Sharif University of Technology
Electrical Engineering Department

Convex Optimization CHW 3

Amir Hossein Yari
99102507

July 4, 2023

Contents

1. Optimal evacuation planning

The optimization problem is :

$$\begin{array}{lll} \textit{minimize} & \displaystyle\sum_{t=1}^{T}(r^T q_t + q_t^T\, diag(s)\, q_t) + \sum_{t=1}^{T-1}(\tilde{r}^T |f_t| + f_t^T\, diag(\tilde{s}) f_t) & \\ \textit{subject to} & q_{t+1} = A\, f_t + q_t & t = 1, \dots, T-1 \\ & 0 \preceq q_t \preceq Q & t = 2, \dots, T \\ & |f_t| \preceq F & t = 1, \dots, T-1 \end{array}$$

Variables are $q_2, \dots, q_T$ and $f_1, \dots, f_{T-1}$.
Objective function is sum of convex functions, So it is convex. Also equality constraints are affine and inequality constraints are linear, So convex. So the optimization problem is convex optimization problem.
For simpler coding, we use the following simplification.

$$q_t^T\, diag(s)\, q_t = s^T\, q_t^2 \quad , \quad f_t^T\, diag(\tilde{s}) f_t = \tilde{s}^T\, f_t^2$$

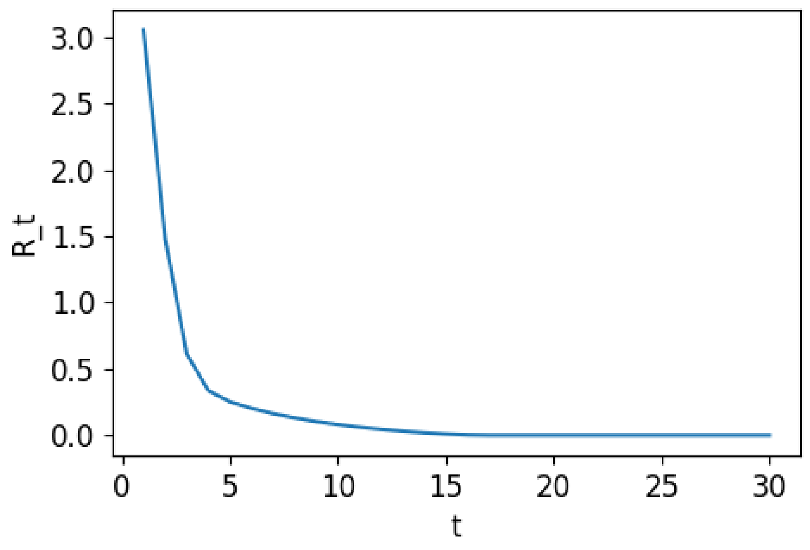
So the problem can be rewritten as follows :

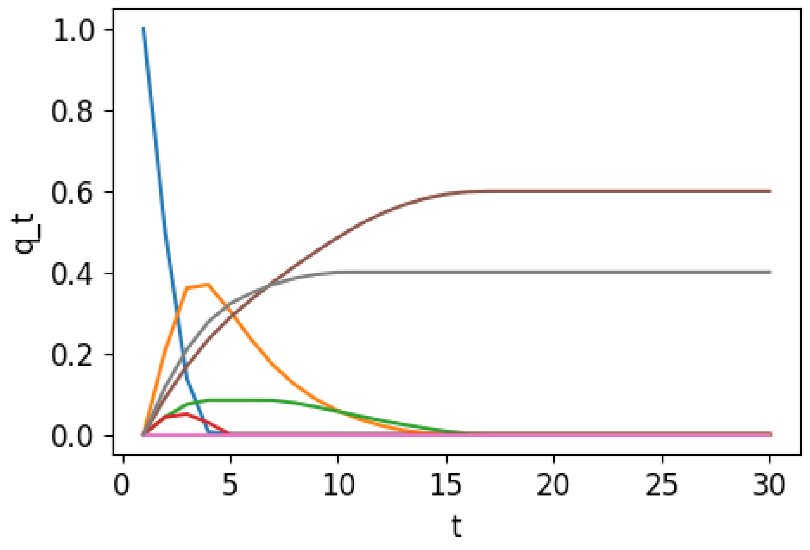
$$\begin{array}{lll} \textit{minimize} & \displaystyle\sum_{t=1}^{T}(r^T q_t + s^T\, q_t^2) + \sum_{t=1}^{T-1}(\tilde{r}^T |f_t| + \tilde{s}^T\, f_t^2) & \\ \textit{subject to} & q_{t+1} = A\, f_t + q_t & t = 1, \dots, T-1 \\ & 0 \preceq q_t \preceq Q & t = 2, \dots, T \\ & |f_t| \preceq F & t = 1, \dots, T-1 \end{array}$$

