ISYE 6663 Project

April 27, 2017

1 Ruilin Li

1.1 Problem 1

```
In [3]: %matplotlib inline
        import numpy as np
        import time
        from optimizer import Optimizer
```

1.2 Extended Rosenblack Function

We set $n = 200$ to test rate of convergence and running time and use different n to compare memory usage between BFGS and L-BFGS methods.

```
In [2]: x0 = np.array([-1.2, 1] * 100) # 200 dimension
        optimizer_erf = Optimizer('erf')
```

1.2.1 Steepest Descent

```
In [23]: print "---- steepest descent start ----"
         start = time.time()
         result_sd = optimizer_erf.steepest_descent(x0)
         end = time.time()
         print "minimum: {}\niteration: {}\ntime: {}s".format(
             result_sd[1], result_sd[2], end - start)
         print "---- steepest descent end ----\n"
```

```
---- steepest descent start ----
minimum: 0.00241920182986
iteration: 1537
time: 1.01918983459s
---- steepest descent end ----
```

1.2.2 Conjugate Gradient

During optimization process of Fletcher-Reeves conjugate gradient method, the line search algorithm provided by scipy.optimize.line_seach() does not converge, causing my program to break down. I implemented a simpler version of line search algorithm which satisfies wolfe condition, slower and unstable, though. This is why my Fletcher-Reeves variant is significantly slower than Polak-Ribiere variant, besides the fact that the latter one required fewer iterations to converge.

```
In [24]: # conjugate gradient - Fletcher-Reeves
         print "---- conjugate gradient-fr start ----"
         start = time.time()
         result_cg_fr = optimizer_erf.conjugate_gradient_fr(x0, scipy_ls=False)
         end = time.time()
         print "minimum: {}\niteration: {}\ntime: {}s".format(
             result_cg_fr[1], result_cg_fr[2], end - start)
         print "---- conjugate gradient-fr end ----\n"

         # conjugate gradient - Polak-Ribiere
         print "---- conjugate gradient-pr start ----"
         start = time.time()
         result_cg_pr = optimizer_erf.conjugate_gradient_pr(x0)
         end = time.time()
         print "minimum: {}\niteration: {}\ntime: {}s".format(
             result_cg_pr[1], result_cg_pr[2], end - start)
         print "---- conjugate gradient-pr end ----\n"
```

```
---- conjugate gradient-fr start ----
minimum: 0.00241913238635
iteration: 118
time: 0.699166059494s
---- conjugate gradient-fr end ----

---- conjugate gradient-pr start ----
iteration 0: value: 2420.0
minimum: 0.000250934132895
iteration: 16
time: 0.0115859508514s
---- conjugate gradient-pr end ----
```

1.2.3 Quasi-Newton

```
In [26]: # BFGS
         print "---- quasi_newton_bfgs start ----"
         start = time.time()
         result_qn_bfgs = optimizer_erf.quasi_newton_bfgs(x0)
         end = time.time()
         print "minimum: {}\niteration: {}\ntime: {}s".format(
```

```
    result_qn_bfgs[1], result_qn_bfgs[2], end - start)
print "---- quasi_newton_bfgs end ----\n"

# DFP
print "---- quasi_newton_dfp start ----"
start = time.time()
result_qn_dfp = optimizer_erf.quasi_newton_dfp(x0)
end = time.time()
print "minimum: {}\niteration: {}\ntime: {}s".format(
    result_qn_dfp[1], result_qn_dfp[2], end - start)
print "---- quasi_newton_dfp end ----\n"
```

```
---- quasi_newton_bfgs start ----
minimum: 0.002353219121
iteration: 415
time: 0.450258016586s
---- quasi_newton_bfgs end ----

---- quasi_newton_dfp start ----
minimum: 0.00241974037583
iteration: 13527
time: 9.63689708 71s
---- quasi_newton_dfp end ----
```

I found that the performance of quasi-newton algorithms was incredibly bad for this function, particularlt DFP algorithm. To check whether it was due to bad implementation, I compared my implementation with the verison Scipy provides.

```
In [34]: # My BFGS
print "---- My BFGS start ----"
start = time.time()
result_qn_bfgs = optimizer_erf.quasi_newton_bfgs(x0, eps=1e-13)
end = time.time()
print "minimum: {}\niteration: {}\ntime: {}s".format(
    result_qn_bfgs[1], result_qn_bfgs[2], end - start)
print "---- My BFGS end ----\n"

# Scipy BFGS
from scipy.optimize import minimize
print "---- Scipy BFGS start ----"
start = time.time()
result = minimize(optimizer_erf.f, x0, method='BFGS',
                  jac=optimizer_erf.grad_f)
end = time.time()
print "minimum: {}\niteration: {}\ntime: {}s".format(
    result.fun, result.nit, end - start)
print "---- Scipy BFGS end ----\n"
```

```
---- My BFGS start ----
minimum: 2.37356548384e-10
iteration: 591
time: 0.580150842667s
---- My BFGS end ----

---- Scipy BFGS start ----
minimum: 1.04187962383e-09
iteration: 760
time: 1.61571884155s
---- Scipy BFGS end ----
```

It turned out that my version achived **better** accuracy with **fewer** iterations in **shorter** time.

