In [2]:
```
# ECON 289 Problem set 2
# Instructor: Ben Brooks
# Spring 2023

# This problem set has a series of cells with different programming tasks. 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.
```

In [3]:
```
# Insert code to load gurobi, numpy, and matplotlib pyplot.

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

import gurobipy as gp
from gurobipy import GRB

import matplotlib.pyplot as plt

import numpy as np

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

# We are going to add the following code which will help us create
# fancy 3d graphs:

from mpl_toolkits import mplot3d
%matplotlib notebook
```

In [4]:

```
# In class we studied correlated equilibria of BoS.
# Now I'd like you to compute the set of all correlated equilibrium payoffs.

# Start by creating a gurobi model, setting parameter values,
# and adding variables to represent the  probability of each of
# the four outcomes (B,B), (S,S), (B,S), (S,B), and add
# a constraint so that these are in fact probabilities.

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

model = gp.Model()
model.Params.Method = 2 # Barrier algorithm
model.Params.Crossover = 0 # Disable crossover

muBB = model.addVar()
muBS = model.addVar()
muSB = model.addVar()
muSS = model.addVar()

probConstr = model.addConstr(muBB + muBS + muSB + muSS == 1)

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

```
Set parameter Username
Academic license - for non-commercial use only - expires 2024-03-14
Set parameter Method to value 2
Set parameter Crossover to value 0
```

In [5]:

```
# Now add the obedience constraints for the model. In particular,
# for every action "recommended" by the mediator and for every
# possible deviation, there is a constraint that the player not
# gain from the deviation, in expectation.

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

# Obedience for B for player 1
obedBtoS1 = model.addConstr(muBB*(3-0)+muBS*(0-1)>=0)
# Obedience for S for player 1
obedStoB1 = model.addConstr(muSB*(0-3)+muSS*(1-0)>=0)
# Obedience for B for player 2
obedBtoS2 = model.addConstr(muBB*(1-0)+muSB*(0-3)>=0)
# Obedience for S for player 2
obedStoB2 = model.addConstr(muSS*(3-0)+muBS*(0-1)>=0)

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

# Now set the objective to maximize the probability of miscoordination.

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

model.setObjective(muSB+muBS,GRB.MAXIMIZE)

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

# And finally, optimize

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

model.optimize()

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

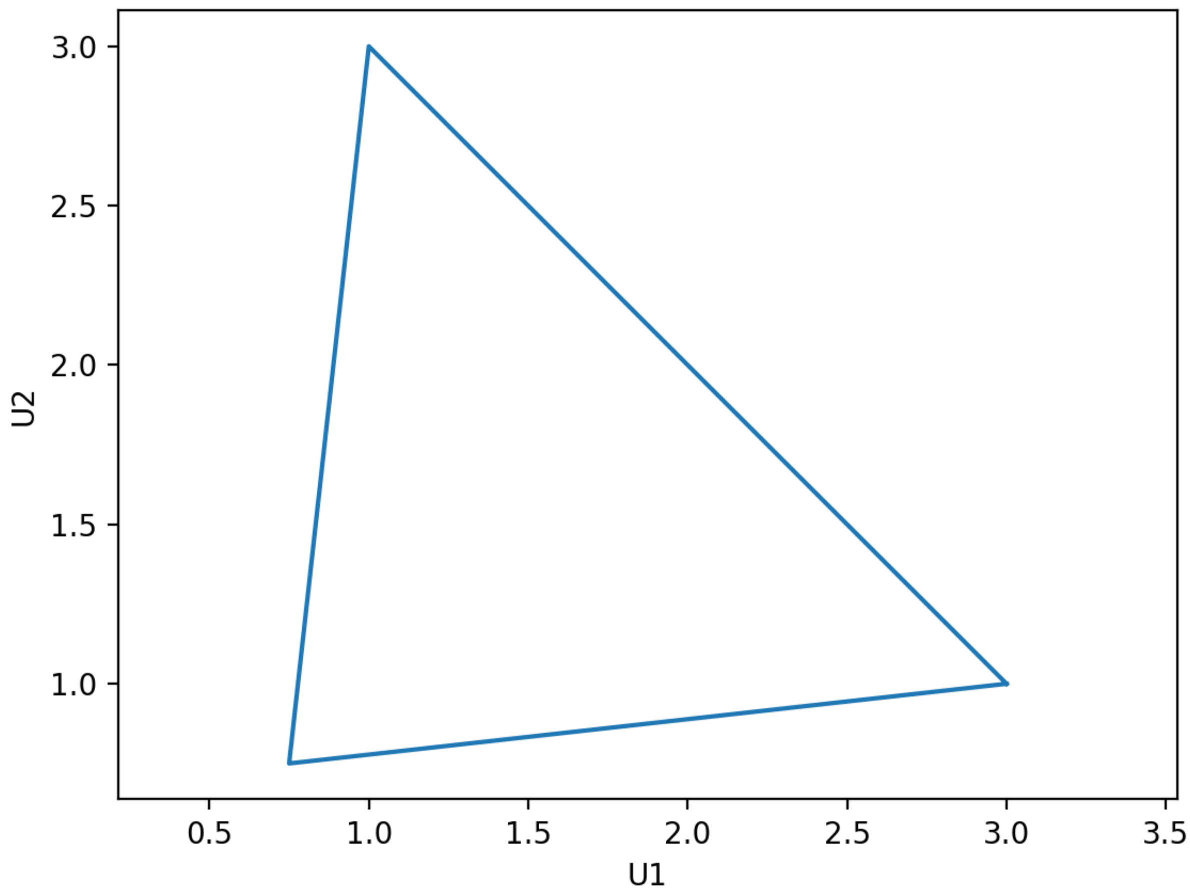
```
Gurobi Optimizer version 9.5.2 build v9.5.2rc0 (mac64[x86])
Thread count: 4 physical cores, 8 logical processors, using up to 8 threads
Optimize a model with 5 rows, 4 columns and 12 nonzeros
Model fingerprint: 0x87470bad
Coefficient statistics:
  Matrix range     [1e+00, 3e+00]
  Objective range  [1e+00, 1e+00]
  Bounds range     [0e+00, 0e+00]
  RHS range        [1e+00, 1e+00]
Presolve removed 1 rows and 1 columns
Presolve time: 0.01s
Presolved: 4 rows, 3 columns, 10 nonzeros
Ordering time: 0.00s

Barrier statistics:
 AA' NZ     : 6.000e+00
 Factor NZ  : 1.000e+01
 Factor Ops : 3.000e+01 (less than 1 second per iteration)
 Threads    : 1

                  Objective                Residual
Iter       Primal          Dual         Primal    Dual     Compl     Time
   0   1.17744397e+00  4.09028728e-01  7.07e-01 4.19e-01  4.61e-01     0s
   1   4.49820687e-01  7.62345849e-01  0.00e+00 0.00e+00  4.46e-02     0s
   2   6.05706196e-01  6.45075381e-01  0.00e+00 0.00e+00  5.62e-03     0s
   3   6.24619889e-01  6.25810713e-01  0.00e+00 2.22e-16  1.70e-04     0s
   4   6.24999633e-01  6.25000824e-01  0.00e+00 8.49e-17  1.70e-07     0s
   5   6.25000000e-01  6.25000001e-01  0.00e+00 0.00e+00  1.70e-10     0s

Barrier solved model in 5 iterations and 0.03 seconds (0.00 work units)
Optimal objective 6.25000000e-01
```

In [6]:
```
# Now print out the probabilities of each
# action. Also print out the multipliers on the obedience
# constraints.

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

print(f'The CE that maximizes the probability of miscoordination is (mu(B,B),mu(B,S),mu(S,B),mu(S,S))=({m
print(f'The optimal multipliers on the obedience constraints are: (probConstr,B to S for 1,S to B for 1,B

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

# Are these numbers the same as what we computed in lecture? Why or why not?
```

```
The CE that maximizes the probability of miscoordination is (mu(B,B),mu(B,S),mu(S,B),mu(S,S))=(0.1875000
002442437,0.5625000003144034,0.062499999318764266,0.18750000012258863).
The optimal multipliers on the obedience constraints are: (probConstr,B to S for 1,S to B for 1,B to S f
or 2, S to B for 2)=(0.6250000008240698,-0.19478750888521262,-0.0843625291512647,-0.04063747416843202,-0
.18021249044025534).
```

In [7]:
```
# Now I want you to do something a bit more complicated. I want you to compute
# the whole set of correlated equilibrium payoffs. First, create expressions U1 and U2
# for the utilities of players 1 and 2.

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

U1 = 3*muBB + muSS
U2 = muBB + 3*muSS

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

# Now we will create a large grid of directions
numDirs = 200
D=range(0,numDirs)
Theta = {d: d*2*3.14/numDirs for d in D}

# I will also create an empty numpy array to store the calculated values of U1 in each direction.
# I create one extra space at the end, for a reason that you'll see in a minute.
u1=np.zeros(numDirs+1)

# Now you create a similar array for U2 called u2:

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

```python
u2=np.zeros(numDirs+1)

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

# Before proceeding, it's prudent to turn off Gurobi's output. Do this by setting the "output" parameter
# to the appropriate value.

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

model.Params.OutputFlag = 0

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

# Now we will use a loop to compute, for each direction, the optimal payoffs
for d in D:
    theta=Theta[d]
    model.setObjective(np.cos(theta)*U1+np.sin(theta)*U2,GRB.MAXIMIZE)
    model.optimize()
    u1[d]=U1.getValue()
    u2[d]=U2.getValue()

u1[numDirs]=u1[0]
u2[numDirs]=u2[0]

# Finally, plot the data you have collected, using similar code as we used in problem set 1.

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

fig, ax = plt.subplots()
plt.plot(u1,u2)
ax.set_xlabel('U1')
ax.set_ylabel('U2')
plt.axis('equal')
plt.show()

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

# What do you notice about the set of correlated equilibrium payoffs? What are its extreme points?
```



In [8]:
```
# Now we will do something a bit more involved. You will compute the revenue
# minimizing BCE of a discretized first price auction. There is a pure common value
# that is uniformly distributed on a uniform grid between 0 and 1. We will parametrize
# the model by the number of values and bids.


# Create a new model. Set the method to barrier, disable crossover, and turn on output.


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

```python
model = gp.Model()

model.Params.Method = 2 # Barrier algorithm
model.Params.Crossover = 0 # Disable crossover
model.Params.OutputFlag = 1 # Enable output

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

# Next, create a variable called "numVals" and set it equal to 51. Then create a new range
# array, called "K", with entries from 0 to numVals.

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

numVals = 51
K=range(0,numVals)

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

# The next task is to create two dictionaries, one called "V" and the other called "B".
# The dictionary V should map k in K into k/(numVals-1). This will be our uniformly spaced grid of common
# values on the interval [0,1]. Then make B map k in K into a uniform grid on the interval [0,0.4].
# This will be large enough for our purposes.

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

V={k:k/(numVals-1) for k in K};
B={k:0.4*k/(numVals-1) for k in K};

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

# Now we will need a function called "payoff" that maps the arguments (v,bi,bj), which will be
# elements of K, respectively, into the payoff of a bidder who bids B[bi], when the other bidder
# bids B[bj], and the value is V[v]. The payoff should be V[v]-B[bi] if B[bi]>B[bj], and 0 if
# B[bi] < B[bj], and there should be a 1/2 chance of winning if the bidders tie.

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

def payoff(v,bi,bj):
    if (bi>bj):
        return V[v]-B[bi]
    elif (bi==bj):
        return 0.5*(V[v]-B[bi])
    return 0
```

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

# Now we are ready to populate the model. Add variables indexed (v,bi,bj) for v in K,
# bi in K, and bj in K.

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

mu = model.addVars(K,K,K)

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

# Next, for each v, add a constraint that the marginal probability of v is 1/numVals.
# Hint: For each v, sum mu across b1 and b2, and set the sum equal to 1/numVals.

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

probConstr = model.addConstrs(sum(mu[v,b1,b2] for b1 in K for b2 in K)==1/numVals for v in K)

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

# Next we need to add the obedience constraints. This is a little tricky, so I'm going to show you how tc
# it with the obedience constraints for bidder 1:

obed1 = model.addConstrs(sum(mu[v,b1,b2]*(payoff(v,b1,b2)-payoff(v,b,b2)) for v in K for b2 in K) >= 0 fc

# Notice that I added a constraint for every recmomended b1 and deviation b. For each (b1,b),
# I summed across (v,b2) the difference in bidder 1's payoff if they bid b1 versus b.
# Now you add the obedience constraints for bidder 2.

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

obed2 = model.addConstrs(sum(mu[v,b1,b2]*(payoff(v,b2,b1)-payoff(v,b,b1)) for v in K for b1 in K) >= 0 fc

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

# Now create expressions for each bidder's payoff and for revenue, and name them U1, U2, and Rev.

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

U1 = sum(mu[v,b1,b2]*payoff(v,b1,b2) for v in K for b1 in K for b2 in K)
U2 = sum(mu[v,b1,b2]*payoff(v,b2,b1) for v in K for b1 in K for b2 in K)
Rev = sum(mu[v,b1,b2]*max(B[b1],B[b2]) for v in K for b1 in K for b2 in K)
```

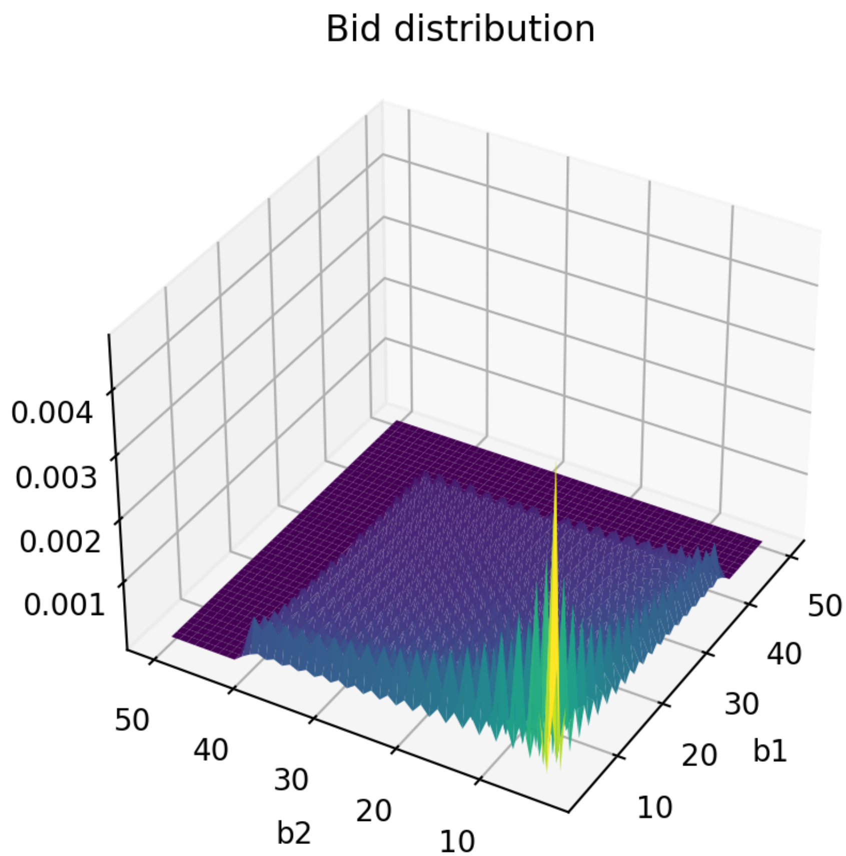
```
# ---------------------------------------------

# Finally, minimize Rev.

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

model.setObjective(Rev,GRB.MINIMIZE)

model.optimize()

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

# What do you get for the approximate value of minimum expected revenue?
```

```
Set parameter Method to value 2
Set parameter Crossover to value 0
Warning for adding constraints: zero or small (< 1e-13) coefficients, ignored
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 5253 rows, 132651 columns and 9128631 nonzeros
Model fingerprint: 0x3d3d1fbc
Coefficient statistics:
  Matrix range     [2e-03, 1e+00]
  Objective range  [8e-03, 4e-01]
  Bounds range     [0e+00, 0e+00]
  RHS range        [2e-02, 2e-02]
Presolve removed 102 rows and 121 columns
Presolve time: 2.88s
Presolved: 5151 rows, 132530 columns, 9123960 nonzeros
Ordering time: 0.00s

Barrier statistics:
 AA' NZ     : 1.297e+06
 Factor NZ  : 2.263e+06 (roughly 50 MB of memory)
 Factor Ops : 2.895e+09 (less than 1 second per iteration)
 Threads    : 4

                  Objective                Residual
Iter       Primal          Dual         Primal    Dual     Compl     Time
   0   3.54466184e+02  0.00000000e+00  6.56e+01 0.00e+00  7.17e-02     6s
   1   1.02039241e+02 -3.42235319e-01  1.90e+01 3.44e-01  2.19e-02     6s
   2   3.54558943e+01 -6.35811902e-01  6.69e+00 1.82e-01  8.01e-03     7s
```

```
   3   1.31181307e+01 -8.50759492e-01  2.53e+00 8.08e-02  3.15e-03     7s
   4   1.42065027e+00 -1.10026174e+00  2.43e-01 4.15e-03  3.76e-04     7s
   5   5.99466613e-01 -8.33072759e-01  7.22e-02 1.76e-03  1.38e-04     7s
   6   2.54040415e-01 -3.09472803e-01  4.07e-05 2.01e-04  3.23e-05     7s
   7   2.22956879e-01  1.19737096e-01  2.28e-15 8.18e-06  5.90e-06     8s
   8   1.96108384e-01  1.42764923e-01  6.59e-15 7.04e-06  3.05e-06     8s
   9   1.80886935e-01  1.51119022e-01  1.29e-14 1.47e-06  1.70e-06     8s
  10   1.74039339e-01  1.57129919e-01  1.07e-14 7.25e-07  9.66e-07     9s
  11   1.70809078e-01  1.59051574e-01  1.09e-14 5.65e-07  6.72e-07     9s
  12   1.64874907e-01  1.59898983e-01  5.81e-14 4.44e-16  2.84e-07     9s
  13   1.63154080e-01  1.60384235e-01  5.56e-14 2.27e-08  1.58e-07     9s
  14   1.61656025e-01  1.60532695e-01  8.29e-14 4.44e-16  6.41e-08    10s
  15   1.60986153e-01  1.60610717e-01  5.53e-13 9.17e-09  2.14e-08    10s
  16   1.60828462e-01  1.60637957e-01  3.30e-13 4.44e-16  1.09e-08    10s
  17   1.60741498e-01  1.60645968e-01  2.53e-13 4.01e-09  5.45e-09    10s
  18   1.60687175e-01  1.60650040e-01  9.28e-14 1.22e-10  2.12e-09    10s
  19   1.60667677e-01  1.60653826e-01  1.48e-13 8.88e-16  7.91e-10    10s
  20   1.60659490e-01  1.60654304e-01  2.38e-12 8.88e-16  2.96e-10    11s
  21   1.60654811e-01  1.60654669e-01  2.40e-11 8.88e-16  8.13e-12    11s
  22   1.60654741e-01  1.60654737e-01  5.42e-13 8.88e-16  2.32e-13    11s

Barrier solved model in 22 iterations and 10.96 seconds (13.37 work units)
Optimal objective 1.60654741e-01
```

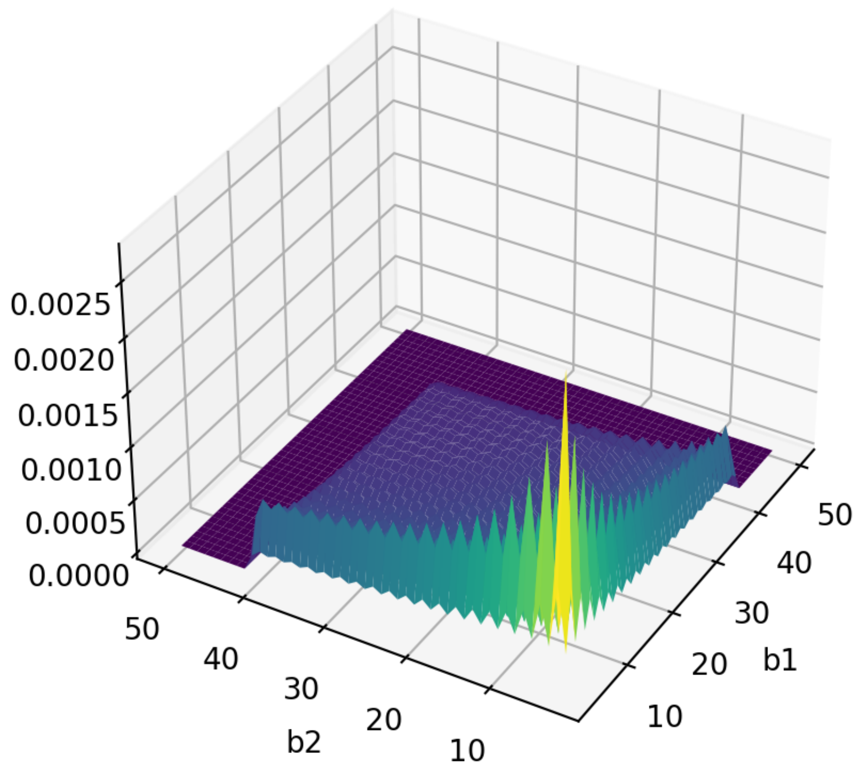
In [9]:

```
# We'll now start exploring the solution. The first task is to plot the joint
# distribution of bids. Create a two-dimensional numpy array whose entries are the
# marginal probabilities of (b1,b2), according to mu.

# Hint: This is similar to how you created the array for the optimal
# flow on problem set 1. But now each entry should be a sum of mu[v,b1,b2] across v in K.

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

K=range(0,numVals)
L=range(3,numVals)
bidDistr = np.array([[sum(mu[v,b1,b2].X for v in K) for b1 in L] for b2 in L])

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

# Now plot the distribution as a surface, as we did with the optimal flow.

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

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

X, Y = np.meshgrid(L,L)

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

ax.set_xlabel('b1')
ax.set_ylabel('b2')
ax.set_title('Bid distribution');
ax.view_init(35,210)

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

# What do you notice about the distribution? What does the support look like? Which bids are
# played with positive probability? Redo the plotting, but at the beginning of the cell,
# redefine K=range(3,numVals), to drop the lowest bids from the plot that have the highest probability,
# in order to gain a clearer view of the support.
```


Bid distribution
0.004
0.003
0.002
0.001
50
40
30
20
10
b2
50
40
30
20
10
b1

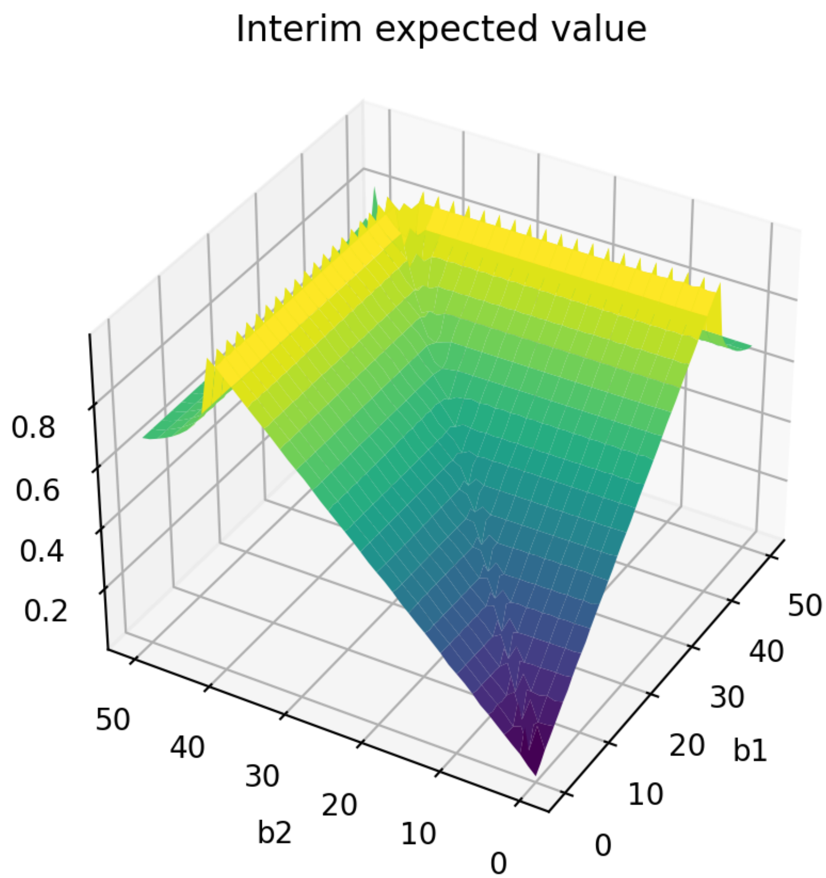
In [10]:
```
# I want to better understand the correlation structure between the bids.
# To that end, let us compute the marginal distribution of each bid. Then
# use these computed marginals to calculate and plot the product of the marginals.
# How does this compare to the actual joint distribution? What does it suggest
# about the correlation structure between the bids?

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

bid1Distr = np.array([sum(mu[v,b1,b2].X for v in K for b2 in K) for b1 in K])
bid2Distr = np.array([sum(mu[v,b1,b2].X for v in K for b1 in K) for b2 in K])
prodDistr = np.array([[bid1Distr[b1]*bid2Distr[b2] for b1 in L] for b2 in L])

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

X, Y = np.meshgrid(L,L)

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

ax.set_xlabel('b1')
ax.set_ylabel('b2')
ax.set_title('Product of the marginals of b2 given b1');
ax.view_init(35,210)

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

Product of the marginals of b2 given b1



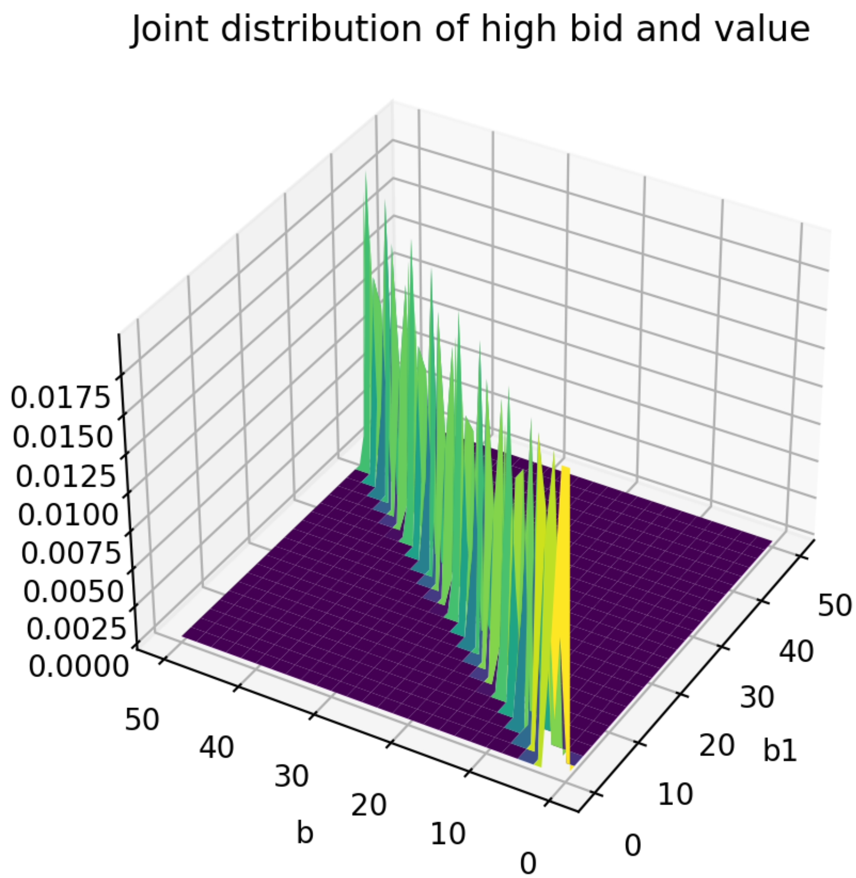
In [11]:

```python
# Let's continue exploring the solution. The next task is to see
# how the expected value is related to the bids. Create a new numpy array,
# called "expVal", that stores the interim expected valuation, conditional on the
# bids (b1,b2). Hint: now each entry of the ray is a ratio of two sums,
# sum( mu[v,b1,b2] * V[v] for v in K)/sum( mu[v,b1,b2] for v in K). Once you have created
# the array, plot it as a surface.

# Hint: Don't forget to restore K to its original definition, if you changed it.

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

K=range(0,numVals)
expVal = np.array([[sum(V[v]*mu[v,b1,b2].X for v in K)/sum(mu[v,b1,b2].X for v in K) for b1 in K] for b2

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

K=range(0,numVals)
X, Y = np.meshgrid(K,K)

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

ax.set_xlabel('b1')
ax.set_ylabel('b2')
ax.set_title('Interim expected value');
ax.view_init(35,210)

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

# What do you notice about the expected value? What does it depend on? How do you interpret
# the expected value for the highest bids?
```


Interim expected value
0.8
0.6
0.4
0.2
50
40
30
20
10
0
b2
50
40
30
20
10
0
b1

In [12]:

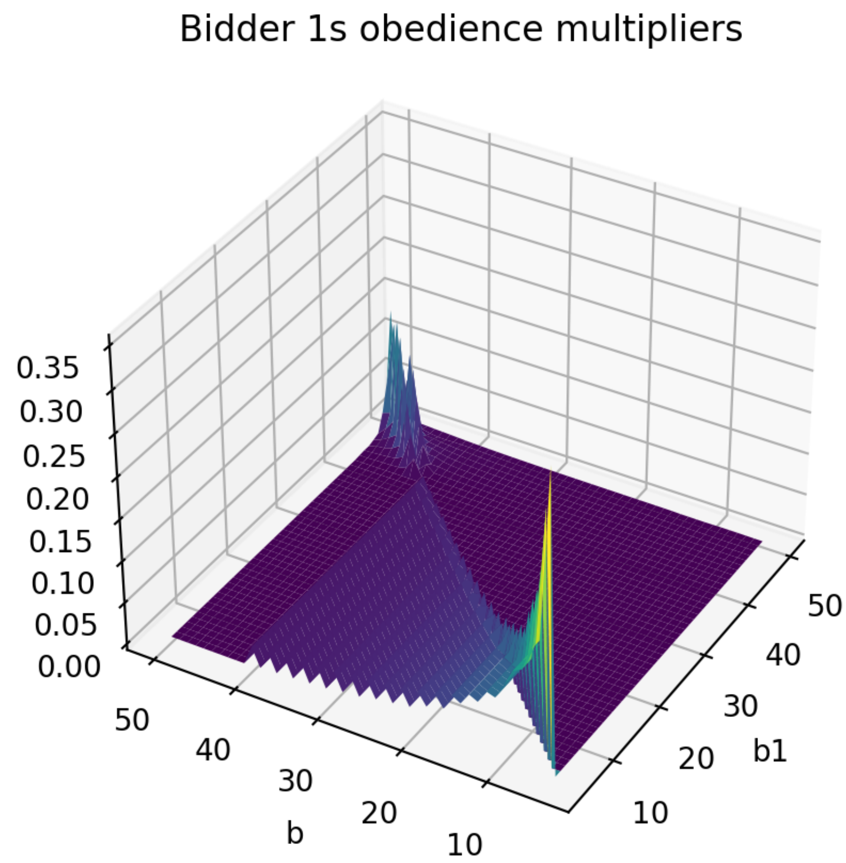
```python
# The previous picture should suggest a particular correlation structure between
# the value and the bids. To gain further insight into this relation, plot the joint
# distribution of the value and the high bid.

# What do you conclude about the correlation structure?

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

valHighBidDistr = np.array([[sum(mu[v,b1,b2].X for b2 in K if b2<=b1) + sum(mu[v,b2,b1].X for b2 in K if

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

X, Y = np.meshgrid(K,K)

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

ax.set_xlabel('b1')
ax.set_ylabel('b')
ax.set_title('Joint distribution of high bid and value');
ax.view_init(35,210)

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



Joint distribution of high bid and value
0.0000
0.0025
0.0050
0.0075
0.0100
0.0125
0.0150
0.0175
b
0
10
20
30
40
50
b1
0
10
20
30
40
50

In [13]:

```python
# Finally, plot the multipliers on bidder 1's obedience constraints.

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

K=range(4,numVals)
obedMult1 = np.array([[obed1[b1,b].Pi for b1 in K] for b in K])

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

X, Y = np.meshgrid(K,K)

ax.plot_surface(X, Y, obedMult1, cmap='viridis')

ax.set_xlabel('b1')
ax.set_ylabel('b')
ax.set_title('Bidder 1''s obedience multipliers');
ax.view_init(35,210)
# ------------------------------------------

# What structure do you notice on the multipliers? Which obedience constraints
# bind? How does the multiplier depend on the recommendation and deviation? You may
# again want to redefine the range to, say, K=range(3,numVals), in order to get a clearer picture.
```



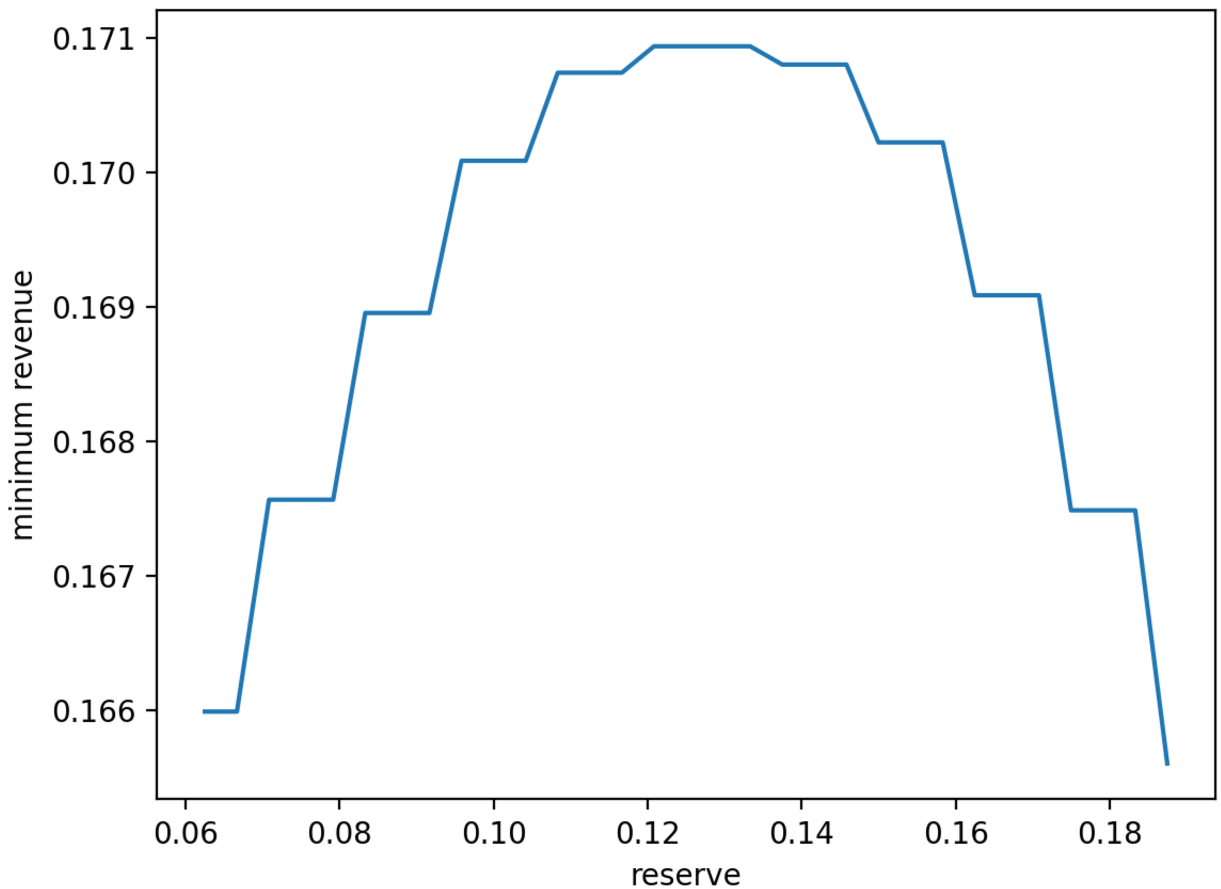
In [14]:

```
# Now repeat the exercise for a first-price auction with a
# reserve price r. Try to determine the value of r that maximizes
# minimum expected revenue.

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




numVals = 31
K=range(0,numVals)
```

```
V={k:k/(numVals-1) for k in K};
B={k:0.4*k/(numVals-1) for k in K};

reserves = np.linspace(1/16,3/16,31)
minrevs= np.zeros(len(reserves))

def payoff(v,bi,bj,r):
    if (bi>bj and B[bi]>=r):
        return V[v]-B[bi]
    elif (bi==bj and B[bi]>=r):
        return 0.5*(V[v]-B[bi])
    return 0


def profit(b1,b2,r):
    if (max(B[b1],B[b2])>=r):
        return (max(B[b1],B[b2]))
    return 0


for l in range(0,len(reserves)):
    r=reserves[l]

    print(f'Calculating for r={r}')

    model = gp.Model()

    model.Params.OutputFlag = 0; # Disable output
    model.Params.Method = 2; # Barrier algorithm
    model.Params.Crossover = 0; # Disable crossover


    mu = model.addVars(K,K,K)

    probConstr = model.addConstrs(sum(mu[v,b1,b2] for b1 in K for b2 in K)==1/numVals for v in K)

    obed1 = model.addConstrs(sum(mu[v,b1,b2]*(payoff(v,b1,b2,r)-payoff(v,b,b2,r)) for v in K for b2 in K)
    obed2 = model.addConstrs(sum(mu[v,b1,b2]*(payoff(v,b2,b1,r)-payoff(v,b,b1,r)) for v in K for b1 in K)

    U1 = sum(mu[v,b1,b2]*payoff(v,b1,b2,r) for v in K for b1 in K for b2 in K)
    U2 = sum(mu[v,b1,b2]*payoff(v,b2,b1,r) for v in K for b1 in K for b2 in K)
    Rev = sum(mu[v,b1,b2]*profit(b1,b2,r) for v in K for b1 in K for b2 in K)
```

```
    model.setObjective(Rev,GRB.MINIMIZE)
    model.optimize()

    minrevs[l]=Rev.getValue()


fig, ax = plt.subplots()
plt.plot(reserves,minrevs)
ax.set_xlabel('reserve')
ax.set_ylabel('minimum revenue')
plt.show()

# Optimum is around 1/8

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

```
Calculating for r=0.0625
Calculating for r=0.06666666666666667
Calculating for r=0.07083333333333333
Calculating for r=0.075
Calculating for r=0.07916666666666666
Calculating for r=0.08333333333333333
Calculating for r=0.0875
Calculating for r=0.09166666666666667
Calculating for r=0.09583333333333333
Calculating for r=0.1
Calculating for r=0.10416666666666666
Calculating for r=0.10833333333333334
Calculating for r=0.1125
Calculating for r=0.11666666666666667
Calculating for r=0.12083333333333333
Calculating for r=0.125
Calculating for r=0.12916666666666665
Calculating for r=0.13333333333333333
Calculating for r=0.1375
Calculating for r=0.14166666666666666
Calculating for r=0.14583333333333331
Calculating for r=0.15
Calculating for r=0.15416666666666667
Calculating for r=0.15833333333333333
Calculating for r=0.1625
Calculating for r=0.16666666666666669
Calculating for r=0.17083333333333334
Calculating for r=0.175
Calculating for r=0.17916666666666667
Calculating for r=0.18333333333333335
Calculating for r=0.1875
```


0.171
0.170
0.169
0.168
0.167
0.166
minimum revenue
0.06
0.08
0.10
0.12
0.14
0.16
0.18
reserve