蒙特克洛模拟的优化

2016-12-09 | 0 | 198

本文将描述我对蒙特卡洛模拟的一些优化，填坑之前的文章 python初探：python实现蒙特卡洛方法计算π值。

蒙特克洛模拟

《 Python 金融大数据分析》中对门特卡洛模拟的描述是这样的：

> 蒙特克洛模拟是金融学和数值科学中最重要的算法之一。它之所以重要，是因为在期权定价或者风险管理问题上有很强的能力。和其他数值方法相比，蒙特卡洛方法很容易处理高维问题，在这种问题上复杂度和计算要求通常以线性方式增大。
>
> 蒙特卡洛方法的缺点是：它本身是高计算需求的，即使对于相当简单的问题也往往血药海量的计算。因此，必须高效地实现蒙特卡洛算法。

下面的例子我将用 Python 的不同实现策略，并提供 3 种不同的基于蒙特克洛模拟的对 π 值估算方法。
原始材料来源于先前一篇文章中遗留的性能优化问题，主要的逻辑思路在其中都有阐述。作为刚刚接触 Numpy 的

小白，写的这篇实践记录中难免有错谬之处，希望各位大神指正。

测试电脑的配置情况：2.7 GHz Intel Core i5，8 GB 1867 MHz DDR3

纯 Python

用纯 Python 模拟 1000 万次，中间用了一个计时器：

```
from __future__ import division
import random
from time import time

seed = 0
t0 = time()

j = 7
counter = 0
I = 10 ** j
for i in range(I):
    x = random.uniform(-1, 1)
    y = random.uniform(-1, 1)
    if x ** 2 + y ** 2 < 1:
        counter = counter + 1
result = 4 * (counter / I)

typ = time() - t0
print 'pi                %7.5f' % result
print 'Duration in Seconds %7.5f' % typ
```

运行以上脚本获得如下输出：

```
In [1]: %run /Users/wonderful/Desktop/Pure_Python.py
pi              3.14176
Duration in Seconds 18.71270
```

使用 Numpy 优化 1

做数值运算的时候，我的第一反应就是上 Numpy ，改动是简单地把生成随机数的 random 改为 numpy：

```
from __future__ import division
import numpy as np
from time import time

seed = 0
t0 = time()

j = 7
counter = 0
I = 10 ** j
for i in range(I):
    x = np.random.uniform(-1, 1)
    y = np.random.uniform(-1, 1)
    if x ** 2 + y ** 2 < 1:
        counter = counter + 1
result = 4 * (counter / I)

typ = time() - t0
print 'pi              %7.5f' % result
print 'Duration in Seconds %7.5f' % typ
```

运行以上脚本：

```
In [2]: %run /Users/wonderful/Desktop/Vectorization_with_Numpy_1.py
pi              3.14194
Duration in Seconds 13.94915
```

这里运行时间减少了 25.5% ，相当于运行速度提升了 34%。用 Numpy 的优势还是比较明显的。

使用 Numpy 优化 2

使用 Numpy 生成伪随机数时的一大优势是，这 1000 万个数字可以只需要一行代码，而不需要循环：

```
x = np.random.uniform(-1, 1, I)
y = np.random.uniform(-1, 1, I)
```

所以改动代码得到：

```
from __future__ import division
import numpy as np
from time import time

seed = 0
t0 = time()

j = 7
counter = 0
I = 10 ** j
x = np.random.uniform(-1, 1, I)
y = np.random.uniform(-1, 1, I)
for i in range(I):
    if (x[i]) ** 2 + (y[i]) ** 2 < 1:
        counter = counter + 1
result = 4 * (counter / I)

typ = time() - t0
print 'pi                %7.5f' % result
print 'Duration in Seconds %7.5f' % typ
```

下面我们运行上述脚本：

```
In [3]: %run /Users/wonderful/Desktop/Vectorization_with_Numpy_2.py
pi               3.14194
Duration in Seconds 13.49130
```

可以发现与 *使用 Numpy 优化 1* 相比，速度依然有所提升——运行时间再减少 3%。

总结

使用 Numpy 的优势是非常明显的，在没有改进逻辑的情况下，速度有了显著的提升。当然，这里的例子其实没有完全展现使用 Numpy 的优势，因为这里的另一个循环——计算落在圆内的数字这一步，是无法避免的，而运用得当的话， Numpy 其实可以带来数十倍的速度提升。

感觉本站内容不错，读后有收获 - - 不妨小额赞助我一下，让我有动力继续写出高质量的教程！

赏

Python # Monte_Carlo_Simulation # Numpy

‹ Restart 一点微小的工作 ›