A nicer numpy

Dima Kogan

March 7, 2020

What is this about?

Two libraries to make working in numpy nicer.

- These are public tools, available for some years
- Installable from Debian and related distros.
- Python2 and Python3 both supported

numpysane (`https://github.com/dkogan/numpysane`)

- Provides some routines to improve core functionality
- These are new functions, so there're no compatibility concerns

gnuplotlib (`https://github.com/dkogan/gnuplotlib`)

- Plotting
- Does a similar thing as `matplotlib` but (I claim) better

What's wrong with numpy?

- Some core functionality is mysterious and unintuitive
- Things work as expected *only* with 2-dimensional arrays, no more and no less

Areas addressed by `numpysane`:

- Nicer array manipulation
- Nicer basic linear algebra routines
- Better broadcasting support

Mostly stolen from the PDL project

Matrix concatenation

Basic example: stick two identical 2D arrays together to extend each row

- The docs say to use `hstack()`

Let's try it:

```
>>> import numpy as np
>>> arr32 = np.arange(3*2).reshape(3,2)
>>> print(arr32)
[[0 1]
 [2 3]
 [4 5]]

>>> print(arr32.shape)
(3, 2)
```

Matrix concatenation

What do we expect `hstack(arr32,arr32)` to do?

```
[[0 1 0 1]
 [2 3 2 3]
 [4 5 4 5]]
```

or

```
[[0 1]
 [2 3]
 [4 5]
 [0 1]
 [2 3]
 [4 5]]
```

?

Matrix concatenation

This was a trick question. Here's what it does:

```
>>> np.hstack(arr32,arr32)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: hstack() takes 1 positional argument ...
    ... but 2 were given
```

Apparently `hstack()` wants an iterable of the arguments, instead of the arguments themselves

Matrix concatenation

Fine. Here's what it does if you feed it what it wants:

```
>>> print(np.hstack((arr32,arr32)))
[[0 1 0 1]
 [2 3 2 3]
 [4 5 4 5]]
```

That makes sense! Looks "horizontal".

Matrix concatenation

What if I don't feed it strictly 2D matrices?

```
>>> arr132 = np.arange(3*2).reshape(1,3,2)
>>> print(arr132)
[[[0 1]
  [2 3]
  [4 5]]]

>>> print(arr132.shape)
(1, 3, 2)
```

Matrix concatenation

Same question as before: what do we expect `hstack((arr132,arr132))` to do?

```
[[[0 1 0 1]
  [2 3 2 3]
  [4 5 4 5]]]
```

or

```
[[[0 1]
  [2 3]
  [4 5]
  [0 1]
  [2 3]
  [4 5]]]
```

or something else?

Matrix concatenation

Here's what it does:

```
>>> print(np.hstack((arr132,arr132)))
[[[0 1]
  [2 3]
  [4 5]
  [0 1]
  [2 3]
  [4 5]]]

>>> np.hstack((arr132,arr132)).shape
(1, 6, 2)
```

Whoa. That is *not* horizontal at all! I would have expected a result with shape `(1,3,4)`

Matrix concatenation

What if I give it 1-dimensional arrays?

```
>>> arr3  = np.arange(3)
>>> arr13 = np.arange(3).reshape(1,3)
>>> print(arr3)
[0 1 2]

>>> arr3.shape
(3,)

>>> print(arr13)
[[0 1 2]]

>>> arr13.shape
(1, 3)
```

Matrix concatenation

```
>>> np.hstack((arr3,arr3)).shape
(6,)

>>> np.hstack((arr13,arr13)).shape
(1, 6)

>>> np.hstack((arr13,arr3)).shape
ValueError: all the input arrays must have ...
    ... same number of dimensions
```

- Do the stacking functions want the dimension counts to match up, or something?

Well, no:

```
>>> np.vstack((arr13,arr3))
[[0 1 2]
 [0 1 2]]
```

Matrix concatenation

So what's wrong?

