Lab - 2 REPORT

CMPE 275

Under the supervision of

Prof. John Gash

Team Titans

Sandeep Birudukota - 016001167

Sriram Chinta - 016002506

Sai Koushik Pentyala - 016597789

CONTENTS

Design

In the context of software development, coupling refers to the level of reliance between two software application modules, classes, or objects.When there are several interdependencies between two modules, this is known as tight coupling. In contrast, loose coupling refers to situations when there are few relationships between two modules.

What technologies exist for language coupling?

SWIG-

A software development tool called SWIG links C and C++ programs with a number of high-level programming languages. SWIG is generally used to parse C/C++ interfaces and generate the 'glue code' required for the above target languages to call into the C/C++ code.

Pybind11-

It is a lightweight header-only library that presents C++ types in Python and vice- versa, primarily to enable the creation of Python bindings for C++ code that already exists. It has similar objectives and syntax as the famous Boost.Library for Python.

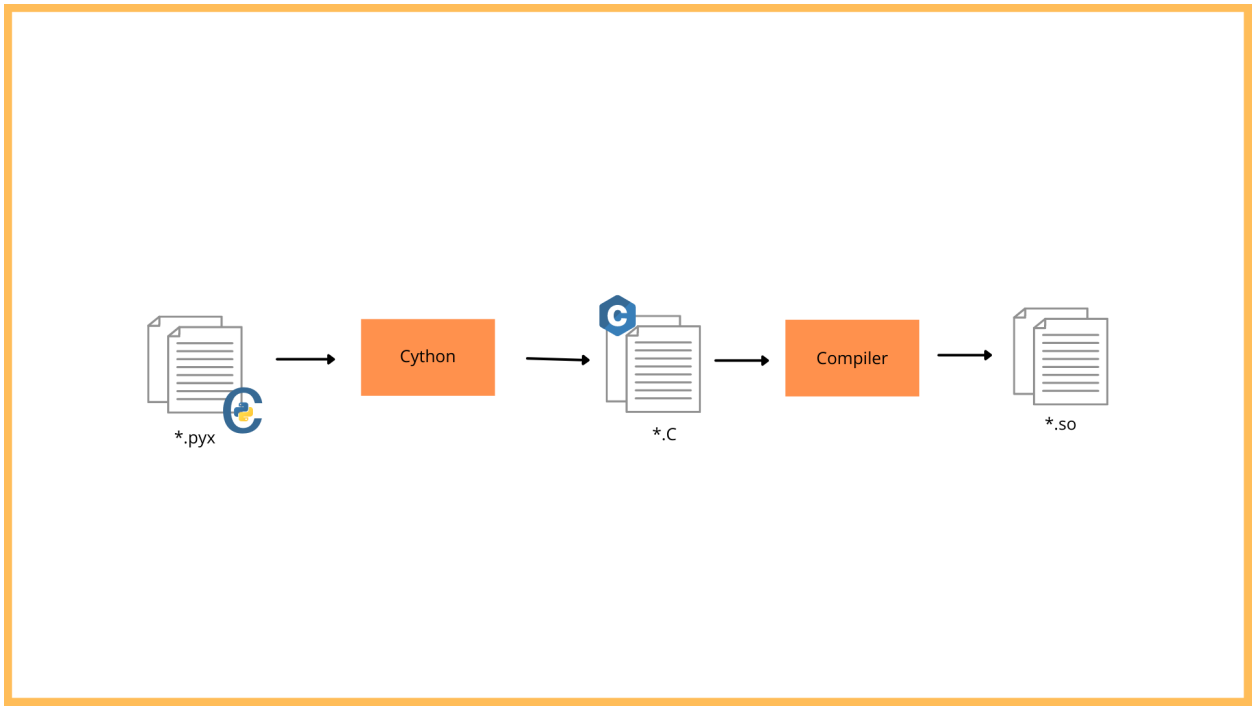
- Have a c++ code and port to python.
- Want to move object oriented code in python to c++ for performance.

- Want to synchronize backend code between python and c++

Cython -

The Cython language is a superset of Python. Cython compiles the python code to c, thus increasing the performance of the code for several magnitudes. The speedups won't be significant while using native object types in Python. However, the gains can be substantial for operations using numbers or any actions not involving Python's own internals. Without having to give up Python's simplicity and convenience, Cython allows you to get around or even go around all of its native limitations.

Architecture



What modern language couplings exist? Lua? Python?

A versatile technique exists in Lua for binding to programs created in other languages. Functions in libraries written in C can be made accessible to your Lua programs through the C application Programming interface.languages like C++ and Pascal are compatible with the C calling convention — these languages work fine with Lua. For python, the above section already presented the technologies useful for coupling.

How is it done?

We want to present how Cython works for the python code with C/C++. Python with C data types is basically Cython. Any cython code is valid in python code. But in cython we can declared c variables which can increase performance.

- A filename.pyx is created
- A setup.py file is created.

  ```
  from setuptools import setup
  from Cython.Build import cythonize

  setup(
      ext_modules = cythonize(" filename .pyx")
  )
  ```

- Now run which builds the cython file. The .pyx file is compiled by cython which gives a .c file.

  ```
  $ python setup.py build_ext --inplace
  ```

- A .c file is created and compiled by c which produces .so file.
- Now import this file simply by writing import to that .so file to our actual code.

