GooFit: A library for massively parallelising maximum-likelihood fits

[1] University of Cincinnati, 2600 Clifton Avenue, Cincinnati OH 45220, USA
[2] Ohio Supercomputer Center, 1224 Kinnear Road, Columbus OH 43212, USA

E-mail: rolfa@slac.stanford.edu

Abstract. Fitting complicated models to large datasets is a bottleneck of many analyses. We present GooFit, a library and tool for constructing arbitrarily-complex probability density functions (PDFs) to be evaluated on nVidia GPUs. The massive parallelisation of dividing up event calculations between hundreds of processors can achieve speedups of factors 200-300 in real-world problems.

1. Introduction

Parameter estimation is a crucial part of many physics analyses. GooFit is an interface between the MINUIT minimisation algorithm and a Graphics Processing Unit (GPU), which allows a probability density function (PDF) to be evaluated in parallel, that is, for several hundred events at once. Since PDF evaluation on large datasets is usually the bottleneck in the MINUIT algorithm, this can result in speedups of up to $\sim$ 200 in real problems - which can be the difference between waiting overnight for the answer, or making a cup of tea.

2. Real-world example: Time-dependent Dalitz-plot fit

To ensure the usefulness of GooFit in real-world physics problems, we have, simultaneously with developing GooFit, used it as our fitting tool in a time-dependent Dalitz-plot analysis of the decay $D^0 \to \pi^+\pi^-\pi^0$. This fit has been our "driver" for GooFit, in that every time we needed a feature for the physics, we added it to the GooFit engine.

This mixing fit is rather complex, involving, for the signal component, 16 complex resonances to describe the Dalitz-plot distribution, a time component where hyperbolic and trigonometric functions are convolved with Gaussian resolution functions, and a distribution of the uncertainty σ_t on the decay time which varies across the Dalitz plot. All in all there are about 40 free parameters in the fit, and the data set is roughly a hundred thousand events; running this on a modern CPU takes between ten and twenty hours, depending on the data. Using GooFit this is reduced to a much more comfortable 3-5 *minutes*, a speedup factor in the region of 200.

3. User-level code

The purpose of GooFit is to give users access to the parallelising power of CUDA without requiring them to write CUDA code. At the most basic level, GooFit objects representing PDFs, `GooPdf`s, can be created and combined in plain C++. Only if a user needs to represent a function outside the existing GooFit classes does he need to do any CUDA coding; Section 5 shows how to create new PDF classes. We intend, however, that this should be a rarity, and that the existing PDF classes should cover all the most common cases.

A GooFit program has four main components:

- The PDF that models the physical process, represented by a `GooPdf` object.
- The fit parameters with respect to which the likelihood is maximised, represented by `Variable`s contained in the `GooPdf`.
- The data, gathered into a `DataSet` object containing one or more `Variable`s.
- A `FitManager` object which forms the interface between MINUIT (or, in principle, a different maximising algorithm) and the `GooPdf`.

Listing 1 shows a simple fit of an exponential function.

Listing 1. *Fit for unknown parameter α in $e^{\alpha x}$. GooFit classes are shown in red, important operations in blue.*

```
int main (int argc, char** argv) {
  // Independent variable. Name, lower limit, upper limit.
  Variable* xvar = new Variable("xvar", 0, log(1+RAND_MAX/2));

  // Create data set
  UnbinnedDataSet data(xvar);
  for (int i = 0; i < 100000; ++i) {
    // Generate toy event
    xvar->value = xvar->upperlimit - log(1+rand()/2);
    // ...and add to data set.
    data.addEvent();
  }

  // Create fit parameter - name, initial value, step size, lower and upper limit.
  Variable* alpha = new Variable("alpha", -2, 0.1, -10, 10);
  // Create GooPdf object - name, independent variable, fit parameter.
  ExpPdf* exppdf = new ExpPdf("exppdf", xvar, alpha);
  // Move data to GPU
  exppdf->setData(&data);

  FitManager fitter(exppdf);
  fitter.fit();

  return 0;
}
```

4. Combining functions

To create arbitrarily-complex PDFs, the user can combine simple ones like the exponential and Gaussian in several ways, the most common being addition and multiplication. For example, a two-dimensional distribution, described by a separate exponential in each of two variables x and y, may be represented as a `ProdPdf` object, as shown in Listing 2.