- numpy is inconsistent about which is the most significant dimension in an array

There's an arbitrary design choice that must be made: if I stack `N` arrays of shape `(A,B,C)` into a new array, do I get

1. an array of shape `(N,A,B,C)` or
2. an array of shape `(A,B,C,N)`?

Most of numpy makes the *first* choice, but some of it (concatenation functions most notably) makes the second choice

Dimensionality example

Example:

- Let's say I have a 1-dimensional array containing simultaneous temperature measurements at different locations:

```
>>> print(T1)
[ t_where0 t_where1 t_where2 ... ]

>>> print(T1.shape)
(Nlocations,)
```

We have one dimension, so the locations are indexed by `axis = 0` and `axis = -1`. These are the same axis.

Dimensionality example

Now, let's say I measured all the temperatures multiple times throughout the day, and I record the measurements into a joint array `T2`.
I have a choice:

```
>>> print(T2.shape)
(Ntimes,Nlocations)
```

or

```
>>> print(T2.shape)
(Nlocations,Ntimes)
```

?

Dimensionality example

When I extend `T1` into `T2` I want consistent printing:
The dimensions printed horizontally and vertically should not change
I.e. I want this:

```
>>> print(T2)
[[ t_when0where0 t_when0where1 t_when0where2 ... ]
 [ t_when1where0 t_when1where1 t_when1where2 ... ]
 ...]

>>> print(T2.shape)
(Ntimes, Nlocations)
```

This way each horizontal row describes *one* point in time and *multiple* locations, just like when printing `T1`

Dimensionality example

When I extend `T1` into `T2` I want consistent indexing:
The axis index corresponding to locations should not change

- For `T1`, locations are in `axis = 0` and `axis = -1` (same axis)
- For `T2`, locations are in `axis = 1` and `axis = -1` (same axis)

So counting *from the back* gives me consistency, and I want to always use `axis = -1`
Thus I want

- The *first* concatenation option: stacking `N` arrays of shape `(A,B,C)` produces an array of shape `(N,A,B,C)`
- All axes to be indexed from the end. Always.

Dimensionality example

If we really wanted to index the axes from the front while remaining self-consistent, numpy could do what PDL does:

- the horizontally-printed dimension is the *first* dimension
- `N` arrays of shape `(A,B,C)` produce an array of shape `(A,B,C,N)`

But then a core convention of linear algebra would be violated: a matrix of `N` rows and `M` columns would have shape `(M,N)`. Can't please everybody.

Matrix concatenation: conclusion

So why are `hstack()` and friends weird?

- Because `hstack()` tries to concatenate along `axis = 1`, while it should use `axis = -1`
- This works for 2D arrays (and 1D arrays because of special-case logic in `hstack()`), but not for others

Many other core functions in numpy have this issue, and routines in `numpysane` do this in a consistent and predictable way.

Matrix concatenation with numpysane

There are two functions, both stolen from the PDL project.

- `glue()` concatenates any N arrays along the given axis
- `cat()` concatenates N arrays along a new outer dimension

These both add leading length-1 dimensions to the input as needed: "something" is logically equivalent to "1 of something". This is one of the *broadcasting* rules I'll get to in a bit

Matrix concatenation with numpysane

`nps.glue()` works as expected:

```
>>> import numpysane as nps

>>> nps.glue(arr32, arr32,  axis=-1).shape
(3, 4)

>>> nps.glue(arr32, arr32,  axis=-2).shape
(6, 2)

>>> nps.glue(arr132,arr132, axis=-1).shape
(1, 3, 4)

>>> nps.glue(arr13, arr3,   axis=-1).shape
(1, 6)

>>> nps.glue(arr13, arr3,   axis=-2).shape
(2, 3)
```

Matrix concatenation with numpysane

`nps.cat()` works as expected too. It always adds a new leading dimension

```
>>> nps.cat(arr32,arr32).shape
(2, 3, 2)

>>> nps.cat(arr132,arr32).shape
(2, 1, 3, 2)
```

