# vg Documentation

## Metabolize

Jan 22, 2020

# Contents

**vg** is a **v**ery **g**ood vector-geometry and linear-algebra toolbelt.

CHAPTER 1

# Motivation

Linear algebra provides a powerful toolset for a wide range of problems, including geometric problems, which are the focus of this library. Though some programmers need to know the math that powers these solutions, often understanding the abstractions is enough. Furthermore, the abstractions express these operations in highly readable forms which clearly communicate the programmer's intention, regardless of the reader's math background.

The goal of vg is to help Python programmers leverage the power of linear algebra to carry out vector-geometry operations. It produces code which is easy to write, understand, and maintain.

NumPy is powerful – and also worth learning! However it's easy to get slowed down by technical concerns like broadcasting and indexing, even when working on basic geometric operations. vg is friendlier: it's NumPy for humans. It checks that your inputs are structured correctly for the narrower use case of 3D geometry, and then it "just works." For example, `vg.euclidean_distance()` is invoked the same for two stacks of points, a stack and a point, or a simple pair of points.

In the name of readability, efficiency is not compromised. You'll find these operations as suited to production code as one-off scripts. If you find anything is dramatically slower than a lower-level equivalent, please let us know so we can fix it!

CHAPTER 2

# Functions

All functions are optionally vectorized, meaning they accept single inputs and stacks of inputs interchangeably. They return The Right Thing – a single result or a stack of results – without the need to reshape inputs or outputs. With the power of NumPy, the vectorized functions are fast.

vg.**normalize**(*vector*)
Return the vector, normalized.

If vector is 2d, treats it as stacked vectors, and normalizes each one.

vg.**perpendicular**(*v1*, *v2*, *normalized=True*)
Given two noncollinear vectors, return a vector perpendicular to both.

Result vectors follow the right-hand rule. When the right index finger points along *v1* and the right middle finger along *v2*, the right thumb points along the result.

When one or both sets of inputs is stacked, compute the perpendicular vectors elementwise, returning a stacked result. (e.g. when *v1* and *v2* are both stacked, *result[k]* is perpendicular to *v1[k]* and *v2[k]*.)

**Parameters**

- **v1** (*np.arraylike*) – A *3x1* vector or a *kx3* stack of vectors.
- **v2** (*np.arraylike*) – A *3x1 vector or a 'kx3* stack of vectors. If stacked, the shape must be the same as *v1*.
- **normalized** (*bool*) – When *True*, the result vector is guaranteed to be unit length.

**Returns** An array with the same shape as *v1* and *v2*.

**Return type** np.arraylike

**See also:**

- https://en.wikipedia.org/wiki/Cross_product#Definition
- https://commons.wikimedia.org/wiki/File:Right_hand_rule_cross_product.svg

vg.**project**(*vector*, *onto*)
Compute the vector projection of *vector* onto the vector *onto*.

*onto* need not be normalized.

vg.**scalar_projection**(*vector*, *onto*)
Compute the scalar projection of *vector* onto the vector *onto*.

*onto* need not be normalized.

vg.**reject**(*vector*, *from_v*)
Compute the vector rejection of *vector* from *from_v* – i.e. the vector component of *vector* perpendicular to *from_v*.

*from_v* need not be normalized.

vg.**reject_axis**(*vector*, *axis*, *squash=False*)
Compute the vector component of *vector* perpendicular to the basis vector specified by *axis*. 0 means x, 1 means y, 2 means z.

In other words, return a copy of vector that zeros the *axis* component.

When *squash* is True, instead of zeroing the component, it drops it, so an input vector (in R3) is mapped to a point in R2.

(N.B. Don't be misled: this meaning of *axis* is pretty different from the typical meaning in numpy.)

vg.**magnitude**(*vector*)
Compute the magnitude of *vector*. For stacked inputs, compute the magnitude of each one.

**Parameters** **vector** (*np.arraylike*) – A *3x1* vector or a *kx3* stack of vectors.

**Returns**

**For *3x1* inputs, a *float* with the magnitude. For *kx1*** inputs, a *kx1* array.

**Return type** object

vg.**euclidean_distance**(*v1*, *v2*)
Compute Euclidean distance, which is the distance between two points in a straight line. This can be done individually by passing in single point for either or both arguments, or pairwise by passing in stacks of points.

**Parameters**

- **v1** (*np.arraylike*) – A *3x1* vector or a *kx3* stack of vectors.
- **v2** (*np.arraylike*) – A *3x1* vector or a *kx3* stack of vectors. If stacks are provided for both *v1* and *v2* they must have the same shape.

