
TensorFlow PhaseSpace Documentation

Release 0.9.0

Albert Puig Navarro

Mar 13, 2019

Contents

1	TensorFlow PhaseSpace	1
2	Table of Contents	5

TensorFlow PhaseSpace

Python implementation of the Raubold and Lynch method for n -body events using TensorFlow as a backend.

The code is based on the GENBOD function (W515 from CERNLIB), documented in [1] and tries to follow it as closely as possible.

Detailed documentation, including the API, can be found in <https://phasespace.readthedocs.io>.

Free software: BSD-3-Clause.

[1] F. James, Monte Carlo Phase Space, CERN 68-15 (1968)

1.1 Why?

Lately, data analysis in High Energy Physics (HEP), traditionally performed within the **ROOT** ecosystem, has been moving more and more towards Python. The possibility of carrying out purely Python-based analyses has become real thanks to the development of many open source Python packages, which have allowed to replace most ROOT functionality with Python-based packages.

One of the aspects where this is still not possible is in the random generation of n -body phase space events, which are widely used in the field, for example to study kinematics of the particle decays of interest, or to perform importance sampling in the case of complex amplitude models. This has been traditionally done with the **TGenPhaseSpace** class, which is based on the GENBOD function of the CERNLIB FORTRAN libraries and which requires a full working ROOT installation.

This package aims to address this issue by providing a TensorFlow-based implementation of such function to generate n -body decays without requiring a ROOT installation. Additionally, an oft-needed functionality to generate complex decay chains, not included in **TGenPhaseSpace**, is also offered, leaving room for decaying resonances (which don't have a fixed mass, but can be seen as a broad peak).

1.2 Installing

To install TensorFlow PhaseSpace, run this command in your terminal:

```
$ pip install phasespace
```

This is the preferred method to install TensorFlow PhaseSpace, as it will always install the most recent stable release. For the newest development version (in case you really need it), you can install the version from git with

```
$ pip install git+https://github.com/zfit/phasespace
```

1.3 How to use

The generation of simple n -body decays can be done using the `generate` function of `phasespace` with a very similar interface to `TGenPhaseSpace`. For example, to generate $B^0 \rightarrow K\pi$, we would do:

```
import phasespace
import tensorflow as tf

B0_MASS = 5279.58
B0_AT_REST = [0.0, 0.0, 0.0, B0_MASS]
PION_MASS = 139.57018
KAON_MASS = 493.677

weights, particles = phasespace.generate(B0_AT_REST,
                                         [PION_MASS, KAON_MASS],
                                         1000)
```

This generates TensorFlow tensors, so no code has been executed yet. To run the TensorFlow graph, we simply do:

```
with tf.Session() as sess:
    weights, particles = sess.run([weights, particles])
```

This returns an array of 1000 elements in the case of `weights` and a list of n *particles* (2) arrays of (4, 1000) shape, where each of the 4-dimensions corresponds to one of the components of the generated Lorentz 4-vector.

Sequential decays can be handled with the `Particle` class (used internally by `generate`) and its `set_children` method. As an example, to build the $B^0 \rightarrow K^*\gamma$ decay in which $K^* \rightarrow K\pi$, we would write:

```
from phasespace import Particle
import tensorflow as tf

B0_MASS = 5279.58
B0_AT_REST = [0.0, 0.0, 0.0, B0_MASS]
KSTARZ_MASS = 895.81
PION_MASS = 139.57018
KAON_MASS = 493.677

pion = Particle('pi+', PION_MASS)
kaon = Particle('K+', KAON_MASS)
kstar = Particle('K*', KSTARZ_MASS).set_children(pion, kaon)
gamma = Particle('gamma', 0)
bz = Particle('B0').set_children(kstar, gamma)

with tf.Session() as sess:
    weights, particles = sess.run(bz.generate(B0_AT_REST, 1000))
```

