Contents

1 Introduction **2**
1.1 Getting started 3

2 User-level code **4**
2.1 Data sets 6
2.2 Fit types 6

3 Creating new PDF classes **8**
3.1 The indices array 9
3.2 Constants 11

4 Program flow **13**
4.1 Copying data 13
4.2 MINUIT setup 14
4.3 PDF evaluation 15

5 Existing PDF classes **24**
5.1 Basic PDFs 24
5.2 Combination PDFs 31
5.3 Specialised amplitude-analysis functions 34

1 Introduction

GooFit[1] is a framework for creating arbitrary probability density functions (PDFs) and evaluating them over large datasets using nVidia Graphics Processing Units (GPUs). New PDFs are written partly in nVidia's CUDA programming language and partly in C++; however, no expertise in CUDA is required to get started, because the already-existing PDFs can be put together in plain C++.

Aside from the mass of unenlightened hominids who have not yet discovered their need for a massively-parallel fitting framework, there are three kinds of GooFit users:

- Initiates, who write "user-level code" - that is, code which instantiates existing PDF classes in some combination. No knowledge of CUDA is required for this level. If your data can be described by a combination of not-too-esoteric functions, even if the combination is complicated, then user code is sufficient. Section 2 gives an example of how to write a simple fit.

- Acolytes, or advanced users, who have grasped the art of creating new PDF classes. This involves some use of CUDA, but is mainly a question of understanding the variable-index organisation that GooFit PDFs use. Section 3 considers this organisation in some depth.

- Ascended Masters, or architects, who by extended meditation have acquired a full understanding of the core engine of GooFit, and can modify it to their desire[2]. Section 4 gives a detailed narrative of the progress of a PDF evaluation through the engine core, thus elucidating its mysteries. It should only rarely be necessary to acquire this level of mastery; in principle only the developers of GooFit need to know its internal details.

Aside from considerations of the user's understanding, GooFit does require a CUDA-capable graphics card to run on, with compute capability at least 2.1. Further, you will need nVidia's CUDA SDK, in particular the `nvcc` compiler. Aside from this, GooFit is known to compile and run on Fedora

[1] Named in homage to RooFit, with the 'G' standing for 'GPU'.

[2] Although, if they are Buddhist masters, they don't even though they can, since they have transcended desire - and suffering with it.

14, Ubuntu 12.04, and OSX 10.8.4. It has been tested on the Tesla, Fermi, and Kepler generations of nVidia GPUs.

1.1 Getting started

You will need to have a CUDA-capable device and to have the development environment (also known as the software development kit or SDK) set up, with access to the compiler `nvcc` and its libraries. If you have the hardware, you can get the SDK from nVidia's website.

With your CUDA environment set up, you can install GooFit thus:

- Clone from the GitHub repository:

  ```
  git clone git://github.com/GooFit/GooFit.git
  cd GooFit
  ```

- If necessary, edit the Makefile so the variable `CUDALOCATION` points to your local CUDA install.

- Compile GooFit with `gmake` or `make`. Do not be alarmed by warning messages saying that such-and-such a function's stack size could not be statically determined; this is an unavoidable (so far) side effect of the function-pointer implementation discussed in section 4.

- Compile and run the 'simpleFitExample' program, which generates three distributions, fits them, and plots the results:

  ```
  cd examples/simpleFit
  gmake
  ./simpleFitExample
  ```

  The expected output is a MINUIT log for three different fits, and three image files.

- Compile and run the Dalitz-plot tutorial, which fits a text file containing toy Monte Carlo data to a coherent sum of Breit-Wigner resonances:

  ```
  cd examples/dalitz
  export CUDALOCATION=/usr/local/cuda/
  gmake
  ./dalitz dalitz_toyMC_000.txt
  ```

Quick troubleshooting: If your shell doesn't like `export`, try instead `setenv CUDALOCATION /usr/local/cuda/`. Check that `/usr/local/cuda/` exists and contains, eg, `bin/nvcc` - otherwise, track down the directory that does and set `CUDALOCATION` to point to that instead. Some installs have `make` in place of `gmake`.

The text file contains information about simulated decays of the D^0 particle to $\pi^+\pi^-\pi^0$; in particular, in each line, the second and third numbers are the Dalitz-plot coordinates $m^2(pi^+\pi^0)$ and $m^2(pi^-\pi^0)$. The `dalitz` program creates a PDF describing the distribution of these two variables in terms of Breit-Wigner resonances, reads the data, sends it to the GPU, and fits the PDF to the data - the floating parameters are the complex coefficients of the resonances. The expected output is a MINUIT fit log showing that the fit converged, with such-and-such values for the real and imaginary parts of the resonance coefficients.

2 User-level code

From the outside, GooFit code should look like ordinary, object-oriented C++ code: The CUDA parts are hidden away inside the engine, invisible to the user. Thus, to construct a simple Gaussian fit, merely declare three `Variable` objects and a `GaussianPdf` object that uses them, and create an appropriate `UnbinnedDataSet` to fit to:

Listing 1 *Simple Gaussian fit.*

```
int main (int argc, char** argv) {
  // Create an object to represent the observable,
  // the number we have measured. Give it a name,
  // upper and lower bounds, and a number of bins
  // to use in numerical integration.
  Variable* xvar = new Variable("xvar", -5, 5);
  xvar->numbins = 1000;

  // A data set to store our observations in.
  UnbinnedDataSet data(xvar);

  // "Observe" ten thousand events and add them
  // to the data set, throwing out any events outside
```

```
  // the allowed range. In a real fit this step would
  // involve reading a previously created file of data
  // from an _actual_ experiment.
  TRandom donram(42);
  for (int i = 0; i < 10000; ++i) {
    fptype val = donram.Gaus(0.2, 1.1);
    if (fabs(val) > 5) {--i; continue;}
    data.addEvent(val);
  }

  // Variables to represent the mean and standard deviation
  // of the Gaussian PDF we're going to fit to the data.
  // They take a name, starting value, optional initial
  // step size and upper and lower bounds. Notice that
  // here only the mean is given a step size; the sigma
  // will use the default step of one-tenth of its range.
  Variable* mean = new Variable("mean", 0, 1, -10, 10);
  Variable* sigm = new Variable("sigm", 1, 0.5, 1.5);

  // The actual PDF. The Gaussian takes a name, an independent
  // (ie observed) variable, and a mean and width.
  GaussianPdf gauss("gauss", xvar, mean, sigm);

  // Copy the data to the GPU.
  gauss.setData(&data);

  // A class that talks to MINUIT and GooFit. It needs
  // to know what PDF it should set up in MINUIT.
  FitManager fitter(&gauss);

  // The actual fit.
  fitter.fit();
  return 0;
}
```

Notice that, behind the scenes, GooFit assumes that there will be exactly one top-level PDF and data set; it is not advised to break this assumption unless you know what you are doing and exactly how you are getting around

it.

2.1 Data sets

To create a data set with several dimensions, supply a `vector` of `Variables`:

```
vector<Variable*> vars;
Variable* xvar = new Variable("xvar", -10, 10);
Variable* yvar = new Variable("yvar", -10, 10);
vars.push_back(xvar);
vars.push_back(yvar);
UnbinnedDataSet data(vars);
```

In this case, to fill the data set, set the `Variable` values and call the `addEvent` method without arguments:

```
xvar->value = 3;
yvar->value = -2;
data.addEvent();
```

This will add an event with the current values of the `Variable` list to the data set. In general, where an unknown number of arguments are wanted, GooFit prefers to use a `vector` of pointers.

2.2 Fit types

By default, GooFit will do an unbinned maximum-likelihood fit, where the goodness-of-fit metric that is minimised[3] is the negative sum of logarithms of probabilities, which is equivalent to maximising the joint overall probability:

$$\mathcal{P} \;=\; -2 \sum_{events} \log(P_i) \tag{1}$$

where P_i is the PDF value for event i.

To get a binned fit, you should create a `BinnedDataSet` instead of the `UnbinnedDataSet`; the procedure is otherwise the same. Notice that the `BinnedDataSet` will use the number of bins that its constituent `Variables` have at the moment of its creation. Supplying a `BinnedDataSet` to a `GooPdf`

[3]For historical reasons, MINUIT always minimises rather than maximising.