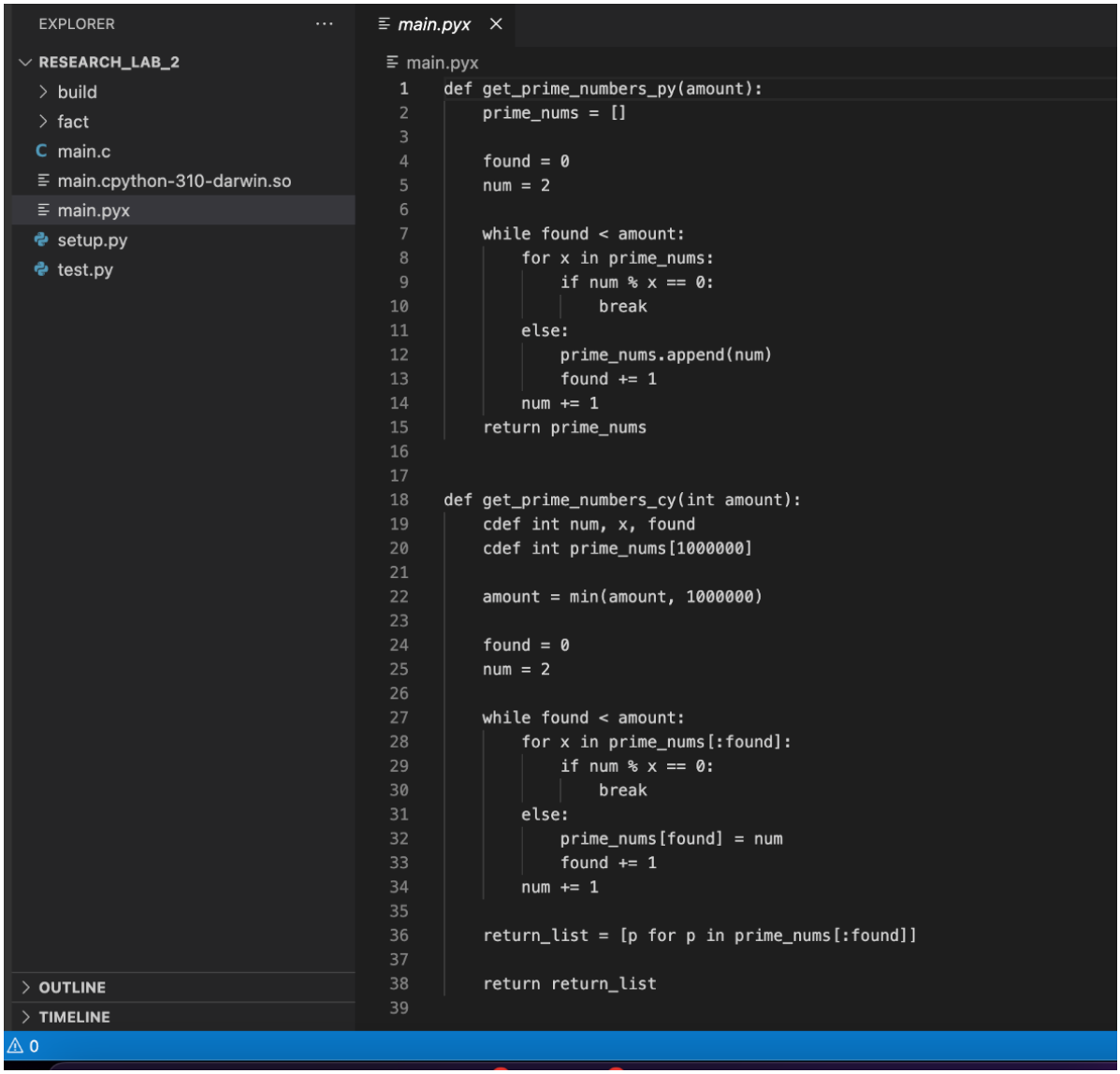
  ```
  Import filename
  ```

- Thus data types in c can be used in python for faster performance.

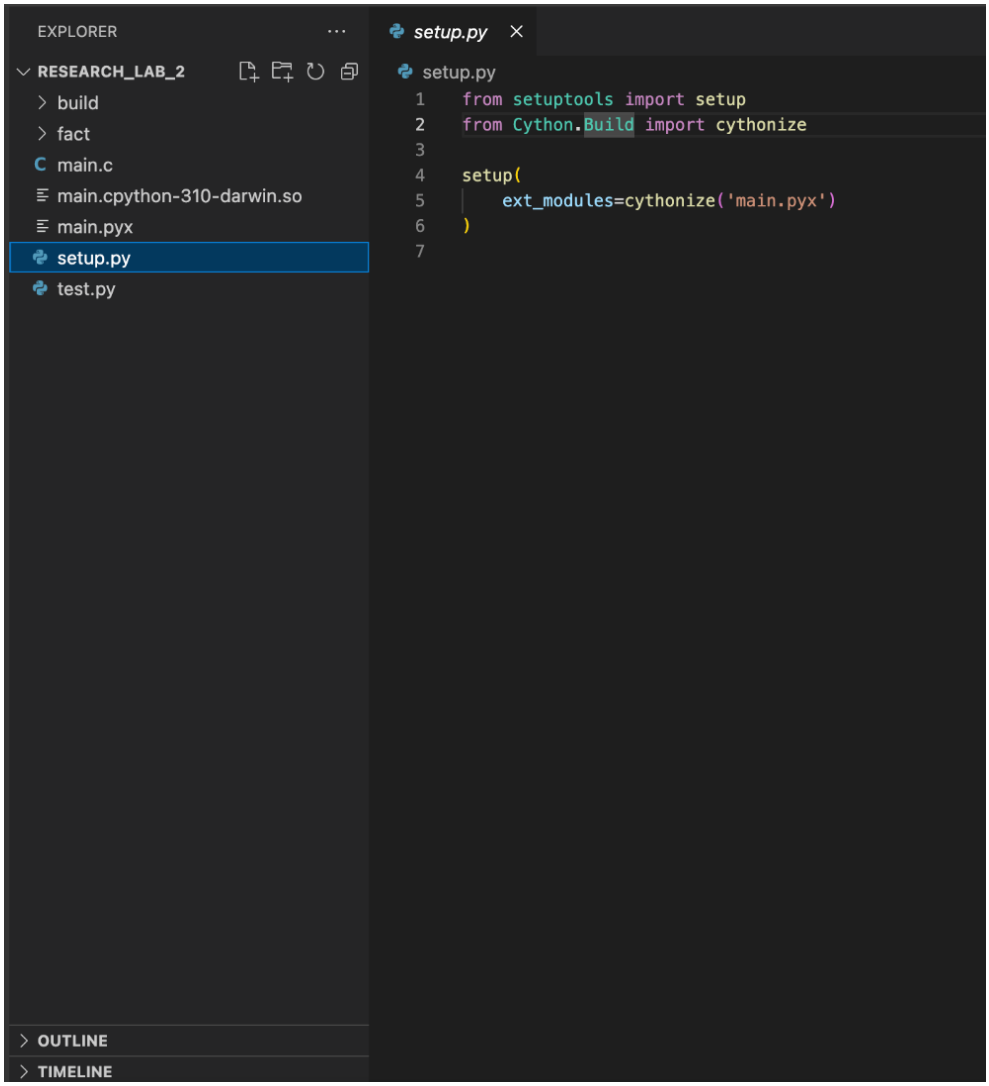
In this lab, we have used cython for coupling languages C/C++ and python.

- Pip Install cython
- Build main.pyx which has generic python function(prime numbers) by defining data types using c language data types.
- Create a setup.py which translates main.pyx file into c/c++ optimized code which generates main.c. this helps in faster execution of the code and also binds the c libraries. Cython compiles .pyx file to .c which contains the python module.
- C compiler compiles the .c file which produces the .so file. This is taken care of by setup.py which has setuptools.
- Imported the module into newly created test.py and call the functions we created.

