Hydra.Python Documentation

Release 0.1.0

Antonio Augusto Alves Junior

Deepanshu Thakur

Eduardo Rodrigues

Sep 01, 2017

Contents

1 About this project 1

2 First steps 3

3 Vector Classes 5

4 Events Class 11

5 PhaseSpace Class 17

6 Random Class 21

7 Phase Space Example 25

CHAPTER 1

About this project

The **Hydra.Python** package provides the Python bindings for the header-only C++ Hydra library. This library is an abstraction over the C++ library, so that daily work can be coded and run with the Python language, concentrating on the logic and leaving all the complex memory management and optimisations to the C++ library.

The bindings are produced with pybind11. The project makes use of CMAKE for what concerns the building of the Hydra.Python library.

The library is written with `Linux` systems in mind, but compatibility with other platforms may be achieved with "hacks". Python versions 2.7, and 3.x are supported.

Core features

The core functionality of Hydra has been exposed to Python. The following core C++ features of Hydra can be mapped to Python:

- The continuous expansion of the original Hydra library.
- Support for particles with `Vector4R` class.
- Support for containers like `Events` or `Decay`.

Supported compilers

1. Clang/LLVM (any non-ancient version with C++11 support).
2. GCC 4.8 or newer.

History

The development of **Hydra.Python** started as a 2017 Google Summer of Code project (GSoC) with student Deepanshu Thakur.

CHAPTER 2

First steps

This section demonstrates the basic features of HydraPython. Before getting started, make sure that the development environment is set up to compile the included set of test cases.

Quick start

On Linux you'll need the Hydra and Pybind11 projects as well as **cmake** to build. The **python-dev** or **python3-dev** package is required too. You can clone and fetch the latest code for both of the mentioned libraries. Then you could create a directory structure similar to below one.

```
Project root -
  - Hydra.Python            (latest code of Hydra Python)
  - Hydra                   (latest code of Hydra)
  - Pybind11                (latest code of Pybind11)
  - build                   (build directory)
```

After downloading the prerequisites, run

```
cd build
cmake -DHYDRA_INSTALL_PATH=../Hydra -DPYBIND11_INSTALL_PATH=../pybind11/include -
↪DTHRUST_INSTALL_PATH=../Hydra ../Hydra.Python
make all
```

The last line will both compile and create a shared `.so` file which is the library imported in python.

The best way to check if the installation is correct or not is to run the test cases provided in the `Hydra.Python/tests/` directory.

Creating a simple Lorentz vector and calculating the particle's mass

Let's start by importing the module:

```
import HydraPython as hp
```

The name `HydraPython` is quite long so generally, we use its alias as `hp`.

Now that we have already imported the module let's just simply create the particle Lorentz vector, i.e. the `Vector4R` instance.

```
import HydraPython as hp
vec4 = hp.Vector4R()
print(vec4)  # (0, 0, 0, 0)
```

So this is it. We've just created a `Vector4R` object represending the 4-momentum vector of a particle. This matter is important when the `PhaseSpace` class will be used to generate Events with N particles.

The next 3 pages explain the Vector classes (`Vector4R` and `Vector3R`), the `Events` classes and the `PhaseSpace` class in more detail.

CHAPTER 3

Vector Classes

There are two vector classes available in Hydra, namely `Vector4R` and `Vector3R`.

Vector4R

The `Vector4R` class available in Python wraps the C++ `Vector4R` class representing four-dimensional relativistic vectors. Three types of constructors allow to instantiate the `Vector4R` class:

- Default empty constructor.
- Copy constructor.
- Constructor from 4 real (`float`) numbers.

```
import HydraPython as hp

vec1 = hp.Vector4R()  # construction with values 0.0 for all 4 particals
vec2 = hp.Vector4R(0.8385, 0.1242, 0.9821, 1.2424)
vec3 = hp.Vector4R(vec2)  # Copy construct the vec3 from vec2

print (vec1)  # (0, 0, 0, 0)
print (vec2)  # (0.8385,0.1242,0.9821,1.2424)
print (vec3)  # (0.8385,0.1242,0.9821,1.2424)
```

The `Vector4R` class also provides a pretty convenient method to create an instance from a python list.

```
list_ = [0.9241, 0.13223, 0.13121, 1.1141]
vec4 = hp.Vector4R(list_)
```

This will construct a new `Vector4R` object with the values passed within a list. The list should contain exactly 4 elements otherwise a `TypeError` will be raised. The `set` methods can be used to set all 4 values or a particular value in a `Vector4R` object, while the `get` method can be used with `Vector4R` to get the value of a particular index. The `__getitem__` and `__setitem` methods can also be used to get or set the value which comes very handy and maintain more pythonic way to access and set the values.

```
vec5 = hp.Vector4R(0.8385, 0.1242, 0.9821, 1.2424)
print (vec5)  # (0.8385,0.1242,0.9821,1.2424)

vec5.set(0, 0.9887)
print (vec5)  # (0.9887,0.1242,0.9821,1.2424)

vec5.set(0.1234, 0.5118, 0.9101, 0.1121)
print (vec5)  # (0.1234,0.5118,0.9101,0.1121)

print (vec5[1])  # 0.5118
print (vec5.get(1))  # 0.5118
vec5[1] = 0.5678
print (vec5)  # (0.1234,0.5678,0.9101,0.1121)
```

