Pythran User Manual

So you want to write algorithms that are easy to maintain as in python and you want performance as in FORTRAN or C++? Let give a try to Pythran! Pythran is a python-to-c++ translator that turns python module into native c++11 module. From a user point of view, you still `import` your module, but under the hood... there is much more happening!

Disclaimer

Pythran is *not* a full python-to-c++ converter, as in *shedskin*. Instead it takes a subset of the python language and turns it into heavily templatized c++ code instantiated for your particular types.

Say hello to:

- polymorphic functions (!)
- lambdas
- list comprehension
- map, reduce and the like
- dictionaries, set, list
- exceptions
- file handling
- (partial) *numpy* support

Say bye bye to:

- classes
- polymorphic variables [but not all of them :-)]

In a nutshell, Pythran makes it possible to write numerical algorithms in Python and to have them run faster. Nuff said.

Prerequisite

Let us assume you're running a Debian/Ubuntu distrib. In that case, all you have to do is:

```
$> sudo apt-get install libboost-python-dev libgoogle-perftools-dev libgmp-dev python-ply python-networkx cmake
```

If you prefer the `easy_install` way, from `setuptools` (`python-setuptools` package under Debian), the way to go is:

```
$>  PYTHONPATH=<my_prefix>/lib/python<my_version>/site-packages \
                easy_install --prefix <my_prefix> ply networkx
```

and set your `PYTHONPATH` appropriately, something like:

```
$> export  PYTHONPATH=<my_prefix>/lib/python<my_version>/site-packages
```

You also need a modern C++11 enabled compiler (e.g. g++>=4.7), that supports for instance atomic operations (N3290) or variadic template (N2555).

Installation

You're almost done! From the source directory, run:

```
$> python setup.py install --prefix=<my_prefix>
```

And set your path to:

```
$> export PATH=$PATH:<my_prefix>/bin
```

It makes the `pythran` command available to you.

Making Sure Everything is working

The `setup.py` scripts automates this. The `test` target, as in:

```
python setup.py test
```

runs a whole (and long) validation suite. The `bench` target, as in:

```
python setup.py bench
```

compares the performance of Pythran-generated code with CPython.

If these tests fail, you are likely missing some of the requirements. You can set site specific flags in your `~/.pythranrc`:

```
[sys]
cppflags = -DENABLE_PYTHON_MODULE
cxxflags = -std=c++11
# update your library names here
ldflags =   -lboost_python
            -lgmp
            -lgmpxx

[user]

cxx = c++
cppflags =
cxxflags = -O2 -g
# update your library paths here
ldflags = -fPIC -ltcmalloc_minimal
```

Be careful, the *user* section is overwritten by the command-line flags!

First Steps

To begin with, you need... a python function in a module. Something like:

```
<<dprod.py>>
def dprod(l0,l1):
        return sum([x*y for x,y in zip(l0,l1)])
```

will be perfect. But due to _o< typing, `l0` and `l1` can be of any type, so Pythran needs a small hint there. Add the following line somewhere in your file, say at the top head, or right before the function definition:

```
#pythran export dprod(int list, int list)
```

This basically tells Pythran the type of the forthcoming arguments.

Afterwards, frenetically type:

```
$> pythran dprod.py
```

o/ a `dprod.so` native module has been created and you can play with it right *now*. The speedup will not be terrible because of the conversion cost from python to C++.

So let's try again with a well-known example. Let me introduce the almighty *matrix multiply*!:

```
<<mm.py>>
def zero(n,m): return [[0]*n for col in range(m)]
def matrix_multiply(m0, m1):
        new_matrix = zero(len(m0),len(m1[0]))
        for i in range(len(m0)):
                for j in range(len(m1[0])):
                        for k in range(len(m1)):
                                new_matrix[i][j] += m0[i][k]*m1[k][j]
        return new_matrix
```

This a slightly more complex example, as a few intrinsics such as `range` or `len` are used, with a function call and even nested list comprehension. But Pythran can make its way through this. As you only want to export the `matrix_multiply` function, you can safely ignore the `zero` function and just add:

```
#pythran export matrix_multiply(float list list, float list list)
```

to the source file. Note how Pythran can combine different types and infers the resulting type. It also respects the nested list structure of python, so you are not limited to matrices...

Enough talk, run:

```
$> pythran mm.py
```

One touch of magic wand and you have your native binary. Be amazed by the generation of a `mm.so` native module that run around 20x faster than the original one. `timeit` approved!