Listing 2. *Product of two exponentials.*

```
int main (int argc, char** argv) {
  Variable* xvar = new Variable("xvar", 0, log(1+RAND_MAX/2));
  Variable* yvar = new Variable("yvar", 0, log(1+RAND_MAX/2));

  vector<Variable*> varList;
  varList.push_back(xvar);
  varList.push_back(yvar);
  UnbinnedDataSet data(varList);
  for (int i = 0; i < 100000; ++i) {
    xvar->value = xvar->upperlimit - log(1+rand()/2);
    yvar->value = yvar->upperlimit - log(1+rand()/2);
    data.addEvent();
  }

  Variable* alpha_x = new Variable("alpha_x", -2.4, 0.1, -10, 10);
  Variable* alpha_y = new Variable("alpha_y", -1.1, 0.1, -10, 10);
  vector<PdfBase*> pdfList;
  pdfList.push_back(new ExpPdf("exp_x", xvar, alpha_x));
  pdfList.push_back(new ExpPdf("exp_y", yvar, alpha_y));

  ProdPdf* product = new ProdPdf("product", pdfList);
  product->setData(&data);

  FitManager fitter(product);
  fitter.fit();

  return 0;
}
```

5. Adding new PDFs

For advanced users who need esoteric functions, GooFit makes it easy to write a new PDF class. There are two steps to the process:

- Write an evaluation method with the required signature, using the provided index array to look up parameters and observables. An example is shown in Listing 3.
- Create a C++ class inheriting from `GooPdf`, in which the constructor populates the index array through the `registerParameter` method. Listing 4 shows an example.

That's it! Putting the new `FooPdf.cu` and `FooPdf.hh` files in the `PDFs` directory of GooFit will cause them to be compiled with the rest of the framework, and be available for use in the same way as the pre-existing PDFs.

Listing 3. *Evaluation function for a Gaussian PDF. Note the double indirection of the index-array lookups. Here* `fptype` *indicates a floating-point number, by default* `double` *precision.*

```
__device__ fptype device_Gaussian (fptype* evt, fptype* p, unsigned int* indices) {
  fptype x = evt[indices[2 + indices[0]]];
  fptype mean = p[indices[1]];
  fptype sigma = p[indices[2]];

  fptype ret = EXP(-0.5*(x-mean)*(x-mean)/(sigma*sigma));
  return ret;
}

__device__ device_function_ptr ptr_to_Gaussian = device_Gaussian;
```

Listing 4. *Populating the index array used in Listing 3.*

```
__host__ GaussianPdf::GaussianPdf (std::string n, Variable* _x,
                                   Variable* mean, Variable* sigma)
  : GooPdf(_x, n)
{
  std::vector<unsigned int> pindices;
  pindices.push_back(registerParameter(mean));
  pindices.push_back(registerParameter(sigma));
  cudaMemcpyFromSymbol((void**) &host_fcn_ptr, ptr_to_Gaussian, sizeof(void*));
  initialise(pindices);
}
```

6. Program flow and organisation

This section should contain a diagram of the various GooFit modules, plus the GPU and MINUIT (perhaps as literally black boxes). Also an explanation in text of what the various things do. Then, a flow diagram of how the numbers pass back and forth during a fit.

7. Summary

We have developed GooFit for use in real-world physics problems, and have achieved speedups of ~ 200 in particular analyses. We believe that we have created a robust framework that is easy to use for a new graduate student, but flexible enough that the most advanced analyses will find it useful.

8. Acknowledgements
NSF funding, code from such-and-such, valuable suggestions and help from Cristoph Deil, feedback from Stefanie and Ollie.