extending numpy with C (coding your own gufunc)

October 2013

Jaime Fernández (jaime.frio@gmail.com)

before we start

These imports precede any Python code:

```
from __future__ import division, print_function
import numpy as np
```

All code samples have been tested with:

```
>>> import sys
>>> sys.version
'2.7.5 (default, May 15 2013, 22:44:16)
[MSC v.1500 64 bit (AMD64)]'
>>> np.__version__
'1.7.1'
```

Several working examples are available, check:

https://github.com/jaimefrio/gufunc_sampler

everything you always wanted to know about ufuncs*

(*but were afraid to ask)

introducing gufuncs

So what exactly is a **gufunc**?

- A **g**eneralized **ufunc**

And a **ufunc** was?

- A **u**niversal **func**tion

All of the cool numpy stuff is built on four objects:

- `np.ndarray` – the multidimensional array,
- `np.dtype` – the data type descriptor,
- `np.nditer` – the array iterator, and
- `np.ufunc` – the universal functions

vectorization

ufuncs provide **vectorization**, i.e. implicit looping over all items of an array: speed-ups of **1000x** are typical (!!!)

What you would do in **Python**:

```
>>> a = [1, 2 ,3]
>>> b = [4, 5, 6]
>>> [i + j for i, j in zip(a, b)]
[5, 7, 9]
```

What you can do with **numpy**:

```
>>> a = np.array([1, 2 ,3])
>>> b = np.array([4, 5, 6])
>>> a + b
array([5, 7, 9])
```

built-in ufuncs

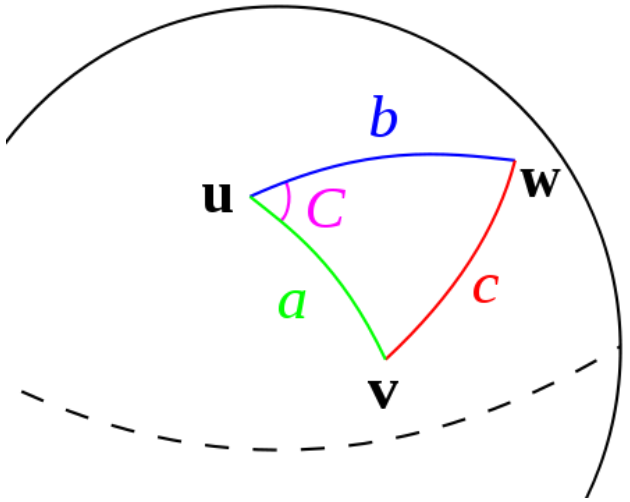
There are over 60 built-in **ufuncs** in **numpy**, see the complete list here:
http://docs.scipy.org/doc/numpy/reference/ufuncs.html

All basic operators on arrays are overloaded by a ufunc:

`+, -, *, /, **`	`np.add, np.subtract, np.multiply, np.divide, np.power`
`&, |, ^, ~`	`np.bitwise_and, np.bitwise_or, np.bitwise_xor, np.bitwise_not`
`>, <, >=, <=, ==, !=`	`np.greater, np.less, np.greater_equal, np.less_equal, np.equal, np.not_equal`

ufuncs as building blocks

ufuncs can be combined into (much) more complicated operations, e.g. the distance on the surface of a sphere:




```
def spherical_dist(pos1, pos2, r=3958.75):
    pos1, pos2 = np.deg2rad(pos1), np.degrad(pos2)
    csl_12 = np.cos(pos1[..., 0]) * np.cos(pos2[..., 0])
    cos_d = np.cos(pos1 - pos2)
    return r * np.arccos(cos_d[..., 0] –
                         csl_12 * (1 - cos_d[..., 1]))

>>> spherical_dist([32.7, 117.2],
...                [[42.3, 75.2], [40.4, 3.7]])
array([ 2369.69275626,  5843.31119488])
```

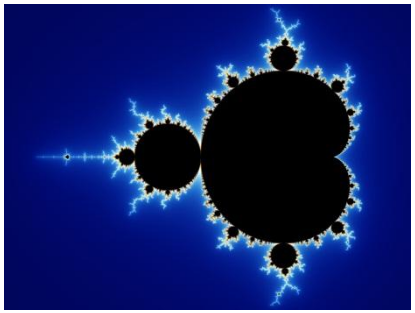
are we done then?

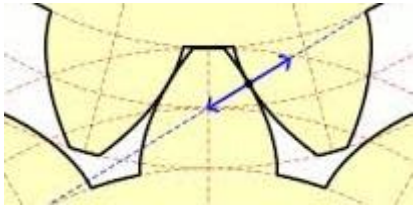
If **ufuncs** are so easy to combine together in Python, why would you ever want to code a new one in C?

