Assignment 3: Optimization II (Solution)

ME C231A, EECS C220B, UC Berkeley

Note: You should now be familiar with `cvxopt` to solve linear and quadratic programs. For certain problems in this assignment you will need to use a nonlinear solver. You may specify `IPOPT` as a solver in `pyomo` which solves constrained, nonlinear programs.

1. Linear and Quadratic Programming:

The following problems include some of the examples from homework 2. This time, use `Pyomo` to solve them. Please submit your solutions as individual functions for each of the 4 parts.

These functions should have no inputs and 4 outputs. The first two outputs are logical values indicating the feasibility and boundedness, where a [1,1] stands for a feasible problem and bounded solution. The third output is the value of the optimizer, which should be an $N \times 1$ vector, where N is the dimension of decision variable. If the problem is infeasible or unbounded, the function should return an empty array here (i.e. `zOpt = []`). The fourth output is the optimal value of the cost function. If the problem is infeasible, the function should return an empty array here. If the problem is unbounded return `+inf` or `-inf` here. For linear programs, use `cbc` as the solver and for quadratic program, use `IPOPT` solver.

Part (a)

$$
\begin{aligned}
\min_{z_1,z_2} \quad & -5z_1 - 7z_2 \\
\text{s.t.} \quad & -3z_1 + 2z_2 \leq 30 \\
& -2z_1 + z_2 \leq 12 \\
& z_1 \geq 0 \\
& z_2 \geq 0
\end{aligned}
$$

Write a function `LPQPa`, with function declaration line

```
def LPQPa():
    return feas, bound, zOpt, JOpt
```

In [1]:

```python
import pyomo.environ as pyo
def check_solver_status(model, results):
    from pyomo.opt import SolverStatus, TerminationCondition
    if (results.solver.status == SolverStatus.ok) and (results.solver.terminat
ion_condition == TerminationCondition.optimal):
        print('===============================================================
=========================')
        print('================ Problem is feasible and the optimal solution i
s found ==================')
        print('z1 optimal=', pyo.value(model.z[1]))
        print('z2 optimal=', pyo.value(model.z[2]))
        print('optimal value=', pyo.value(model.obj))
        print('===============================================================
=========================')
        bound = True
        feas = True
        zOpt = np.array([pyo.value(model.z[1]), pyo.value(model.z[2])])
        JOpt = pyo.value(model.obj)
    elif (results.solver.termination_condition == TerminationCondition.infeasi
ble):
        print('========================================================')
        print('================ Problem is infeasible ==================')
        print('========================================================')
        feas = False
        zOpt = []
        JOpt = []
        if (results.solver.termination_condition == TerminationCondition.unbou
nded):
            print('================ Problem is unbounded ==================')
            bound = False
        else:
            bound = True

    else:
        if (results.solver.termination_condition == TerminationCondition.unbou
nded):
            print('================ Problem is unbounded ==================')
            bound = False
            feas = True
            zOpt = []
            JOpt = np.inf
        else:
            bound = True
            feas = True
            zOpt = []
            JOpt = np.inf

    return feas, bound, zOpt, JOpt
```

In [2]:
```
from __future__ import division
import pyomo.environ as pyo
import numpy as np

def LPQPa():

    model = pyo.ConcreteModel()
    model.z = pyo.Var([1,2], domain=pyo.NonNegativeReals)
    model.obj = pyo.Objective(expr = -5*model.z[1] - 7*model.z[2])
    model.Constraint1 = pyo.Constraint(expr = -3*model.z[1] + 2*model.z[2] <= 
30.0)
    model.Constraint2 = pyo.Constraint(expr = -2*model.z[1] + model.z[2] <= 1
2.0)

    solver = pyo.SolverFactory('cbc')
    results = solver.solve(model)

    return check_solver_status(model, results)

# call the function:
feas, bound, zOpt, JOpt = LPQPa()
```

```
WARNING: Loading a SolverResults object with a warning status into
    model=unknown;
        message from solver=<undefined>
================ Problem is unbounded ==================
```

Part (b)

$$
\begin{aligned}
\min_{z_1,z_2}\ & 3z_1 + z_2 \\
\text{s.t. } & z_1 - z_2 \leq 1 \\
& 3z_1 + 2z_2 \leq 12 \\
& 2z_1 + 3z_2 \leq 3 \\
& -2z_1 + 3z_2 \geq 9 \\
& z_1 \geq 0 \\
& z_2 \geq 0
\end{aligned}
$$

Write a function **LPQPb**, with function declaration line

```
def LPQPb():
    return feas, bound, zOpt, JOpt
```

In [3]:
```
from __future__ import division
import pyomo.environ as pyo

def LPQPb():

    model = pyo.ConcreteModel()

    model.z = pyo.Var([1,2], domain=pyo.NonNegativeReals)

    model.obj = pyo.Objective(expr = 3*model.z[1] + model.z[2])

    model.Constraint1 = pyo.Constraint(expr = model.z[1] - model.z[2] <= 1)
    model.Constraint2 = pyo.Constraint(expr = 3*model.z[1] + 2*model.z[2] <= 1
2)
    model.Constraint3 = pyo.Constraint(expr = 2*model.z[1] + 3*model.z[2] <= 3
)
    model.Constraint4 = pyo.Constraint(expr = -2*model.z[1] + 3*model.z[2] >=
9)

    solver = pyo.SolverFactory('cbc')

    results = solver.solve(model)

    return check_solver_status(model, results)

# call the function:
feas, bound, zOpt, JOpt = LPQPb()
```

```
WARNING: Loading a SolverResults object with a warning status into
    model=unknown;
        message from solver=<undefined>
========================================================
================ Problem is infeasible ==================
========================================================
```

Part (c)

$$
\begin{aligned}
\min \quad & \left\| \begin{bmatrix} z_1 \\ z_2 + 5 \end{bmatrix} \right\|_1 + \left\| \begin{bmatrix} z_1 - 2 \\ z_2 \end{bmatrix} \right\|_\infty \\
\text{subject to} \quad & 3z_1 + 2z_2 \leq -3 \\
& 0 \leq z_1 \leq 2 \\
& -2 \leq z_2 \leq 3
\end{aligned}
$$

Note: Use the LP formulation of this problem. Hint: You have already done this in HW2!

