Hydra Documentation

Release 2.0

Antonio Augusto Alves Junior

Mar 19, 2018

Contents:

CHAPTER 1

Introduction

1.1 Hydra framework

Despite the ongoing efforts of modernization, a large fraction of the software used in HEP remain based on legacy. It mostly consists of libraries assembling single threaded, Fortran and C++03 mono-platform routines *[3]*. Concomitantly, HEP experiments keep collecting samples with unprecedented large statistics and data analyses become increasingly complex. Are not rare the situations where computers spend days performing calculations to reach a result, which very often needs re-tune.

On the other hand, computer processors will not increase clock frequency any more in order to reach higher performance. Indeed, the current road-map to improve overall performance is to deploy different levels of concurrency, which for example has been leading to the proliferation of multi-thread friendly and multi-platform environments among HPC data-centers. Unfortunately, HEP software is not completely prepared yet to fully exploit concurrency and to deploy more opportunistic computing strategies.

The Hydra framework proposes a computing model to approach these issues. The Hydra provides collection of parallelized high-level algorithms, addressing some of of typical computing bottlenecks commonly found in HEP, and a set of optimized containers and types, through a modern and functional interface, allowing to enhance HEP software productivity and performance and at same time keeping the portability between NVidia GPUs, multi-core CPUs and other devices compatible with CUDA *[4]*, TBB *[5]* and OpenMP *[6]* computing models.

1.2 Design highlights

Hydra is basically a header-only C++11 template framework organized using a variety of static polymorphism idioms and patterns. This ensure the predictability of the stack at compile time,

which is critical for stability and performance when running on GPUs and minimizes the overhead introduced by the user interface when engaging the actual calculations. Furthermore, the implementation of static polymorphism via extensive usage of templates allows to expose the maximum amount of code to the compiler, in the context in which the code will be used, contributing to activate many compile time optimizations that could not be accessible otherwise. Hydra's interface and implementation details extensively deploys patterns and idioms that enforce thread-safeness and efficient in memory access and management. The following list summarizes some of the main design choices adopted in Hydra:

- Hydra provides a set of optimized STL-like containers that can store multidimensional datasets using *[7]* layout.
- Data handled using iterators and all classes manages resources using RAII idiom.
- The framework is type and thread-safe.
- There is no limitation on the maximum number of dimensions that containers and algorithms can handle.

The types of devices which Hydra can be deployed are classified by back-end type, according with the device compatibility with certain computing models. Currently, Hydra supports four back-ends, which are CPP *[8]*, OpenMP *[6]*, CUDA *[4]* and TBB *[5]*. Code can be dispatched and executed in all supported back-ends concurrently and asynchronously in the same program, using the suitable policies represented by the symbols `hydra::omp::sys` , `hydra::cuda::sys`, `hydra::tbb::sys`, `hydra::cpp::sys` , `hydra::host::sys` and `hydra::device::sys`. Where applicable, these policies define the memory space where resources should be allocated to run algorithms and store data.

For mono-backend applications, source files written using Hydra and standard C++ compile for GPU and CPU just exchanging the extension from .cu to .cpp and one or two compiler flags. So, basically, there is no need to refractory code to deploy different back-ends.

1.3 Basic features

Currently, Hydra provides collection of parallelized high-level algorithms, addressing some computing-intensive tasks commonly found in data analyses in HEP. The available high-level algorithms are listed below,

- Interface to **Minuit2** minimization package *[1]*, allowing to accelerate maximum likelihood fits over multidimensional large data-sets.
- Parallel implementation of the **SPlot** technique, a very popular procedure for statistical unfolding of data distributions *[2]* .
- Phase-space Monte Carlo generation, integration and modeling.
- Multidimensional p.d.f. sampling.
- Parallel function evaluation on multidimensional data-sets.

- Five fully parallelized numerical integration algorithms: Genz-Malik *[9][10]*, self-adaptive and static Gauss-Kronrod quadratures, plain, self-adaptive importance sampling and phase-space Monte Carlo integration.

1.4 How does this manual is organized?

By the time it was written, this manual covers the usage of most of the Hydra features. This manual was written to be read sequentially. The sections are organized by subject and are sorted to make available the functionality described in a given section usable in the next parts.

CHAPTER 2

Functors and C++11 lambdas.

The user's code is passed to Hydra's algorithms through functors and C++11 lambda functions. Hydra adds type information and functionality to functors and lambdas using CRTP idiom. Functors and lambdas are entities not attached to a specific back-end. The signatures conventions adopted for functors and lambdas as well as the added functionality will be discussed in the following lines.

2.1 Functors

In C++, a functor, sometimes also referred as a function object, is any class or structure that overloads the function call operator `òperator()(Args ...x)`. In Hydra, all functors derives from the class template `hydra::BaseFunctor<Functor, ReturnType, NParameters>`. The template parameters are described below:

- `Functor` : the type of the functor.
- `ReturnType`: the type returned by the functor.
- `NParameters`: the number of parameters the functor has.

The user needs only to implement the method `Evaluate(...)` and Hydra will take care of implementing the function call operator. The signature of `Evaluate(...)` depends on the type of data that will be passed. There are two possibilities:

1. The functor is supposed to take as arguments (one-)multidimensional data with the same type. In this case the signature of the `Evaluate(...)` method will be

```
template<typename T>
__host__ __device__
ReturnType Evaluate(unsigned int n , T* x);
```

where `T` is the data type, `n` the number of arguments and `x` a pointer to an array of arguments. The symbols `__host__ __device__` are the necessary to make the functor callable on host and device memory spaces.

2. The functor is supposed to take as arguments multidimensional data with different types, so data will be compacted in a `hydra::tuple` object. In this case the signature of the function call operator will be

```
template<typename T>
__host__ __device__
ReturnType Evaluate(T& x);
```

where T is the dataset entry type, in this case a `hydra::tuple` of arguments.

The parameters are represented by the `hydra::Parameter`. The parameters can be named, store maximum and minimum values and error. The objects of the class `hydra::Parameter` can be instantiated using named field idiom or field list idiom, like this

```
#include <Parameter.h>
#include <string>

auto p1 = hydra::Parameter::Create().Name("p1").Value(0.0).Limits(-
↪1.0, 1.0).Error(0.01);

auto p2 = hydra::Parameter("p1",0.0,0.001,-1.0, 1.0);
```

Hydra does not check uniqueness of the name of the parameters at creation time in any way. It is up to the user to care about the contexts where parameters can have or not the same name. The parameters of a functor are accessible via the `_par[]` subscript operator or invoking the `GetParameter(unsigned int i)` and `GetParameter(const char* name)` functor member function.

As an example, let's consider the Gaussian function with mean μ and sigma σ

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}(\frac{x-\mu}{\sigma})^2}$$

and suppose the corresponding functor will take as arguments data with same type and evaluate the Gaussian on argument set by the template parameter `ArgIndex`.