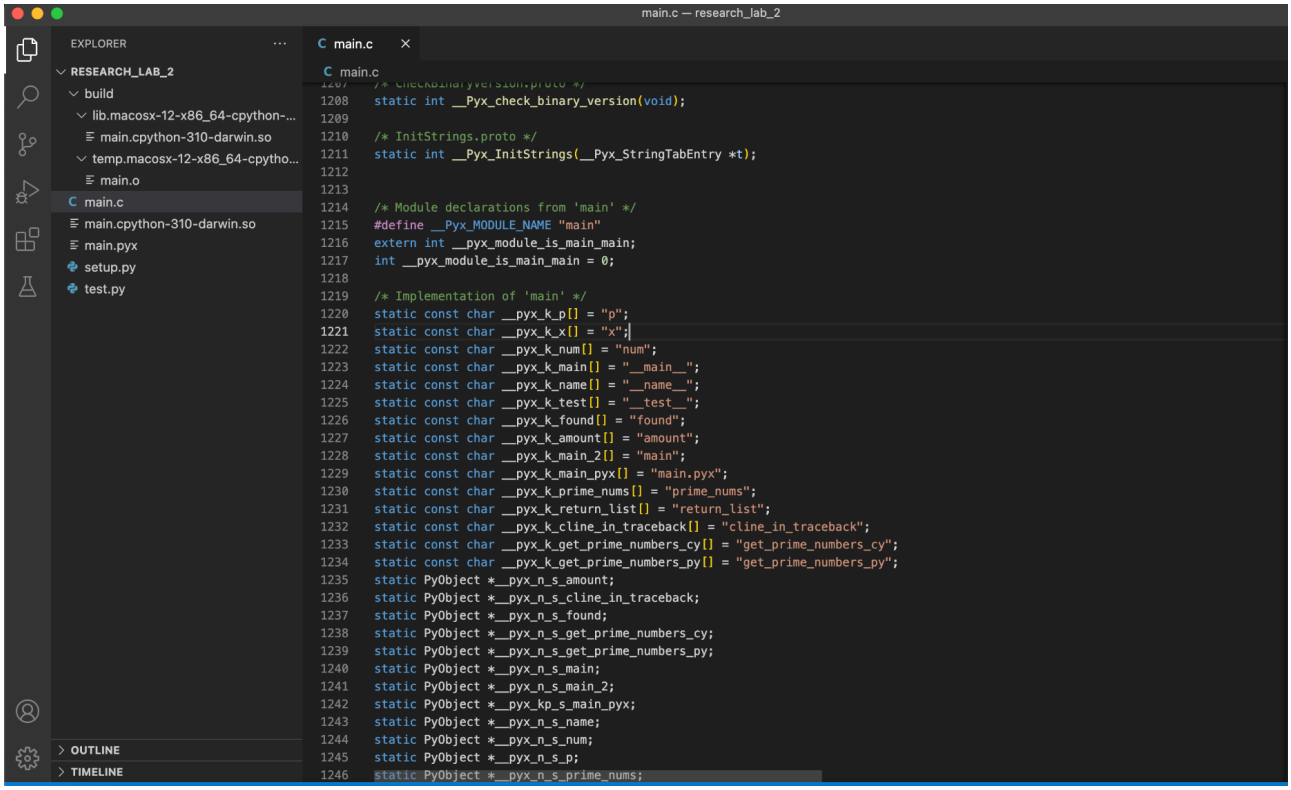
1. Create a main.pyx file

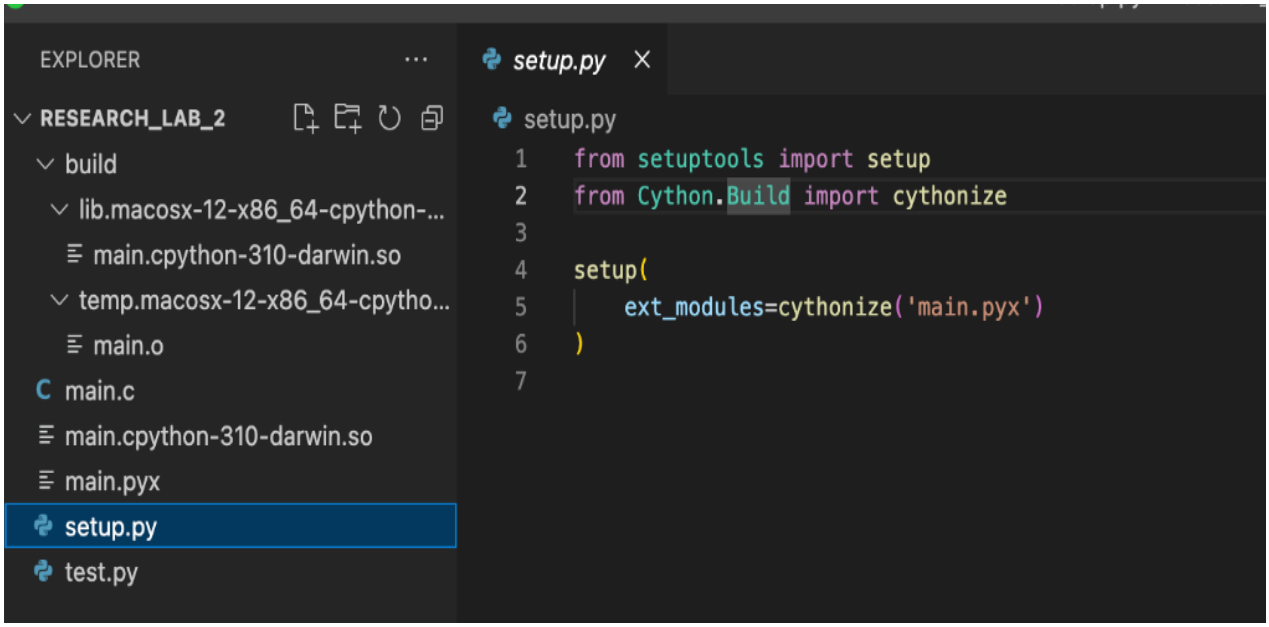


2. Create a setup.py file where we have setuptools which gives platform related compilation options and cythonize which helps the .pyx file in compiling the code to C/C++ .

