functions and compiling them with JIT of Cling.



```
T h5(const T& x) { T result = x; for(Int_t i = 0; i < 64; ++i) { result[i]
= 1.000000*sin(x[i])+10.000000*x[i]*x[i]*x[i];} return result;}
```

In the case of a multidimensional function, the variables are stored in a vector in the case of one dimension variables or a matrix in the case of std::vectors or Vc vectors as variables.

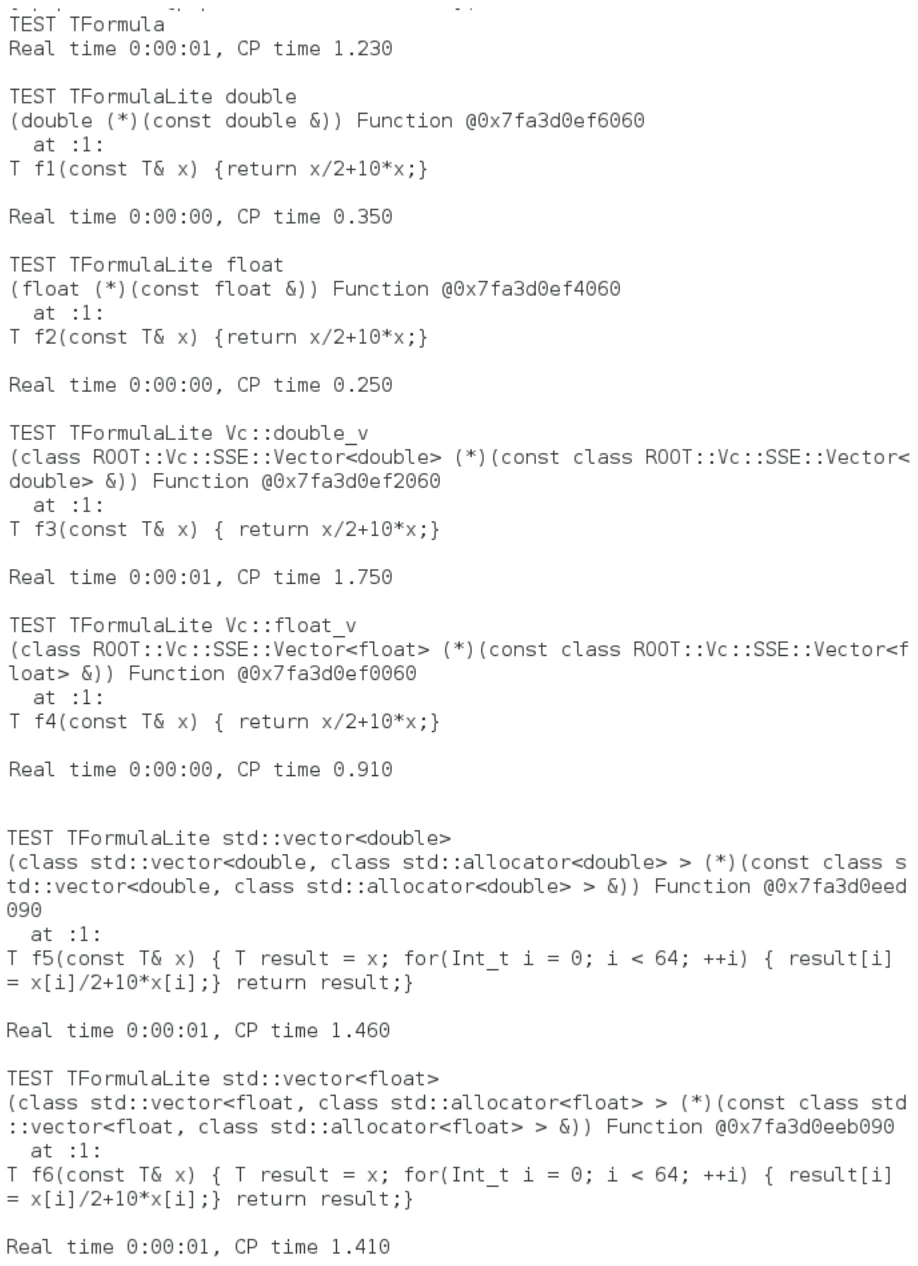
```
fForm = "template<class T> T ";
fForm += GetName();
fForm += "(const T *x) {return " + fFormula + ";}";
```

```
fForm = "template<class T> T ";
fForm += GetName();
fForm += "(const T *x) { T result = *x; ";
fForm += "for(Int_t i = 0; i < ";
fForm += dimension;
fForm += "; ++i) { result[i] = " ;
fForm += fFormula;
fForm += ";} ";
fForm += "return result;}";
```

