Numpy, Scipy Links

- Scipy Lecture Notes
- Numpy
- SciPy
- Matplotlib

Numpy Arrays

Arrays in `numpy`

- have elements all of the same type and occupy the same amount of storage, an element can be composed of other simple types,
- have a fixed size (cannot grow like lists),
- have a shape specified with a tuple, the tuple gives the size of each dimension,
- are indexed by integers.

A `numpy` array is very similar to arrays in static languages like C , Fortran, and Java. Arrays in `numpy` are multi-dimensional. Operations on arrays are, in general, faster than lists. Why?

Python meets APL (A Programming Language).

Creating arrays

Arrays can be created with python sequences or initialized with constant values of 0 or 1, or uninitialized. Some of the array element types are byte, int, float, complex, uint8, uint16, uint64, int8, int16, int32, int64, float32, float64, float96, complex64, complex128, and complex192.

```
>>> import numpy as np
>>> np.zeros( (12), dtype=np.int8 )
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], dtype=int8)
>>> # or np.zeros( 12, dtype=np.int8 )
>>> np.zeros( (2,6), dtype=np.int16 )
array([[0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0]], dtype=int16)
>>> np.ones( (6,2), dtype=np.int32 )
array([[1, 1],
       [1, 1],
       [1, 1],
       [1, 1],
       [1, 1],
       [1, 1]], dtype=int32)
>>> np.ones( (2,2,3), dtype=np.int64 )
array([[[1, 1, 1],
        [1, 1, 1]],

       [[1, 1, 1],
        [1, 1, 1]]])
>>> a = np.ones( (2,3,2) )
>>> a.dtype # type of each element
dtype('float64')
>>> a.ndim  # number of dimension
3
>>> a.shape # tuple of dimension sizes
(2, 3, 2)
>>> a.size # total number of elements
12
>>> a.itemsize # number of bytes of storage per element
8
>>> np.array( [ [1,2,3], [4,5,6] ] )
array([[1, 2, 3],
       [4, 5, 6]])
>>> a = _
>>> a.dtype
dtype('int64')
>>> a.shape
(2, 3)
>>> np.array( range(7), np.float32 )
array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.], dtype=float32)
>>> # identity 2d array
>>> np.eye( 3 )
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
>>> # list comprehension
>>> np.array( [ [ int(i != j) for j in range(3)] for i in range(4)] )
array([[0, 1, 1],
       [1, 0, 1],
       [1, 1, 0],
       [1, 1, 1]])
>>> _.shape
(4, 3)
>>> np.empty( (2,3), dtype=np.int16) # uninitialized, any advantages
array([[0, 0, 0],
       [0, 0, 0]], dtype=int16)
>>>
```

Finite Integer Data Types in numpy

Unlike integers in python, all numpy integers have fixed ranges.

```
>>> import numpy as np
>>> b = np.zeros( 2, dtype=np.int8 )
>>> type(b)
<class 'numpy.ndarray'>
>>> b[0] = 47
>>> b[0]
47
>>> b[0] += 255
>>> b[0]
46
>>> b[0] + 255
301
>>> # why are the above two values different.
>>> b[0] = 1
>>> b[0] += 254
>>> b[0]
-1
>>> # why is it negative?
>>> b[0] = 42
>>> b[0] += 128
>>> b[0]
-86
>>> b[0] += 128
>>> b[0]
42
>>> b[0] = 128
>>> b[0]
-128
```

Also, unlike regular python, numpy has unsigned integers.

```
>>> import numpy as np
>>> b = np.zeros( 2, dtype=np.uint8 )
>>> type(b)
<class 'numpy.ndarray'>
>>> b[0] = 47
>>> b[0]
47
>>> b[0] += 255
>>> b[0]
46
>>> b[0] + 255
301
>>> # why are the above two values different.
>>> b[0] = 1
>>> b[0] += 254
>>> b[0]
255
>>> # why is different than above?
```

numpy support many different data types (dtype) .

Reshaping arrays

Arrays in `numpy` can be reshaped. The reshaped array must have the same number or elements. Why?

```
>>> import numpy as np
>>> a = np.array( range(12) ) # equivalent to np.arange( 12, dtype=np.int64)
>>> a
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
>>> a.shape
(12,)
>>> b = a.reshape(2, 6) # b still refers to a's memory
>>> a.shape
(12,)
>>> b.shape
(2, 6)
>>> a[6] = 106 # access 7th element
>>> b # changes in a are reflected in b
array([[  0,   1,   2,   3,   4,   5],
       [106,   7,   8,   9,  10,  11]])
>>> a
array([  0,   1,   2,   3,   4,   5, 106,   7,   8,   9,  10,  11])
>>> b[1,0] = 6 # changes in b are reflected in a
>>> b
array([[ 0,  1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10, 11]])
>>> a
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
>>> c = b.transpose() # swaps the indices order
>>> c.shape
(6, 2)
```

```
>>> b[1,4]
10
>>> c[4,1]
10
>>>
```

