Improve performance through the use of Pythran

Self-Introduction

- Name: Xingyu Liu
- University: Harvard University, MA, USA
- Major: First Year Data Science Master Student
- GitHub: https://github.com/charlotte12l
- Previous Programming Experience:
    - Programming Language: Python,C/C++/C++11,Shell,MATLAB,Assembly
    - Selected Projects
        - EM-pipeline-mito: Mitochondria Segmentation via deep learning, paper accepted by MICCAI 2020 (top-tier medical imaging conference)
        - Awesome-AD: A package of automatic differentiation with both Forward Mode and Reserve Mode. Not open source yet.
        - Radiomics_System: A radiomics-computing system which supports 3D medical image visualization, annotation, feature extraction and analysis.

Past Contribution to SciPy

- BENCH: add benchmark for Kendalltau
- BUG: fix cosine distance range to 0-2
- BUG: update _kendall_p_exact ValueError to f-string
- substitute np.math.factorial with math.factorial : `np.math` is an alias of the stdlib `math` module. Here I removed the `np.`
- Do not use numpy 1.20(conda-forge) if you are setting up a development environment on MacOSX : I shared some bugs and solutions during building SciPy in this issue
- Pythran_vs_cython: did a small experiment on `_kendall_dis()` in scipy.stats._stats.pyx to compare the performance of Pythran vs Cython

Project Summary

Currently, there are a lot of algorithms in scipy that use Cython to improve the performance of code that would be too slow as pure Python, e.g. algorithms in scipy.interpolate, scipy.spatial, `scipy.stats` and `scipy.optimize`. Also, there are some algorithms that use Pythran for performance improvement, like `scipy.optimize._group_columns, scipy.signal._spectral`.

Cython is a superset of the Python language which can be compiled to C-code. Ideally, if one eliminates most accesses to Python objects it can achieve almost the same speed as C. However, the more you optimize Cython the more it will look like C and less like Python [1] .Comparing with Cython, Pythran is an ahead of time compiler for numerical and scientific python that can take advantage of SIMD instructions to speed up your code[1]. Besides, Pythran is relatively easier to use - no need to rewrite the code, it takes a Python module annotated with a few interface descriptions and turns it into a native Python module with the same interface, but faster[2]. Therefore, I choose to use Pythran to improve the performance.

After some investigation, I found the following algorithms could be accelerated (not finalized yet, may add more during GSoC).

- Scipy.stats
  - multiscale_graphcorr
  - Ttest_ind
  - _kendall_p_exact
  - _count_paths_outside_method
  - _compute_prob_inside_method
  - _cdf_distance
  - _moment
- scipy.optimize
  - differential_evolution
  - _nonmonotone_line_search_cruz
  - _nonmonotone_line_search_cheng
- scipy.sparse.random
- scipy.ndimage.generic_filter

Also, I noticed that many algorithms currently do not have corresponding benchmarks, e.g., in scipy.stats, most of the Inferential Stats functions, many Correlation Functions, ANOVA Functions. I plan to write benchmarks for them, too.

Project Design

Part A: Writing Benchmark

Currently, SciPy uses Airspeed Velocity for performance benchmarking.
All benchmarks are in benchmarks/. To write a benchmark, we need to specify the params_names and params, do setup and call the function in time_xxx(). Below is a benchmark I wrote and I will also follow this code style in the future.

```
class Kendalltau(Benchmark):
   param_names = ['nan_policy','method','variant']
   params = [
       ['propagate', 'raise', 'omit'],
       ['auto', 'asymptotic', 'exact'],
       ['b', 'c']
   ]

   def setup(self, nan_policy, method, variant):
       np.random.seed(12345678)
       a = np.arange(200)
       np.random.shuffle(a)
       b = np.arange(200)
       np.random.shuffle(b)
       self.a = a
       self.b = b
```

```
    def time_kendalltau(self, nan_policy, method, variant):
        tau, p_value = stats.kendalltau(self.a, self.b, nan_policy=nan_policy, method=method,
variant=variant)
```