The `Vector4R` object can be multiplied or divided by a real value while it can be added or subtracted with another `Vector4R` object. One Vector4R object can be multiplied by another `Vector4R` object.

```
vec6 = hp.Vector4R(0.8215, 0.9241, 0.0105, 1.1994)
vec6 *= 1.1
print (vec6)  # (0.90365,1.01651,0.01155,1.31934)
vec6 /= 0.6
print (vec6)  # (1.50608,1.69418,0.01925,2.1989)

vec7 = hp.Vector4R(0.1223, 0.6433, 0.1234, 0.3010)
vec6 += vec7  # Add vec6 with the values of vec7
print (vec6)  # (1.62838,2.33748,0.14265,2.4999)

vec6 -= vec7
print (vec6)  # (1.50608,1.69418,0.01925,2.1989)
```

Two `Vector4R` objects can easily be added, subtracted or multiplied:

- `v = v1 + v2` # Returns a Vector4R object
- `v = v1 - v2` # Returns a Vector4R object
- `v = v1 * v2` # Returns a real number

All above three are valid for any `Vector4R` object. There are various other methods available in `Vector4R`. The list of `Vector4R` methods can be found on[1].

[1] The method list for Vector4R

- `set` Set the value at an index or all 4 values of Vector4R. Syntax:
  - vec1.set(idx, float)
  - vec1.set(float, float, float, float)
- `get` Get the value at an index for a Vector4R. Syntax:
  - vec1.get(idx)
- `assign` Assigns one Vector4R content to other Vector4R. Syntax:
  - vec1.assign(vec2)
- `cont` Finds the cont of the Vector4R object. Syntax:
  - result = vec1.cont(vec2)
- `mass` Returns the mass of the Vector4R object. Syntax:
  - result = vec1.mass()
- `mass2` Returns the mass2 of the Vector4R object. Syntax:
  - result = vec1.mass()
- `applyRotateEuler` Apply the rotate Eular on given Vector4R object. Syntax:
  - vec1.applyRotateEuler(float, float, float)
- `applyBoostTo` Apply the boost on the given Vector4R object. Syntax:

The `Vector4R` provides an `assign` method to assign or copy the Vector4R object. This is a very useful method to avoid the nasty bugs for example:

```
vec = hp.Vector4R(0.2010, 0.3010, 0.0210, 0.8385)
vec2 = hp.Vector4R()

# Do things and later in code ...
vec2.assign(vec)
vec == vec2  # True since all values are equal
vec is vec2  # False

vec = vec2   # Reference is copied
vec == vec2  # True
vec is vec2  # True
```

Vector3R

The `Vector43` class available in Python wraps the C++ `Vector3R` class representing three-dimensional Euclidian vectors. Three types of constructors allow to instantiate the `Vector3R` class:

- Default empty constructor.
- Copy constructor.
- Constructor from 3 real (`float`) numbers.

```
import HydraPython as hp

vec1 = hp.Vector3R()  # construction with values 0.0 for all 3 particals
vec2 = hp.Vector3R(0.8385, 0.1242, 0.9821)
vec3 = hp.Vector3R(vec2)  # Copy construct the vec3 from vec2

print (vec1)  # (0,0,0)
print (vec2)  # (0.8385,0.1242,0.9821)
print (vec3)  # (0.8385,0.1242,0.9821)
```

The `Vector3R` class also provides a pretty convenient method to create an object from python list.

  - vec1.applyBoostTo(vec2, bool)
  - vec1.applyBoostTo(Vector3R, bool) # Pay attention to Vector3R object
  - vec1.applyBoostTo(float, float, float, bool)
- `cross` Returns the cross product of two Vector4R. Syntax:
  - result_vector = vec1.cross(vec2)
- `dot` Returns the dot product of two Vector4R. Syntax:
  - result = vec1.dot(vec2)
- `d3mag` Returns the d3mag for two Vector4R. Syntax:
  - result = vec1.d3mag()
- `dotr3` Returns the dot product of three Vector4R. Syntax:
  - result = vec1.dotr3(vec2, vec3)
- `mag2r3` Returns the mag2r3 of two Vector4R. Syntax:
  - result = vec1.mag2r3(vec2)
- `magr3` Returns the magr3 of two Vector4R. Syntax:
  - result_vector = vec1.magr3(vec2)

```
list_ = [0.9241, 0.13223, 0.13121]
vec4 = hp.Vector3R(list_)
```