(which is the base class of all the `FooPdf` classes such as `GaussianPdf`) will, by default, make it do a binned negative-log-likelihood fit, in which the goodness-of-fit criterion is the sum of the logarithms of the Poisson probability of each bin:

$$\mathcal{P} \quad = \quad -2 * \sum_{bins} (N * \log(E) - E) \tag{2}$$

where E is the expected number of events in a bin and N is the observed number.

There are two non-default variants of binned fits: A chisquare fit where the error on a bin entry is taken as the square root of the number of observed entries in it (or 1 if the bin is empty), and a "bin error" fit where the error on each bin is supplied by the `BinnedDataSet`. To do such a fit, in addition to supplying the `BinnedDataSet` (and providing the errors through the `setBinError` method in the case of the bin error fit), you should create a suitable `FitControl` object and send it to the top-level `GooPdf`:

```
decayTime = new Variable("decayTime", 100, 0, 10);
BinnedDataSet* ratioData = new BinnedDataSet(decayTime);
for (int i = 0; i < 100; ++i) {
  ratioData->SetBinContent(getRatio(i));
  ratioData->SetBinError(getError(i));
}

vector<Variable*> weights;
weights.push_back(new Variable("constaCoef", 0.03, 0.01, -1, 1));
weights.push_back(new Variable("linearCoef", 0, 0.01, -1, 1));
weights.push_back(new Variable("secondCoef", 0, 0.01, -1, 1));

PolynomialPdf* poly;
poly = new PolynomialPdf("poly", decayTime, weights);
poly->setFitControl(new BinnedErrorFit());
poly->setData(ratioData);
```

The `FitControl` classes are `UnbinnedNLLFit` (the default), `BinnedNLLFit` (the default for binned fits), `BinnedErrorFit` and `BinnedChisqFit`.

3 Creating new PDF classes

The simplest way to create a new PDF is to take the existing `GaussianPdf` class as a template. The existence of a `FooPdf.cu` file in the `FPOINTER` directory is, because of Makefile magic, sufficient to get the `Foo` PDF included in the GooFit library. However, a certain amount of boilerplate is necessary to make the PDF actually work. First of all, it needs a device-side function with a particular signature:

Listing 2 *Signature of evaluation functions.*

```
__device__ fptype device_Gaussian (fptype* evt,
                                   fptype* p,
                                   unsigned int* indices);
```

Notice that this is a standalone function, not part of any class; `nvcc` does not play entirely nicely with device-side polymorphism, which is why we organise the code using a table of function pointers - a poor man's implementation of the virtual-function lookups built into C++. Second, we need a pointer to the evaluation function:

```
__device__ device_function_ptr ptr_to_Gaussian = device_Gaussian;
```

where `device_function_ptr` is defined (using `typedef`) as a pointer to a function with the signature shown in listing 2:

```
typedef fptype (*device_function_ptr) (fptype*,
                                       fptype*,
                                       unsigned int*);
```

This pointer[4] will be copied into the `device_function_table` array, and its index in that array is the PDF's internal representation of "my evaluation function".

Finally, the new PDF needs a bog-standard C++ class definition, extending the `GooPdf` superclass, which will allow it to be instantiated and passed around in user-level code. Section 3.1 discusses what should happen in the constructor; otherwise the class may have any supporting paraphernalia that are necessary or useful to its evaluation - caches, lists of components, pointers to device-side constants, whatever.

[4]You might ask, why not copy the function directly? The reason is that `cudaMemcpy` doesn't like to get the address of a function, but `nvcc` is perfectly happy to statically initialise a pointer. It's a workaround, in other words.

3.1 The indices array

The heart of a PDF's organisation is its index array, which appears in the arguments to its device-side evaluation function as `unsigned int* indices`. The index array stores the position of the parameters of the PDF within the global parameter array; this allows different PDFs to share the same parameters, as in two Gaussians with a common mean. It also stores the position of the event variables, sometimes called observables, within the event array passed to the evaluation function; this is the argument `fptype* evt`.

The index array is created by the constructor of a PDF class; in particular, the constructor should call `registerParameter` so as to obtain the global indices of its parameters, store these numbers in a `vector<unsigned int>` (conventionally called `pindices`), and pass this `vector` to `initialise`. The PDF constructor should also call `registerObservable` on each of the event variables it depends on.

The `initialise` method constructs the array that is used on the GPU side, which consists of four parts. First is stored the number of parameters, which is equal to the size of the `pindices vector`. Next come the indices of the parameters, in the order they were put into `pindices`. Then comes the number of observables, and finally the indices of the observables, again in the order they were registered.

An example may be useful at this point. Consider the simple Gaussian PDF constructor:

```
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);
}
```

This is almost the simplest possible PDF: Two parameters, one observable, no messing about! Notice that the call to `registerObservable` is done in the parent `GooPdf` constructor - this saves some boilerplate in the constructors of one-observable PDFs. For the second and subsequent observables the calls should be done manually. The device-side index array for the Gaussian, assuming it is the only PDF in the system, looks like this:

```
index  0 1 2 3 4
value  2 0 1 1 0
```

Here the initial 2 is the number of parameters - mean and sigma. Then come their respective indices; since by assumption the Gaussian is the only PDF we're constructing, these will simply be 0 and 1. Then comes the number of observables, which is 1, and finally the index of the observable - which, as it is the only observable registered, must be 0. Now we can consider how the device-side code makes use of this:

```
__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;
}
```

The calculation of the Gaussian is straightforward enough, but let's look at where the numbers `mean, sigma` and especially `x` come from. The function is passed a pointer to the particular event it is to calculate the value for, a global parameter array, and the index array. The parameter array, in the case of a single Gaussian, consists simply of the values for the mean and sigma in the current MINUIT iteration. Let us replace the index lookups in those lines with their values from above:

```
  fptype mean = p[0];
  fptype sigma = p[1];
```

which is exactly what we want. The fetching of `x` appears a little more formidable, with its double `indices` lookup; it calls for some explanation. First, `indices[0]` is the number of parameters of the function; we want to skip ahead by this number to get to the 'event' part of the array. In the Gaussian, this is known at compile-time to be 2, but not every PDF writer is so fortunate; a polynomial PDF, for example, could have an arbitrary number of parameters. (Or it might specify a maximum number, say 10, and in most cases leave seven or eight of them fixed at zero - but then there would be a lot of wasted multiplications-by-zero and additions-of-zero.) Thus, as a convention, lookups of event variables should always use `indices[0]` even if the coder knows what that number is going to be. Then, 2 must be added to this number to account for the space taken by the number-of-parameters and number-of-observables entries in the array. So, replacing the first level of lookup by the values, we have:

```
fptype x = evt[indices[4]];
```

and `indices[4]` is just 0; so in other words, `x` is the first observable in the event. In the case of the single Gaussian, it is also the only observable, so we've done quite a bit of work to arrive at a zero that we knew from the start; but in more complex fits this would not be true. The `x` variable could be observable number 5, for all we know to the contrary in the general case. Likewise the mean and sigma could be stored at positions 80 and 101 of the global parameter array.

3.2 Constants

There are two ways of storing constants, or three if we count registering a `Variable` as a parameter and telling MINUIT to keep it fixed. For integer constants, we may simply store them in the index array; since it is up to the programmer to interpret the indices, there is no rule that says it absolutely must be taken as an offset into the global parameter array! An index can also store integers for any other purpose - the maximum degree of a polynomial, flagging the use of an optional parameter, or anything else you can think of. Indeed, this is exactly what the framework does in enforcing the convention that the first number in the index array is the number of parameters.