To run the benchmark, do `python runtests.py --bench stats.Kendalltau` and we can get the following output.

```
(scipydev) charlotte@CHARLOTLIU-MB0 scipy % python runtests.py --bench stats.Kendalltau
Building, see build.log...
Build OK (0:00:08.047040 elapsed)
Running benchmarks for Scipy version 1.7.0.dev0+f679c65 at /Users/charlotte/Desktop/internship/gsoc/scipy/build/
· No `environment_type` specified in asv.conf.json. This will be required in the future.
· Discovering benchmarks
· Running 1 total benchmarks (1 commits * 1 environments * 1 benchmarks)
[  0.00%] ·· Benchmarking existing-py_Users_charlotte_anaconda3_envs_scipydev_bin_python
[ 50.00%] ··· Running (stats.Kendalltau.time_kendalltau--).
[100.00%] ··· stats.Kendalltau.time_kendalltau                                                ok
[100.00%] ··· ============ ============ ============= =============
              --                                variant
              ------------------------ -----------------------------
               nan_policy     method          b             c
              ============ ============ ============= =============
               propagate       auto        209±10µs      250±8µs
               propagate    asymptotic     202±10µs      257±20µs
               propagate      exact       9.01±0.4ms   8.68±0.09ms
                 raise         auto        199±3µs       236±2µs
                 raise      asymptotic     198±2µs       238±6µs
                 raise        exact      8.61±0.05ms   8.62±0.05ms
                  omit         auto        199±7µs       238±6µs
                  omit      asymptotic     199±3µs       231±4µs
                  omit         exact      8.60±0.04ms   8.66±0.09ms
              ============ ============ ============= =============
```

To visualize the all the benchmarks, do

```
asv run --skip-existing-commits --steps 10 ALL
asv publish
asv preview
```

Open http://127.0.0.1:8080/ on your local machine and then you can see the visualization results.

```
(scipydev) charlotte@CHARLOTLIU-MB0 benchmarks % asv preview
· Serving at http://127.0.0.1:8080/
· Press ^C to abort
```

Part B: Improving Performance

Currently, there are many slow algorithms in SciPy that could be accelerated, but how can we identify the potential algorithms ? Pytest and line_profiler are good profiling tools that would be useful here. After finding the slow algorithms, we can use Pythran to accelerate them. I will use `kendall_p_exact` as an example to illustrate the following steps:

1. Identify slow algorithms via pytest: we can find the N slowest test calls using pytest with "–duration=N" option
2. Analyze the function via line_profiler: line_profiler can help analyze the function line by line
3. Improve the Function via Pythran
4. Compare the performance via line_profiler

Step 1: Identify slow algorithms via pytest

Pytest is a mature Python testing tool with features like unittest and duration profiling. Using "–duration=N" option, we can find out N slowest test execution calls. For example, run `pytest --durations=10` inside scipy/stats/, we can get the slowest 10 durations, as is shown below.

```
============================= slowest 10 durations =============================
17.36s call     scipy/stats/tests/test_stats.py::TestMGCStat::test_twosamp
17.15s call     scipy/stats/tests/test_continuous_basic.py::test_cont_basic[500-200-kstwo-arg57]
16.68s call     scipy/stats/tests/test_distributions.py::TestLevyStable::test_pdf_nolan_samples
16.09s call
scipy/stats/tests/test_stats.py::Test_ttest_ind_permutations::test_ttest_ind_randperm_alternative2
9.56s call     scipy/stats/tests/test_mstats_basic.py::TestCorr::test_kendall_p_exact_large
9.36s call     scipy/stats/tests/test_continuous_basic.py::test_moments[kstwo-arg57-True-True-False]
9.26s call     scipy/stats/tests/test_continuous_basic.py::test_moments[vonmises-arg100-False-False-
False]
6.98s call     scipy/stats/tests/test_continuous_basic.py::test_moments[vonmises_line-arg102-True-True-
False]
6.66s call     scipy/stats/tests/test_continuous_basic.py::test_moments[ksone-arg56-True-True-False]
5.66s call     scipy/stats/tests/test_continuous_basic.py::test_cont_basic[500-200-wrapcauchy-arg104]
```

In `test_stats.py`, the slowest test is `TestMGCStat::test_twosamp` and `test_ttest_ind_randperm_alternative2`, I'm still working on it (see [WIP] stats.multiscale_graphcorr and [WIP] test_ttest_ind_randperm_alternative2 analysis).

Therefore, for better illustration, I will use the 5th one, `test_kendall_p_exact_large` as an example for the following steps.

Step 2: Analyze the function via line_profiler

Line_profiler is a useful tool that can profile the time individual lines of code take to execute. After decorating the functions you want to profile with @profile, do `kernprof -v -l script_to_profile.py` and then you can see the line-by -line time analysis of the function.

The following figure shows the profiling for `test_kendall_p_exact_large`. As we can see, mstats_basic._kendall_p_exact spent almost 100% of the time.

```
Total time: 11.4932 s
File: lxy_test3.py
Function: test_kendall_p_exact_large at line 4

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     4                                           @profile
     5                                           def test_kendall_p_exact_large():
     6                                               # Test for the exact method with large samples (n >= 171)
     7                                               # expected values generated using SymPy
     8         2          7.0      3.5      0.0      expectations = {(400, 38965): 0.48444283672113314099,
     9         1          1.0      1.0      0.0                     (401, 39516): 0.66363159823474837662,
    10         1          1.0      1.0      0.0                     (800, 156772): 0.42265448483120932055,
    11         1          1.0      1.0      0.0                     (801, 157849): 0.53437553412194416236,
    12         1          0.0      0.0      0.0                     (1600, 637472): 0.84200727400323538419,
    13         1          1.0      1.0      0.0                     (1601, 630304): 0.34465255088058593946}
    14
    15         7         10.0      1.4      0.0      for nc, expected in expectations.items():
    16         6   11492472.0 1915412.0    100.0          res = mstats_basic._kendall_p_exact(nc[0], nc[1])
    17         6        709.0    118.2      0.0          assert_almost_equal(res, expected)
```