Indexing and slicing arrays

Arrays are indexed with comma separated list of indices. Unlike list, slices do not copy the array, but provide another view into the same data.

```
>>> import numpy as np
>>> t = np.array( range(24), np.uint8 ) # unsigned 8 bit integer
>>> t
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
       17, 18, 19, 20, 21, 22, 23], dtype=uint8)
>>> t = t.reshape(2,3,4)
>>> t
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]],

       [[12, 13, 14, 15],
        [16, 17, 18, 19],
        [20, 21, 22, 23]]], dtype=uint8)
>>> t[0,0,0] # index = i*(3*4) + j*4 + k
0
>>> t[1,1,1]
17
>>> t[1,2,3] # the last index advances serially
23
>>> # the transpose reverses the order of the indices
>>> s = t.transpose()
>>> s
array([[[ 0, 12],
        [ 4, 16],
        [ 8, 20]],

       [[ 1, 13],
        [ 5, 17],
        [ 9, 21]],

       [[ 2, 14],
        [ 6, 18],
        [10, 22]],

       [[ 3, 15],
        [ 7, 19],
        [11, 23]]], dtype=uint8)
>>> s.shape
(4, 3, 2)
>>> s[3,2,1] # index i + j*4 + k*(3*4)
23
>>> # change s
>>> s[3,2,1] = 255
>>> s[3,2,1]
255
>>> t[1,2,3] # s and t reference the same array
255
>>> # in the transpose the first index advances serially
>>> s[0,1,0]
4
>>> s[1,1,0]
5
>>> # slice examples
>>> t1 = t[1,2] # partial indices return slices
>>> t1
array([ 20,  21,  22, 255], dtype=uint8)
>>> t1.shape
(4,)
>>> t1[0] = 100 # in numpy a slice is similar to a reshape
>>> t           # t1 references the same array
array([[[  0,   1,   2,   3],
        [  4,   5,   6,   7],
        [  8,   9,  10,  11]],

       [[ 12,  13,  14,  15],
        [ 16,  17,  18,  19],
        [100,  21,  22, 255]]], dtype=uint8)
>>> t = np.array( range(24), np.uint8 ) ; t = t.reshape(2,3,4) # reset
>>> t[0,1]
array([4, 5, 6, 7], dtype=uint8)
>>> t[0,1] = 200 # assignment to slices changes all elements
>>> t
array([[[  0,   1,   2,   3],
        [200, 200, 200, 200],
        [  8,   9,  10,  11]],

       [[ 12,  13,  14,  15],
        [ 16,  17,  18,  19],
        [ 20,  21,  22,  23]]], dtype=uint8)
>>>
```

Multi-dimensional indexing

More than one dimension can be indexed.

```
>>> import numpy as np
>>> z = np.zeros((5,5),np.int); z # 5 x 5 integers
array([[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]])
>>> z[0] = 1; z # first row, 1 is broadcast to all entries
array([[1, 1, 1, 1, 1],
       [0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0]])
>>> z[:,0] = 2; z # first column, 2 is broadcast
array([[2, 1, 1, 1, 1],
       [2, 0, 0, 0, 0],
       [2, 0, 0, 0, 0],
       [2, 0, 0, 0, 0],
       [2, 0, 0, 0, 0]])
>>> z[2:4,2:5] = 3; z # (2,3) block, 3 is broadcast
array([[2, 1, 1, 1, 1],
       [2, 0, 0, 0, 0],
       [2, 0, 3, 3, 3],
       [2, 0, 3, 3, 3],
       [2, 0, 0, 0, 0]])
>>> z[:,-1] = 4; z # last column, 4 is broadcast
array([[2, 1, 1, 1, 4],
       [2, 0, 0, 0, 4],
       [2, 0, 3, 3, 4],
       [2, 0, 3, 3, 4],
       [2, 0, 0, 0, 4]])
>>> z[-1,:] = 5; z # last row, 5 is broadcast
array([[2, 1, 1, 1, 4],
       [2, 0, 0, 0, 4],
       [2, 0, 3, 3, 4],
       [2, 0, 3, 3, 4],
       [5, 5, 5, 5, 5]])
>>>
```

Matrix

A numpy matrix is a specialized 2-D array.

```
>>> import numpy as np
>>> a = np.matrix('7 0 0; 0 7 0; 0 0 7')
>>> a
matrix([[7, 0, 0],
        [0, 7, 0],
        [0, 0, 7]])
>>> a.T  # transpose
matrix([[7, 0, 0],
        [0, 7, 0],
        [0, 0, 7]])
>>> a.I  # inverse
matrix([[ 0.14285714,  0.        ,  0.        ],
        [ 0.        ,  0.14285714,  0.        ],
        [ 0.        ,  0.        ,  0.14285714]])
>>> a.I.I  # inverse of inverse
matrix([[ 7.,  0.,  0.],
        [ 0.,  7.,  0.],
        [ 0.,  0.,  7.]])
>>> a.shape # shape
(3, 3)
>>> a.size  # number of elements
9
```

Arithmetic, Vector and Matrix Operations

The standard math operators can be used with `numpy` arrays. Matrix operations are also defined.

```
>>> import numpy as np
>>> i = np.identity( 3, np.int16 )
>>> i
array([[1, 0, 0],
       [0, 1, 0],
       [0, 0, 1]], dtype=int16)
>>> i + i # add element to element
array([[2, 0, 0],
       [0, 2, 0],
       [0, 0, 2]], dtype=int16)
>>> i + 4 # add a scalar to every entry
array([[5, 4, 4],
       [4, 5, 4],
       [4, 4, 5]], dtype=int16)
>>> a = np.array( range(1,10), np.int16 ) # equiv to arange(1,10,dtype=int16):w
>>> a = a.reshape(3,3)
>>> i * a # element to element
array([[1, 0, 0],
       [0, 5, 0],
       [0, 0, 9]], dtype=int16)
>>> x = np.array( [1,2,3], np.int32 )
>>> x
array([1, 2, 3], dtype=int32)
```

```
>>> y = np.array( [ [4], [5], [6] ], np.int32 )
>>> y
array([[4],
       [5],
       [6]], dtype=int32)
>>> x + y
array([[5, 6, 7],
       [6, 7, 8],
       [7, 8, 9]], dtype=int32)
>>> # what about y + x
>>> y + x
array([[5, 6, 7],
       [6, 7, 8],
       [7, 8, 9]], dtype=int32)
>>> np.array([4]) + np.array([1,2,3]) # one element with three
array([5, 6, 7])
>>> # 1 and 2d arrays can be turned into matrices with mat()
>>> # matrices redefine * to be dot
>>> mx = np.mat(x)
>>> mx.shape
(1, 3)
>>> my = np.mat(y)
>>> my.shape
(3, 1)
>>> mx * my  # matrix multiplication
matrix([[32]], dtype=int32)
>>> my * mx  # matrix multiplication
matrix([[ 4,  8, 12],
        [ 5, 10, 15],
        [ 6, 12, 18]], dtype=int32)
>>>
```