Where we have used the fact that `set_children` returns the parent particle. In this case, `particles` is a dict with the particle names as keys:

```

>>> particles
{'K*': array([[ -2259.88717495,   742.20158838, -1419.57804967, ...,
              385.51632682,   890.89417859, -1938.80489221],
             [ -491.3119786 , -2348.67021741, -2049.19459865, ...,
              -932.58261761, -1054.16217965, -1669.40481126],
             [-1106.5946257 ,   711.27644522, -598.85626591, ...,
             -2356.84025605, -2160.57372728, -164.77965753],
             [ 2715.78804872,  2715.78804872,  2715.78804872, ...,
              2715.78804872,  2715.78804872,  2715.78804872]]),
 'K+': array([[ -1918.74294565,   363.10302225, -830.13803095, ...,
               9.28960349,   850.87382095, -895.29815921],
             [ -566.15415012, -956.94044749, -1217.14751182, ...,
              -243.52446264, -1095.04308712, -1078.03237584],
             [-1108.26109897,   534.79579335, -652.41135612, ...,
              -901.56453631, -2069.39723754, -244.1159568 ],
             [ 2339.67191226,  1255.90698132,  1685.21060224, ...,
              1056.37401241,  2539.53293518,  1505.66336806]]),
 'gamma': array([[2259.88717495, -742.20158838, 1419.57804967, ..., -385.51632682,
                 -890.89417859, 1938.80489221],
                 [ 491.3119786 , 2348.67021741, 2049.19459865, ...,  932.58261761,
                 1054.16217965, 1669.40481126],
                 [1106.5946257 , -711.27644522,  598.85626591, ..., 2356.84025605,
                 2160.57372728,  164.77965753],
                 [2563.79195128, 2563.79195128, 2563.79195128, ..., 2563.79195128,
                 2563.79195128, 2563.79195128]]),
 'pi+': array([[ -341.14422931,   379.09856613, -589.44001872, ...,
                 376.22672333,   40.02035764, -1043.506733 ],
               [  74.84217153, -1391.72976992, -832.04708683, ...,
                -689.05815497,   40.88090746, -591.37243542],
               [   1.66647327,  176.48065186,   53.55509021, ...,
                -1455.27571974,  -91.17648974,   79.33629927],
               [ 376.11613646,  1459.8810674 ,  1030.57744648, ...,
                1659.41403631,  176.25511354,  1210.12468065]]})

```

It is also important to note the mass is not necessary for the top particle, as it is determined from the input 4-momentum.

More examples can be found in the `tests` folder and in the [documentation](#).

1.4 Physics validation

Physics validation is performed continuously in the included tests (`tests/test_physics.py`), run through Travis CI. This validation is performed at two levels:

- In simple n -body decays, the results of `phasespace` are checked against `TGenPhaseSpace`.
- For sequential decays, the results of `phasespace` are checked against [RapidSim](#), a “fast Monte Carlo generator for simulation of heavy-quark hadron decays”. In the case of resonances, differences are expected because our tests don’t include proper modelling of their mass shape, as it would require the introduction of further dependencies. However, the results of the comparison can be expected visually.

The results of all physics validation performed by the `tests_physics.py` test are written in `tests/plots`.

1.5 Contributing

Contributions are always welcome, please have a look at the [Contributing guide](#).

2.1 Usage

The base of `phasespace` is the `Particle` object. This object, which represents a particle, either stable or decaying, has only one mandatory argument, its name.

In most cases (except for the top particle of a decay), one wants to also specify its mass, which can be either a number or `tf.constant`, or a function. Functions are used to specify the mass of particles such as resonances, which are not fixed but vary according to a broad distribution. These mass functions get three arguments, and must return a `TensorFlow Tensor`:

- The minimum mass allowed by the decay chain.
- The maximum mass available.
- The number of events to generate, *i.e.*, the size of the output tensor, which must be of shape $(1, n_events)$.

This function signature allows to handle threshold effects cleanly, giving enough information to produce kinematically allowed decays (NB: `phasespace` will throw an error if a kinematically forbidden decay is requested).