The corresponding code in Hydra would look like this

```
#include <hydra/Parameters.h>
#include <hydra/Function.h>

template<unsigned int ArgIndex=0>
class Gaussian: public BaseFunctor<Gaussian<ArgIndex>, double, 2> {

using BaseFunctor<Gaussian<ArgIndex>, double, 2>::_par;
```

```
public:

Gaussian()=delete;

Gaussian(Parameter const& mean, Parameter const& sigma ):
BaseFunctor<Gaussian<ArgIndex>, double, 2>({mean, sigma})
{}

__host__ __device__
Gaussian(Gaussian<ArgIndex> const& other ):
BaseFunctor<Gaussian<ArgIndex>, double,2>(other)
{}

__host__ __device__
Gaussian<ArgIndex>&
operator=(Gaussian<ArgIndex> const& other ){

if(this==&other) return  *this;
BaseFunctor<Gaussian<ArgIndex>,double, 2>::operator=(other);
return  *this;

}

template<typename T>
__host__ __device__ inline
double Evaluate(unsigned int, T*x)  const       {

double m2 = (x[ArgIndex] - _par[0])*(x[ArgIndex] - _par[0] );
double s2 = _par[1]*_par[1];

return exp(-m2/(2.0 * s2 ));

}

template<typename T>
__host__ __device__ inline
double Evaluate(T x)  const {

double m2 = ( get<ArgIndex>(x) - _par[0])*(get<ArgIndex>(x) - _
↪par[0] );
double s2 = _par[1]*_par[1];

return exp(-m2/(2.0 * s2 ));

}

};

...
```

```
auto m = hydra::Parameter::Create().Name("mean").Value(0.0).Limits(-
↪1.0, 1.0).Error(0.01);

auto s = hydra::Parameter::Create().Name("sigma").Value(1.0).
↪Limits(0.01, 5.0).Error(0.01);

Gaussian gauss(m, s);

double args_single(1.0);
hydra::tuple<int, double> args_tuple{0, 1.0};
double args_array[2]{0.0, 1.0};

// the following calls produces the same results
std::cout       << gauss(args_single)    << " "
                << gauss1(args_tuple)    << " "
                        << gauss1(2, args_array) << std::endl;
```

Actually, Hydra users will rarely call functors directly. Functors are used to encapsulate user's code that will be called in parallelized calculations by the Hydra algorithms in multi-threaded CPU and GPU environments. **It is user's responsibility care about race conditions and other problems bad coded functors can cause. It is strongly advised to avoid dynamic memory allocation inside functors.**

2.2 C++11 Lambdas

Hydra fully supports C++11 lambdas. Before to pass C++11 lambdas to Hydra's algorithms, users need to wrap it into a suitable Hydra object. This is done invoking the function template `hydra::wrap_lambda()`.

As well as for functors, the signature of the lambda function depends on the type of data that will be passed. There are two possibilities:

1. The functor is supposed to take as arguments data with the same type. In this case the signature of the function call operator will be

```
[=]__host__ __device__(unsigned n, T* x){
 //implementation goes here
};
```

where `T` is the data type, `n` the number of arguments and `x` a pointer to an array of arguments. The symbols `__host__` `__device__` are the necessary to make the lambda callable on host and device memory spaces.

2. The functor is supposed to take as arguments data with different types. In this case the signature of the function call operator will be

```
[=]__host__ __device__(T x){
 //implementation goes here
};
```

where T is the data type, in this case a `hydra::tuple` of arguments.

Hydra can also handle "parametric lambdas". Parametric lambdas are wrapped C++11 lambdas that can hold named parameters (`hydra::Parameters` objecs). The signatures for parametric lambdas are:

1. The functor is supposed to take as arguments data with the same type. In this case the signature of the function call operator will be

```
[=]__host__ __device__(unsigned int np, hydra::Parameters*␣
↪p, unsigned na, T* args)
{
 //implementation goes here
};
```

where `nparams` is the number of parameters, `params` is a pointer to the array of parameters, `T` is the data type, `nargs` the number of arguments and `args` a pointer to the array of arguments. The symbols `__host__ __device__` are the necessary to make the lambda callable on host and device memory spaces.

2. The functor is supposed to take as arguments data with different types. In this case the signature of the function call operator will be

```
[=]__host__ __device__(unsigned int nparams,␣
↪hydra::Parameters* params, T args)
{
 //implementation goes here
};
```

where `nparams` is the number of parameters, `params` is a pointer to the array of parameters and `T` is the data type, in this case, a `hydra::tuple` of arguments.

The following example shows how to wrap a lambda to calculate a Gaussian function capturing the mean and sigma from the lambda's enclosing scope:

```
#include <hydra/FunctorWrapper.h>

...

double mean  = 0.0;
double sigma = 1.0;

auto raw_gaussian = [=] __host__ __device__ (unsigned int nargs,␣
↪double* args){

        double m2 = (x[0] - mean )*(x[0] - mean );
        double s2 = sigma*sigma;

        return exp(-m2/(2.0 * s2 ))/( sqrt(2.0*s2*PI));

};
```

```
auto wrapped_gaussian = hydra::wrap_lambda(raw_gaussian);
```

In the *previous example* the mean and the sigma of the Gaussian can not be changed once the lambda is instantiated. The user can overcome this limitation instantiating a parametric lambda:

```
#include <hydra/FunctorWrapper.h>
#include <hydra/Parameter.h>

...

auto raw_gaussian = [=] __host__ __device__ (unsigned int nparams,␣
↪hydra::Parameters* params,
        unsigned int nargs, double* args) {

        double m2 = (x[0] - params[0] )*(x[0] - params[0] );
        double s2 = params[1]*params[1];

        return exp(-m2/(2.0 * s2 ))/( sqrt(2.0*s2*PI));

};

auto mean = hydra::Parameter::Create().Name("mean").Value(0.0).
↪Limits(-1.0, 1.0).Error(0.01);

auto sigma = hydra::Parameter::Create().Name("sigma").Value(1.0).
↪Limits(0.01, 5.0).Error(0.01);

auto wrapped_gaussian = hydra::wrap_lambda(raw_gaussian, mean,␣
↪sigma);

//set the parameters to different values
wrapped_gaussian.SetParameter(0, 1.0);
wrapped_gaussian.SetParameter(1, 2.0);
```

The `wrapped_gaussian` of the previous example has the same functionality of the functor coded in the example.

Wrapped lambdas, parametric or not, also derives from `hydra::BaseFunctor` and provide the same functionality of the Hydra functors.

CHAPTER 3

Containers

Hydra framework provides one dimensional STL-like vector containers for each supported back-end, aliasing the underlying Thrust types. Beyond this, Hydra implements two native multidimensional containers: `hydra::multivector` and `hydra::multiarray` . In these containers, the data corresponding to each dimension is stored in contiguous memory regions and when the container is traversed, each entry is accessed as a `hydra::tuple`, where each field holds a value corresponding to a dimension. Both classes implement an interface completely compliant with a STL vector and also provides constant and non-constant accessors for the single dimensional data. The container `hydra::multivector` is suitable to store data-sets where the dimensions are represented by entries with different POD types. `hydra::multiarray` is designed to store data-sets where all dimensions are represented by fields with the same type. Data is always copyable across different back-ends and movable between containers on the same back-end.

3.1 One-dimensional containers

Hydra's one-dimensional containers are aliases to the corresponding [Thrust] vectors and defined for each supported back-end. They are:

1. `hydra::device::vector` : storage allocated in the device back-end defined at compile time using the macro `THRUST_DEVICE_SYSTEM`
2. `hydra::host::vector` : storage allocated in the device back-end defined at compile time using the macro `THRUST_HOST_SYSTEM`
3. `hydra::omp::vector` : storage allocated in the [OpenMP] back-end. Usually the CPU memory space.
4. `hydra::tbb::vector` : storage allocated in the [TBB] back-end. Usually the CPU memory space.

5. `hydra::cuda::vector` : storage allocated in the [CUDA] back-end. The GPU memory space.
6. `hydra::cpp::vector` : storage allocated in the [CPP] back-end. Usually the CPU memory

The usage of these containers is extensively documented in the [Thrust] library.

3.2 Multi-dimensional containers

Hydra implements two multidimensional containers:`hydra::multivector` and `hydra::multiarray`. These containers store data using [SoA] layout and provides a STL-vector compliant interface.

