Preface

Statistics

All statistics are shown in the form $\mu \pm \sigma$. Where

$$\mu = \frac{1}{n}\sum_{i=1}^{n} x_i$$

and

$$\sigma = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2}$$

The implementations for these can be found in **stats_generator.py** and immediately below.

In [24]:
```
@classmethod
def unbiased_expected_value(cls, data):
    if len(data) == 0:
        return 0
    return sum(data) / len(data)

@classmethod
def std_dev(cls, data):
    if len(data) <= 1:
        return 0
    avg_val = cls.unbiased_expected_value(data)
    variance_list = list(map(lambda x : (x - avg_val) ** 2, data))
    bessel_correction = 1 / (len(data) - 1)
    return math.sqrt(bessel_correction * sum(variance_list))
```

Problem 1

Constrained Linear Programming Comparison (2 algs.)

I decided to use a custom problem I named "FactoryTaxEmissionsProblem" in which 10 factories are taxed on their output and are regulated by emission laws. The emission laws place constraints on how much product two neighboring factories can produce. For example, one such law might be that factories 1 and 2 can only produce 1600 combined units due to the emissions produced as byproducts. The goal of the problem is to determine the optimal number of units that each factory should produce given some emission constraints and tax values. The cost function represents the profit from the ten factories, where the number of units produced by each factory is multiplied by some modifier to represent the post-tax value.

Formally, the cost function is defined as the following:

$$C(\mathbf{x}) = \mathbf{c}^{\mathbf{T}}\mathbf{x}$$

Where:

$$\mathbf{c} = \begin{pmatrix} .8 \\ 1 \\ .95 \\ .72 \\ .9 \\ .87 \\ .72 \\ .82 \\ .93 \\ .67 \end{pmatrix}$$

$$\mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \\ x_{10} \end{pmatrix}$$

The constraints on the variables on the factory emissions:

$$x_1 \leq 1000$$
$$x_2 \leq 1000$$
$$x_3 \leq 3000$$
$$x_4 \leq 1500$$
$$x_5 \leq 1600$$
$$x_6 \leq 750$$
$$x_7 \leq 2000$$
$$x_8 \leq 1100$$
$$x_9 \leq 500$$
$$x_{10} \leq 350$$
$$x_1 + x_2 \leq 1500$$
$$x_6 + x_7 + x_8 = 1800$$

In [5]:
```
import numpy

FactoryCostCoefficients = [.8, 1, .95, .72, .9, .87, .72, .82, .93, .67]
FactoryConstraints = [
        1000, 1000, 3000, 1500, 1600, 750, 2000, 1100, 500, 350, 1500
    ]


class ConstrainedFactoryTaxProblem:
    c = numpy.asarray(FactoryCostCoefficients) * -1
    A_u = numpy.eye(10)
    A_u = numpy.append(A_u, [[1, 1] + [0] * 8], axis=0)
    b_u = FactoryConstraints

    A_eq = numpy.zeros((10, 10))
    A_eq[0] = [0] * 5 + [1, 1, 1, 0, 0]

    b_eq = [1800] + [0] * 9
```

I tested two versions of the scipy **optimize.linprog()** method and the **scipy.minimize()** method (unconstrained). The two constrained scipy methods were the **simplex** and **interior-point** methods. They were tested using the following Python code which ran each algorithm 1000 times. Note that the scipy API does not support providing a starting point.

In [8]:
```
import stats_generator
import scipy.optimize
import time

def test_scipy_method(method, iters):
    test_results = []
    for i in range(iters):
        start = time.time()
        res = scipy.optimize.linprog(c=ConstrainedFactoryTaxProblem.c, A_eq=Constr
ainedFactoryTaxProblem.A_eq, b_eq=ConstrainedFactoryTaxProblem.b_eq,
                                     A_ub=ConstrainedFactoryTaxProblem.A_u, b_ub=C
onstrainedFactoryTaxProblem.b_u, method=method)
        end = time.time()
        test_results.append((res.fun, (end-start) * 1000))

    func_vals = list(map(lambda x: x[0], test_results))
    time_vals = list(map(lambda x: x[1], test_results))

    return ((stats_generator.StatsGenerator.unbiased_expected_value(func_vals), st
ats_generator.StatsGenerator.std_dev(func_vals)),
            (stats_generator.StatsGenerator.unbiased_expected_value(time_vals), st
ats_generator.StatsGenerator.std_dev((time_vals))))

def print_stats(header, stats):
    print(header)
    print("Avg. function value: ", stats_generator.StatsGenerator.stats_to_string(
stats[0]))
    print("Avg. time value (ms): ", stats_generator.StatsGenerator.stats_to_string
(stats[1]))
    print("\n")

stats = test_scipy_method("simplex", 1000)
print_stats("SCIPY SIMPLEX METHOD", stats)

stats = test_scipy_method("interior-point", 1000)
print_stats("SCIPY INTERIOR-POINT METHOD", stats)
```

```
SCIPY SIMPLEX METHOD
Avg. function value:  -8.98e+03±0.00e+00
Avg. time value (ms):  3.95e+00±4.89e-01


SCIPY INTERIOR-POINT METHOD
Avg. function value:  -8.98e+03±2.24e-10
Avg. time value (ms):  4.66e+00±2.60e-01
```

The simplex method achieved a more accurate optimal value in a shorter amount of time compared to the interior-point method.

Unconstrained Optimization with Penalties (1 alg.)

To test the same problem with an unconstrained solver I wrote a new class with a cost() and penalty() function. The cost function is equivalent to that in the previous section, but with the addition of the penalty evaluation of the input vector $\mathbf{x}$.

In [9]:
```
class UnconstrainedFactoryProblem:
    PUNISHMENT = 1e6

    @staticmethod
    def penalty(x):
        total_penalty = 0
        ## Greater than zero constraint
        for i in range(len(x)):
            total_penalty += UnconstrainedFactoryProblem.PUNISHMENT * min(x[i], 0)
** 2
        ## LE constraints
        # x < b
        for i in range(len(x)):
            total_penalty += UnconstrainedFactoryProblem.PUNISHMENT * max(0, x[i]
- FactoryConstraints[i]) ** 2
        # x_0 + x_ 1 <= 1500
        total_penalty += UnconstrainedFactoryProblem.PUNISHMENT * max(0, x[0] + x[
1] - 1500) ** 2

        ## EQ constraints
        fifth_factory_sum = x[5] + x[6] + x[7]
        total_penalty += UnconstrainedFactoryProblem.PUNISHMENT * (fifth_factory_s
um - 1800) ** 2
        return total_penalty

    @staticmethod
    def cost(x):
        return sum(numpy.asarray(FactoryCostCoefficients) * x * -1) + Unconstraine
dFactoryProblem.penalty(x)
```

The unconstrained optimizer from scipy was tested using 20 iterations of the **scipy.optimize.minimize()** function. The starting values were drawn from a uniform distribution [0, 3000).

In [10]:
```
def test_unconstrained_lp(method, iters):
    print("TESTING UNCONSTRAINED METHOD:", method)
    test_results = []

    for i in range(iters):
        x0 = numpy.random.rand(10) * 3000
        start = time.time()
        res = scipy.optimize.minimize(UnconstrainedFactoryProblem.cost, x0=x0, met
hod=method)
        if i == 0:
            print(res)
        end = time.time()
        test_results.append((res.fun, (end-start) * 1000))

    func_vals = list(map(lambda x: x[0], test_results))
    time_vals = list(map(lambda x: x[1], test_results))
    return ((stats_generator.StatsGenerator.unbiased_expected_value(func_vals), st
ats_generator.StatsGenerator.std_dev(func_vals)),
            (stats_generator.StatsGenerator.unbiased_expected_value(time_vals), st
ats_generator.StatsGenerator.std_dev((time_vals))))

stats = test_unconstrained_lp("cg", 20)
print_stats("SCIPY UNCONSTRAINED", stats)
```

```
TESTING UNCONSTRAINED METHOD: cg
     fun: -8157.513800504918
     jac: array([  -0.79998779,   -1.        ,   -0.95001221,   -0.7199707 ,
         -0.90002441, -200.63995361, -200.4899292 , -200.58996582,
         27.86889648,   -0.67004395])
 message: 'Maximum number of iterations has been exceeded.'
    nfev: 120336
     nit: 2000
    njev: 10028
  status: 1
 success: False
       x: array([ 533.70959181,  965.57472274, 2150.64582249, 1499.9973562 ,
       1599.54992783,  708.20695366,   85.74426971, 1006.04867674,
        500.00001439,  349.90921623])
SCIPY UNCONSTRAINED
Avg. function value:  -7.96e+03±8.22e+02
Avg. time value (ms):  3.21e+03±6.23e+02
```

The unconstrained optimizer performed much worse than the two linear programming solvers from scipy with regards to both time and accuracy. I printed out one example output from the 20 iterations above.

Problem 2

Quadratic Constrained Solver Comparison (2 algs.)

For the quadratic programming problem I minimized the objective function

$$C(x) = \frac{1}{2}x^T Px + q^T x$$

constrained by the forms

$$Gx \leq h$$
$$Ax = b$$

This standard quadratic form assumes that P is symmetric. Because of this I used a second variable M and defined P as $M^T M$. Where M is defined as the following 10x10 matrix in Python using the numpy array function. M is a randomly generated matrix with numbers spanning from 0 to 200.

In [2]:
```
from numpy import *

M = array([[  80.69613754,  197.54341511,  186.43701167,   98.54964723,
         164.37171838,    9.50579752,   68.145483  ,  154.00383058,
         169.48552714,  172.72380941],
       [  34.70433164,   16.60245971,   76.15190389,   18.15800297,
         121.98984701,  194.74941502,   78.48914043,   24.55111931,
           5.83971525,   21.34164605],
       [  90.45237248,  109.9525822 ,  110.0320875 ,  102.38458083,
         172.42295615,  161.35723648,  160.54410401,   76.91975236,
          25.70611638,  115.44204323],
       [ 158.14914377,  140.10941826,    6.97604024,   43.8955222 ,
         145.93047681,  169.67423241,   54.50121987,   99.39273651,
         168.20215549,  146.73845218],
       [  90.21767006,  142.06945479,  198.43810048,  177.15544106,
         167.29301826,   72.94240731,   53.68668631,  126.99782335,
          28.50310634,   71.66276249],
       [  14.58114404,   46.01701953,  117.31195745,  105.13531344,
           3.60907301,   28.51901317,  189.24300084,  129.21358421,
          45.8422409 ,  178.48013532],
       [   1.73526545,  138.68711512,  144.87741972,  104.96990744,
         197.14148281,   92.1795195 ,  179.73987467,  100.00951112,
          54.96659061,  187.72828915],
       [  75.26125141,  119.98781851,  171.30441459,   36.78236051,
         140.79091813,  171.33580861,   48.52333065,  180.75702528,
          80.90227772,  102.83179743],
       [  47.2959556 ,  164.9578006 ,  123.84237216,  124.33504258,
          84.47609066,   37.04529343,    6.96773132,  180.37474901,
          10.7212393 ,   17.40343904],
       [  87.77507908,  192.56951201,  186.84885636,  167.97170772,
          67.2229434 ,  114.09052636,   21.56437041,    6.86327417,
         131.5040131 ,  137.8434112 ]])
```

Python definition for P:

In [3]:
```
P = dot(M.T, M)
```

Definiton for q (10 x 1):

In [4]:
```
q = dot(array([ 104.96545721,   89.52108804,  131.41880265,   48.0142488 ,
        117.87025338,   71.49959947,  187.47670014,   95.15310958,
        160.26497417,   73.06495801]), M).reshape((10,))
```

The constraint matrix G was synthesized using random values between 0 and 50. The constraint vector h was generated using random values between 0 and 200. I decided not to use any equality constraints. Due to the complexity of the inequality constraints I decided not to enforce the > 0 constraint as advised in the textbook. The references for the algorithm also did not use this constraint.

In [5]:
```
G = array([[ 15.80329383,  37.23892782,  47.7786633 ,   9.19228381,
         40.57733093,   9.90679605,  28.14676156,  39.32781356,
         37.12118315,  15.87228955],
       [ 43.27668213,  26.44885916,  41.42125893,  35.78354661,
         29.94610786,   2.85057763,  30.25055607,  33.45756588,
         46.38999009,  45.60324026],
       [ 39.44298274,  45.06131374,  35.03604878,  42.17292742,
         25.63598569,  24.64210732,   3.70977457,  20.5902921 ,
          1.27951776,  28.67218038],
       [ 45.65260265,  29.45461644,  45.97589935,  15.38474593,
         16.37986728,  36.72102556,  45.93156277,  37.22686868,
         40.22524935,   4.59660548],
       [  4.04383479,  44.73317497,  26.82846191,  15.3392613 ,
         30.79706094,  26.13894597,  13.59064127,  49.76311441,
         25.02106243,   6.30114704],
       [ 41.98628128,  25.9705214 ,  10.28335199,   2.77472287,
          5.7108503 ,  23.45671499,   2.71313764,   3.7822363 ,
         18.61527093,   4.42257031],
       [ 27.76025925,  40.3914235 ,  43.67169401,  34.49917188,
          2.00756052,  10.30928905,  38.53562056,   9.20011528,
         11.82462632,  46.58849645],
       [ 25.61764567,  36.31465585,  45.66559564,  48.96504435,
         25.94750581,  36.26049   ,  24.46752968,  39.84317833,
         16.31109554,  22.66888219],
       [ 34.61476803,  17.50168948,  32.58135685,   0.10132606,
         28.63538219,  42.87073243,  31.79667229,  13.60056144,
         31.79662649,   8.5557422 ],
       [ 44.59563999,   3.25900081,  36.96505858,  19.68861835,
          8.53496148,  21.61004563,  35.50452822,  49.29895152,
          3.30418814,  29.34439322]])
```

In [6]:
```
h = array([  77.92369082,    1.57602341,   24.33257481,  121.40012023,
        114.21005826,  185.20003602,   13.87404591,   82.31918607,
        120.27771735,  103.28880513])
```

I modeled the cost function using the following Python code:

In [7]:
```
class QuadConstrained:

    @staticmethod
    def cost(x):
        return .5 * x.T.dot(P).dot(x) + q.T.dot(x)
```

I tested two methods from the **qpsolvers** package that can be found here: https://github.com/stephane-caron/qpsolvers (https://github.com/stephane-caron/qpsolvers). The implementations in this package were derived from the **quadprog** and **cvxopt** Python packages. In testing both methods I ran the solver 1,000 times on the above problem using the Python code shown below. Note that that the optimizers did not accept initial point parameters so I could not vary them.

In [12]:
```
import stats_generator
import time
import qpsolvers

def test_constrained_quad_solver(method, iters):
    results = []
    for i in range(iters):
        start = time.time()
        res = qpsolvers.solve_qp(P, q, G, h, solver=method)
        end = time.time()
        results.append((QuadConstrained.cost(res), (end-start) * 1000))

    func_vals = list(map(lambda x: x[0], results))
    time_vals = list(map(lambda x: x[1], results))

    return ((stats_generator.StatsGenerator.unbiased_expected_value(func_vals),
             stats_generator.StatsGenerator.std_dev(func_vals)),
            (stats_generator.StatsGenerator.unbiased_expected_value(time_vals),
             stats_generator.StatsGenerator.std_dev((time_vals))))

def print_stats(header, stats):
    print(header)
    print("Avg. function value: ", stats_generator.StatsGenerator.stats_to_string(
stats[0]))
    print("Avg. time value (ms): ", stats_generator.StatsGenerator.stats_to_string
(stats[1]))
    print("\n")
```

Results for the **quadprog** method:

In [13]:
```
stats = test_constrained_quad_solver("quadprog", 1000)
print_stats("QPSOLVERS QUADPROG", stats)
```

```
QPSOLVERS QUADPROG
Avg. function value:  -6.64e+04±8.88e-10
Avg. time value (ms):  4.88e-02±1.69e-02
```

Results for the **cvxopt** method:

In [14]:
```
stats = test_constrained_quad_solver("cvxopt", 1000)
print_stats("QPSOLVERS CVXOPT", stats)
```

```
QPSOLVERS CVXOPT
Avg. function value:  -6.64e+04±2.18e-10
Avg. time value (ms):  8.99e-01±1.24e-01
```

Both methods were able to solve the problem rather quickly. The solutions were relatively consistent in both algorithms, however the **quadprog** method was able to find the solution quicker on average.

Quadratic Unconstrained Solver Comparison (1 alg.)

I modeled the unconstrained version of the quadratic problem above using the following class:

In [16]:
```
class QuadUnconstrained:

    @staticmethod
    def penalty(x):
        penalty = 0
        for i in range(len(x)):
            if x.dot(G[i]) > h[i]:
                penalty += 1e5 * (x.dot(G[i]) - h[i]) ** 2
        return penalty

    @staticmethod
    def cost(x):
        return .5 * x.T.dot(P).dot(x) + q.T.dot(x) + QuadUnconstrained.penalty(x)
```

For the penalty I used the function $(x_i - h_i)^2$ to replicate the constraints as seen above. I ran the unconstrained solver 100 times using the following test code. I used 100 instead of 1,000 because the unconstrained solver took much longer to execute. The indices for the starting point were each drawn from a uniform distribution [0, 100).

In [22]:
```
import numpy
import scipy.optimize

def test_unconstrained(method, iters):
    test_results = []

    for i in range(iters):
        x0 = numpy.random.rand(10) * 100
        start = time.time()
        res = scipy.optimize.minimize(QuadUnconstrained.cost, x0=x0, method=method
)
        if i == 0:
            print("OUTPUT FROM ITERATION:", str(i))
            print(res)
        end = time.time()
        test_results.append((res.fun, (end-start) * 1000))

    func_vals = list(map(lambda x: x[0], test_results))
    time_vals = list(map(lambda x: x[1], test_results))
    return ((stats_generator.StatsGenerator.unbiased_expected_value(func_vals), st
ats_generator.StatsGenerator.std_dev(func_vals)),
            (stats_generator.StatsGenerator.unbiased_expected_value(time_vals), st
ats_generator.StatsGenerator.std_dev((time_vals))))

stats = test_unconstrained("cg", 100)
print()
print()
print_stats("SCIPY UNCONSTRAINED STATISTICS", stats)
```

```
OUTPUT FROM ITERATION: 0
     fun: -66418.38694559212
     jac: array([-104.91699219,  -63.32617188, -101.66113281,  -89.203125  ,
        -74.88085938,   -8.546875  ,  -72.13378906,  -82.70605469,
       -113.41113281, -113.73730469])
 message: 'Desired error not necessarily achieved due to precision loss.'
    nfev: 32841
     nit: 578
    njev: 2735
  status: 2
 success: False
       x: array([ 0.89945467, -0.68431643,  0.21354664, -0.35873425, -0.24046906
,
       -0.25957474, -0.59294514, -0.2582278 , -0.27660128,  0.70369092])


SCIPY UNCONSTRAINED STATISTICS
Avg. function value:  -6.64e+04±5.85e-01
Avg. time value (ms):  6.01e+02±1.24e+02
```

I included an example output from one run iteration above under OUTPUT FROM ITERATION: X. The statistics for the solver can be observed after the line SCIPY UNCONSTRAINED STATISTICS. I tested the scipy conjugate gradient solver as in Problem 1.

The unconstrained solver was able to achieve reasonable solutions, but took much longer to execute than the previous constrained solvers (600 ms on average compared to .048 [quadprog] and .889 [cvxopt]).