- The quick answer is that **you probably won't** most of the time…
- …but that's not the case with **gufuncs**.

Intrinsically iterative computations may still benefit from a C coded ufunc, e.g.:

- the Mandelbrot set
- the inverse involute function





Check implementations of both here.

the letter g

How are **gufuncs** different from **ufuncs**?

- **ufuncs** operate on array items, one at a time
- **gufuncs** operate on subarrays, one at a time

Let's see it with an example:

```
>>> from numpy.core.umath_tests import inner1d
>>> print(inner1d.__doc__)
inner1d(x1, x2[, out])

inner on the last dimension and broadcast on the rest
     "(i),(i)->()"
```

the gufunc signature

All gufuncs have a signature:

```
>>> inner1d.signature
'(i),(i)->()'
```

What the signature tells us:

- It expects two inputs, `(i)` and `(i)`, each a 1D array
- The index `i` is repeated, so both inputs must have the same length
- It returns a single output, `()`, which is a zero-dimensional array, i.e. a scalar

is it any good?

`inner1d` computes “inner on the last dimension”, i.e. the inner (or dot) product of two vectors, here’s a pure Python equivalent:

```
a, b = [1, 2, 3], [4, 5, 6]
>>> sum(i * j for i, j in zip(a, b))
32
```

And while it works fine…

```
>>> inner1d(a, b)
32
```

…it duplicates `np.dot`, which is faster:

```
>>> np.dot(a, b)
32
```

So what exactly is the point?

broadcasting to the rescue

The cool stuff comes from “and broadcast on the rest”:

```
vecs = np.random.rand(1000, 3)
more_vecs = np.random.rand(1000, 3)
>>> inner_prods = inner1d(vecs, more_vecs)
```

This is equivalent, but (much) faster, than:

```
>>> dot_prods = np.empty(vecs.shape[:1])
>>> for j, (vec1, vec2) in enumerate(zip(vecs,
...                                      more_vecs)):
...     dot_prods[j] = np.dot(vec1, vec2)
...
>>> np.allclose(inner_prods, dot_prods)
True
```

broadcasting madness

More broadcasting trickery:

```
vecs = np.random.rand(1000, 3)
more_vecs = np.random.rand(2000, 3)
>>> inner_prods = inner1d(vecs[:, np.newaxis],
...                       more_vecs)
>>> inner_prods.shape
(1000L, 2000L)
```

- `inner_prods[i, j]` holds the inner product of `vecs[i]` with `more_vecs[j]`
- We can compute the Cartesian (dot) product of both sets of vectors with a single line of code (!!!)

another signature

A more complicated signature example:

```
>>> from numpy.core.umath_tests import matrix_multiply
>>> matrix_multiply.signature
'(m,n),(n,p)->(m,p)'
```

- Performs matrix multiplication (with broadcasting)
- Takes two 2D inputs, `(m,n)` and `(n,p)`, returns a single 2D output, `(m,p)`
- The columns of the first input must equal the rows of the second, `n`, and the output has as many rows as the first input, `m`, and as many columns as the second, `p`.

the DIY gufunc

are you in or are you out?

Lets try to put together a `point_in_polygon` gufunc:

- Takes a list of 2D points, the polygon vertices…
- …and a 2D point…
- …and returns a single boolean scalar, `True` if the point is inside the polygon, `False` if not

What should the signature be?

- This may seem like the right thing: `(n, 2),(2)->()`
- But we cannot have numbers in the signature, so perhaps this is it then: `(n, d),(d)->()`
- What do we do if `d != 2`? Raise an error? From C!?
- We could always wrap it in a Python function to handle error checking, but let's try something different…

enabling awesomeness

It is better to enforce the dimensionality in the signature:

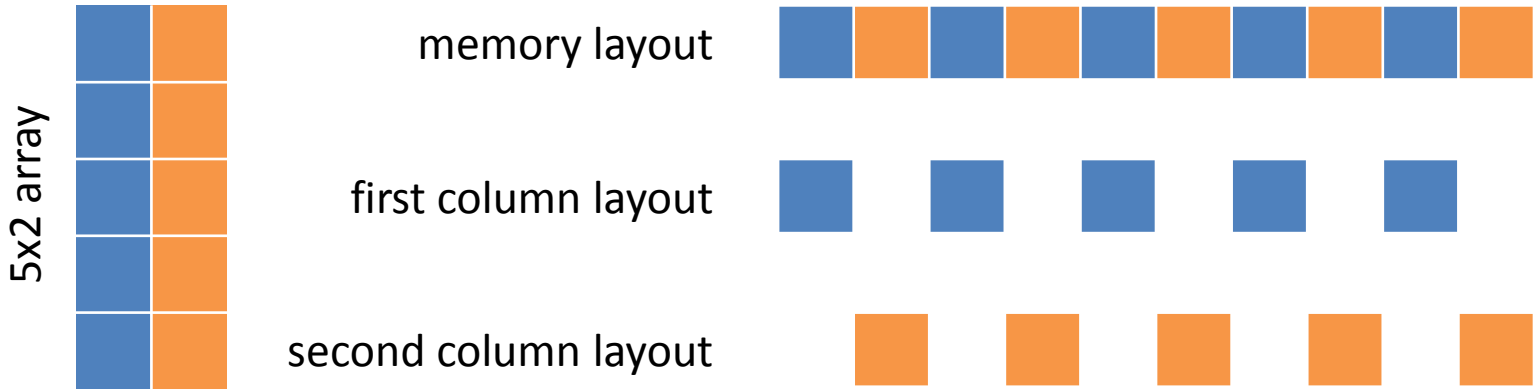
`(n),(n),(),()->()`

This will also allow some wicked broadcasting tricks, e.g. check for points in a grid inside the polygon with a single gufunc call:

```
>>> import point_in_polygon as pip
>>> poly_vtx = np.array([[1, 5], [4, 1], [6, 8]])
>>> grid_rows = np.arange(10)
>>> grid_cols = np.arange(20)
>>> pip.point_in_polygon(poly_vtx[:, 0], poly_vtx[:, 1],
...                      grid_rows[:, np.newaxis],
...                      grid_cols)
```

contiguousness

Note that the two arrays we are passing in for the vertices are not contiguous, i.e. the arrays are stored in row major order, and we send in a single column, so each array's stride is doubled:

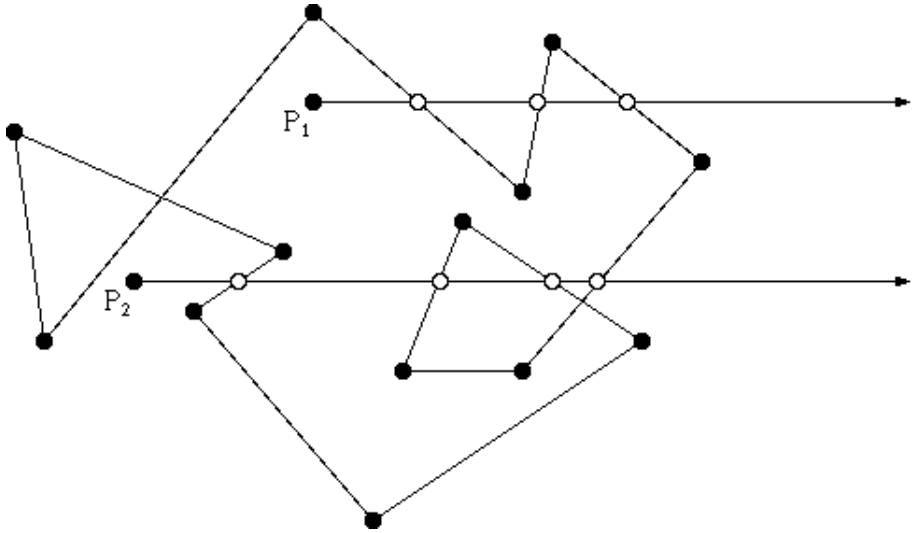


This is not a problem for a C gufunc, as we will soon experience, but it can be a headache e.g. for Cython

the algorithm

We will use the ray casting algorithm:

- A point is inside the polygon if a ray casted from it intersects an odd number of edges…
- …out otherwise



To check a horizontal ray from (px, py) against an edge going from (vxp, vyp) to (vx, vy) we do (XOR swaps parity):

```
if ((vy <= py and py < vyp) or
    (vyp <= py and py < vy)) and
   ((px - vx) < (vxp - vx) * (py - vy) / (vyp - vy)):
    out ^= 1
```

one kernel per type

Before we start happily coding away, some more info of what goes on under the hood of gufuncs (and ufuncs):

```
>>> np.add.types
['??->?', 'bb->b', 'BB->B', 'hh->h', 'HH->H', 'ii->i',
'II->I', 'll->l', 'LL->L', 'qq->q', 'QQ->Q', 'ee->e',
'ff->f', 'dd->d', 'gg->g', 'FF->F', 'DD->D', 'GG->G',
'Mm->M', 'mm->m', 'mM->M', 'OO->O']
>>> inner1d.types
['ll->l', 'dd->d']
```