The best way to understand how these containers operates is to visualize them as a table, there each row corresponds to a entry and each column to a dimension. The design of `hydra::multivector` and `hydra::multiarray` makes possible to iterate over the container to access a complete row or to iterate over one or more columns to access only the data of interest in a given entry, without to load the entire row.

When the user iterates over the whole container, each entry (row) is returned as a `hydra::tuple`. If the user iterates over one single column, the entries have the type of the column. If two or more columns are accessed, entry's data is returned as again as `hydra::tuple` containing only the elements of interest. Hydra's multi-dimensional containers can hold any type of data per dimension, but there is not real gain using these containers for describing dimensions with non-POD data.

Both containers can store the state of arbitrary objects and perform type conversions on-the-fly, using suitable overloaded iterators and `push_back()` methods.

3.2.1 `hydra::multivector`

`hydra::multivector` templates are instantiated passing the type list corresponding to each dimension via a `hydra::tuple` and the back-end where memory will be allocated. The snippet *below* show how to instantiate a `hydra::multivector` to store four-dimensional data, two columns for integers and two columns for doubles:

```
#include <hydra/device/System.h>
#include<hydra/multivector.h>

...

hydra::multivector<hydra::tuple<int, int, double, double>,␣
↪hydra::device::sys_t> mvector;

for(int i=0; i<10;i++){
        mvector.push_back(hydra::make_tuple( i, 2*i, i, 2*i));
}
```

```
for(auto x:mvector) std::cout << x << std::endl;
```

this will print in stdout something like it :

```
(0, 0, 0.0, 0.0)
(1, 2, 1.0, 2.0)
(2, 4, 2.0, 4.0)
...
(9, 18, 9.0, 18.0)
```

To access the columns the user needs to deploy `hydra::placeholders`: _0, _1, _2,...,_99;

```
#include <hydra/device/System.h>
#include<hydra/multivector.h>
#include<hydra/Placeholders.h>

using namespace hydra::placeholders;

...

hydra::multivector<hydra::tuple<int, int, double, double>,␣
↪hydra::device::sys_t> mvector;

for(int i=0; i<10;i++){
        mvector.push_back(hydra::make_tuple( i, 2*i, i, 2*i));
}

for(auto x = mvector.begin(_1, _3);
                x != mvector.end(_1, _3); x++ )
                        std::cout << *x << std::endl;
```

now in stdout the user will get:

```
(0, 0.0)
(2, 2.0)
(4, 4.0)
...
(18, 18.0)
```

Now suppose that one want to interpret the data stored in `mvector` as a pair of complex numbers, represented by the types `hydra::complex<int>` and `hydra::complex<double>`. It is not necessary to access each field stored in each entry to perform a conversion invoking the corresponding constructors. The next example shows how this can be accomplished in a more elegant way using a lambda function:

```
    #include <hydra/device/System.h>
    #include<hydra/multivector.h>
    #include<hydra/Complex.h>
```

```
    ...

    hydra::multivector<hydra::tuple<int, int, double, double>,␣
↪hydra::device::sys_t> mvector;

    for(int i=0; i<10;i++){
            mvector.push_back(hydra::make_tuple( i, 2*i, i, 2*i));
    }

    auto caster = [] __host__ device__ (hydra::tuple<int, int,␣
↪double, double>& entry )
    {

    hydra::complex<int> c_int(hydra::get<0>(entry), hydra::get<1>
↪(entry));
    hydra::complex<double> c_double(hydra::get<2>(entry), hydra::get
↪<2>(entry));

    return hydra::make_pair(  c_int, c_double );
};

    for(auto x = mvector.begin(caster); x != mvector.end(caster);␣
↪x++ )
            std::cout << *x << std::endl;
```

stdout will look like:

```
((0, 0), (0.0, 0.0))
((1, 2), (1.0, 2.0))
((2, 4), (2.0, 4.0))
...
((9, 18), (9.0, 18.0))
```

3.2.2 hydra::multiarray

`hydra::multiarray` templates are instantiated passing the type and the number of dimensions via and the back-end where memory will be allocated. The snippet *below* show how to instantiate a `hydra::multiarray` to store four-dimensional data, two columns for integers and two columns for doubles:

```
#include <hydra/device/System.h>
#include<hydra/multiarray.h>

...

hydra::multiarray<4, double, hydra::device::sys_t> marray;

for(int i=0; i<10;i++){
```

```
        marray.push_back(hydra::make_tuple( i, 2*i, 4*i, 8*i));
}

for(auto x:marray) std::cout << x << std::endl;
```

this will print in stdout something like it :

```
(0.0, 0.0, 0.0, 0.0)
(1.0, 2.0, 4.0, 8.0)
(2.0, 4.0, 8.0, 16.0)
...
(9.0, 18.0, 36.0, 72.0)
```

To access the columns the user can deploy `hydra::placeholders`: _0, _1, _2... or use `unsigned it` indexes.

```
#include <hydra/device/System.h>
#include<hydra/multiarray.h>
#include<hydra/Placeholders.h>

using namespace hydra::placeholders;

...

hydra::multiarray<4, double, hydra::device::sys_t> marray;

for(int i=0; i<10;i++){
        marray.push_back(hydra::make_tuple( i, 2*i, i, 2*i));
}

for(auto x = marray.begin(_1, _3);
                x != marray.end(_1, _3); x++ )
                        std::cout << *x << std::endl;
```

now in stdout the user will get:

```
(0.0, 0.0)
(2.0, 8.0)
(4.0, 16.0)
...
(18.0, 72.0)
```

Now suppose that one want to interpret the data stored in mvector as a pair of complex numbers, represented by the types `hydra::complex<double>` and `hydra::complex<double>`. It is not necessary to access each field stored in each entry to perform a conversion invoking the corresponding constructors. The next example shows how this can be accomplished in a more elegant way using a lambda function:

```
    #include <hydra/device/System.h>
    #include<hydra/multiarray.h>
```

```
    #include<hydra/Complex.h>

    ...

    hydra::multiarray<4, double, hydra::device::sys_t> marray;

    for(int i=0; i<10;i++){
            marray.push_back(hydra::make_tuple( i, 2*i, i, 2*i));
    }

    auto caster = [] __host__ device__ (hydra::tuple<double, double,
↪ double, double>& entry ){

    hydra::complex<double> c1(hydra::get<0>(entry), hydra::get<1>
↪(entry));
    hydra::complex<double> c2(hydra::get<2>(entry), hydra::get<2>
↪(entry));
    return hydra::make_pair(  c1, c2);

};

    for(auto x = marray.begin(caster); x != marray.end(caster); x++␣
↪)
            std::cout << *x << std::endl;
```

stdout will look like:

```
((0, 0), (0.0, 0.0))
((1, 2), (1.0, 2.0))
((2, 4), (2.0, 4.0))
...
((9, 18), (9.0, 18.0))
```

CHAPTER 4

Histograms

Hydra implements two classes dedicated to calculate multidimensional histograms in parallel. One class for dense histograms and other for sparse histograms. These classes provide only the basic functionality to calculate the histogram using one of the supported parallel back-ends. Once calculated, the histogram contents can be exported to external libraries, like ROOT, for drawing etc.

The histograms classes does not process event-by-event. They takes iterators pointing to containers storing the data and process it at once. This approach is orders of magnitude more efficient than iterate over the container and histogram in entry-by-entry basis.

4.1 Binning convention

In Hydra, a histogram with N bins is stored in a array with length N+2. In range contents are indexed starting from 0 to N-1. Underflow contents are stored in bin N and overflow contents are stored in bin N+1.

4.2 Global and dimensional binning

The histogram contents is organized in a linear array of length N+2, where N is total number of bins, obtained multiplying the number of bins configured in for each dimension. The conversion between global bin number and dimensional bin numbers is performed by the methods `GetBin(...)` and `GetIndexes(...)`, implemented in both classes. The internal indexing convention used in Hydra in general does not match the one used in other libraries and interfaces. Users are advised to always export the histogram contents using the bin numbers per bin.