Therefore, a more reasonable explanation is that the performance of BFGS is worse than that of conjugate gradient methods for this function and the performance of DFP is even worser.

1.2.4 Limited Memory BFGS

```
In [4]: print "---- Limited Memory BFGS start ----"
        start = time.time()
        result_qn_l_bfgs = optimizer_erf.quasi_newton_l_bfgs(x0)
        end = time.time()
        print "minimum: {}\niteration: {}\ntime: {}s".format(
           result_qn_l_bfgs[1], result_qn_l_bfgs[2], end - start)
        print "---- Limited Memory BFGS end ----\n"
```

```
---- Limited Memory BFGS start ----
minimum: 0.00186858918025
iteration: 60
time: 0.0499820709229s
---- Limited Memory BFGS end ----
```

The performance of limited memory BFGS is significanly better than that of BFGS and DFP.

1.2.5 Summary ($n = 200$)

Method	Iteration	Time(s)
Steepest Descent	1537	1.0191898345
C-G-FR	118	0.6991660594
C-G-PR	16	0.0115859508
BFGS	415	0.4502580165
DFP	13527	9.6368970871
L-BFGS	60	0.0499820709

To profile the memory usage of BFGS and L-BFGS methods, I modified my code because the first several iterations of L-BFGS algorithm is the same as that of BFGS algorithm. The diffenrence of memory usage would be negligible without modification.

Dimension	Memory(BFGS)	Memory(L-BFGS)
200	2.9 Mb	< 0.1Mb
400	6.7 Mb	< 0.1Mb
800	25.3 Mb	< 0.1Mb
1600	99.2 Mb	< 0.1Mb
3200	2.3 Mb	< 0.1Mb

I used Python memory_profiler module to profile the memory usage. Since L-BFGS depends on BFGS, hence these two algorithms need to run at the same time, so I highly doubt that the memory usage of L-BFGS is underestimated. However, we observe that the memory usage of BFGS grows quadraticly in dimension n when n is large, which is consistent with theoretic analysis.

1.3 Extended Powell Singular Function

We set $n = 1600$ to test rate of convergence and running time and use different nn to compare memory usage between BFGS and L-BFGS methods.

```
In [5]: y0 = np.array([3.0, -1.0, 0.0, 1.0] * 400) # 1600 dimension
        optimizer_epsf = Optimizer('epsf')
```

1.3.1 Steepest Descent

```
In [28]: # steepest_descent
         print "---- steepest descent start ----"
         start = time.time()
         result_sd = optimizer_epsf.steepest_descent(y0)
         end = time.time()
         print "\nminimum: {}\niteration: {}\ntime: {}s".format(
             result_sd[1], result_sd[2], end - start)
         print "---- steepest descent end ----\n"
```

```
---- steepest descent start ----

minimum: 0.0859546513591
iteration: 2175
time: 3.0121819973s
---- steepest descent end ----
```

1.3.2 Conjugate Gradient

It in interesting that the scipy.optimize.line_search() works well with this function. I did not use my own version of line search algorithm for extended Powell singular function.

```
In [29]: # conjugate_gradient_fr
         print "---- conjugate gradient-fr start ----"
         start = time.time()
         result_cg_fr = optimizer_epsf.conjugate_gradient_fr(y0)
         end = time.time()
         print "minimum: {}\niteration: {}\ntime: {}s".format(
            result_cg_fr[1], result_cg_fr[2], end - start)
         print "---- conjugate gradient-fr end ----\n"

         # conjugate_gradient_pr
         print "---- conjugate gradient-pr start ----"
         start = time.time()
         result_cg_pr = optimizer_epsf.conjugate_gradient_pr(y0)
         end = time.time()
         print "minimum: {}\niteration: {}\ntime: {}s".format(
            result_cg_pr[1], result_cg_pr[2], end - start)
         print "---- conjugate gradient-pr end ----\n"
```

```
---- conjugate gradient-fr start ----
minimum: 0.0859644484262
iteration: 71
time: 0.110360145569s
---- conjugate gradient-fr end ----

---- conjugate gradient-pr start ----
iteration 0: value: 86000.0
minimum: 0.0175363601136
iteration: 20
time: 0.0370700359344s
---- conjugate gradient-pr end ----
```