Matrix multiplication

The funny business extends to other core areas of numpy. For instance multiplying matrices is non-trivial

- Up until numpy 1.10.0 (2015-2016) `np.dot()` was the function for that, and it is surprising in all sorts of ways (which should be expected since a "dot product" is not the same thing as "matrix multiplication")
- In 1.10.0 we got `np.matmul`, which is *much* better, but even then it has strange corners. Trying to compute an outer product:

```
>>> a = np.arange(5).reshape(5,1)
>>> b = np.arange(3)

>>> np.matmul(a,b)
ValueError: matmul: Input operand 1 has a mismatch in
   its core dimension 0, with gufunc signature
   (n?,k),(k,m?)->(n?,m?) (size 3 is different from 1)
```

Matrix multiplication with numpysane

numpysane provides its own `matmult()` routine that does what one expects:

```
>>> nps.matmult(a,b).shape

(5, 3)
```

There're many more functions in numpysane in this area. Everything's documented, and I'd like to move on to...

Broadcasting

What is broadcasting?

- *Broadcasting* is a generic way to vectorize functions
- A broadcasting-aware function has a *prototype*: it knows the dimensionality of its inputs and of its outputs
- When calling a broadcasting-aware function, any extra dimensions in the input are automatically used for vectorization

Broadcasting: an example

This is best described with an example: a broadcasting-aware inner product. An inner product (also known as a dot product) is a function that

- takes in two identically-sized 1-dimensional arrays
- outputs a scalar

`inner( [ 1 2 3 4], [1 2 3 4] )` $\rightarrow$ 30

Broadcasting: an example

If one calls a broadcasting-aware inner product (such as `nps.inner()`) with two arrays of shape (2,3,4) as input, it would

- compute 6 inner products of length-4 each
- report the output in an array of shape (2,3)

Broadcasting: an example

Let

```
>>> a234 = np.arange(2*3*4).reshape(2,3,4)

>>> print(a234)
[[[ 0  1  2  3]
  [ 4  5  6  7]
  [ 8  9 10 11]]

 [[12 13 14 15]
  [16 17 18 19]
  [20 21 22 23]]]

>>> a4 = np.arange(4)

>>> print(a4)
[0 1 2 3]
```

Broadcasting: an example

So we can give it two (2,3,4) arrays, and get inner products of each corresponding row:

```
>>> print(nps.inner(a234,a234))
[[  14  126  366]
 [ 734 1230 1854]]
```

The values in the output are

```
[[ inner([0,1,2,3],     [0,1,2,3]),         ..... ]
 [ inner([12,13,14,15], [12,13,14,15]),     ..... ]]
```

and so on

Broadcasting: an example

Or we can given it one (2,3,4) array and a (4,) array to compute the inner product of every row in the larger array with the one (4,) array:

```
>>> print(nps.inner(a234,a4))
[[ 14  38  62]
 [ 86 110 134]]
```

The values in the output are

```
[[ inner([0,1,2,3],     [0,1,2,3]),     ..... ]
 [ inner([12,13,14,15], [0,1,2,3]),     ..... ]]
```

and so on

Broadcasting rules

1. Line up the shapes of the inputs to their *trailing* dimensions
2. Match the trailing dimensions with the expected shapes of the inputs. If anything doesn't match, throw an exception
3. The extra leading dimensions must be compatible across all the inputs. This means that each leading dimension must either
   - equal 1
   - be missing (thus assumed to equal 1)
   - equal to some positive integer >1, consistent across all arguments
4. Any extra leading dimensions are used for vectorization, and determine the shape of the output

OK, so what about broadcasting?