4.3 Dense histograms

Dense histograms store all bins, including ones with zero content. In Hydra, they are represented by the class `hydra::DenseHistogram<Type, NDimensions, Backend>`, where `NDimensions` is the number of dimensions, `Type` is the type of the histogram's values and `Backend` is memory space where the histogram is allocated.

The code snippet below shows how to instantiate and fill a dense histogram in Hydra:

```
#include <hydra/device/System.h>
#include <hydra/multiarray.h>
#include <hydra/DenseHistogram.h>
#include <array>

...

hydra::multiarray<4, double, hydra::device::sys_t> mvector;

...
// fill mvector with the data of interest...
...

//histogram ranges
std::array<double, 4>max{ 1.0,  2.0,  3.0,  4.0};
std::array<double, 4>min{-1.0, -2.0, -3.0, -4.0};

//bins per dimension
std::array<size_t, 3> nbins{10, 20, 30, 40};

//create histogram
hydra::DenseHistogram<3, double> Histogram(nbins, min, max);

Histogram.Fill( mvector.begin(), mvector.end());

//getting bin content [0, 2, 3, 1]
Histogram.GetBinContent({0, 2, 3, 1});
```

4.4 Sparse histograms

Sparse histograms store only bins with non-zero content. In Hydra, they are represented by the class `hydra::SparseHistogram<Type, NDimensions, Backend>`, where `NDimensions` is the number of dimensions, `Type` is the type of the histogram's values and `Backend` is memory space where the histogram is allocated.

```
#include <hydra/device/System.h>
#include <hydra/multiarray.h>
#include <hydra/SparseHistogram.h>
#include <array>
```

```
...

hydra::multiarray<4, double, hydra::device::sys_t> mvector;

...
// fill mvector with the data of interest...
...

//histogram ranges
std::array<double, 4>max{ 1.0,  2.0,  3.0,  4.0};
std::array<double, 4>min{-1.0, -2.0, -3.0, -4.0};

//bins per dimension
std::array<size_t, 3> nbins{10, 20, 30, 40};

//create histogram
hydra::SparseHistogram<3, double> Histogram(nbins, min, max);

Histogram.Fill( mvector.begin(), mvector.end());

//getting bin content [0, 2, 3, 1]
Histogram.GetBinContent({0, 2, 3, 1});
```

CHAPTER 5

Random number generation and PDF sampling

The generation of random numbers and sampling of multidimensional PDFs is supported in Hydra through the class `hydra::Random<typename Engine>`, where `Engine` is the random number generator engine. The only argument of the `hydra::Random` constructor is the seed of random number generator. There are four random number engines available

1. `hydra::minstd_rand0`: implements a version of the Minimal Standard random number generation algorithm.
2. `hydra::minstd_rand`: implements a version of the Minimal Standard random number generation algorithm.
3. `hydra::ranlux24`: RANLUX level-3 random number generation algorithm.
4. `hydra::ranlux48`: RANLUX level-4 random number generation algorithm.
5. `hydra::taus88`: L'Ecuyer's 1996 three-component Tausworthe random number generator.

The default random number generation engine is `hydra::minstd_rand0`, which produces pseudo-random numbers with quality appropriated for most applications. This class provides methods that take iterators pointing to containers that will be filled with random numbers distributed according the requested distributions. If an explicit back-end policy is passed, the generation is parallelized in the corresponding back-end, otherwise the class will process the random number generation in the back-end the containers is allocated.

5.1 Sampling basic distributions

`hydra::Random` defines four methods to generate predefined one-dimensional. These methods are summarized below, where begin and end are iterators pointing to the range that will filled with random numbers. The other parameters represent the standard definitions:

1. `hydra::Random::Gauss( mean, sigma, begin, end)` for Gaussian distribution.
2. `hydra::Random::Exp(tau, begin, end)` for exponential distribution.
3. `hydra::Random::Uniform(min, max, begin, end)` for an uniform distribution.
4. `hydra::Random::BreitWigner(mean, width, begin, end)` for non-relativistic Breit-Wigner distribution.

The example below show how to use these methods

```
#include <hydra/device/System.h>
#include <hydra/Random.h>

...

hydra::Random<> Generator(4598635);
hydra::device::vector<double>  data(1e6);

//uniform distribution in the interval [-5,5]
Generator.Uniform(-5.0, 5.0, data.begin(), data.end());

//Gaussian distribion with mean=0 and sigma =1
Generator.Gauss(0.0, 1.0, data.begin(), data.end());

//exponential distribion with tau=1
Generator.Exp(1.0, data.begin(), data.end());

//Breit-Wigner with mean 2.0 width 0.2
Generator.BreitWigner(2.0, 0.2, data_d.begin(), data_d.end());
```

5.2 Multidimensional PDF sampling

The class `hydra::Random` also supports the sampling of multidimensional probability density functions (PDF) through the method `hydra::Random::Sample(begin, end, min, max, functor)`, where `min` and `max` are static arrays or `std::array` objects representing the limits of the multidimensional region. `functor` is a Hydra functor representing the PDF.

The PDFs are sampled using a parallel version of the accept-reject method. The
`hdra::Random::Sample` returns a `hydra::GenericRange` object pointing to a range filled with the sampled numbers. The PDF sampling

is processed filling the container with random numbers and reordering it to reproduce the shape of the PDF, no memory reallocation is performed during this process. The range returned by the method points to a sub-set of the original container, the size of this range depends on the efficiency of the accept-reject for the given PDF.

The code below shows how to sample a three-dimensional PDF

```
#include <hydra/device/System.h>
#include <hydra/Random.h>
#include <hydra/FunctionWrapper.h>

...

double mean   =  0.0;
double sigma  =  1.0;

auto gaussian = hydra::wrap_lambda(
        [=] __host__ __device__ (unsigned int n,double* x ){

                double g = 1.0;

                for(size_t i=0; i<3; i++){
                        double m2 = (x[i] - mean )*(x[i] - mean );
                        double s2 = sigma*sigma;
                        g *= exp(-m2/(2.0 * s2 ))/( sqrt(2.
↪0*s2*PI));
                }

                return g;
        }
);

std::array<double, 3> max{ 6.0,  6.0,  6.0};
std::array<double, 3> min{-6.0, -6.0, -6.0};

hydra::multiarray<3, double, hydra::device::sys_t> data;
auto range = Generator.Sample(data.begin(),  data.end(), min, max,␣
↪gaussian);
```

CHAPTER 6

Phase-space Monte Carlo

Phase-Space Monte Carlo simulates the kinematics of a particle with a given four-momentum decaying to a n-particle final state, without intermediate resonances. Samples of phase-space Monte Carlo events are widely used in HEP studies where the calculation of phase-space volume is required as well as a starting point to implement and describe the properties of models with one or more resonances or even to simulate the response of the detector to decay's products [James].

Hydra provides an implementation of the Raubold-Lynch method [James] and can generate the full kinematics of decays with any number of particles in the final state. Sequential decays, evaluation of models, production of weighted and unweighted samples and many other features are also supported.

6.1 Decays and decay chains

The four-vector of the generated final-state particles are stored in the dedicated vector-like container `hydra::Decays<N,BACKEND>` where `N` is the number of particles in the final state and `BACKEND``is the memory space where allocate the storage.``hydra::Decays<N,BACKEND>` can be aggregated to describe sequential decays using `hydra::Chains<Decays...>` objects.

Both classes are iterable, but the `hydra::Chains<Decays...>` container does not implement a full vector-like interface. Pre-allocated `hydra::Decays<N,BACKEND>` can not be added to a `hydra::Chains<Decays...>`.

