Cython

Naoya Kawakami

August 8, 2024

Contents

Chapter 1 Accelerate python code with Cython 1

1.1 Introduction 1

1.2 Python code 1

1.3 Using Cython 3

1.4 Type defenition 4

1.5 Cython with cimport 5

1.6 Comparison of each code 6

Chapter 2 Codes 7

2.1 Python 7

2.2 Just use cythone 10

2.3 Type defined cython code 11

2.4 C import cython code 14

Chapter 1

Accelerate python code with Cython

1.1 Introduction

Python is one of the most widely used computing languages because of its simple coding and various useful libraries. However, the execution speed of Python is slow compared with other languages. The slow execution speed can be problematic when performing a long-time simulation with many cycles of iterations. One of the solutions to improve the speed of Python is to use Cython. In Cython, the Python code is automatically translated into C (or C++), resulting in a faster speed, while Cython still shares similar grammar as Python. In this brief review, I will introduce the process of converting Python code into Cython with a simple example. All the codes are shown in Chapter 2.

1.2 Python code

Firstly, I prepared codes for simulating ”random walk.” In this program, a particle randomly moves around in a square lattice under the periodic boundary condition. The position of the particle is recorded at each step, and the number of occupations at each lattice site is output as a color map. I coded the program a little complicated way, while the simulation itself is very simple, to sufficiently explain how to convert Python to Cython. Here, I just briefly introduce each code. If you need a detailed explanation of the coding of Python, refer to the other textbook.

1.2.1 Python_program.py

Python_program.py (2.1) is used to start the calculation. This code will not be changed anymore (even after converting to Cython).

1.2.2 main_functions.py

The components for conducting the simulation are defined in the class ”functions” in main_function.py (2.2). The class ”function” is inherited in class ”random_walk” in Python_program.py.

1.2.3 sub_function.py

In sub_function.py (2.3), several additional functions are defined. The functions are imported and used in main_functions.py.

1.2.4 recording.py

A function for recording the simulation result is defined in recording.py (2.4). The defined function (record_data) is imported and used in main_functions.py.

1.2.5 InputParameter.py

A class for reading the simulation parameter is defined in InputParameter.py (2.5).

1.2.6 input.yml

This is not a Python file but a Yaml (.yml) file. Yaml form is often used to describe serialized data. Using the Yaml format, even a large number of parameters can be easily incorporated into the program as written in InputParameter.py.

”cell_size_xy” defines the number of lattice sites in one direction. ”probabilities” defines the probability of hopping in each direction. ”total_steps” determines the number of iterations. The progress of simulation is output at each ”print_per” step. This file will not be changed anymore.

1.2.7 Calculation speed

It takes about 2 min 34 sec to run this program. Next, we will see how this program becomes fast by using Cython.

1.3 Using Cython

The first step in converting Python codes to Cython is to change the name of the file. Change the extension of the files from ".py" to ".pyx", except for the python_program.py and input.yml. It is not necessary to rewrite the code inside. Then, prepare a file "setup.py", as shown in 2.7. The setup.py file is used to convert the .pyx file to a C code (.c) and then make an execution file from the C code. Run the setup file by using a command

```
python setup.py build_ext inplace
```

Several files will be formed.*[1] When running "python_program.py", the execution files are automatically used.

1.3.1 Calculation speed

Try running "python_program.py". The calculation speed is about 2 min 22 sec. So even without changing the code, the calculation speed becomes ten % faster than the normal Python code.

*[1] Additional note on 7th/Aug./2024: A error ïo.h :No such file or directory erroröccured, which was solved by installing Windows SDK.

1.4 Type defenition

For further speed up, we should declare the type of all the variables in the Cython code. The codes are shown in Chapter 3.3. The revised and additional parts from the original Python code are highlighted in red. The following revisions are made for all the .pyx codes.

