Python for High Performance Computing

Exercise Solutions

Overview

About

4 exercises are given in this notebook:

- Exercise 1: calculate the median from a list;
- Exercise 2: basic NumPy array indexing;
- Exercise 3: Monte--Carlo integration;
- Exercise 4: function integration using the trapezium rule;

Exercise 1: calculate the median from a list

Define a function called `my_median()` that

- takes a list of years (assumed to have an odd number of elements $N \geq 3$),
- computes the median based on the elapsed number of years since 1900 for each entry in the list and
- return the median *and* the value either side of the median as a list.

For instance, if the input is `years = [1989, 1955, 2011, 1943, 1975]`, then the returned result should be `[55, 75, 89]`.

Programming in the cell below , follow these steps:

1. Define the function and use the above example list as input to check you get the right answer.
2. The list needs to be sorted -- how?
3. List indexing may help you to get the final result, *e.g.* `list[3:5]`.

> *Hint*: the `list.append()` method is very useful for this exercise.

In [1]:

```
def my_median(years):
    ages = []
    for y in years:
        ages.append(y - 1900)

    ages.sort()
    mid = len(ages)/2

    return ages[mid-1:mid+2]

years = [1989, 1955, 2011, 1943, 1975]
print my_median(years)
```

```
[55, 75, 89]
```

Exercise 2: basic NumPy array indexing

Using the cell below for programming, follow the following steps:

1. Define a 1D `NumPy` array `x`, containing random numbers (real, double precision).
2. Using index slicing and negative indices, assign the "interior" of the array `x` (the entire array except the first and the last entry) to a new array variable `y`.
3. Using slicing, create yet another array `z` that contains every other element of `x`, starting with the first.
4. Using fancy indexing, create an array `z2` that contains all the negative elements of `z`.
5. Reshape the array `x` into a 2D array and assign the result to `x2`. Verify whether the two arrays share the same data or not.

In [2]:

```
import numpy as np

x = np.random.uniform(-10,10,10)
print x

y = x[1:-1:1]
print y

z = x[0::2]
print z

z2 = z[ z <0 ]
print z2

x2 = np.reshape(x, (-1,2))
print x2

print "id(x)=",id(x)
print "id(x2)=",id(x2)
```

```
[ 7.26197262  8.97474938 -0.07695525  2.50310748  2.5784886  -7.17230828
  6.68965218  4.21433916  5.74431354 -8.87489389]
[ 8.97474938 -0.07695525  2.50310748  2.5784886  -7.17230828  6.68965218
  4.21433916  5.74431354]
[ 7.26197262 -0.07695525  2.5784886   6.68965218  5.74431354]
[-0.07695525]
[[ 7.26197262  8.97474938]
 [-0.07695525  2.50310748]
 [ 2.5784886  -7.17230828]
 [ 6.68965218  4.21433916]
 [ 5.74431354 -8.87489389]]
id(x)= 143282224
id(x2)= 143121504
```

Exercise 3: Monte--Carlo integration

Computing π using Monte-Carlo integration

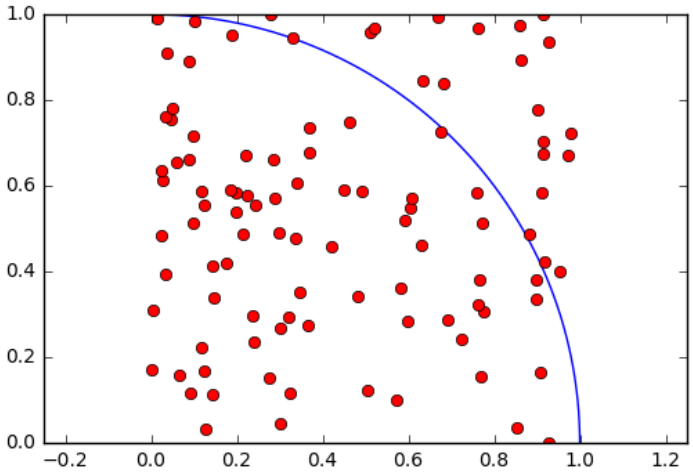
Computing high-dimensional definite integrals with complicated boundaries was the first use of the Monte-Carlo method. Using this approach, π can be computed as the surface integral over the unit circle of a constant unit integrand.