6.2 Phase-space Monte Carlo generator.

The phase-space Monte Carlo generator is represented by the class `hydra::PhaseSpace<N,RNG>`, where is the number of particles and `RGN` is underlying random number generator to be used. The constructor of the `hydra::PhaseSpace` takes as parameter an array with the masses of the final state particles. The decays are generated invoking the overloaded `hydra::PhaseSpace::Generate(...)` method. This method can take a `hydra::Vector4R`, describing momentum of a only mother particle or iterators pointing for a container storing a list of mother particles and the iterators pointing to the `hydra::Decays<N,BACKEND>` container that will hold the generated final states. If an explicit policy policy is passed, the generation is parallelized in the corresponding back-end, otherwise the class will process the random number generation in the back-end where the containers are allocated.

6.2.1 Generating one-level decays

The code below shows how to generate a sample of 20 million $B^0 \to J/\psi K^+\pi^-$ decays and fill a Dalitz's plot, i.e. a histogram $M^2(J/\psi\pi^-) vs M^2(K^+\pi^-)$:

```
#include <hydra/Types.h>
#include <hydra/Vector4R.h>
#include <hydra/PhaseSpace.h>
#include <hydra/device/System.h>
#include <hydra/Decays.h>

...

size_t  nentries  = 20e6;        // number of events to generate
double B0_mass    = 5.27955;     // B0 mass
double Jpsi_mass  = 3.0969;      // J/psi mass
double K_mass     = 0.493677;    // K+ mass
double pi_mass    = 0.13957061; // pi mass

// mother particle
hydra::Vector4R B0(B0_mass, 0.0, 0.0, 0.0);

// decays container
hydra::Decays<3, hydra::device::sys_t > Events(nentries);

hydra::PhaseSpace<3> phsp{Jpsi_mass, K_mass, pi_mass};

// generate the final state particles
phsp.Generate(B0, Events.begin(), Events.end());

// functor to calculate Dalitz variables
auto dalitz_calculator = hydra::wrap_lambda(
        [=] __host__ __device__ (unsigned int np, hydra::Vector*␣
↪particles){
```

```
                hydra::Vector4R Jpsi = event[0];
                hydra::Vector4R K    = event[1];
                hydra::Vector4R pi   = event[2];

                double M2_Jpsi_pi = (Jpsi + pi).mass2();
                double M2_Kpi     = (K + pi).mass2();

                return hydra::make_tuple( M2_Jpsi_pi, M2_Kpi);
        }
);

//
auto particles        = Events.GetUnweightedDecays();

// use an smart-range to calculate the Dalitz variables
// without have to store it. ;)
auto dalitz_variables = hydra::make_range( particles.begin(),␣
↪particles.end(), dalitz_calculator);

// get the event's weights
auto dalitz_weights   = Events.GetWeights();

// instantiate 2D histogram
hydra::DenseHistogram<2, double> Hist_Dalitz({100,100}, {pow(Jpsi_
↪mass + pi_mass,2), pow(K_mass + pi_mass,2)},
        {pow(B0_mass - K_mass,2)   , pow(B0_mass - Jpsi_mass,2)} );

//fill the histogram
Hist_Dalitz.Fill(dalitz_variables.begin(), dalitz_variables.end(),␣
↪dalitz_weights.begin());

...
```

In the previous example, the user can foward the `Hydra::DenseHistogram` to ROOT and draw it.

6.2.2 Generating sequential decays

The code below shows how to generate a sample of 20 million decay chains $B^0 \to J/\psi K^+\pi^-$ with $J/\psi \to \mu^+\mu^-$.

The first step to process the decay chain is to the generate the decays $B^0 \to J/\psi K^+\pi^-$, then the list of J/ψ candidates is passed to the instance of `hydra::PhaseSpace` to generate the $J/\psi \to \mu^+\mu^-$ corresponding to each J/ψ mother. Notice that the decay events stored in a given chain are accessed using a `hydra::placeholder`.

```
#include <hydra/Types.h>
#include <hydra/Vector4R.h>
```

```
#include <hydra/PhaseSpace.h>
#include <hydra/device/System.h>
#include <hydra/Chains.h>
#include <hydra/Placeholders.h>

...

using namespace hydra::placeholders;

...

size_t nentries   = 20e6;          // number of events to generate
double B0_mass    = 5.27955;       // B0 mass
double Jpsi_mass  = 3.0969;        // J/psi mass
double K_mass     = 0.493677;      // K+ mass
double pi_mass    = 0.13957061;    // pi mass
double mu_mass    = 0.1056583745 ;// mu mass


// mother particle
hydra::Vector4R B0(B0_mass, 0.0, 0.0, 0.0);

// create PhaseSpace object for B0 -> K pi J/psi
hydra::PhaseSpace<3> phsp_B2JpsiKpi{Jpsi_mass, K_mass, pi_mass };

// create PhaseSpace object for J/psi -> mu+ mu-
hydra::PhaseSpace<2> phsp_Jpsi2mumu{mu_mass , mu_mass};

//      allocate memory to hold the final states particles
auto Events = hydra::make_chain<3,2>(hydra::device::sys, nentries);

//generate the final state particles for B0 -> K pi J/psi
phsp_B2JpsiKpi.Generate(B0, Events.GetDecay(_0).begin(),
                        Events.GetDecay(_0).end());

//pass the list of J/psi to generate the final
//state particles for J/psi -> mu+ mu-
phsp_Jpsi2mumu.Generate(Events.GetDecay(_0).GetDaughters(0).begin(),
         Events.GetDecay(_0).GetDaughters(0).end(),
          Events.GetDecay(_1).begin());
```

6.3 Other features

The classes of the phase-space module provides many other functionality. The list below summarizes some of them:

- Calculate the mean and the variance of a functor over a phase-space without the need to generate and store events.

- Evaluate functors and stored the result without the need to generate and store events.
- Unweight and re-weight events stored in `hydra::decay` objects to match .
- Access single particle's `Vector4R` or its components of events stored in `hydra::decay` objects and interact with it.

For brevity, the user is adivesed to look the doxygen documentation and the examples to learn what is available and how to deploy it.

CHAPTER 7

Numerical integration

Numerical integration of multidimensional functions is not easy. Numerical integration algorithms tend to be involved and resource hungry, since they demand a large number of evaluations of the integrand. Indeed, it is common to express the efficiency of a given algorithm in terms of the minimum required number of integrand evaluations to achieve given precision for the integral estimation.

The best strategy to perform numerical integration largely depends on the number of dimensions of the integration region and on the features shown by the integrand in this region. Given the detailed information about integrand behavior is usually not available, the numerical integration algorithms need to handle situations specializing routines based on broad properties of the integrand, such as the presence or the absence of narrow peaks, periodicity etc. Highly specialized algorithms, optimized to handle only a given class of problems tend to be more efficient, but on the other hand, such approaches are usually not applicable to different class of problems. In other words, on the field of numerical integration, flexibility usually comes at expenses of efficiency. The basics of numerical integration, followed by comprehensive list of references on the subject, can be found in the Wikipedia pages https://en.wikipedia.org/wiki/Numerical_integration and https://en.wikipedia.org/wiki/Monte_Carlo_integration .

Hydra provides a set of parallelized implementations for generic and popular algorithms to compute one- and multidimensional numerical integration. Hydra parallelizes the calls to the integrand in a collection of threads in requested back-end. The algorithms share the same basic interface, manage resources using RAII idiom and estimate the integral and the associated error. Hydra also supports analytical integration, which should be implemented through functors.