5 Tests

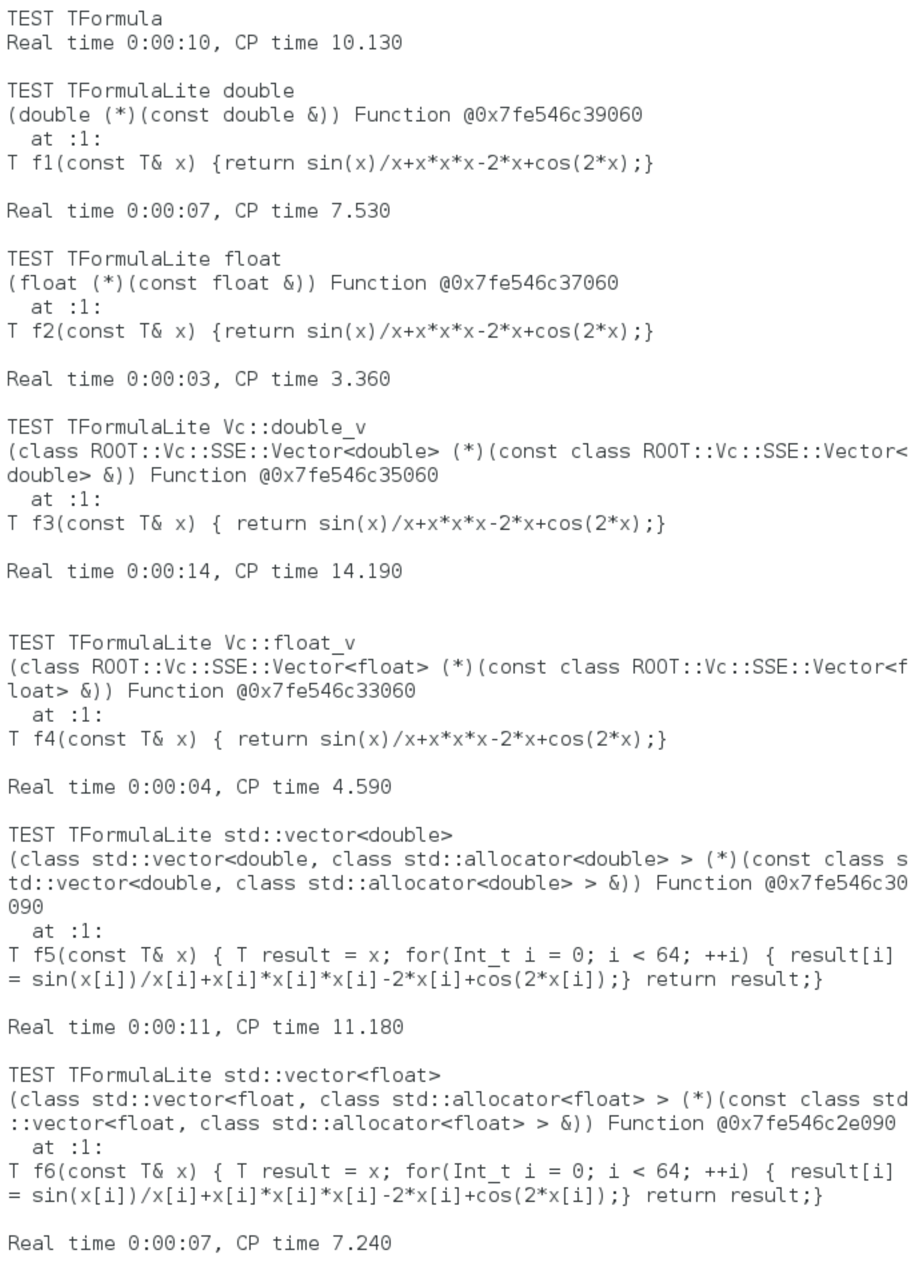
To analyze the performance of the intermediate implementation, the class was tested with simple and complicated functions by computing it with the Eval() method in a large amount of calls. The executable file uses the Vc library, the TStopWatch library and the TSystem library with the purpose to obtain the CPU time passed by evaluating the formula created.

The simple tests made do not represent a justificative evidence of the new implemented class proficiency, because these tests are made of simple formulas which are computed very fast (for example, the adding operation is a very fast and cheap one). Due to the CPU power, the time results usually are null, because of the flunctuations and the high speed of computation.