The algorithm of computing π is the area of the unit circle in the 2D plane is to "throw darts" and to count those that fall in the circle. More specifically,

- generate a number of N (uniformly distributed) points in the 2D plane ;
- count the number N_{in} of points in the unit circle;
- compute the approximation as $\pi \approx N/N_{\text{in}}$.

Exploiting the symmetry of the unit circle, only the area of a single quadrant of the circle can be computer. Thus, the random points are generated with coordinates in the $[0, 1)$ interval and the estimated value of pi is

$$\pi_{approx} = 4\frac{N}{N_{\text{in}}} \longrightarrow \pi \qquad \text{as} \qquad N \longrightarrow \infty$$



Setup

The initial Python scripts for this exercise is found in the directory `darts/`, which you are asked to modify using a text editor of your choice. All Python scripts are to be run directly at the command line.

Aim

Building on the provided scripts which use only Python, you are also asked to provide alternative implementations of the method scripted using

- `NumPy`
- `Numba`
- `multiprocessing`

Instructions

Go into the directory `darts/` and follow the following steps:

Step 1: Edit the files `darts.py`,`run_darts.py` and `plot_results.py` and understand their structure and functionality. Run the test file `run_darts.py` and observe the output. Notice how the number of tries needed for the function `numQuadrantHitsPython` are generated using the `NumPy` function `logspace`. Verify the code computes π with increasing accuracy as the number of tries (controlled by `numPoints`) increases.

Step 2: Add a new function `numQuadrantHitsNumPy` to `darts.py` that performs the same operations as `numQuadrantHitsPython` but using `NumPy` vectorised operations:

- generate `x` and `y` as random `NumPy` arrays using the `numPy` function `random.ranf`;
- use fancy indexing and array functions to compute the number of "hits".

  Modify the file `run_darts.py` to time the execution of the function `numQuadrantHitsNumPy` and verify the accuracy of its result.

```
#
# ===== Numpy implementation
#
import numpy
def numQuadrantHitsNumPy (numPoints):

    x = numpy.random.uniform (low=0.0, high=1.0, size=numPoints)
    y = numpy.random.uniform (low=0.0, high=1.0, size=numPoints)
    r = numpy.sqrt (x*x + y*y)
    c = r [ r <= 1.0 ]
    return c.size
```

Step 3: Similarly, add a function `numQuadrantHitsNumba` to `darts.py` that performs the same operations as `numQuadrantHitsPython` using the `Numba` JIT compiler:

- start with the same code as for `numQuadrantHitsPython`;
- import the numba module and add the appropriate decorator;
- remember to run the new function once (on a small number of tries `numPoints`) to trigger the JIT compilation and cache the results.

  Again, modify the file `run_darts.py` to time the execution of the function `numQuadrantHitsNumba` and verify the accuracy of its result.

```
#
# ===== Numba implementation
#
from numba import jit
@jit
def numQuadrantHitsNumba (numPoints):
    nhits = 0
    for n in range(numPoints):
        # generate random point inside the unit square (0,0), (0,1), (1,1), (1,0)
        x = random.random ()
        y = random.random ()
        # if inside unit circle, count a hit
        if (x*x + y*y) <= 1.0:
            nhits += 1
    return nhits
```

Step 4: Edit the file `darts.py` again and implement a solution that uses the `multiprocessing` module. For this, create a funtion called `numQuadrantHitsMP` that

- opens a pool of processes
- use `map_async` to call the function `numQuadrantHitsNumpy` independently from each process
- each call to `numQuadrantHitsNumpy` computes a partial result of the hits calculation

  `map_async` is mapped to a Python iterable (array or list) that contains the partial number of tries per process repeated a number of times; this partial number is `numPoints / numProcs`. While each process carries out only a part of the calculation, together, the processes carry out all the calculations cooncurrently. The raw results from `map_async` is a Python object; use the `get()` method to retrieve the partial hit counts into a list and the function `sum()` to sum up all the partial hit counts into the total.