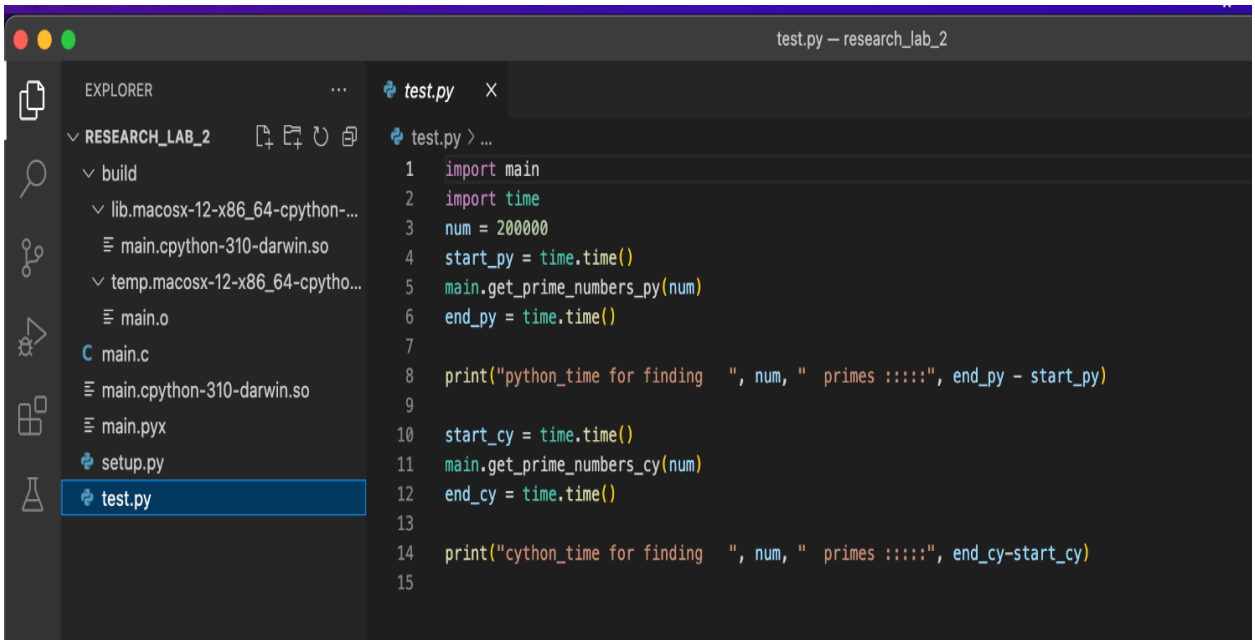


3. Run $ python setup.py build_ext --inplace
4. A .c file is generated and the c compiler compiles that .c file. A build directory which contains .o file is generated by the compiler. And a .so file which is a compiled library file.


main.c — research_lab_2
EXPLORER
RESEARCH_LAB_2
build
lib.macosx-12-x86_64-cpython-...
main.cpython-310-darwin.so
temp.macosx-12-x86_64-cpytho...
main.o
main.c
main.cpython-310-darwin.so
main.pyx
setup.py
test.py
OUTLINE
TIMELINE
main.c
1208 static int __Pyx_check_binary_version(void);
1209
1210 /* InitStrings.proto */
1211 static int __Pyx_InitStrings(__Pyx_StringTabEntry *t);
1212
1213
1214 /* Module declarations from 'main' */
1215 #define __Pyx_MODULE_NAME "main"
1216 extern int __pyx_module_is_main_main;
1217 int __pyx_module_is_main_main = 0;
1218
1219 /* Implementation of 'main' */
1220 static const char __pyx_k_p[] = "p";
1221 static const char __pyx_k_x[] = "x";
1222 static const char __pyx_k_num[] = "num";
1223 static const char __pyx_k_main[] = "__main__";
1224 static const char __pyx_k_name[] = "__name__";
1225 static const char __pyx_k_test[] = "__test__";
1226 static const char __pyx_k_found[] = "found";
1227 static const char __pyx_k_amount[] = "amount";
1228 static const char __pyx_k_main_2[] = "main";
1229 static const char __pyx_k_main_pyx[] = "main.pyx";
1230 static const char __pyx_k_prime_nums[] = "prime_nums";
1231 static const char __pyx_k_return_list[] = "return_list";
1232 static const char __pyx_k_cline_in_traceback[] = "cline_in_traceback";
1233 static const char __pyx_k_get_prime_numbers_cy[] = "get_prime_numbers_cy";
1234 static const char __pyx_k_get_prime_numbers_py[] = "get_prime_numbers_py";
1235 static PyObject *__pyx_n_s_amount;
1236 static PyObject *__pyx_n_s_cline_in_traceback;
1237 static PyObject *__pyx_n_s_found;
1238 static PyObject *__pyx_n_s_get_prime_numbers_cy;
1239 static PyObject *__pyx_n_s_get_prime_numbers_py;
1240 static PyObject *__pyx_n_s_main;
1241 static PyObject *__pyx_n_s_main_2;
1242 static PyObject *__pyx_kp_s_main_pyx;
1243 static PyObject *__pyx_n_s_name;
1244 static PyObject *__pyx_n_s_num;
1245 static PyObject *__pyx_n_s_p;
1246 static PyObject *__pyx_n_s_prime_nums;


EXPLORER
RESEARCH_LAB_2
build
lib.macosx-12-x86_64-cpython-...
main.cpython-310-darwin.so
temp.macosx-12-x86_64-cpytho...
main.o
main.c
main.cpython-310-darwin.so
main.pyx
setup.py
test.py
setup.py
setup.py
1 from setuptools import setup
2 from Cython.Build import cythonize
3
4 setup(
5 ext_modules=cythonize('main.pyx')
6)
7

