Inside SciPy

Optimizing Scientific Python

Kai Striega

Cartesian Software & SciPy

2025-06-26

How to get the slides

- Available on GitHub
- `https://github.com/Kai-Striega/speeches/blob/main/inside-scipy-rgi/out/inside_scipy_rgi.pdf`

Who am I, and how do I fit in?

- Hi, I'm Kai!
- Maintainer of SciPy since 2018
- Senior Software Engineer at Cartesian Software
- BSc (Mathematics)
- **Really don't like slow code**

What this talk is about

- Show how we approach performance optimisation
- Using SciPy's RegularGridInterpolator as an example
- What we'll cover (quickly)
    - Benchmarking
    - Profiling
    - Extending Python with native languages
    - Cython, and Cython optimisations

Disclaimers & Thanks

- This is the abridged version

Disclaimers & Thanks

- This is the abridged version
- This is only *partly* my own work

Disclaimers & Thanks

- This is the abridged version
- This is only *partly* my own work
- Co-contributors:
  - Evgeni Burovski (@ev-br)

Disclaimers & Thanks

- This is the abridged version
- This is only *partly* my own work
- Co-contributors:
  - Evgeni Burovski (@ev-br)
- Reviewers/Mentors:
  - Julien Jerphanion (@jjerphan)
  - Pamphile Roy (@tupui)

About SciPy

- "Fundamental algorithms for scientific computing in Python"
- Free and Open Source
- Broadly applicable
- Foundational
- Easy to use
- **Performant**

SciPy: A *performant* Python library?

- SciPy wraps highly-optimized implementations written in low-level languages
- Enjoy the flexibility of Python with the speed of compiled code
- Is it performant?

SciPy: A *performant* Python library?

- SciPy wraps highly-optimized implementations written in low-level languages
- Enjoy the flexibility of Python with the speed of compiled code
- Is it performant?
- Yes!

SciPy: A *performant* Python library?

- SciPy wraps highly-optimized implementations written in low-level languages
- Enjoy the flexibility of Python with the speed of compiled code
- Is it performant?
- Yes!
- . . . *most of the time*

Regular Grid Interpolation: One of SciPy's tools

- SciPy had a function *interp2d*
- *interp2d* interpolated points on a regular grid in 2d space
- Written in Fortran
- It was removed in SciPy 1.14.0 due to being unmaintainable and (very) buggy

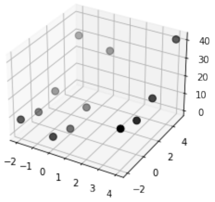

Figure: Some point to interpolate.

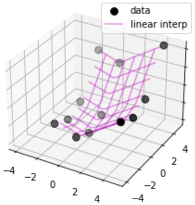

Figure: The interpolated values.

RegularGridInterpolator

- *interp2d* was replaced by several new classes
- One of these is the *RegularGridInterpolator* (RGI) class
- Which is user-friendly, written in pure python and less buggy

Benchmarking

- Benchmarking means to evaluate or test a system in order to gain an idea of its performance level.
- **Benchmarking is complicated**
- SciPy has its own benchmarking suite based on air speed velocity
- All benchmarks were conducted on the same machine with an i9-12900k processor

An example us of the *RGI* class

```
import numpy as np
from scipy.interpolate import RegularGridInterpolator

n_points_in_dim = 2 ** 14  # 16,384 points
rng = np.random.default_rng(42)
x = np.arange(n_points_in_dim)
y = np.arange(n_points_in_dim)
z = rng.uniform(size=(n_points_in_dim, n_points_in_dim))

xg, yg = np.meshgrid(x, y, indexing='ij', sparse=True)
rgi = RegularGridInterpolator((x, y), z.T, method='linear')
rgi((xg, yg))
```

Benchmarking Results

Scenario	Time (s)	Relative Speedup
Raw Python	36.8	1.0

Is this *fast*?

- We had a second function *interp2d*
- I originally claimed that this was faster
- Let's see how it compares ...

interp2d benchmark result

Scenario	Time (s)	Relative Speedup
Raw Python	36.8	1.0
interp2d	11.7	3.17

Is this *fast*?

- *interp2d* is $>$ 3 times faster than *RGI*

Is this *fast*?

- *interp2d* is >3 times faster than *RGI*
- :(

Where are we wasting time?

- A profile is a set of statistics that describes how often and for how long various parts of the program are executed
- Useful as we can focus our attention on the slowest parts of the code
- We will use Python's inbuilt *cProfile* module
- ... but there are many other profilers, I recommend *Scalene*

Abridged Profiling Output

123 function calls in 63.184 seconds

ncalls	tottime	% time	function
1	26.850	42	_evaluate_linear
1	14.728	23	_find_indices
2	9.042	14	'searchsorted' of 'numpy.ndarray'
	⋮		

Abridged Profiling Output

123 function calls in 63.184 seconds

ncalls	tottime	% time	function
1	26.850	42	_evaluate_linear
1	14.728	23	_find_indices
2	9.042	14	'searchsorted' of 'numpy.ndarray'
	⋮		

- Profiling adds **overhead**; our run has slowed down significantly
- We spend $\approx$ 65% of time in the functions _evaluate_linear and _find_indices
- Let's look at those!

