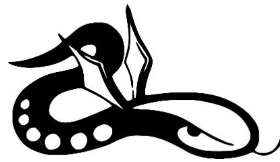
Building SciPy kernels with Pythran

Serge Guelton - *Pythran main dev, Namek*
Ralf Gommers - *SciPy maintainer, Quansight Labs*

14 July 2021

Why we embarked on this journey

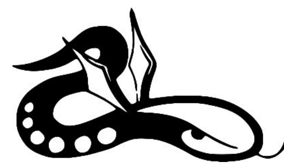


SciPy contains a lot of algorithmic code. It needs to be fast.
Existing approaches in SciPy were:

- Python: for the glue, and non critical parts
- Cython: for critical parts
- Fortran 77: for very old critical parts
- C & C++: for ultra critical parts :-)

Our goal: make it easier to write fast SciPy kernels!

The Pythran approach



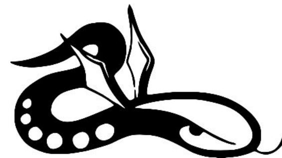
Keep input code portable and high-level:

- takes pure Python code as input
- understands NumPy high-level constructs
- delivers performance by transpiling to C++

But still:

- efficient explicit looping in Python
- without any runtime dependencies

A typical Pythran kernel for SciPy



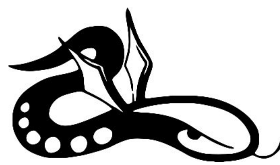
```
#pythran export _max_len_seq_inner(intp[], int8[], int, int, int8[])
def _max_len_seq_inner(taps, state, nbits, length, seq):
    n_taps = taps.shape[0]
    idx = 0
    for i in range(length):
        feedback = state[idx]
        seq[i] = feedback
        for ti in range(n_taps):
            feedback ^= state[(taps[ti] + idx) % nbits]
        state[idx] = feedback
        idx = (idx + 1) % nbits
    return np.roll(state, -idx, axis=0)
```

⇐ *Explicit looping*

⇖ *Explicit indexing* ⇙

Understands NumPy function calls

Works in a Jupyter notebook



```
[1]: import numpy as np
     import pythran
     %load_ext pythran.magic
```

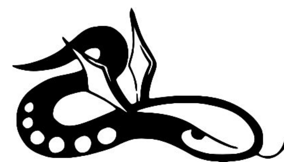
```
[2]: %%pythran
     import numpy as np

     # pythran export polynomial_matrix(float[:, :], int[:, :])
     def polynomial_matrix(x, powers):
         out = np.empty((x.shape[0], powers.shape[0]), dtype=float)
         for i in range(x.shape[0]):
             for j in range(powers.shape[0]):
                 out[i, j] = np.prod(x[i]**powers[j])

         return out
```

```
[3]: x = np.random.rand(3, 4)
     powers = np.arange(12).reshape((3, 4))
```

```
[4]: %timeit polynomial_matrix(x, powers)  # pure Python version takes 33 us
```

```
377 ns ± 3.91 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

Easy build system integration

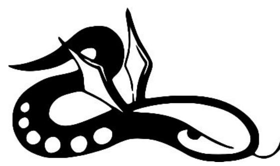


```
from distutils.core import setup
from pythran.dist import PythranExtension, PythranBuildExt

setup(...,
        ext_modules=[PythranExtension("mymodule", ["mymodule.py"])],
        cmdclass={"build_ext": PythranBuildExt})
```

Or precompile to C++ to use with any build system:

```
$ pythran -E mykernel.py -o mykernel.cpp
```

Isn’t Cython enough?



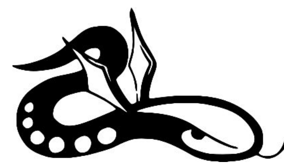
Cython is a ***great*** tool

- incremental conversion / mixed mode
- great for gluing existing native code/library with Python
- good portability, no runtime requirements

However, keeping in mind our “easier to write” goal:

- still has a non-negligible learning curve
- tends to be closer to C than Python when performance matters

Then what about Numba?



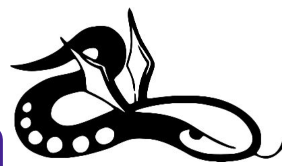
Numba is a ***great*** tool

- Just-in-Time compilation
- GPU support
- pure Python syntax

However:

- It has more runtime dependencies
- tends to require lower-level programming for best performance

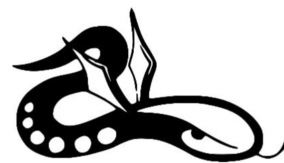
Comparing Cython, Numba & Pythran



	Cython	Numba	Pythran
Portability	++	+	++
Runtime dependency	++	--	++
Maturity	++	+	+
Maintenance status	0	+	+
Features	++	+	0
Ease of use	--	++	+
Debugging & optimization	0	+	0
Size of binaries	-	++	+

For all tools: performance excellent, bus factor is ~ 1-2

When do I use which tool?



Our advice:

- For higher-level, pure Python packages: use Numba
- If you have any compiled code in your package:
  - Use Pythran for standalone kernels
  - Use Cython for binding C/C++ code, and in case you need to interact with the Python or NumPy C API

SciPy build/run-time dependencies

Build-time dependency

Build-time & run-time dependency

SciPy

NumPy

OpenBLAS
or
MKL
or
...

pybind11

BLAS/LAPACK

Cython

Pythran

Fortran compiler

clang-cl.exe

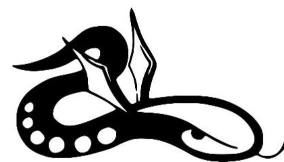
Python

C++ compiler

C compiler

Build system not shown;
SciPy dependencies on Python
and C compiler not drawn.

Current Pythran usage in SciPy



One large extension: `RBFInterpolator`

Several smaller extensions:

```
$ git grep -l '#pythran'
scipy/optimize/_group_columns.py
scipy/signal/_max_len_seq_inner.py
scipy/signal/_spectral.py
scipy/stats/_hypotests_pythran.py
```

More PRs in progress

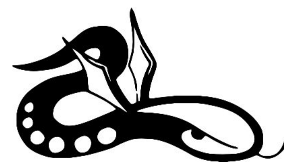
GSoC student - Xingyu Liu

Xingyu is going through SciPy's code base, looking for kernels to benchmark and accelerate:

- `stats.binned_statistic_dd`: 2-30x speedup
- `stats.somersd`: 4-20x speedup
- `spatial.SphericalVoronoi.sort_vertices_of_regions`: 3x speedup

With more to come; read the blog of her journey at https://blogs.python-gsoc.org/en/xingyu-lius-blog/

Benefits for SciPy



Key benefit: easiest way to write fast kernels

- Developer experience about as good as with Numba, accessible to almost every contributor
- It's fast - typically >= Cython, even without SIMD
- Produced binaries are much smaller than those from Cython
- Pythran itself is easy to contribute to, and has a responsive maintainer
- Build system integration is easy(-ish)

Pythran limitations

There are still gaps in functionality, not all of NumPy is covered:

- numpy.random
- APIs with too much "dynamic behaviour" (e.g., `keepdims` keyword)
- There is no escape hatch - if it's not supported, it must be added to Pythran itself first
- No threading in SciPy. Pythran can use OpenMP, but this is forbidden in SciPy (only custom thread pools allowed).
- Extra constraint on Windows: must build with clang-cl

Integration status

Currently Pythran is:

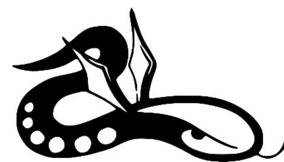
- enabled by default in the SciPy build
- Still an optional dependency (to disable: `export SCIPY_USE_PYTHRAN=0`)

Lessons from the recent SciPy 1.7.0 release:

- A small portability issue on AIX (already resolved)
- Status with PyPy unclear (PyPy + SciPy has other issues, so can't test)
- Other than that, mostly smooth sailing

Initial integration required two Pythran releases to fix some build issues

Conclusions



- **SciPy contributors like Pythran!** *“This is very elegant”, “Surprised it’s that fast”*
- **Initial goal achieved!** *Pythran is indeed an easier way to write fast kernels*
- **The journey continues!** *Pythran will likely become a hard build-time dependency for or after SciPy 1.8.0*

Bonus question: can we combine Pythran with CuPy's Python-to-CUDA JIT? It emits C++ code too, so we could get fast CPU + GPU code like that.