Write a function **LPQPc**, with function declaration line

```
def LPQPc():
    return feas, bound, zOpt, JOpt
```

In [4]:
```
from __future__ import division
import pyomo.environ as pyo
import numpy as np
from pyomo.opt import SolverStatus, TerminationCondition

def LPQPc():

    model = pyo.ConcreteModel()

    model.z = pyo.Var([1,2])
    model.t_one = pyo.Var([1,2])
    model.t_inf = pyo.Var()

    model.obj = pyo.Objective(expr = model.t_one[1] + model.t_one[2] + model.t
_inf)

    model.constraint = pyo.ConstraintList()
    model.constraint.add(expr = 3*model.z[1] + 2*model.z[2] <= -3)
    model.constraint.add(expr = (0, model.z[1], 2))
    model.constraint.add(expr = (-2, model.z[2], 3))
    model.constraint.add(expr = -model.t_one[1] <= model.z[1])
    model.constraint.add(expr = model.z[1] <= model.t_one[1])
    model.constraint.add(expr = -model.t_one[2]-5 <= model.z[2])
    model.constraint.add(expr = model.z[2] <= model.t_one[2]-5)

    model.constraint.add(expr = -model.t_inf+2 <= model.z[1])
    model.constraint.add(expr = model.z[1] <= model.t_inf+2)
    model.constraint.add(expr = -model.t_inf <= model.z[2])
    model.constraint.add(expr = model.z[2] <= model.t_inf)

    solver = pyo.SolverFactory('cbc')

    results = solver.solve(model)

    return check_solver_status(model, results)

# call the function:
feas, bound, zOpt, JOpt = LPQPc()
```

```
=============================================================================
===========
================ Problem is feasible and the optimal solution is found ======
============
z1 optimal= 0.0
z2 optimal= -2.0
optimal value= 5.0
=============================================================================
===========
```

Part (d)

$$\begin{aligned}\min_{z_1,z_2}\ & z_1^2+z_2^2\\ \text{s.t. } & z_1 \le -3\\ & z_2 \le 4\\ & 0 \ge 4z_1+3z_2\end{aligned}$$

Write a function `LPQPd`, with function declaration line

```
def LPQPd():
    return feas, bound, zOpt, JOpt
```

In [5]:
```python
from __future__ import division
import pyomo.environ as pyo

def LPQPd():

    model = pyo.ConcreteModel()
    model.z = pyo.Var([1,2])
    model.obj = pyo.Objective(expr = model.z[1]**2 + model.z[2]**2)


    model.constraint1 = pyo.Constraint(expr = model.z[1] <= -3)
    model.constraint2 = pyo.Constraint(expr = model.z[2] <= 4)
    model.constraint3 = pyo.Constraint(expr = 4*model.z[1] + 3*model.z[2] <= 0
)

    solver = pyo.SolverFactory('ipopt')
    results = solver.solve(model)

    return check_solver_status(model, results)

# call the function:
feas, bound, zOpt, JOpt = LPQPd()
```

```
============================================================================
===========
================ Problem is feasible and the optimal solution is found ======
============
z1 optimal= -2.9999999704176505
z2 optimal= -6.265025272784229e-10
optimal value= 8.999999822505904
============================================================================
===========
```

2. Nonlinear Programming I:

Part (a)

Write a function **NLP1**, which solves the optimization problem defined below using **pyomo**, with function declaration line

```
def NLP1(z0):
    return zOpt, JOpt
```

where the input **z0** is the initial guess for your optimizer. The function should have 2 outputs. **zOpt** is the optimizer and **JOpt** is the optimal cost. Also, call the function and print the outputs.

$$\begin{aligned}\min_{z_1,z_2}\ & 3\sin(-2\pi z_1) + 2z_1 + 4 + \cos(2\pi z_2) + z_2 \\ \text{s.t.}\ & -1 \leq z_1 \leq 1 \\ & -1 \leq z_2 \leq 1\end{aligned}$$

In [6]:

```
## without initialization:

def NLP1(z0=[]):

    model = pyo.ConcreteModel()
    model.z1 = pyo.Var()
    model.z2 = pyo.Var()
    model.obj = pyo.Objective(expr = 3*pyo.sin(-2*np.pi*model.z1)+ 2*model.z1
+ 4 + pyo.cos(2*np.pi*model.z2) + model.z2)
    model.constraint1 = pyo.Constraint(expr = (-1, model.z1, 1))
    model.constraint2 = pyo.Constraint(expr = (-1, model.z2, 1))
    if z0:
        model.z1 = z0[0]
        model.z2 = z0[1]

    solver = pyo.SolverFactory('ipopt')
    results = solver.solve(model)

    return np.array([pyo.value(model.z1), pyo.value(model.z2)]), pyo.value(mod
el.obj)

# call the function:
zOpt, JOpt = NLP1()
print('zOpt = ', zOpt)
print('JOpt = ', JOpt)
```

```
zOpt =  [ 0.23308129 -0.52543847]
JOpt =  -0.02959484805254975
```

Part (b)

Show the outputs of your function `NLP1` for 10 random initial guesses, drawing them from a uniform random distribution across your feasible set.

In [7]:
```
## with random initialization:
# if the initialization is way off, the solver can't find a solution
z1 = []
z2 = []
J = []
solver = pyo.SolverFactory('ipopt')
for _ in range(10):

    z1_init = np.random.uniform(low=-1.0, high=1.0)
    z2_init = np.random.uniform(low=-1.0, high=1.0)

    zOpt, JOpt = NLP1([z1_init, z2_init])

    z1.append(zOpt[0])
    z2.append(zOpt[1])
    J.append(JOpt)

print('z1Opt=', z1)
print('z2Opt=', z2)
print('opt_value=', J)
```

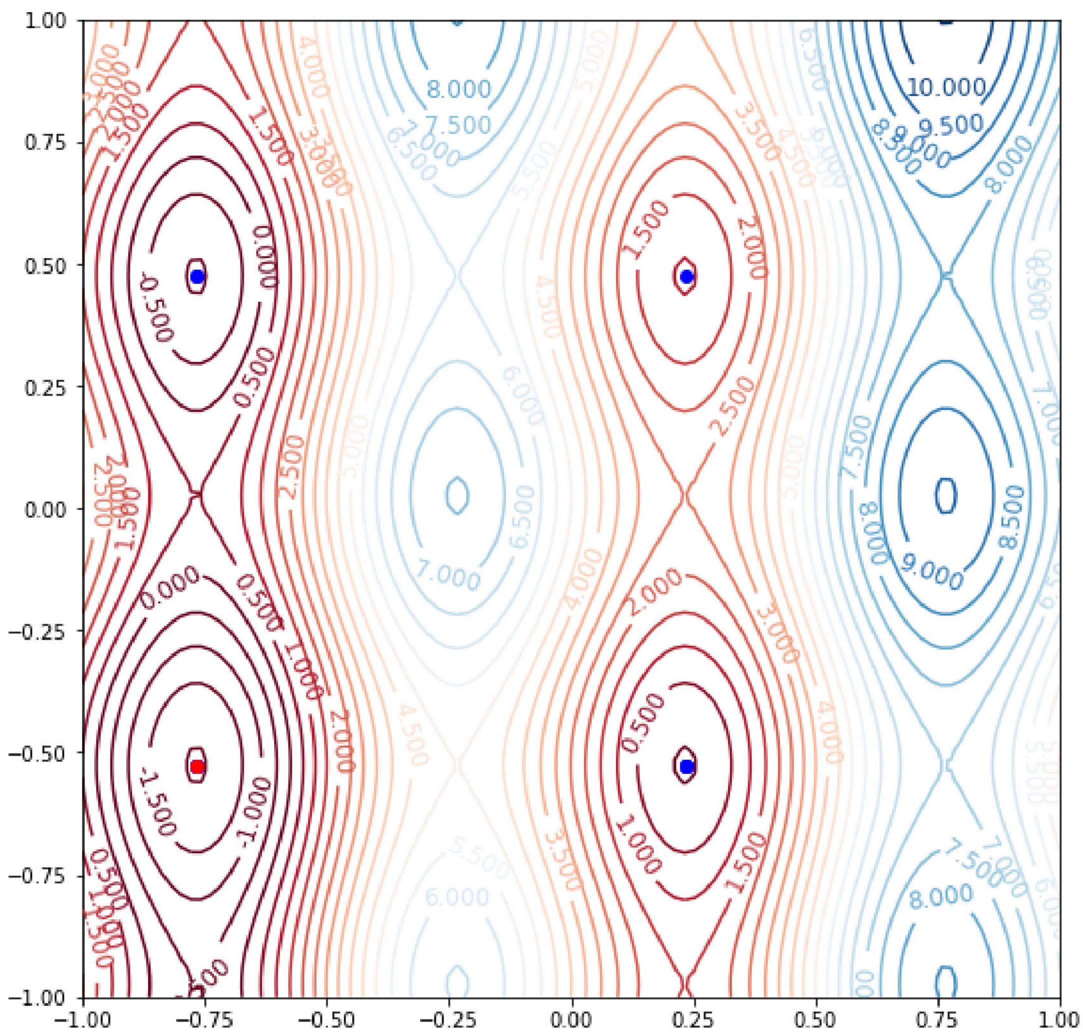
```
z1Opt= [-0.7669187105942515, 0.2330812893159912, -0.7669187105942515, -0.7669
187105942515, 0.2330812893159912, -0.7669187105942515, 0.2330812893159912, 0.
2330812893159912, -0.7669187105942515, 0.2330812893159912]
z2Opt= [-0.5254384707730885, -0.5254384707730885, -0.5254384707730885, 0.4745
6152905478827, 0.4745615290547882, 0.47456152905478827, -0.5254384707730885,
-0.5254384707730885, -0.5254384707730885, -0.5254384707730885]
opt_value= [-2.02959484805255, -0.029594848052550193, -2.02959484805255, -1.0
295948480525496, 0.9704051519474498, -1.0295948480525496, -0.0295948480525501
93, -0.029594848052550193, -2.02959484805255, -0.029594848052550193]
```

Part (c)

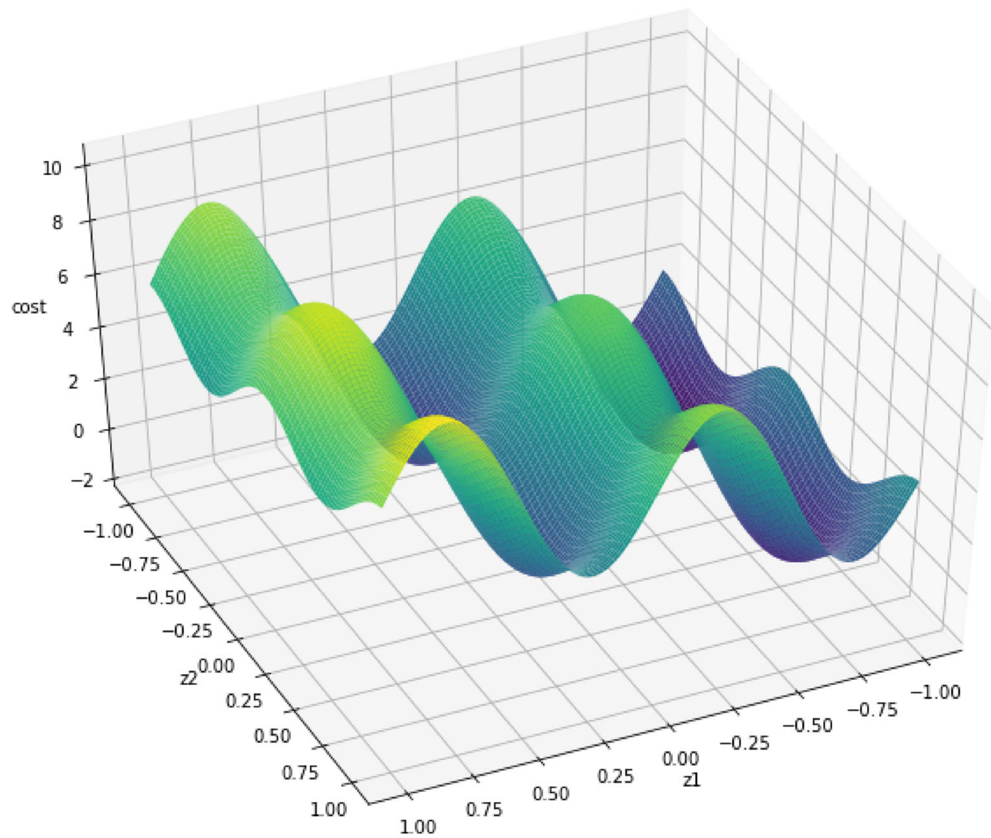
Print out a plot of the cost function contour, mark out the initial guesses from Part (b) and the optimal solutions. Show whether the obtained solutions are global or local optima. Finally, plot the cost function in a 3D plot as well. Provide printed plots as well as your code.

In [8]:

```python
import matplotlib.pyplot as plt

z1_opt = z1
z2_opt = z2

fig, ax = plt.subplots(figsize=(12,9))
z = np.linspace(-1, 1, 100)
z1, z2 = np.meshgrid(z, z)
C = 3*np.sin(-2*np.pi*z1)+ 2*z1 + 4 + np.cos(2*np.pi*z2) + z2
CS = ax.contour(z1, z2, C, cmap=plt.cm.RdBu, vmin=abs(C).min(), vmax=abs(C).ma
x(), levels=30)
ax.clabel(CS, inline=1, fontsize=12)
ax.axis('square')
ax.scatter(z1_opt, z2_opt, c='b')
idx = np.argmin(J)
ax.scatter(z1_opt[idx], z2_opt[idx] , c='r')
plt.show()
```



In [9]:
```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

fig = plt.figure(figsize=(12,9))
ax = plt.axes(projection='3d')
z = np.linspace(-1, 1, 100)
z1, z2 = np.meshgrid(z, z)
C = 3*np.sin(-2*np.pi*z1)+ 2*z1 + 4 + np.cos(2*np.pi*z2) + z2
ax.plot_surface(z1, z2, C, rstride=1, cstride=1, cmap='viridis', edgecolor='no
ne')
ax.set_xlabel('z1')
ax.set_ylabel('z2')
ax.set_zlabel('cost')
ax.view_init(45, 65)
plt.show()
```



---

3. Nonlinear Programming II

Using `pyomo`, repeat all parts of Problem 2 but with the optimization problem defined below. Write a function `NLP2` with function declaration line

```
def NLP2(z0):
    return zOpt, JOpt
```

$$\begin{aligned} \min_{z_1,z_2}\ & \log(1+z_1^2) - z_2 \\ \text{s.t.}\ & -(1+z_1^2)^2 + z_2^2 = 4 \end{aligned}$$

Part (a)

```
In [10]: import numpy as np
import pyomo.environ as pyo
# import logging
# logging.getLogger('pyomo.core').setLevel(logging.ERROR)

def const_rule(model):
    return (- (1 + model.z1 * model.z1)**2 + model.z2**2 == 4)  # For equality
constraint a rule is defined.
```

```
In [11]: ## without initialization:

def NLP2(z0=[]):

    model = pyo.ConcreteModel()
    model.z1 = pyo.Var()
    model.z2 = pyo.Var()
    model.obj = pyo.Objective(expr = pyo.log(1 + model.z1**2) - model.z2)
    model.constraint = pyo.Constraint(rule = const_rule)

    if z0:
        model.z1 = z0[0]
        model.z2 = z0[1]

    solver = pyo.SolverFactory('ipopt')
    results = solver.solve(model)

    return np.array([pyo.value(model.z1), pyo.value(model.z2)]), pyo.value(mod
el.obj)

# call the function:
zOpt, JOpt = NLP2()
print('zOpt = ', zOpt)
print('JOpt = ', JOpt)
```

```
zOpt =  [0.         2.23606798]
JOpt =  -2.23606797749979
```

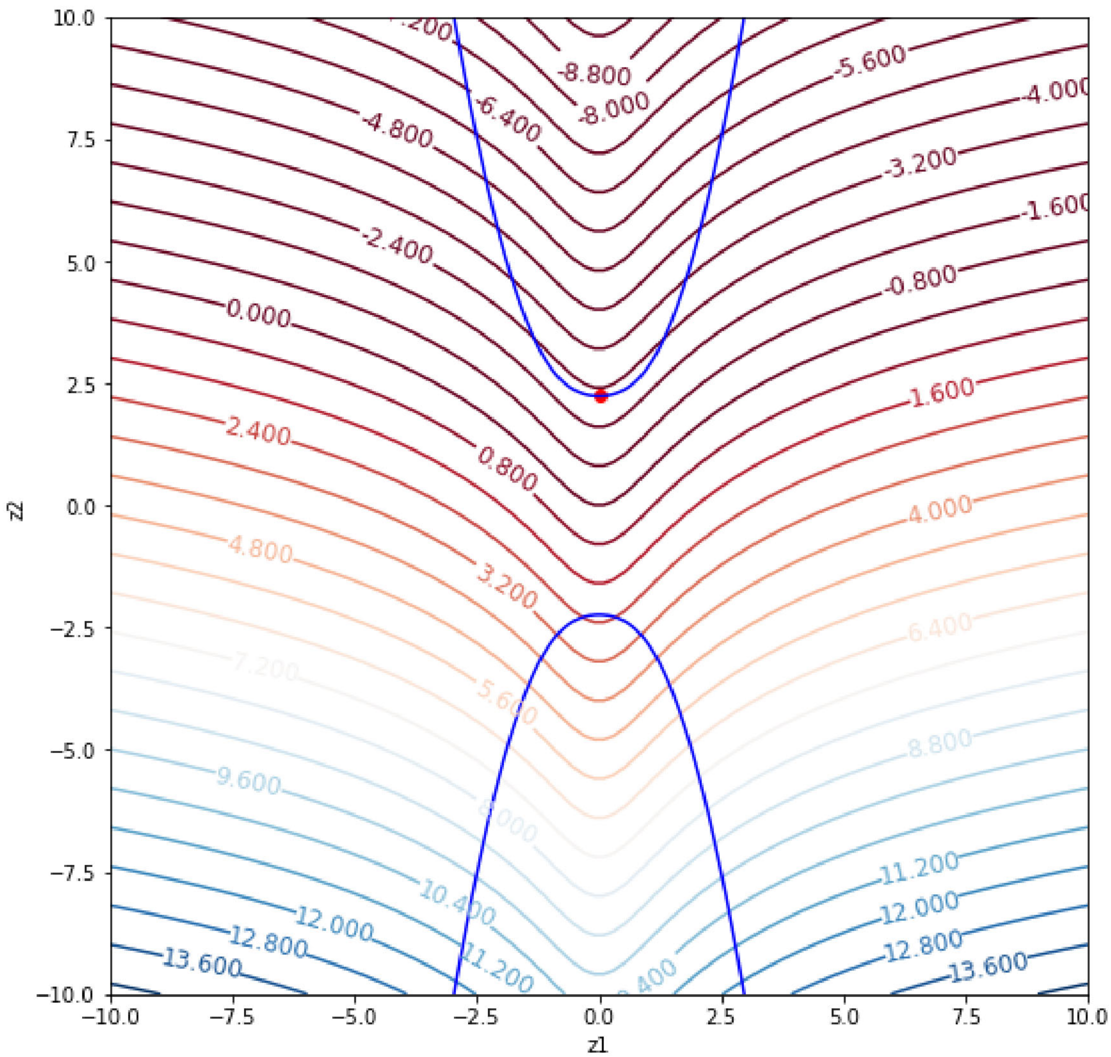
Part (b)

```python
In [12]: ## with random initialization:
         # if the initialization is way off, the solver can't find a solution
         z1 = []
         z2 = []
         J = []
         solver = pyo.SolverFactory('ipopt')
         for _ in range(10):

             z1_init = np.random.uniform(low=-1.0, high=1.0)
             z2_init = np.sqrt(4 + (1 + z1_init**2)**2)

             zOpt, JOpt = NLP2([z1_init, z2_init])

             z1.append(zOpt[0])
             z2.append(zOpt[1])
             J.append(JOpt)

         print('z1Opt=', z1)
         print('z2Opt=', z2)
         print('opt_value=', J)
```

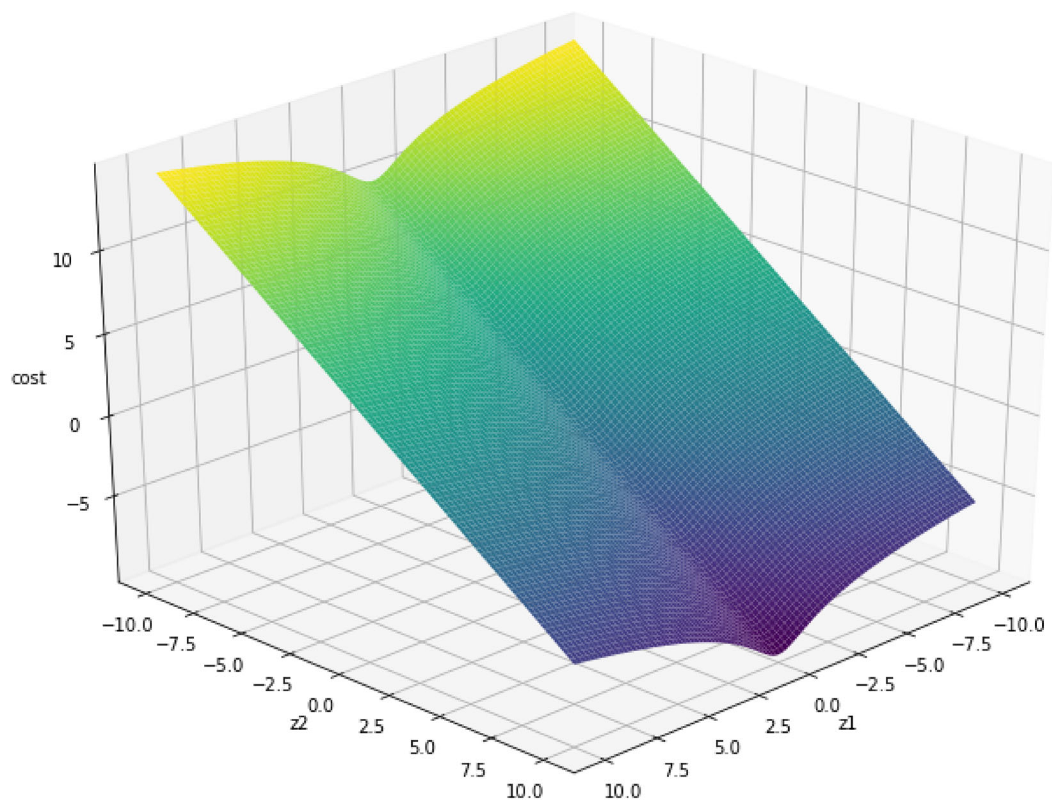
```
z1Opt= [9.384813389804706e-13, -1.5962975665580263e-14, -3.5744708121359314e-
17, -2.5986399709797248e-14, -1.9828323858316847e-14, -1.4216594877914561e-1
1, -4.75793469683158e-13, 5.95024953618697e-11, -1.0328603265318911e-14, 1.26
60333037869042e-13]
z2Opt= [2.2360679774990304, 2.2360679774997814, 2.236067977499794, 2.23606797
74997774, 2.23606797749978, 2.2360679774681973, 2.236067977498724, 2.23606797
7425389, 2.2360679774997863, 2.2360679774997387]
opt_value= [-2.2360679774990304, -2.2360679774997814, -2.236067977499794, -2.
2360679774997774, -2.23606797749978, -2.2360679774681973, -2.236067977498724,
-2.236067977425389, -2.2360679774997863, -2.2360679774997387]
```

Part (c)

In [13]:
```
import matplotlib.pyplot as plt

fig, ax = plt.subplots(figsize=(12,9))
z = np.linspace(-10, 10, 100)
z1, z2 = np.meshgrid(z, z)
C = np.log(1 + z1**2) - z2
CS = ax.contour(z1, z2, C, cmap=plt.cm.RdBu, vmin=abs(C).min(), vmax=abs(C).ma
x(), levels=30)
ax.clabel(CS, inline=1, fontsize=12)
ax.axis('square')
z1 = np.linspace(-3, 3, 100)
z2 = np.sqrt(4 + (1 + z1**2)**2)
ax.plot(z1, z2, color='b')
z2 = -np.sqrt(4 + (1 + z1**2)**2)
ax.plot(z1, z2, color='b')
plt.xlabel('z1')
plt.ylabel('z2')
ax.scatter(0.0, np.sqrt(5), c='r')
plt.show()
```



In [14]:
```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

fig = plt.figure(figsize=(12,9))
ax = plt.axes(projection='3d')
z = np.linspace(-10, 10, 100)
z1, z2 = np.meshgrid(z, z)
C = np.log(1 + z1**2) - z2
ax.plot_surface(z1, z2, C, rstride=1, cstride=1, cmap='viridis', edgecolor='no
ne')
ax.set_xlabel('z1')
ax.set_ylabel('z2')
ax.set_zlabel('cost')
ax.view_init(30, 45)
plt.show()
```



4. Mixed-integer Programming

Use `pyomo` to solve the two following optimization problems. Write individual functions for each optimization problem.

Part (a)

Write a function `MIPa`, with function declaration line

```
def MIPa():
    return zOpt, JOpt
```

Hint: Use `pyo.Integers` to define integer decision variables.

In [15]:
```python
def MIPa():

    model = pyo.ConcreteModel()

    model.z = pyo.Var([1, 2], within = pyo.Integers)

    model.obj = pyo.Objective(expr = -6*model.z[1] - 5*model.z[2])

    model.Constraint1 = pyo.Constraint(expr = model.z[1] + 4*model.z[2] <= 16)
    model.Constraint2 = pyo.Constraint(expr = 6*model.z[1] + 4*model.z[2] <= 2
8)
    model.Constraint3 = pyo.Constraint(expr = 2*model.z[1] - 5*model.z[2] <= 6
)
    model.Constraint4 = pyo.Constraint(expr = (0, model.z[1], 10))
    model.Constraint5 = pyo.Constraint(expr = (0, model.z[2], 10))

    solver = pyo.SolverFactory('glpk')
    results = solver.solve(model)

    return np.array([pyo.value(model.z[1]), pyo.value(model.z[2])]), pyo.value
(model.obj)

# call the fucnrion:
zOpt, JOpt = MIPa()
print('zOpt = ', zOpt)
print('JOpt = ', JOpt)
```

```
zOpt =  [4. 1.]
JOpt =  -29.0
```

$$
\begin{aligned}
\min_{z_1,z_2} \ & -6z_1 - 5z_2 \\
\text{s.t.} \ & z_1 + 4z_2 \leq 16 \\
& 6z_1 + 4z_2 \leq 28 \\
& 2z_1 - 5z_2 \leq 6 \\
& 0 \leq z_1 \leq 10 \\
& 0 \leq z_2 \leq 10 \\
& z_1, z_2 \in \mathbf{Z}, \text{(integer)}
\end{aligned}
$$

Part (b)

Write a function `MIPb`, with function declaration line

```
def MIPb():
    return zOpt, JOpt
```

$$
\begin{aligned}
\min_{z_1,z_2} \ & -z_1 - 2z_2 \\
\text{s.t.} \ & \text{either } 3z_1 + 4z_2 \leq 12 \ \text{ or } 4z_1 + 3z_2 \leq 12 \\
& z_1 \geq 0 \\
& z_2 \geq 0
\end{aligned}
$$

Hint: Use `pyo.Binary` to define binary decision variables.

In [16]:

```
def MIPb():
    model = pyo.ConcreteModel()

    model.z = pyo.Var([1,2], domain=pyo.NonNegativeReals)
    model.bin_var = pyo.Var(within=pyo.Binary)

    model.obj = pyo.Objective(expr = -model.z[1] - 2*model.z[2])

    model.Constraint1 = pyo.Constraint(expr = 3*model.z[1] + 4*model.z[2] <= 1
2 + model.bin_var*10000)
    model.Constraint2 = pyo.Constraint(expr = 4*model.z[1] + 3*model.z[2] <= 1
2 + (1-model.bin_var)*10000)

    # solver = pyo.SolverFactory('mindtpy').solve(model, mip_solver='glpk', nl
p_solver='ipopt')
    solver = pyo.SolverFactory('glpk')
    results = solver.solve(model)

    return np.array([pyo.value(model.z[1]), pyo.value(model.z[2])]), pyo.value
(model.obj)

# call the fucnrion:
zOpt, JOpt = MIPb()
print('zOpt = ', zOpt)
print('JOpt = ', JOpt)
```

```
zOpt =  [3.97903932e-13 4.00000000e+00]
JOpt =  -8.000000000000398
```

---

5. KKT Conditions I

The following problems also include some of the examples from Homework 2. Write functions that return a single logical variable that reflects whether all KKT conditions are satisfied for the constrained problem (i.e., 1 stands for all KKT conditions satisfied). Please submit your solutions as individual functions for each of the 4 parts.

Part (a)

$$\begin{aligned} \min_{z_1,z_2} \quad & -5z_1 - 7z_2 \\ \text{s.t.} \quad & -3z_1 + 2z_2 \leq 30 \\ & -2z_1 + z_2 \leq 12 \\ & z_1 \geq 0 \\ & z_2 \geq 0 \end{aligned}$$

Write a function **`LPQPkkta`**, with function declaration line

```
def LPQPkkta():
    return KKTsat
```

In [17]:

```
from pyomo.opt import SolverStatus, TerminationCondition
import numpy as np
import pyomo.environ as pyo
def LPQPkkta():

    KKTsat = False
    model = pyo.ConcreteModel()

    model.z = pyo.Var([1,2], domain = pyo.NonNegativeReals)

    model.obj = pyo.Objective(expr = -5*model.z[1] - 7*model.z[2])

    model.constraint1 = pyo.Constraint(expr = -3*model.z[1] + 2*model.z[2] <=
30.0)
    model.constraint2 = pyo.Constraint(expr = -2*model.z[1] + model.z[2] <= 1
2.0)

    model.dual = pyo.Suffix(direction=pyo.Suffix.IMPORT)

    solver = pyo.SolverFactory('cbc')

    results = solver.solve(model)

    if results.solver.termination_condition != TerminationCondition.optimal:
        KKTsat = False
    else:
        zOpt = np.array([pyo.value(model.z[1]), pyo.value(model.z[2])])
        A = np.array([[-3, 2],[-2, 1],[-1, 0],[0, -1]])
        b = np.array([30, 12, 0, 0])

        y = []
        for c in model.component_objects(pyo.Constraint, active=True):
            print ("Constraint", c)
            for index in c:
        #         print ("      ", index, model.dual[c[index]])
                y.append(model.dual[c[index]])


        y = np.asarray(y)
        flag_ineq = np.all(A@zOpt <= b)
        flag_dual = np.all(y >= 0)
        flag_cs = np.all(y*(A@zOpt-b) == 0)
        flag_grad = np.all([-5, -7] + y.T@A == 0)
        KKT_conditions = np.array([flag_ineq, flag_dual, flag_cs, flag_grad])
        if all(KKT_conditions == 1):
            KKTsat = True
        else:
            KKTsat = False
    return KKTsat

# Calling the function
KKTsat = LPQPkkta()
print(KKTsat)
```

```
WARNING: Loading a SolverResults object with a warning status into
    model=unknown;
        message from solver=<undefined>
False
```

Part (b)

$$\begin{aligned} \min_{x,y,z} \quad & x + y + z \\ \text{subject to } & 2 \leq x \\ & -1 \leq y \\ & -3 \leq z \\ & x - y + z \geq 4 \end{aligned}$$

Write a function `LPQPkktb`, with function declaration line

In [18]:

```
import pyomo.environ as pyo
import numpy as np

model = pyo.ConcreteModel()
model.x = pyo.Var()
model.y = pyo.Var()
model.z = pyo.Var()

model.Obj = pyo.Objective(expr = model.x + model.y + model.z)
# model.constraint1 = pyo.Constraint(expr = -2 <= model.x)
# model.constraint2 = pyo.Constraint(expr = -1 <= model.y)
# model.constraint3 = pyo.Constraint(expr = -3 <= model.z)
# model.constraint4 = pyo.Constraint(expr = model.x - model.y + model.z >= 4)

model.constraint1 = pyo.Constraint(expr = -model.x-2 <= 0)
model.constraint2 = pyo.Constraint(expr = -model.y-1 <= 0)
model.constraint3 = pyo.Constraint(expr = -model.z-3 <= 0)
model.constraint4 = pyo.Constraint(expr = -model.x + model.y - model.z <= -4)

model.dual = pyo.Suffix(direction=pyo.Suffix.IMPORT)

solver = pyo.SolverFactory('cbc')
results = solver.solve(model)

if results.solver.termination_condition != TerminationCondition.optimal:
    KKTsat = False
else:
    A = np.array([[-1, 0, 0], [0, -1, 0], [0, 0, -1], [-1, 1, -1]])   # 2D arr
ay
    b = np.array([2,1,3,-4])   # 1D array
    zOpt = np.array([pyo.value(model.x), pyo.value(model.y), pyo.value(model.z
)])
    y = []
    for c in model.component_objects(pyo.Constraint, active=True):
        print ("Constraint", c)
        for index in c:
            y.append(-model.dual[c[index]])
    y = np.asarray(y)
    flag_ineq = np.all(A@zOpt <= b)
    flag_dual = np.all(y >= 0)
    flag_cs = np.all(y*(A@zOpt-b) == 0)
    flag_grad = np.all([1,1,1] + y.T@A == 0)
    KKT_conditions = np.array([flag_ineq, flag_dual, flag_cs, flag_grad])
    if all(KKT_conditions == 1):
        KKTsat = True
    else:
        KKTsat = False

print(KKTsat)

# print('dual 1:', model.dual[model.constraint1])
# print('dual 2:', model.dual[model.constraint2])
# print('dual 3:', model.dual[model.constraint3])
# print('dual 4:', model.dual[model.constraint4])
model.pprint()
```

```
Constraint constraint1
Constraint constraint2
Constraint constraint3
Constraint constraint4
True
3 Var Declarations
    x : Size=1, Index=None
        Key  : Lower : Value : Upper : Fixed : Stale : Domain
        None :  None :   6.0 :  None : False : False :  Reals
    y : Size=1, Index=None
        Key  : Lower : Value : Upper : Fixed : Stale : Domain
        None :  None :  -1.0 :  None : False : False :  Reals
    z : Size=1, Index=None
        Key  : Lower : Value : Upper : Fixed : Stale : Domain
        None :  None :  -3.0 :  None : False : False :  Reals

1 Objective Declarations
    Obj : Size=1, Index=None, Active=True
        Key  : Active : Sense    : Expression
        None :   True : minimize : x + y + z

4 Constraint Declarations
    constraint1 : Size=1, Index=None, Active=True
        Key  : Lower : Body    : Upper : Active
        None :  -Inf : - x - 2 :   0.0 :   True
    constraint2 : Size=1, Index=None, Active=True
        Key  : Lower : Body    : Upper : Active
        None :  -Inf : - y - 1 :   0.0 :   True
    constraint3 : Size=1, Index=None, Active=True
        Key  : Lower : Body    : Upper : Active
        None :  -Inf : - z - 3 :   0.0 :   True
    constraint4 : Size=1, Index=None, Active=True
        Key  : Lower : Body        : Upper : Active
        None :  -Inf : - x + y - z :  -4.0 :   True

1 Suffix Declarations
    dual : Direction=Suffix.IMPORT, Datatype=Suffix.FLOAT
        Key         : Value
        constraint1 :   0.0
        constraint2 :  -2.0
        constraint3 :  -0.0
        constraint4 :  -1.0

9 Declarations: x y z Obj constraint1 constraint2 constraint3 constraint4 dua
l
```

Part (c)

$$\begin{aligned} \min_{z_1,z_2} \ & z_1^2 + z_2^2 \\ \text{s.t. } & z_1 \leq -3 \\ & z_2 \leq 4 \\ & 0 \geq 4z_1 + 3z_2 \end{aligned}$$

Write a function `LPQPkktc`, with function declaration line

```
def LPQPkktc():
    return KKTsat
```

In [19]:
```
def LPQPkktc():

    threshold = 1e-5 # This is the threshold to specify the values close to ze
ro.
    model = pyo.ConcreteModel()
    model.z = pyo.Var([1,2])
    model.obj = pyo.Objective(expr = model.z[1]**2 + model.z[2]**2)


    model.Constraint1 = pyo.Constraint(expr = model.z[1] <= -3)
    model.Constraint2 = pyo.Constraint(expr = model.z[2] <= 4)
    model.Constraint3 = pyo.Constraint(expr = 4*model.z[1] + 3*model.z[2] <= 0
)

    model.dual = pyo.Suffix(direction=pyo.Suffix.IMPORT)

    solver = pyo.SolverFactory('ipopt')
    results = solver.solve(model)

    if results.solver.termination_condition != TerminationCondition.optimal:
        KKTsat = False
    else:
        A = np.array([[1, 0],
                      [0, 1],
                      [4, 3]])
        b = np.array([-3, 4, 0])
        zOpt = np.array([pyo.value(model.z[1]), pyo.value(model.z[2])])
        y = []
        for c in model.component_objects(pyo.Constraint, active=True):
            print ("Constraint", c)
            for index in c:
                y.append(-model.dual[c[index]]) # The duals in pyomo are defin
ed as -y<=0, so we add a negative sign.
                print(model.dual[c[index]])
        y = np.asarray(y)
        for i in range(len(y)):
            if (y[i] < threshold) & (y[i] > -threshold):
                y[i] = 0
        flag_ineq = np.any(np.all(A@zOpt <= b + threshold) | np.all(A@zOpt <=
b - threshold))
        flag_dual = np.all(y >= 0)
        flag_cs = np.all(np.multiply(y,(A@zOpt-b)) < threshold) & np.all(np.mu
ltiply(y,(A@zOpt-b)) > -threshold)
        grad_lagrangian = [2*zOpt[0],2*zOpt[1]] + y.T@A
        for i in range(len(grad_lagrangian)):
            if (grad_lagrangian[i] < threshold) & (grad_lagrangian[i] > -thres
hold):
                grad_lagrangian[i] = 0
        flag_grad = np.all(grad_lagrangian == 0)
        KKT_conditions = np.array([flag_ineq, flag_dual, flag_cs, flag_grad])
        if all(KKT_conditions == 1):
            KKTsat = True
        else:
            KKTsat = False
        return KKTsat
```

```python
# Calling the function
KKTsat = LPQPkktc()
print(KKTsat)
```

```
Constraint Constraint1
-5.999999939999967
Constraint Constraint2
-6.265039564695445e-10
Constraint Constraint3
-2.0883369936244261e-10
True
```

6. KKT Conditions II

Write a function `NLPkkt1`, which solves the optimization problem defined below, with function declaration line

```python
def NLPkkt1(z0):
    return kktsat
```

where the input `z0` is the initial guess for your optimizer. The function should output a single logical variable that reflects whether all KKT conditions are satisfied for the constrained nonlinear problem (i.e., 1 stands for all KKT conditions satisfied).

$$
\begin{aligned}
\min_{z_1, z_2} \ & 3\sin(-2\pi z_1) + 2z_1 + 4 + \cos(2\pi z_2) + z_2 \\
\text{s.t.} \ & -1 \leq z_1 \leq 1 \\
& -1 \leq z_2 \leq 1
\end{aligned}
$$

In [20]:
```
## without initialization:
from pyomo.opt import SolverStatus, TerminationCondition
import numpy as np
import pyomo.environ as pyo

threshold = 1e-5 # This is the threshold to specify the values close to zero.

def NLPkkt1(z0=[]):

    model = pyo.ConcreteModel()
    model.z1 = pyo.Var()
    model.z2 = pyo.Var()
    model.obj = pyo.Objective(expr = 3*pyo.sin(-2*np.pi*model.z1)+ 2*model.z1
+ 4 + pyo.cos(2*np.pi*model.z2) + model.z2)
    # model.constraint1 = pyo.Constraint(expr = (-1, model.z1, 1))
    # model.constraint2 = pyo.Constraint(expr = (-1, model.z2, 1))
    model.constraint1 = pyo.Constraint(expr = -1 - model.z1 <= 0)
    model.constraint2 = pyo.Constraint(expr = model.z1 - 1 <= 0)
    model.constraint3 = pyo.Constraint(expr = -1 - model.z2 <= 0)
    model.constraint4 = pyo.Constraint(expr = model.z2 - 1 <= 0)

    if z0:
        model.z1 = z0[0]

    model.dual = pyo.Suffix(direction=pyo.Suffix.IMPORT)

    solver = pyo.SolverFactory('ipopt')
    results = solver.solve(model)


    if results.solver.termination_condition != TerminationCondition.optimal:
        KKTsat = False
    else:
        A = np.array([[1, 0],
                      [-1, 0],
                      [1, 0],
                      [-1, 0]])
        b = np.array([1, 1, 1, 1])
        zOpt = np.array([pyo.value(model.z1), pyo.value(model.z2)])
        y = []
        for c in model.component_objects(pyo.Constraint, active=True):
            print ("Constraint", c)
            for index in c:
                y.append(model.dual[c[index]])
        y = np.asarray(y)
        for i in range(len(y)):
            if (y[i] < threshold) & (y[i] > -threshold):
                y[i] = 0
        flag_ineq = np.all(A@zOpt <= b)
        flag_dual = np.all(y >= 0)
        flag_cs = np.all(np.multiply(y,(A@zOpt-b)) == 0) # np.multiply is used
for element-wise multiplication
        grad_lagrangian = [-6*np.pi*np.cos(-2*np.pi*zOpt[0])+2,-2*np.pi*np.sin
(2*np.pi*zOpt[1])+1] + y.T@A
        for i in range(len(grad_lagrangian)):
            if (grad_lagrangian[i] < threshold) & (grad_lagrangian[i] > -thres
```

```
hold):
                grad_lagrangian[i] = 0
        flag_grad = np.all(grad_lagrangian == 0)
        KKT_conditions = np.array([flag_ineq, flag_dual, flag_cs, flag_grad])
        if all(KKT_conditions == 1):
            KKTsat = True
        else:
            KKTsat = False
    return KKTsat

# Calling the function
KKTsat = NLPkkt1()
print(KKTsat)
```

```
Constraint constraint1
Constraint constraint2
Constraint constraint3
Constraint constraint4
True
```