5. Import the Cython file just like the python module.

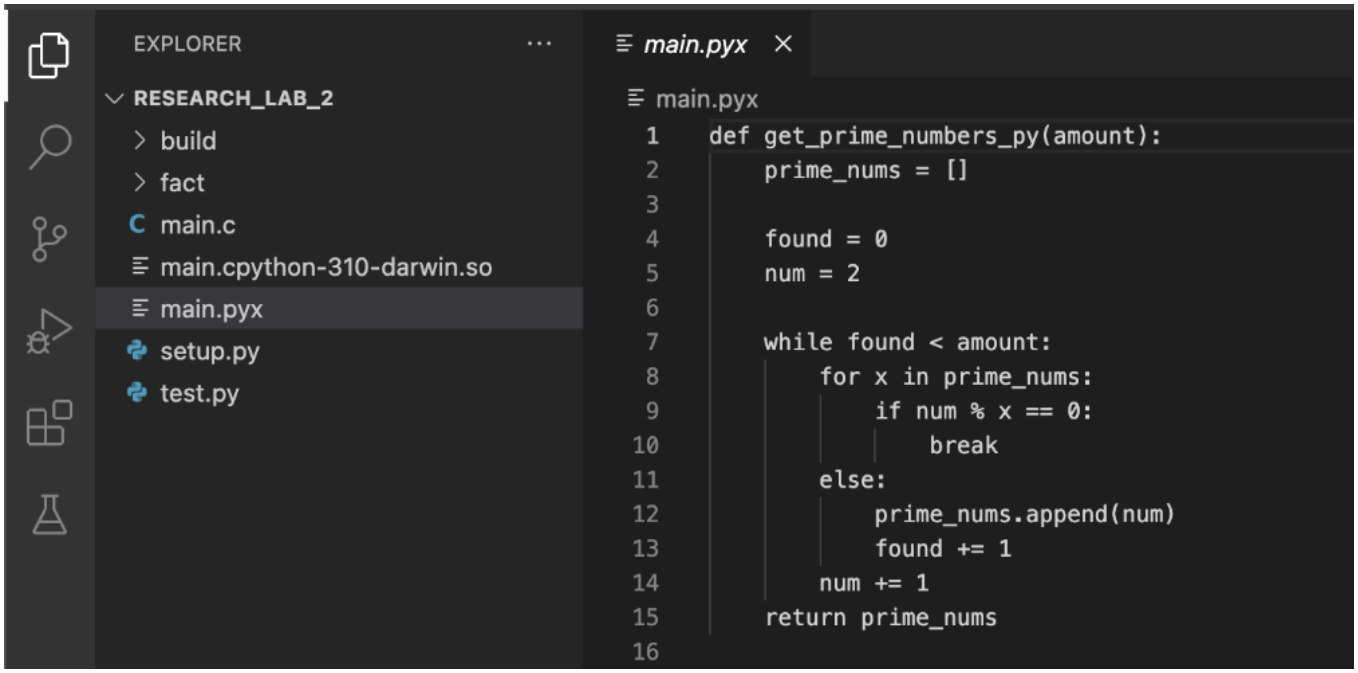


6. Now run the python file.

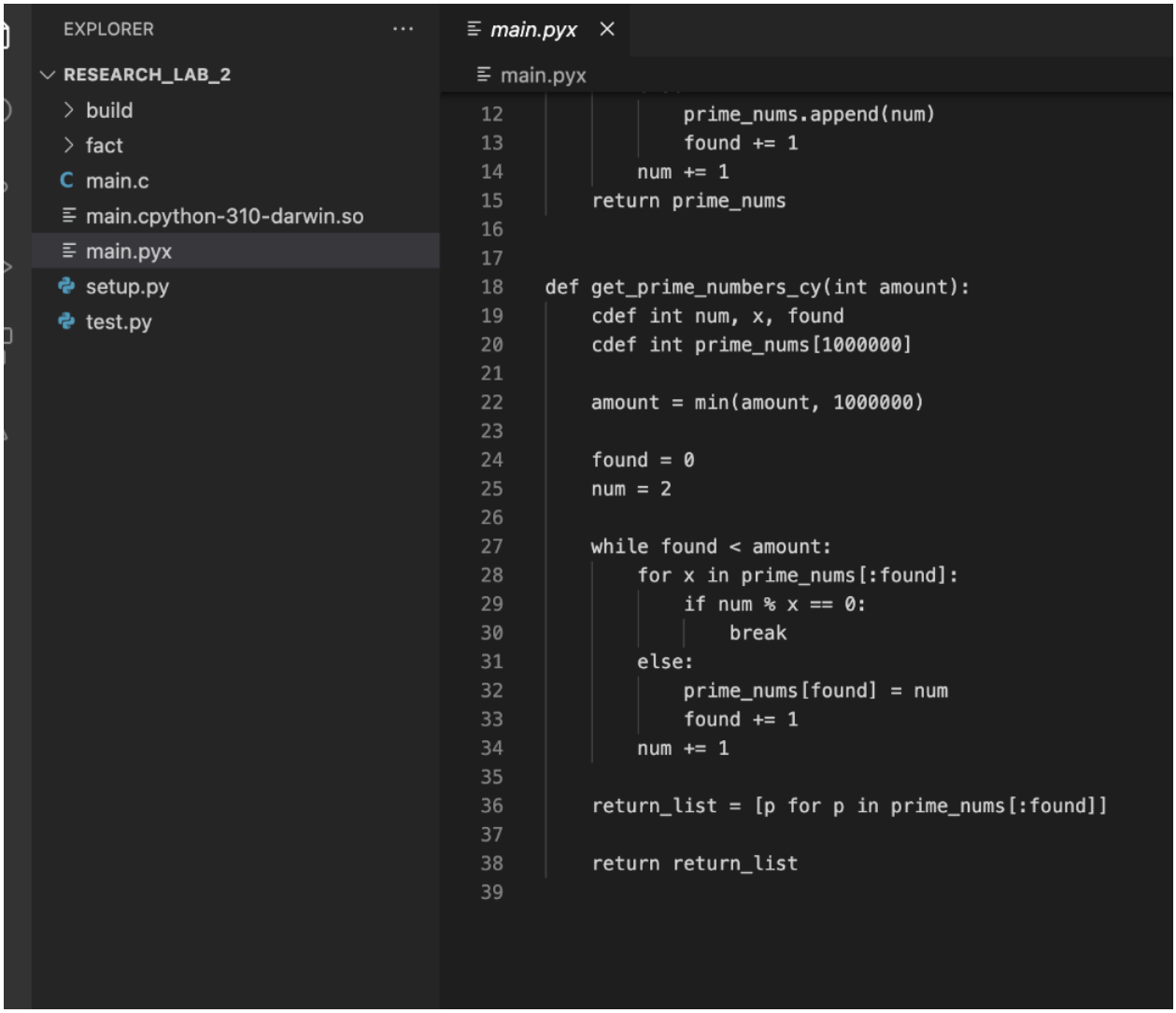
Results

We have written both pure python and cython codes in the main.pyx file inorder to compare the results for both code performances.