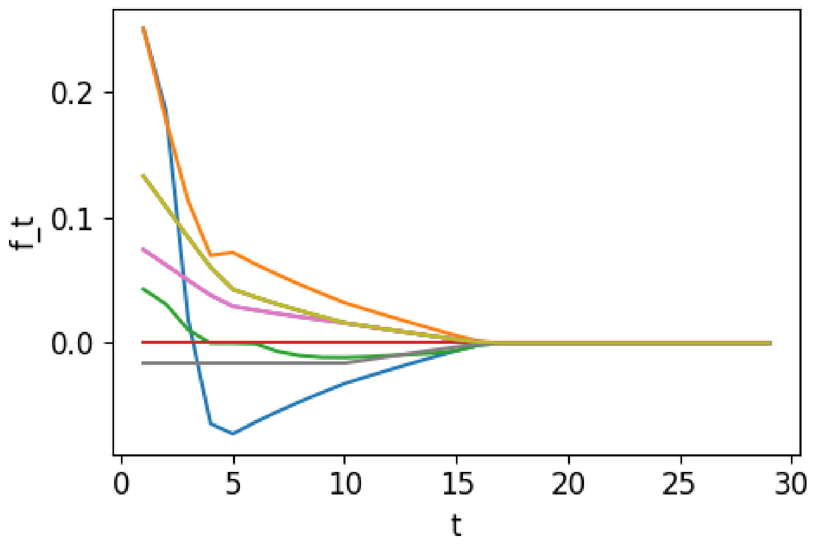
```
# Import required library
import numpy as np
import cvxpy as cvx
import matplotlib.pyplot as plt
import matplotlib
from opt_evac_data import *

# Initialize parameter
n,m = A.shape
f = cvx.Variable([m,T-1])
q = cvx.Variable([n,T])
node_risk = q.T @ r + cvx.square(q).T @ s
edge_risk = cvx.hstack([cvx.abs(f).T @ rtild + cvx.square(f).T @ stild, np.zeros(1)])
risk = node_risk + edge_risk

# Solve problem
constr = [q[:,0] == q1, q[:,1:] == A @ f + q[:,:-1], 0 <= q, q <= np.tile(Q,(T,1)).T, cvx
                                            .abs(f) <= np.tile(F,(T-1,1)).T]
p = cvx.Problem(cvx.Minimize(sum(risk)), constr)
p.solve(verbose=True, solver=cvx.ECOS)
arr = lambda _: np.array(_.value)
q, f, risk, node_risk = map(arr, (q, f, risk, node_risk))

# Print output
print("Total risk is ", p.value)
print("Evacuated at t =", (node_risk <= 1e-4).nonzero()[0][0] + 1)

# Plotting
plt.rcParams['text.usetex'] = False
plt.rcParams['font.size'] = 12
fig, axs = plt.subplots(3,1,figsize=(7,15))
axs[0].plot(np.arange(1,T+1), risk)
axs[0].set_ylabel("R_t")
axs[0].set_xlabel("t")
axs[1].plot(np.arange(1,T+1), q.T)
axs[1].set_ylabel("q_t")
axs[1].set_xlabel("t")
axs[2].plot(np.arange(1,T), f.T)
axs[2].set_ylabel("f_t")
axs[2].set_xlabel("t")
fig.tight_layout()
plt.show()
```