**Returns**

**When both inputs are *3x1*, a *float* with the distance. Otherwise** a *kx1* array.

**Return type** object

vg.**angle**(*v1*, *v2*, *look=None*, *assume_normalized=False*, *units='deg'*)
Compute the unsigned angle between two vectors. For stacked inputs, the angle is computed pairwise.

When *look* is provided, the angle is computed in that viewing plane (*look* is the normal). Otherwise the angle is computed in 3-space.

**Parameters**

- **v1** (*np.arraylike*) – A *3x1* vector or a *kx3* stack of vectors.
- **v2** (*np.arraylike*) – A vector or stack of vectors with the same shape as *v1*.

- **look** (*np.arraylike*) – A *3x1* vector specifying the normal of a viewing plane, or *None* to compute the angle in 3-space.
- **assume_normalized** (*bool*) – When *True*, assume the input vectors are unit length. This improves performance, however when the inputs are not normalized, setting this will cause an incorrect results.
- **units** (*str*) – *'deg'* to return degrees or *'rad'* to return radians.

**Returns**

**For *3x1* inputs, a *float* with the angle. For *kx1* inputs,** a *kx1* array.

**Return type** object

vg.**signed_angle**(*v1*, *v2*, *look*, *units='deg'*)

Compute the signed angle between two vectors. For stacked inputs, the angle is computed pairwise.

Results are in the range -180 and 180 (or *-math.pi* and *math.pi*). A positive number indicates a clockwise sweep from *v1* to *v2*. A negative number is counterclockwise.

**Parameters**

- **v1** (*np.arraylike*) – A *3x1* vector or a *kx3* stack of vectors.
- **v2** (*np.arraylike*) – A vector or stack of vectors with the same shape as *v1*.
- **look** (*np.arraylike*) – A *3x1* vector specifying the normal of the viewing plane.
- **units** (*str*) – *'deg'* to return degrees or *'rad'* to return radians.

**Returns**

**For *3x1* inputs, a *float* with the angle. For *kx1* inputs,** a *kx1* array.

**Return type** object

vg.**rotate**(*vector*, *around_axis*, *angle*, *units='deg'*, *assume_normalized=False*)

Rotate a point or vector around a given axis. The direction of rotation around *around_axis* is determined by the right-hand rule.

**Parameters**

- **vector** (*np.arraylike*) – A *3x1* vector or a *kx3* stack of vectors.
- **around_axis** (*np.arraylike*) – A *3x1* vector specifying the axis of rotation.
- **assume_normalized** (*bool*) – When *True*, assume *around_axis* is unit length. This improves performance marginally, however when the inputs are not normalized, setting this will cause an incorrect results.
- **units** (*str*) – *'deg'* to specify *angle* in degrees or *'rad'* to specify radians.

**Returns**

**The transformed point or points. This has the same shape as** *vector*.

**Return type** np.arraylike

**See also:**

- https://en.wikipedia.org/wiki/Cross_product#Definition
- https://commons.wikimedia.org/wiki/File:Right_hand_rule_cross_product.svg

vg.**scale_factor**(*v1*, *v2*)

Given two parallel vectors, compute the scale factor *k* such that *k * v1* is approximately equal to *v2*.

**Parameters**

- **v1** (*np.arraylike*) – A vector in *R^3* or a *kx3* stack of vectors.
- **v2** (*np.arraylike*) – A second vector in *R^3* or a *kx3* stack of vectors. If *v1* and *v2* are both stacked, they must be the same shape.

**Returns** A float containing the scale factor *k*, or *nan* if *v1* is the zero vector. If either input is stacked, the result will also be stacked.

**Return type** object

vg.**orient**(*vector*, *along*, *reverse=False*)

Given two vectors, flip the first if necessary, so that it points (approximately) along the second vector rather than (approximately) opposite it.

**Parameters**

- **vector** (*np.arraylike*) – A vector in *R^3*.
- **along** (*np.arraylike*) – A second vector in *R^3*.
- **reverse** (*bool*) – When *True*, reverse the logic, returning a vector that points against *along*.

**Returns** Either *vector* or *-vector*.

**Return type** np.arraylike

vg.**almost_zero**(*v*, *atol=1e-08*)

Test if v is almost the zero vector.

vg.**almost_unit_length**(*vector*, *atol=1e-08*)

Test if the *vector* has almost unit length. For stacked inputs, test each one.

