Using fortran from python

Nov 29, 2015 • Alex Rogozhnikov

I like **numpy** (a core library for numerical computations in python), but in some very-very rare cases I need to achieve the maximal possible speed of number crunching.

There is a number of options you can use (see my post with benchmarking of number crunchers). So the option I choose is to use fortran.

Several reasons for such inobvious choice:

- speed (fortran usually gives slightly faster programs then C++)
- vector operations, which shorten your code (i.e. to add two matrices and save it to third, you can write `x(:, :) = y(:, :) + z(:, :)`)
- simple memory management

However, there are demerits that I discovered during experimenting:

- 1-based indexing parameters.
  In fortran enumeration goes from 1 by default, but you can change it for all arrays, for the exception of assumed-shape parameters (and assumed-shape parameters are the most convenient way to interface with python)
- pointers cannot be set to allocatable arrays, which seems to be artificial limitation
- slow casting - you cannot convert int64[:] to int8[:], because such operations are forbidden. (however, you can use transfer, but this is copying, not casting)
- lack of support for unsigned integers (and ways to substitute it, for instance, `zext` functions)
- non-trivial vector operations frequently give worse results compared to explicitly written loops.

So just in case: fortran isn't friendly to different bit hacks.

Fortran magic

The most convenient way to experiment with python + fortran is fortran magic for ipython.

Installation is trivial: `pip install -U fortran-magic`

Create a new ipython notebook for experiments, now you can use fortran cells to define new functions, which will be automatically wrapped to numpy

```
In[1]: %load_ext fortranmagic  # activating magic

In[2]: %%fortran
       subroutine my_function(x, y, z)
            real, intent(in) :: x(:), y(:)
            real, intent(out) :: z(size(x))
            ! using vector operations
            z(:) = sin(x(:) + y(:))
       end subroutine

In[3]: # Now we can use this function
       import numpy
       x = numpy.random.normal(size=100)
       y = numpy.random.normal(size=100)
       z = my_function(x, y)
```

Result of computations `z` is numpy array. This code doesn't look like those old fortran77 programs with lots of 'C' and strange indentations, because it uses fortran 90 standard, which is rather permissive.

To debug compilation, use `-vvv` options, this will print all compiler warnings and output:

```
In[4]: %%fortran -vvv
       subroutine my_function(x, y, z)
            real, intent(in) :: x(:), y(:)
            real, intent(out) :: z(size(x))
            ! using vector operations
            z(:) = sin(x(:) + y(:))
       end subroutine
```

If you want to use `openmp` with fortran in python, this is how you can do it with gfortan:

```
In[5]: %%fortran -vvv --f90flags='-fopenmp' --extra='-lgomp'
       subroutine my_function(x, y, z)
            use omp_lib
            real, intent(in) :: x(:), y(:)
            real, intent(out) :: z(size(x))
            ! your code
       end subroutine
```

Also useful:

- `--opt='-O3'` for O3 optimization level

- Fortran 90 best practices

F2py

Fortran magic uses `f2py` under the hood. `f2py` currently is a part of `numpy` installation. That's the function you need to get `openmp + fortan 90` working with python:

```
def compile_fortran(source, module_name, extra_args=''):
    import os
    import tempfile
    import sys
    import numpy.f2py  # just to check it presents
    from numpy.distutils.exec_command import exec_command

    folder = os.path.dirname(os.path.realpath(__file__))
    with tempfile.NamedTemporaryFile(suffix='.f90') as f:
        f.write(source)
        f.flush()

        args = ' -c -m {} {} {}'.format(module_name, f.name, extra_args)
        command = 'cd "{}" && "{}" -c "import numpy.f2py as f2py;f2py.main()" {}'
        status, output = exec_command(command)
        return status, output, command
```

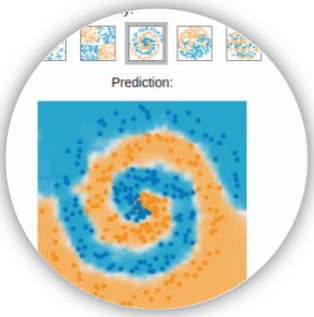
Now, the usage is:

```
fortran_source = '''
  subroutine my_function(x, y, z)
      real, intent(in) :: x(:), y(:)

      real, intent(out) :: z(size(x))
      ! using vector operations
      z(:) = sin(x(:) + y(:))
  end subroutine
'''

status, output, command = compile_fortran(fortran_source, modulename='mymodule',
                                          extra_args="--f90flags='-fopenmp' -lgor
```

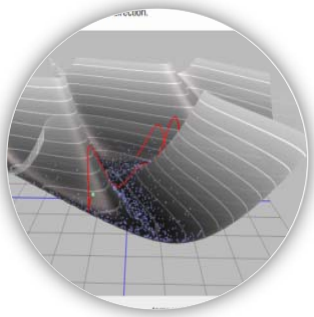
Now we can use the compiled module:

```
from mymodule import my_function
```

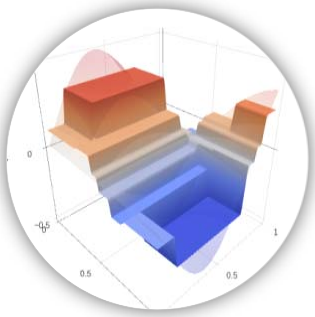
Top posts at "brilliantly wrong": (all posts)



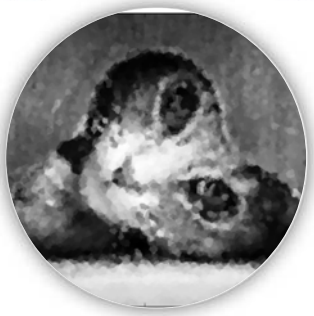
Gradient boosting playground



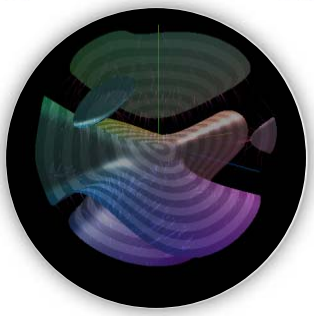
Hamiltonian MC explained



Gradient boosting explained



Reconstructing pictures with ML



Neural Networks visualized in 3d

- **Jupyter (IPython) notebooks features**
- Numpy tips and tricks: **part 1**, **part 2**
- **Reweighting with Boosted Decision Trees**
- **Machine Learning in Science and Industry**

- **Speed benchmarks: numpy vs all.**
- Machine learning in COMET: **part 1**, **part 2**
- **ROC curve explained**
- **Optimal control of oscillations**

Brilliantly wrong

Alex Rogozhnikov
alex.rogozhnikov@yandex.ru

arogozhnikov

Brilliantly Wrong — Alex Rogozhnikov's blog about math, machine learning, programming and high energy physics.