output

```
Total risk is 6.589671916236878
Evacuated at t = 17
```


R_t
t


q_t
t


f_t
t

2. Optimal circuit design

We must have blended designs. it means that $w = \sum_{i=1}^{K} \theta_i w^{(i)}$ where $\theta \succeq 0$ and $\mathbf{1}^T\theta = 1$. When f is convex function, by Jensen's inequality we can say:

$$f(w) \leq \sum_{i=1}^{K} \theta_i f(w^{(i)})$$

We know that When f is a posynomial function then $g(x) = f(log(e^x))$ is convex. So the functions $logP$, $logA$, $logD$ are convex functions of the variables $x = logw$. According to the above explanation, it can be said:

$$log(w) = \sum_{i=1}^{K} \theta_i \, log(w^{(i)}) \Rightarrow log(P(w)) \leq \sum_{i=1}^{K} \theta_i \, log(P(w^{(i)}))$$

In this method we predict an upper bound on how large the power can be. Also we can say similar inequalities hold for the delay and area.
So the optimization problem is :

Find	θ
Subject to	$\sum_{i=1}^{K} \theta_i \, log(P(w^{(i)}) \leq log(P_{spec})$
.	$\sum_{i=1}^{K} \theta_i \, log(D(w^{(i)}) \leq log(D_{spec})$
.	$\sum_{i=1}^{K} \theta_i \, log(A(w^{(i)}) \leq log(A_{spec})$
.	$\mathbf{1}^T\theta = 1$
.	$\theta \succeq 0$

If the optimization above is feasible, then the blend obtained from the feasible θ must satisfy the design specifications. But if the optimization above is infeasible, then we can say nothing. The design specifications could be infeasible, or feasible.

```python
# Import required library
import numpy as np
import cvxpy as cvx
from blend_design_data import *

# Initialize variable
theta = cvx.Variable(k)

# Solve problem
objective = cvx.Minimize(0)
constraints = [np.log(P) @ theta <= np.log(P_spec)]
constraints += [np.log(D) @ theta <= np.log(D_spec)]
constraints += [np.log(A) @ theta <= np.log(A_spec)]
constraints += [cvx.sum(theta)==1, theta>=0]
cvx.Problem(objective,constraints).solve()

# Create design
w = np.exp(np.log(W) @ theta.value)
print("w = ", w)
print("theta = ", theta.value)
```

Output

w = [2.63859837 3.28069166 2.96934036 3.26617754 2.32426888 3.66532996 2.92976112 3.68571815 3.8940303 3.3972013]
θ = [0.01758686 0.49673786 0.00452331 0.47531367 0.00408995 0.00174836]

3. Filling the covariance matrix

1. A simple way to fill the covariance matrix

if $n_1 = n_2 = n_3 = 1$ and $S = T = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \Rightarrow C_{sim} = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 1 \end{bmatrix}$

The matrix C_{sim} is not positive semidefinite, So it can not be a covariance matrix.

2. Convex optimization problem

if C_{sim} be a positive semidefinite matrix, then it is the solution of optimization problem in question.

3. Solve optimization problem

```
# Import required library
import numpy as np
import cvxpy as cp

# Initialize variable
n1 = n2 = n3 = 1
n = n1 + n2 + n3
S = np.ones([n1+n2, n1+n2])
T = np.ones([n2+n3, n2+n3])
C_sim = np.matrix([[1,1,0],[1,1,1],[0,1,1]])

# Solve problem
C = cp.Variable((n,n), symmetric=True)
objective = cp.Minimize(cp.norm(C - C_sim, "fro")**2)
constraints = [C >> 0]
prob = cp.Problem(objective, constraints)
prob.solve()

# Print output
print("C = \n",C.value)
if all(np.linalg.eigvals(C_sim) >= 0):
print("The matrix C_sim is positive semidefinite.")
else:
print("The matrix C_sim is not positive semidefinite.")

if all(np.round(np.linalg.eigvals(C.value)) >= 0):
print("The matrix recovered C is positive semidefinite.")
else:
print("The matrix recovered C is not positive semidefinite.")
```