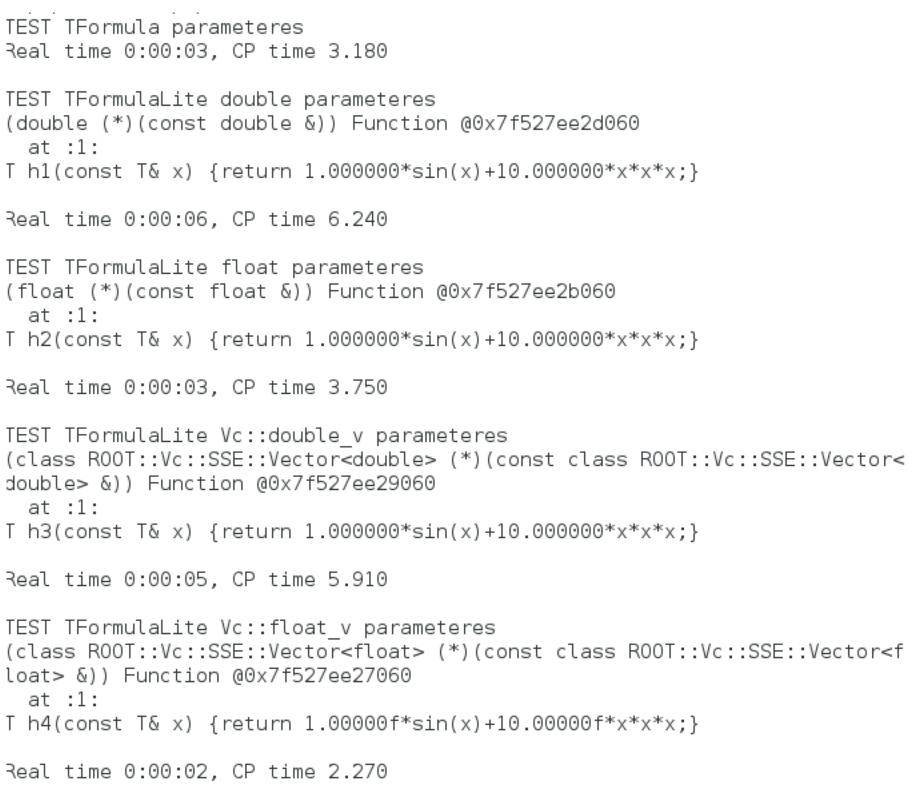


```
TEST TFormula
Real time 0:00:01, CP time 1.230

TEST TFormulaLite double
(double (*)(const double &)) Function @0x7fa3d0ef6060
  at :1:
T f1(const T& x) {return x/2+10*x;}

Real time 0:00:00, CP time 0.350

TEST TFormulaLite float
(float (*)(const float &)) Function @0x7fa3d0ef4060
  at :1:
T f2(const T& x) {return x/2+10*x;}

Real time 0:00:00, CP time 0.250

TEST TFormulaLite Vc::double_v
(class ROOT::Vc::SSE::Vector<double> (*)(const class ROOT::Vc::SSE::Vector<
double> &)) Function @0x7fa3d0ef2060
  at :1:
T f3(const T& x) { return x/2+10*x;}

Real time 0:00:01, CP time 1.750

TEST TFormulaLite Vc::float_v
(class ROOT::Vc::SSE::Vector<float> (*)(const class ROOT::Vc::SSE::Vector<f
loat> &)) Function @0x7fa3d0ef0060
  at :1:
T f4(const T& x) { return x/2+10*x;}

Real time 0:00:00, CP time 0.910


TEST TFormulaLite std::vector<double>
(class std::vector<double, class std::allocator<double> > (*)(const class s
td::vector<double, class std::allocator<double> > &)) Function @0x7fa3d0eed
090
  at :1:
T f5(const T& x) { T result = x; for(Int_t i = 0; i < 64; ++i) { result[i]
= x[i]/2+10*x[i];} return result;}

Real time 0:00:01, CP time 1.460

TEST TFormulaLite std::vector<float>
(class std::vector<float, class std::allocator<float> > (*)(const class std
::vector<float, class std::allocator<float> > &)) Function @0x7fa3d0eeb090
  at :1:
T f6(const T& x) { T result = x; for(Int_t i = 0; i < 64; ++i) { result[i]
= x[i]/2+10*x[i];} return result;}

Real time 0:00:01, CP time 1.410
```