- For the class object, change ”def” to ”cdef”.
- For the function, change ”def” to ”cdef” or ”cpdef”. The function defined by ”cdef” is the fastest, but it cannot be called from a Python file. ”cpdef” can be called from both Python and Cython codes. If you cannot judge how the functions are called, using ”cpdef” is the safest choice.
- Describe all the types of variables used in each function. For class objects, the self variables should be declared just after defining the class. The local variables should be declared in the function. The declaration style is as follows:

  ```
  cdef [type] [variable name]
  ```

  For example, if variables ”a” and ”b” is used for storing integers (such as a = 1, b= 2),

  ```
  cdef int a, b
  ```

  should be written. Note that the ”float” objects in Python are declared by ”double” in Cython. Do not forget to declare the type of iteration variables (typically i in for loop).
- If the function returns an object, declare the type of the return just after ”cdef (or cpdef)”. Therefore, the function with return should be written in the format as follows:

  ```
  cdef [type of return object] [function name]():
  ...
  ```

  If a function with the name of ”test_function” returns an integer ”int_a”, it will be

  ```
  cdef int test_function():
  cdef int int_a
      int_a = 1
  return int_a
  ```
- If the function takes arguments, declare the type of the arguments before the parameters.

  ```
  cdef [return type] [function name]([type] [parameter name]):
  ```

  If the previous ”test_function” takes two integer arguments (arg_a and arg_b), the function should be written as follows:

  ```
  cdef int test_function(int arg_a, int arg_b):
      cdef int int_a
      int_a = arg_a + arg_b
  return int_a
  ```

In addition,

```
# cython: language_level=3, boundscheck=False, wraparound=False
```

are added at the beginning of all the pyx files. The ”language_lebel” declare that the code is written based on Python 3. The ”boundscheck=False, wraparound=False” disables some functions for lists in Python. In Python code, the final object in a list can be read by list[-1], but this format will not be used if declared. Instead, this declaration contributes about ten % speed up.

1.4.1 Calculation speed

After the above revisions, do not forget to run ”setup.py” again. The calculation speed is 36 sec, four times faster than the original Python code.

1.5 Cython with cimport

We will change the import method of the functions. By using ”cimport” instead of ”import”, the program can know the type of arguments and returns of the imported functions in advance, which contribute to the faster calculation time. For using ”cimport”, we should prepare ”.pxd” files for all the ”.pxy” files. In the .pxd files, all the types of self variables (for class) and functions are declared. The functions declared in .pxd can be imported by the ”cimport” method. See Code 2.12 - 2.17 for the example of the .pxd files.

For including the pxd file in translation, the setup.py file should be revised. See Code 2.18.

1.5.1 Calculation speed

After the revision by running setup.py, we can try the calculation. The calculation time is 29 sec, which is five times faster than the original Python code.

1.6 Comparison of each code

Table 1.1: Calculation time

Method	time	/Python
Python	2:34	1
Cython	2:22	0.92
Cython (Type defined)	0:36	0.23
Cython (cimport)	0:29	0.19

Finally, Cython code achieved five times faster speed compared with normal Python. Further improvement might be possible by replacing Python functions with corresponding C (or C++) functions, but it requires significant modifications to the code because some of the useful functions in Python will not be available. Considering the balance between the benefit of speed up and time for revision, the c import Cython would be the efficient choice.

Chapter 2

Codes

The change of the code are highlited by red. The aditional comments (not nessesary in code) are highlighted by blue.

2.1 Python

Listing 2.1: Python_ program.py

```python
from main_function import functions

class random_walk(functions):
    def __init__(self):
        functions.__init__(self)

    def end(self):
        print("Recording")
        self.end_of_loop()

    def start(self):
        print("Calculation␣start")
        self.start_setting()
        print("Loop␣start")
        self.loop()
        self.end()

if __name__ == "__main__":
    rw_class = random_walk()
    rw_class.start()
```