This will construct a new `Vector3R` object with the values passed within a list. The list should contain exactly 3 elements otherwise a `TypeError` will be raised. The `set` methods can be used to set all 3 values or a particular value in a `Vector3R` object, while the `get` method can be used with `Vector3R` to get the value of a particular index. The `__getitem__` and `__setitem` methods can also be used to get or set the value which comes very handy and maintain more pythonic way to access and set the values.

```
vec5 = hp.Vector3R(0.8385, 0.1242, 0.9821)
print (vec5)  # (0.8385,0.1242,0.9821)

vec5.set(0, 0.9887)
print (vec5)  # (0.9887,0.1242,0.9821)

vec5.set(0.1234, 0.5118, 0.9101)
print (vec5)  # (0.1234,0.5118,0.9101)

print (vec5[1])  # 0.5118
print (vec5.get(1))  # 0.5118
vec5[1] = 0.5678
print (vec5)  # (0.1234,0.5678,0.9101)
```

The `Vector3R` object can be multiplied or divided by a real value while it can be added or subtracted with another `Vector3R` object. One `Vector3R` object can be multiplied by another `Vector3R` object.

```
vec6 = hp.Vector3R(0.8215, 0.9241, 0.0105)
vec6 *= 1.1
print (vec6)  # (0.90365,1.01651,0.01155)
vec6 /= 0.6
print (vec6)  # (1.50608,1.69418,0.01925)

vec7 = hp.Vector3R(0.1223, 0.6433, 0.1234)
vec6 += vec7  # Add vec6 with the values of vec7
print (vec6)  # (1.62838,2.33748,0.14265)

vec6 -= vec7
print (vec6)  # (1.50608,1.69418,0.01925)
```

Two `Vector3R` objects can easily be added, subtracted or multiplied:

- `v = v1 + v2` # Returns a Vector3R object
- `v = v1 - v2` # Returns a Vector3R object
- `v = v1 * v2` # Returns a real number

All above three are valid for any `Vector3R` object. There are various other methods available in `Vector3R`. The list of `Vector3R` methods can be found on[2].

[2] The method list for Vector3R

- `set` Set the value at an index or all 3 values of Vector3R. Syntax:
  - vec1.set(idx, float)
  - vec1.set(float, float, float)
- `get` Get the value at an index for a Vector3R. Syntax:
  - vec1.get(idx)
- `assign` Assigns one Vector4R content to other Vector3R. Syntax:
  - vec1.assign(vec2)
- `dot` Returns the dot product of two Vector3R. Syntax:

The `Vector3R` provides an `assign` method to assign or copy the Vector3R object. This is a very useful method to avoid the nasty bugs for example:

```
vec = hp.Vector3R(0.2010, 0.3010, 0.0210)
vec2 = hp.Vector3R()

# Do things and later in code ...
vec2.assign(vec)
vec == vec2  # True since all values are equal
vec is vec2  # False

vec = vec2   # Reference is copied
vec == vec2  # True
vec is vec2  # True
```

  - result = vec1.dot(vec2)

- `d3mag` Returns the d3mag for two Vector3R. Syntax:
  - result = vec1.d3mag()

CHAPTER 4

Events Class

The Event class is a container class that holds the information corresponding to generated events. The Event class will not store the mother particle and store the N particle tuples with the element 0 storing the weight and rest of the elements storing the `Vector4R` of each particle. There are two types of Events one that runs on `host` and `device`. Events container currently supports up to (N=10) particles in final state with any number of Events. Both Host and Device Event classes add number (1 to 10) as their suffix to create Event for that number of particles and the type (host or device) is added as their prefix.

Host

The host is generally defined as the CPU. This class is a wrapper of C++ Events class that will work on CPU. This class is a container to hold the position of particles. We have 4 types of constructors to instantiate the Events class:

- Default empty constructor
- Constructor with number of events
- Copy constructor (from host to host)
- Copy constructor (from device to host)

```
import HydraPython as hp

h_events_5 = hp.host_events_5()  # construct host Event with 5 particles and 0 Events
print (h_events_5.size())  # 0

h_events_7_100 = hp.host_events_7(100)
print (h_events_7_100.size())  # 100
```

The `host_events_N` object can be copy constructed with the `host_events_N` or `device_events_N` object.

```
import HydraPython as hp
h_events_3 = hp.host_events_3(4)
print (list(h_events_3.Flags()))  # [False, False, False, False]
```

```
h_events_3.setFlag(1, True)
h_events_3_copy = hp.host_events_3(h_events_3)
print(list(h_events_3_copy.Flags()))  # [False, True, False, False]
```

The `setFlags` method as demonstrated above can be used to set the particular Flag value and the `getFlag` method can be used the get the particular flag value with the index.

```
h_event = hp.host_events_5(8)
h_event.setFlag(1, True)
print (h_event.getFlag(1))  # True
```

The host Events class provides an `assign` method to assign or copy the Events object. This is a very useful method to avoid the nasty bugs for example:

```
h_event = hp.host_events_5(10)
h_event2 = hp.host_events_5()

# Do things and later in the code ...
h_event2.assign(h_event)
# This will create the exact same copy of the h_event in h_event2
```

The host Events class also provides a method to set the Maximum weight of the Events. The method is useful to set

the maximum weight. The complete list of the classes in the Events container can be found on[1]. The complete method list provided by the Event classes can be found on[2].

The Events classes also provide a pythonic way to access the events with the `[]` operator. For example, an event value can be access like this.