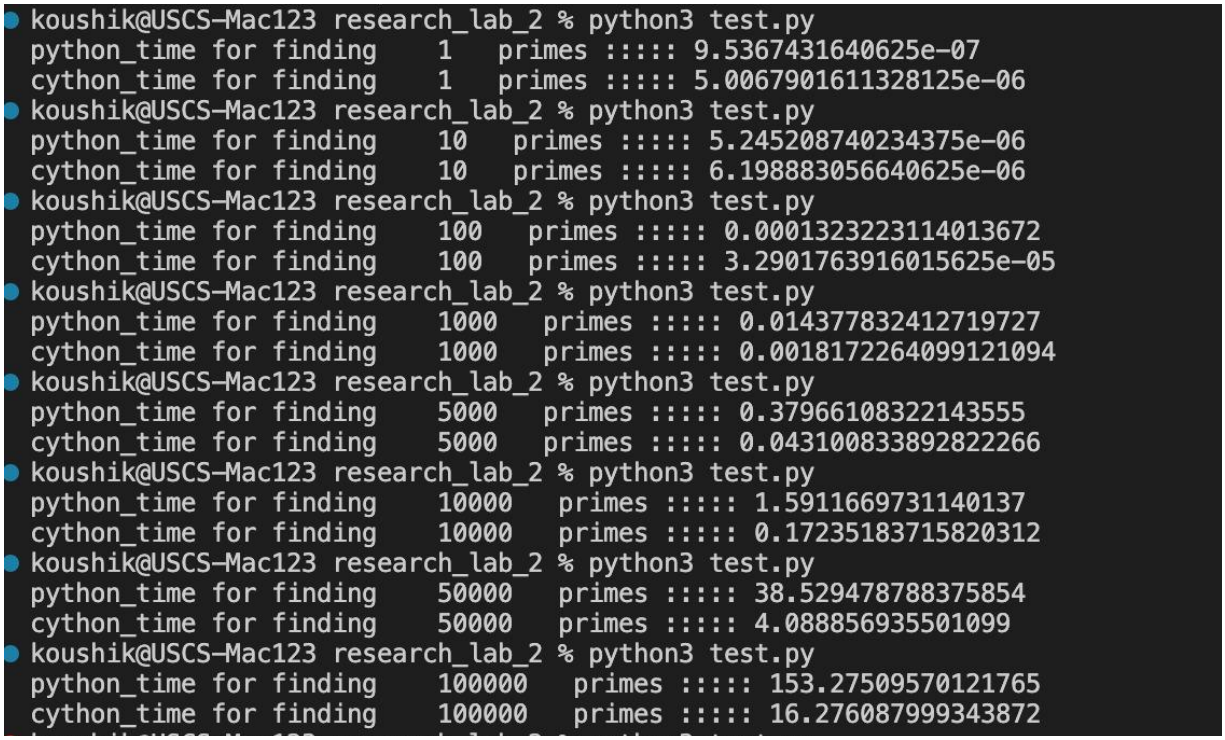
Pure python code -



Cython code-

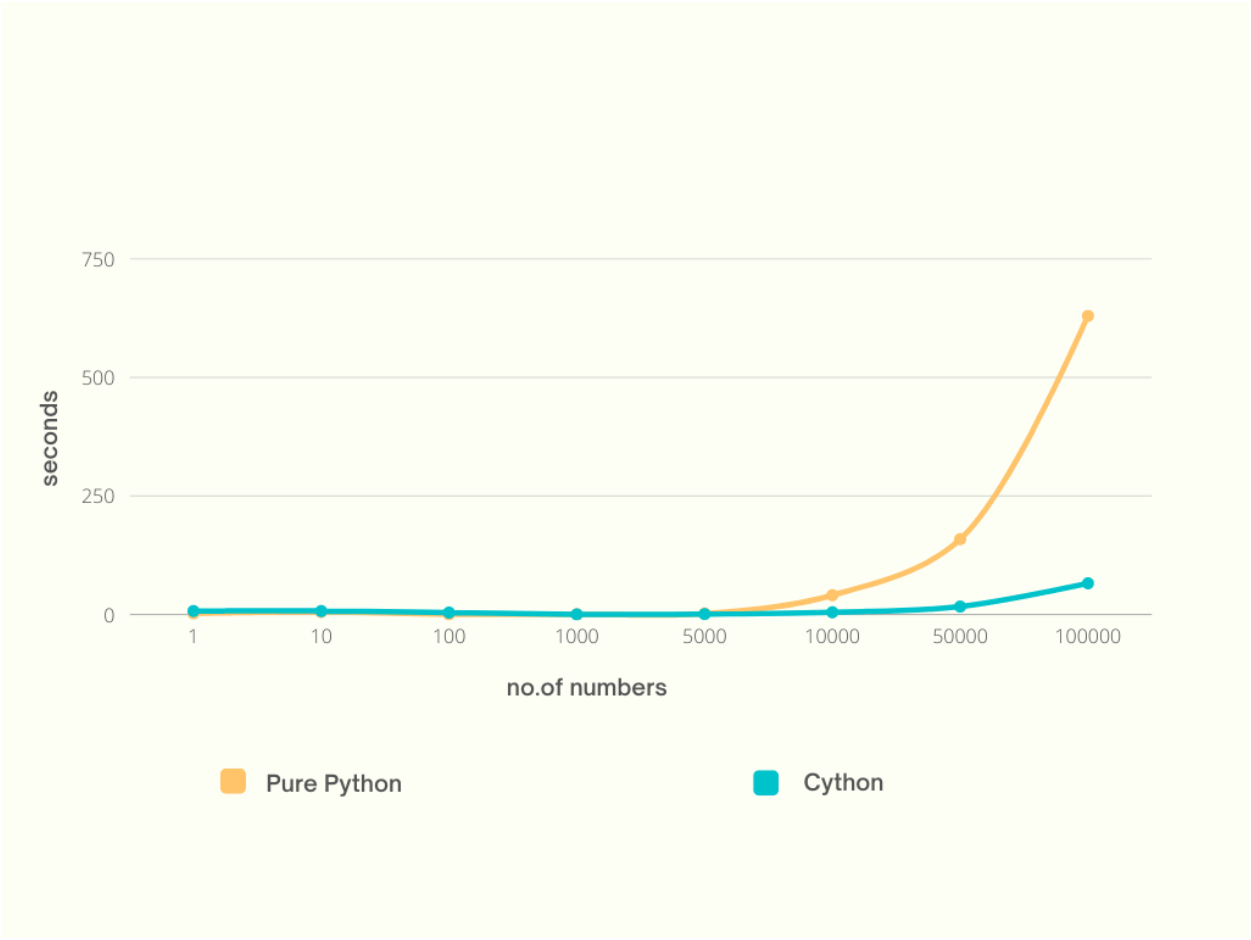

EXPLORER
RESEARCH_LAB_2
build
fact
main.c
main.cpython-310-darwin.so
main.pyx
setup.py
test.py
main.pyx
main.pyx
prime_nums.append(num)
found += 1
num += 1
return prime_nums
def get_prime_numbers_cy(int amount):
cdef int num, x, found
cdef int prime_nums[1000000]
amount = min(amount, 1000000)
found = 0
num = 2
while found < amount:
for x in prime_nums[:found]:
if num % x == 0:
break
else:
prime_nums[found] = num
found += 1
num += 1
return_list = [p for p in prime_nums[:found]]
return return_list