output

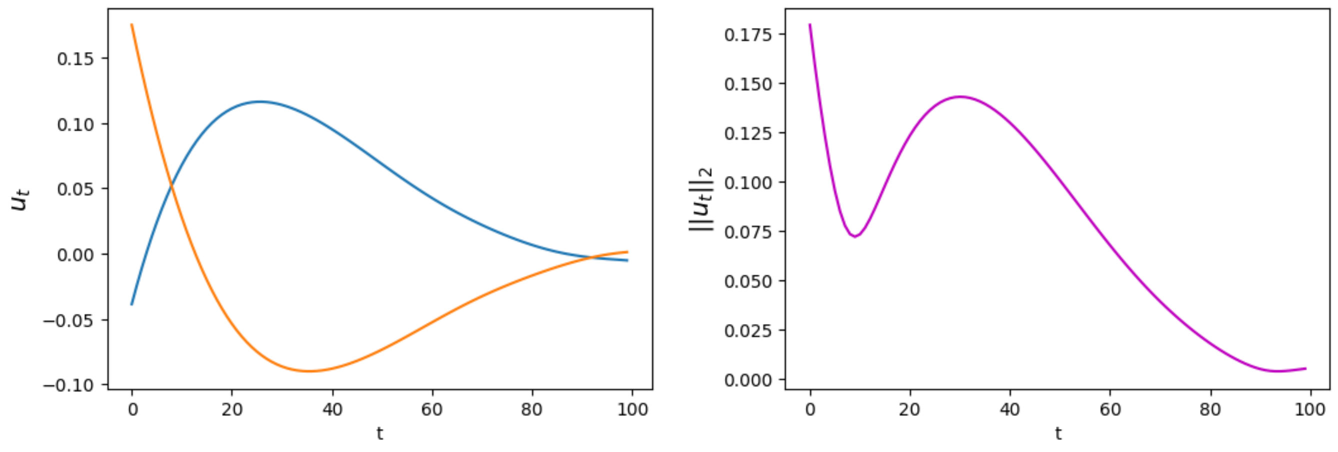
$$C = \begin{bmatrix} 1.10355273 & 0.8535544 & 0.1035527 \\ 0.8535544 & 1.20710542 & 0.8535544 \\ 0.1035527 & 0.8535544 & 1.10355273 \end{bmatrix}$$

The matrix C_{sim} is not positive semidefinite.
The matrix recovered C is positive semidefinite.