In stock numpy, broadcasting is documented, but

- it is sparse and incomplete
- little end-user awareness that it exists

`numpysane` provides routines to add broadcasting awareness

- to any python function (via a decorator)
- to any C function (via generated C code that produces an extension module)

Broadcasting: an example

Let's add broadcasting-awareness to an existing inner product function

```
import numpysane as nps
@nps.broadcast_define( (('n',), ('n',)), () )
def inner(a,b):
    ....
```

- We had a function `inner(a,b)` that computes *one* inner product. It knows nothing about vectorization
- Then we applied the `nps.broadcast_define()` decorator, and we get dimensionality checking and vectorization logic

Plotting: gnuplotlib

Let's switch gears, and talk about plotting.

- As with the numpy core, there's a dominant choice here: matplotlib
- I'm not aware of any *major* issues: if it's not pissing you off right now, there probably isn't a lot of reason to switch to my library

However, matplotlib ...

- is python-specific
- is slow
- has a weird API
- is missing useful interactivity

Plotting: gnuplotlib

`gnuplotlib`: a plotting library for numpy

- Uses `gnuplot` as the plotting backend, so
  - The plots look and feel like `gnuplot` plots have for decades
  - It's fast
  - Lots of features and backends available
- Has a reasonable API (I claim)
- A direct port of `PDL::Graphics::Gnuplot`

Plotting: gnuplotlib design choices

One `plot()` function does everything

- Can still build up the plot components programmatically: using python

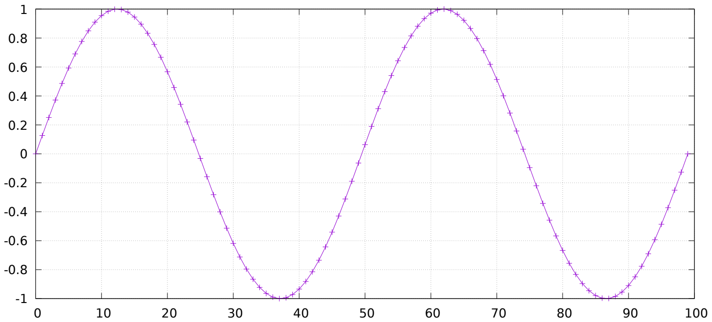
`gnuplotlib` is a thin shim

- strings are passed to `gnuplot` verbatim (like in `feedgnuplot`)
- so we get a powerful library and a friendly learning curve

Plotting: gnuplotlib: a *very* brief tutorial

To plot something, just call plot:

```
import numpy      as np
import numpysane  as nps
import gnuplotlib as gp
th = np.linspace(-2.*np.pi, 2.*np.pi, 100)
gp.plot(np.sin(th))
```



Plotting: gnuplotlib: a *very* brief tutorial

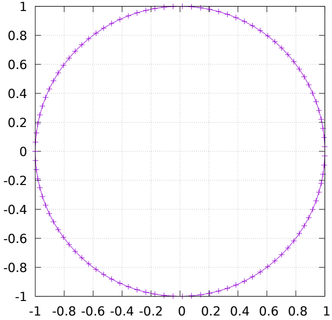
```
import numpy      as np
import numpysane  as nps
import gnuplotlib as gp
th = np.linspace(-2.*np.pi, 2.*np.pi, 100)
gp.plot(np.sin(th))
```

- We're plotting in 2D, so default is `tuplesize=2` arrays
- We gave it just 1 array, so integers 0,1,2,... were used for the x

Plotting: gnuplotlib: a *very* brief tutorial

- We can pass in 2 arrays to make an x-y plot:

```
th = np.linspace(-np.pi, np.pi, 100)
gp.plot(np.cos(th), np.sin(th), square = True)
```



Plotting: gnuplotlib: a *very* brief tutorial

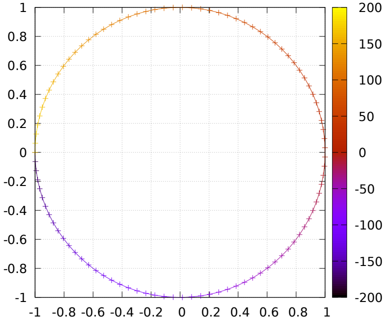
```
th = np.linspace(-np.pi, np.pi, 100)
gp.plot(np.cos(th), np.sin(th), square = True)
```