Output-

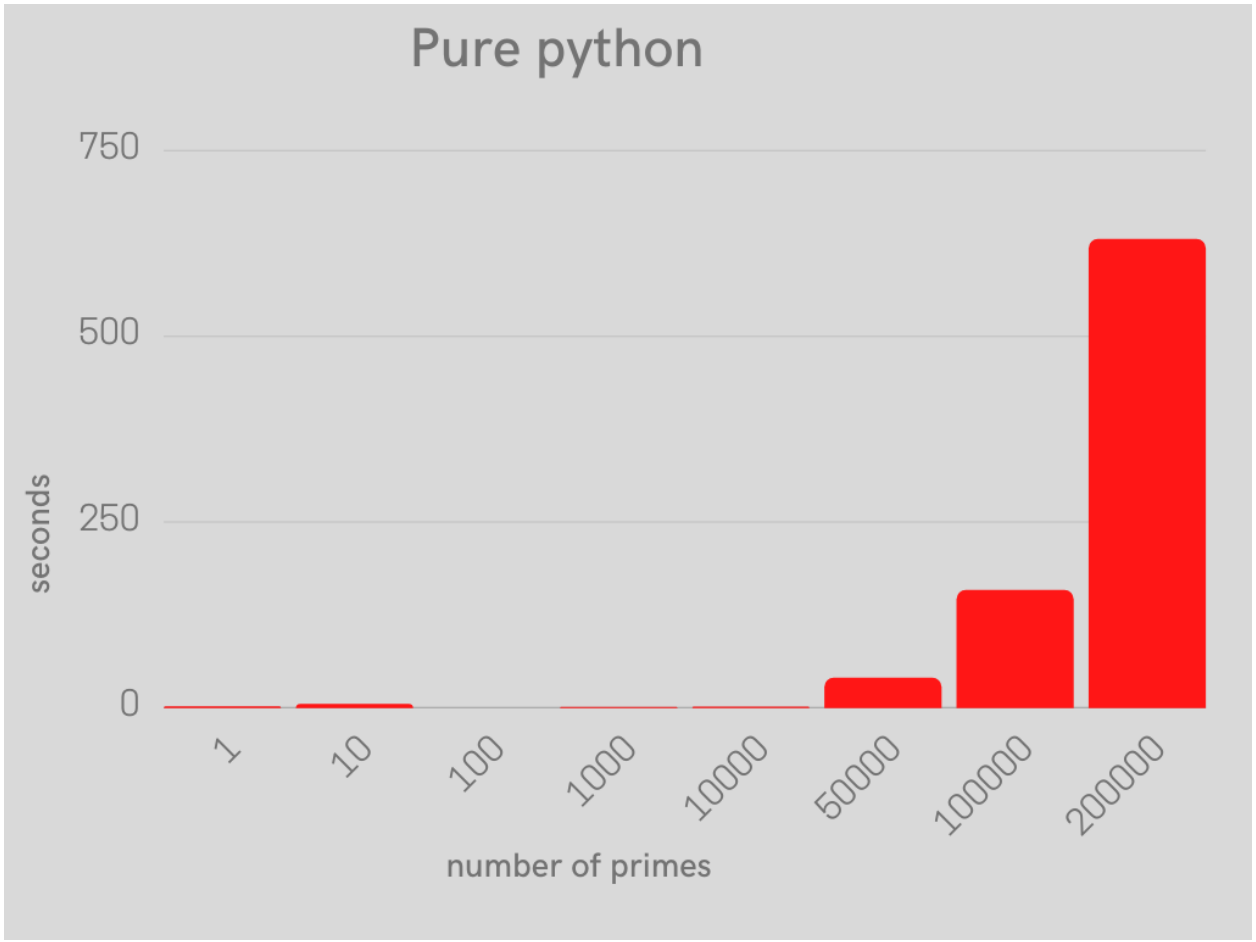


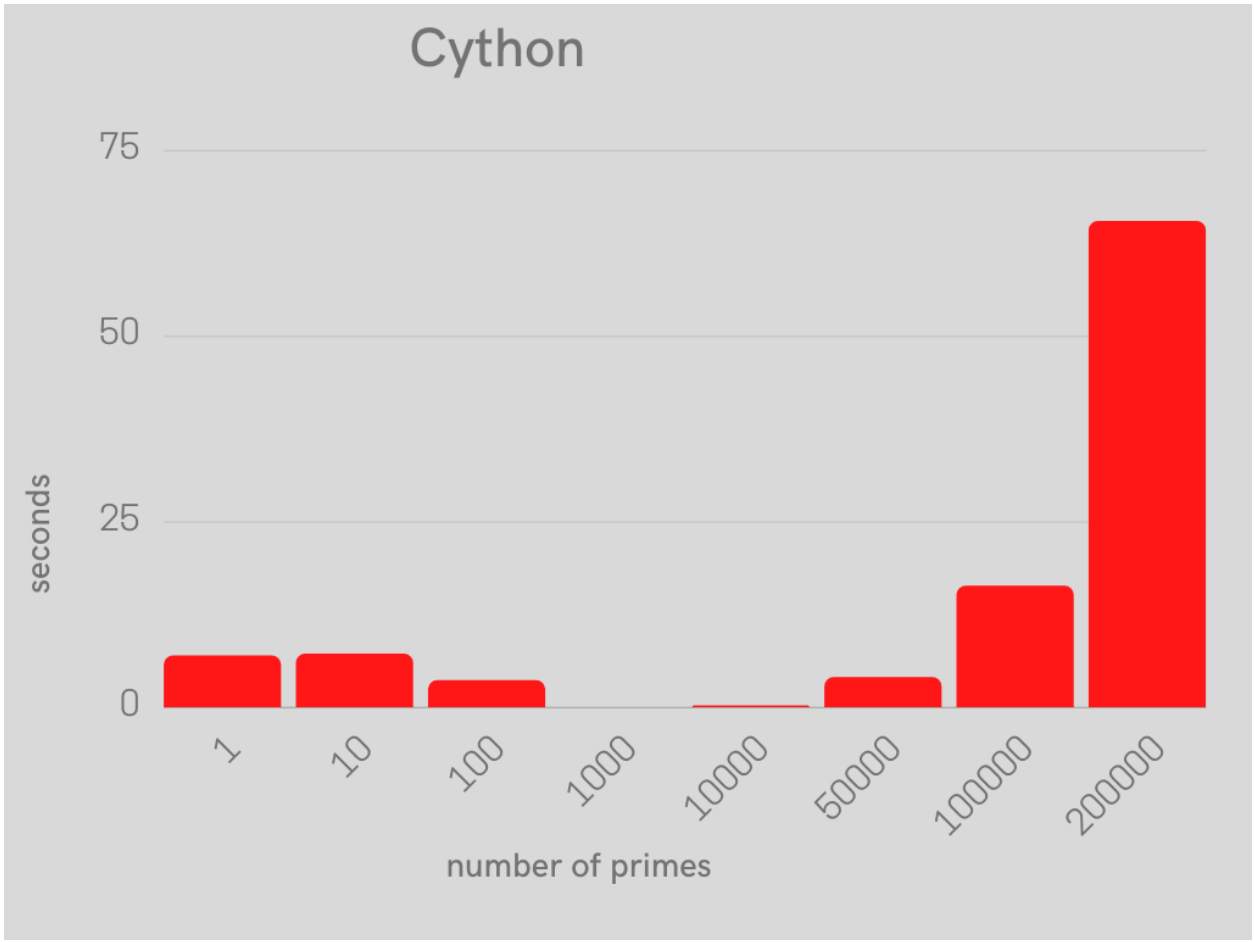
At first python code performance is better when compared to cython code. But eventually, when the number of for loops increased, the performance for the cython code was increased as it utilized cdef to declare variables in cython for c data types.

