Tufts University - Department of Mathematics
Math 87 Homework 2 Solutions

1 Consider the function $f(x) = e^x - xe^x$.

1. **Find the root of f.**

$$f(x) = e^x - xe^x = e^x(1 - x) = 0$$

The root is $x = 1$.

2. **Test Newton's method, the secant method, and bisection for finding this root. For Newton, use initial points $x_0 = \frac{1}{2}, 2, 10, -\frac{1}{2}, -5$. For secant, use initial points $(x_0, x_1) = (0, 2), (0, 10), (-1, 2), (-5, 5), (-10, 2)$. For bisection, use $[x_L, x_R] = [0, 2], [-5, 5], [-10, 2], [-1, 2], [0, 1]$. Report the iteration count and approximation to the root for each method. You may use the SciPy optimization library.**

The following experiments are done with the max number of iterations set to 100.

x_0	iters	x_*
1/2	7	1.0
2	7	1.0
10	17	1.0
$-1/2$	100	-105.7953
-5	100	-107.9896

Table 1: Newton's Method

(x_0, x_1)	iters	x_*
$(0, 2)$	13	1.0
$(0, 10)$	100	NaN
$(-1, 2)$	100	-73.3405
$(-5, 5)$	100	-75.7921
$(-10, 2)$	100	-80.1959

Table 2: Secant Method

$[x_L, x_R]$	iters	x_*
$[0, 2]$	1	1.0
$[-5, 5]$	43	1.0
$[-10, 2]$	43	0.9999
$[-1, 2]$	41	0.9999
$[0, 1]$	32555	1.0

Table 3: Bisection Method

Newton's Method

```
import numpy as np
from scipy import optimize

#start point
x0 = -10

#f and derivative
def f(x):
  return np.exp(x) - x*np.exp(x)

def df(x):
  return  - x*np.exp(x)

#call newton
optimize.newton(f,x0,fprime=df,maxiter=100,full_output=True)
```

Secant Method

```
import numpy as np
from scipy import optimize

#start points
x0 = 0
x1 = 2
#function
def f(x):
  return np.exp(x) - x*np.exp(x)

#call secant
optimize.newton(f,x0,maxiter=100,x1=x1,full_output=True)
```

Bisection Method

```
import numpy as np
from scipy import optimize

#start points
a = 0
b = 1
#function
def f(x):
  return np.exp(x) - x*np.exp(x)

#call bisect
optimize.bisect(f,a,b,full_output=True)
```

3. **How do the initial parameters impact the success in finding the root for the three methods?**

   The start parameters have significant effect on the performance of Newton and secant, while bisection appears to converge with any initial parameters. Secant and Newton both need to have initial data very close to the root in order to converge.

4. **Sketch or plot the function f. Use a geometric argument to explain why some initial parameters are not successful for one or more of the methods. What happens as $x \to -\infty$?**

   From the plot, we can see that as $x \to -\infty$, $f \to 0$. If the initial guesses are too "left" of the root, Newton and secant methods will iterate further to the left to find a root, but it's really an asymptote that approaches zero but never hits zero. The secant method also diverges for start points $(x_0, x_1) = (0, 10)$. This is too far to the right of the root. We see that right of the root, the function quickly drops off toward $-\infty$. The iterates of the secant method go too far to the right, and quickly blow up.

Plot the function

```
import numpy as np
import matplotlib.pyplot as plt

#domain
x = np.linspace(-3, 3, 1000)

#function
def f(x):
  return np.exp(x) - x*np.exp(x)

#plot
plt.plot(x,f(x))
plt.title('Plot of $f(x) = e^x - x e^x$')
plt.xlabel('$x$')
plt.ylabel('$f(x)$')
#change x,y axis location
plt.axhline(y=0, color = 'k')
plt.axvline(x=0, color = 'k')
#turn grid on
plt.grid(True)
plt.show()
```

5. **This is an example as to why a "stopping criteria" to stop iterating at the kth step should NOT be checking $f(x_k) \approx 0$. How does your answer from (d) justify that? What is a better termination condition to determine convergence?**

   At first thought, checking that $f(x_k) <$ tolerance where tolerance is close to zero (maybe something like 10^{-8}) would be a sufficient stopping condition for root finding. However, the asymptotic behavior of the function, where $f \to 0$ as $x \to -\infty$, shows that this is not sufficient for determining whether the method has found a root. For example, let's say $x_k = -22$. Then $f(-22) = 6.4 \times 10^{-9}$, which meets the criteria, and the method would declare the root -22.

But we know this is nowhere close to the root. This means that checking that f is close to zero is not a good way to determine convergence.

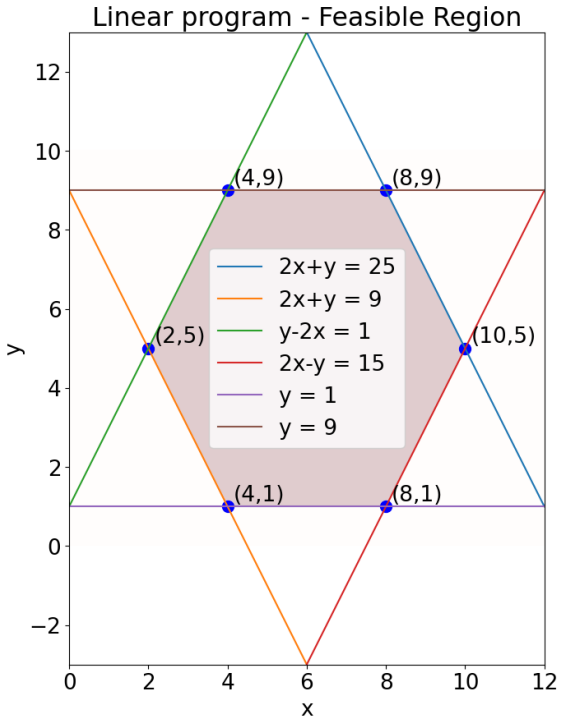
A better way to check for convergence would be to examine the actual sequence of iterates that the method generates, $\{x_0, x_1, x_2, ..., x_k\}$ where $x_k \to x_*$ as $k \to \infty$. We could check the difference between x_k and x_*, but we might not know x_*. However, we do know that with each iteration, we expect the iterates to be getting closer to x_*, which means that the difference between successive iterates should be getting smaller. With this reasoning, a better stopping condition would be to check if $|x_{k-1} - x_k|$ is close to zero, i.e., $|x_{k-1} - x_k| <$ tolerance, where tolerance should be very small.

2 Linear Program

Consider the optimization problem, max $f(x, y) = x + 2y$ subject to:

$$\begin{aligned} y &\leq 9 \\ -y &\leq -1 \\ 2x + y &\leq 25 \\ -2x - y &\leq -9 \\ -2x + y &\leq 1 \\ 2x - y &\leq 15. \end{aligned}$$

1. **Draw the feasible region. Label the boundary curves and corner points.**



2. **Find the maximum value of f and the point where it occurs.**

   The function to optimize is linear, so there are no critical points in the interior to check for a possible max. Furthermore, the function restricted to any of the boundary curves will be linear, so the extrema on the boundary curves is only possible if the value along the boundary

is constant. The max of a linear function on a closed interval will occur at an endpoint, so only the corners of the feasible region need to be checked.

$$
\begin{aligned}
f(4,1) &= 6 \\
f(8,1) &= 10 \\
f(2,5) &= 12 \\
f(10,5) &= 20 \\
f(4,9) &= 22 \\
\boxed{f(8,9) &= 26 \quad \textbf{MAX}}
\end{aligned}
$$

The maximal value is attained at $(8, 9)$.

3. **Verify your answer using SciPy.**

   Re-formulate the problem in matrix form, max $\boldsymbol{c}^T\boldsymbol{x}$ such that $A\boldsymbol{x} \leq \boldsymbol{b}$.

$$
\max \quad f(x,y) = \begin{bmatrix} 1 & 2 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix} \quad \text{such that}
$$

$$
\begin{bmatrix} 0 & 1 \\ 0 & -1 \\ 2 & 1 \\ -2 & -1 \\ -2 & 1 \\ 2 & -1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix} \leq \begin{bmatrix} 9 \\ -3 \\ 25 \\ -12 \\ 1 \\ 15 \end{bmatrix}
$$

The `linprog` function from SciPy takes in $-\boldsymbol{c}^T = [-1, -2]$ as the coefficient matrix, as well as A and $\boldsymbol{b}$. The code return a max of -26.0 at point $(8.000e+00, 9.000e+00)$, which is confirming the result from (b).

Linear Program

```
from scipy.optimize import linprog
import numpy as np

#coefficient vector
c = np.array([1,2])

#LHS constraint matrix
A = np.array([[0,1],[0,-1],[2,1],[-2,-1],[-2,1],[2,-1]])

#RHS constraint vector
b = np.array([9, -1, 25, -9, 1, 15])

#call linear program and print
res=linprog((-1)*c,A_ub=A,b_ub=b)
print(res)
```

3 Some very Type A bakers

A bakery wants to sell forty five Valentine's Day gift bags. They have decided to offer two types of bags: Bags of type A will contain four cupcakes and two cookies, and bags of type B will contain two cupcakes and five cookies. Baskets of type A will be sold for \$12 and baskets of type B will be sold for \$16. The bakery has 90 cookies and 115 cupcakes in total.

Write the bakery's optimization problem as a primal problem. Solve this to determine how many baskets of both types should be made. If a fractional solution is obtained, round down to whole number solutions. What is the maximum profit? You may solve this by drawing the feasible region or using python.

We can rewrite this as a linear program as follows. Let w be the number of baskets of type A to be produced and B be the number of baskets of type B. We know that the profit is given by $p = 12w + 16c$. We also know that the number of cookies is bounded by $2w + 5c \leq 90$, the cupcakes are bounded by $4w + 2c \leq 115$, and the total number of bags is given by $c + w \leq 45$. Of course, we can only have a positive number of bags. Thus, the linear program is:

$$\begin{aligned} \text{Maximize } & 12w + 16c \\ \text{subject to } & w + c \leq 45 \\ & 2w + 5c \leq 90 \\ & 4w + 2c \leq 115 \\ & c, w \geq 0 \end{aligned}$$

The feasible region may be plotted by plotting these inequalities in the plane and shading the region bounded by them. Alternatively, using SciPy's `linprog` with input $c = -[12, 16]$, $A = \begin{bmatrix} 1 & 1 \\ 2 & 5 \\ 4 & 2 \end{bmatrix}$ and $b = \begin{bmatrix} 45 \\ 90 \\ 115 \end{bmatrix}$, we get max profit of 426.25 at point $(24.687, 8.1249)$. However, since we need to round to a whole number value, we choose $(24, 8)$, which has profit 416.

Sample code:

Primal Linear Program

```
from scipy.optimize import linprog
import numpy as np


c = np.array([12,16])
A = np.array([[1,1],[2,5], [4,2]])
b = np.array([45,90,115])

res=linprog((-1)*c, A_ub=A, b_ub=b)
print(res)
```

Appendix: code to draw feasible regions

Problem 2

```
%matplotlib notebook
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
plt.rcParams.update({'font.size': 19})

# plot the feasible region
d1 = np.linspace(0,12,500)
d2= np.linspace(0,10,500)
t,b = np.meshgrid(d1,d2)

#fig,ax = plt.subplots(figsize=(20,10))
pl=plt.figure(figsize=(10,10))
ax = plt.axes()

ax.imshow(((b>=1) & (b<=9) & (2*t + b <= 25) & (-2*t - b <= -9) &
           (-2*t + b <= 1) & (2*t - b <= 15)).astype(int),
          extent=(t.min(),t.max(),b.min(),b.max()),
          origin="lower",
          cmap="Reds",
          alpha = 0.2)

# plot the lines defining the constraints
t = np.linspace(0,12,500)

ax.plot(t, (25 - 2*t), label="2x+y=25")
ax.plot(t, (9 - 2*t), label="2x+y=9")
ax.plot(t, (1 + 2*t), label="y-2x=1")
ax.plot(t, (-15 + 2*t), label="2x-y=15")
ax.plot(t, np.ones(len(t)), label="y=1")
ax.plot(t, 9*np.ones(len(t)), label="y=9")
ax.set_ylim(ymin=-3, ymax=13)

ax.legend()
ax.set_title("Linear program - Feasible Region")
ax.set_xlabel("x")
ax.set_ylabel("y")

def ann_pt(x,y):
  s = f"({x},{y})"
  ax.annotate(s,xy=(x,y),xytext=(5,5),textcoords='offset points')

ax.scatter(4, 1, s=100,color="blue")
ax.scatter(8, 1, s=100,color="blue")
ax.scatter(2, 5, s=100,color="blue")
ax.scatter(10,5, s=100,color="blue")
ax.scatter(4, 9, s=100,color="blue")
ax.scatter(8, 9, s=100,color="blue")
ann_pt(4, 1)
ann_pt(8, 1)
ann_pt(2, 5)
ann_pt(10,5)
ann_pt(4, 9)
ann_pt(8, 9)
```