Implementing Matrix Multiply

```
>>> import numpy as np
>>>
>>> def mat_mult( m1, m2 ) :
...     r1, c1 = m1.shape
...     r2, c2 = m2.shape
...     # columns in m1 must match rows in m2
...     if c1 != r2 :
...         raise Exception('bad dimensions')
...     res = np.empty( (r1, c2), m1.dtype )
...     # aligns slices
...     tm2 = m2.transpose()
...     for i in range(r1) :
...         for j in range(c2) :
...             # get vector slices and perform dot
...             res[i,j] = np.dot(m1[i], tm2[j])
...     return res
...
>>> m1 = np.array( [[1,2,3], [4,5,6], [7,8,9]] )
>>> m2 = np.array( [[2,1,1 ], [1,2,1], [1,1,2]] )
>>> m3 = np.array( [[1,2],[3,4]] )
>>> m4 = np.array( [[1,2,3,4,5], [10,20,30,40,50]] )
>>> m5 = np.array( [[1,2], [3,4], [5,6], [7,8], [9,0]] )
>>>
>>> print( mat_mult( m1, m2 ) )
[[ 7  8  9]
 [19 20 21]
 [31 32 33]]
>>> print( np.mat(m1) * np.mat(m2) )
[[ 7  8  9]
 [19 20 21]
 [31 32 33]]
>>> print( mat_mult( m3, m4 ) )
[[ 21  42  63  84 105]
 [ 43  86 129 172 215]]
>>> print( np.mat(m3) * np.mat(m4) )
[[ 21  42  63  84 105]
 [ 43  86 129 172 215]]
>>> print( mat_mult( m5, m3 ) )
[[ 7 10]
 [15 22]
 [23 34]
 [31 46]
 [ 9 18]]
>>> print( np.mat(m5) * np.mat(m3) )
[[ 7 10]
 [15 22]
 [23 34]
 [31 46]
 [ 9 18]]
>>> print( mat_mult( m5, m4 ) )
[[ 21  42  63  84 105]
 [ 43  86 129 172 215]
 [ 65 130 195 260 325]
 [ 87 174 261 348 435]
 [  9  18  27  36  45]]
>>> print( np.mat(m5) * np.mat(m4) )
[[ 21  42  63  84 105]
 [ 43  86 129 172 215]
 [ 65 130 195 260 325]
```

```
 [ 87 174 261 348 435]
 [  9  18  27  36  45]]
```

Relational operators

Relational operators are also defined for `numpy` arrays. An array of booleans is returned.

```
>>> import numpy as np
>>> a = np.array([3,6,8,9])
>>> a == 6
array([False,  True, False, False], dtype=bool)
>>> a >= 7
array([False, False,  True,  True], dtype=bool)
>>> a < 5
array([ True, False, False, False], dtype=bool)
>>> # count all the even numbers
>>> np.sum( (a%2) == 0 )
2
>>> b = np.array([2,6,7,10])
>>> a == b
array([False,  True, False, False], dtype=bool)
>>> a < b
array([False, False, False,  True], dtype=bool)
>>> # any is true if any value is true
>>> np.any( a == 3 )
True
>>> np.any( a == 10 )
False
>>> a == a
array([ True,  True,  True,  True], dtype=bool)
>>> # true if all values are true
>>> np.all( a == a )
True
>>> np.all( a != b )
False
>>>
```

Indexing with boolean arrays

In addition to slices, a numpy array can be indexed with an equivalently shaped boolean array (i.e., their shape are the same).

```
>>> import numpy as np
>>> a = np.arange(1, 13).reshape(3,4); a
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
>>> i1 = (a % 2) == 1; i1 # odd entries
array([[ True, False,  True, False],
       [ True, False,  True, False],
       [ True, False,  True, False]], dtype=bool)
>>> a[ i1 ] # select all the odd entries
array([ 1,  3,  5,  7,  9, 11])
>>> a[ i1 ].shape # notice anything?
(6,)
>>> i2 = (a >= 1) & (a < 4) ; i2 # and
array([[ True,  True,  True, False],
       [False, False, False, False],
       [False, False, False, False]], dtype=bool)
>>> a[ i2 ] # values greater than or equal to 1 and less than 4
array([1, 2, 3])
>>>
```

linspace, logspace, geomspace

Linear, logarithmic, geometric sequences can be generated with the following:

```
>>> import numpy as np
>>> np.linspace(0.0, 10.0, 11)
array([  0.,   1.,   2.,   3.,   4.,   5.,   6.,   7.,   8.,   9.,  10.])
>>> np.linspace(0.0, 10.0, 6)
array([  0.,   2.,   4.,   6.,   8.,  10.])
>>> np.linspace(0.0, 10.0, 6, dtype='i2')
array([ 0,  2,  4,  6,  8, 10], dtype=int16)
>>> np.linspace(0.0, 10.0, 6, dtype='i4')
array([ 0,  2,  4,  6,  8, 10], dtype=int32)
>>> np.linspace(0.0, 10.0, 6, dtype='f2')
array([  0.,   2.,   4.,   6.,   8.,  10.], dtype=float16)
>>> np.linspace(0.0, 10.0, 6, dtype='f4')
array([  0.,   2.,   4.,   6.,   8.,  10.], dtype=float32)
>>> np.linspace(0.0, 10.0, 6, dtype='f8')
array([  0.,   2.,   4.,   6.,   8.,  10.])
>>> np.linspace(0.0, 10.0, 6, dtype='f16')
array([ 0.0,  2.0,  4.0,  6.0,  8.0,  10.0], dtype=float128)
```

`np.logspace` is equivalent to:

```
y = np.linspace(start, stop, num=num)
np.power(base, y)
```

```
>>> import numpy as np
>>> np.logspace(0,2,3,base=10)
array([   1.,   10.,  100.])
>>> np.logspace(0,7,8,base=2)
array([   1.,    2.,    4.,    8.,   16.,   32.,   64.,  128.])
```

`np.geomspace` specifies the start and end values of a geometric sequence.

```
>>> import numpy as np
>>> np.geomspace(1,128,8)
array([   1.,    2.,    4.,    8.,   16.,   32.,   64.,  128.])
>>> np.geomspace(1,100,3)
array([   1.,   10.,  100.])
```