However, this will not serve for non-integer-valued constants. They must either go through MINUIT as fixed parameters, or else go into the `functorConstants`

array. `functorConstants` works just like the global parameters array, except that it does not update on every MINUIT iteration since it is meant for storing constants. To use it, you should first reserve some space in it using the `registerConstants` method, which takes the number of constants you want as an argument and returns the index of the first one. Usually you will want to put that index in the `pindices` array. For example, suppose I want to store $\sqrt{2\pi}$ as a constant for use in the Gaussian. Then I would modify the constructor thus:

```
__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));

  pindices.push_back(registerConstants(1));
  fptype sqrt2pi = SQRT(2*M_PI);
  cudaMemcpyToSymbol(functorConstants, &sqrt2pi, sizeof(fptype),
                     cIndex*sizeof(fptype), cudaMemcpyHostToDevice);

  cudaMemcpyFromSymbol((void**) &host_fcn_ptr, ptr_to_Gaussian, sizeof(void*));
  initialise(pindices);
}
```

Notice the member variable `cIndex`, which is set (and returned) by `registerConstants`; it is the index of the first constant belonging to this object. To extract my constant for use in the device function, I look it up as though it were a parameter, but the target array is `functorConstants` instead of the passed-in `p`:

```
__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 sqrt2pi = functorConstants[indices[3]];

  fptype ret = EXP(-0.5*(x-mean)*(x-mean)/(sigma*sigma));
  ret /= sqrt2pi;
  return ret;
}
```

If I had registered two constants instead of one, the second one would be looked up by `functorConstants[indices[3] + 1]`, not the `functorConstants[indices[4]]` one might naively expect. This is because the constant is stored next to the first one registered, but its index is not stored at all; it has to be calculated from the index of the first constant. Thus the `+1` must go outside the indices lookup, not inside it! Keeping the levels of indirection straight when constructing this sort of code calls for some care and attention.

Note that `functorConstants[0]` is reserved for the number of events in the fit.

4 Program flow

This section narrates the course of a fit after it is created, passing through MINUIT and the core GooFit engine. In particular, we will consider the example Gaussian fit shown in listing 1 and look at what happens in these innocent-looking lines:

Listing 3 *Data transfer and fit invocation.*

```
gauss.setData(&data);
FitManager fitter(&gauss);
fitter.fit();
```

4.1 Copying data

The `setData` method copies the contents of the supplied `DataSet` to the GPU:

Listing 4 *Internals of the setData method.*

```
setIndices();
int dimensions = observables.size();
numEntries = data->getNumEvents();
numEvents = numEntries;

fptype* host_array = new fptype[numEntries*dimensions];
for (int i = 0; i < numEntries; ++i) {
  for (obsIter v = obsBegin(); v != obsEnd(); ++v) {
    fptype currVal = data->getValue((*v), i);
    host_array[i*dimensions + (*v)->index] = currVal;
  }
}

cudaMalloc((void**) &cudaDataArray, dimensions*numEntries*sizeof(fptype));
cudaMemcpy(cudaDataArray, host_array, dimensions*numEntries*sizeof(fptype), cudaMe
cudaMemcpyToSymbol(functorConstants, &numEvents, sizeof(fptype), 0, cudaMemcpyHost
delete[] host_array;
```

Notice the call to `setIndices`; this is where the indices of observables passed to the PDF are decided and copied into the indices array. This step cannot be done before all the subcomponents of the PDF have had a chance to register their observables. Hence `setData` should be called only after the creation of all PDF components, and only on the top-level PDF.

The array thus created has the simple structure `x1 y1 z1 x2 y2 z2 ... xN yN zN`, that is, the events are laid out contiguously in memory, each event consisting simply of the observables, in the same order every time. Notice that if the `DataSet` contains `Variable`s that have not been registered as observables, they are ignored. If `setData` is called with an `BinnedDataSet` object, the procedure is similar except that each 'event' consists of the coordinates of the bin center, the number of events in the bin, and either the bin error or the bin size. We will see later how the engine uses the `cudaDataArray` either as a list of events or a list of bins.

4.2 MINUIT setup

Having copied the data to the GPU, the next task is to create the MINUIT object that will do the actual fit; this is done by creating a `FitManager` object, with the top-level PDF as its argument, and calling its `fit` method.

The `fit` method does two things: First it calls the `getParameters` method of the supplied PDF, which recursively gets the registered parameters of all the component PDFs, and from the resulting list of `Variable`s it creates MINUIT parameters by calling `DefineParameter`. Second, it sets the method `FitFun` to be MINUIT's function-to-minimise, and calls MINUIT's `mnmigr` method.

A few variants on the above procedure exist. Most obviously, ROOT contains three implementations of the MINUIT algorithm, named `TMinuit`, `TMinuit2`, and `TVirtualFitter`[5]. One can switch between these by setting the constant `MINUIT_VERSION` in FitManager.hh to, respectively, 1, 2, and 3. The interfaces differ, but the essential procedure is the one described above: Define parameters, set function-to-minimise, run MIGRAD. (NB: As of v0.2, GooFit has not recently been tested with `MINUIT_VERSION` set to 2 or 3.) In the case of `TMinuit`, one can call `setMaxCalls` to override the usual MINUIT limitation on the number of iterations, although my experience is that this is not usually helpful because running into the iteration limit tends to indicate a deeper problem with the fit. Finally, the underlying `TMinuit` object is available through the `getMinuitObject` method, allowing fine-grained control of what MINUIT does, for example by calling `mnhess` in place of `mnmigr`.

4.3 PDF evaluation

We have copied the data to the GPU, set up MINUIT, and invoked `mnmigr`. Program flow now passes to MINUIT, which for purposes of this documentation is a black box, for some time; it returns to GooFit by calling the `FitFun` method with a list of parameters for which MINUIT would like us to evaluate the NLL. `FitFun` translates MINUIT indices into GooFit indices, and calls `copyParams`, which eponymously copies the parameter array to `cudaArray` on the GPU. `FitFun` then returns the value from `GooPdf::calculateNLL` to MINUIT, which absorbs the number into its inner workings and eventually comes back with another set of parameters to be evaluated. Control continues to pass back and forth in this way until MINUIT converges or gives up,

[5]These are, respectively, ancient FORTRAN code translated line-by-line into C++, almost literally by the addition of semicolons; someone's obsessively-detailed object-oriented implementation of the same algorithm, with the same spaghetti logic chopped into classes instead of lines of code; and what seems to be intended as a common interface for a large number of possible fitting backends, which falls a little flat since it has only the MINUIT backend to talk to. You pays your money and takes your choice.

or until GooFit crashes.

The `calculateNLL` method does two things: First it calls the `normalise` function of the PDF, which in turn will usually recursively normalise the components; the results of the `normalise` call are copied into the `normalisationFactors` array on the GPU. Next it calls `sumOfNll` and returns the resulting value. Particular PDF implementations may override `sumOfNll`; most notably `AddPdf` does so in order to have the option of returning an 'extended' likelihood, with a term for the Poisson probability of the observed number of events in addition to the event probabilities.

The `normalise` method, by default, simply evaluates the PDF at a grid of points, returning the sum of all the values multiplied by the grid fineness - a primitive algorithm for numerical integration, but one which takes advantage of the GPU's massive parallelisation. The fineness of the grid usually depends on the `numbins` member of the observables; in the case of the example Gaussian fit in listing 1, the PDF will be evaluated at 1000 points, evenly spaced between -5 and 5. However, this behaviour can be overridden by calling the `setIntegrationFineness` method of the PDF object, in which case the number of bins (in each observable) will be equal to the supplied fineness.

Stripped of complications, the essential part of the `normalise` function is a call to `transform_reduce`:

Listing 5 *Normalisation code.*

```
  fptype dummy = 0;
  static plus<fptype> cudaPlus;
  constant_iterator<fptype*> arrayAddress(normRanges);
  constant_iterator<int> eventSize(observables.size());
  counting_iterator<int> binIndex(0);

  fptype sum = transform_reduce(make_zip_iterator(
                                  make_tuple(binIndex,
                                             eventSize,
                                             arrayAddress)),
                                make_zip_iterator(
                                  make_tuple(binIndex + totalBins,
                                             eventSize,
                                             arrayAddress)),
*logger, dummy, cudaPlus);
```

Here `normRanges` is an array of triplets `lower, upper, bins` for each observable, created by the `generateNormRanges` method. The member `logger` points to an instance of the `MetricTaker` class, which has an operator method that Thrust will invoke on each bin index between the initial value of zero and the final value of `totalBins−1`. This operator method, which is invoked once per thread with a separate (global) bin number for each invocation, calculates the bin center and returns the value of the PDF at that point. The `dummy` and `cudaPlus` variables merely indicate that Thrust should add (rather than, say, multiply) all the returned values, and that it should start the sum at zero. The `normalisation` method returns this sum, but stores its inverse in the `host_normalisation` array that will eventually be copied to `normalisationFactors` on the GPU; this is to allow the micro-optimisation of multiplying by the inverse rather than dividing in every thread.