**Parameters** **vector** (*np.arraylike*) – A *3x1* vector or a *kx3* stack of vectors.

**Returns** For *3x1* inputs, a *bool*. For *kx1* inputs, a *kx1* array.

**Return type** object

vg.**almost_collinear**(*v1*, *v2*, *atol=1e-08*)

Test if *v1* and *v2* are almost collinear.

This will return true if either *v1* or *v2* is the zero vector, because mathematically speaking, the zero vector is collinear to everything.

Geometrically that doesn't necessarily make sense, so if you want to handle zero vectors specially, you can test your inputs with *vg.almost_zero()*.

vg.**almost_equal**(*v1*, *v2*, *atol=1e-08*)

Test if *v1* and *v2* are equal within the given absolute tolerance.

**See also:**

- https://docs.scipy.org/doc/numpy/reference/generated/numpy.allclose.html

vg.**principal_components**(*coords*)

Compute the principal components of the input coordinates. These are useful for dimensionality reduction and feature modeling.

**Parameters** **coords** (*np.arraylike*) – A *nxk* stack of coordinates.

**Returns** A *kxk* stack of vectors.

**Return type** np.ndarray

**See also:**

- http://setosa.io/ev/principal-component-analysis/
- https://en.wikipedia.org/wiki/Principal_component_analysis
- https://plot.ly/ipython-notebooks/principal-component-analysis/

vg.**major_axis**(*coords*)

Compute the first principal component of the input coordinates. This is the vector which best describes the multidimensional data using a single dimension.

**Parameters** **coords** (*np.arraylike*) – A *nxk* stack of coordinates.

**Returns** A *kx1* vector.

**Return type** np.ndarray

**See also:**

- http://setosa.io/ev/principal-component-analysis/
- https://en.wikipedia.org/wiki/Principal_component_analysis
- https://plot.ly/ipython-notebooks/principal-component-analysis/

vg.**apex**(*points*, *along*)

Find the most extreme point in the direction provided.

**Parameters**

- **points** (*np.arraylike*) – A *kx3* stack of points in R^3.
- **along** (*np.arraylike*) – A *3x1* vector specifying the direction of interest.

**Returns** A copy of a point taken from *points*.

**Return type** np.ndarray

vg.**nearest**(*from_points*, *to_point*, *ret_index=False*)

Find the point nearest to the given point.

**Parameters**

- **from_points** (*np.arraylike*) – A *kx3* stack of points in R^3.
- **to_point** (*np.arraylike*) – A *3x1* point of interest.
- **ret_index** (*bool*) – When *True*, return both the point and its index.

**Returns** A *3x1* vector taken from *from_points*.

**Return type** np.ndarray

vg.**farthest**(*from_points*, *to_point*, *ret_index=False*)

Find the point farthest from the given point.

**Parameters**

- **from_points** (*np.arraylike*) – A *kx3* stack of points in R^3.
- **to_point** (*np.arraylike*) – A *3x1* point of interest.
- **ret_index** (*bool*) – When *True*, return both the point and its index.

**Returns** A *3x1* vector taken from *from_points*.

**Return type** np.ndarray

vg.**within**(*points*, *radius*, *of_point*, *atol=1e-08*, *ret_indices=False*)

Select points within a given radius of a point.

**Parameters**

- **points** (*np.arraylike*) – A *kx3* stack of points in R^3.
- **radius** (*float*) – The radius of the sphere of interest centered on *of_point*.
- **of_point** (*np.arraylike*) – The *3x1* point of interest.
- **atol** (*float*) – The distance tolerance. Points within *radius* + *atol* of *of_point* are selected.
- **ret_indexes** (*bool*) – When *True*, return both the points and their indices.

**Returns** A *3x1* vector taken from *points*.

**Return type** np.ndarray

vg.**average**(*values*, *weights=None*, *ret_sum_of_weights=False*)

Compute a weighted or unweighted average of the 3D input values. The inputs could be points or vectors.

**Parameters**

- **values** (*np.arraylike*) – A *kx3* stack of vectors.
- **weights** (*array-convertible*) – An optional *k* array of weights.
- **ret_sum_of_weights** (*bool*) – When *True*, the sum of the weights is returned. When *weights* is *None*, this is the number of elements over which the average is taken.

**Returns** A *(3,)* vector with the weighted or unweighted average.

**Return type** np.ndarray

vg.**cross**(*v1*, *v2*)

Compute individual or pairwise cross products.

**Parameters**

- **v1** (*np.arraylike*) – A *3x1* vector or a *kx3* stack of vectors.
- **v2** (*np.arraylike*) – A *3x1* vector or a *kx3* stack of vectors. If stacks are provided for both *v1* and *v2* they must have the same shape.

vg.**dot**(*v1*, *v2*)

Compute individual or pairwise dot products.