Math functions

Most of the math functions can be used with `numpy` arrays. These functions are actually redefined. Some of the redefined math functions are: arccos, arcsin, arctan, arctan2, ceil, cos, cosh, degrees, exp, fabs, floor, fmod, frexp, hypot, ldexp, log, log2, log10, modf, power, radians, sin, sinh, sqrt, tan, and tanh.

```
>>> import numpy as np
>>> t = np.linspace(0, np.pi, 7) # 7 values linearly between 0 and pi
>>> t
array([ 0.        ,  0.52359878,  1.04719755,  1.57079633,  2.0943951 ,
        2.61799388,  3.14159265])
>>> np.degrees( t )
array([   0.,   30.,   60.,   90.,  120.,  150.,  180.])
>>> st = np.sin(t)
>>> st
array([  0.00000000e+00,   5.00000000e-01,   8.66025404e-01,
         1.00000000e+00,   8.66025404e-01,   5.00000000e-01,
         1.22464680e-16])
>>> ast = np.arcsin( st )
>>> ast - t
array([ 0.        ,  0.        ,  0.        ,  0.        , -1.04719755,
       -2.0943951 , -3.14159265])
>>> np.degrees( ast - t )
array([   0.,    0.,    0.,    0.,  -60., -120., -180.])
>>> # why are there non-zero entries
>>> np.floor( t )
array([ 0.,  0.,  1.,  1.,  2.,  2.,  3.])
>>> np.log( t + 1 ) # adding 1 avoids log(0)
array([ 0.        ,  0.42107515,  0.71647181,  0.94421571,  1.12959244,
        1.28591969,  1.42108041])
>>> np.linspace(0.0,180.0, 7 )
array([   0.,   30.,   60.,   90.,  120.,  150.,  180.])
>>> np.radians( np.linspace(0.0,180.0, 7 ) )
array([ 0.        ,  0.52359878,  1.04719755,  1.57079633,  2.0943951 ,
        2.61799388,  3.14159265])
>>> np.log2( np.geomspace(1,128,8) )
array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.])
>>>
```

Random and Statistics In Standard python

Standard python can generate random sequences with the `random` module. This sequence can be operated on with `sum`, `max`, and `min`.

```
>>> import random
>>> mu = 0.0
>>> sigma = 1.0
>>> s = [ random.normalvariate(mu, sigma) for _ in range(20) ]
>>> print( '\n'.join([ str(x) for x in s]) )
-1.2478812908738406
-2.1429336047666436
0.2806013378054951
-1.5637259368908056
-0.02571920252353633
-1.2154549265527368
0.909805671311639
1.0514289620159762
-0.3740234823084445
1.4358221362743957
-0.5038243851105205
-0.9009404457093462
0.41289679255556033
1.4334025204449603
0.03152555317996402
-1.6286237650771551
0.5403884403529654
-0.0032843394509644375
0.06658562290770029
0.9096519268026145
>>> sum(s)
-2.534302415612724
```

```
>>> max(s)
1.4358221362743957
>>> min(s)
-2.1429336047666436
>>> sum(s)/len(s)
-0.1267151207806362
>>>
```

Python 3 also provides the `statistics` module.

```
>>> import statistics as st
>>> st.mean(s)   # average, how close to 0?
-0.12671512078063613
>>> st.median(s) # middle value
0.014120606864499792
>>> # does the mode make sense for floating point
>>> st.pstdev( s ) # population standard deviation, how close to 1?
1.0254982442712195
>>> st.pvariance( s ) # population variance
1.0516466490033538
```

Changing the mean and std of mu=0.0 and signma=1.0

```
>>> # change mu=-8.0 and signma=5.0
>>> t = [ x*5.0 - 8.0 for x in s ]
>>> st.mean(t)
-8.63357560390318
>>> st.pstdev(t)
5.127491221356098
>>> st.pvariance(t)
26.29116622508385
```

array to scalar operators

`numpy` also defines operators that reduce the number of elements. These operators are: max, mean, min, std, sum, var.

```
>>> import numpy as np
>>> s = np.random.standard_normal( 20 ) # mu = 0.0 sigma = 1.0
>>> s.shape
(20,)
>>> s # notice the pretty printing
array([ 0.34400031,  0.57069875,  0.6098522 , -2.4339711 , -1.32972374,
       -0.0883326 , -0.12963154,  1.8635653 , -0.22045277, -0.06491497,
       -1.77645665,  0.3822882 , -1.41812451,  1.41331153, -0.64529758,
       -1.12993603, -0.7633204 ,  2.54414421,  0.56501371,  0.54078789])
>>> s.max() # max of all elements
2.5441442121488285
>>> s.min() # min of all the elements
-2.4339711037589371
>>> s.sum()  # sum of all elements
-1.1664997901290586
>>> s.prod() # product of all elements
-6.9686734469627149e-05
>>> s.mean() # mean/average of all elements
-0.058324989506452929
>>> s.std() # square root of population variance
1.1971058174039078
>>> s.var() # population variance, square of population std
1.4330623380622782
>>> # for regular standard deviation and variance
>>> s.std(ddof=1)
1.2282046538116127
>>> s.var(ddof=1)
1.5084866716445033
>>>
```

Changing the mean and variance.

```
>>> import numpy as np
>>> s = np.random.standard_normal( 500 ) # mu = 0.0 sigma = 1.0
>>> t = s*5.0 + -8.0 # mu = -8.0 sigma = 5.0
>>> t.mean()
-7.7031833558472194
>>> t.std()
4.8024695885483801
>>> t.var()
23.063714148932046
>>> # bigger sample size
>>> s = np.random.standard_normal( 5000 ) # mu = 0.0 sigma = 1.0
>>> t = s*5.0 + -8.0 # mu = -8.0 sigma = 5.0
>>> t.mean()
-7.9125116720723625
>>> t.std()
5.0746140333696852
>>> t.var()
```

```
25.751707587672549
>>>
```