PDF implementations may override the `normalisation` method, and among the default PDFs, both `AddPdf` and `ProdPdf` do so to ensure that their components are correctly normalised. Among the more specialised implementations, `TddpPdf` overrides `normalise` so that it may cache the slowly-changing Breit-Wigner calculations, and also because its time dependence is analytically integrable and it is a good optimisation to do only the Dalitz-plot part numerically. This points to a more general rule, that once a PDF depends on three or four observables, the relatively primitive numerical integration outlined above may become unmanageable because of the number of points it creates. Finally, note that PDFs may, without overriding `normalise`, advertise an analytical integral by overriding `GooPdf`'s `hasAnalyticIntegral` method to return `true`, and then implementing an `integrate` method to be evaluated on the CPU.

The `logger` object will appear again in the actual PDF evaluation, performing a very similar function, so it is worth taking a moment to consider in detail exactly what the `transform_reduce` call does. The first two parameters (involving `make_tuple` calls) define the range of evaluation: In this case, global bins[6] 0 through $N - 1$. They also specify which `operator` method of `MetricTaker` should be called: It is the one which takes as arguments two integers (the bin index and event size) and an `fptype` array (holding the `normRanges` values), in that order. Conceptually, Thrust will create one

[6] A global bin ranges from 0 to $n_1 n_2 \dots n_N - 1$ where n_j is the number of bins in the jth variable and N is the number of variables. In two dimensions, with three bins in each of x and y, the global bin is given by $3b_y + b_x$, where $b_{x,y}$ is the bin number in x or y

thread for each unique value of the iterator range thus created - that is, one per global bin - and have each thread invoke the indicated `operator` method. As a matter of organisation on the physical chip, it is likely that Thrust will actually create a thousand or so threads and have each thread evaluate as many bins as needed; but at any rate, the `operator(int, int, fptype*)` method will be called once per global bin. The last two arguments indicate that the return value should be calculated as the sum of the return values from each `operator` invocation, and that the sum should start at zero. Finally, the `*logger` argument indicates the specific `MetricTaker` object to use, which is important because this is where the function-pointer and parameter indices are stored.

The `operator` does two things: First it calculates the bin centers, in each observable, of the global bin:

Listing 6 *Bin-center calculation.*

```
__shared__ fptype binCenters[1024*MAX_NUM_OBSERVABLES];

// To convert global bin number to (x,y,z...) coordinates:
// For each dimension, take the mod with the number of bins
// in that dimension. Then divide by the number of bins, in
// effect collapsing so the grid has one fewer dimension.
// Rinse and repeat.

int offset = threadIdx.x*MAX_NUM_OBSERVABLES;
unsigned int* indices = paramIndices + parameters;
for (int i = 0; i < evtSize; ++i) {
  fptype lowerBound = thrust::get<2>(t)[3*i+0];
  fptype upperBound = thrust::get<2>(t)[3*i+1];
  int numBins    = (int) FLOOR(thrust::get<2>(t)[3*i+2] + 0.5);
  int localBin = binNumber % numBins;
```

---

respectively, as shown here:

2	6	7	8
1	3	4	5
0	0	1	2
	0	1	2

where the leftmost column and bottom row indicate the y and x bin number.

```
  fptype x = upperBound - lowerBound;
  x /= numBins;
  x *= (localBin + 0.5);
  x += lowerBound;
  binCenters[indices[indices[0] + 2 + i]+offset] = x;
  binNumber /= numBins;
}
```

in the straightforward way, and stores the bin centers in a fake event. Since events are just lists of observables, all that's necessary is to keep track of which part of the `__shared__ binCenters` array is owned by this thread, look up the index-within-events of each observable, and set the entries of the locally-owned part of `binCenters` accordingly. This fake event is then sent to the PDF for evaluation:

```
fptype ret = callFunction(binCenters+offset,
                          functionIdx,
                          parameters);
```

where `callFunction` is just a wrapper for looking up the function referred to by `functionIdx` and calling it with the right part of the parameter array:

Listing 7 *Code to call device-side PDF implementations (some lines broken up for clarity).*

```
__device__ fptype callFunction (fptype* eventAddress,
                                unsigned int functionIdx,
                                unsigned int paramIdx) {
  void* rawPtr = device_function_table[functionIdx];
  device_function_ptr fcn;
  fcn = reinterpret_cast<device_function_ptr>(rawPtr);
  return (*fcn)(eventAddress,
                cudaArray,
                paramIndices + paramIdx);
}
```

This, finally, is where the `__device__` function from the PDF definitions in section 3 is called; we have now connected all this engine code with the evaluation code for the Gaussian, Breit-Wigner, polynomial, sum of functions, or whatever calculation we happen to be doing today.

Having found the integral of the PDF, either using fake events as outlined above or with an analytic calculation, we are now ready to find the actual NLL, or sum of chi-squares, or other goodness-of-fit metric, using the actual, observed events that we copied across in `setData`. The procedure is similar to that for the normalisation:

Listing 8 *Goodness-of-fit evaluation.*

```
transform_reduce(make_zip_iterator(make_tuple(eventIndex,
                                              arrayAddress,
                                              eventSize)),
                 make_zip_iterator(make_tuple(eventIndex + numEntries,
                                              arrayAddress,
                                              eventSize)),
                 *logger, dummy, cudaPlus);
```

Here the `*logger`, `dummy`, and `cudaPlus` arguments are doing the same jobs as before. The tuple arguments, however, differ: In particular, they are now indicating the range 0 to $N - 1$ in events, not bins, and `arrayAddress` this time points to the array of events, not to a set of normalisation triplets from which bin centers can be calculated. Since the order of the arguments differs - it is now `int, fptype*, int` - a different `operator` method is called:

Listing 9 *Main evaluation operator (some lines broken up for clarity).*

```
__device__ fptype MetricTaker::operator ()
  (thrust::tuple<int, fptype*, int> t) const {
  // Calculate event offset for this thread.
  int eventIndex = thrust::get<0>(t);
  int eventSize  = thrust::get<2>(t);
  fptype* eventAddress = thrust::get<1>(t);
  eventAddress += (eventIndex * abs(eventSize));

  // Causes stack size to be statically undeterminable.
  fptype ret = callFunction(eventAddress, functionIdx, parameters);

  // Notice assumption here! For unbinned fits the
  // eventAddress pointer won't be used in the metric,
```

```
  // so it doesn't matter what it is. For binned fits it
  // is assumed that the structure of the event is
  // (obs1 obs2... binentry binvolume), so that the array
  // passed to the metric consists of (binentry binvolume).
  void* fcnAddr = device_function_table[metricIndex];
  device_metric_ptr fcnPtr;
  fcnPtr = reinterpret_cast<device_metric_ptr>(fcnAddr);
  eventAddress += abs(eventSize)-2;
  ret = (*fcnPtr)(ret, eventAddress, parameters);
  return ret;
}
```

Observe that, for binned events, `eventSize` is negative; in this case the event array looks like `x1 y1 n1 v1 x2 y2 n2 v2 ... xN yN nN vN` where `x` and `y` are bin centers, `n` is the number of entries, and `v` is the bin volume or error. This does not matter for the PDF evaluation invoked by `callFunction`, which will just get a pointer to the start of the event and read off the bin centers as event variables; hence the `abs(eventSize)` in the calculation of the event address allows binned and unbinned PDFs to be treated the same. However, it very much does matter for the goodness-of-fit metric. Suppose the fit is the default NLL: Then all the operator needs to do at this point is take the logarithm of what the PDF returned, multiply by -2, and be on its way. But if it is a chi-square fit, then it must calculate the expected number of hits in the bin, which depends on the PDF value, the bin volume, and the total number of events[7], subtract the observed number, square, and divide by the observed number. Hence there is a second function-pointer lookup, but now the `void*` stored in `device_function_table` is to be interpreted as a different kind of function - a "take the metric" function rather than a "calculate the PDF" function. The `metricIndex` member of `MetricTaker` is set by the `FitControl` object of the PDF; it points to one of the `calculateFoo` functions:

Listing 10 *Metric-taking functions.*

```
__device__ fptype calculateEval (fptype rawPdf,
                                 fptype* evtVal,
                                 unsigned int par) {
```

[7]This is why `functorConstants[0]` is reserved for that value!

```
  // Just return the raw PDF value, for use
  // in (eg) normalisation.
  return rawPdf;
}

__device__ fptype calculateNLL (fptype rawPdf,
                                fptype* evtVal,
                                unsigned int par) {
  rawPdf *= normalisationFactors[par];
  return rawPdf > 0 ? -LOG(rawPdf) : 0;
}

__device__ fptype calculateProb (fptype rawPdf,
                                fptype* evtVal,
                                unsigned int par) {
  // Return probability, ie normalised PDF value.
  return rawPdf * normalisationFactors[par];
}

__device__ fptype calculateBinAvg (fptype rawPdf,
                                fptype* evtVal,
                                unsigned int par) {
  rawPdf *= normalisationFactors[par];
  rawPdf *= evtVal[1]; // Bin volume
  // Log-likelihood of numEvents with expectation of exp
  // is (-exp + numEvents*ln(exp) - ln(numEvents!)).
  // The last is constant, so we drop it; and then multiply
  // by minus one to get the negative log-likelihood.
  if (rawPdf > 0) {
    fptype expEvents = functorConstants[0]*rawPdf;
    return (expEvents - evtVal[0]*log(expEvents));
  }
  return 0;
}

__device__ fptype calculateBinWithError (fptype rawPdf,
                                fptype* evtVal,
                                unsigned int par) {
```

```
  // In this case interpret the rawPdf as just a number,
  // not a number of events. Do not divide by integral over
  // phase space, do not multiply by bin volume, and do not
  // collect 200 dollars. evtVal should have the structure
  // (bin entry, bin error).
  rawPdf -= evtVal[0]; // Subtract observed value.
  rawPdf /= evtVal[1]; // Divide by error.
  rawPdf *= rawPdf;
  return rawPdf;
}

__device__ fptype calculateChisq (fptype rawPdf,
                                  fptype* evtVal,
                                  unsigned int par) {
  rawPdf *= normalisationFactors[par];
  rawPdf *= evtVal[1]; // Bin volume

  fptype ret = pow(rawPdf * functorConstants[0] - evtVal[0], 2);
  ret /= (evtVal[0] > 1 ? evtVal[0] : 1);
  return ret;
}
```

Notice the use of `normalisationFactors` in most of the metric functions, and the special cases when the PDF or the observed number of events is zero.

It is worth noting that the PDF evaluation function may itself call other functions, either using `callFunction` or manually casting a function index into other kinds of functions, as in the metric calculation of listing 9. For example, in `DalitzPlotPdf`, each resonance may be parametrised by a relativistic Breit-Wigner, a Gaussian, a Flatte function, or more esoteric forms; so the main function is supplied with a list of function indices and parameter indices for them, and interprets the `void` pointer from `device_function_table` as a specialised function type taking Dalitz-plot location (rather than a generic event) as its argument. More prosaically, `AddPdf` simply carries a list of PDF function indices and indices of weights to assign them, and invokes `callFunction` several times, multiplying the results by its weight parameters and returning the sum.

We have now calculated the function value that we ask MINUIT to minimise, for a single set of parameters; this value is passed back to MINUIT,

which does its thing and comes up with another set of parameters for us, completing the loop. Ends here the saga of the fit iteration; you now know the entire essential functionality of GooFit's core engine.

5 Existing PDF classes

The GooFit PDFs, like ancient Gaul, are roughly divisible into three:

- Basic functions, written because they are (expected to be) frequently used, such as the Gaussian and polynomial PDFs.
- Combiners, functions that take other functions as arguments and spit out some combination of the inputs, for example sums and products.
- Specialised PDFs, written for the $D^0 \to \pi\pi\pi^0$ mixing analysis that is the driving test case for GooFit's capabilities.

In the lists below, note that all the constructors take pointers to `Variable` objects; rather than repetitively repeat "`Variable` pointer" in a redundantly recurring manner, we just say `Variable`. Additionally, the first argument in every constructor is the name of the object being created; again this is not mentioned in every item. By convention, constructors take observables first, then parameters.

5.1 Basic PDFs

Basic PDFs are relatively straightforward: They take one or more observables and one or more parameters, and implement operations that are in some sense 'atomic' - they do not combine several functions in any way. Usually they have a reasonably well-known given name, for example "the threshold function" or "a polynomial". The canonical example is the Gaussian PDF.

- `ArgusPdf`: Implements a threshold function

$$P(x; m_0, a, p) = \begin{cases} 0 & x \leq m_0 \\ x\left(\frac{x^2-m_0^2}{m_0^2}\right)^p e^{a\frac{x^2-m_0^2}{m_0^2}} & x > m_0 \end{cases} \quad (3)$$

  where the power p is, by default, fixed at 0.5. The constructor takes `Variable`s representing x, m_0, and a, followed by a boolean indicating

whether the threshold is an upper or lower bound. The equation above shows the PDF for a lower bound; for upper bounds, $x^2 - m_0^2$ becomes instead $m_0^2 - x^2$, and the value is zero above rather than below m_0. The constructor also takes an optional `Variable` representing the power p; if not given, a default parameter with value 0.5 is created.

- `BifurGaussPdf`: A two-sided Gaussian, with a σ that varies depending on which side of the mean you are on:

$$P(x; m, \sigma_L, \sigma_R) = \begin{cases} e^{-\frac{(x-m)^2}{2\sigma_L^2}} & x \leq m \\ e^{-\frac{(x-m)^2}{2\sigma_R^2}} & x > m. \end{cases} \quad (4)$$

  The constructor takes the observable x, mean m, and left and right sigmas $\sigma_{L,R}$.

- `BWPdf`: A non-relativistic Breit-Wigner function, sometimes called a Cauchy function:

$$P(x; m, \Gamma) = \frac{1}{2\sqrt{\pi}} \frac{\Gamma}{(x-m)^2 + \Gamma^2/4} \quad (5)$$

  The constructor takes the observable x, mean m, and width Γ.

- `CorrGaussianPdf`: A correlated Gaussian - that is, a function of two variables x and y, each described by a Gaussian distribution, but the width of the y distribution depends on x:

$$P(x, y; \bar{x}, \sigma_x, \bar{y}, \sigma_y, k) = e^{-\frac{(x-\bar{x})^2}{2\sigma_x^2}} e^{-\frac{(y-\bar{y})^2}{2(1+k(\frac{x-\bar{x}}{\sigma_x})^2)\sigma_y^2}} \quad (6)$$

  In other words, the effective σ_y grows quadratically in the normalised distance from the mean of x, with the quadratic term having coefficient k. The constructor takes observables x and y, means and widths $\bar{x}$, σ_x, $\bar{y}$ and σ_y, and coefficient k. Notice that if k is zero, the function reduces to a product of two Gaussians, $P(x, y; \bar{x}, \sigma_x, \bar{y}, \sigma_y) = G(x; \bar{x}, \sigma_x)G(y; \bar{y}, \sigma_y)$.

- `CrystalBallPdf`: A Gaussian with a power-law tail on one side:

$$P(x; m, \sigma, \alpha, p) = \begin{cases} e^{-\frac{(x-m)^2}{2\sigma^2}} & \mathrm{sg}(\alpha)\frac{x-m}{\sigma} \leq \mathrm{sg}(\alpha)\alpha \\ e^{-\alpha^2/2}\left(\frac{p/\alpha}{p/\alpha - \alpha + \frac{x-m}{\sigma}}\right)^p & \text{otherwise}(\alpha \neq 0). \end{cases} \quad (7)$$

The constructor takes the observable x, the mean m, width σ, cutoff α, and power p. Note that if α is negative, the power-law tail is on the right; if positive, on the left. For $\alpha = 0$, the function reduces to a simple Gaussian in order to avoid p/α blowing up.