**Parameters**

- **v1** (*np.arraylike*) – A *3x1* vector or a *kx3* stack of vectors.
- **v2** (*np.arraylike*) – A *3x1* vector or a *kx3* stack of vectors. If stacks are provided for both *v1* and *v2* they must have the same shape.

vg.matrix.**pad_with_ones**(*matrix*)

**Add a column of ones. Transform from:**

**array([[1., 2., 3.],** [2., 3., 4.], [5., 6., 7.]])

**to:**

**array([[1., 2., 3., 1.],** [2., 3., 4., 1.], [5., 6., 7., 1.]])

vg.matrix.**transform**(*vertices*, *transform*)
Apply the given transformation matrix to the vertices using homogenous coordinates.

vg.matrix.**unpad**(*matrix*)

**Strip off a column (e.g. of ones). Transform from:**

**array([[1., 2., 3., 1.],** [2., 3., 4., 1.], [5., 6., 7., 1.]])

**to:**

**array([[1., 2., 3.],** [2., 3., 4.], [5., 6., 7.]])

vg.shape.**check**(*locals_namespace*, *name*, *shape*)
Convenience function for invoking *vg.shape.check_value()* with a *locals()* dict.

**Parameters**

- **namespace** (*dict*) – A subscriptable object, typically *locals()*.
- **name** (*str*) – Key to pull from *namespace*.
- **shape** (*list*) – Shape to validate. To require 3 by 1, pass *(3,)*. To require n by 3, pass *(-1, 3)*.

**Returns** The wildcard dimension (if one) or a tuple of wildcard dimensions (if more than one).

**Return type** object

### Example

```
>>> def my_fun_function(points):
...     vg.shape.check(locals(), 'points', (-1, 3))
...     # Proceed with confidence that `points` is a k x 3 array.
```

### Example

```
>>> def my_fun_function(points):
...     k = vg.shape.check(locals(), 'points', (-1, 3))
...     print("my_fun_function invoked with {} points".format(k))
```

vg.shape.**check_value**(*a*, *shape*, ***kwargs*)
Check that the given argument has the expected shape. Shape dimensions can be ints or -1 for a wildcard. The wildcard dimensions are returned, which allows them to be used for subsequent validation or elsewhere in the function.

**Parameters**

- **a** (*np.arraylike*) – An array-like input.
- **shape** (*list*) – Shape to validate. To require 3 by 1, pass *(3,)*. To require n by 3, pass *(-1, 3)*.
- **name** (*str*) – Variable name to embed in the error message.

**Returns** The wildcard dimension (if one) or a tuple of wildcard dimensions (if more than one).

**Return type** object

### Example

```
>>> vg.shape.check_value(np.zeros((4, 3)), (-1, 3))
>>> # Proceed with confidence that `points` is a k x 3 array.
```

### Example

```
>>> k = vg.shape.check_value(np.zeros((4, 3)), (-1, 3))
>>> k
4
```

CHAPTER 3

# Constants

vg.**basis**
: The cartesian basis vectors.

```
vg.basis.x = array([1., 0., 0.])
vg.basis.neg_x = array([-1., 0., 0.])
vg.basis.y = array([0., 1., 0.])
vg.basis.neg_y = array([ 0., -1., 0.])
vg.basis.z = array([0., 0., 1.])
vg.basis.neg_z = array([ 0., 0., -1.])
```

# CHAPTER 4

# Style guide

Use the named secondary arguments. They tend to make the code more readable:

```
import vg
result = vg.proj(v1, onto=v2)
```

CHAPTER 5

# Design principles

Linear algebra is useful and it doesn't have to be dificult to use. With the power of abstractions, simple operations can be made simple, without poring through lecture slides, textbooks, inscrutable Stack Overflow answers, or dense NumPy docs. Code that uses linear algebra and geometric transformation should be readable like English, without compromising efficiency.

These common operations should be abstracted for a few reasons:

1. If a developer is not programming linalg every day, they might forget the underlying formula. These forms are easier to remember and more easily referenced.
2. These forms tend to be self-documenting in a way that the NumPy forms are not. If a developer is not programming linalg every day, this will again come in handy.
3. These implementations are more robust. They automatically inspect `ndim` on their arguments, so they work equally well if the argument is a vector or a stack of vectors. They are more careful about checking edge cases like a zero norm or zero cross product and returning a correct result or raising an appropriate error.

CHAPTER 6

# Versioning

This library adheres to Semantic Versioning.

# Python Module Index

## V

# Index