TensorFlow PhaseSpace Documentation

Release 0.0.0

Albert Puig Navarro

Oct 13, 2019

Contents

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. Don't hesitate to join our gitter channel for questions and comments.

If you use phasespace in a scientific publication we would appreciate citations to the zenodo publication:

```
@article{phasespace-2019,
 title={phasespace: n-body phase space generation in Python},
 DOI={10.5281/zenodo.2926058},
 publisher={Zenodo},
 author={Albert Puig and Jonas Eschle},
 year={2019},
 month={Mar}}
```

Free software: BSD-3-Clause.

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

CHAPTER 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 of 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 a 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).

CHAPTER 2

Installing

To install `phasespace`, run this command in your terminal:

```
$ pip install phasespace
```

This is the preferred method to install `phasespace`, as it will always install the most recent stable release.

For the newest development version, which may be unstable, you can install the version from git with

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

CHAPTER 3

How to use

The generation of simple *n*-body decays can be done using the `nbody_decay` shortcut to create a decay chain with a very simple interface: one needs to pass the mass of the top particle and the masses of the children particle as a list, optionally giving the names of the particles. Then, the *generate* method can be used to produce the desired sample. For example, to generate $B^0 \to K\pi$, we would do:

```
import phasespace

B0_MASS = 5279.58
PION_MASS = 139.57018
KAON_MASS = 493.677

weights, particles = phasespace.nbody_decay(B0_MASS,
                                            [PION_MASS, KAON_MASS]).generate(n_
↪events=1000)
```

This returns a numpy array of 1000 elements in the case of `weights` and a list of `n particles` (2) arrays of (1000, 4) shape, where each of the 4-dimensions corresponds to one of the components of the generated Lorentz 4-vector. All particles are generated in the rest frame of the top particle; boosting to a certain momentum (or list of momenta) can be achieved by passing the momenta to the `boost_to` argument.

Behind the scenes, this function runs the TensorFlow graph, but no caching of the graph or reusing the session is performed. If we want to get the graph to avoid an immediate execution, we can use the *generate_tensor* method. Then, to produce the equivalent result to the previous example, we simply do:

```
import tensorflow as tf

with tf.Session() as sess:
    weights, particles = phasespace.nbody_decay(B0_MASS,
                                                [PION_MASS, KAON_MASS]).generate_
↪tensor(n_events=1000)
    weights, particles = sess.run([weights, particles])
```

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

```
from phasespace import GenParticle

B0_MASS = 5279.58
KSTARZ_MASS = 895.81
PION_MASS = 139.57018
KAON_MASS = 493.677

pion = GenParticle('pi+', PION_MASS)
kaon = GenParticle('K+', KAON_MASS)
kstar = GenParticle('K*', KSTARZ_MASS).set_children(pion, kaon)
gamma = GenParticle('gamma', 0)
bz = GenParticle('B0', B0_MASS).set_children(kstar, gamma)

weights, particles = bz.generate(n_events=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([[ 1732.79325872, -1632.88873127,   950.85807735,  2715.78804872],
       [-1633.95329448,   239.88921123, -1961.0402768 ,  2715.78804872],
       [  407.15613764, -2236.6569286 , -1185.16616251,  2715.78804872],
       ...,
       [ 1091.64603395, -1301.78721269,  1920.07503991,  2715.78804872],
       [ -517.3125083 ,  1901.39296899,  1640.15905194,  2715.78804872],
       [  656.56413668,  -804.76922982,  2343.99214816,  2715.78804872]]),
 'K+': array([[  750.08077976,  -547.22569019,   224.6920906 ,  1075.30490935],
       [-1499.90049089,   289.19714633, -1935.27960292,  2514.43047106],
       [   97.64746732, -1236.68112923,  -381.09526192,  1388.47607911],
       ...,
       [  508.66157459,  -917.93523639,  1474.7064148 ,  1876.11771642],
       [ -212.28646168,   540.26381432,   610.86656669,   976.63988936],
       [  177.16656666,  -535.98777569,   946.12636904,  1207.28744488]]),
 'gamma': array([[-1732.79325872,  1632.88873127,  -950.85807735,  2563.79195128],
       [ 1633.95329448,  -239.88921123,  1961.0402768 ,  2563.79195128],
       [ -407.15613764,  2236.6569286 ,  1185.16616251,  2563.79195128],
       ...,
       [-1091.64603395,  1301.78721269, -1920.07503991,  2563.79195128],
       [  517.3125083 , -1901.39296899, -1640.15905194,  2563.79195128],
       [ -656.56413668,   804.76922982, -2343.99214816,  2563.79195128]]),
 'pi+': array([[  982.71247896, -1085.66304109,   726.16598675,  1640.48313937],
       [ -134.0528036 ,   -49.3079351 ,   -25.76067389,   201.35757766],
       [  309.50867032,  -999.97579937,  -804.0709006 ,  1327.31196961],
       ...,
       [  582.98445936,  -383.85197629,   445.36862511,   839.6703323 ],
       [ -305.02604662,  1361.12915468,  1029.29248526,  1739.14815935],
       [  479.39757002,  -268.78145413,  1397.86577911,  1508.50060384]])}
```

