Google Summer of Code 2017

Improvements in Vectorization and Parallelization of ROOT Math Libraries

Alejandro García Montoro

Table of Contents

Abstract

This document describes the proposal that the author submits for the project *Improvements in vectorization and parallelization of ROOT Math libraries*, from CERN-HSF organization.
This project aims to improve the efficiency of the mathematical functions and fitting methods of ROOT, a software framework developed mainly by CERN, by means of parallelization and vectorization of the corresponding code. Some work in this direction has already been done, but there is still work to do as outlined by the task ideas: 1) Completion of parallelization and vectorization of all the fitting methods available in ROOT, 2) Adapt gradient function interfaces for thread-based parallelization and vectorization, 3) Vectorization of `TFormula` and "predefined ROOT functions" and 4) Vectorization of most used mathematical and statistical functions in `ROOT::Math` and `TMath`.
The first section of this document describes the proposal, studying first, in Starting Point subsection, the current state of the code in order to understand the work that has to be done. In the Coding Plan and Methods subsection, the main tasks are detailed, along with the optional ones, outlining what needs to be done in order to implement the features proposed by the project: specific files, classes and functions that will have to be modified or created are discussed. All this work is then structured in the Timeline section, defining subtasks in a per-week basis. Possible Schedule Conflicts are finally discussed. Subsection Management of Coding Project details the methods of work (as the contact with the mentors or the frequency of commits and pull requests) that will be followed in the coding periods. Finally, Test subsection explains the test implemented by the author to get used to the codebase and as a way for the mentors to know the author skills.
The last section, Student Information, details information about the author's biography, how to contact him and his affiliation.

Description of the Proposal

The project *Improvements in vectorization and parallelization of ROOT Math libraries*, proposed by CERN-HSF organization and mentored by Xavier Valls Pla and Lorenzo Moneta, aims to vectorize and parallelize the mathematical function interfaces as well as the fitting functions of ROOT.
In this section I will study the work that has already been done in this direction, describe the intended work that I plan to do in summer in the case I am selected and outline a timeline with a set of deliverables scheduled.

Starting Point

The code implemented in summer will continue the work currently located in the GSoC branch of Xavier Valls ROOT fork; this branch should be merged in the main ROOT repo by the start of GSoC, and in fact some of the commits are already in ROOT master. A summary of the work that has been done in the branch, and that could serve as an example for the code that has to be implemented in summer, follows:

1. `TF1` interface has been adapted to implement vectorized evaluations (see commit 56a31ba, merged in ROOT in 5c4caed). `WrappedMultiTF1` has been generalised to ease the implementation of different vectorization backends and types (see commit 116ffbe, merged in ROOT in 398101b).
2. `Vc` backend has already been linked to `Mathcore` (9cec295) and a new type describing a vectorized double, `Double_v`, has been defined (9daf281).
3. Some interfaces of `Fitter` have been adapted to add vectorization (cee8a7f). The classes to manage (un)binned data have been refactored and adapted (54243ee).
4. Chi2 fitting has been vectorized and parallelized (fbc96fa, 5bb680a and d121e3a; in ec9911a Chi2 has been moved into Evaluate<T>, that will hold other evaluations, as the `LogLikelihood` one).
5. Unbinned likelihood fit has been vectorized and parallelized (fac6143).
6. `FitUtil` has been adapted to allow parallelization (1ab4fa3 and 53d57cf, with needed previous work on 86234a4 and specially e836b84).

1, 2 and 3, along with 6, code the basics needed for the actual implementation of the vectorization and parallelization of ROOT. It is important to note in 2 that the linking of `Vc` backend in `Mathcore` will be eventually generalized: `VecCore` should abstract the backend used, so ROOT should not care about which one is being used.

4 and 5 are really good examples of the kind of work that will be done in summer. We see that in order to vectorize and parallelize existing methods we will usually need to generalize them, templating their definitions and adapting the existing code to use the APIs exposed by vectorization and parallelization libraries used (vectorized types, functions and masks, reduction maps…).

This proposal takes the state of this branch as the starting point to accomplish the proposed tasks.

Coding Plan and Methods

As a rule, the general coding plan for all the tasks will follow a simple workflow based on test driven development:

1. Study of the task goal to obtain a detailed description of the work needed.
2. Design of the interfaces for the new (or adapted) code.
3. Implementation of the tests that will check that the code works as expected.
4. Implementation of the benchmarks that will check that the code improves the efficiency with respect to the old one.
5. Actual implementation of the code.
6. Testing.

Obviously, steps 5 and 6 will need some iteration to fix possible bugs and assure everything works as expected. It is possible that even some of the tests in steps 3 and 4 have to be redefined once the actual implementation has started.
The documentation of all the code will be a constant through all the method and will occur in parallel in every step. Doxygen documentation will be used, following the conventions outlined by the ROOT guidelines. The code implemented will always follow the ROOT coding conventions.

Then, all deliverables will be composed of three items:

1. The code that implements the new functionality.
2. The test that checks that every new or modified function works.
3. The benchmark that checks that the efficiency is improved.

The code implementing the functionality will be submitted to the main ROOT repository through a pull request, whereas the tests and benchmarks will be submitted to the roottest repository with a parallel pull request. It is important to note that ROOT has a continuous integration system that will assure every pull request passes the tests from roottest and that it follows the coding style. The test driven development will assure every chunk of code that is added to the repo will be tested and, if something fails, we will be notified immediately.

The main tasks enumerated in the project description have their own needs, so the following subsections are dedicated to detail the issues of each one of them, as far as I know to this day.

High-Priority Tasks

Completion of Parallelization and Vectorization of all the Fitting Methods Available in ROOT

To this day, only Binned Chi2 and Unbinned Likelihood fitting methods are adapted (see `Evaluate<T>::EvalChi2` and `Evaluate<T>::EvalLogL` in `FitUtil` namespace, in the GSoC branch of Xavier's fork).

The main task here is to vectorize and parallelize the rest of the methods; i.e., to adapt the classes `Chi2FCN`, `LogLikelihoodFCN` and `PoissonLikelihoodFCN`.

The recent changes on `FitUtil.h` and `{Un,}BinData.{h,cxx}` will be useful here.

Adapt Gradient Function Interfaces for Thread-Based Parallelization and Vectorization

The gradient function interfaces need to be generalized in order to be able to implement parallelization and vectorization. This task is probably the largest one, as there are several classes that need to be templated and the changes to their children classes should be taken with care.

To this day, the class `IParametricGradFunctionMultiDim` has been templated to `IParametricGradFunctionMultiDimTempl<T>` (see 116ffbe), change that is based on the generalization of the class `IParamMultiFunctionMultiDim` in the class `IParametricFunctionMultiDimTempl<T>`. `WrappedMultiTF1Templ` inherits from `IParametricGradFunctionMultiDimTempl<T>`, so it should be studied before starting the implementation.

The fitting methods depend also on these classes, particularly on the `DoParameterDerivative` function, which is a virtual function that has to be implemented by the classes inheriting from them.

Particular care should be given to the methods `EvaluatePoissonLogLGradient`, `EvaluateChi2Gradient` and `EvaluateLogLGradient` in `FitUtil` namespace.

Vectorization of TFormula and "Predefined ROOT Functions"

`TFormula` receives a `const char*` string from the user, parses it to create a formula and eventually evaluates it. The goal of this task is to vectorize this class, which will automatically vectorize every user-defined function.

In order to accomplish this task, the parsing of the parameters should be adapted, their types should be templated to generalize the backend vectorized type and the predefined functions (`gaus`, `landau`, `expo`, `crystalball`, `breitwigner`, `cheb[0-9]` and `bigaus`) should be vectorized. This has to be previously studied, as it is possible that the

vectorization of `TFormula` gives us *for free* the vectorization of some of these predefined functions.

Vectorization of Most Used Mathematical and Statistical Functions in ROOT::Math and TMath

In order to actually improve the efficiency of the final user-defined code, the vectorization of ROOT should also cover the most usual functions in `Math` and `TMath`.

This task should be straightforward, and the only difficulty may reside in the templating of the functions to cover both the vectorized and linear cases. In fact, some of the basic functions are already implemented in `VecCore`, and the work would reduce to just call those functions; see, e.g., `Abs`, `Exp`, `SinCos`, `Log`, `Sqrt`... in `VecMath.h` file in `VecCore` repository.

In fact, this task has already been started as, as a test, I have already vectorized, tested and benchmarked `TMath::Gaus` (see section Test).

Optional Tasks

There are some libraries in ROOT that implement specific functions used in several mathematical methods. If time allows it, these functions should also be vectorized/parallelized.

Special Mathematical Functions

ROOT has a library of special functions, enumerated in the documentation and implemented in `SpecFuncMath{C,M}ore.{h,cxx}`. The complexity of this task is similar to the vectorization of `TMath` usual functions, and the majority of these functions are implemented using the functions from the GNU Scientific Library, so its documentation will be also useful.

Statistical Functions

ROOT has a library of statistical functions, enumerated in the documentation, with modules for Probability density functions, Cumulative distribution functions, Statistical functions for the truncated distributions and Quantile functions. These methods usually depend on the special mathematical functions, so this task depends on the previous one.

Timeline

Since 15th May I will be 100% available to work on GSoC. During the coding period (from 30th May to 29th August) I will be available to work as in a full-time job; i.e., 40 hours a week.

The scheduled actions for each week of the programme are outlined in the following subsections in a very conservative way. There are some slots in between the coding periods; these gaps will give me time to do the evaluations, could be used as extra days to finish delayed work or, hopefully and more probable, can be used as days to start the next tasks ahead of time.

Community Bonding Period (1st May - 29th May)

1st week (1-7): Get used to ROOT as a regular user, read and eventually participate in the ROOT forum, know the community, investigate cool stuff done with ROOT at CERN :)
2nd week (8-14): Iteratively fine-tune this timeline with the help of the mentors.
3rd week (15-21): Study well-known vectorization techniques, understand `VecCore` library internals.
4th week (22-28): Study the integration of `VecCore` library with ROOT. Get used to fitting methods and the gradient interfaces.

1st Coding Period (30th May - 26th June)

1st week (30-2): Study of `Math` and `TMath` usual functions to decide the methods that have to be vectorized. Code the tests that the new methods should pass.
2nd week (5-9): Implementation of the vectorization of the `Math` and `TMath` methods.
3rd week (12-16): Finish the testing and benchmarking and submit a pull request.
4th week (19-23): Study Fitting methods and gradient interfaces and define which of them should be vectorized/parallelized[1]. Start defining the tests that the code should pass.

2nd Coding Period (30th June - 24th July)

1st week (30): Finish tests and start coding.
2nd week (3-7): Finish coding, make sure the code passes the test and submit a pull request with all the vectorized Fitting methods.
3rd week (10-14): Study needed work for adapting the gradient function interfaces. This is probably the most difficult part, so it should be done with care[2]. Define the tests needed to check the gradient adaptation has been implemented correctly.
4th week (17-21): Start coding, probably templating first the corresponding code and making sure everything come together fine. Depending on the study of the previous week, a first pull request can be submitted.

3rd Coding Period (28th July - 29th August)

1st week (30-4): Finish coding and start testing and benchmarking.
2nd week (7-11): Finish the testing and submit a pull request with the final gradient code.
3rd week (14-18): Define the tests to vectorize evaluation of `TFormula` and start implementing the vectorization.
4th week (21-25): Finish the code, make sure it passes the tests and submit a pull request.
Last two days (28-29): Proofread all the documentation (it should have been written every time a function was modified or added). Submit a pull request with the possible typos.

---

[1] Some of the fitting methods depend on the gradient interfaces, so both tasks could run in parallel sometimes.

[2] Some information would be available from the previous task, as the interdependency between them is strong. It is highly possible that some of the work is already done from the previous task.

Schedule Conflicts

I have no schedule conflicts *a priori,* since I will not be studying nor working for any other project. I would probably spend some of my free time searching for a job to start when GSoC finishes, so the only conflict that could arise would happen if I have to attend a job interview. In such a case, I could exceptionally work more hours the previous or following days to make up the time spent in the interview.

However, unexpected schedule conflicts could cause a delay on some deliverables. In this case, I will get in contact immediately with the mentors to explain the situation and discuss a solution with which both parts agree.

In any case, the proposed timeline is very conservative, and there are some *spare* days in between the coding periods that can be used in these situations. The priority for these days is:

1. Submit the evaluations (this should not take more than half a day).
2. Finish possible unfinished work from the previous days.
3. Get a head start on the work of the following days.

Management of Coding Project

As a basic rule, I plan to be in contact with the mentors on a daily basis, so every change on the timeline, the deliverables expected, or any other problem will be solved as soon as possible.

I am used to commit quite a lot, modifying things and fixing the possible errors in a fast iteration loop, with one or more commit per day. The pull requests should be submitted with every new implemented feature (the timeline shows the most important ones), or in some cases when a bunch of functions have been adapted (as in the case of `TMath`, where it makes no sense to submit a pull request per function). I can adapt to the frequency of commits and pull requests preferred by the mentors, and my personal commits (probably one or more per day) can be squashed in one or two commits per pull request if the main developers prefer it that way.

Concerning the repositories, forks, branches and general project management, I am open to do it in the easiest way for the main developers to merge the code when it is ready.

Test

I have been in constant contact with Xavier Valls from 24th March until the application deadline, exchanging emails almost every day discussing the project information and this proposal. In these emails I was asked to code one or more of some proposed tests. The test I finally chose consisted in the vectorization of `TMath::Gaus` function; the patch sent to the mentors with the code implementing this new function can be see in this gist.

The first thing I tried when coding the test was to add the new function, `TMath::Gaus_v`, to the common header, `TMath.h`. However, when including `Math/Math_vectypes.hxx` in `TMath.h`, a bug appeared: the compiler complained about some undefined references of `Vc`. I explained the problem to Xavier, and the situation ended in fixes decoupling `TMath.h` from `Core` in ROOT/master (see the commit f32175e and this open pull request).

In the meantime, I added the declaration of the new function to a different file, `math/mathcore/inc/TMath_VecTest.h`, to avoid the problem, whose solution was outside the scope of the test. The new file looks like this:

```
#include "Math/Math_vectypes.hxx"

namespace TMath{
       Double_v Gaus_v(Double_v x, Double_t mean=0, Double_t sigma=1,
                       Bool_t norm=kFALSE);
}
```

This function is similar to `TMath::Gaus`, except for the `x` parameter, that is now a `Double_v`.

The new function was implemented in `math/mathcore/src/TMath.cxx`, based on the implementation of the linear version and using the API provided by `VecCore`, as can be seen in the use of the masks or of the vectorized functions such as `Exp`:

```
Double_v TMath::Gaus_v(Double_v x, Double_t mean, Double_t sigma, Bool_t norm)
{
   if (sigma == 0)
        return 1.e30;

   Double_v arg = (x-mean)/sigma;

   // for |arg| > 39  result is zero in double precision
   vecCore::Mask_v<Double_v> mask = !(arg < -39.0 || arg > 39.0);

   // Initialize the result to 0.0
   Double_v res(0.0);

   // Compute the function only when the arg meets the criteria,
   // using the mask computed before
   vecCore::MaskedAssign<Double_v>(res, mask,
                                   vecCore::math::Exp<Double_v>(-0.5 * arg * arg));
```

```
   if (!norm)
       return res;

   return res/(2.50662827463100024*sigma); //sqrt(2*Pi)=2.50662827463100024
}
```

In order to test that the new function works as the old one with an improved efficiency, a benchmark integrated with a simple test has been implemented in the file `math/mathcore/test/vectorizationTest.cxx`:

```
#include "Math/Random.h"

#include "TRandom1.h"
#include "TStopwatch.h"
#include "TMath_VecTest.h"

#include <iostream>
#include <random>

// using namespace ROOT::Math;
using namespace ROOT;
using namespace std;

int main(){
        // Test and vectorization size
        const int numRepetitions = 1000;
        const int inputSize = 100000;
        const int vecSize = vecCore::VectorSize<Double_v>();

        // Vectorized and linear input
        Double_v vectorInput[inputSize];
        Double_t linearInput[inputSize * vecSize];

        // Vectorized and linear output
        Double_v vectorOutput[inputSize];
        Double_t linearOutput[inputSize * vecSize];

        // Parameters vector
        Double_t mean[inputSize];
        Double_t sigma[inputSize];

        // Randomize input data and parameters
        TRandom1 rndmzr;
        rndmzr.RndmArray(inputSize * vecSize, linearInput);
        rndmzr.RndmArray(inputSize, mean);
        rndmzr.RndmArray(inputSize, sigma);

        // Set -100 < mean < 100 and 0 < sigma < 200
        for (size_t i = 0; i < inputSize; i++) {
          mean[i] = mean[i] * 200 - 100;
          sigma[i] *= 200;
        }
```

```
        // Copy input linear data to the vectorized array
        for (size_t caseIdx = 0; caseIdx < inputSize; caseIdx++) {
          for (size_t vecIdx = 0; vecIdx < vecSize; vecIdx++) {
                vectorInput[caseIdx][vecIdx] = linearInput[vecSize * caseIdx + vecIdx];
          }
        }

        // Clocks to measure (l)inear and (v)ectorized performance
        TStopwatch clock_l, clock_v;

        // Vectorized computation
        clock_v.Start();
        for (size_t _ = 0; _ < numRepetitions; _++) {
          for (size_t caseIdx = 0; caseIdx < inputSize; caseIdx++) {
                vectorOutput[caseIdx] = TMath::Gaus_v(vectorInput[caseIdx],
                                                      mean[caseIdx], sigma[caseIdx]);
          }
        }
        clock_v.Stop();

        // Linear computation
        int idx;
        clock_l.Start();
        for (size_t _ = 0; _ < numRepetitions; _++) {
          for (size_t caseIdx = 0; caseIdx < inputSize; caseIdx++) {
                for (size_t vecIdx = 0; vecIdx < vecSize; vecIdx++) {
                  idx = caseIdx * vecSize + vecIdx;
                  linearOutput[idx] = TMath::Gaus(linearInput[idx], mean[caseIdx],
                                                  sigma[caseIdx]);
                }
          }
        }
        clock_l.Stop();

        // Check
        for (size_t caseIdx = 0; caseIdx < inputSize; caseIdx++) {
          for (size_t vecIdx = 0; vecIdx < vecSize; vecIdx++) {
                Double_t diff = TMath::Abs(linearOutput[caseIdx*vecSize + vecIdx] -
                                           vectorOutput[caseIdx][vecIdx]);
                if(diff > 1e-15)
                  std::cout << diff << std::endl;
          }
        }

        cout << "Linear:      "; clock_l.Print();
        cout << "Vectorized: "; clock_v.Print();
        cout << "SPEEDUP:     x" << clock_l.RealTime() / clock_v.RealTime() << endl;
}
```

This benchmark was executed in my machine, whose processor (i7-930) has the extension SSE4.2 but not AVX2; as FMA is not present in SSE4.2, the upper bound of the speedup is 2. Running this benchmark in this specific configuration, the mean speedup obtained is of approximately 1.7.

Student Information

Bio of Student

My name is Alejandro García Montoro, I recently graduated with a Bachelor's degree in Computer Science and a Bachelor's degree in Mathematics at the University of Granada, Spain.

Currently, I have a full time job at the Free Software Office of the University of Granada, working for the CERN team that is developing KiCad. The contract with CERN finishes May 15th and therefore I will have enough time for GSoC if I am selected. My work there can be seen at Launchpad.

My experience with parallelization comes mainly from my Bachelor's thesis: I developed a relativistic raytracer implemented with a CUDA-parallelized RungeKutta 5(4) solver with adaptive size (see my GitHub repo, with both the software and the thesis).

Back in 2014 I participated in the BEXUS 19 mission with GranaSAT team: we successfully tested a prototype of an attitude determination system for a cubesat on a stratospheric balloon launched from Kiruna, Sweden. I was in charge of all the communication software plus one of the prototypes: a horizon sensor. All of my code (in C) can be found at Github.

Last summer I contributed to OpenSEMBA, an open source suite for electromagnetic simulations developed at my university. During my contribution I implemented the Vector Fitting method from B. Gustavsen and A. Semlyen in C++.

Contact Information

Name:	Alejandro García Montoro
Postal address:	C/Cañaveral, 8, 7B, 18003, Granada, Spain
Telephone:	+34606217192
Emails:	alejandro.garciamontoro@gmail.com agarciamontoro@correo.ugr.es
Telegram:	@agarciamontoro
Skype:	alejandro.garciamontoro
Hangouts:	alejandro.garciamontoro@gmail.com
IRC:	xiroux
Github:	@agarciamontoro

Student Affiliation

Institution:	University of Granada, Spain
Program:	Double Degree in Computer Science and Mathematics
Stage of completion:	100%