- `ExpGausPdf`: An exponential decay convolved with a Gaussian resolution:

$$P(t; m, \sigma, \tau) = e^{-t/\tau} \otimes e^{-\frac{(t-m)^2}{2\sigma^2}} \quad (8)$$

$$= (\tau/2) e^{(\tau/2)(2m+\tau\sigma^2-2t)} \mathrm{erfc}\left(\frac{m+\tau\sigma^2-t}{\sigma\sqrt{2}}\right) \quad (9)$$

  where erfc is the complementary error function. The constructor takes the observed time t, mean m and width σ of the resolution, and lifetime τ. Note that the original decay function is zero for $t < 0$.

- `ExpPdf`: A plain exponential,

$$P(x; \alpha, x_0) = e^{\alpha(x-x_0)} \quad (10)$$

  taking the observable x, exponential constant α, and optional offset x_0. If x_0 is not specified it defaults to zero. A variant constructor takes, in place of α, a `vector` of coefficients (in the order α_0 to α_n) to form a polynomial in the exponent:

$$P(x; \alpha_0, \alpha_1, \ldots \alpha_n, x_0) = e^{\alpha_0+\alpha_1(x-x_0)+\alpha_2(x-x_0)^2+\ldots+\alpha_n(x-x_0)^n} \quad (11)$$

  The offset x_0 is again optional and defaults to zero.

- `GaussianPdf`: What can I say? It's a normal distribution, the potato of PDFs. Kind of bland, but goes with anything. National cuisines have been based on it.

$$P(x; m, \sigma) = e^{-\frac{(x-m)^2}{2\sigma^2}} \quad (12)$$

  The constructor takes the observable x, mean m, and width σ.

- `InterHistPdf`: An interpolating histogram; in one dimension:

$$P(x) = \frac{f(x, b(x))H[b(x)] + f(x, 1+b(x))H[b(x)+1]}{f(x, b(x)) + f(x, 1+b(x))} \quad (13)$$

where H is a histogram, $H[n]$ is the content of its bin with index n, $b(x)$ is a function that returns the bin number that x falls into, and $f(x, n)$ is the distance between x and the center of bin n. In other words, it does linear interpolation between bins. However, there are two complicating factors. First, the histogram may have up to ten[8] dimensions. Second, the dimensions may be either observables or fit parameters. So, for example, suppose we want to fit for the width σ of a Gaussian distribution, without using the potato of PDFs. We can do this by making a two-dimensional histogram: The x dimension is the observable, the y is σ. Fill the histogram with the value of the Gaussian[9] at each x given the σ in that bin. Now when the fit asks the PDF, "What is your value at x given this σ?", the PDF responds by interpolating linearly between four bins - ones that were precalculated with σ values close to what the fit is asking about. For the Gaussian this is rather un-necessary, but may save some time for computationally expensive functions.

The constructor takes a `BinnedDataSet` representing the underlying histogram, a `vector` of fit parameters, and a `vector` of observables.

- `JohnsonSUPdf`: Another modified Gaussian. You can eat potatoes a lot of different ways:

$$P(x; m, \sigma, \gamma, \delta) = \frac{\delta}{\sigma\sqrt{2\pi(1 + \frac{(x-m)^2}{\sigma^2})}} e^{-\frac{1}{2}\left(\gamma+\delta\log(\frac{x-m}{\sigma}+\sqrt{1+\frac{(x-m)^2}{\sigma^2}})\right)^2} \quad (14)$$

  The constructor takes the observable x, mean m, width σ, scale parameter γ, and shape parameter δ.

- `KinLimitBWPdf`: A relativistic Breit-Wigner function modified by a factor accounting for limited phase space[10]; for example, in the decay $D^{*+} \to D^0\pi^+$, the difference between the D^* and D^0 masses is only slightly more than the pion mass. Consequently, the distribution of $\Delta m = m(D^*) - m(D^0)$ is slightly asymmetric: The left side of the

[8] On the grounds that ten dimensions should be enough for anyone!

[9] Oops, there's that potato after all. It's a contrived example.

[10] If this seems complicated, spare a thought for the hapless undergrad who had to code the original CPU version.

peak, where the phase space narrows rapidly, is less likely than the right side.

$$P(x;x_0,\Gamma,M,m) = \begin{cases} 0 & \lambda(x_0,M,m) \leq 0 \\ \frac{S(x,x_0,M,m)x_0'\Gamma^2}{\left(x_0'-x'^2\right)^2+x_0'\Gamma^2 S^2(x,x_0,M,m)} & \text{otherwise.} \end{cases} \quad (15)$$

Here priming indicates addition of M, so that $x' = x+M$, $x_0' = x_0+M$; the phase-space function S and its supporting characters λ, p, and b_W are given by

$$S(x,x_0,M,m) = \left(\frac{p(x,M,m)}{p(x_0,M,m)}\right)^3 \left(\frac{b_W(x,M,m)}{b_W(x_0,M,m)}\right)^2 \quad (16)$$

$$b_W(x,M,m) = \frac{1}{\sqrt{1+r^2p^2(x,M,m)}} \quad (17)$$

$$p(x,M,m) = \sqrt{\lambda(x,M,m)/(2x)} \quad (18)$$

$$\lambda(x,M,m) = \left(x'^2-(M-m)^2\right)\left(x'^2-(M+m)^2\right). \quad (19)$$

The radius r that appears in b_W (which does not stand for Breit-Wigner, but Blatt-Weisskopf!) is hardcoded to be 1.6.

The constructor takes the observable x, mean x_0, and width Γ. The large and small masses M and m, which determine the phase space, are by default 1.8645 (the D^0 mass) and 0.13957 (mass of a charged pion), but can be set with a call to `setMasses`. Note that they are constants, not fit parameters.

- `LandauPdf`: A shape with a long right-hand tail - so long, in fact, that its moments are not defined. If the most probable value (note that this is not a mean) and the width are taken as 0 and 1, the PDF is

$$P(x) = \frac{1}{\pi}\int_0^\infty e^{-t\log t - xt}\sin(t\pi)\mathrm{d}t \quad (20)$$

but the GooFit implementation is a lookup table stolen from CERNLIB. The constructor takes the observable x, most probable value μ (which shifts the above expression) and the width σ (which scales it).

- `NovosibirskPdf`: A custom shape with a long tail:

$$P(x;m,\sigma,t) = e^{-\frac{1}{2}\left(\log^2(1+t\frac{x-m}{\sigma}\frac{\sinh(t\sqrt{\log(4)})}{\sqrt{\log(4)}})/t+t^2\right)} \quad (21)$$

The constructor takes the observable x, mean m, width σ, and tail factor t. If t is less than 10^{-7}, the function returns a simple Gaussian, which probably indicates that it approximates a Gaussian for small tail parameters, but I'd hate to have to show such a thing.

- `PolynomialPdf`: If the Gaussian is the potato, what is the polynomial? Bread? Milk? Nothing exotic, at any rate. The GooFit version does have some subtleties, to allow for polynomials over an arbitrary number[11] of dimensions:

$$P(\vec{x};\vec{a},\vec{x}_0,N) = \sum_{p_1+p_2+\ldots+p_n\leq N} a_{p_1p_2\ldots p_n}\prod_{i=1}^{n}(\vec{x}-\vec{x}_0)_i^{p_i} \qquad (22)$$

where N is the highest degree of the polynomial and n is the number of dimensions. The constructor takes a `vector` of observables, denoted $\vec{x}$ above; a `vector` of coefficients, $\vec{a}$, a `vector` of optional offsets $\vec{x}_0$ (if not specified, these default to zero), and the maximum degree N. The coefficients are in the order $a_{p_0p_0\ldots p_0}, a_{p_1p_0\ldots p_0}, \ldots a_{p_Np_0\ldots p_0}, a_{p_0p_1\ldots p_0}, a_{p_1p_1\ldots p_0}, \ldots a_{p_0p_0\ldots p_N}$. In other words, start at the index for the constant term, and increment the power of the leftmost observable. Every time the sum of the powers reaches N, reset the leftmost power to zero and increment the next-leftmost. When the next-leftmost reaches N, reset it to zero and increment the third-leftmost, and so on. An example may be helpful; for two dimensions x and y, and a maximum power of 3, the order is $a_{00}, a_{10}, a_{20}, a_{30}, a_{01}, a_{11}, a_{21}, a_{02}, a_{12}, a_{03}$. This can be visualised as picking boxes out of a matrix and discarding the ones where the powers exceed the maximum:

$$\begin{array}{cccc} 9:x^0y^3 & - & - & - \\ 7:x^0y^2 & 8:x^1y^2 & - & - \\ 4:x^0y^1 & 5:x^1y^1 & 6:x^2y^1 & - \\ 0:x^0y^0 & 1:x^1y^0 & 2:x^2y^0 & 3:x^3y^0 \end{array}$$

starting in the lower-lefthand corner and going right, then up.

