In [41]:

```
# ECON 289 Problem set 1
# Instructor: Ben Brooks
# Spring 2023

# This problem set has a series of cells with different programming tasks.
# It will teach you how to solve linear programs using Gurobi. You will be asked to
# run code that I have written, and also add and run your own code. Add your code between
# that look like this:

# ----------------------------------------

# To complete the problem set, add your own code, run all the cells, and then submit a copy of the
# notebook on canvas. The easiest way to do so is to select "print preview" from the
# file menu, and then save the new page that opens as a pdf document.

# Please work together to complete the problem set. Also, remember, Google
# is your friend. Only ask me for help after you have looked for the
# answer on stack overflow.

# First, install Gurobi and set up an academic license using grbgetkey.
# Then install the gurobipy module by entering
# python -m pip install gurobipy in the terminal.

# Code in Jupyter notebook is broken up into cells. You can run each cell individually.
# Once you have correctly installed Gurobi and gurobi-py, you should be able to load
# gurobi using the following commends. Run this cell now.
import gurobipy as gp
from gurobipy import GRB

# Notice that each line in Python is a separate command.

# One of the most useful commands in Python is "print", which simply displays a message.
print("Success! You have installed Gurobi and gurobipy.")

# Now you add a line that prints "Hello world!" Brian Kernighan will be proud of you.

# ----------------------------------------

print("Hello world!")

# ----------------------------------------
```

```
Success! You have installed Gurobi and gurobipy.
Hello world!
```

In [42]:
```python
# Now we can create a model using the following code.

model = gp.Model()

# The Gurobi model has various parameters. The Method parameter controls
# which algorithms are used to solve the LP. Here I will set the method to the
# barrier (interior point) algorithm. You can find a list of parameters and possible
# values on Gurobi's website.

model.Params.Method = 2 # Barrier algorithm

# There is another parameter called "Crossover" which controls
# whether the barrier algorithm switches over to simplex when it gets close to
# an optimum. Add code below to disable crossover and run the cell.

# ----------------------------------------

model.Params.Crossover = 0 # turn off crossover

# ----------------------------------------
```

```
Set parameter Method to value 2
Set parameter Crossover to value 0
```

In [43]:
```python
# Now let's start adding variables to our model. My code will add a variable x:

x = model.addVar()

# Now you add code below to create another variable called "y" and run this cell.


# ----------------------------------------

y = model.addVar()

# ----------------------------------------
```

In [44]:
```
# Mathematical expressions involving variables can be built in the natural way.
# Following the example in class, I will add the constraint that x+2y<=1 to our model:
c1 = model.addConstr(x+2*y<=1)
# I have assigned the constraint to the variable c1, so we can later look at properties
# of the constraint.

# Now you add the constraint that 2x+y <= 1, and run the cell.

# -----------------------------------------
c2 = model.addConstr(2*x+y<=1)
# -----------------------------------------
```

In [45]:
```
# Now that we have our two variables and two constraints, we can set
# the objective.
model.setObjective(2*x+7*y, GRB.MINIMIZE)

# Actually, I made a mistake; I meant to set the objective to maximize x+y.
# Please correct the mistake by adding another line below with the right
# objective and direction, and run the cell.

# -----------------------------------------

model.setObjective(x+y, GRB.MAXIMIZE)

# -----------------------------------------
```

In [46]:
```
# Each type of object in Gurobi, e.g., a model, a variable, a constraint,
# has attributes, which you can query. For example, whether we maximize or minimize
# is described by the ModelSense attribute. My code below reports the current value of
# ModelSense. Incidentally, this code shows how the print comand can be used to format
# combinations of text and numbers.

print(f'The current objective sense is {model.ModelSense}, where 1 indicates minimize and -1 indicates ma

# Now you write a line that prints the number of constraints in the model. (Hint:
# Look at the Attributes section on the gurobi documentation, https://www.gurobi.com/documentation/9.5/re

# ----------------------------------------

print(f'The number of constraints is {model.NumConstrs}.')

# ----------------------------------------
```

```
The current objective sense is 1, where 1 indicates minimize and -1 indicates maximize.
The number of constraints is 0.
```

In [47]:
```
# Do you notice something odd about the number of constraints? Gurobi doesn't update the number of consti
# variable until you "update" the model. With the python API, this is done automatically when it is neede
# and you usually don't need to worry about it. But try running the model.update() method and then query
# the number of constraints:

# ----------------------------------------

model.update()
print(f'The number of constraints is {model.NumConstrs}.')

# ----------------------------------------
```

```
The number of constraints is 2.
```

In [49]:
```
# Now we are ready to solve the model! This line will run the optimization routine:
model.optimize()

# After it finishes, I have written code to print the value of the solution.
# Notice how I am querying the attribute "X" from each of the variables,
# and printing the value to 3 decimal places.
print(f'\nI solved the model and found the optimal solution (x,y)=({x.X:.3f},{y.X:.3f}).')

# You add a line below that prints the attribute "Pi" from each of the constraints,
# which is the optimal Lagrange multiplier. Then run this cell.

# ----------------------------------------
print(f'\nThe Lagrange multiplier on the constraint x+2y<=1 is {c1.Pi:.3f}.')
print(f'\nThe Lagrange multiplier on the constraint 2x+y<=1 is {c2.Pi:.3f}.')

# ----------------------------------------
```

```
Gurobi Optimizer version 9.5.2 build v9.5.2rc0 (mac64[x86])
Thread count: 4 physical cores, 8 logical processors, using up to 8 threads
Optimize a model with 2 rows, 2 columns and 4 nonzeros
Model fingerprint: 0xa95902f3
Coefficient statistics:
  Matrix range     [1e+00, 2e+00]
  Objective range  [1e+00, 1e+00]
  Bounds range     [0e+00, 0e+00]
  RHS range        [1e+00, 1e+00]

I solved the model and found the optimal solution (x,y)=(0.333,0.333).

The Lagrange multiplier on the constraint x+2y<=1 is 0.333.

The Lagrange multiplier on the constraint 2x+y<=1 is 0.333.
```

In [52]:

```
# Now that you have learned the basics, please set up and solve the production example
# from Dorfman, Samuleson, and Solow (1958) that we discussed in class. Please print out
# the optimal numbers of cars and trucks, and also the Lagrange multipliers on the capacity
# constraints for each of the production lines. Set it all up in this cell and run it at one go.

# -----------------------------------------

model=gp.Model()
model.Params.Method=2
model.Params.Crossover=0
x1=model.addVar()
x2=model.addVar()
c1=model.addConstr(100>=0.004*x1+0.00286*x2)
c2=model.addConstr(100>=0.003*x1+0.006*x2)
c3=model.addConstr(100>=0.00444*x1)
c4=model.addConstr(100>=0.00667*x2)
model.setObjective(300*x1+250*x2,GRB.MAXIMIZE)
model.optimize()

print(f'\nThe optimal number of cars is {x1.X:.3f} and the optimal number of trucks is {x2.X:.3f}).')
print(f'\nThe optimal Lagrange multiplier on 100>=0.004*x1+0.00286*x2 is {c1.Pi:.3f}.')
print(f'\nThe optimal Lagrange multiplier on 100>=0.003*x1+0.006*x2 is {c2.Pi:.3f}.')
print(f'\nThe optimal Lagrange multiplier on 100>=0.004444*x1 is {c3.Pi:.3f}.')
print(f'\nThe optimal Lagrange multiplier on 100>=0.00667*x2 is {c4.Pi:.3f}.')


# -----------------------------------------
```

```
Set parameter Method to value 2
Set parameter Crossover to value 0
Gurobi Optimizer version 9.5.2 build v9.5.2rc0 (mac64[x86])
Thread count: 4 physical cores, 8 logical processors, using up to 8 threads
Optimize a model with 4 rows, 2 columns and 6 nonzeros
Model fingerprint: 0xcb3b9091
Coefficient statistics:
  Matrix range     [3e-03, 7e-03]
  Objective range  [2e+02, 3e+02]
  Bounds range     [0e+00, 0e+00]
  RHS range        [1e+02, 1e+02]
Presolve time: 0.00s
Presolved: 2 rows, 4 columns, 6 nonzeros
Ordering time: 0.00s
```

```
Barrier statistics:
 AA' NZ     : 1.000e+00
 Factor NZ  : 3.000e+00
 Factor Ops : 5.000e+00 (less than 1 second per iteration)
 Threads    : 1

                 Objective                Residual
Iter       Primal          Dual         Primal    Dual     Compl     Time
   0   2.23419283e+07  4.69781364e+06  0.00e+00 0.00e+00  4.06e+06     0s
   1   9.95762932e+06  6.01250036e+06  0.00e+00 0.00e+00  5.74e+05     0s
   2   7.94787202e+06  7.69756646e+06  0.00e+00 0.00e+00  4.18e+04     0s
   3   7.73072610e+06  7.71684456e+06  0.00e+00 2.27e-13  2.31e+03     0s
   4   7.73022185e+06  7.73020711e+06  0.00e+00 0.00e+00  2.46e+00     0s
   5   7.73022049e+06  7.73022049e+06  0.00e+00 0.00e+00  2.46e-06     0s
   6   7.73022049e+06  7.73022049e+06  1.36e-12 1.14e-13  2.46e-12     0s

Barrier solved model in 6 iterations and 0.02 seconds (0.00 work units)
Optimal objective 7.73022049e+06


The optimal number of cars is 20363.165 and the optimal number of trucks is 6485.084).

The optimal Lagrange multiplier on 100>=0.004*x1+0.00286*x2 is 68093.385.

The optimal Lagrange multiplier on 100>=0.003*x1+0.006*x2 is 9208.820.

The optimal Lagrange multiplier on 100>=0.004444*x1 is 0.000.

The optimal Lagrange multiplier on 100>=0.00667*x2 is 0.000.
```

In [54]:
```
# Also print out the slack in each of the constraints. Does the solution you found satisfy
# complementary slackness for the primal constraints and the dual variables (i.e., Lagrange
# multipliers on the primal constraints)?

# ----------------------------------------
print(f'\nThe slack in 100>=0.004*x1+0.00286*x2 is {c1.Slack:.3f}.')
print(f'\nThe slack in 100>=0.003*x1+0.006*x2 is {c2.Slack:.3f}.')
print(f'\nThe slack in 100>=0.004444*x1 is {c3.Slack:.3f}.')
print(f'\nThe slack in 100>=0.00667*x2 is {c4.Slack:.3f}.')

print('\nYes, complementary slackness is satisfied: The multiplier is non-zero only when there is no slac

# ----------------------------------------
```

```
The slack in 100>=0.004*x1+0.00286*x2 is 0.000.

The slack in 100>=0.003*x1+0.006*x2 is 0.000.

The slack in 100>=0.004444*x1 is 9.588.

The slack in 100>=0.00667*x2 is 56.744.

Yes, complementary slackness is satisfied: The multiplier is non-zero only when there is no slack in the
constraint.
```

In [55]:
```
# Now that we've covered the basics, let's get into a somewhat meatier problem.
# We'll solve a version of the optimal transport problem. There are n warehouses and m stores.
# They are ordered in terms of ease of access. Since I want to solve this model for a general
# n and m, the first thing we will do is set those parameters:
n = 10
m = 5

# Next, we will set up an index set that we will use in setting up the rest of the model:
model = gp.Model()

W = range(0,n)

# You set up a range called "S" for the stores on the next line

# ----------------------------------------
S = range(0,m)
# ----------------------------------------
```

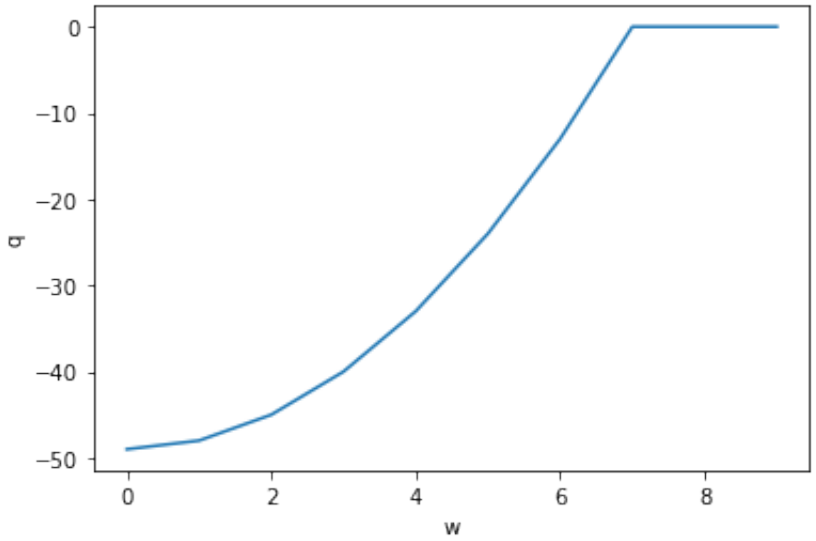
```
# Now, we want to add a variables that indicates how much inventory flows from
# warehouse w to store s. We can do this with the following line:
f = model.addVars(W,S)

# f is a data structure called a "dictionary" that maps each pair w in W and s in S to
# a variable f[w,s]. Pretty neat! Notice how I told Gurobi to make the lower bound zero.
# It actually does this anyway by default.

# Each warehouse has a supply, which is w/(n-1). I am going to store this information
# in a new dictionary:
supply = {w:w/(n-1) for w in W}
print(f'The supply of the second warehouse is {supply[1]}')
# Notice how I used the "for" keyword to assign values for each warehouse.

# Now you create a dictionary for the store demands, where each store s demands (m-s)/m.

# ----------------------------------------
demand = {s:(m-s)/m for s in S}
# ----------------------------------------

# Now we need some constraints in the model. I will add constraints that the
# flow out of each warehouse not exceed its supply. Notice how I use the "sum"
# command to rapidly build linear expressions. I also put a semicolon at the
# end of the line to suppress the output.
supplyConstrs=model.addConstrs(sum(f[w,s] for s in S)<= supply[w] for w in W);

# Next, add a constraint that the net flow into each store is at least its
# demand, which is s/(m-1). Call these constraints "demandConstrs".
# Leave off the semicolon at the end so we see all the garbage
# that Gurobi sends back as we add constraints.

# ----------------------------------------

demandConstrs = model.addConstrs(sum(f[w,s] for w in W)>=demand[s] for s in S)
# ----------------------------------------

# The next task is to create the objective. It turns out that moving flow
# from w to s has a kind of complicated form. If w > s, then it costs 2*w, and if
# s >= w, then it costs s*s. Let's encode this using a function:
def bensCost(w,s):
    if (w>s):
        return 2*w
```

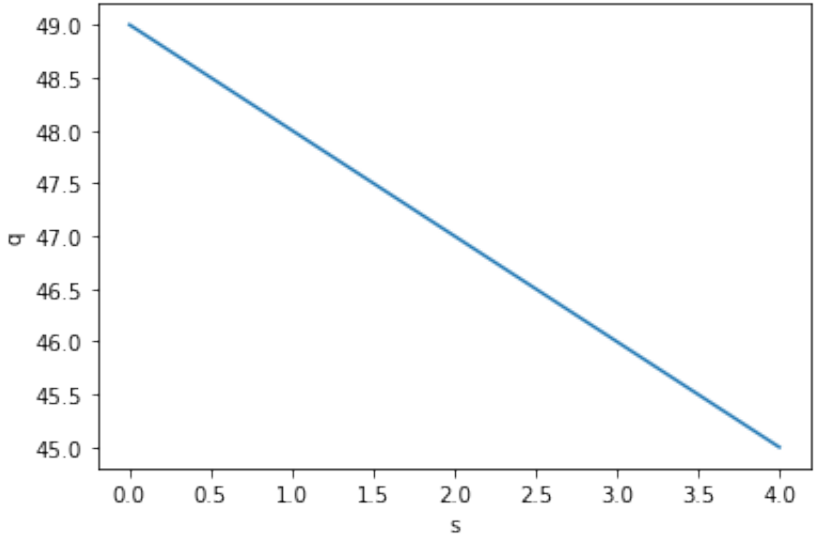
```
    else:
        return s*s
    return 0

# Notice how I defined this function using conditional statements. Now I can use it to set
# the objective, which is to minimize cost:
model.setObjective(sum(bensCost(w,s)*f[w,s] for w in W for s in S),GRB.MINIMIZE)

# But wait! I actually made a mistake. I wanted the cost to be w*w-s if w>s and s+2*w if w<=s.
# Please create a new function called "yourCost" that implements the right cost function, and
# set the objective. Then optimize the model.

# ------------------------------------------
def yourCost(w,s):
    if (w>s):
        return w*w-s
    else:
        return s+2*w
    return 0

model.setObjective(sum(yourCost(w,s)*f[w,s] for w in W for s in S),GRB.MINIMIZE)
model.optimize()
# ------------------------------------------
```

```
The supply of the second warehouse is 0.1111111111111111
Gurobi Optimizer version 9.5.2 build v9.5.2rc0 (mac64[x86])
Thread count: 4 physical cores, 8 logical processors, using up to 8 threads
Optimize a model with 15 rows, 50 columns and 100 nonzeros
Model fingerprint: 0xc9371f79
Coefficient statistics:
  Matrix range     [1e+00, 1e+00]
  Objective range  [1e+00, 8e+01]
  Bounds range     [0e+00, 0e+00]
  RHS range        [1e-01, 1e+00]
Presolve removed 1 rows and 5 columns
Presolve time: 0.00s
Presolved: 14 rows, 45 columns, 90 nonzeros

Iteration    Objective       Primal Inf.    Dual Inf.      Time
       0    0.0000000e+00   3.000000e+00   0.000000e+00      0s
      19    7.7666667e+01   0.000000e+00   0.000000e+00      0s

Solved in 19 iterations and 0.01 seconds (0.00 work units)
Optimal objective  7.766666667e+01
```

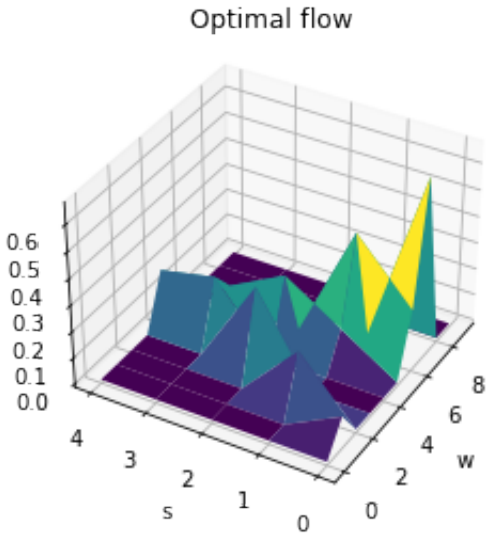
In [61]:
```
# We've solved our model. Now what? We want to visualize the solution
# and see what it means! To do this, we will load some additional libraries.
# matplotlib has plotting tools and numpy has numerical routines. They will
# work together to create pretty pictures.

import matplotlib.pyplot as plt

import numpy as np

# This line creates a figure.
fig = plt.figure()

# I want to create a plot of the optimal multiplier on each warehouse's supply.
# To do so, first I store the optimal multipliers in a numpy array, and then plot it.
Q = np.array([supplyConstrs[w].Pi for w in W])
fig, ax = plt.subplots()
plt.plot(W,Q)
ax.set_xlabel('w')
ax.set_ylabel('q')
plt.show()

# Now you create your own plot showing the optimal multipliers on each store's demand.

# ------------------------------------------
P = np.array([demandConstrs[s].Pi for s in S])
fig, ax = plt.subplots()
plt.plot(S,P)
ax.set_xlabel('s')
ax.set_ylabel('q')
plt.show()

# ------------------------------------------
```

```
<Figure size 432x288 with 0 Axes>
```


0
−10
−20
−30
−40
−50
q
0
2
4
6
8
w


49.0
48.5
48.0
47.5
47.0
46.5
46.0
45.5
45.0
q
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
s

In [38]:
```
# This works well enough for one-dimensional plots. But I'd
# also like to visualize the optimal flow, which is two-dimensional.
# For this, we will load another library:

# This complicated line creates a two-dimensional numpy array,
# which is an array of arrays!

fig = plt.figure()
ax = plt.axes(projection='3d')

optFlow = np.array([[f[w,s].X for w in W] for s in S])
X, Y = np.meshgrid(W,S)

ax.plot_surface(X, Y, optFlow, cmap='viridis')

ax.set_xlabel('w')
ax.set_ylabel('s')
ax.set_title('Optimal flow');
ax.view_init(35,210)
```



In [62]:
```
# But wait! I realized I made a mistake, and in fact I wanted the demand to be
# just 1/m at each store. Please repeat the computation of the cost minimizing flow,
# and plot the optimal flow below.

# ----------------------------------------
```

```
n = 10
m = 5

model = gp.Model()

W = range(0,n)
S = range(0,m)

f = model.addVars(W,S)

supply = {w:w/(n-1) for w in W}
demand = {s:1/m for s in S}

supplyConstrs=model.addConstrs(sum(f[w,s] for s in S)<= supply[w] for w in W);
demandConstrs = model.addConstrs(sum(f[w,s] for w in W)>=demand[s] for s in S);

def yourCost(w,s):
    if (w>s):
        return w*w-s
    else:
        return s+2*w
    return 0

model.setObjective(sum(yourCost(w,s)*f[w,s] for w in W for s in S),GRB.MINIMIZE)
model.optimize()

fig = plt.figure()
ax = plt.axes(projection='3d')

optFlow = np.array([[f[w,s].X for w in W] for s in S])
X, Y = np.meshgrid(W,S)

ax.plot_surface(X, Y, optFlow, cmap='viridis')

ax.set_xlabel('w')
ax.set_ylabel('s')
ax.set_title('Optimal flow');
ax.view_init(35,210)

# ----------------------------------------
```

```
Gurobi Optimizer version 9.5.2 build v9.5.2rc0 (mac64[x86])
Thread count: 4 physical cores, 8 logical processors, using up to 8 threads
Optimize a model with 15 rows, 50 columns and 100 nonzeros
Model fingerprint: 0x3c6866d1
Coefficient statistics:
  Matrix range     [1e+00, 1e+00]
  Objective range  [1e+00, 8e+01]
  Bounds range     [0e+00, 0e+00]
  RHS range        [1e-01, 1e+00]
Presolve removed 1 rows and 5 columns
Presolve time: 0.01s
Presolved: 14 rows, 45 columns, 90 nonzeros

Iteration    Objective       Primal Inf.    Dual Inf.      Time
       0    0.0000000e+00   1.000000e+00   0.000000e+00      0s
      12    7.5333333e+00   0.000000e+00   0.000000e+00      0s

Solved in 12 iterations and 0.01 seconds (0.00 work units)
Optimal objective  7.533333333e+00
```