Listing 2.2: main_function.py

```python
from InputParameter import Params
import time
from sub_functions import pos_rec_lattice, choice_event
from recording import record_data
import os
import math

class functions:
    def __init__(self):
        self.init_value = Params("input.yml")
        self.cell_size = self.init_value.cell_size_xy
        self.count = 0
        self.output_count = self.init_value.print_per
        self.dir_rec = [0, 0, 0, 0, 0]

    def start_setting(self):
        self.direction = [0, 0]
        self.current_position = [int(self.cell_size / 2), int(self.cell_size / 2)]
        if os.path.exists("Record") is False:
            os.mkdir("Record")
        self.pos_rec = pos_rec_lattice(self.cell_size)
        self.probability_set()
        self.start_time = time.time()

    def probability_set(self):
        self.up = self.init_value.probabilities["up"]
        self.right = self.init_value.probabilities["right"]
        self.down = self.init_value.probabilities["down"]
        self.left = self.init_value.probabilities["left"]
        total = self.up + self.down + self.left + self.right
        self.up = self.up / total
        self.left = self.left / total + self.up
        self.down = self.down / total + self.left
        self.right = self.right / total + self.down

    def loop(self):
        for _ in range(self.init_value.total_steps):
            self.function_loop()

    def function_loop(self):
        self.direction = choice_event(self.up, self.left, self.down)
        self.direction_record()
        self.update_events()
        self.record_position()
        self.count += 1
        if self.count == self.output_count:
            percentage = self.count / self.init_value.total_steps * 100
            print(f"Progress:␣{percentage:.2f}␣%")
            self.output_count += self.init_value.print_per

    def direction_record(self):
        if self.direction == [0, -1]:
            self.dir_rec[0] += 1
        elif self.direction == [-1, 0]:
            self.dir_rec[1] += 1
        elif self.direction == [0, 1]:
            self.dir_rec[2] += 1
        elif self.direction == [1, 0]:
            self.dir_rec[3] += 1
        else:
            self.dir_rec[4] += 1

    def update_events(self):
        self.current_position[0] = self.boundary_check(
            self.current_position[0] + self.direction[0]
        )
        self.current_position[1] = self.boundary_check(
            self.current_position[1] + self.direction[1]
        )

    def boundary_check(self, val):
        if val < 0:
            return self.cell_size - 1
        elif val >= self.cell_size:
            return 0
        else:
            return val

    def record_position(self):
        self.pos_rec[self.current_position[0]][self.current_position[1]] += 1

    def end_of_loop(self):
        self.elapsed_time = time.time() - self.start_time
        self.minute = math.floor(self.elapsed_time / 60)
        self.second = int(self.elapsed_time % 60)
        record_data(self.pos_rec, self.minute, self.second)
        print(self.dir_rec)
        print("Finished:␣" + str(self.minute) + "␣min␣" + str(self.second) + "␣sec")
```

Listing 2.3: sub_function.py

```python
import random

def pos_rec_lattice(cell_size):
    pos_rec = [[0 for _ in range(cell_size)] for _ in range(cell_size)]
    return pos_rec

def choice_event(up, left, down):
    random_val = random.random()
    if random_val <= up:
        return [0, -1]
    elif random_val <= left:
        return [-1, 0]
    elif random_val <= down:
        return [0, 1]
    else:
        return [1, 0]
```

Listing 2.4: recording.py

```python
import seaborn as sns
import matplotlib.pyplot as plt

def record_data(pos_rec, minute, second):
    plt.figure()
    sns.heatmap(pos_rec)
    plt.savefig("Record/heat_map.png")
    #
    file_data = open("Record/execution_time.txt", "a")
    file_data.write(str(minute) + "␣min␣" + str(second) + "␣sec" "\n")
    file_data.close()
```

Listing 2.5: InputParameter.py

```python
import yaml

class Params:
    def __init__(self, filename):
        if filename:
            with open(filename) as f:
                input_yaml = yaml.safe_load(f)
            for k, v in input_yaml.items():
                setattr(self, k, v)
```

Listing 2.6: input.yml

```yaml
cell_size_xy: 100
probabilities:
  up: 1.0
  left: 1.0
  down: 1.0
  right: 1.0
total_steps: 100000000
print_per: 10000000
```

2.2 Just use cythone

Listing 2.7: setup.py

```python
from setuptools import setup
from Cython.Build import cythonize
import Cython.Compiler.Options

Cython.Compiler.Options.annotate = True

setup(ext_modules=cythonize("sub_functions.pyx"), annotate=True)
setup(ext_modules=cythonize("recording.pyx"), annotate=True)
setup(ext_modules=cythonize("InputParameter.pyx"), annotate=True)
setup(ext_modules=cythonize("main_function.pyx"), annotate=True)
```