The *GenParticle* class is able to cache the graphs so it is possible to generate in a loop without overhead:

```
for i in range(10):
    weights, particles = bz.generate(n_events=1000)
    ...
    (do something with weights and particles)
    ...
```

This way of generating is recommended in the case of large samples, as it allows to benefit from parallelisation while at the same time keep the memory usage low.

If we want to operate with the TensorFlow graph instead, we can use the *generate_tensor* method of *GenParticle*, which has the same signature as *generate*.

More examples can be found in the `tests` folder and in the documentation.

CHAPTER 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`.

CHAPTER 5

Contributing

Contributions are always welcome, please have a look at the Contributing guide.

5.1 Credits

5.1.1 Development Lead

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

5.1.2 Core Developers

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

5.1.3 Contributors

None yet. Why not be the first?

5.2 Table of Contents

5.2.1 Usage

The base of `phasespace` is the `GenParticle` 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, which will be of shape *(n_events,)*.
- The maximum mass available, which will be of shape *(n_events,)*.
- The number of events to generate.

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 `GenParticle` class (which returns the class itself). As an example, to build the $B^0 \to K^*\gamma$ decay in which $K^* \to K\pi$ with a fixed mass, we would write:

```
from phasespace import GenParticle

B0_MASS = 5279.58
KSTARZ_MASS = 895.81
PION_MASS = 139.57018
KAON_MASS = 493.677

pion = GenParticle('pi+', PION_MASS)
kaon = GenParticle('K+', KAON_MASS)
kstar = GenParticle('K*', KSTARZ_MASS).set_children(pion, kaon)
gamma = GenParticle('gamma', 0)
bz = GenParticle('B0', B0_MASS).set_children(kstar, gamma)
```

Phasespace events can be generated using the `generate` method, which gets the number of events to generate as input. The method returns:

- The normalized weights of each event, as an array of dimension (n_events,).
- The 4-momenta of the generated particles as values of a dictionary with the particle name as key. These momenta are expressed as arrays of dimension (n_events, 4).

```
N_EVENTS = 1000

weights, particles = bz.generate(n_events=N_EVENTS)
```

The `generate` method directly produces numpy arrays; for advanced usage, `generate_tensor` returns the same objects with the numpy arrays replaced by `tf.Tensor` of the same shape. So one can do, equivalent to the previous example:

```
import tensorflow as tf

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

In both cases, the particles are generated in the rest frame of the top particle. To produce them at a given momentum of the top particle, one can pass these momenta with the `boost_to` argument in both `generate` and ~'tf.Tensor'. 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_k1gamma_kstarnonresonant_lhc` in `tests/test_physics.py` to see how this could be done).

Additionally, it is possible to obtain the unnormalized weights by using the `generate_unnormalized` flag in `generate` and `generate_tensor`. In this case, the method returns the unnormalized weights, the per-event maximum weight and the particle dictionary.

```
>>> particles
{'K*': array([[ 1732.79325872, -1632.88873127,   950.85807735,  2715.78804872],
```

(continues on next page)

(continued from previous page)

```
       [-1633.95329448,   239.88921123, -1961.0402768 ,  2715.78804872],
       [  407.15613764, -2236.6569286 , -1185.16616251,  2715.78804872],
       ...,
       [ 1091.64603395, -1301.78721269,  1920.07503991,  2715.78804872],
       [ -517.3125083 ,  1901.39296899,  1640.15905194,  2715.78804872],
       [  656.56413668,  -804.76922982,  2343.99214816,  2715.78804872]]),
 'K+': array([[  750.08077976,  -547.22569019,   224.6920906 ,  1075.30490935],
       [-1499.90049089,   289.19714633, -1935.27960292,  2514.43047106],
       [   97.64746732, -1236.68112923,  -381.09526192,  1388.47607911],
       ...,
       [  508.66157459,  -917.93523639,  1474.7064148 ,  1876.11771642],
       [ -212.28646168,   540.26381432,   610.86656669,   976.63988936],
       [  177.16656666,  -535.98777569,   946.12636904,  1207.28744488]]),
 'gamma': array([[-1732.79325872,  1632.88873127,  -950.85807735,  2563.79195128],
       [ 1633.95329448,  -239.88921123,  1961.0402768 ,  2563.79195128],
       [ -407.15613764,  2236.6569286 ,  1185.16616251,  2563.79195128],
       ...,
       [-1091.64603395,  1301.78721269, -1920.07503991,  2563.79195128],
       [  517.3125083 , -1901.39296899, -1640.15905194,  2563.79195128],
       [ -656.56413668,   804.76922982, -2343.99214816,  2563.79195128]]),
 'pi+': array([[  982.71247896, -1085.66304109,   726.16598675,  1640.48313937],
       [ -134.0528036 ,   -49.3079351 ,   -25.76067389,   201.35757766],
       [  309.50867032,  -999.97579937,  -804.0709006 ,  1327.31196961],
       ...,
       [  582.98445936,  -383.85197629,   445.36862511,   839.6703323 ],
       [ -305.02604662,  1361.12915468,  1029.29248526,  1739.14815935],
       [  479.39757002,  -268.78145413,  1397.86577911,  1508.50060384]])}
```

It is worth noting that the graph generation is cached even when using `generate`, so iterative generation can be performed using normal python loops without loss in performance:

```
for i in range(10):
    weights, particles = bz.generate(n_events=1000)
    ...
    (do something with weights and particles)
    ...
```

To generate the mass of a resonance, we need to give a function as its mass instead of a floating number. This function should take as input the per-event lower mass allowed, per-event upper mass allowed and the number of events, and should return a ~‘tf.Tensor‘ with the generated masses and shape (nevents,). Well suited for this task are the TensorFlow Probability distributions or, for more customized mass shapes, the zfit pdfs *(currently an experimental feature is needed, contact the ‘zfit developers <https://github.com/zfit/zfit>‘_ to learn more).*

Following with the same example as above, and approximating the resonance shape by a gaussian, we could write the $B^0 \to K^*\gamma$ decay chain as (more details can be found in `tests/helpers/decays.py`):

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

KSTARZ_MASS = 895.81
KSTARZ_WIDTH = 47.4

  def kstar_mass(min_mass, max_mass, n_events):
     min_mass = tf.cast(min_mass, tf.float64)
     max_mass = tf.cast(max_mass, tf.float64)
```

(continues on next page)

(continued from previous page)

```
    kstar_width_cast = tf.cast(KSTARZ_WIDTH, tf.float64)
    kstar_mass_cast = tf.cast(KSTARZ_MASS, dtype=tf.float64)

    kstar_mass = tf.broadcast_to(kstar_mass_cast, shape=(n_events,))
    if kstar_width > 0:
        kstar_mass = tfp.distributions.TruncatedNormal(loc=kstar_mass,
                                                       scale=kstar_width_cast,
                                                       low=min_mass,
                                                       high=max_mass).sample()
    return kstar_mass

bz = GenParticle('B0', B0_MASS).set_children(GenParticle('K*0', mass=kstar_mass)
                                             .set_children(GenParticle('K+',␣
↪mass=KAON_MASS),
                                                           GenParticle('pi-',␣
↪mass=PION_MASS)),
                                             GenParticle('gamma', mass=0.0))
```

Shortcut for simple decays

The generation of simple *n*-body decay chains can be done using the `nbody_decay` function of `phasespace`, which takes

- The mass of the top particle.
- The mass of children particles as a list.
- The name of the top particle (optional).
- The names of the children particles (optional).

If the names are not given, *top* and *p_{i}* are assigned. For example, to generate $B^0 \to K\pi$, one would do:

```
import phasespace

N_EVENTS = 1000

B0_MASS = 5279.58
PION_MASS = 139.57018
KAON_MASS = 493.677

decay = phasespace.nbody_decay(B0_MASS, [PION_MASS, KAON_MASS],
                               top_name="B0", names=["pi", "K"])
weights, particles = decay.generate(n_events=N_EVENTS)
```

In this example, `decay` is simply a `GenParticle` with the corresponding children.

5.2.2 phasespace package

phasespace.phasespace module

Implementation of the Raubold and Lynch method to generate n-body events.

The code is based on the GENBOD function (W515 from CERNLIB), documented in

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

class phasespace.phasespace.**GenParticle**(*name: str, mass: Union[Callable, int, float]*)
Bases: object

Representation of a particle.

Instances of this class can be combined with each other to build decay chains, which can then be used to generate phase space events through the *generate* or *generate_tensor* method.