- Each type listed is mapped to a C function operating on the specified data types
- `inner1d` only handles `longs` (`l`) and `doubles` (`d`)
- numpy will cast other types to one of those (if possible)

the kernel prototype

All the gufunc kernels, i.e. the actual C functions that do the heavy computational lifting, have the same prototype:

```
static void pip_kernel(char **args,
                       npy_intp *dimensions,
                       npy_intp *steps,
                       void *data)
```

How do these four arguments map to our vertices and points?

- the last one is easy: `data` is unused, don't worry about it

extracting the arguments

- `**args` is an array of pointers, one for each input, followed by one for each output.

- They come in as `char`, i.e. signed bytes, so we will need to do some casting.

- But lets extract the actual arguments:

```
char *vertex_x = args[0],
     *vertex_y = args[1],
     *point_x = args[2],
     *point_y = args[3],
     *out = args[4];
```

extracting the dimensions

- `*dimensions` is an array of `npy_intp`
- `npy_intp` is an integer type, large enough to hold a pointer

- It's first entry is the **loop length**, more on this later, followed by all the unique signature parameters, in the order in which they appear

- In our case, that's just the number of vertices, `n`:

```
npy_intp loop_len = dimensions[0],
         vtx_len = dimensions[1];
```

extracting the strides

- `*steps` is another array of `npy_intp`.
- It holds an entry per input and output, the **loop strides**…
- …followed by one entry for each dimension in the signature, the corresponding stride:

```
npy_intp vtx_x_loop_str = steps[0],
         vtx_y_loop_str = steps[1],
         pnt_x_loop_str = steps[2],
         pnt_y_loop_str = steps[3],
         out_loop_str = steps[4],
         vtx_x_str = steps[5],
         vtx_y_str = steps[6];
```

understanding strides

Lets first figure out what are `vtx_x_str` and `vtx_y_str`…

Both `vtx_x` and `vtx_y` are arrays, but they need not be contiguous in memory, so:

- `vtx_x_str` is the number of bytes in memory between `vtx_x` entries, and
- `vtx_y_str` is the number of bytes in memory between `vtx_y` entries

striding galore

- What about the **loop length** and **loop strides**?

- Even though our signature is:
  `(n),(n),(),()->()`
- the kernel must be written as if it were:
  `(m,n),(m,n),(m),(m)->(m)`

- The `m` above is the **loop length**, and the **loop strides** are the strides of the corresponding dimensions of the enlarged input arrays
- In a normal call, `m = 1`, and the loop strides are unused
- Setting things up like this makes calls with broadcasting more efficient

that's about it!

You now have access, within a C function, to the underlying numpy array data of your input and output arguments.

Go do whatever crazy thing you want to them!

There are important implementation details missing, check the github repository for examples:

- gufunc registration
- module initialization
- compiler set-up…

the signature's the limit

The output's shape has to be constructed entirely from the inputs' dimensions, this can be ~~a real bitch~~ challenging
A very ugly (but effective) way around it is having a spoof input the shape of the output, and using a Python wrapper:

- if we want a gufunc that does, `(n)->(2*n)`
- we code one that that does, `(n),(m)->(m)`
- and we wrap it in something like this:

```
def my_gufunc_wrapper(x):
    x = np.asarray(x)
    out = np.empty(x.shape[:-1] + (2*x.shape[-1],),
                   dtype=x.dtype)
    _my_gufunc(x, out, out=out)
    return out
```

the joy of compiling

compiling

Compiling and installing the extension module, once everything is properly set up, is a piece of cake, just run:

```
python setup.py install
```

Getting this to work under Windows is a little tricky:

- First, install the right Microsoft SDK version.
- You want `GRMSDKX_EN_DVD.iso` from:
  - Python 2.6, 2.7 & 3.2: SDK 3.5 SP1
  - Python 3.3: SDK 4
- Before running `setup.py`, do this in the SDK shell:

```
C:\Microsoft SDKs\Windows\v7.0>set DISTUTILS_USE_SDK=1
```

- and depending on your target (32 or 64 bit), also one of:

```
C:\Microsoft SDKs\Windows\v7.0>setenv /x86 /release
C:\Microsoft SDKs\Windows\v7.0>setenv /x64 /release
```