2.3 Type defined cython code

Listing 2.8: main_function.pyx

```
# cython: language_level=3, boundscheck=False, wraparound=False

from InputParameter import Params
import time
from sub_functions import pos_rec_lattice, choice_event
from recording import record_data
import os
import math


cdef class functions:
    cdef int cell_size, count, output_count, minute, second
    cdef double start_time, up, left, down, right, elapsed_time
    cdef list dir_rec
    cdef list direction
    cdef list current_position
    cdef list pos_rec


    def __init__(self):
        self.init_value = Params("input.yml")
        self.cell_size = self.init_value.cell_size_xy
        self.count = 0
        self.output_count = self.init_value.print_per
        self.dir_rec = [0, 0, 0, 0, 0]

    cpdef start_setting(self):
        self.direction = [0, 0]
        self.current_position = [int(self.cell_size / 2), int(self.cell_size / 2)]
        if os.path.exists("Record") is False:
            os.mkdir("Record")
        self.pos_rec = pos_rec_lattice(self.cell_size)
        self.probability_set()
        self.start_time = time.time()

    cdef probability_set(self):
        cdef double total
        self.up = self.init_value.probabilities["up"]
        self.right = self.init_value.probabilities["right"]
        self.down = self.init_value.probabilities["down"]
        self.left = self.init_value.probabilities["left"]
        total = self.up + self.down + self.left + self.right
        self.up = self.up / total
        self.left = self.left / total + self.up
        self.down = self.down / total + self.left
        self.right = self.right / total + self.down

    cpdef loop(self):
        cdef int _
        for _ in range(self.init_value.total_steps):
            self.function_loop()

    cdef function_loop(self):
        cdef double percentage
        self.direction = choice_event(self.up, self.left, self.down)
        self.direction_record()
        self.update_events()
        self.record_position()
        self.count += 1
        if self.count == self.output_count:
            percentage = self.count / self.init_value.total_steps * 100
            print(f"Progress:␣{percentage:.2f}␣%")
            self.output_count += self.init_value.print_per

    cdef direction_record(self):
        if self.direction == [0, -1]:
            self.dir_rec[0] += 1
        elif self.direction == [-1, 0]:
            self.dir_rec[1] += 1
        elif self.direction == [0, 1]:
            self.dir_rec[2] += 1
        elif self.direction == [1, 0]:
            self.dir_rec[3] += 1
        else:
            self.dir_rec[4] += 1

    cdef update_events(self):
        self.current_position[0] = self.boundary_check(
            self.current_position[0] + self.direction[0]
        )
        self.current_position[1] = self.boundary_check(
            self.current_position[1] + self.direction[1]
        )

    cdef int boundary_check(self, int val):
        if val < 0:
            return self.cell_size - 1
        elif val >= self.cell_size:
            return 0
        else:
            return val

    cdef record_position(self):
        self.pos_rec[self.current_position[0]][self.current_position[1]] += 1

    cpdef end_of_loop(self):
        self.elapsed_time = time.time() - self.start_time
        self.minute = math.floor(self.elapsed_time / 60)
        self.second = int(self.elapsed_time % 60)
        record_data(self.pos_rec, self.minute, self.second)
        print(self.dir_rec)
        print("Finished:␣" + str(self.minute) + "␣min␣" + str(self.second) + "␣sec")
```

Listing 2.9: sub_function.pyx

```
# cython: language_level=3, boundscheck=False, wraparound=False

import random

cpdef list pos_rec_lattice(int cell_size):
    cdef list pos_rec
    pos_rec = [[0 for _ in range(cell_size)] for _ in range(cell_size)]
    return pos_rec


cpdef list choice_event(double up, double left, double down):
    cdef double random_val
    random_val = random.random()
    if random_val <= up:
        return [0, -1]
    elif random_val <= left:
        return [-1, 0]
    elif random_val <= down:
        return [0, 1]
    else:
        return [1, 0]
```