There is also a simpler version of the constructor for the case of a polynomial with only one dimension; it takes the observable, a `vector`

[11] Although being honest, just supporting the special cases of one and two would likely have sufficed.

of coefficients, an optional offset, and the lowest (not highest) degree of the polynomial; the latter two both default to zero. In this case the order of the coefficients is from lowest to highest power.

- `ScaledGaussianPdf`: Another Gaussian variant. This one moves its mean by a bias b and scales its width by a scale factor ϵ:

$$P(x; m, \sigma, b, \epsilon) = e^{-\frac{(x+b-m)^2}{2(\sigma(1+\epsilon))^2}}. \tag{23}$$

  This has a somewhat specialised function: It allows fitting Monte Carlo to, for example, a sum of two Gaussians, whose means and widths are then frozen. Then real data can be fit for a common bias and ϵ.

  The constructor takes the observable x, mean m, width σ, bias b and scale factor ϵ.

- `SmoothHistogramPdf`: Another histogram, but this one does smoothing in place of interpolation. That is, suppose the event falls in bin N of a one-dimensional histogram; then the returned value is a weighted average of bins $N-1$, N, and $N+1$. For multidimensional cases the weighted average is over all the neighbouring bins, including diagonals:

$$P(\vec{x}; s; H) = \frac{H(\mathrm{bin}(\vec{x})) + s \sum_{i=\mathrm{neighbours}} \delta i H(i)}{1 + s \sum_{i=\mathrm{neighbours}} \delta i} \tag{24}$$

  where δ_i is zero for bins that fall outside the histogram limits, and one otherwise. The constructor takes the underlying histogram H (which also defines the event vector $\vec{x}$) and the smoothing factor s; notice that if s is zero, the PDF reduces to a simple histogram lookup. The `BinnedDataSet` representing H may be empty; in that case the lookup table should be set later using the `copyHistogramToDevice` method.

- `StepPdf`: Also known as the Heaviside function. Zero up to a point, then 1 after that point:

$$P(x; x_0) = \begin{cases} 0 & x \leq x_0 \\ 1 & x > x_0 \end{cases} \tag{25}$$

  The constructor takes the observable x and threshold x_0.

- `VoigtianPdf`: A convolution of a classical Breit-Wigner and a Gaussian resolution:

$$P(x; m, \sigma, \Gamma) = \int_{-\infty}^{\infty} \frac{\Gamma}{(t-m)^2 - \Gamma^2/4} e^{-\frac{(t-x)^2}{2\sigma^2}} \mathrm{d}t. \quad (26)$$

  The actual implementation is a horrible lookup-table-interpolation; had Lovecraft been aware of this sort of thing, he would not have piffled about writing about mere incomprehensible horrors from the depths of time. The constructor takes the observable x, mean m, Gaussian resolution width σ, and Breit-Wigner width Γ.

5.2 Combination PDFs

These are the tools that allow GooFit to be more than a collection of special cases. The most obvious example is a sum of PDFs - without a class for this, you'd have to write a new PDF every time you added a Gaussian to your fit.

- `AddPdf`: A weighted sum of two or more PDFs. There are two variants, 'extended' and 'unextended'. In the extended version the weights are interpreted as numbers of events, and N PDFs have N weights; in the unextended version the weights are probabilities (i.e., between 0 and 1) and N PDFs have $N-1$ weights, with the probability of the last PDF being 1 minus the sum of the weights of the others.

$$\begin{aligned} P(F_1, \ldots, F_n, w_1, \ldots, w_n) &= w_1 F_1 + \ldots + w_n F_n & (27) \\ P(F_1, \ldots, F_n, w_1, \ldots, w_{n-1}) &= w_1 F_1 + \ldots + w_{n-1} F_{n-1} & (28) \\ &\quad + (1 - w_1 - \ldots - w_{n-1}) F_n. & (29) \end{aligned}$$

  The constructor takes a `vector` of weights w_i and a `vector` of components F_i. If the two `vector`s are of equal length the extended version is used; if there is one more component than weight, the unextended version; anything else is an error. There is also a special-case constructor taking a single weight and two components, to save creating the `vector`s in this common case.

  Note that this PDF overrides the `sumOfNll` method; if an extended `AddPdf` is used as a top-level PDF (that is, sent to `FitManager` for

fitting), an additional term for the number of events will be added to the NLL.

Also note that if the `AddPdf`'s options mask (set by calling `setSpecialMask`) includes `ForceCommonNorm`, the normalisation changes. By default the components are normalised separately, so that

$$P(x;\vec{F},\vec{w}) \;=\; \sum_i \frac{w_i F_i(x)}{\int F_i(x)\mathrm{d}x}, \tag{30}$$

but with `ForceCommonNorm` set, the integral is instead taken at the level of the sum:

$$P(x;\vec{F},\vec{w}) \;=\; \frac{\sum_i w_i F_i(x)}{\int \sum_i w_i F_i(x)\mathrm{d}x}. \tag{31}$$

The difference is subtle but sometimes important.

- `BinTransformPdf`: Returns the global bin of its argument; in one dimension:

$$P(x;l,s) \;=\; \mathrm{floor}\left(\frac{x-l}{s}\right) \tag{32}$$

where l is the lower limit and s is the bin size. The utility of this is perhaps not immediately obvious; one application is as an intermediate step in a `MappedPdf`. For example, suppose I want to model a y distribution with a different set of parameters in five slices of x; then I would use a `BinTransformPdf` to calculate which slice each event is in.

The constructor takes `vector`s of the observables $\vec{x}$, lower bounds $\vec{l}$, bin sizes $\vec{b}$, and number of bins $\vec{n}$. The last is used for converting local (i.e. one-dimensional) bins into global bins in the case of multiple dimensions.

- `CompositePdf`: A chained function,

$$P(x) \;=\; h(g(x)). \tag{33}$$

The constructor takes the kernel function g and the shell function h. Note that only one-dimensional composites are supported - h cannot take more than one argument. The core function g can take any number.

- `ConvolutionPdf`: Numerically calculates a convolution integral

$$P(x; f, g) = f \otimes g = \int_{-\infty}^{\infty} f(t)g(x-t)\mathrm{d}t. \tag{34}$$

  The constructor takes the observable x, model function f, and resolution function g.

  The implementation of this function is a little complicated and relies on caching. There is a variant constructor for cases where several convolutions may run at the same time, eg a `MappedPdf` where all the targets are convolutions. This variant does cooperative loading of the caches, which is a really neat optimisation and ought to work a lot better than it, actually, does. Its constructor takes the observable, model, and resolution as before, and an integer indicating how many other convolutions are going to be using the same cache space.

- `EventWeightedAddPdf`: A variant of `AddPdf`, in which the weights are not fit parameters but rather observables. It otherwise works the same way as `AddPdf`; the constructor takes `vector`s of the weights and components, and it has extended and non-extended variants. Note that you should not mix-and-match; the weights must be either all observables or all fit parameters.

- `MappedPdf`: A function having the form

$$F(x) = \begin{cases} F_1(x) & x_0 \leq x \leq x_1 \\ F_2(x) & x_1 < x \leq x_2 \\ (\ldots) & (\ldots) \\ F_n(x) & x_{n-1} < x \leq x_n \end{cases} \tag{35}$$

  The constructor takes a mapping function m, which returns an index; and a `vector` of evaluation functions $\vec{F}$, so that if m is zero, the PDF returns F_0, and so on. Notice that m does not strictly need to return an integer - in fact the constraints of GooFit force it to return a floating-point number - since `MappedPdf` will round the result to the nearest whole number. The canonical example of a mapping function is `BinTransformPdf`.