Let's look at the profiling of `stats_basic._kendall_p_exact_large` , the slowest part is the for-loop.

```
(scipydev) charlotte@CHARLOTLIU-MB0 tests % kernprof -v -l lxy_test3.py
Wrote profile results to lxy_test3.py.lprof
Timer unit: 1e-06 s

Total time: 11.5716 s
File: /Users/charlotte/Desktop/internship/gsoc/scipy/scipy/stats/mstats_basic.py
Function: _kendall_p_exact at line 535

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
   535                                           @profile
   536                                           def _kendall_p_exact(n, c):
   537                                               # Exact p-value, see Maurice G. Kendall, "Rank Correlation Methods" (4th Edition), Charles Griffin & Co., 1970.
   538         6          4.0      0.7      0.0      if n <= 0:
   539                                                   raise ValueError('n ({n}) must be positive')
   540         6          8.0      1.3      0.0      elif c < 0 or 4*c > n*(n-1):
   541                                                   raise ValueError(f'c ({c}) must satisfy 0 <= 4c <= n(n-1) = {n*(n-1)}.')
   542         6          3.0      0.5      0.0      elif n == 1:
   543                                                   prob = 1.0
   544         6          3.0      0.5      0.0      elif n == 2:
   545                                                   prob = 1.0
   546         6          4.0      0.7      0.0      elif c == 0:
   547                                                   prob = 2.0/np.math.factorial(n) if n < 171 else 0.0
   548         6          4.0      0.7      0.0      elif c == 1:
   549                                                   prob = 2.0/np.math.factorial(n-1) if n < 172 else 0.0
   550         6          5.0      0.8      0.0      elif 4*c == n*(n-1):
   551                                                   prob = 1.0
   552         6          4.0      0.7      0.0      elif n < 171:
   553                                                   new = np.zeros(c+1)
   554                                                   new[0:2] = 1.0
   555                                                   for j in range(3,n+1):
   556                                                       new = np.cumsum(new)
   557                                                       if j <= c:
   558                                                           new[j:] -= new[:c+1-j]
   559                                                   prob = 2.0*np.sum(new)/np.math.factorial(n)
   560                                               else:
   561         6        481.0     80.2      0.0          new = np.zeros(c+1)
   562         6         45.0      7.5      0.0          new[0:2] = 1.0
   563      5597       7671.0      1.4      0.1          for j in range(3, n+1):
   564      5591    8762608.0   1567.3     75.7              new = np.cumsum(new)/j
   565      5591       9105.0      1.6      0.1              if j <= c:
   566      5591    2790230.0    499.1     24.1                  new[j:] -= new[:c+1-j]
   567         6        940.0    156.7      0.0          prob = np.sum(new)
   568
   569         6        484.0     80.7      0.0      return np.clip(prob, 0, 1)
```

Step 3: Improve the Function via Pythran

The first idea came to me is accelerating it with Pythran. So I rewrite `_kendall_p_exact` as follows (I rewrite the f-string part and change np.math.factorial to math.factorial because Pythran does not support np.math).

Then do `pythran _my_kendall_p_exact.py` to generate the .so file.

```
#scipy/stats/_my_kendall_p_exact.py
import numpy as np
import math

#pythran export _kendall_p_exact(int,int)
def _kendall_p_exact(n, c):
   # Exact p-value, see Maurice G. Kendall, "Rank Correlation Methods" (4th Edition),
Charles Griffin & Co., 1970.
   if n <= 0:
       raise ValueError('n ({n}) must be positive')
   elif c < 0 or 4*c > n*(n-1):
       raise ValueError('c ({c}) must satisfy 0 <= 4c <= n(n-1) = '+ str(n*(n-1)))
   elif n == 1:
       prob = 1.0
   elif n == 2:
       prob = 1.0
   elif c == 0:
       prob = 2.0/math.factorial(n) if n < 171 else 0.0
   elif c == 1:
       prob = 2.0/math.factorial(n-1) if n < 172 else 0.0
   elif 4*c == n*(n-1):
       prob = 1.0
   elif n < 171:
       new = np.zeros(c+1)
       new[0:2] = 1.0
       for j in range(3,n+1):
           new = np.cumsum(new)
           if j <= c:
               new[j:] -= new[:c+1-j]
       prob = 2.0*np.sum(new)/math.factorial(n)
   else:
       new = np.zeros(c+1)
       new[0:2] = 1.0
       for j in range(3, n+1):
           new = np.cumsum(new)/j
           if j <= c:
               new[j:] -= new[:c+1-j]
       prob = np.sum(new)

   return np.clip(prob, 0, 1)
```

Step 4: Compare the performance via line_profiler

We can still use line_profiler to see whether the performance improved. As is shown below, the time of `_kendall_p_exact` now is 16295747, unfortunately, it is larger than before - 11492472.