7.1 Gauss-Kronrod quadrature

See also:

Good didactic introduction on Gauss-Kronrod quadrature can be found in the Wikipedia page https://en.wikipedia.org/wiki/Gauss-Kronrod_quadrature_formula.

The class `hydra::GaussKronrodQuadrature<NRULE, NBIN, Backend>` implements a non-adaptive procedure which divides the integration interval in `NBIN` sub-intervals and applies to each sub-interval a fixed Gauss-Kronrod rule of order `NRULE`. `hydra::GaussKronrodQuadrature<NRULE, NBIN, Backend>` allows the fast integration of smooth one-dimensional functions. The code snippet below show how to use this quadrature to calculate the integral of a Gaussian function:

```
#include <hydra/GaussKronrodQuadrature.h>
#include <hydra/FunctionWrapper.h>
#include <hydra/device/System.h>

...

//integration region limits
double  min  = -6.0;
double  max =  6.0;

//Gaussian parameters
double mean  = 0.0;
double sigma = 1.0;


//wrap the lambda
auto gaussian = hydra::wrap_lambda(
[=] __host__ __device__ (unsigned int n, double* x ){

        double m2 = (x[0] - mean )*(x[0] - mean );
        double s2 = sigma*sigma;
        double f = exp(-m2/(2.0 * s2 ))/( sqrt(2.0*s2*PI));

        return f;
} );

...

// 61- degree quadrature
hydra::GaussKronrodQuadrature<61,100, hydra::device::sys_t> GKQ61_
↪d(min,  max);

auto result = GKQ61_d.Integrate(gaussian);

std::cout << "Result: " << result.first << "  +-  " << result.
↪second <<std::endl
```

7.2 Self-Adaptive Gauss-Kronrod quadrature

See also:

Good didactic introduction on Gauss-Kronrod quadrature can be found in the Wikipedia page https://en.wikipedia.org/wiki/Gauss-Kronrod_quadrature_formula.

The class `hydra::GaussKronrodAdaptiveQuadrature<NRULE, NBIN, Backend>` implements a self-adaptive algorithm which initially divides the integration interval in `NBIN` sub-intervals and applies to each sub-interval a Gauss-Kronrod rule of order `NRULE`. The algorith selects the interval with larger relative error in the integral estimation and re-applies the procedure. The algorithme keeps performing this loop until the integram estimation reaches the requested maximum error level.

`hydra::GaussKronrodQAdaptiveuadrature<NRULE, NBIN, Backend>` performs less calls to the integrand and is best suitable for very featured and expensive functions. The code snippet below show how to use this quadrature to calculate the integral of a Gaussian function:

```
    #include <hydra/GaussKronrodAdaptiveQuadrature.h>
    #include <hydra/FunctionWrapper.h>
    #include <hydra/device/System.h>

    ...

    //integration region limits
    double  min  = -6.0;
    double  max  =  6.0;
double max_error = 1e-6;

    //Gaussian parameters
    double mean  = 0.0;
    double sigma = 1.0;


    //wrap the lambda
    auto gaussian = hydra::wrap_lambda(
    [=] __host__ __device__ (unsigned int n, double* x ){

            double m2 = (x[0] - mean )*(x[0] - mean );
            double s2 = sigma*sigma;
            double f = exp(-m2/(2.0 * s2 ))/( sqrt(2.0*s2*PI));

            return f;
    } );

    ...

    // 61- degree quadrature
    hydra::GaussKronrodQuadrature<61,10, hydra::device::sys_t>␣
↪GKQ61(min,  max, max_error);
```

```
    auto result = GKQ61.Integrate(gaussian);

    std::cout << "Result: " << result.first << "  +-  " << result.
↪second <<std::endl
```

7.3 Genz-Malik multidimensional quadrature

This method implements a polynomial interpolatory rule of degree 7, which integrates exactly all monomials ${x_1}^{k_1}, {x_2}^{k_2}...{x_n}^{k_d}$ with $\sum k_i \leq 7$ and fails to integrate exactly at least one monomial of degree 8. In the [Genz-Malik] multidimensional quadrature, all integration nodes are inside integration domain and $2^d + 2d^2 + 2d + 1$ integrand evaluations are required to integrate a function in a rectangular hypercube with d dimensions. Due the fast increase in the number of evaluations as a function of the dimension, this method is most advantageous for problems with d < 10 and is superseded for high-dimensional integrals by Monte Carlo based methods. A degree 5 rule embedded in the degree 7 rule is used for error estimation, in a such way that no additional integrand evaluations are necessary.

The class template `hydra::GenzMalikQuadrature<N, BackendPolicy >` implements a static version of Genz-Malik multidimensional quadrature. This version divides the ‘‘N‘‘dimensional integration region in a series of sub-regions, according the configuration, passed by the user and applies the rule to each sub-region.

The code snippet below shows to use the `hydra::GenzMalikQuadrature<N, BackendPolicy >` class to integrate a five-dimensional Gaussian distribution. In this example each dimension is divided in 10 segments, resulting in 10^5 sub-regions.

```
#include <hydra/GaussKronrodAdaptiveQuadrature.h>
#include <hydra/FunctionWrapper.h>
#include <hydra/device/System.h>

...

//number of dimensions (user can change it)
constexpr size_t N = 5;

//integration region limits
double  min[N];
double  max[N];
size_t  grid[N];

//5D Gaussian parameters
double mean  = 0.0;
double sigma = 1.0;

//set Gaussian parameters and
//integration region limits
for(size_t i=0; i< N; i++){
```

```
        min[i]    = -6.0;
        max[i]    =  6.0;
        grid[10] =  10;
}

//wrap the lambda
auto gaussian = hydra::wrap_lambda( [=] __host__ __device__␣
↪(unsigned int n, double* x ){

        double g = 1.0;
        double f = 0.0;

        for(size_t i=0; i<N; i++){

                double m2 = (x[i] - mean )*(x[i] - mean );
                double s2 = sigma*sigma;
                f = exp(-m2/(2.0 * s2 ))/( sqrt(2.0*s2*PI));
                g *= f;
        }

        return g;
});

hydra::GenzMalikQuadrature<N, hydra::device::sys_t> GMQ(min, max,␣
↪grid);

auto result = GMQ.Integrate(gaussian);

std::cout << "Result: " << result.first << "  +-  " << result.
↪second <<std::endl
```

7.4 Plain Monte Carlo

The plain Monte Carlo algorithm samples points randomly from the integration region to estimate the integral and its error. Using this algorithm the estimate of the integral E(f; N) for N randomly distributed points x_i is given by,

$$E(f;N) = V < f >= (V/N)\sum_{i}^{N} f(x_i)$$

where V is the volume of the integration region. The error on this estimate $\sigma(E;N)$ is calculated from the estimated variance of the mean,

$$\sigma^2(E;N) = (V^2/N^2)\sum_{i}^{N}(f(x_i) - < f >)^2$$

For large N this variance decreases asymptotically as $Var(f)/N$, where $Var(f)$ is the true variance of the function over the integration region. The error estimate itself should decrease

as $\sigma(f)/\sqrt{N}$, which implies that to reduce the error by a factor of 10, a 100-fold increase in the number of sample points is required.

Hydra implements the plain Monte Carlo method in the class `hydra::Plain<N, BackendPolicy>`, where N is the number of dimensions and `BackendPolicy` is the back-end to parallelize the calculation.