- `ProdPdf`: A product of two or more PDFs:

$$P(x;\vec{F}) \;=\; \prod_i F_i(x). \tag{36}$$

The constructor just takes a `vector` of the functions to be multiplied.

`ProdPdf` does allow variable overlaps, that is, the components may depend on the same variable, eg $P(x) = A(x)B(x)$. If this happens, the entire `ProdPdf` object will be normalised together, since in general $\int A(x)B(x)\mathrm{d}x \neq \int A(x)\mathrm{d}x \int B(x)\mathrm{d}x$. However, if any of the components have the flag `ForceSeparateNorm` set, as well as in the default case that the components depend on separate observables, each component will be normalised individually. Some care is indicated when using the `ForceSeparateNorm` flag, and possibly a rethink of why there is a product of two PDFs depending on the same variable in the first place.

5.3 Specialised amplitude-analysis functions

These functions exist mainly for use in a specific physics analysis, mixing in $D^0 \to \pi\pi\pi^0$. Nonetheless, if you are doing a Dalitz-plot analysis, you may find them, and conceivably even this documentation, helpful.

- `DalitzPlotPdf`: A time-independent description of the Dalitz plot as a coherent sum of resonances:

$$P(m_{12}^2, m_{13}^2; \vec{\alpha}) \;=\; \left|\sum_i \alpha_i B_i(m_{12}^2, m_{13}^2)\right|^2 \epsilon(m_{12}^2, m_{13}^2) \tag{37}$$

where α_i is a complex coefficient, B_i is a resonance parametrisation (see `ResonancePdf`, below), and ϵ is a real-valued efficiency function. The constructor takes the squared-mass variables m_{12} and m_{13}, an event index (this is used in caching), a `DecayInfo` object which contains a `vector` of `ResonancePdf`s as well as some global information like the mother and daughter masses, and the efficiency function.

- `DalitzVetoPdf`: Tests whether a point is in a particular region of the Dalitz plot, and returns zero if so, one otherwise. Intended for use as part of an efficiency function, excluding particular regions - canonically

the one containing the $K^0 \to \pi\pi$ decay, as a large source of backgrounds that proved hard to model. The constructor takes the squared-mass variables m_{12} and m_{13}, the masses (contained in `Variable`s) of the mother and three daughter particles involved in the decay, and a `vector` of `VetoInfo` objects. The `VetoInfo` objects just contain a cyclic index (either `PAIR_12`, `PAIR_13`, or `PAIR_23`) and the lower and upper bounds of the veto region.

- `IncoherentSumPdf`: Similar to `DalitzPlotPdf`, but the resonances are added incoherently:

$$P(m_{12}^2, m_{13}^2; \vec{\alpha}) \;=\; \sum_i \left|\alpha_i B_i(m_{12}^2, m_{13}^2)\right|^2 \epsilon(m_{12}^2, m_{13}^2) \qquad (38)$$

  The constructor is the same, but note that the `amp_imag` member of `ResonancePdf` is not used, so the α are in effect interpreted as real numbers.

- `MixingTimeResolution_Aux`: The abstract base class of `TruthResolution_Aux` and `ThreeGaussResolution_Aux`. Represents a parametrisation of the time resolution.

- `ResonancePdf`: Represents a resonance-shape parametrisation, the B_i that appear in the equations for `DalitzPlotPdf`, `IncoherentSumPdf`, and `TddpPdf`. Canonically a relativistic Breit-Wigner. The constructor takes the real and imaginary parts of the coefficient α (note that this is actually used by the containing function), and additional parameters depending on which function the resonance is modelled by:

  - Relativistic Breit-Wigner: Mass, width, spin, and cyclic index. The two last are integer constants. Only spins 0, 1, and 2 are supported.
  - Gounaris-Sakurai parametrisation: Spin, mass, width, and cyclic index. Notice that this is the same list as for the relativistic BW, just a different order.
  - Nonresonant component (ie, constant across the Dalitz plot): Nothing additional.
  - Gaussian: Mean and width of the Gaussian, cyclic index. Notice that the Gaussian takes the mass $m_{12,13,23}$ as its argument, not the squared mass $m_{12,13,23}^2$ like the other parametrisations.

- `TddpPdf`: If the Gaussian is a potato, this is a five-course banquet dinner involving entire roasted animals stuffed with other animals, large dance troupes performing between the courses, an orchestra playing in the background, and lengthy speeches. There will not be a vegetarian option. Without going too deeply into the physics, the function models a decay, eg $D^0 \to \pi\pi\pi^0$, that can happen either directly or through a mixed path $D^0 \to \overline{D^0} \to \pi\pi\pi^0$. (Although developed for the $\pi\pi\pi^0$ case, it should be useful for any decay where the final state is its own anti-state.) The probability of the mixing path depends on the decay time, and quantum-mechanically interferes with the direct path. Consequently the full Time-Dependent Dalitz-Plot (Tddp) amplitude is (suppressing the dependence on squared masses, for clarity):

$$\begin{aligned} P(m_{12}^2, m_{13}^2, t, \sigma_t; x, y, \tau, \vec{\alpha}) &= e^{-t/\tau}\Big(|A+B|^2\cosh(yt/\tau) && (39)\\ &\quad +|A-B|^2\cos(xt/\tau) && (40)\\ &\quad -2\Re(AB^*)\sinh(yt/\tau) && (41)\\ &\quad -2\Im(AB^*)\sin(xt/\tau)\Big) && (42) \end{aligned}$$

  where (notice the reversed masses in the B calculation)

$$A = \sum_i \alpha_i B_i(m_{12}^2, m_{13}^2) \quad (43)$$

$$B = \sum_i \alpha_i B_i(m_{13}^2, m_{12}^2), \quad (44)$$

  convolved with a time-resolution function and multiplied by an efficiency. The implementation involves a large amount of caching of the intermediate B_i values, because these are expected to change slowly relative to the coefficients α (in many cases, not at all, since masses and widths are often held constant) and are relatively expensive to calculate.

  The constructor takes the measured decay time t, error on decay time σ_t, squared masses m_{12}^2 and m_{13}^2, event number, decay information (the same class as in `DalitzPlotPdf`; it also holds the mixing parameters x and y and lifetime τ), time-resolution function, efficiency, and optionally a mistag fraction. A variant constructor takes, instead of a single time-resolution function, a `vector` of functions and an additional observable m_{D^0}; in this case the resolution function used depends on

which bin of m_{D^0} the event is in, and the number of bins is taken as equal to the number of resolution functions supplied.

It is not suggested to try to use this thing from scratch. Start with a working example and modify it gradually.

- `ThreeGaussResolution_Aux`: A resolution function consisting of a sum of three Gaussians, referred to as the 'core', 'tail', and 'outlier' components. The constructor takes the core and tail fractions (the outlier fraction is 1 minus the other two), core mean and width, tail mean and width, and outlier mean and width. Notice that this is a resolution function, so the full probability is found by convolving Gaussians with Equation 39, and this runs to a page or so of algebra involving error functions. It is beyond the scope of this documentation.

- `TrigThresholdPdf`: Intended as part of an efficiency function, modelling a gradual fall-off near the edges of phase space:

$$P(x; a, b, t) = \begin{cases} 1 & d > 1/2 \\ a + (1-a)\sin(d\pi) & \text{otherwise} \end{cases} \quad (45)$$

  where $d = b(x - t)$ or $d = b(t - x)$ depending on whether the function is modelling a lower or upper threshold. The constructor takes the observable x (which will be either m^2_{12} or m^2_{13}), threshold value t, trig constant b, linear constant a, and a boolean which if true indicates an upper threshold. A variant constructor, for modelling a threshold in the "third" Dalitz-plot dimension m^2_{23}, takes both m^2_{12} and m^2_{13}, and an additional mass constant m; it then forms $x = m - m^2_{12} - m^2_{13}$, and otherwise does the same calculation.

- `TruthResolution_Aux`: The simplest possible resolution function, a simple delta spike at zero - i.e., time is always measured perfectly. The constructor takes no arguments at all!