With these considerations in mind, one can build a decay chain by using the `set_children` method of the `Particle` class (which returns the class itself). As an example, to build the $B^0 \rightarrow K^* \gamma$ decay in which $K^* \rightarrow K \pi$, we would write:

```
from phasespace import Particle

KSTARZ_MASS = 895.81
PION_MASS = 139.57018
KAON_MASS = 493.677

pion = Particle('pi+', PION_MASS)
kaon = Particle('K+', KAON_MASS)
kstar = Particle('K*', KSTARZ_MASS).set_children(pion, kaon)
gamma = Particle('gamma', 0)
bz = Particle('B0').set_children(kstar, gamma)
```

The graph can then be generated by using the `generate` method, which, similarly to `TGenPhaseSpace`, gets an input 4-vector (shape (4,)) of the top particle, and, optionally, the number of events to generate. If the input 4-vector is of shape (4, n), then n events are generated. Since `generate` just creates a TensorFlow graph, we need to use `tf.Session.run()` to actually run the generation:

```
N_EVENTS = 1000
B0_MASS = 5279.58
B0_AT_REST = [0.0, 0.0, 0.0, B0_MASS]

with tf.Session() as sess:
    weights, particles = sess.run(bz.generate(B0_AT_REST, N_EVENTS))
```

As discussed, this would be equivalent to:

```
B0_MASS = 5279.58
B0s_AT_REST = [[0.0, 0.0, 0.0, B0_MASS] for _ in range(N_EVENTS)]

with tf.Session() as sess:
    weights, particles = sess.run(bz.generate(B0s_AT_REST))
```

This latter approach can be useful if the momentum of the top particle is generated according to some distribution, for example the kinematics of the LHC (see `test_kstargamma_kstarnonresonant_lhc` and `test_klgamma_kstarnonresonant_lhc` in `tests/test_physics.py` to see how this could be done).

The objects returned by `Particle.generate` are the normalized weights vector (the weight of each event divided by the maximum size of the phase space given the decay chain), of shape (N_EVENTS ,), and a dictionary containing the generated 4-momenta for each particle, of shape (4, N_EVENTS), labelled by their name:

```
>>> particles
{'K*': array([[ -2259.88717495,   742.20158838, -1419.57804967, ...,
    385.51632682,   890.89417859, -1938.80489221],
 [ -491.3119786 , -2348.67021741, -2049.19459865, ...,
   -932.58261761, -1054.16217965, -1669.40481126],
 [-1106.5946257 ,   711.27644522, -598.85626591, ...,
 -2356.84025605, -2160.57372728, -164.77965753],
 [ 2715.78804872,  2715.78804872,  2715.78804872, ...,
  2715.78804872,  2715.78804872,  2715.78804872]]),
 'K+': array([[ -1918.74294565,   363.10302225, -830.13803095, ...,
    9.28960349,   850.87382095, -895.29815921],
 [ -566.15415012, -956.94044749, -1217.14751182, ...,
  -243.52446264, -1095.04308712, -1078.03237584],
 [-1108.26109897,   534.79579335, -652.41135612, ...,
  -901.56453631, -2069.39723754, -244.1159568 ],
 [ 2339.67191226,  1255.90698132,  1685.21060224, ...,
  1056.37401241,  2539.53293518,  1505.66336806]]),
 'gamma': array([[2259.88717495, -742.20158838, 1419.57804967, ..., -385.51632682,
   -890.89417859, 1938.80489221],
 [ 491.3119786 , 2348.67021741, 2049.19459865, ...,  932.58261761,
  1054.16217965, 1669.40481126],
 [1106.5946257 , -711.27644522,  598.85626591, ..., 2356.84025605,
  2160.57372728,  164.77965753],
 [2563.79195128, 2563.79195128, 2563.79195128, ..., 2563.79195128,
  2563.79195128, 2563.79195128]]),
 'pi+': array([[ -341.14422931,   379.09856613, -589.44001872, ...,
    376.22672333,   40.02035764, -1043.506733 ],
 [  74.84217153, -1391.72976992, -832.04708683, ...,
  -689.05815497,   40.88090746, -591.37243542],
 [   1.66647327,  176.48065186,   53.55509021, ...,
```

(continues on next page)

(continued from previous page)

```
-1455.27571974, -91.17648974, 79.33629927],
[ 376.11613646, 1459.8810674, 1030.57744648, ...,
 1659.41403631, 176.25511354, 1210.12468065]]})
```

If interested in the unnormalized event weights, one can use the `generate_unnormalized` method, which returns the raw weights, the per-event maximum weight and the particle dictionary as before.