- We passed in *two* arrays
- We also passed in `square = True`. This is a *plot option* to autoscale the x and y axes evenly. Otherwise the circle will looks like an ellipse

Plotting: gnuplotlib: a *very* brief tutorial

- It's possible to have more values per point. For instance:

```
th = np.linspace(-np.pi, np.pi, 100)
gp.plot(np.cos(th), np.sin(th),
        # The angle (in degrees) is shown as the color
        th * 180./np.pi,
        tuplesize = 3,
        _with     = 'linespoints palette',
        square    = True)
```



Plotting: gnuplotlib: a *very* brief tutorial

```
th = np.linspace(-np.pi, np.pi, 100)
gp.plot(np.cos(th), np.sin(th),
        # The angle (in degrees) is shown as the color
        th * 180./np.pi,
        tuplesize = 3,
        _with     = 'linespoints palette',
        square    = True)
```

- The style `linespoints palette` is given to `gnuplot` directly. `gnuplotlib` doesn't know what that means
- `tuplesize=3` tells `gnuplotlib` that there are 3 values per point. Because of `palette`, these will be interpreted as x,y,color
- The `gnuplot` documentation talks in detail about what kind of input each style expects

Plotting: gnuplotlib: a *very* brief tutorial

An explicit invocation of `plot()` looks like this:

```
plot( curve, curve, ..., plot_options )
```

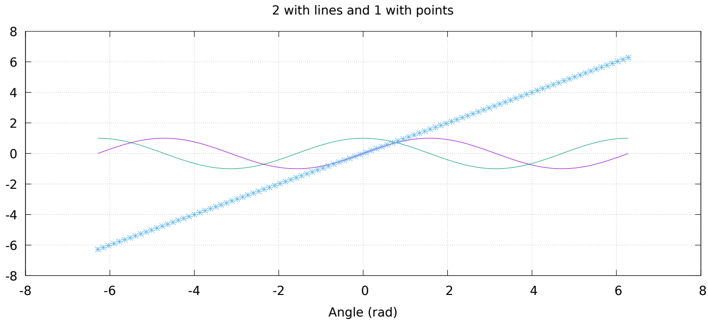
where each `curve` is a `tuple`:

```
curve = (array, array, ..., curve_options)
```

- *plot options* apply to the whole plot, and are given as keyword args to `plot()`
- *curve options* apply to each separate curve (dataset); given in a `dict()` in the end of each `curve` tuple. Or defaults given in the `plot()` kwargs
- If we have one dataset, we can inline the tuples, like we did above

Plotting: gnuplotlib: a *very* brief tutorial

```
th = np.linspace(-2.*np.pi, 2.*np.pi, 100)
gp.plot( ( th, np.sin(th), ),
         ( th, np.cos(th), ),
         ( th, th, dict(_with = 'points ps 1') ),
         _with = 'lines',
         xlabel = "Angle (rad)",
         title  = "2 with lines and 1 with points")
```



Plotting: gnuplotlib: a *very* brief tutorial

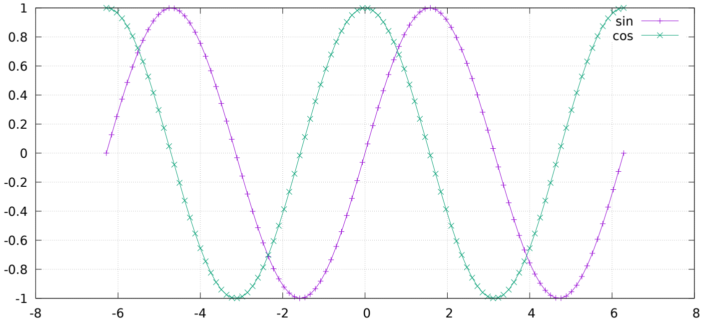
```
th = np.linspace(-2.*np.pi, 2.*np.pi, 100)
gp.plot( ( th, np.sin(th), ),
         ( th, np.cos(th), ),
         ( th, th, dict(_with = 'points ps 1') ),
         _with = 'lines',
         xlabel = "Angle (rad)",
         title  = "2 with lines and 1 with points")
```