One reason is that I rewrote `` `f'c ({c}) must satisfy 0 <= 4c <= n(n-1) = {n*(n-1)}.'` `` to `` `'c ({c}) must satisfy 0 <= 4c <= n(n-1) = '+ str(n*(n-1))` `` because Pythran requires f-strings with format specifier and I thought format specifier may be slower than using str() but it turns out not.
However, even if the original numpy version and the Pythran version use the same string, the Pythran version is still slower on my machine. I opened an issue in pythran, and the main author suggested that array copy (`new[j:] -= new[:c+1-j]`) optimization would be helpful but is not implemented in Pythran yet.

```
Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     5                                           @profile
     6                                           def test_kendall_p_exact_large():
     7                                               # Test for the exact method with large samples (n >= 171)
     8                                               # expected values generated using SymPy
     9         2          6.0      3.0      0.0      expectations = {(400, 38965): 0.48444283672113314099,
    10         1          1.0      1.0      0.0                     (401, 39516): 0.66363159823474837662,
    11         1          0.0      0.0      0.0                     (800, 156772): 0.42265448483120932055,
    12         1          1.0      1.0      0.0                     (801, 157849): 0.53437553412194416236,
    13         1          0.0      0.0      0.0                     (1600, 637472): 0.84200727400323538419,
    14         1          1.0      1.0      0.0                     (1601, 630304): 0.34465255088058593946}
    15
    16         7         13.0      1.9      0.0      for nc, expected in expectations.items():
    17                                                   # res = mstats_basic._kendall_p_exact(nc[0], nc[1])
    18         6   16295747.0 2715957.8    100.0          res = _kendall_p_exact(nc[0], nc[1])
    19         6        976.0    162.7      0.0          assert_almost_equal(res, expected)
```

Project Schedule

Community Bonding Period

May 17 - May 23

- Introduce myself and get to know more about my mentors, the organization, and other resources to look for help
- Actively take part in discussions regarding my project.

May 24 - May 30

- Investigate, discuss and decide the algorithms needed to be improved with my mentors
- Investigate, discuss and decide the benchmarks needed to be wrote with my mentors

May 31 - June 6

- Read the Pythran tutorial thoroughly.
  https://pythran.readthedocs.io/en/latest/MANUAL.html
- Study profiling Cython code
  https://cython.readthedocs.io/en/latest/src/tutorial/profiling_tutorial.html

Development Phase

June 7 - June 13

- Finish writing benchmarks for those scipy.stats functions (they don't have benchmarks currently)
  - all the inferential stats functions
  - many correlation functions
  - all the ANOVA functions

June 14 - June 20

- Finish writing benchmarks for the selected slow algorithms if they don't have benchmarks then.

June 21 - June 27

- Finish accelerating three algorithms:
  - `scipy.statsmultiscale_graphcorr`
  - `scipy.stats.ttest_ind`
  - `scipy.stats._kendall_p_exact`

June 28 - July 4

- Finish accelerating two algorithms:
  - `scipy.stats._compute_prob_inside_method`
  - `scipy._count_paths_outside_method`

July 5 - July 11

- Prepare for the first evaluations and initial demo.

July 12 - July 18

- Finish accelerating two algorithms:
  - `scipy.stats._cdf_distance`
  - `scipy.stats._moment`

July 19 - July 25

- Finish accelerating three algorithms:
  - `scipy.optimize.differential_evolution`
  - `scipy.optimize._nonmonotone_line_search_cruz`
  - `scipy.optimize._nonmonotone_line_search_cheng`

July 26 - August 1

- Finish accelerating two algorithms:
  - `scipy.sparse.random`
  - `scipy.ndimage.generic_filter`

Project Completion, testing, and documentation

August 2 - August 8

- Testing and debugging.
- Write documentation about writing and running benchmarks.

August 9 - August 15

- Prepare a demo and write a blog about this project

August 16 - August 23

- Discuss future prospects and developments with the mentors
- Help implement z-test (see the issue) if time permits

Acknowledgement

Many thanks to Ralf for helping me iterating this proposal and providing so many suggestions these days!!!

Reference

[1] https://jochenschroeder.com/blog/articles/DSP_with_Python2/
[2] https://pythran.readthedocs.io/en/latest/