[1] The list of Events classes

- `host_events_1` Generate 1 particle Event. Syntax:
  - h_event = hp.host_events_1(entries)
- `host_events_2` Generate 2 particle Event. Syntax:
  - h_event = hp.host_events_2(entries)
- `host_events_3` Generate 3 particle Event. Syntax:
  - h_event = hp.host_events_3(entries)
- `host_events_4` Generate 4 particle Event. Syntax:
  - h_event = hp.host_events_4(entries)
- `host_events_5` Generate 5 particle Event. Syntax:
  - h_event = hp.host_events_5(entries)
- `host_events_6` Generate 6 particle Event. Syntax:
  - h_event = hp.host_events_6(entries)
- `host_events_7` Generate 7 particle Event. Syntax:
  - h_event = hp.host_events_7(entries)
- `host_events_8` Generate 8 particle Event. Syntax:
  - h_event = hp.host_events_8(entries)
- `host_events_9` Generate 9 particle Event. Syntax:
  - h_event = hp.host_events_9(entries)
- `host_events_10` Generate 10 particle Event. Syntax:
  - h_event = hp.host_events_10(entries)
- `device_events_1` Generate 1 particle Event. Syntax:
  - d_event = hp.device_events_1(entries)
- `device_events_2` Generate 2 particle Event. Syntax:
  - d_event = hp.device_events_2(entries)
- `device_events_3` Generate 3 particle Event. Syntax:
  - d_event = hp.device_events_3(entries)
- `device_events_4` Generate 4 particle Event. Syntax:
  - d_event = hp.device_events_4(entries)
- `device_events_5` Generate 5 particle Event. Syntax:
  - d_event = hp.device_events_5(entries)
- `device_events_6` Generate 6 particle Event. Syntax:
  - d_event = hp.device_events_6(entries)
- `device_events_7` Generate 7 particle Event. Syntax:
  - d_event = hp.device_events_7(entries)
- `device_events_8` Generate 8 particle Event. Syntax:
  - d_event = hp.device_events_8(entries)
- `device_events_9` Generate 9 particle Event. Syntax:
  - d_event = hp.device_events_9(entries)
- `device_events_10` Generate 10 particle Event. Syntax:
  - d_event = hp.device_events_10(entries)

[2] The method list for Events classes

- `assign` Assigns one Events content to other Events. Syntax:
  - event2.assign(event1) # event1's content will be assigned to event2.
  - event2_device.assign(event1_host) # event1_host's (which is a host event) content will be assigned to event2_device (which is a device event)
  - event2_host.assign(event1_device) # event1_device's (which is a device event) content will be assigned to event2_host (which is a

```
event = hp.host_events_5(5)
print(event[1])  # (0.0, (0.0, 0.0, 0.0, 0.0), (0.0, 0.0, 0.0, 0.0), (0.0, 0.0, 0.0,␣
↪0.0), (0.0, 0.0, 0.0, 0.0), (0.0, 0.0, 0.0, 0.0))
```

host event)

- event2_device.assign(event1_device) # event1_device's (which is a device event) content will be assigned to event2_device (which is also a device event)
- event2_host.assign(event1_host) # event1_host's (which is a host event) content will be assigned to event2_host (which is also a host event)

- `GetMaxWeight` Gets the maximum weight of the Event's container. Syntax:
  - event.GetMaxWeight()
  - event_host.GetMaxWeight() # Get's the maximum weight of the host Event's container.
  - event_device.GetMaxWeight() # Get's the maximum weight of the device Event's container.
- `GetNEvents` Gets the number of events. Syntax:
  - event.GetNEvents()
  - event_host.GetNEvents() # Get's the number of events in host Event's container.
  - event_device.GetNEvents() # Get's the number of events in device Event's container.
- `Flags` This method returns the iterator of flags. Syntax:
  - iterator = event.Flags() # event can be of the host or device type and then can be used to iterator over the values. For example `for flag in iterator: print(flag)`
- `Weights` This method returns the iterator of weights. Syntax:
  - iterator = event.Weights() # event can be of the host or device type and then can be used to iterator over the values. For example `for weight in iterator: print(weight)`
- `Daughters` This method returns the iterator of daughters at given index (index <= number of the particle). Syntax:
  - iterator = event.Daughters(i) # event can be of the host or device type and then can be used to iterator over the values. For example `for daughter in iterator: print(daughter)` Will print out the ith particle state in all the events.
- `getDaughters` This method returns the daughter particles at given index.
  - vector_float4 = event.getDaughters(i)
- `Events` This method returns the iterator of events. Syntax:
  - iterator = event.Events() # event can be of the host or device type and then can be used to iterator over the values. For example `for e in iterator: print(e)` Will print out all the events where each event will have the N daughters and the weight of the event.
- `SetMaxWeight` Sets the maximum weight of the events. Syntax:
  - event.SetMaxWeight(float) # Sets the maximum weight of the events.
- `resize` resize the number of events. Syntax:
  - event.resize(knumber) # Sets the events container to hold `knumber` of events.
- `size` Gets the size or the number of events in the container. Syntax:
  - event.size() # Return the total number of events.
- `unweight` Unweight the events with the given seed. Syntax:
  - events.unweight(seed)
- `setFlag` Set the particular flag with given value. Syntax:
  - event.setFlag(idx, bool)
- `getFlag` Gets the particular flag value. Syntax:
  - events.getFlag(idx)
- `setWeight` Set the particular event's weight given value. Syntax:
  - event.setWeight(idx, float)
- `getWeight` Gets the particular event's weight. Syntax:
  - event.getWeight(idx)

Device

The device is defined as the GPU and any other multicore CPU. The device Event class is exactly similar to the Host Events class but the only major difference is HOST Events class work on the single CPU while the DEVICE Events class work on the multiple CPUs or the GPU devices.