The operator can also select an axis (direction) for grouping the elements to be operated on.

```
>>> import numpy as np
>>> a = np.array( [[4, -9, 5, 7], [100, -100, 50, 21], [20, 40, 0, 11]] )
>>> a
array([[   4,   -9,    5,    7],
       [ 100, -100,   50,   21],
       [  20,   40,    0,   11]])
>>> a.max()
100
>>> a.min()
-100
>>> a.sum()
149
>>> a.min(0) # along the columns, varies last index
array([   4, -100,    0,    7])
>>> a.max(0) # along the columns
array([100,  40,  50,  21])
>>> a.sum(0) # along the columns
array([124, -69,  55,  39])
>>> a.min(1) # along the rows, varies first index
array([  -9, -100,    0])
>>> a.max(1) # along the rows
array([  7, 100,  40])
>>> a.sum(1) # along the rows
array([ 7, 71, 71])
>>>
```

argsort -- rank

`argsort` can be used to find the sorted position of each element in an array.

```
>>> import numpy as np
>>> s = np.random.standard_normal( 10 ) # mu = 0.0 sigma = 1.0
>>> s = s*5.0 + -8.0 # mu = -8.0 sigma = 5.0
>>> s
array([-10.87921802, -12.06678048,  -4.24797156,  -2.56391005,
        -6.40796588, -10.72305419,   5.53539   , -11.23874329,
        -6.55749501,  -7.52055858])
>>> # get sorted indexes
>>> a_s = s.argsort()
>>> a_s
array([1, 7, 0, 5, 9, 8, 4, 2, 3, 6])
>>> # sorted by indexes
>>> s[ a_s ]
array([-12.06678048, -11.23874329, -10.87921802, -10.72305419,
        -7.52055858,  -6.55749501,  -6.40796588,  -4.24797156,
        -2.56391005,   5.53539   ])
>>> # sorted by value
>>> s.sort()
>>> s
array([-12.06678048, -11.23874329, -10.87921802, -10.72305419,
        -7.52055858,  -6.55749501,  -6.40796588,  -4.24797156,
        -2.56391005,   5.53539   ])
```

Comparing numpy to standard python

std_compare.py

```
#
# The following compares standard python standard deviation
# from statistics module to the numpy module
#

import numpy as np
import statistics as st
import random

def expanded_std( arr, ddof = 0 ) :
    arr_mean = arr.mean()  # mean of arr
    arr_diff = arr - arr_mean  # difference from mean
    arr_diff_sq = arr_diff ** 2  # square of difference
    arr_diff_sq_sum = arr_diff_sq.sum() # sum of diff squared
    arr_diff_sq_sum_mean = arr_diff_sq_sum / (len(arr)-ddof) # mean
    arr_std = np.sqrt( arr_diff_sq_sum_mean ) # square root
    return arr_std

def comparison(m, a, b ) :
    print( '%-12s %12.7g - %12.7g = %12.7g' % (m, a, b, a-b) )

def group_compare( a ) :
    b = np.array(a)
    # comparison of mean between statistics and numpy
    comparison('mean', st.mean(a),  b.mean() )
    # comparison of std between statistics and numpy
    comparison('std', st.stdev(a),  b.std(ddof=1) )
```

```
    # comparison of pop std between statistics and numpy
    comparison('pop std', st.pstdev(a),  b.std(ddof=0) )
    # comparison with expandand std calculation
    comparison('expand std', expanded_std(b,1),  b.std(ddof=1) )
    comparison('expand pstd', expanded_std(b,0),  b.std(ddof=0) )

print( '(range(10,21)')
a = list(range(10,21))
group_compare( a )
print()

mu = 12.0
sigma = 2.0
s = [ random.normalvariate(mu, sigma) for _ in range(1000) ]
print( 'random mu = 12.0 sigma = 2.0' )
group_compare( s )
```

Command:

```
python3 std_compare.py
```

Standard output:

```
(range(10,21)
mean                    15 -           15 =            0
std               3.316625 -     3.316625 =            0
pop std           3.162278 -     3.162278 =            0
expand std        3.316625 -     3.316625 =            0
expand pstd       3.162278 -     3.162278 =            0

random mu = 12.0 sigma = 2.0
mean              12.02322 -     12.02322 = 1.776357e-15
std               1.981029 -     1.981029 = -2.220446e-16
pop std           1.980038 -     1.980038 = -2.220446e-16
expand std        1.981029 -     1.981029 =            0
expand pstd       1.980038 -     1.980038 =            0
```

Timing numpy to standard python

tim_comparison.py

```
import random
import timeit
import numpy as np
import statistics as st
import time

N = 100000
seq = [ random.normalvariate(0.0, 1.0) for _ in range(N) ]
seq_np = np.array( seq )

def st_std():
    return st.stdev( seq )

def np_std() :
    return seq_np.std(ddof=1)

def expanded_std( arr, ddof = 0 ) :
    arr_mean = arr.mean()  # mean of arr
    arr_diff = arr - arr_mean  # difference from mean
    arr_diff_sq = arr_diff ** 2  # square of difference
    arr_diff_sq_sum = arr_diff_sq.sum() # sum of diff squared
    arr_diff_sq_sum_mean = arr_diff_sq_sum / (len(arr)-ddof) # mean
    arr_std = np.sqrt( arr_diff_sq_sum_mean ) # square root
    return arr_std

def ex_std():
    return expanded_std( seq_np, 1.0 )

# uses time.perf_counter to time measurement
# time.perf_counter is elapsed time

# call st_std 10 times and timeit
st_time = timeit.timeit( st_std, number=10)

# call np_std 10 times and timeit
np_time = timeit.timeit( np_std, number=10)

# call np_std 10 times and timeit
ex_time = timeit.timeit( ex_std, number=10)

print( 'st time', st_time)
print( 'np time', np_time)
print( 'ex time', ex_time)
print( 'st/np', st_time/np_time )
print( 'ex/np', ex_time/np_time )
```

Command:

```
python3 tim_comparison.py
```

Standard output:

```
st time 3.0546854063868523
np time 0.002604355104267597
ex time 0.0060159265995025635
st/np 1172.914323926613
ex/np 2.3099486662339683
```

The timeit module is used to time the calculations.

calculating pi

```
>>> import numpy as np
>>> def pie( size ) :
...     y = np.random.uniform(0.0,1.0,size)
...     x = np.random.uniform(0.0,1.0,size)
...     h = np.hypot(x,y)
...     t = h <= 1.0
...     # t is a boolean array
...     return (4.0*t.sum())/size
...
>>> pie(10)
3.2000000000000002
>>> pie(100)
3.0
>>> pie(1000)
3.1880000000000002
>>> pie(10000)
3.1440000000000001
>>> pie(100000)
3.1377999999999999
>>>
```