The following code snippet shows to use the `hydra::Plain<N, BackendPolicy >` class to integrate a five-dimensional Gaussian distribution performing 100

```
#include <hydra/FunctionWrapper.h>
#include <hydra/device/System.h>
#include <hydra/Plain.h>

...

//number of dimensions (user can change it)
constexpr size_t N = 5;

//integration region limits
double  min[N];
double  max[N];
size_t  ncalls = 1e6;

//5D Gaussian parameters
double mean  = 0.0;
double sigma = 1.0;

//set Gaussian parameters and
//integration region limits
for(size_t i=0; i< N; i++){
        min[i]   = -6.0;
        max[i]   =  6.0;
}

//wrap the lambda
auto gaussian = hydra::wrap_lambda( [=] __host__ __device__
↪(unsigned int n, double* x ){

        double g = 1.0;
        double f = 0.0;

        for(size_t i=0; i<N; i++){

                double m2 = (x[i] - mean )*(x[i] - mean );
                double s2 = sigma*sigma;
                f = exp(-m2/(2.0 * s2 ))/( sqrt(2.0*s2*PI));
                g *= f;
        }

        return g;
});
```

```
hydra::Plain<N, hydra::device::sys_t> PlainMC(min, max, ncalls);

auto result = PlainMC.Integrate(gaussian);

std::cout << "Result: " << result.first << "  +-  " << result.
↪second <<std::endl
```

7.5 Self-adaptive importance sampling (Vegas)

Note: from GSL's Manual, chapter 'Monte Carlo integration' https://www.gnu.org/software/gsl/manual/html_node/VEGAS.html :

> The VEGAS algorithm of [Lepage] is based on importance sampling. It samples points from the probability distribution described by the function $|f|$, so that the points are concentrated in the regions that make the largest contribution to the integral.
>
> In general, if the Monte Carlo integral of f is sampled with points distributed according to a probability distribution described by the function g, we obtain an estimate $E_g(f; N)$,
>
> $$E_g(f; N) = E(f/g; N)$$
>
> with a corresponding variance,
>
> $$Var_g(f; N) = Var(f/g; N).$$
>
> If the probability distribution is chosen as $g = |f|/I(|f|)$ then it can be shown that the variance {Var}_g(f; N) vanishes, and the error in the estimate will be zero. In practice it is not possible to sample from the exact distribution g for an arbitrary function, so importance sampling algorithms aim to produce efficient approximations to the desired distribution.
>
> The VEGAS algorithm approximates the exact distribution by making a number of passes over the integration region while histogramming the function f. Each histogram is used to define a sampling distribution for the next pass. Asymptotically this procedure converges to the desired distribution. In order to avoid the number of histogram bins growing like K^d the probability distribution is approximated by a separable function: $g(x_1, x_2, ...) = g_1(x_1)g_2(x_2)...$ so that the number of bins required is only K_d. This is equivalent to locating the peaks of the function from the projections of the integrand onto the coordinate axes. The efficiency of VEGAS depends on the validity of this assumption. It is most efficient when the peaks of the integrand are well-localized. If an integrand can be rewritten in a form which is approximately separable this will increase the efficiency of integration with VEGAS.
>
> ...

The implementation of VEGAS in Hydra parallelizes the Monte Carlo generation, the function calls and the computing of the result of each iteration. The algorithm is implemented in the `hydra::Vegas<N, BackendPolicy>`. The auxiliary class `hydra::VegasState<N, BackendPolicy>` manages the resources and configuration necessary to perform the integration. The code snippet below shows how to use the VEGAS algorithm to integrate five-dimensional Gaussian distribution:

```
#include <hydra/Vegas.h>
#include <hydra/FunctionWrapper.h>
#include <hydra/device/System.h>

...

//number of dimensions (user can change it)
constexpr size_t N = 5;

//integration region limits
double  min[N];
double  max[N];
size_t  ncalls = 1e5;

//5D Gaussian parameters
double mean  = 0.0;
double sigma = 1.0;

//set Gaussian parameters and
//integration region limits
for(size_t i=0; i< N; i++){
        min[i]    = -6.0;
        max[i]    =  6.0;
}

//wrap the lambda
auto gaussian = hydra::wrap_lambda(
        [=] __host__ __device__ (unsigned int n, double* x )
↪{

                double g = 1.0;
                double f = 0.0;

                for(size_t i=0; i<N; i++){

                        double m2 = (x[i] - mean )*(x[i] -␣
↪mean );

                        double s2 = sigma*sigma;
                        f = exp(-m2/(2.0 * s2 ))/( sqrt(2.
↪0*s2*PI));

                        g *= f;
                }
```

```
                return g;
        }
);

//vegas integrator
hydra::Vegas<N,  hydra::device::sys_t> Vegas(min, max,␣
↪ncalls);

//configuration
Vegas.GetState().SetVerbose(-2);
Vegas.GetState().SetAlpha(1.5);
Vegas.GetState().SetIterations( iterations );
Vegas.GetState().SetUseRelativeError(1);
Vegas.GetState().SetMaxError( max_error );
Vegas.GetState().SetCalls( calls );
Vegas.GetState().SetTrainingCalls( calls/10 );
Vegas.GetState().SetTrainingIterations(2);

auto result = Vegas_d.Integrate(gaussian);
std::cout << "Result: " << result.first << "  +-  " <<␣
↪result.second <<std::endl
```

7.6 Implementing analytical integration

Hydra supports analysical integration as well. To integrate functions analytically the user needs to implement the integral formula in a suitable functor `Functor` deriving from the class `hydra::Integrator<Functor>`. Analytical integration is not parallelized.

CHAPTER 8

Parameter estimation

The best Minuit description can be found on it's own user's manual *[1]* :

> Minuit is conceived as a tool to find the minimum value of a multi-parameter function, usually called "FCN", and analyze the shape of this function around the minimum. The principal application is foreseen for statistical analysis, working on chi-square or log-likelihood functions, to compute the best-fit parameter values and uncertain- ties, including correlations between the parameters. It is especially suited to handle difficult problems, including those which may require guidance in order to find the correct solution.
>
> —Minuit User's Guide, Fred James and Matthias Winkler, June 16, 2004 - CERN, Geneva.

Hydra implements an interface to Minuit2 that parallelizes the FCN calculation. This dramatically accelerates the calculations over large data-sets. Hydra normalizes the pdfs on-the-fly using analytical or numerical integration algorithms provided by the framework and handles data using iterators.

Hydra also provides an implementation of SPlot *[2]*, a very popular technique for statistical unfolding of data distributions.

8.1 Defining PDFs

In Hydra, PDFs are represented by the `hydra::Pdf<Functor, Integrator>` class template and is defined binding a positive defined functor and a integrator. PDFs can be conveniently built using the template function `hydra::make_pdf( pdf, integrator)`. The snippet below shows how wrap a parametric lambda representing a Gaussian and bind it to a Gauss-Kronrod integrator, to build a pdf object:

```
#include <hydra/device/System.h>
#include <hydra/FunctionWrapper.h>
#include <hydra/Pdf.h>
#include <hydra/Parameter.h>
#include <hydra/GaussKronrodQuadrature.h>

...

std::string Mean("Mean");       // mean of gaussian
std::string Sigma("Sigma"); // sigma of gaussian

hydra::Parameter  mean_p  = hydra::Parameter::Create()
        .Name(Mean)
        .Value(0.5)
        .Error(0.0001)
        .Limits(-1.0, 1.0);

hydra::Parameter  sigma_p = hydra::Parameter::Create()
        .Name(Sigma)
        .Value(0.5)
        .Error(0.0001)
        .Limits(0.01, 1.5);

//wrap a parametric lambda
auto gaussian = hydra::wrap_lambda( [=] __host__ __device__
 ↪(unsigned int npar,
        const hydra::Parameter* params,  unsigned int narg, double*
 ↪x ){

        double m2 = (x[0] -  params[0])*(x[0] - params[0] );
        double s2 = params[1]*params[1];

        return exp(-m2/(2.0 * s2 ))/( sqrt(2.0*s2*PI));
}, mean_p, sigma_p);


double min   = -5.0;  double max   =  5.0;

//numerical integral to normalize the pdf
hydra::GaussKronrodQuadrature<61,100, hydra::device::sys_t>
 ↪GKQ61(min,  max);

//build the PDF
auto PDF = hydra::make_pdf(gaussian, GKQ61 );

...
```