_find_indicies

```
def _find_indices(self, xi):
    indices = []
    norm_distances = []

    for x, grid in zip(xi, self.grid):
        i = np.searchsorted(grid, x) - 1
        i[i < 0] = 0
        i[i > grid.size - 2] = grid.size - 2
        indices.append(i)

        denom = grid[i + 1] - grid[i]
        with np.errstate(divide='ignore', invalid='ignore'):
            norm_dist = np.where(denom != 0, (x - grid[i]) / denom, 0)
        norm_distances.append(norm_dist)
    return indices, norm_dist
```

_evaluate_linear

```python
def _evaluate_linear(self, indices, norm_distances):
    vslice = (slice(None),) + (None,) * (self.values.ndim - len(indices))
    shift_norm_distances = [1 - yi for yi in norm_distances]
    shift_indices = [i + 1 for i in indices]

    zipped1 = zip(indices, shift_norm_distances)
    zipped2 = zip(shift_indices, norm_distances)

    hypercube = itertools.product(zipped1, zipped2)
    value = np.array([0.])
    for h in hypercube:
        edge_indices, weights = zip(*h)
        weight = np.array([1.])
        for w in weights:
            weight = weight * w
        term = np.asarray(self.values[edge_indices]) * weight[vslice]
        value = value + term
    return value
```

Speeding up Python

- Things to consider:
  1. Algorithmic Complexity
  2. More optimal datastructures (np.array vs lists/tuples)
  3. Micro optimisations
- Pure Python $\implies$ computational overhead
- Could remove overhead by extending with a low level language

Extending Python

- Manual C/C++ extension module
- Fortran + f2py
- Numba
- Pythran
- Cython

Cython

- Compiles a super set of Python to C
- Supports optional static type declarations
- Allows for very fast program execution

Naively Cythonizing

- Cython can be run on any .py/.pyx file
- Generates a .c file with python bindings
- What happens if we move *_evaluate_linear* and *_find_indices* to a .pyx file and compile it?

Naive Cython benchmark result

Scenario	Time (s)	Relative Speedup
Raw Python	36.8	1.0
interp2d	11.7	3.17
Naive Cython	46.4	0.79

Wait, Cython is slower!?

- Never seen this before
- Don't really understand why
- Hypothesis:
  - Some overhead converting Python data to C data
  - Compiler can't optimise it well without static typing
  - Heavy use of Python functions such as "itertools", "zip"

Investigating Python Interaction

- Cython compiles Python to C
- This does not mean it avoids the Python interpreter
- The interpreter is (usually) slow
- "–annotate" generates a HTML file that shows us how Cython interacts with the interpreter
- Yellow highlighting $\implies$ interaction with the interpreter

View the Naive Cython interactions

Open the Naive Cython Annotations file

Add static typing

- Cython supports static type annotations
- Allows Cython to bypass the dynamic nature of Python
- All C types are available for annotation

Remove Python specific functions

- zip
- list comprehensions
- itertools.product
- np.where
- np.searchsorted

Use memory views

- Everything in Python is an object
- Every Python object contains a pointer to the heap
- A list of numbers is a list of pointers to numbers
- ... which is **very** slow
- Memory views directly store a block of contiguous numbers

View the Typed Cython interactions

Open the Typed Cython Annotations file

Typed Cython benchmark result

Scenario	Time (s)	Relative Speedup
Raw Python	36.8	1.0
interp2d	11.7	3.17
Naive Cython	46.4	0.79
Typed Cython	18.8	1.95

Compiler Directives

- Compiler directives are instructions that affect the behaviour of the Cython code
- Can be used to further speed up the behaviour of the code
- Make a trade-off between performance and being Pythonic

```
@cython.wraparound(False)
@cython.boundscheck(False)
@cython.cdivision(True)
@cython.initializedcheck(False)
```

Final Cython benchmark result

Scenario	Time (s)	Relative Speedup
Raw Python	36.8	1.0
interp2d	11.7	3.17
Naive Cython	46.4	0.79
Typed Cython	18.8	1.95
Final Cython	16.4	2.24

Consequences

- Sped up the benchmark by a factor of $\approx$2
  - Lower end of gains usually seen with Cython
  - Hints at large Python interaction
  - Only Cythonized the parts that are bottlenecks
- Added a compilation step to the project
  - SciPy already has a compilation step
  - May matter for your project
- Made the code less Pythonic

This work is not finished!

Cython benchmark result

Scenario	Time (s)	Relative Speedup
Raw Python	36.8	1.0
Naive Cython	46.4	0.79
Typed Cython	18.8	1.95
Final Cython	16.4	2.24
Your Contribution	?	?