dOpt Programming Project - Subproject 2 - Class 2 Group 34

Christian Zhuang-Qing Nielsen[*]

March 16, 2018

Contents

1 Subproject 2: Time analysis **1**
1.1 The method 1
1.2 Code-Snippet 1
1.3 The results 2
1.4 Conclusion 3

1 Subproject 2: Time analysis

1.1 The method

To test the speed of my LPSolve, I have used the built in time.time() function, and then simply measured the difference between the call and return of the LPSolve method. Because time() works in seconds, the linear programs must be sufficiently large to show a measurable difference in time. I have converted the time results to milliseconds afterwards. As the algorithm is still the one-phase simplex method, it only works on feasible and unbounded linear problems. To generate sufficiently large linear programs, I have utilised the included function of RandomLP, generating 30×30-size matrices. To increase similarity between scipy and LPSolve, I have forced scipy to utilise Bland's rule for anti-cycling as well (as that is what LPSolve currently uses).

1.2 Code-Snippet

Below is a snippet of the code I use to compare the running times:

```
# Time testing example
c, A, b = RandomLP(30, 30)
```

[*]201504624, christian@czn.dk

```
pre_time = time.time()
Res, D = LPSolve(c, A, b, Fraction)
post_time = time.time()
time_res = (post_time - pre_time) * 1000
print("Random␣Example␣with␣Fraction:")
print(D)
print("Time␣spent:", time_res, "ms")
print()

pre_time_float = time.time()
Res, D = LPSolve(c, A, b, np.float64)
post_time_float = time.time()
time_res_float = (post_time_float - pre_time_float) * 1000
print("Random␣Example␣with␣Float:")
print(D)
print("Time␣spent:", time_res, "ms")
print()

print("Random␣Example␣with␣Scipy")
pre_time_scipy = time.time()
print(scipy.optimize.linprog(-c, A, b, options={'bland': True}))
post_time_scipy = time.time()
time_res_scipy = (post_time_scipy - pre_time_scipy) * 1000
print("Scipy␣Time␣spent:", time_res_scipy, "ms")
print()

print("Time␣comparison:\n\nFractions:", time_res, "ms\nFloat:",
      time_res_float, "ms\nScipy:", time_res_scipy, "ms")
```

1.3 The results

The recorded time results in milliseconds:

```
*Time comparison: (Optimal example)*
Fractions: 1078.1760215759277 ms
Float: 46.849966049194336 ms
Scipy: 31.22425079345703 ms

*Time comparison: (Optimal example)*
Fractions: 1890.7215595245361 ms
Float: 46.87166213989258 ms
Scipy: 78.13191413879395 ms

*Time comparison: (Unbounded example)*
Fractions: 2421.9985008239746 ms
Float: 109.38382148742676 ms
```

```
Scipy: 78.1252384185791 ms

*Time comparison: (optimal)*
Fractions: 2812.6401901245117 ms
Float: 46.86903953552246 ms
Scipy: 62.47711181640625 ms

*Time comparison: (Optimal example)*
Fractions: 2171.9815731048584 ms
Float: 78.10401916503906 ms
Scipy: 93.72973442077637 ms
```

1.4 Conclusion

Evidently, using fractions is significantly slower than using floats (as the numbers has to be exact). In smaller linear programs, the precise numbers may be worth it, but in real-life huge-scale applications the computational time could take a uselessly large amount of time. Floats, while having a short computaitonal time are not as precise, is fine to a degree. The problem arises if the amount of iterations is huge (which gives more and more rounding errors with each multiplications), and thus a less-than-optimal result. Comparing my LPSolve to the `scipy.optimize.linprog`, it is really only fair to compare the floats, as that is what scipy is using. In general my LPSolve performs a bit better, but this is obviously due to the fact my LPSolve currently assumes that the given linear program isn't unfeasible, which reduces it computational time. However, this cannot excuse more than the difference between the two at least.

All in all, the running time of my algorithm is satisfactory. I have currently not used list comprehensions (which runs faster than normal loops in python) in my code, which would probably speed up LPSolve even further if done so. This is too not important for the project though and the algorithm already runs fast.