polynomials

Polynomials are supported by numpy. Some examples are:

```
>>> import numpy as np
>>> p1 = np.poly1d([2.0,1.0]) # 2*x + 1, using coefficients
>>> print(p1)

2 x + 1
>>> # using the polynomials roots
>>> p1a = np.poly1d([-0.5],r=1) # p1(-0.5) = 0
>>> print(p1a)

1 x + 0.5
>>> # quadratic with roots (x-5)(x-3)
>>> p2 = np.poly1d( [5.0, 3.0], r=1 )
>>> print(p2)
   2
1 x - 8 x + 15
>>> # polynomials can be multiplied
>>> p3 = np.poly1d([1.0,-5]) * np.poly1d([1.0,3.0])
>>> print(p3)
   2
1 x - 2 x - 15
>>> # or added
>>> np.poly1d([1.0,-5]) + np.poly1d([1.0,3.0])
poly1d([ 2., -2.])
>>> # or subtracted
>>> np.poly1d([3.0,1.0,-5]) - np.poly1d([1.0,3.0])
poly1d([ 3.,  0., -8.])
>>> # or division with remainder
>>> quot, rem =  np.poly1d([3.0,1.0,-5]) /  np.poly1d([1.0,3.0])
>>> print(quot)

3 x - 8
>>> print(rem)

19
>>> # polynomials can be evaluated
>>> p3(-3)
0.0
>>> p3(5)
0.0
>>> p3(1)
-16.0
>>> # the polynomials roots are given by
>>> p3.r
array([ 5., -3.])
>>> # its coefficients
>>> p3.c
array([  1.,  -2., -15.])
>>> # its order
>>> p3.o
```

```
2
>>> # the first derivative
>>> print(p3.deriv())

2 x - 2
>>> print(p3.deriv(1))

2 x - 2
>>> # the second derivative
>>> print(p3.deriv(2))

2
>>>
```

Generate Random Rectangular Arrays

Random rectangular arrays can be generated with:

gen_rand_mat.py

```
import sys
import numpy as np

if __name__ == '__main__' :
    n = int( sys.argv[1] )   # data files
    c = int( sys.argv[2] )   # columns
    r = int( sys.argv[3] )   # rows

    for i in range( n ) :
        fname = 'mat%d.txt' % i
        m = np.random.randint( 0,100, size=(r,c), dtype='i' )
        # save the array as a text file
        np.savetxt( fname, m, fmt="%5d" )
```

Pairwise Minimum

The minimum of aligned elements is calculated with:

np_min.py

```
import numpy as np
import sys

def read_mat( fname ) :
    with open(fname) as f :
        mat = [ [int(e) for e in l.split()] for l in f ]
    return mat

if __name__ == '__main__' :

    mats = []
    for fname in sys.argv[1:] :
        mats.append( np.array(read_mat( fname ), np.int32) )

    mn = mats[0].copy()
    for m in mats[1:] :
        mn = np.minimum( mn, m )

    # output minimum
    np.savetxt( sys.stdout.buffer, mn, fmt='%5d' )
```

Command:

```
python3 np_min.py mat0.txt mat1.txt mat2.txt
```

Standard output:

```
   19     8     0    35    12    18    26     7
   20    32     3    69    21     7    51     5
   24     9    31    18    12    10    10     6
   46    77    34    20    11     0    66    11
    3     0     5     2    10    41    49     9
   44    52    25    54     8    42    19     6
    2     6     2     9    30    30    24    37
    8     7    18    60    34    19    18    12
   21    31    63    25    26     6     2    60
```

Pairwise Minimum (loadtxt)

Arrays can be loaded with `np.loadtxt`.

np_min1.py

```
import numpy as np
import sys

if __name__ == '__main__' :
```

```
    mats = []
    for fname in sys.argv[1:] :
        mats.append( np.loadtxt(fname, np.int32) )

    mn = mats[0].copy()
    for m in mats[1:] :
        mn = np.minimum( mn, m )

    # output minimum
    np.savetxt( sys.stdout.buffer, mn, fmt='%5d' )
```

Command:

```
python3 np_min1.py mat0.txt mat1.txt mat2.txt
```

Standard output:

```
   19     8     0    35    12    18    26     7
   20    32     3    69    21     7    51     5
   24     9    31    18    12    10    10     6
   46    77    34    20    11     0    66    11
    3     0     5     2    10    41    49     9
   44    52    25    54     8    42    19     6
    2     6     2     9    30    30    24    37
    8     7    18    60    34    19    18    12
   21    31    63    25    26     6     2    60
```

Searching

List the indices for the searched item

np_search.py

```
import numpy as np
import sys

if __name__ == '__main__' :

    search = int(sys.argv[1])
    for fname in sys.argv[2:] :
        mat = np.loadtxt(fname, np.int32)
        ind = np.where( mat == search)
        ind = zip(ind[0],ind[1])
        for ix in ind :
            print( fname, ix[0], ix[1] )
```

Command:

```
python3 np_search.py 6 mat0.txt mat1.txt mat2.txt
```

Standard output:

```
mat0.txt 2 7
mat0.txt 5 7
mat1.txt 6 1
mat2.txt 1 2
mat2.txt 8 5
```

Course record file

The students records for a course is stored in the following csv file.

course.csv

```
Name, Assignments, Midterm1, Midterm2, Final
N Student, 80, 65, 72, 74
M Student, 75, 52, 62, 64
O Student, 100, 70, 80, 80
P Student, 94, 45, 65, 55
```

The weights for the term work, midterm 1, midterm 2, and the final is 10, 20, 20, and 50.

Course report

A report giving statistics on each work product, and the student's course grade is generated by:

course_report.py

```
#!/usr/bin/env python
import numpy as np
import csv

# get head and names
```

```
with open( 'course.csv' ) as f:
    reader = csv.reader( f )
    headings = next( reader )
    # get student names
    students = []
    for row in reader :
        students.append( row[0] )

# get workproducts, skipping name column
with open(  'course.csv' ) as f:
    wps = np.loadtxt( f, delimiter=",",
                      usecols=(1,2,3,4), skiprows=1 )

# value of each workproduct item
value = np.array( [10., 20., 20., 50.] )
# report final mark
print('Course mark')
marks = [] # save the marks
for i, name in enumerate( students ) :
    mark = np.dot( value, wps[i]/100.0 )
    marks.append( mark )
    print("%-20s: %5.1f" % (name, mark))

marks = np.array( marks ) # convert to array

print("\n"*2)
print('Statistics')
# report on statistics of each workproduct
wp_names = headings[1:]
for i, w in enumerate( wp_names ) :
    print(w.strip())
    print('    minimum %6.2f' %  wps[:,i].min() )
    print('    maximum %6.2f' %  wps[:,i].max() )
    print('       mean %6.2f' %  wps[:,i].mean())
    print('        std %6.2f' %  wps[:,i].std() )

# generated histogram of As, Bs, Cs, Ds, Fs
bins = np.array([0.0, 49.5, 54.5, 64.5, 79.5, 100.0] ) # F,D,C,B,A
N,bins = np.histogram(marks,bins)

N = list(N) # convert to list
N.reverse()
for letter,n in zip( "ABCDF", N) :
    print(letter, n)
```