In HydraPython the device Events classes support all the method defined by the host Event classes. The device Event class have `device` as their prefix and the number of particle N (up to 10) as their suffix.

It is only the fact that all the methods that can be used with the host can also be used with the device classes, even the name of the methods are same, just the declaration of the objects is different. So if you want to create a device object of particle 7 you will do something like this,

```
import HydraPython
device_event_with_7_particle = HydraPython.device_events_7()

# This will create a device Events with 0 states and 7 particles.
```

CHAPTER 5

PhaseSpace Class

This class implements the phase-space Monte Carlo event generation where N is the number of particles in the final state. Currently PhaseSpace class supports up-to N=10 number of particles in the Final state. Most of the PhaseSpace class methods can work on both `HOST` and `DEVICE`. The number of particles is associated with suffix with the class name.

This class is the wrapper for the C++ PhaseSpace class. The PhaseSpace class contains one constructor to instantiate it:

- Constructor with N number of daughter masses.

```
import HydraPython as hypy

p = hypy.PhaseSpace4([3.096916, 0.493677, 0.13957018, 0.0195018])
# This will construct the PhaseSpace object with the 4 daughter masses in the list.
```

The PhaseSpace classes provides a method to generate a phase-space decay given an output range or a phase-space given a range of mother particles and an output range.

```
# The below example generates and fills 3 states of 4 particle host events
vec4 = hypy.Vector4R(5.2795, 0.0, 0.0, 0.0)
ps = hypy.PhaseSpace4([3.096916, 0.493677, 0.13957018, 0.0195018])
e_host = hypy.host_events_4(3)
e_device = hypy.device_events_4(3)
ps.GenerateOnhost(vec4, e_host)  # Generate particle on host
ps.GenerateOndevice(vec4, e_device)  # Generate particle on device
```

```
B0_mass = 5.27955
B0 = hypy.Vector4R(B0_mass, 0.0, 0.0, 0.0)

mothers = hypy.host_vector_float4(5)
# Fill mother with some particles
mothers[0] = (3.326536152819228, -0.7376241292510032, 0.9527533342879685, 0.
↪15239715864543849)
mothers[1] = (3.3327060111834546, -0.44741166640978447, 1.012640505284964, -0.
↪5390007001803998)
```

```
mothers[2] = (3.4673036097962844, 0.6781637974979919, -1.4020213115136253, -0.
↪0763859825560801)
mothers[3] = (3.5042443315560945, 1.5383404921780213, -0.1442073504412384, -0.
↪5492280905481964)
mothers[4] = (3.4406218104833015, -0.16339927010014546, 1.363729549941791, 0.
↪6005257912194031)

phsp2 = hypy.PhaseSpace2([0.1056583745, 0.1056583745])
grand_daughter = hypy.host_events_2(5)
phsp2.GenerateOnhost(mothers, grand_daughter)

for i in grand_daughter: print(i)
```

The `AverageOnhost` and `AverageOndevice` method by PhaseSpace classes calculate the `mean` and `sqrt(variance)` of a functor over the phase-space with n-samples or of a functor over the phase-space given a list of mother particles.

```
import HydraPython as hypy
import math
def foo(*data):
    p1, p2, p3 = data[0], data[1], data[2]
    p = p1 + p2 + p3
    q = p2 + p3
    pd = p * p2
    pq = p * q
    qd = q * p2
    mp2 = p.mass2()
    mq2 = q.mass2()
    md2 = p2.mass2()
    return (pd * mq2 - pq * qd) / math.sqrt((pq * pq - mq2 * mp2) * (qd * qd - mq2 *␣
↪md2))

vec4 = hypy.Vector4R(5.2795, 0.0, 0.0, 0.0)
p = hypy.PhaseSpace4([3.096916, 0.493677, 0.13957018, 0.0195018])
tup = p.AverageOnhost(vec4, foo, 10)  # Average of host, currently passing functor to␣
↪device will fail
print (tup[0])  # Mean
print (tup[1])  # sqrt of variance
```

Like generators, the AverageOn method also can accept the list of mother particle instead of one mother particle and calculate the `mean` and `sqrt(variance)`.

The `EvaluateOnhost` and `EvaluateOndevice` evaluates a functor over the passed one mother particle or the list of mother particles.