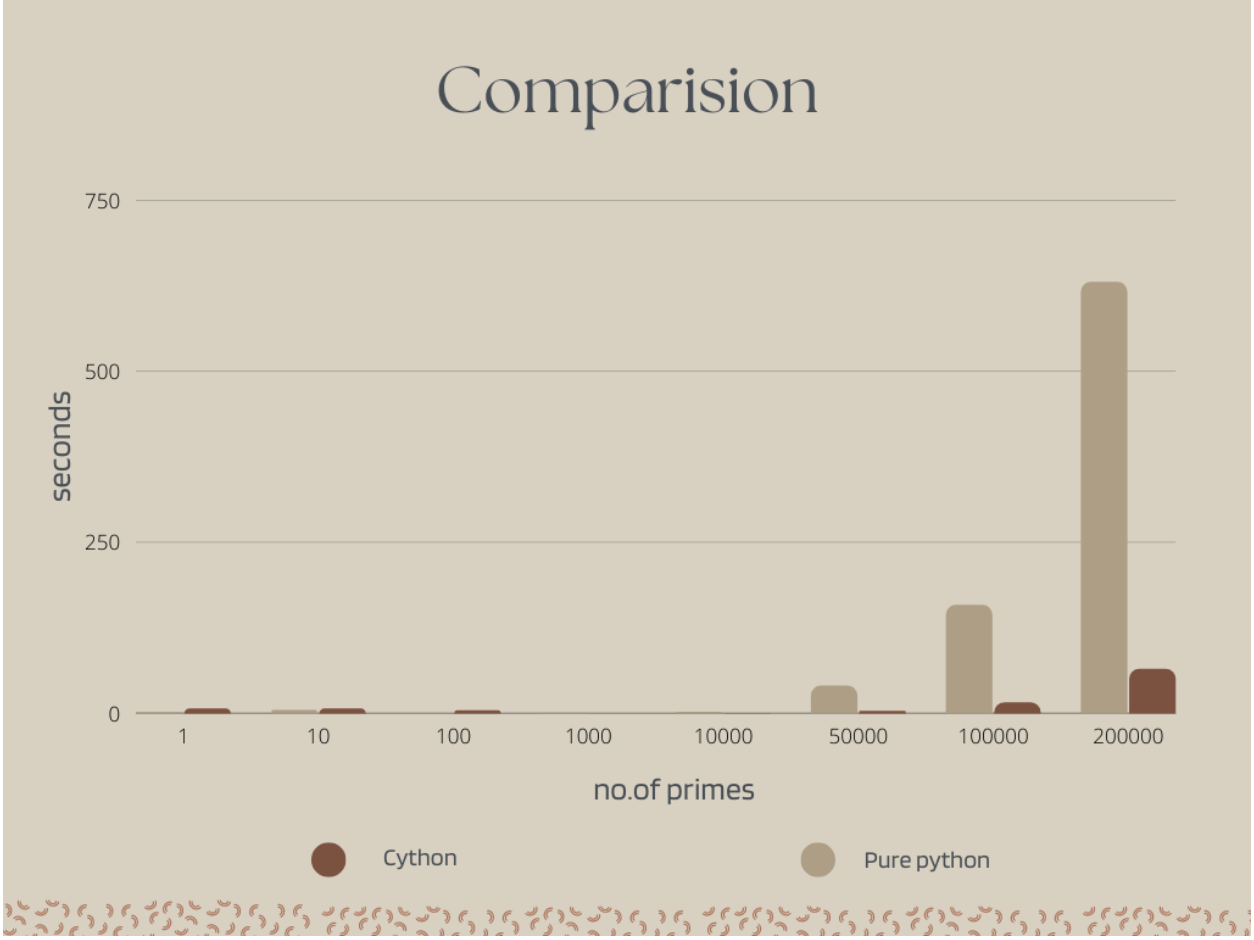
Graphs


750
500
250
0
seconds
1
10
100
1000
5000
10000
50000
100000
no.of numbers
Pure Python
Cython

```
main.c:353:34: note: expanded from macro 'CYTHON_FALLTHROUGH'
koushik@USCS-Mac123 research_lab_2 % python3 test.py
python_time for finding    1   primes ::::: 1.9073486328125e-06
cython_time for finding    1   primes ::::: 6.9141387939453125e-06
koushik@USCS-Mac123 research_lab_2 % python3 test.py
python_time for finding    10   primes ::::: 5.0067901611328125e-06
cython_time for finding    10   primes ::::: 7.152557373046875e-06
koushik@USCS-Mac123 research_lab_2 % python3 test.py
python_time for finding    100   primes ::::: 0.00013518333435058594
cython_time for finding    100   primes ::::: 3.62396240234375e-05
koushik@USCS-Mac123 research_lab_2 % python3 test.py
python_time for finding    1000   primes ::::: 0.014393091201782227
cython_time for finding    1000   primes ::::: 0.002371072769165039
koushik@USCS-Mac123 research_lab_2 % python3 test.py
python_time for finding    10000   primes ::::: 1.5317418575286865
cython_time for finding    10000   primes ::::: 0.17069125175476074
koushik@USCS-Mac123 research_lab_2 % python3 test.py
python_time for finding    50000   primes ::::: 40.20154404640198
cython_time for finding    50000   primes ::::: 4.090368032455444
koushik@USCS-Mac123 research_lab_2 % python3 test.py
python_time for finding    100000   primes ::::: 158.30770683288574
cython_time for finding    100000   primes ::::: 16.31757688522339
koushik@USCS-Mac123 research_lab_2 % python3 test.py
python_time for finding    200000   primes ::::: 630.2040741443634
cython_time for finding    200000   primes ::::: 65.44018483161926
koushik@USCS-Mac123 research_lab_2 %
```


Pure python
750
500
250
0
seconds
1
10
100
1000
10000
50000
100000
200000
number of primes


Cython
75
50
25
0
seconds
1
10
100
1000
10000
50000
100000
200000
number of primes



Github Link-

https://github.com/sandeepbirudukota/lab2binding275

Supporting reasons

Citations- https://cython.org

Contributions

Sriram- Studied Research papers, Done research on SWIG and Cython, written code for python in main.pyx, done ppt slides and graphs.

Sandeep- Studied Research papers, Done research on Pybind11 and Cython, written code for cython in main.pyx, done documentation and graphs.

Koushik- Studied Research papers, Done research on Cython, executed code and retrieved different results, done documentation and graphs.