The produced report is:

Command:

```
python3 course_report.py
```

Standard output:

```
Course mark
N Student            :  72.4
M Student            :  62.3
O Student            :  80.0
P Student            :  58.9



Statistics
Assignments
    minimum  75.00
    maximum 100.00
       mean  87.25
        std  10.13
Midterm1
    minimum  45.00
    maximum  70.00
       mean  58.00
        std   9.97
Midterm2
    minimum  62.00
    maximum  80.00
       mean  69.75
        std   6.94
Final
    minimum  55.00
    maximum  80.00
       mean  68.25
        std   9.55
A 1
B 1
C 2
D 0
F 0
```

Conway's game of life

The games consists of an infinite board composed of a grid of squares. Each square can contain a zero or a one. Each square has eight neighbours, the next square connected either horizontally, vertically, or on a diagonal. Give a board configuration, the next configuration is determined by:

- A square with a 1 and fewer than two neighbours with a 1 is set to 0.
- A square with a 1 and more than three neighbours with a 1 is set to 0.
- A square with a 1 and two or three neighbours with a 1, remains set to 1.
- A square with a 0, and exactly three neighbours with a 1 is set to a 1.

Conway's game of life in numpy

Conway's game can be easily implemented with `numpy`'s array operations. A bounded implementation (i.e., the board is not infinite) where the borders are always zero is realized by:

simple_life.py

```
#!/usr/bin/env python
import numpy as np

# compute next generation of conway's game of life
def next_generation( current, next ) :
    next[:,:] = 0 # zero out next board
    # bound examination area
    bend0 = (current.shape[0]-3)+1
    bend1 = (current.shape[1]-3)+1
    for i in range( bend0 ) :
        for j in range( bend1 ) :
            neighbours = np.sum( current[i:i+3, j:j+3] )
            if current[i+1, j+1] == 1 :
                neighbours -= 1 # do not count yourself
                if 2 <= neighbours <= 3 :
                    next[i+1, j+1] = 1
            else:
                if neighbours == 3 :
                    next[i+1,j+1] = 1

board = np.zeros( (5,5), np.uint8)
next_board = np.zeros_like( board ) # same shape

# create inital configuration
board[2,1:4] = 1 # 3 squares

for i in range(4) :
    next_generation( board, next_board )
    board, next_board = next_board, board # swap boards
    print(board[1:-1,1:-1] )
```

Four generation of this configuration yeilds:

Command:

```
python3 simple_life.py
```

Standard output:

```
[[0 1 0]
 [0 1 0]
 [0 1 0]]
[[0 0 0]
 [1 1 1]
 [0 0 0]]
[[0 1 0]
 [0 1 0]
 [0 1 0]]
[[0 0 0]
 [1 1 1]
 [0 0 0]]
```

Game of life with user input

An input string of ones and zeros can be used to initialize the board. This version is identical to the previous version with the exception of the command line arguments.

life.py

```
#!/usr/bin/env python3
from numpy import *
import sys
import math

# compute next generation of conway's game of life
def next_generation( current, next ) :
    next[:,:] = 0 # zero out next board
    # bound examination area
    bend0 = (current.shape[0]-3)+1
    bend1 = (current.shape[1]-3)+1
    for i in range( bend0 ) :
```

```
        for j in range( bend1 ) :
            neighbours = sum( current[i:i+3, j:j+3] )
            if current[i+1, j+1] == 1 :
                neighbours -= 1 # do not count yourself
                if 2 <= neighbours <= 3 :
                    next[i+1, j+1] = 1
            else:
                if neighbours == 3 :
                    next[i+1,j+1] = 1

if len(sys.argv) != 3 :
    print("usage:", sys.argv[0], "init-board generations")
    sys.exit( 1 )

init = sys.argv[1]
generations = int( sys.argv[2] )

board_sz = math.ceil( math.sqrt(len(init)) )

# board_sz+2 includes the border of zeros
board = zeros( (board_sz+2,board_sz+2), uint8)
next_board = zeros_like( board ) # same shape

# fill the board
i = 0
j = 0
for index,ch in enumerate(init) :
    if ch == '1' :
        board[i+1,j+1] = 1
    j = (j+1) % board_sz
    if ((index+1) % board_sz) == 0 : i += 1

for gen in range(generations) :
    next_generation( board, next_board )
    board, next_board = next_board, board # swap boards
    print(board[1:-1,1:-1]) # do not print the border
```

Solving 2 unknowns with 2 linear equations

```
>>> import numpy as np
>>> from numpy.linalg import solve
>>>
>>> # 1 * x0 - 2 * x1 = 1
>>> # 2 * x0 - 6 * x1 = -2
>>> a = np.array( [[1,-2], [2,-6]] )
>>> b = np.array( [1, -2 ] )
>>> x = solve( a, b ); x
array([ 5.,  2.])
>>> np.dot( a, x ) # check solution
array([ 1., -2.])
>>>
```

Solving 2 unknowns with 2 linear equations (2)

```
>>> import numpy as np
>>> from numpy.linalg import solve
>>>
>>> # 1 * x0 - 2 * x1 = 1
>>> # 2 * x0 - 4 * x1 = 2
>>> # any problems with the above equations
>>> a = np.array( [[1,-2], [2,-4]] )
>>> b = np.array( [1, 2 ] )
>>> x = solve( a, b )
Traceback (most recent call last):
  File "<console>", line 1, in <module>
  File "/users/cs/faculty/rod/miniconda3/lib/python3.6/site-packages/numpy/linalg/linalg.py", line 375, in solve
    r = gufunc(a, b, signature=signature, extobj=extobj)
  File "/users/cs/faculty/rod/miniconda3/lib/python3.6/site-packages/numpy/linalg/linalg.py", line 90, in _raise_linalgerror_singular
    raise LinAlgError("Singular matrix")
numpy.linalg.linalg.LinAlgError: Singular matrix
>>>
```