  After retrieving the partial hit counts using `get()`, print the values. What do you notice? What happens?

Modify the function `numQuadrantHitsNumpy` (preferably creating a new one) by adding a random number generator seed using the function `numpy.random.seed()` and the process rank. In order to map the new function to data using `map_async`, you will need to

- allow `numQuadrantHitsNumpy` to take a tuple as argument, to contain the original `numPoints` as well as the value `seed`;
- create an appropriate list of tuples to map the function onto.

Check the accuracy of the results using `darts_test.py` and make sure the `multiprocessing` implementation scales in performance with the number of processes (*i.e.* the processing time roughly halves when the number pf processes doubles).

```
#
# ===== Numpy implementation MODIFIED for multiprocessing
#
import numpy
def numQuadrantHitsNumpyMOD ((numPoints, seed)):

    numpy.random.seed (seed)
    x = numpy.random.uniform (low=0.0, high=1.0, size=numPoints)
    y = numpy.random.uniform (low=0.0, high=1.0, size=numPoints)
    r = numpy.sqrt (x*x + y*y)
    c = r [ r <= 1.0 ]
    return c.size


#
# ===== Multi-Processing implementation
#
import multiprocessing
def numQuadrantHitsMP (numPoints, numProcs=4):

    # number of points dealt with by each process
    numPointsProc = numPoints / numProcs

    # array with values to map to
    numPointsProcVec = [(numPointsProc, sd) for sd in range(numProcs)]

    # open a pool
    procPool = multiprocessing.Pool (numProcs)

    # asynchronous map
    rawNumHitsProc = procPool.map_async (numQuadrantHitsNumpyMOD, numPointsProcVec
)
    rawNumHitsProc.wait () # not needed in fact

    # clean up pool
    procPool.close () # close task pool (cannot submit new tasks from here on)
    procPool.join ()  # __main__ must wait for all tasks to complete

    # retrieve results
    numHitsProcVec = rawNumHitsProc.get()

    # return result = sum of the list numHitsProcVec
    return sum(numHitsProcVec)
```

Step 5: Finally, run both `run_darts.py` and use `plot_results.py` in order to cross-compare all solutions. Thus, for the same number of tries, make sure that

- the same level of accuracy in estimating π is achieved with all solutions;

Exercise 4: Function integration using the trapezium rule

Integration by the trapezium rule

You are asked to evaluate a definite integral using the trapezium rule.

The function is defined as $f(x) = 2.0 * sqrt(1.0 - x^2)$ which is integrated over -1.0 to +1.0 to obtain π

Setup

The initial Python scripts for this exercise is found in the directory `integral/`, which you are asked to modify using a text editor of your choice. All Python scripts are to be run directly at the command line.

Aim

Building on the provided scripts which use only Python, you are also asked to provide alternative implementations of the method scripted using

- `NumPy`
- `Numba`
- `Cython`
- `mpi4py`

Instructions

Go into the directory `integral/` and follow the following steps:

Step 1: Edit the files `integral.py`, `run_integral.py` and `plot_results.py` and understand their structure and functionality. Run the test file `run_integral.py` followed by `plot_results.py` and observe the output. Notice how the number of integration intervals `N` needed for the function `trapintPython` are generated using the `NumPy` function `logspace`. Verify the code computes the value of π with increasing accuracy as the number of intervals (controlled by `N`) increases.

Step 2: The function `trapintPython` computes the integral of a mathematical function defined by `funcPython`, which, in turn, uses the sqrt function from `math`. Add a new function `trapintNumPy` to `integral.py` that performs the same operations as `trapintPython` but using `NumPy` vectorised operations. For this purpose, create a `NumPy` version of `funcPython` called `funcNumpy` that uses the sqrt function from `numpy` to operate on an entire array rather than a single scalar.