It is also possible to represent models composed by the sum of two or more PDFs using the class templates `hydra::PDFSumExtendable<Pdf1, Pdf2,...>` and `hydra::PDFSumNonExtendabl<Pdf1, Pdf2,...>` . Given N normalized pdfs F_i

, theses classes define objects representing the sum

$$F_t = \sum_i^N c_i \times F_i$$

The coefficients c_i can represent fractions or yields. If the number of coefficients is equal to the number of PDFs, the coefficients are interpreted as yields and `hydra::PDFSumExtendable<Pdf1, Pdf2,...>` is used. If the number of coefficients is $(N-1)$, the class template `hydra::PDFSumNonExtendabl<Pdf1, Pdf2, ...>` is used and the coefficients are interpreted as fractions defined in the interval [0,1]. The coefficient of the last term is calculated as $c_N = 1 - \sum_i^{(N-1)} c_i$.

`hydra::PDFSumExtendable<Pdf1, Pdf2,...>` and `hydra::PDFSumNonExtendabl<Pdf1, Pdf2,...>` objects can be conveniently created using the function template `hydra::add_pdfs(...)`. The code snippet below continues the *example* and defines a new PDF representing an exponential distribution and add it to the previous Gaussian PDF to build a extended model, which can be used to predict the yields:

```
...

//tau of the exponential
std::string  Tau("Tau");
hydra::Parameter  tau_p  = hydra::Parameter::Create()
        .Name(Tau)
        .Value(1.0)
        .Error(0.0001)
        .Limits(-2.0, 2.0);

//wrap a parametric lambda
auto exponential = hydra::wrap_lambda( [=] __host__ __device__
↪(unsigned int npar,
        const hydra::Parameter* params,unsigned int narg, double* x
↪){

        double tau = params[0];
        return exp( -(x[0]-min)*tau);

}, tau_p );

// build the PDF
auto PDF = hydra::make_pdf( exponential, GKQ61 );

//yields
std::string NG("N_Gauss");
std::string NE("N_Exp");
hydra::Parameter NG_p(NG , 1e4, 100.0, 1000 , 2e4) ;
hydra::Parameter NE_p(NE , 1e4, 100.0, 1000 , 2e4) ;

//add the pdfs
auto model = hydra::add_pdfs({NG_p, NE_p}, gaussian, exponential );
```

```
...
```

The user can get a reference to one of the component PDFs using the method `PDF( hydra::placeholder )`. This is useful, for example, to change the state of a component PDF "in place". Same operation can be performed for coeficients using the method `Coeficient( unsigned int )`:

```
#include<hydra/Placeholders.h>

using namespace hydra::placeholders;

...

//change the mean of the Gaussian to 2.0
model.PDF( _0 ).SetParameter(0, 2.0);

//set Gaussian coeficient  to 1.5e4
model.Coeficient(0).SetValue(1.5e4);
```

The Hydra classes representing PDFs are not dumb arithmetic beasts. These classes are lazy and implements a series of optimizations in order to forward to the thread collection only code that need effectively be evaluated. In particular, functor normalization is cached in a such way that only new parameters settings will trigger the calculation of integrals.

8.2 Defining FCNs and invoking the ROOT::Minuit2 interfaces

In general, a FCN is defined binding a PDF to the data the PDF is supposed to describe. Hydra implements classes and interfaces to allow the definition of FCNs suitable to perform maximum likelihood fits on unbinned and binned datasets. The different use cases for Likelihood FCNs are covered by the specialization of the class template `hydra::LogLikelihoodFCN<PDF, Iterator, Extensions...>`.

Objects representing likelihood FCNs can be conveniently instantiated using the function template `hydra::make_likelihood_fcn(data_begin, data_end , PDF)` and `hydra::make_likelihood_fcn(data_begin, data_end , weights_begin, PDF)`, where `data_begin`, `data_end` and `weights_begin` are iterators pointing to the dataset and the weights or bin-contents.

```
#include <hydra/LogLikelihoodFCN.h>

...

// get the fcn...
auto fcn    = hydra::make_loglikehood_fcn(dataset.begin(), dataset.
↪end(), model);
```

```
// and invoke Migrad minimizer from Minuit2
MnMigrad migrad(fcn, fcn.GetParameters().GetMnState(),␣
↪MnStrategy(2));
```

8.3 sPlots

The sPlot technique is used to unfold the contributions of different sources to the data sample in a given variable. The sPlot tool applies in the context of a Likelihood fit which needs to be performed on the data sample to determine the yields corresponding to the various sources.

Hydra handles sPlots using the class `hydra::SPlot<PDF1, PDF2,PDFs...>` where `PDF1`, `PDF2` and `PDFs...` are the probability density functions describing the populations contributing to the dataset as modeled in a given variable referred as discriminating variable. The other variables of interest, present in the dataset are referred as control variables and are statistically unfolded using the so called *sweights*. For each entry in the dataset, `hydra::SPlot<PDF1, PDF2,PDFs...>` calculates a set of weights, where each one corresponds to a data source described by the corresponding PDF. It is responsibility of the user to allocate memory to store the *sweights*.

The weights are calculated invoking the method `hydra::SPlot::Generate`, which returns the covariant matrix among the yields in the data sample.

```
#include <hydra/SPlot.h>

...

//splot 2 components (gaussian + exponential )
//hold weights
hydra::multiarray<2, double, hydra::device::sys_t> sweigts(dataset.
↪size());

//create splot
auto splot  = hydra::make_splot( fcn.GetPDF() );

auto covarm = splot.Generate( dataset.begin(), dataset.end(),␣
↪sweigts.begin());
```

CHAPTER 9

References

CHAPTER 10

Indices and tables

- genindex
- modindex
- search

Bibliography

[1] Minuit2 Package. https://root.cern.ch/root/html/MATH_MINUIT2_Index.html.

[2] Muriel Pivk and Francois R. Le Diberder. SPlot: A Statistical tool to unfold data distributions. *Nucl. Instrum. Meth.*, A555:356–369, 2005. arXiv:physics/0402083, doi:10.1016/j.nima.2005.08.106.

[3] CERNLIB - CERN Program Library. https://cernlib.web.cern.ch/cernlib.

[4] NVIDIA CUDA Toolkit. http://developer.nvidia.com/cuda-toolkit.

[5] Intel Threading Building Blocks. http://www.threadingbuildingblocks.org/.

[6] The OpenMP API specification for parallel programming. http://www.openmp.org/.

[7] Structure of arrays or SoA is a layout separating elements of a structure into one parallel array per field. https://en.wikipedia.org/wiki/AOS_and_SOA.

[8] Sequential back-end defined in Thrust.

[9] A.C. Genz and A.A. Malik. Remarks on algorithm 006: An adaptive algorithm for numerical integration over an N-dimensional rectangular region. *Journal of Computational and Applied Mathematics*, 6(4):295 – 302, 1980. URL: http://www.sciencedirect.com/science/article/pii/0771050X8090039X, doi:https://doi.org/10.1016/0771-050X(80)90039-X.

[10] Jarle Berntsen, Terje O. Espelid, and Alan Genz. An adaptive algorithm for the approximate calculation of multiple integrals. *ACM Trans. Math. Softw.*, 17(4):437–451, Dec 1991. URL: http://doi.acm.org/10.1145/210232.210233, doi:https://doi.org/10.1145/210232.210233.