To generate the mass of a resonance, we need to give a function as its mass. Following with the same example as above, and approximating the resonance shape by a gaussian, we could write the $B^0 \rightarrow K^*\gamma$ decay chain as:

```
import tensorflow as tf
import tensorflow_probability as tfp
from phasespace import Particle

KSTARZ_MASS = 895.81
KSTARZ_WIDTH = 47.4

def kstar_mass(min_mass, max_mass, n_events):
    kstar_mass = KSTARZ_MASS * ones
    min_mass = tf.broadcast_to(min_mass, (1, n_events))
    max_mass = tf.broadcast_to(max_mass, (1, n_events))
    kstar_mass = tfp.distributions.TruncatedNormal(loc=KSTARZ_MASS,
                                                  scale=KSTARZ_WIDTH,
                                                  low=min_mass,
                                                  high=max_mass).sample()

bz = Particle('B0').set_children(Particle('K*0', mass=kstar_mass)
                                .set_children(Particle('K+', mass=KAON_MASS),
                                              Particle('pi-', mass=PION_MASS)),
                                Particle('gamma', mass=0.0))
```

2.1.1 Shortcut for simple decays

The generation of simple n -body decays can be done using the `generate` function of `phasespace`, which has a very similar interface to `TGenPhaseSpace`. For example, to generate $B^0 \rightarrow K\pi$, we would do:

```
import phasespace
import tensorflow as tf

N_EVENTS = 1000

B0_MASS = 5279.58
B0_AT_REST = [0.0, 0.0, 0.0, B0_MASS]
PION_MASS = 139.57018
KAON_MASS = 493.677

with tf.Session() as sess:
    weights, particles = sess.run(phasespace.generate(B0_AT_REST,
                                                    [PION_MASS, KAON_MASS],
                                                    N_EVENTS))
```

In this case, since particles are unnamed, the `particles` object contains a list of $(4, N_EVENTS)$ tensors in the order of the particles specified in the `generate` call.

Internally, this function builds a decay chain using `Particle`, and therefore the same considerations as before apply; for example, it is possible to not specify the number of events and give a list of input momenta.

2.2 Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

2.2.1 Types of Contributions

Report Bugs

Report bugs at <https://github.com/zfit/phasespace/issues>.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with “bug” and “help wanted” is open to whoever wants to implement it.

Implement Features

Look through the GitHub issues for features. Anything tagged with “enhancement” and “help wanted” is open to whoever wants to implement it.

Write Documentation

TensorFlow PhaseSpace could always use more documentation, whether as part of the official TensorFlow PhaseSpace docs, in docstrings, or even on the web in blog posts, articles, and such.

Submit Feedback

The best way to send feedback is to file an issue at <https://github.com/zfit/phasespace/issues>.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

2.2.2 Get Started!

Ready to contribute? Here’s how to set up *phasespace* for local development.

1. Fork the *phasespace* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/phasespace.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv phasespace
$ cd phasespace/
$ python setup.py develop
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ flake8 phasespace tests
$ python setup.py test or py.test
$ tox
```

To get flake8 and tox, just pip install them into your virtualenv.

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

2.2.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 2.7, 3.4, 3.5 and 3.6, and for PyPy. Check https://travis-ci.org/zfit/phasespace/pull_requests and make sure that the tests pass for all supported Python versions.

2.2.4 Tips

To run a subset of tests (for example those in *tests/test_generate.py*):

```
$ pytest -k test_generate
```

2.2.5 Deploying

A reminder for the maintainers on how to deploy. Make sure all your changes are committed (including an entry in HISTORY.rst). Then run:

```
$ bumpversion patch # possible: major / minor / patch
$ git push
$ git push --tags
```

Travis will then deploy to PyPI if tests pass.

2.3 Credits

2.3.1 Development Lead

- Albert Puig Navarro <albert.puig@cern.ch>

2.3.2 Core Developers

- Jonas Eschle <jonas.eschle@cern.ch>

2.3.3 Contributors

None yet. Why not be the first?