Modify the file `run_integral.py` to time the execution of the function `trapintNumPy` and verify the accuracy of its result. Satisfy yourself the execution of the `NumPy` solution is faster than the pure Python implementation.

```
    # use numpy.sin, which can work on numpy arrays
    import numpy

    #
    # ----- function to integrate
    #
    def funcNumpy (x):
        """Function to integrate"""
        return 2.0 * numpy.sqrt (1.0 - x*x)

    #
    # ----- integrator
    #
    def trapintNumPy (a, b, N):
        """Compute a definite integral using the trapezium rule and NumPy arrays"""

        # interval length (N intervals = N+1 nodes)
        h = (b - a) / float (N)

        # initial and final point only count with weight half
        v = (funcNumpy (a) + funcNumpy (b)) / 2.0

        # N+1 nodes
        x = numpy.linspace(a, b, num=N+1)

        # evaluate function at interior nodes
        y = funcNumpy (x[1:-1])

        # add the interior points
        v = v + y.sum()

        # scale by the interval width
        return v*h
```

Step 3: Similarly, add a function `trapintNumba` to `integral.py` that performs the same operations as `trapintPython` using the `Numba` JIT compiler:

- start with the same code as for `trapintPython`;
- import the numba module and add the appropriate decorator;
- remember to run the new function once (on a small number of interval `N`) to trigger the JIT compilation and cache the results.

  Again, modify the file `run_integral.py` to time the execution of the function `trapintNumba` and verify the accuracy of its result.

```
#
# ===== Numba implementation
#

#
# ----- integrator
#
from numba import jit
from numba import vectorize

@vectorize("float64 (float64)", nopython=True, target="parallel")
def funcNumba (x):
    """Function to integrate"""
    return 2.0 * numpy.sqrt (1.0 - x*x)

@jit ("float64 (float64, float64, int64)")
def trapintNumba (a, b, N):
    """Compute a definite integral using the trapezium rule and pure Python"""

    # interval length (N intervals = N+1 nodes)
    h = (b - a) / float (N)

    # initial and final point only count with weight half
    v = (funcNumba (a) + funcNumba (b)) / 2.0

    # N+1 nodes
    x = numpy.linspace(a, b, num=N+1)

    # evaluate function at interior nodes
    y = funcNumba (x[1:-1])

    # add the interior points
    v = v + y.sum()

    # scale by the interval width
    return v*h
```

Step 4: Now, copy the source of the `trapintNumpy` function into a file `integral.pyx` and modify this source to create a `Cython` function `trapint` that performs the optimised function of the original `trapintNumpy`.

Start by generating a `Cython` implementation from the original pure Python source:

- create a `distutil` setup file and install the Cython module with `python setup build_ext --inplace`;
- modify the file `run_integral.py` to time the execution of the `Cython` function `trapint` and verify the accuracy of its result.

  Next, work on improving the performance of the `Cython` function `trapint`. In particular
- type all variables as well as the function itself;
- replace the use of the Python `math` sqrt function with the C equivalent (see hint below);
- verify the results of these steps by running the test `run_integral.py`.

> *Hint*: Import the standard C math sqrt funtion with
>
> `from libc.math cimport sqrt`

Further, improve the performance of of the Cython solution via multithreading. This involves two important aspects:

- releasing the GIL for the main loop in `trapint` and
- using the parallel variant `prange` of the iterator.

Pass the number of threads to use as an argument to the function `trapint` and use `openmp.omp_set_num_threads()` to set the number of OpenMP threads. Using different number of threads

- test that the Cython function produces accurate results and
- ensure the performance of the threaded function scales with the number of threads (*i.e.* execution time is nearly halved by doubling the number of threads).

```
import cython
# import modules for multithreading
from cython.parallel import prange, parallel
cimport openmp
# import external C mathematical functions
from libc.math cimport sqrt

# eliminate Python checks
@cython.cdivision(True)


#
# ----- function to integrate
#
cdef double func (double x) nogil:
    """Function to integrate"""
    return 2.0*sqrt(1.0-x*x)


#
# ----- integrator
#
cpdef double trapint (double a, double b, int N, int nt=4):
    """Compute a definite integral using the trapezium rule and pure Python"""

    # variables
    cdef:
        int n
        double x, h, v

    # set number of threads
    openmp.omp_set_num_threads(nt)

    # interval length (N intervals = N+1 nodes)
    h = (b - a) / float (N)

    # initial and final point only count with weight half
    v = (func (a) + func (b)) / 2.0

    # add the interior points
    #   * thread-locality and reductions are automatically inferred for variables
    with nogil:
        for n in prange(1,N):
            x = a + n*h
            # NB reductions are recognised from "+=" only
            v +=  func (x)

    # scale by the interval width
    return v*h
```