A *GenParticle* must have

- **a *name*, which is ensured not to clash with any others in** the decay chain.
- **a *mass*, which can be either a number or a function to generate it according to** a certain distribution. The returned ~'tf.Tensor' needs to have shape (nevents,). In this case, the particle is not considered as having a fixed mass and the *has_fixed_mass* method will return False.

It may also have:

- Children, ie, decay products, which are also *GenParticle* instances.

Parameters

- **name** (*str*) – Name of the particle.
- **mass** (*float, Tensor`, callable*) – Mass of the particle. If it's a float, it get converted to a *tf.constant*.

generate(*n_events: int*, *boost_to=None*, *normalize_weights: bool = True*)
Generate normalized n-body phase space as numpy arrays.

Events are generated in the rest frame of the particle, unless *boost_to* is given.

Note: In this method, the event weights are returned normalized to their maximum.

Parameters

- **n_events** (*int*) – Number of events to generate.
- **boost_to** (*optional*) – Momentum vector of shape (x, 4), where x is optional, to where the resulting events will be boosted. If not specified, events are generated in the rest frame of the particle.
- **normalize_weights** (*bool, optional*) – Normalize the event weight to its max?

Returns

Result of the generation, which varies with the value of *normalize_weights*:

- If True, the tuple elements are the normalized event weights as an array of shape

(n_events,), and the momenta generated particles as a dictionary of arrays of shape (4, n_events) with particle names as keys.

- If False, the tuple weights are the unnormalized event weights as an array of shape

(n_events,), the maximum per-event weights as an array of shape (n_events,) and the momenta generated particles as a dictionary of arrays of shape (4, n_events) with particle names as keys.

Return type tuple

Raise: tf.errors.InvalidArgumentError: If the the decay is kinematically forbidden. ValueError: If *n_events* and the size of *boost_to* don't match. See *GenParticle.generate_unnormalized*.

generate_tensor(*n_events: Union[int, tensorflow.python.framework.ops.Tensor, tensorflow.python.ops.variables.VariableV1], boost_to: Optional[tensorflow.python.framework.ops.Tensor] = None, normalize_weights: bool = True*) → Tuple[tensorflow.python.framework.ops.Tensor, Dict[str, tensorflow.python.framework.ops.Tensor]]

Generate normalized n-body phase space as tensorflow tensors.

Events are generated in the rest frame of the particle, unless *boost_to* is given.

Note: In this method, the event weights are returned normalized to their maximum.

Parameters

- **n_events** (*int*) – Number of events to generate.
- **boost_to** (*optional*) – Momentum vector of shape (x, 4), where x is optional, to where the resulting events will be boosted. If not specified, events are generated in the rest frame of the particle.
- **normalize_weights** (*bool, optional*) – Normalize the event weight to its max?

Returns

Result of the generation, which varies with the value of *normalize_weights*:

- If True, the tuple elements are the normalized event weights as a tensor of shape

(n_events,), and the momenta generated particles as a dictionary of tensors of shape (4, n_events) with particle names as keys.

- If False, the tuple weights are the unnormalized event weights as a tensor of shape

(n_events,), the maximum per-event weights as a tensor of shape (n_events,) and the momenta generated particles as a dictionary of tensors of shape (4, n_events) with particle names as keys.

Return type tuple

Raise: tf.errors.InvalidArgumentError: If the the decay is kinematically forbidden. ValueError: If *n_events* and the size of *boost_to* don't match. See *GenParticle.generate_unnormalized*.

get_mass(*min_mass: tensorflow.python.framework.ops.Tensor = None, max_mass: tensorflow.python.framework.ops.Tensor = None, n_events: Union[tensorflow.python.framework.ops.Tensor, tensorflow.python.ops.variables.VariableV1] = None*) → tensorflow.python.framework.ops.Tensor

Get the particle mass.

If the particle is resonant, the mass function will be called with the *min_mass*, *max_mass* and *n_events* parameters.

Parameters

- **min_mass** (*tensor*) – Lower mass range. Defaults to None, which is only valid in the case of fixed mass.
- **max_mass** (*tensor*) – Upper mass range. Defaults to None, which is only valid in the case of fixed mass.

- **()** (*n_events*) – Number of events to produce. Has to be specified if the particle is resonant.

Returns Mass of the particles, either a scalar or shape (nevents,)

Return type Tensor‘

Raise: ValueError: If the mass is requested and has not been set.

has_children
Does the particle have children?

Type bool

has_fixed_mass
Is the mass a callable function?

Type bool

has_grandchildren
Does the particle have grandchildren?

Type bool

set_children(**children*)
Assign children.

Parameters **children** (GenParticle) – Two or more children to assign to the current particle.

Returns self

Raise: ValueError: If there is an inconsistency in the parent/children relationship, ie, if children were already set, if their parent was or if less than two children were given. KeyError: If there is a particle name clash.

class phasespace.phasespace.**Particle**
Bases: object

Deprecated Particle class.

Renamed to GenParticle.

phasespace.phasespace.**generate_decay**(**args*, ***kwargs*)
Deprecated.

phasespace.phasespace.**nbody_decay**(*mass_top: float*, *masses: list*, *top_name: str = ”*, *names: list = None*)
Shortcut to build an n-body decay of a GenParticle.

If the particle names are not given, the top particle is called ‘top’ and the children ‘p_{i}’, where i corresponds to their position in the *masses* sequence.

Parameters

- **mass_top** (*tensor, list*) – Mass of the top particle. Can be a list of 4-vectors.
- **masses** (*list*) – Masses of the child particles.
- **name_top** (*str, optional*) – Name of the top particle. If not given, the top particle is named top.
- **names** (*list, optional*) – Names of the child particles. If not given, they are build as ‘p_{i}’, where i is given by their ordering in the *masses* list.

Returns Particle decay.

Return type *GenParticle*

Raise: ValueError: If the length of *masses* and *names* doesn't match.

phasespace.phasespace.**pdk**(*a*, *b*, *c*)
Calculate the PDK (2-body phase space) function.

Based on Eq. (9.17) in CERN 68-15 (1968).

Parameters

- **a** (*Tensor`*) – M_{i+1} in Eq. (9.17).
- **b** (*Tensor`*) – M_i in Eq. (9.17).
- **c** (*Tensor`*) – m_{i+1} in Eq. (9.17).

Returns ~'tf.Tensor'

phasespace.phasespace.**process_list_to_tensor**(*lst*)
Convert a list to a tensor.

The list is converted to a tensor and transposed to get the proper shape.

Note: If *lst* is a tensor, nothing is done to it other than convert it to *tf.float64*.

Parameters **lst** (*list*) – List to convert.

Returns ~'tf.Tensor'

phasespace.kinematics module

Basic kinematics.

phasespace.kinematics.**beta**(*vector*)
Calculate beta of a given 4-vector.

Parameters **vector** – Input Lorentz momentum vector.

phasespace.kinematics.**boost_components**(*vector*)
Get the boost components of a given 4-vector.

Parameters **vector** – Input Lorentz momentum vector.

phasespace.kinematics.**lorentz_boost**(*vector*, *boostvector*)
Perform Lorentz boost.

Parameters

- **vector** – 4-vector to be boosted
- **boostvector** – Boost vector. Can be either 3-vector or 4-vector, since only spatial components are used.

phasespace.kinematics.**lorentz_vector**(*space*, *time*)
Make a Lorentz vector from spatial and time components.

Parameters

- **space** – 3-vector of spatial components.

- **time** – Time component.

phasespace.kinematics.**mass**(*vector*)
Calculate mass scalar for Lorentz 4-momentum.

Parameters **vector** – Input Lorentz momentum vector.

phasespace.kinematics.**metric_tensor**()
Metric tensor for Lorentz space (constant).

phasespace.kinematics.**scalar_product**(*vec1*, *vec2*)
Calculate scalar product of two 3-vectors.

Parameters

- **vec1** – First vector.
- **vec2** – Second vector.

phasespace.kinematics.**spatial_component**(*vector*)
Extract spatial components of the input Lorentz vector.

Parameters **vector** – Input Lorentz vector (where indexes 0-2 are space, index 3 is time).

phasespace.kinematics.**time_component**(*vector*)
Extract time component of the input Lorentz vector.

Parameters **vector** – Input Lorentz vector (where indexes 0-2 are space, index 3 is time).

phasespace.kinematics.**x_component**(*vector*)
Extract spatial X component of the input Lorentz or 3-vector.

Parameters **vector** – Input vector.

phasespace.kinematics.**y_component**(*vector*)
Extract spatial Y component of the input Lorentz or 3-vector.

Parameters **vector** – Input vector.

phasespace.kinematics.**z_component**(*vector*)
Extract spatial Z component of the input Lorentz or 3-vector.

Parameters **vector** – Input vector.

5.2.3 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:

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 :)

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.

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.

Tips

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

```
$ pytest -k test_generate
```

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.

5.2.4 Credits

Development Lead

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

Core Developers

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

Contributors

None yet. Why not be the first?

5.2.5 Changelog

Develop

Major Features and Improvements

Behavioral changes

Bug fixes and small changes

Requirement changes

Thanks

1.0.4 (13-10-2019)

Major Features and Improvements

Release to conda-forge, thanks to Chris Burr

Python Module Index

p

Index