1.3.3 Quasi-Newton

```
In [30]: # BFGS
         print "---- quasi_newton_bfgs start ----"
         start = time.time()
         result_qn_bfgs = optimizer_epsf.quasi_newton_bfgs(y0)
         end = time.time()
         print "minimum: {}\niteration: {}\ntime: {}s".format(
            result_qn_bfgs[1], result_qn_bfgs[2], end - start)
         print "---- quasi_newton_bfgs end ----\n"

         # DFP
         print "---- quasi_newton_dfp start ----"
         start = time.time()
         result_qn_dfp = optimizer_epsf.quasi_newton_dfp(y0)
```

```
        end = time.time()
        print "minimum: {}\niteration: {}\ntime: {}s".format(
           result_qn_dfp[1], result_qn_dfp[2], end - start)
        print "---- quasi_newton_dfp end ----\n"
```

```
---- quasi_newton_bfgs start ----
minimum: 0.0740397334593
iteration: 35
time: 3.4599738121s
---- quasi_newton_bfgs end ----

---- quasi_newton_dfp start ----
minimum: 0.0854359840647
iteration: 72
time: 4.78077602386s
---- quasi_newton_dfp end ----
```

Quasi-Newton methods work well with function, converging to minimizer much faster than the above one.

1.3.4 Limited Memory BFGS

```
In [6]: print "---- Limited Memory BFGS start ----"
        start = time.time()
        result_qn_l_bfgs = optimizer_epsf.quasi_newton_l_bfgs(y0)
        end = time.time()
        print "minimum: {}\niteration: {}\ntime: {}s".format(
           result_qn_l_bfgs[1], result_qn_l_bfgs[2], end - start)
        print "---- Limited Memory BFGS end ----\n"
```

```
---- Limited Memory BFGS start ----
minimum: 0.0717799128993
iteration: 26
time: 0.975878953934s
---- Limited Memory BFGS end ----
```

1.3.5 Summary ($n = 1600$)

Method	Iteration	Time(s)
Steepest Descent	2175	3.0121819973
C-G-FR	71	0.1103601455
C-G-PR	20	0.0370700359
BFGS	35	3.4599738121

Method	Iteration	Time(s)
DFP	72	4.7807760238
L-BFGS	26	0.9758789539

Dimension	Memory(BFGS)	Memory(L-BFGS)
200	1.9 Mb	$<$ 0.1Mb
400	6.7 Mb	$<$ 0.1Mb
800	25 Mb	$<$ 0.1Mb
1600	99.7 Mb	$<$ 0.1Mb
3200	392 Mb	0.1Mb

1.4 Problem 2

For constrained problem, I used projected Nesterov's accelarate gradient descent method.

```
In [4]: from problem2 import f, grad_f, nesterov_1, nesterov_2
        x0 = np.zeros(1000) # initial point
```

```
In [8]: # Unconstrained optimization
        print "---- unconstrained Nesterov start ----"
        start = time.time()
        result1 = nesterov_1(x0)
        end = time.time()
        print "minimum: {}\niteration: {}\ntime: {}s".format(
            result1[1], result1[2], end - start)
        print "---- unconstrained Nesterov end ----\n"

        # Scipy's Answer
        from scipy.optimize import minimize
        print "---- Scipy minimize start ----"
        start = time.time()
        res = minimize(f, x0)
        end = time.time()
        print "minimum: {}\niteration: {}\ntime: {}s".format(
            res.fun, res.nit, end - start)
        print "---- Scipy minimize end ----\n"


        print "The distance of minimizers found by two algorithms: {}".format(
                np.linalg.norm(result1[0]-res.x))
```

```
---- unconstrained Nesterov start ----
minimum: 5.75535254312e-07
iteration: 140
time: 0.016471862793s
---- unconstrained Nesterov end ----
```

```
---- Scipy minimize start ----
minimum: 1.74715724085e-06
iteration: 36
time: 9.7752058506s
---- Scipy minimize end ----

The distance of minimizers found by two algorithms: 0.00141348344192
```

```
In [9]: # Constrained optimization
        print "---- unconstrained Nesterov start ----"
        start = time.time()
        result2 = nesterov_2(x0)
        end = time.time()
        print "minimum: {}\niteration: {}\ntime: {}s".format(
            result2[1], result2[2], end - start)
        print "---- unconstrained Nesterov end ----\n"

        bnds = tuple([(0, None)] * 1000)
        # Scipy's Answer
        from scipy.optimize import minimize
        print "---- Scipy minimize start ----"
        start = time.time()
        res = minimize(f, x0, bounds=bnds)
        end = time.time()
        print "minimum: {}\niteration: {}\ntime: {}s".format(
            res.fun, res.nit, end - start)
        print "---- Scipy minimize end ----\n"

        print "The distance of minimizers found by two algorithms: {}".format(
                np.linalg.norm(result2[0]-res.x))
```

```
---- unconstrained Nesterov start ----
minimum: 3.41287079867e-05
iteration: 123
time: 0.0140008926392s
---- unconstrained Nesterov end ----

---- Scipy minimize start ----
minimum: 0.0153878457611
iteration: 7
time: 0.357044935226s
---- Scipy minimize end ----

The distance of minimizers found by two algorithms: 0.00796750237519
```

The following code shows that the minimizer obtained from constrained problem actually satisfy the constraints because it is an empty array.

```
In [13]: result2[0][result2[0]<0]
```

```
Out[13]: array([], dtype=float64)
```