Solving 3 unknowns with 3 linear equations

```
>>> import numpy as np
>>> from numpy.linalg import solve
>>> from numpy.linalg import inv
>>>
>>> # 1 * x0 - 2 * x1 + 5 * x2 = 1
>>> # 2 * x0 - 6 * x1 - 10 *x2 = -2
>>> #-4 * x0 + 1 * x1 + 3 * x2 = 3
>>> a = np.array( [[1, -2, 5], [2,-6,-10], [-4,1,3] ] )
>>> b = np.array( [1, -2, 3 ] )
>>> x = solve( a, b ); x
array([-0.64516129, -0.25806452,  0.22580645])
>>> np.dot( a, x ) # check solution
array([ 1., -2.,  3.])
>>>
>>> # a * x = b
>>> # inv(a) * a * x = inv(a) * b
>>> # I * x = inv(a) * b
```

```
>>> # x = inv(a) * b
>>> x = np.dot( inv(a), b ); x
array([-0.64516129, -0.25806452,  0.22580645])
>>> # the identity
>>> np.dot( inv(a), a)
array([[  1.00000000e+00,   5.55111512e-17,   0.00000000e+00],
       [ -5.55111512e-17,   1.00000000e+00,   1.66533454e-16],
       [  6.93889390e-18,  -1.90819582e-17,   1.00000000e+00]])
>>> np.dot( a, inv(a))
array([[  1.00000000e+00,   5.55111512e-17,   1.38777878e-17],
       [  0.00000000e+00,   1.00000000e+00,   2.77555756e-17],
       [  0.00000000e+00,   1.38777878e-17,   1.00000000e+00]])
```

2D line segment intersection

Many geometric problems are solved using vector notation. Intesection of lines can be solved with:

lines.py

```
#
# line segment intersection using vectors
# see Computer Graphics by F.S. Hill
#
import numpy as np
def perp( a ) :
    b = np.empty_like(a)
    b[0] = -a[1]
    b[1] = a[0]
    return b

# line segment a given by endpoints a1, a2
# line segment b given by endpoints b1, b2
# return
def seg_intersect(a1,a2, b1,b2) :
    da = a2-a1
    db = b2-b1
    dp = a1-b1
    dap = perp(da)
    denom = np.dot( dap, db)
    num = np.dot( dap, dp )
    return (num / denom)*db + b1

p1 = np.array( [0.0, 0.0] )
p2 = np.array( [1.0, 0.0] )

p3 = np.array( [4.0, -5.0] )
p4 = np.array( [4.0, 2.0] )

print(seg_intersect( p1,p2, p3,p4))

p1 = np.array( [2.0, 2.0] )
p2 = np.array( [4.0, 3.0] )

p3 = np.array( [6.0, 0.0] )
p4 = np.array( [6.0, 7.0] )

print(seg_intersect( p1,p2, p3,p4))
```

Its sample execution is:

Command:

```
python3 lines.py
```

Standard output:

```
[ 4.  0.]
[ 6.  4.]
```

Points on circle "without" sin/cos

All the points around a circle can be generated with matrix operations. Multiplication is usually cheaper than calculationg sin or cos.

circle.py

```
import numpy as np
import math

# sin/cos just once
a = math.cos( math.radians(-30.0) )
b = math.sin( math.radians(-30.0) )

# rotation matrix, 30 degrees clockwise
R = np.mat( [[a, -b], [b, a]] )
v = np.mat( [1,0] )   # unit vector along x-axis

print("Repeated rotations by 30")
# rotate (1,0) by 30 degree increments
```

```
for i in range(12) :
    nv = v * R
    vx,vy = v.T  # transpose
    x = math.cos( math.radians(30.0 * i ) )
    y = math.sin( math.radians(30.0 * i ) )
    print("%12.9f %12.9f -- %12.9f %12.9f" % (vx,vy, x,y))
    v = nv
print()

print("Rotation by multiples of 30")
# rotate (1,0) by 30, 60, 90, ...
for i in range(12) :
    r = v * pow(R,i)
    vx,vy = r.T  # transpose
    x = math.cos( math.radians(30.0 * i ) )
    y = math.sin( math.radians(30.0 * i ) )
    print("%12.9f %12.9f -- %12.9f %12.9f" % (vx,vy, x,y))
```

The points are:

Command:

```
python3 circle.py
```

Standard output:

```
Repeated rotations by 30
 1.000000000  0.000000000 --  1.000000000  0.000000000
 0.866025404  0.500000000 --  0.866025404  0.500000000
 0.500000000  0.866025404 --  0.500000000  0.866025404
 0.000000000  1.000000000 --  0.000000000  1.000000000
-0.500000000  0.866025404 -- -0.500000000  0.866025404
-0.866025404  0.500000000 -- -0.866025404  0.500000000
-1.000000000  0.000000000 -- -1.000000000  0.000000000
-0.866025404 -0.500000000 -- -0.866025404 -0.500000000
-0.500000000 -0.866025404 -- -0.500000000 -0.866025404
-0.000000000 -1.000000000 -- -0.000000000 -1.000000000
 0.500000000 -0.866025404 --  0.500000000 -0.866025404
 0.866025404 -0.500000000 --  0.866025404 -0.500000000

Rotation by multiples of 30
 1.000000000 -0.000000000 --  1.000000000  0.000000000
 0.866025404  0.500000000 --  0.866025404  0.500000000
 0.500000000  0.866025404 --  0.500000000  0.866025404
 0.000000000  1.000000000 --  0.000000000  1.000000000
-0.500000000  0.866025404 -- -0.500000000  0.866025404
-0.866025404  0.500000000 -- -0.866025404  0.500000000
-1.000000000  0.000000000 -- -1.000000000  0.000000000
-0.866025404 -0.500000000 -- -0.866025404 -0.500000000
-0.500000000 -0.866025404 -- -0.500000000 -0.866025404
-0.000000000 -1.000000000 -- -0.000000000 -1.000000000
 0.500000000 -0.866025404 --  0.500000000 -0.866025404
 0.866025404 -0.500000000 --  0.866025404 -0.500000000
```