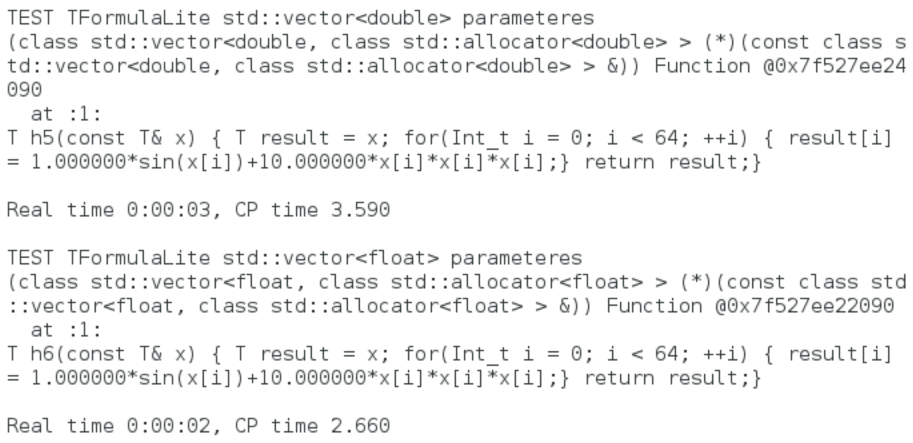
In order to have a relevant comparison, some complicated formulas were provided.



```
TEST TFormula
Real time 0:00:10, CP time 10.130

TEST TFormulaLite double
(double (*)(const double &)) Function @0x7fe546c39060
  at :1:
T f1(const T& x) {return sin(x)/x+x*x*x-2*x+cos(2*x);}

Real time 0:00:07, CP time 7.530

TEST TFormulaLite float
(float (*)(const float &)) Function @0x7fe546c37060
  at :1:
T f2(const T& x) {return sin(x)/x+x*x*x-2*x+cos(2*x);}

Real time 0:00:03, CP time 3.360

TEST TFormulaLite Vc::double_v
(class ROOT::Vc::SSE::Vector<double> (*)(const class ROOT::Vc::SSE::Vector<
double> &)) Function @0x7fe546c35060
  at :1:
T f3(const T& x) { return sin(x)/x+x*x*x-2*x+cos(2*x);}

Real time 0:00:14, CP time 14.190


TEST TFormulaLite Vc::float_v
(class ROOT::Vc::SSE::Vector<float> (*)(const class ROOT::Vc::SSE::Vector<f
loat> &)) Function @0x7fe546c33060
  at :1:
T f4(const T& x) { return sin(x)/x+x*x*x-2*x+cos(2*x);}

Real time 0:00:04, CP time 4.590

TEST TFormulaLite std::vector<double>
(class std::vector<double, class std::allocator<double> > (*)(const class s
td::vector<double, class std::allocator<double> > &)) Function @0x7fe546c30
090
  at :1:
T f5(const T& x) { T result = x; for(Int_t i = 0; i < 64; ++i) { result[i]
= sin(x[i])/x[i]+x[i]*x[i]*x[i]-2*x[i]+cos(2*x[i]);} return result;}

Real time 0:00:11, CP time 11.180

TEST TFormulaLite std::vector<float>
(class std::vector<float, class std::allocator<float> > (*)(const class std
::vector<float, class std::allocator<float> > &)) Function @0x7fe546c2e090
  at :1:
T f6(const T& x) { T result = x; for(Int_t i = 0; i < 64; ++i) { result[i]
= sin(x[i])/x[i]+x[i]*x[i]*x[i]-2*x[i]+cos(2*x[i]);} return result;}

Real time 0:00:07, CP time 7.240
```



```
TEST TFormula parameteres
Real time 0:00:03, CP time 3.180

TEST TFormulaLite double parameteres
(double (*)(const double &)) Function @0x7f527ee2d060
  at :1:
T h1(const T& x) {return 1.000000*sin(x)+10.000000*x*x*x;}

Real time 0:00:06, CP time 6.240

TEST TFormulaLite float parameteres
(float (*)(const float &)) Function @0x7f527ee2b060
  at :1:
T h2(const T& x) {return 1.000000*sin(x)+10.000000*x*x*x;}

Real time 0:00:03, CP time 3.750

TEST TFormulaLite Vc::double_v parameteres
(class ROOT::Vc::SSE::Vector<double> (*)(const class ROOT::Vc::SSE::Vector<
double> &)) Function @0x7f527ee29060
  at :1:
T h3(const T& x) {return 1.000000*sin(x)+10.000000*x*x*x;}

Real time 0:00:05, CP time 5.910

TEST TFormulaLite Vc::float_v parameteres
(class ROOT::Vc::SSE::Vector<float> (*)(const class ROOT::Vc::SSE::Vector<f
loat> &)) Function @0x7f527ee27060
  at :1:
T h4(const T& x) {return 1.00000f*sin(x)+10.00000f*x*x*x;}

Real time 0:00:02, CP time 2.270
```



```
TEST TFormulaLite std::vector<double> parameteres
(class std::vector<double, class std::allocator<double> > (*)(const class s
td::vector<double, class std::allocator<double> > &)) Function @0x7f527ee24
090
  at :1:
T h5(const T& x) { T result = x; for(Int_t i = 0; i < 64; ++i) { result[i]
= 1.000000*sin(x[i])+10.000000*x[i]*x[i]*x[i];} return result;}

Real time 0:00:03, CP time 3.590

TEST TFormulaLite std::vector<float> parameteres
(class std::vector<float, class std::allocator<float> > (*)(const class std
::vector<float, class std::allocator<float> > &)) Function @0x7f527ee22090
  at :1:
T h6(const T& x) { T result = x; for(Int_t i = 0; i < 64; ++i) { result[i]
= 1.000000*sin(x[i])+10.000000*x[i]*x[i]*x[i];} return result;}

Real time 0:00:02, CP time 2.660
```

The results obtained could be analized and compared with the aid of the CPU time outcome. In all the cases, the float speciaizations offer a cheaper evaluation in terms of time, because of the loss of the precision in the computation.

6 Integration in TFormula Class

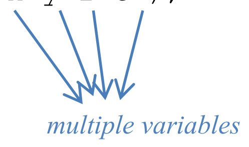
In order to use the new way of evaluation C++ functions/formulas, the properly functions from the TFormulaLite Class are implemented in the TFormula Class, implementation that transforms the TFormula Class also in a templated one with all the characteristics from the previous class. Most of the TFormula class methods were shifted in the header file (TFormula.h), because of the templated characteristics which implied the specification of a T type.
The modification from a simple class to a templated one involved several changings in different files of ROOT, such as: TF1 class, TLinearFitter class, MethodFDA class, TActivationSignoid class, TActivationRadial class, AnalyticalIntegrals class, WrappedTF1 class.
How to use the new TFormula Class:

- *constructing*
  - 1D without parameters

```
TFormula<double> f("f", "x*x+x+10.5");
```

  - 2D/3D/4D without parameters

```
TFormula<double> f("f", "x+y+z+t");
```

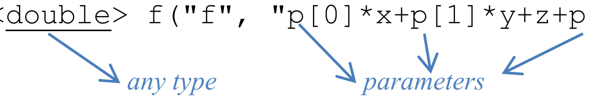


  - 1D with parameters

```
TFormula<double> f("f", "p[0]*x*x+p[1]*x+p[2]*10.5");
```

  - 2D/3D/4D without parameters

```
TFormula<double> f("f", "p[0]*x+p[1]*y+z+p[2]*t");
```



- *evaluation*
  - 1D without parameters

```
f.Eval(value);
```

  - 2D/3D/4D without parameters

```
f.Eval(varVector);
```

  - 1D with parameters

```
f.Eval(value,p);
```

  - 2D/3D/4D without parameters

```
f.Eval(varVector,p);
```

value = any value of the type passed in the constructor

varVector = vector/matrix that stores the variables (PODs, std::vectors or Vc vectors)

p = vector that stores the parameters

7 Conclusions

To sum up, the intermediate solution TFormulaLite Class provides some extra benefits than the basic TFormula Class implementation: the new evaluating method valid also for vector types and for the Vc library. The implementation created with this feature is applicable from one dimension to four dimensions and as well in the case of a parametrized formula.

The Cling interpreter gets through the TFormulaLite Class a C++ function created from the function provided by the user, compiling it at initialization time and in the instance of a vector variable receiving it as a vectorised formula.

The deficient characteristic of this new class is represented by the fact of the necessity of specifying the type of the result wanted at constructing time.

```
TFormulaLite<double> f1("f1", "x/2+10*x");
```

8 Future Work

The implementation of the TFormulaLite Class is a project which can be developed and improved by optimizing the C++ code or by using the last extensions of the compilers.

Some planned developments on this project would be:

- analysis of the Vc results

- new version in which the user does not need to specify the type of the result

- eliminating several bugs in the Vc vectors evaluation

9 References

[1] ROOT Users Guide
https://root.cern.ch/drupal/content/users-guide#UG

[2] ROOT Reference Guide
https://root.cern.ch/drupal/content/reference-guide

[3] ROOT Classes - TFormula Class
https://root.cern.ch/root/html/TFormula.html

[4] *The Pragmatic Programmer: From Journeyman to Master* – Andrew Hunt, David Thoma

[5] VC library