Listing 2.10: recording.pyx

```
# cython: language_level=3, boundscheck=False, wraparound=False

import seaborn as sns
import matplotlib.pyplot as plt


cpdef record_data(list pos_rec, int minute, int second):
    plt.figure()
    sns.heatmap(pos_rec)
    plt.savefig("Record/heat_map.png")
    #
    file_data = open("Record/execution_time.txt", "a")
    file_data.write(str(minute) + " min " + str(second) + " sec" "\n")
    file_data.close()
```

Listing 2.11: recording.pyx

```
# cython: language_level=3, boundscheck=False, wraparound=False

import yaml

cdef class Params:
    cdef public int cell_size_xy
    cdef public dict probabilities
    cdef public int total_steps
    cdef public int print_per
    def __init__(self, filename):
        if filename:
            with open(filename) as f:
                input_yaml = yaml.safe_load(f)
            for k, v in input_yaml.items():
                setattr(self, k, v)
```

2.4 C import cython code

Listing 2.12: main functions.pyx

```
# cython: language_level=3, boundscheck=False, wraparound=False

from InputParameter cimport Params
import time
from sub_functions cimport pos_rec_lattice, choice_event
from recording cimport record_data
import os
import math


cdef class functions:
    !# Type declareion of self variables are moved to pxd file!
    def __init__(self):
......
```

Listing 2.13: main functions.pxd

```
# cython: language_level=3, boundscheck=False, wraparound=False

from InputParameter cimport Params

cdef class functions:
    # Declaretion of self variables, which are moved from pyx file.
    cdef Params init_value
    # In pxd file, user defined class can be used to type assignment. Here, type "
        Params" is defined as the type of "init_value".
    cdef int cell_size, count, output_count, minute, second
    cdef double start_time, up, left, down, right, elapsed_time
    cdef list dir_rec
    cdef list direction
    cdef list current_position
    cdef list pos_rec
    #
    # Declaretion of functions. These should be consistent with those in pyx file.
    cpdef start_setting(self)
    cdef probability_set(self)
    cpdef loop(self)
    cdef function_loop(self)
    cdef direction_record(self)
    cdef update_events(self)
    cdef int boundary_check(self, int val)
    cdef record_position(self)
    cpdef end_of_loop(self)
```

Listing 2.14: sub functions.pxd

```
# cython: language_level=3, boundscheck=False, wraparound=False

cdef list pos_rec_lattice(int cell_size)
cdef list choice_event(double up, double left, double down)
```

Listing 2.15: recording.pxd

```

# cython: language_level=3, boundscheck=False, wraparound=False

cdef record_data(list pos_rec, int minute, int second)
```

Listing 2.16: InputParameter.pyx

```
# cython: language_level=3, boundscheck=False, wraparound=False

import yaml

cdef class Params:
    def __init__(self, filename):
        if filename:
            with open(filename) as f:
                input_yaml = yaml.safe_load(f)
            for k, v in input_yaml.items():
                setattr(self, k, v)
```

Listing 2.17: InputParameter.pxd

```
# cython: language_level=3, boundscheck=False, wraparound=False

cdef class Params:
    cdef public int cell_size_xy
    cdef public dict probabilities
    cdef public int total_steps
    cdef public int print_per
```

Listing 2.18: InputParameter.pxd

```
from setuptools import setup
from Cython.Build import cythonize
import Cython.Compiler.Options
from distutils.extension import Extension

Cython.Compiler.Options.annotate = True

ext_modules = [
    Extension("InputParameter", ["InputParameter.pyx"]),
]
setup(ext_modules=cythonize(ext_modules, annotate=True))
#
ext_modules = [
    Extension("recording", ["recording.pyx"]),
]
setup(ext_modules=cythonize(ext_modules, annotate=True))
#
ext_modules = [
    Extension("sub_functions", ["sub_functions.pyx"]),
]
setup(ext_modules=cythonize(ext_modules, annotate=True))
#
ext_modules = [
    Extension("main_function", ["main_function.pyx"]),
]
setup(ext_modules=cythonize(ext_modules, annotate=True))
#
```