- We passed in 3 tuples, one for each dataset
- We passed in the `xlabel` plot option to label the x axis
- We passed in the `title` plot option to title the plot
- We passed in the default `with` curve option: `lines`
- 2/3 datasets don't set their own `with`, so they use `lines`
- 1/3 plots with `points ps 1` instead. `gnuplotlib` doesn't know what that is, but gnuplot knows that ps is a synonym for `pointsize`

Plotting: gnuplotlib: a *very* brief tutorial

- Broadcasting is fully supported:

```
th = np.linspace(-2.*np.pi, 2.*np.pi, 100)
gp.plot( th,
         nps.cat(np.sin(th),
                 np.cos(th)),
         legend = np.array( ("sin", "cos"), ) )
```



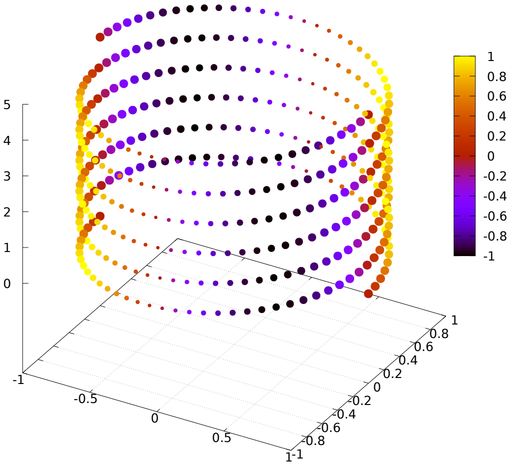
Plotting: gnuplotlib: a *very* brief tutorial

```
th = np.linspace(-2.*np.pi, 2.*np.pi, 100)
gp.plot( th,
         nps.cat(np.sin(th),
                 np.cos(th)),
         legend = np.array( ("sin", "cos"), ) )
```

- I plotted two datasets, but didn't use tuples
- Using default `tuplesize=2`, and gave it two arrays:
  - First array has the expected shape of `(100,)`
  - Second array has the shape `(2,100)`
- This thus broadcasts: I get two plots: `sin(th)` vs `th` and `cos(th)` vs `th`
- curve options broadcast too: I have it two different `legend` options, and `gnuplotlib` knows to use each one for the two datasets

Plotting: gnuplotlib: a *very* brief tutorial

Let's make this plot:



Plotting: gnuplotlib: a *very* brief tutorial

```
th    = np.linspace(0, 6*np.pi, 200)
z     = np.linspace(0, 5,       200)
size  = 0.5 + np.abs(np.cos(th))
color = np.sin(2*th)

gp.plot3d( np.cos(th) * nps.transpose(np.array((1,-1))),
           np.sin(th) * nps.transpose(np.array((1,-1))),
           z,
           size,
           color,
           tuplesize = 5,
           _with = 'points ps variable pt 7 palette',
           squarexy = True)
```

Plotting: gnuplotlib

That's it for the overview. Lots of examples in the guide:

- `https://github.com/dkogan/gnuplotlib/blob/master/guide/guide.org`

The API docs are on the main page:

- `https://github.com/dkogan/gnuplotlib`

Thanks for listening!

The documentation and sources and links to this talk:

- `https://github.com/dkogan/numpysane`
- `https://github.com/dkogan/gnuplotlib`

Or you can

```
apt install python3-numpysane python3-gnuplotlib
```