The complete list of class implementations can be found at[1] and the complete list of methods supported can be found

[1] The list of PhaseSpace classe implementations

- `PhaseSpace2` Generate the phase-space with 2 particles. Syntax:
  - p = hypy.PhaseSpace2([2 daughter masses])
- `PhaseSpace3` Generate the phase-space with 3 particles. Syntax:
  - p = hypy.PhaseSpace3([3 daughter masses])
- `PhaseSpace4` Generate the phase-space with 4 particles. Syntax:
  - p = hypy.PhaseSpace4([4 daughter masses])
- `PhaseSpace5` Generate the phase-space with 5 particles. Syntax:
  - p = hypy.PhaseSpace5([5 daughter masses])
- `PhaseSpace6` Generate the phase-space with 6 particles. Syntax:
  - p = hypy.PhaseSpace6([6 daughter masses])

at[2].

- `PhaseSpace7` Generate the phase-space with 7 particles. Syntax:
  - p = hypy.PhaseSpace7([7 daughter masses])
- `PhaseSpace8` Generate the phase-space with 8 particles. Syntax:
  - p = hypy.PhaseSpace8([8 daughter masses])
- `PhaseSpace9` Generate the phase-space with 9 particles. Syntax:
  - p = hypy.PhaseSpace9([9 daughter masses])
- `PhaseSpace10` Generate the phase-space with 10 particles. Syntax:
  - p = hypy.PhaseSpace10([10 daughter masses])

[2] The list of methods for the PhaseSpace classes

- `GetSeed` Get the seed. Syntax:
  - p.GetSeed()
- `SetSeed` Set seed. Syntax:
  - p.SetSeed(seed)
- `GenerateOnhost` Generate the phase-space. Syntax:
  - p.GenerateOnhost(vector4R, event)
  - p.GenerateOnhost(hypy.host_vector_float4& mothers, event)
- `GenerateOndevice` Generate the phase-space. Syntax:
  - p.GenerateOndevice(vector4R, event)
  - p.GenerateOndevice(hypy.device_vector_float4& mothers, event)
- `AverageOnhost` Get the mean and sqrt of variance. Syntax:
  - p.AverageOnhost(vector4R, functor, number_of_entires)
  - p.AverageOnhost(hypy.host_vector_float4& mothers, functor)
- `AverageOndevice` Get the mean and sqrt of variance. Syntax:
- `AverageOndevice` Get the mean and sqrt of variance. Syntax:
  - p.AverageOndevice(vector4R, functor, number_of_entires)
  - p.AverageOndevice(hypy.device_vector_float4& mothers, functor)
- `EvaluateOnhost` Evaluate a function over the given particle or list of particles:
  - p.EvaluateOnhost(vector4R, hypy.host_vector_float2& result, functor)
  - p.EvaluateOnhost(hypy.host_vector_float4& mothers, hypy.host_vector_float2& result, functor)
- `EvaluateOndevice` Evaluate a function over the given particle or list of particles:
  - p.EvaluateOndevice(vector4R, hypy.device_vector_float2& result, functor)
  - p.EvaluateOndevice(hypy.device_vector_float4& mothers, hypy.device_vector_float2& result, functor)

CHAPTER 6

Random Class

This class implements functionalities associated with random number generation and pdf sampling. This class can sample and fill ranges with data corresponding to `Gaussian`, `Exponential`, `Uniform` and `Breit-Wigner` distributions.

This class is a wrapper of `hydra C++ Random class`. The Random class have two constructors to instantiate the Random class:

- Constructor with empty seed. The default seed value is `7895123`.
- Constructor expecting the seed.

```
import HydraPython as hp

r = hp.Random()          # default seed
r2 = hp.Random(8385977)  # Seed value
# This will construct the 2 Random class's object one with default seed and
# one with the seed value 8385977
```

Apart from setting the seed value at the time of creation the seed can be set or get with setter and getter methods named `SetSeed` and `GetSeed`.

```
r = hp.Random()
print (r.GetSeed())          # Give the seed value of object r. 7895123
r.SetSeed(988763)            # New seed is 988763
print (r.GetSeed())          # 988763
```

The Random class provides a method named `Uniform` to generate the numbers between range (min, max) (both min and max exclusive) and fill them into the container according to the Continuous Uniform distribution. The container can be any of the `host_vector_float` or `device_vector_float`. In below examples, I have used device_vector_float extensively but they both can be used interchangeably in place of each other.

```
import HydraPython as hp

container = hp.device_vector_float(1000000)    # Continer to hold 1000000 objects
r = hp.Random()                               # Random number generator object
```

```
r.Uniform(1.1, 1.5, container)  # Minimum number 1.1, maximum 1.5 and container
# Above will generate 1000000 numbers between (1.1, 1.5)

print (container[:10])
```

The Gauss random number generation method can also be used with the Random class. The `Gauss` method generate the numbers with the given `mean` and `sigma` values.

```
import HydraPython as hp

container = hp.device_vector_float(1000000)    # Continer to hold 1000000 objects
r = hp.Random()                                # Random number generator object
r.Gauss(-2.0, 1.0, container)
# Above will generate 1000000 with mean -2.0 and sigma as 1.0
```

The Exponential random number generation method or `Exp` method in Random class generates the numbers with the given `tau` value of the Exponential distribution.

```
import HydraPython as hp

container = hp.host_vector_float(100)    # Continer to hold 100 objects.
r = hp.Random()                          # Random number generator object
r.Exp(1.0, container)                    # tau is 1.0
print (container)
```

The Random class also provides a `BreitWigner` method to generate random number according to a BreitWigner with `mean` and `width`.

```
import HydraPython as hp

container = hp.device_vector_float(10000)    # Continer to hold 10000 objects.
r = hp.Random()                              # Random number generator object
r.BreitWigner(2.0, 0.2, container)           # mean=2.0, width=0.2
print (container)
```

Apart from all these distributions, you can also define your own distribution and pass it as a function to the method. The `Sample` method allows you to pass a function that will be sampled for the given sampling range (lower, upper) and store the result in the container.

```
import HydraPython as hp

# The functon which will be sampled.
import math
def gauss1(*args):
    g = 1.0
    mean = -2.0
    sigma = 1.0
    for i in range(3):
        m2 = (args[i] - mean) * (args[i] - mean)
        s2 = sigma * sigma
        g *= math.e ** ((-m2/(2.0 * s2 ))/( math.sqrt(2.0*s2*math.pi)))
    return g

container = hp.host_vector_float3(10000)  # Container with 10000 objects each having␣
↪3 floats
r = hp.Random()                           # Random object
r.Sample(d, [6, 6, 6], [-6, -6, -6], gauss1)
```

```
# d is container, [6, 6, 6] is the start range (1 for each float in container),
# [-6, -6, -6] is end range, gauss1 is the functor.
```

In sample method, the start range and end range should have the same number of arguments as in the container. So for example, if you are using container of `float7` than start range and end range each should contain `7` elements.

Warning: Any of device containers will not work with `Sample` method.

The complete method list supported by Random class can be found on[1].

[1] The method list for Random classes

- `GetSeed` Get the seed. Syntax:
  - seed = r.GetSeed()
- `SetSeed` Set seed. Syntax:
  - r.SetSeed(seed)
- `Gauss` Generate the Gauss distribution. Syntax:
  - r.Gauss(mean, sigma, container) # container can be [device/host]_vector_float
- `Uniform` Generate the Continuous Uniform distribution. Syntax:
  - r.Uniform(min, max, container) # container can be [device/host]_vector_float
- `Exp` Generate the Exponential distribution. Syntax:
  - r.Exp(tau, container) # container can be [device/host]_vector_float
- `BreitWigner` Generate the BreitWigner distribution. Syntax:
  - r.BreitWigner(mean, width, container) # container can be [device/host]_vector_float
- `Sample` sample the given function. Syntax:
  - iterator_accepted_events = r.Sample(container, [min_values_list], [max_limit_list], function) # Container could be any of the container listed below

The container list that can be passed to `Sample` method can be found on[2].

[2] The list of available containers to use with Random.

- `host_vector_float` host container with 1 float. Syntax:
  - h_container1 = hp.host_vector_float(size)
- `host_vector_float2` host container with 2 float. Syntax:
  - h_container2 = hp.host_vector_float2(size)
- `host_vector_float3` host container with 3 float. Syntax:
  - h_container3 = hp.host_vector_float3(size)
- `host_vector_float4` host container with 4 float. Syntax:
  - h_container4 = hp.host_vector_float4(size)
- `host_vector_float5` host container with 5 float. Syntax:
  - h_container5 = hp.host_vector_float5(size)
- `host_vector_float6` host container with 6 float. Syntax:
  - h_container6 = hp.host_vector_float6(size)
- `host_vector_float7` host container with 7 float. Syntax:
  - h_container7 = hp.host_vector_float7(size)
- `host_vector_float8` host container with 8 float. Syntax:
  - h_container8 = hp.host_vector_float8(size)
- `host_vector_float9` host container with 9 float. Syntax:
  - h_container9 = hp.host_vector_float9(size)
- `host_vector_float10` host container with 10 float. Syntax:
  - h_container10 = hp.host_vector_float10(size)
- `device_vector_float` device container with 1 float. Syntax:
  - d_container1 = hp.device_vector_float(size)
- `device_vector_float2` device container with 2 float. Syntax:
  - d_container2 = hp.device_vector_float2(size)
- `device_vector_float3` device container with 3 float. Syntax:
  - d_container3 = hp.device_vector_float3(size)
- `device_vector_float4` device container with 4 float. Syntax:
  - d_container4 = hp.device_vector_float4(size)
- `device_vector_float5` device container with 5 float. Syntax:
  - d_container5 = hp.device_vector_float5(size)
- `device_vector_float6` device container with 6 float. Syntax:
  - d_container6 = hp.device_vector_float6(size)
- `device_vector_float7` device container with 7 float. Syntax:
  - d_container7 = hp.device_vector_float7(size)
- `device_vector_float8` device container with 8 float. Syntax:
  - d_container8 = hp.device_vector_float8(size)
- `device_vector_float9` device container with 9 float. Syntax:
  - d_container9 = hp.device_vector_float9(size)
- `device_vector_float10` device container with 10 float. Syntax:
  - d_container10 = hp.device_vector_float10(size)

CHAPTER 7

Phase Space Example

This page is basically to demonstrate, how the PhaseSpace class with N particles can be used to generate the Events.

```
import HydraPython as hp
```

Above line will bring the classes in HydraPython in the scope of interpreter with the alias hp.

I will be using the Generate method of PhaseSpace here.

```
ps = hp.PhaseSpace4([3.096916, 0.493677, 0.13957018, 0.0195018])
```

Above will create a PhaseSpace class object with `N=4` number of particles in the final state. Since the number HydraPython currently supports particles up-to N=10 in the final state each PhaseSpace class have a suffix number from 1 to 10 is associated with it. The argument to the PhaseSpace class constructor is the list of `daughter masses`. The size of this list should be equal to the number of particles in the final state.

Now that we have defined a PhaseSpace object, let's create an `Event` container to contain or save the states of the particle generated by the `Generate` method.

```
e_host = hp.host_events_4(3)
```

Above I have defined a host Event container with N=4 particle and number of states as 3. So this container will contain the 3 states of 4 particles each generated by the PhaseSpace.

Above will generate the events or states of N particles using host and save the result in the passed `e_host` container.

Iterating over `e_host` will produce the output like this.

```
iterator = e_host.Events()
for state in iterator:
    print(state)
```

The output is similar to this.

```
(0.0005371556105645586, (3.26523953659142, 0.8636638960657156, 0.0039751958746361005,␣
↪-0.5700608675519644), (0.5205929150762441, 0.1361899815237809, 0.005650876525868165,
↪ -0.09338286473236444), (0.20194244730558714, -0.1422365383415909, 0.
↪02243309740186762, 0.023800003783548303), (1.0705417836594209, -0.8576173392479055,␣
↪-0.03205916980237188, 0.6396437285007806))

(0.05958088064572496, (3.165087693874953, -0.03009443713313225, 0.6184073639056892, 0.
↪2087056683071267), (0.5809611490129989, -0.016410682480807473, -0.
↪054177669092790454, -0.30098894665035486), (0.7999891064682725, 0.08709929588193556,
↪ -0.6686502155923885, -0.40721411710277927), (0.5122787332764478, -0.
↪04059417626799582, 0.10442052077948974, 0.4994973954460073))

(0.03738710351970522, (3.376147992537914, -0.4901521345374072, 1.085407051180553, 0.
↪6238020316717038), (1.0297008095722722, 0.22021896692371404, -0.8251558826920553, -
↪0.29527640063259364), (0.49365860519565796, 0.27558785182792184, -0.
↪33498661390711465, -0.18987966654280578), (0.15880927532682793, -0.
↪005654684214228855, 0.07473544541861718, -0.13864596449630434))
```

So what is this? It is the tuple of output in which the first element of tuple represent the `weight` and the remaining number of elements are the Vector4R of each particle for N particle. (In this case 4)

If you will closely follow the result, you will see that the each particle in every event has the mass specified by the list of daughter masses at the time of the creation of PhaseSpace.

```
state1 = e_host[0]  # first state particle
d_particle0, d_particle1, d_particle2, d_particle3 = state1[1], state1[2], state1[3],␣
↪state1[4]

d_particle0 = hp.Vector4R(d_particle0)
d_particle1 = hp.Vector4R(d_particle1)
d_particle2 = hp.Vector4R(d_particle2)
d_particle3 = hp.Vector4R(d_particle3)

print(d_particle0.mass(), d_particle1.mass(), d_particle2.mass(), d_particle3.mass(),␣
↪sep=', ')

# Output is
# 3.096916, 0.493677, 0.13957017999999996, 0.01950179999999231
# This is exactly the weight given for each daughter while creation of PhaseSpace
# Same thing is true for rest of the states.
```

So this is a simple PhaseSpace example of 4 particles in the final state. For the sake of completeness, all the code showed in the doc is below.

```
import HydraPython as hp

mother_particle = hp.Vector4R(5.2795, 0.83859, 0.77825, 0.98876)
daughter_masses = [3.096916, 0.493677, 0.13957018, 0.0195018]
print("Daughter masses at the time of creation of PhaseSpace:", daughter_masses)
print()

ps = hp.PhaseSpace4(daughter_masses)
e_host = hp.host_events_4(3)
ps.Generatehost(mother_particle, e_host)

iterator = e_host.Events()
for idx, state in enumerate(iterator):
    print("State", idx, ": ", state)
```

```
state1 = e_host[0]  # first state particle
d_particle0, d_particle1, d_particle2, d_particle3 = state1[1], state1[2], state1[3],␣
↪state1[4]

d_particle0 = hp.Vector4R(d_particle0)
d_particle1 = hp.Vector4R(d_particle1)
d_particle2 = hp.Vector4R(d_particle2)
d_particle3 = hp.Vector4R(d_particle3)

print('\nDaughter masses:', d_particle0.mass(), d_particle1.mass(), d_particle2.
↪mass(), d_particle3.mass(), sep=', ')
```