But scientific computing in Python usually means Numpy. Here is a well-known Numpy snippet:

```
    <<arc_distance.py>>
import numpy as np
def arc_distance(theta_1, phi_1, theta_2, phi_2):
    """
    Calculates the pairwise arc distance
    between all points in vector a and b.
    """
    temp = (np.sin((theta_2-theta_1)/2)**2
           + np.cos(theta_1)*np.cos(theta_2) * np.sin((phi_2-phi_1)/2)**2)
    distance_matrix = 2 * np.arctan2(sqrt(temp), sqrt(1-temp))
    return distance_matrix
```

This example uses a lot of Numpy *ufunc*. Pythran is reasonably good at handling such expressions. As you already now, you need to **export** it, giving its argument type by adding:

```
#pythran export arc_distance(float[], float[], float[], float[])
```

To the input file. You can compile it as the previous code:

```
$> pythran arc_distance.py
```

and you'll get a decent binary. But what you really wanted to do was:

```
$> pythran -fopenmp -march=avx arc_distance.py
```

which basically tells the compiler to parallelize and vectorize loops. Then you'll get **really** fast code!

Concerning Pythran specifications

The `#pythran export` commands are critical to Pythran. In fact if they are missing, Pythran will complain loudly (and fail miserably). So let us dive into these complex language!

There is currently only one Pythran command, the `export` command. Its syntax is:

```
#pythran export function_name(argument_type*)
```

where `function_name` is the name of a function defined in the module, and `argument_type*` is a comma separated list of argument types, composed of any combination of basic types and constructed types. What is a basic type? Anything that looks like a python basic type! Constructed types are either tuples, introduced by parenthesis, like `(int, (float, str))` or lists (resp. set), introduced by the `list` (resp. `set`) keyword:

```
argument_type = basic_type
                          | (argument_type*)    # this is a tuple
                          | argument_type list  # this is a list
                          | argument_type set   # this is a set
                          | argument_type:argument_type dict    # this is a dictionary

basic_type = bool | int | long | float | str
```

Easy enough, isn't it?

Note

It is in fact possible to analyse a code without specifications, but you cannot go further that generic (a.k.a. heavily templated) c++ code. Use the `-e` switch!

Advanced Usage

A failing compilation? A lust of c++ tangled code? Give a try to the `-E` switch that stops the compilation process right after c++ code generation, so that you can inspect it.

Want more performance? Big fan of `-Ofast -march=native`? Pythran automagically forwards these switches to the underlying compiler! Pythran is sensible to the `-DNDEBUG` switch too.

Tired of typing the same compiler switches again and again? Store them in `$XDG_CONFIG_HOME/.pythranrc`!

Wants to try your own compiler? Update the *c++* field from your *pythranrc*!

The careful reader might have noticed the `-p` flag from the command line. It makes it possible to define your own optimization sequence:

```
pythran -pConstantFodling -pmy_package.MyOptimization
```

runs the ConstantFolding optimization from pythran.optimizations followed by a custom optimization found in the my_package package, loaded from PYTHONPATH.

Adding OpenMP directives

OpenMP is a standard set of directives for C, C++ and FORTRAN that makes it somehow easier to turn a sequential program into a multithreaded one. Pythran translates OpenMP-like code annotation into OpenMP directives:

```
r=0
"omp parallel for reduction(+:r) private(x,y)"
for x,y in zip(l1,l2):
    r+=x*y
```

Note that as in python, all variables have function-level scope, x and y must be explicitly listed as private variables.

OpenMP directive parsing is enabled by -fopenmp when using g++ as the backend compiler.

Alternatively, one can run the great:

```
pythran -ppythran.analysis.ParallelMaps -e as.py
```

which runs a code analyzer that displays extra information concerning parallel map found in the code.

Getting Pure C++

Pythran can be used to generate raw templated C++ code, without any python glue. To do so use the -e switch. It will turn the python code into c++ code you can call from a C++ code. In that case there is **no** need for a particular Pythran specification.

F.A.Q.

1. Supported compiler versions:
   - *g++* version 4.7
   - *clang++* version 3.1-8

Troubleshooting

Plenty of them! Seriously, Pythran is software, so it will crash. You must make it abort in unusual ways! And more important, you must provide feedback to serge_sans_paille using its email serge.guelton@telecom-bretagne.eu, the IRC channel #pythran on FreeNode, or the mailing list pythran@freelists.org

glhf!