Step 5: Write a file `integral_mpi.py` to implement a `mpi4py` version of the test in `run_integral.py`. Just as the original, the `mpi4py` implementation loops over a range of values for `N` and computes an approximation for the definite integral using the trapezium rule. Unlike `run_integral.py`, the `mpi4py` version computes the intergal value in a distributed fashion, with each process resposible for calculating the integral for only a part of the original interval of integration.

Here are the steps to follow in writing `integral_mpi.py`:

- Loop over the same range of values for `N` as in `integral_test.py`. For each value of `N`, each process computes the integral of the function for a part of the interval from `a` to `b`. Namely, the integration for process `rank` of `size` number of processes is between `aProc` and `bProc`, where

  ```
  N1 = (float(N)* rank)   / size
  N2 =  float(N)*(rank+1) / size
  aProc = a + N1 * h
  bProc = a + N2 * h
  ```

- Each process uses the `NumPy` function `trapintNumPy` for the calculation of a partial integral value `valProc`, computed between the limits `aProc` and `bProc` and using a number of intervals equal to `N2-N1`.
- The total value of the integral (between `a` and `b`) is obtained from sum-reducing all the partial results `valProc`. This is achieved by placing the partial result on each process in a vector of size 1 and by reducing all these partial vectors across all processes to a final result vector on a single process, typically the rank 0.
- Use the function `MPI.Wtime()` to measure execution time for each value of `N`.
- The approximated value for the integral (or the absolute error) is printed by the process of rank 0 only, along with the execution time.

  Demonstrate to yourself that the performance of `integrate_mpi.py` scales with the number of processes used (*i.e.* execution time is nearly halved by doubling the number of processes).

```
from mpi4py import MPI
import math
import numpy
import integral

def main (comm):
    """Compute a definite integral using the trapezium rule"""

    # size and rank of communicator
    size = comm.Get_size()
    rank = comm.Get_rank()

    # interval ends
    a,b = -1.0, +1.0

    # array with values for number of points
    NVals = numpy.logspace (1, 8, base=10.0, num=6, dtype=numpy.int)

    if rank == 0:
        fh =  open( "results_mpi_{:d}.txt".format(size), "w" )

    # compute Pi (as a definite integral) using two methods
    for N in NVals:

        # use Wtime to time processing
        wt = MPI.Wtime()

        # calculate local integration limits

        h = (b - a) / float (N)
        N1 = (float(N)* rank)   / size
        N2 =  float(N)*(rank+1) / size
        aProc = a + N1 * h
        bProc = a + N2 * h

        # local integral value
        valProc = integral.trapintNumPy (aProc, bProc, N2-N1)

        # reduce values
        valProcVec = numpy.array ([valProc])
        valVec     = numpy.zeros (1, dtype=float)
        comm.Reduce (valProcVec, valVec, op=MPI.SUM, root=0)

        # use Wtime to time processing
        wt = MPI.Wtime() - wt

        # root process writes result
        if rank == 0:
            fh.write("{:d} {:6.4e}\n".format(N, wt))
            e1 = math.fabs (valVec[0] - math.pi)
        print "MPI({:d}) N={:9d} Time={:6.4e} Error={:12.8e}".format( size, N, wt,
e1 )
```

```
        if rank ==0:
            fh.close()

    if __name__ == "__main__":
        """Run with a large number of points"""
        main (MPI.COMM_WORLD)
```

Step 6: Finally, run both `run_integral.py` and `integral_mpi.py` once more in order to cross-compare all the 4 solutions. Thus, for the same number `N` of intervals, make sure that

- the same level of accuracy in estimating the integral is achieved with all solutions;
- execution time for the `Cython` implementation (with 1 thread) and the `Numba` one should be comparable;
- execution time for the `Cython` implementation (with 2 and 4 threads) and the `mi4py` one (with 2 and 4 processes, respectively) should also be comparable.

In [ ]: