ProblemSet5

March 25, 2021

1 Math 210

1.1 Aaron Graybill

1.1.1 Problem Set 5

1.1.2 3/25/21

Click on 1.a in the table of contents

1.2 Imports

[2]:
```
import numpy as np
from scipy.optimize import linprog
```

[3]:
```
def print_tableau(a,indep_names,dep_names):
#
# Given matrix "a" and lists of variables names "indep_names" and "dep_names",
# this function prints the matrix and labels in standard tableau format
# (including adding the -1, the minus signs in the last column, and labeling␣
↪the lower-right as obj)
#
# First, check the inputs: indep_names should be one shorter than the number of␣
↪columns of A
#                           dep_names should be one shorter than the number of␣
↪rows of A
#
    nrows = a.shape[0]     # use the shape function to determine number of rows␣
↪and cols in A
    ncols = a.shape[1]
    nindep = len(indep_names)
    ndep = len(dep_names)
    if nindep != ncols-1:
        print("WARNING: # of indep vbles should be one fewer than # columns of␣
↪matrix")
    if ndep != nrows-1:
        print("WARNING: # of dep vbles should be one fewer than # rows of␣
↪matrix")
```

```
# Now do the printing (uses a variety of formatting techniques in Python)
    for j in range(ncols-1):                       # Print the independent variables in the first row
        print(indep_names[j].rjust(10),end="")  # rjust(10) makes fields 10 wide and right-justifies;
                                                #    the end command prevents newline)
    print("        -1")                         # Tack on the -1 at the end of the first row
    for i in range(nrows-1):
        for j in range(ncols):                    # Print all but the last row of the matrix
            print("%10.3f" % a[i][j],end="") # The syntax prints in a field 10 wide, showing 3 decimal points
        lab = "= -" + dep_names[i]
        print(lab.rjust(10))
    for j in range(ncols):
        print("%10.3f" % a[nrows-1][j],end="")  # Print the last row of the matrix, with label "obj" at end
    lab = "= obj"
    print(lab.rjust(10))
    print(" ")    # Put blank line at bottom
```

[4]:

```
def pivot(a,pivrow,pivcol,indep_names,dep_names) :
#
# Given matrix "a", a row number "pivrow" and column number "pivcol",
#  and lists of variable names "indep_names" and "dep_names", this
#  function does three things:
#    (1) outputs the new version of the matrix after a pivot,
#    (2) updates the lists of variable names post-pivot
#    (3) prints the new matrix, including labels showing the variable names
#
# First, check the inputs: indep_names should be one shorter than the number of columns of A
#                          dep_names should be one shorter than the number of rows of A
#                          you should not be pivoting on the last row or last column
#
    a = a.astype(float)   # make sure entries are treated as floating point numbers
    nrows = a.shape[0]    # use the shape function to determine number of rows and cols in A
    ncols = a.shape[1]
    nindep = len(indep_names)
```

```
    ndep = len(dep_names)
    if nindep != ncols-1:
        print("WARNING: # of indep vbles should be one fewer than # columns of␣
↪matrix")
    if ndep != nrows-1:
        print("WARNING: # of dep vbles should be one fewer than # rows of␣
↪matrix")
    if pivrow > nrows-1 or pivcol > ncols-1:
        print("WARNING: should not pivot on last row or column")
    newa = a.copy()       # make a copy of A, to be filled in below with result␣
↪of pivot
    p = a[pivrow-1][pivcol-1]    # identify pivot element
    newa[pivrow-1][pivcol-1] = 1/p    # set new value of pivot element
    # Set entries in p's row
    for j in range(ncols):
        if j != pivcol-1:
            newa[pivrow-1][j]=a[pivrow-1][j]/p;
    # Set entries in p's column
    for i in range(nrows):
        if i != pivrow-1:
            newa[i][pivcol-1]=-a[i][pivcol-1]/p;
    # Set all other entries
    for i in range(nrows):
        for j in range(ncols):
            if i != pivrow-1 and j != pivcol-1:
                r = a[i][pivcol-1]
                q = a[pivrow-1][j]
                s = a[i][j]
                newa[i][j]=(p*s-q*r)/p
    # Now swap the variable names
    temp = indep_names[pivcol-1]
    indep_names[pivcol-1]=dep_names[pivrow-1]
    dep_names[pivrow-1]=temp
    print_tableau(newa,indep_names,dep_names) # Print the matrix with updated␣
↪labels
    return newa;
```

[5]:
```
def target(a) :
    nrows = a.shape[0]     # use the shape function to determine number of rows␣
↪and cols in "a"
    ncols = a.shape[1]
    import numpy as np
    v = np.empty(ncols-1)
    for i in range(ncols-1):
        v[i]=a[nrows-1,i]
    biggest_c = np.max(v)
    where_is_biggest_c = np.argmax(v)+1
```

```
    if biggest_c > 0 :
        return where_is_biggest_c
    else :
        return -1
```

[6]:

```
def select(a,pivcolnum) :
    nrows = a.shape[0]     # use the shape function to determine number of rows␣
 ↪and cols in A
    ncols = a.shape[1]
# First task: work down the column and record the b/a ratios in a vector v
#      except record -1 if a is negative or zero
    import numpy as np
    v = np.zeros(nrows-1)
    for i in range(nrows-1):
        if a[i,pivcolnum-1]>0 :
            v[i] = a[i,ncols-1]/a[i,pivcolnum-1]
        else :
            v[i] = -1
# Second task: if max b/a > -1, find min b/a by hand (ignoring zero entries in␣
 ↪v)
    if np.max(v) > -1 :
        min_so_far = np.max(v)+1  # Initialize variable to be for-sure bigger␣
 ↪than the min
        for i in range(nrows-1):
            if v[i] > -1 and v[i] < min_so_far :
                min_so_far = v[i]
                where_is_min = i+1    # Add 1 to use human numbering
        return where_is_min      # Once we've scanned v for min, we can return␣
 ↪result
    else :
        return -1
```

[7]:

```
# Create Simplex BF
def SimplexBF(a,indep_names,dep_names):
  nrows, ncols = a.shape
  a_new = a
  print_tableau(a_new,indep_names,dep_names)
  while np.max(a_new[nrows-1,:-1])>0:
    pivcol=target(a_new)
    pivrow=select(a_new,pivcol)
    if pivrow == -1:
      return("Unbounded")
    else:
      a_new=pivot(a_new,pivrow,pivcol,indep_names,dep_names)
      print_tableau(a_new,indep_names,dep_names)
```

[8]:
```
def column_delete(a,col_to_remove,indep_names,dep_names) :
    import numpy as np
    anew = np.delete(a,col_to_remove-1,axis=1)
    del indep_names[col_to_remove-1]
    print_tableau(anew,indep_names,dep_names)
    return anew
```

[9]:
```
def row_delete(a,row_to_remove,indep_names,dep_names) :
    import numpy as np
    anew = np.delete(a,row_to_remove-1,axis=0)
    del dep_names[row_to_remove-1]
    print_tableau(anew,indep_names,dep_names)
    return anew
```

[10]:
```
def targetnbf(tab):
  nrows, ncols = tab.shape
  new_i = -1
  for i in range(nrows-1) :# don't check obj fn row
    if tab[i,ncols-1] < 0:
      new_i = i+1
  return(new_i)

def candidateone(tab,targetedrow):
  #don't specify a row that's the obj fn row, I have no protocol against it
  nrows, ncols = tab.shape
  for i in range(ncols-1): #don't check last col bc it's the b's
    if tab[targetedrow-1,i]<0 :
      return(i+1)#bailout if found one
  return(-1) #none found in loop, return -1

def  selectnbf(tab,targetedrow,pivcolumn) :
  nrows, ncols = tab.shape

  #subset of two columns in question exclude obj fn row
  candidate_column =tab[targetedrow-1:nrows-1,pivcolumn-1]
  b_column=tab[targetedrow-1:nrows-1,ncols-1]

  # compute ratios
  ratios=b_column/candidate_column

  # Find row in subset and convert back to index in full table:
  #don't need to start at first row, let those be the starting values
  cur_ratio= ratios[0]
  best_row=0
  # the seemingly misplaced -1s and +1s are because we start at second row
  for i in range(len(candidate_column)-1):
    if candidate_column[i+1]>0 and ratios[i+1]<cur_ratio :
```

```
      cur_ratio=ratios[i+1]
      best_row=i+1

  #the targetedrow already has the +1, so this converts to start at 1
    #and converts table subset index to full tableau index
  best_row=best_row+targetedrow

  return([best_row,pivcolumn]) #output

def simplexnbf(tab,indep_names,dep_names):
  nrows, ncols = tab.shape #get dims
  current_target_row=np.Inf #set current target row to nonsense
  tab_new=tab # get primer value for tableau iterator

  NotReadyForBF = True #changed when problem is BF
  SolutionPossible = True #false when no solution is discovered

  print_tableau(tab,indep_names,dep_names) #print initial tableau

  # while NBF and solution still possible apply NBF rules
  while NotReadyForBF and SolutionPossible:
    current_target_row=targetnbf(tab_new) # find target
    if current_target_row ==-1 : #if target is -1, Ready for BF
      NotReadyForBF = False
    else : #find candidate from targeted row
      current_candidate_column=candidateone(tab_new,current_target_row)
      if current_candidate_column ==-1 : #if no candidates, no solution
        SolutionPossible = False
      else : #pivot based off of the computed selection
        ␣
↪pivot_row,pivot_col=selectnbf(tab_new,current_target_row,current_candidate_column)
        tab_new=pivot(tab_new,pivot_row,pivot_col,indep_names,dep_names)
  if not NotReadyForBF : #if we bailed because ready for BF apply SimplexBF
    SimplexBF(tab_new,indep_names,dep_names)
  if not SolutionPossible: # if we bailed bc constraint set emtpy, say so
    return("-1, Constraint set empty, sorry  ")

  #we can't exit the while loop for anything but the reasons above,
  # so don't need any else statements
```

[11]:
```
def dual_print_tableau(a,indep_names,dep_names,indep_names_dual,dep_names_dual):
#
# Given matrix "a" and lists of variables names "indep_names" and "dep_names",
#   and (for the dual) "indep_names_dual" and "dep_names_dual",
# this function prints the matrix and labels in standard tableau format
# (including adding the -1, the minus signs in the last column, and labeling␣
↪the lower-right as obj)
```

```
#
# First, check the inputs: indep_names and dep_names_dual should be one shorter than the number of columns of A
#                            dep_names and indep_names_dual should be one shorter than the number of rows of A
#
    nrows = a.shape[0]     # use the shape function to determine number of rows and cols in A
    ncols = a.shape[1]
    nindep = len(indep_names)
    nindep_dual = len(indep_names_dual)
    ndep = len(dep_names)
    ndep_dual = len(dep_names_dual)
    if nindep != ncols-1:
        print("WARNING: # of indep vbles should be one fewer than # columns of matrix")
    if ndep != nrows-1:
        print("WARNING: # of dep vbles should be one fewer than # rows of matrix")
    if nindep_dual != nrows-1:
        print("WARNING: # of indep dual vbles should be one fewer than # rows of matrix")
    if ndep_dual != ncols-1:
        print("WARNING: # of dep dual vbles should be one fewer than # columns of matrix")
# Now do the printing (uses a variety of formatting techniques in Python)
    print("          ",end="")       # On first line, leave blank space so we can fit in dual labels lower down
    for j in range(ncols-1):                       # Print the independent variables in the first row
        print(indep_names[j].rjust(10),end="")  # rjust(10) makes fields 10 wide and right-justifies;
                                                   #    the end command prevents newline)
    print("        -1")                            # Tack on the -1 at the end of the first row
    for i in range(nrows-1):
        print(indep_names_dual[i].rjust(10),end="")
        for j in range(ncols):                     # Print all but the last row of the matrix
            print("%10.3f" % a[i][j],end="") # The syntax prints in a field 10 wide, showing 3 decimal points
        lab = "= -" + dep_names[i]
        print(lab.rjust(10))
    print("        -1",end="")
    for j in range(ncols):
```

```
        print("%10.3f" % a[nrows-1][j],end="")  # Print the last row of the matrix, with label "obj" at end
    lab = "= obj"
    print(lab.rjust(10))
    print("          ",end="")
    for j in range(ncols-1):
        lab = "=" + dep_names_dual[j]
        print(lab.rjust(10),end="")
    print("  =dualobj")
    print(" ")    # Put blank line at bottom
```

[12]:
```
def dual_pivot(a,pivrow,pivcol,indep_names,dep_names,indep_names_dual,dep_names_dual):
#
# Given matrix "a", a row number "pivrow" and column number "pivcol",
#  and lists of variable names "indep_names" and "dep_names", this
#  function does three things:
#    (1) outputs the new version of the matrix after a pivot,
#    (2) updates the lists of variable names post-pivot
#    (3) prints the new matrix, including labels showing the variable names
#
# First, check the inputs: indep_names should be one shorter than the number of columns of A
#                          dep_names should be one shorter than the number of rows of A
#                          you should not be pivoting on the last row or last column
#
    a = a.astype(float)   # make sure entries are treated as floating point numbers
    nrows = a.shape[0]    # use the shape function to determine number of rows and cols in A
    ncols = a.shape[1]
    nindep = len(indep_names)
    nindep_dual = len(indep_names_dual)
    ndep = len(dep_names)
    ndep_dual = len(dep_names_dual)
    if nindep != ncols-1:
        print("WARNING: # of indep vbles should be one fewer than # columns of matrix")
    if ndep != nrows-1:
        print("WARNING: # of dep vbles should be one fewer than # rows of matrix")
    if nindep_dual != nrows-1:
```

```
        print("WARNING: # of indep dual vbles should be one fewer than # rows␣
↪of matrix")
    if ndep_dual != ncols-1:
        print("WARNING: # of dep dual vbles should be one fewer than # columns␣
↪of matrix")
    if pivrow > nrows-1 or pivcol > ncols-1:
        print("WARNING: should not pivot on last row or column")
    newa = a.copy()       # make a copy of A, to be filled in below with result␣
↪of pivot
    p = a[pivrow-1][pivcol-1]    # identify pivot element
    newa[pivrow-1][pivcol-1] = 1/p    # set new value of pivot element
    # Set entries in p's row
    for j in range(ncols):
        if j != pivcol-1:
            newa[pivrow-1][j]=a[pivrow-1][j]/p;
    # Set entries in p's column
    for i in range(nrows):
        if i != pivrow-1:
            newa[i][pivcol-1]=-a[i][pivcol-1]/p;
    # Set all other entries
    for i in range(nrows):
        for j in range(ncols):
            if i != pivrow-1 and j != pivcol-1:
                r = a[i][pivcol-1]
                q = a[pivrow-1][j]
                s = a[i][j]
                newa[i][j]=(p*s-q*r)/p
    # Now swap the variable names
    temp = indep_names[pivcol-1]
    indep_names[pivcol-1]=dep_names[pivrow-1]
    dep_names[pivrow-1]=temp
    temp = indep_names_dual[pivrow-1]
    indep_names_dual[pivrow-1]=dep_names_dual[pivcol-1]
    dep_names_dual[pivcol-1]=temp
    ␣
↪dual_print_tableau(newa,indep_names,dep_names,indep_names_dual,dep_names_dual)␣
↪# Print the matrix with updated labels
    return newa;
```

1.3 Problem 1.

1.3.1 a.

As a tableau we have the following (ebing sure to convert the geq equation to leq and flipping the sign on the obj to minimize):

```
[13]: a=np.array([[-1,-2,-3,-24],[-2,-4,-3,-36],[-3,-1,-2,0]])
id=["x","y","z"]
dp=["t1","0"]
print_tableau(a,id,dp)
```

```
        x         y         z        -1
   -1.000    -2.000    -3.000   -24.000     = -t1
   -2.000    -4.000    -3.000   -36.000      = -0
   -3.000    -1.000    -2.000     0.000     = obj
```

Following the rules for when we have an equality constraint, I will pivot on x and then delete the corresponding column.

```
[14]: a2=pivot(a,2,1,id,dp)
a3=column_delete(a2,1,id,dp)
simplexnbf(a3,id,dp)
```

```
        0         y         z        -1
   -0.500    -0.000    -1.500    -6.000     = -t1
   -0.500     2.000     1.500    18.000      = -x
   -1.500     5.000     2.500    54.000     = obj

        y         z        -1
   -0.000    -1.500    -6.000     = -t1
    2.000     1.500    18.000      = -x
    5.000     2.500    54.000     = obj

        y         z        -1
   -0.000    -1.500    -6.000     = -t1
    2.000     1.500    18.000      = -x
    5.000     2.500    54.000     = obj

        y        t1        -1
    0.000    -0.667     4.000      = -z
    2.000     1.000    12.000      = -x
    5.000     1.667    44.000     = obj

        y        t1        -1
    0.000    -0.667     4.000      = -z
    2.000     1.000    12.000      = -x
    5.000     1.667    44.000     = obj

        x        t1        -1
   -0.000    -0.667     4.000      = -z
    0.500     0.500     6.000      = -y
   -2.500    -0.833    14.000     = obj
```

```
        x         t1         -1
   -0.000     -0.667      4.000        = -z
    0.500      0.500      6.000        = -y
   -2.500     -0.833     14.000      = obj
```

We now have a solution which states that $x = 0, y = 6, z = 4$ and the function attains a minimum of 14

1.4 b.

`linprog` takes leq contraints by default, so I convert to leq constraints where necessary. however, it minimizes, so no need to transform obj.

[15]:
```
A1=np.array([[-1,-2,-3]])
A2=np.array([[2,4,3]])
b1=[-24]
b2=[36]

c=[3,1,2]

linprog(c,A_ub=A1,b_ub=b1,A_eq=A2,b_eq=b2,method='simplex')
```

[15]:
```
     con: array([0.])
     fun: 14.0
 message: 'Optimization terminated successfully.'
     nit: 2
   slack: array([0.])
  status: 0
 success: True
       x: array([0., 6., 4.])
```

Note that this output is identical to doing things manually.

1.5 Problem 2.

1.5.1 a.

I set up the tableau as follows:

[16]:
```
a=np.array([[1,-1,2,6],[1,0,2,8],[0,-1,-2,-2],[3,-2,3,0]])
id=["xcirc","y","z"]
dp=["0","0","t3"]
print_tableau(a,id,dp)
```

```
    xcirc          y          z         -1
    1.000     -1.000      2.000      6.000        = -0
    1.000      0.000      2.000      8.000        = -0
    0.000     -1.000     -2.000     -2.000       = -t3
```

```
   3.000    -2.000     3.000     0.000     = obj
```

Killing two birds with one stone, I pivot on one one and delete the fiirst column and row because our unconstrained indep var and equality constraint rules tell us to do those things:

[17]:
```
a2=pivot(a,1,1,id,dp)
a3=column_delete(a2,1,id,dp)
a4=row_delete(a3,1,id,dp)
```

```
       0         y         z        -1
   1.000    -1.000     2.000     6.000  = -xcirc
  -1.000     1.000     0.000     2.000      = -0
  -0.000    -1.000    -2.000    -2.000     = -t3
  -3.000     1.000    -3.000   -18.000     = obj

       y         z        -1
  -1.000     2.000     6.000  = -xcirc
   1.000     0.000     2.000      = -0
  -1.000    -2.000    -2.000     = -t3
   1.000    -3.000   -18.000     = obj

       y         z        -1
   1.000     0.000     2.000      = -0
  -1.000    -2.000    -2.000     = -t3
   1.000    -3.000   -18.000     = obj
```

Pivoting again on one-one allows us to delete the first column.

[18]:
```
a5=pivot(a4,1,1,id,dp)
a6=column_delete(a5,1,id,dp)
```

```
       0         z        -1
   1.000     0.000     2.000      = -y
   1.000    -2.000     0.000     = -t3
  -1.000    -3.000   -20.000     = obj

       z        -1
   0.000     2.000      = -y
  -2.000     0.000     = -t3
  -3.000   -20.000     = obj
```

This is now conveniently solved. We have that the maximum is 20 which is attained at $y = 2, z = 0$ and then plugging back into the first equality constraint implies that $x^* = 6-2z^*+y^* = 6-0+2 = 8$.

1.5.2 b.

Implementing this with `linprog` is noting that we have to flip one constranit and the obj function because we're maximinzing:

[19]:
```
a=np.array([[1,-1,2,6],[1,0,2,8],[0,-1,-2,-2],[3,-2,3,0]])

A1=np.array([[0,-1,-2]])
A2=np.array([[1,-1,2],[1,0,2,]])
b1=[-2]
b2=[6,8]

v = [(None,None) , (0,None) , (0,None) ]

c=[-3,2,-3]

linprog(c,A_ub=A1,b_ub=b1,A_eq=A2,b_eq=b2,bounds=v,method='simplex')
```

[19]:
```
     con: array([0., 0.])
     fun: -20.0
 message: 'Optimization terminated successfully.'
     nit: 4
   slack: array([0.])
  status: 0
 success: True
       x: array([8., 2., 0.])
```

Woohoo! that's the same result.

1.6 Problem 3.

###a. Using the $c_N^{T-c}_B^T

[31]:
```
B=np.array([[5,0],[8,1]])
N=np.array([[1,3,1],[4,2,0]])
#print(B)
binv=np.linalg.inv(B)
cnt=np.array([[4,2,0]])
cbt=np.array([[5,0]])
temp=cbt.dot(binv)

cnt-temp.dot(N)
```

[31]:
```
array([[ 3., -1., -1.]])
```

1.6.1 b.

We are required to pivot on column 1 as is the only positive value.

1.6.2 c.

We only need compute column 1 and the last column. We can do that in the following way:

[45]:
```
b=np.array([[4],[10]])

print("new b is" + str(binv@b))
print("new A is" +str(binv@N[:,0]))
```

```
new b is[[0.8]
 [3.6]]
new A is[0.2 2.4]
```

The ratios are $\frac{.8}{.2} = 4$ and $\frac{2.4}{3.6} = \frac{2}{3}$, so we pivot on 2,1.

1.7 Problem 4.

1.7.1 a.

The fast way to do this is to treat the given problem as the dual problem and reverse engineer the original problem (because we're given a minimize obj style problem. I will read in the rows of the constraints as columns in the following way (i'm just going to transpose the given data).

[20]:
```
a=np.transpose(np.array([[1,2,2,200],[2,2,1,150],[20,30,25,0]]))
id=["x1","x2"]
dp=["t1","t2","t3"]
id2=["y1","y2","y3"]
dp2=["s1","s2"]
dual_print_tableau(a,id,dp,id2,dp2)
```

```
              x1        x2         -1
    y1     1.000     2.000     20.000      = -t1
    y2     2.000     2.000     30.000      = -t2
    y3     2.000     1.000     25.000      = -t3
    -1   200.000   150.000      0.000      = obj
             =s1       =s2   =dualobj
```

So reverse engineering the dual problem we have, maximize $200x_1 + 150x_2$ such that:

$$x_1 + 2x_2 \leq 20$$
$$2x_1 + 2x_2 \leq 30$$
$$2x_1 + 1x_2 \leq 20$$

1.7.2 b.

I will apply simplex BF manually to track the variables effectively:

[21]:
```
a2=dual_pivot(a,1,2,id,dp,id2,dp2)
a3=dual_pivot(a2,2,1,id,dp,id2,dp2)
```

```
a4=dual_pivot(a3,3,2,id,dp,id2,dp2)
```

```
              x1        t1         -1
s2         0.500     0.500     10.000      = -x2
y2         1.000    -1.000     10.000      = -t2
y3         1.500    -0.500     15.000      = -t3
-1       125.000   -75.000  -1500.000      = obj
             =s1       =y1   =dualobj

              t2        t1         -1
s2        -0.500     1.000      5.000      = -x2
s1         1.000    -1.000     10.000      = -x1
y3        -1.500     1.000      0.000      = -t3
-1      -125.000    50.000  -2750.000      = obj
             =y2       =y1   =dualobj

              t2        t3         -1
s2         1.000    -1.000      5.000      = -x2
s1        -0.500     1.000     10.000      = -x1
y1        -1.500     1.000      0.000      = -t1
-1       -50.000   -50.000  -2750.000      = obj
             =y2       =y3   =dualobj
```

1.7.3 c.

Thankfully the problem only took one pivot to complete. The solution is that $x_1 = 10$,$x_2 = 5, y_1 = 0, y_2 = 50, y_3 = 50$ and the maximum/minimum is 2750

1.8 Problem 5.

1.8.1 a.

We are tasked with minimizing:

$$.8Milk + .6Banana + .4BrusselSprouts + 2Salmon + Liverwurst$$

While satisfying:

$$3.1Milk + 1.1Banana + 2.6BrusselSprouts + 27Salmon + 12Liverwurst \geq 80 \tag{1}$$
$$64Milk + 89Banana + 36BrusselSprouts + 184Salmon + 305Liverwurst \geq 1800 \tag{2}$$

Thankfully no manipulations are necessary to convert this to canonical min.

1.8.2 b.

I will use `linprog` to solve the problem:

```
[22]: A1=-1*np.array([[3.1,1.1,2.6,27,12],[64,89,36,184,305]])
b1=[-80,-1800]



v = [(0,None), (0,None),(0,None), (0,None),(0,None)]

c=[.8,.6,.4,2,1]

linprog(c,A_ub=A1,b_ub=b1,bounds=v,method='simplex')
```

```
[22]:      con: array([], dtype=float64)
     fun: 6.550522648083624
 message: 'Optimization terminated successfully.'
     nit: 2
   slack: array([0., 0.])
  status: 0
 success: True
       x: array([0.        , 0.        , 0.        , 0.46457607, 5.6213705 ])
```

The cost minimizing bundle is .46 units of salmon and 5.62 units of liverwurst costing a total of \$6.55

1.8.3 c.

Now let's solve the problem using the dual maximization problem. We would have the following tableau. Note that we no longer need to multiply by negative ones, because we are treating the problems as the columns which have the desired signs by default.

```
[23]: a=np.transpose(np.array([[3.1,1.1,2.6,27,12,80],[64,89,36,184,305,1800],[.8,.6,.
 ↪4,2,1,0]]))
id=["x1","x2"]
dp=["t1","t2","t3","t4","t5"]
id2=["Milk","Bananas","Brussel","Salmon","Liver"]
dp2=["s1","s2"]
dual_print_tableau(a,id,dp,id2,dp2)
```

```
                 x1        x2        -1
      Milk     3.100    64.000     0.800     = -t1
   Bananas     1.100    89.000     0.600     = -t2
   Brussel     2.600    36.000     0.400     = -t3
    Salmon    27.000   184.000     2.000     = -t4
     Liver    12.000   305.000     1.000     = -t5
        -1    80.000  1800.000     0.000     = obj
                 =s1       =s2  =dualobj
```

You can read off the canonical max problem from the chart above. Running simplex on this code gives:

[24]:
```
simplexnbf(a,id,id2)
```

```
     x1         x2         -1
  3.100     64.000      0.800    = -Milk
  1.100     89.000      0.600= -Bananas
  2.600     36.000      0.400= -Brussel
 27.000    184.000      2.000 = -Salmon
 12.000    305.000      1.000  = -Liver
 80.000   1800.000      0.000      = obj

     x1         x2         -1
  3.100     64.000      0.800    = -Milk
  1.100     89.000      0.600= -Bananas
  2.600     36.000      0.400= -Brussel
 27.000    184.000      2.000 = -Salmon
 12.000    305.000      1.000  = -Liver
 80.000   1800.000      0.000      = obj

     x1      Liver         -1
  0.582     -0.210      0.590    = -Milk
 -2.402     -0.292      0.308= -Bananas
  1.184     -0.118      0.282= -Brussel
 19.761     -0.603      1.397 = -Salmon
  0.039      0.003      0.003      = -x2
  9.180     -5.902     -5.902      = obj

     x1      Liver         -1
  0.582     -0.210      0.590    = -Milk
 -2.402     -0.292      0.308= -Bananas
  1.184     -0.118      0.282= -Brussel
 19.761     -0.603      1.397 = -Salmon
  0.039      0.003      0.003      = -x2
  9.180     -5.902     -5.902      = obj

 Salmon      Liver         -1
 -0.029     -0.192      0.549    = -Milk
  0.122     -0.365      0.478= -Bananas
 -0.060     -0.082      0.198= -Brussel
  0.051     -0.031      0.071      = -x1
 -0.002      0.004      0.000      = -x2
 -0.465     -5.621     -6.551      = obj

 Salmon      Liver         -1
 -0.029     -0.192      0.549    = -Milk
  0.122     -0.365      0.478= -Bananas
 -0.060     -0.082      0.198= -Brussel
  0.051     -0.031      0.071      = -x1
```

```
-0.002      0.004      0.000      = -x2
-0.465     -5.621     -6.551      = obj
```

This notation is more than a little naughty, but it does allow us to read off the desired answer in real world terms. The $x_1 = .071$ an $x_2 = 0$. But reading off of the real world values we have that Salmon is .465 and Liver is 5.621 being sure to account for the signs while the others are zero and the total cost is \$6.55

1.9 Problem 6.

1.9.1 a.

[25]:
```
import pandas as pd
url = 'https://raw.githubusercontent.com/aarongraybill/Math210/main/ProblemSets/
↪ProblemSet5/nutritional_data_by_food_group.csv'
df = pd.read_csv(url, error_bad_lines=False)
food_groups=df.to_numpy()[:,1:]

dep_food=list(df.columns)[1:]
indep_food=df.to_numpy()[:,0]

#print the dimensions of the matrix
food_groups.shape
```

[25]:
```
(9, 7)
```

1.9.2 b.

Let's define a per capita-kilogram cost vector:

[26]:
```
print(indep_food)

cost_percap=[2.7,1.2,1.0,3.2,2.4,3.2,3.2,2.0,2.9]
#cost_percap = np.array([[x / 83000 for x in cost_percap]])

print(cost_percap)
```

```
['Cereal' 'Vegetables' 'Fruit' 'Fats/oils' 'Dairy' 'Fish' 'Meat' 'Eggs'
 'Pulses']
[2.7, 1.2, 1.0, 3.2, 2.4, 3.2, 3.2, 2.0, 2.9]
```

1.9.3 c.

First lets consider the cofficients in the LHS of the constraints. We have all of the coefficients in the table we read in. Further more, in the last constraint, we have a the raw sum of vegetables and fruits meaning we have 1 in those columns and 0 otherwise. The b values are a little more tricky. If we had no donations, we would just need the requirements as given plus one additional row = 2 for the micronutrient constraint. But we do have donations so that makes this slightly more complicated.

Let $\preceq$ be the element-wise $\leq$. Normally, we can write the constraints as $Ax \preceq b$ (like if we had no donations. But in the problem at hand each food (x) is given a percapita donation d_i. Putting those percapita donations gives vector d. We can then rewrite the constraints as: $A(x + d) \preceq b$. Basic matrix algebra gives that $Ax \preceq b - Ad$. Treating $b - Ad$ as a set of new b's means that we now have a full set of complete constraints. The objective function is simply the price of each good per kilogram summed together. No adjustment is necessary for the donations because the x vector is the purchases above donation per capita. I code that as follows:

[27]:
```
#compute donation per capita
donation_percap=[52000,69000,94000,13000,73000,5800,3700,0,13000]
donation_percap = np.array([[x / 83000 for x in donation_percap]])

#the required macronutrient vector with micronutrient
b=np.array([[910,210,340,100,9.5,360,15500,2]])

#add the fruit constraint
indep_final=indep_food
dep_final=np.append(dep_food,"micronut")
food_groups_w_mirco=np.append(food_groups,np.
 ↪transpose([[0,1,1,0,0,0,0,0,0]]),axis=1)

#matrix mult to compute the remaining nutrients after donation
new_b_w_donations=b-donation_percap@food_groups_w_mirco



#add nutrient vector to new constraints
tableau1=np.append(food_groups_w_mirco,new_b_w_donations,axis=0)
tableau1=np.transpose(tableau1)
#print(tableau1)


#add obj fn

obj_row=np.array([np.append(cost_percap,0)])
#print(obj_row)
tableau_final=-1*np.append(tableau1,obj_row,axis=0)
#print(tableau3)
#

print_tableau(tableau_final,indep_final,dep_final)
```

```
    CerealVegetables      Fruit Fats/oils      Dairy       Fish       Meat       Eggs
Pulses          -1
  -590.000    -86.000  -130.000     -0.710   -64.000     -2.700     -3.800     -0.000
```

```
-440.000  -195.964= -Carbs (g)
   -33.000    -18.000    -19.000     -0.000     -0.180     -0.000     -0.000     -0.000
-86.000  -139.215= -Fiber (g)
   -53.000     -2.700    -12.000   -970.000    -37.000    -70.000   -120.000   -100.000
-28.000   -91.864= -Lipids (g)
    -7.100     -0.230     -1.400    -83.000     -3.900    -23.000    -30.000     -8.200
-4.400   -73.711= -Omega-6 Acid (g)
    -0.440     -0.099     -0.130     -3.800     -0.600     -7.100     -3.400     -0.770
-0.038    -7.218= -Omega-3 Acid (g)
   -84.000    -17.000    -12.000     -0.860    -45.000   -170.000   -170.000   -120.000
-180.000  -192.287= -Protein (g)
 -3200.000   -440.000   -730.000 -8600.000   -750.000 -1400.000 -1900.000 -1400.000
-3000.000 -9643.614= -Energy (kcal)
     0.000     -1.000     -1.000      0.000      0.000      0.000      0.000      0.000
0.000    -0.036= -micronut
    -2.700     -1.200     -1.000     -3.200     -2.400     -3.200     -3.200     -2.000
-2.900    -0.000      = obj
```

[28]:
```python
A=tableau_final[:-1,:-1]
b=-new_b_w_donations
c=cost_percap

result=linprog(c,A_ub=A,b_ub=b,method='simplex')

#print(result)
print("Total cost is "+str(result.fun*83000))
print("Percapita cost is "+str(result.fun))
print("\n\n")
print(np.transpose(np.array([indep_final,np.round(result.x,3)])))
print("\n\n\n")
print(np.transpose(np.array([dep_final,np.round(result.slack,3)])))
```

```
Total cost is 734073.2775406948
Percapita cost is 8.844256355911986



[['Cereal' 0.0]
 ['Vegetables' 0.0]
 ['Fruit' 0.036]
 ['Fats/oils' 0.614]
 ['Dairy' 0.0]
 ['Fish' 0.679]
 ['Meat' 0.0]
 ['Eggs' 0.0]
 ['Pulses' 1.611]]
```

```
[['Carbs (g)' '519.753']
 ['Fiber (g)' '0.0']
 ['Lipids (g)' '596.762']
 ['Omega-6 Acid (g)' '0.0']
 ['Omega-3 Acid (g)' '0.0']
 ['Protein (g)' '214.006']
 ['Energy (kcal)' '1445.789']
 ['micronut' '-0.0']]
```

The results above show that at the optimum costs $8.84 dollars per person and costs $734,073 to feed all 83 thousand. 36g of fruits are bought per person, .614 g of fats and oils are bought per person, 679 g of fish per person, and 1.611 kg of pulses per person. No other food is bought.