4. Control with various objectives

1. $\sum_{t=0}^{T-1} ||\mathbf{u}_t||_2^2$

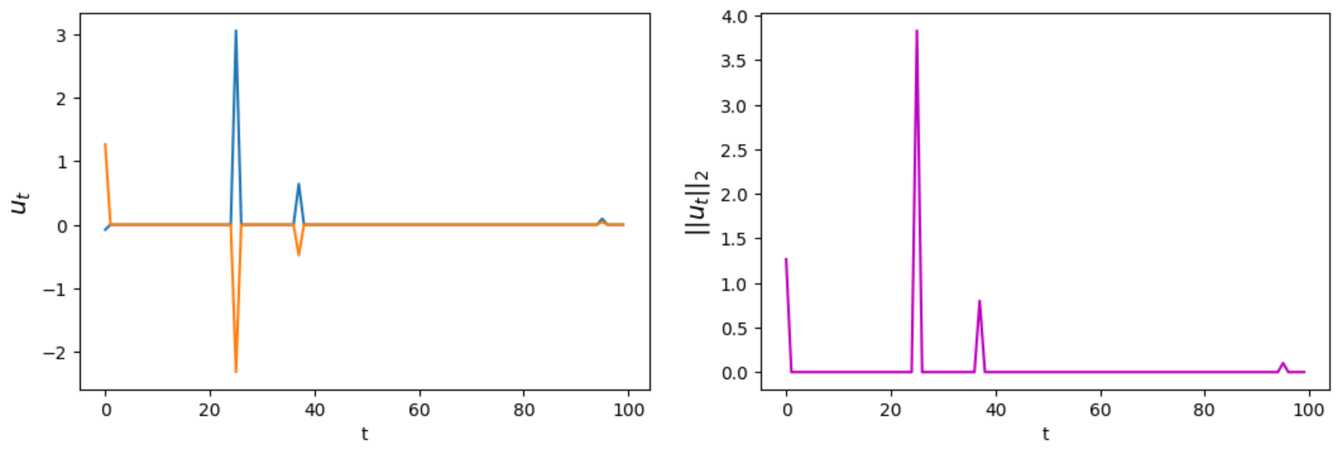
```python
# Import required library
import numpy as np
import cvxpy as cp
import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec
from various_obj_regulator_data import *

# Initialize variable
x = cp.Variable((n,T+1))
u = cp.Variable((m,T))

# Solve problem
objective = cp.Minimize(cp.sum_squares(u))
const  = [x[:,-1] == np.zeros(n)]
const.append(x[:,0] == x_init)
for t in range(1,T+1):
  const.append(x[:,t] == A @ x[:,t-1] + B @ u[:,t-1])
prob = cp.Problem(objective, const)
prob.solve()

# Plotting
fig = plt.figure(figsize=(11,4))
plt.subplot(1,2,1)
plt.plot(u.value.T)
plt.ylabel("$u_t$", fontsize=14)
plt.xlabel("t")
plt.subplot(1,2,2)
plt.xlabel("t")
plt.plot(np.linalg.norm(u.value,axis=0),"m")
plt.ylabel("$||u_t||_2$", fontsize=14)
plt.tight_layout(pad = 2)
```



As we expect the control inputs are small, but not sparse.

2. $\sum_{t=0}^{T-1} ||\mathbf{u}_t||_2$

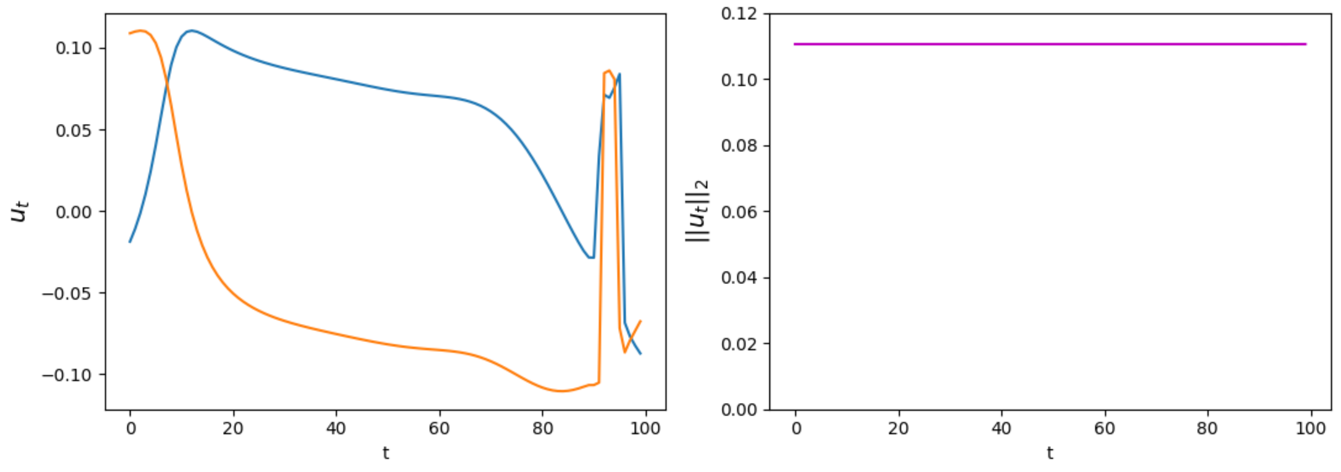
```
# Import required library
import numpy as np
import cvxpy as cp
import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec
from various_obj_regulator_data import *

# Initialize variable
x = cp.Variable((n,T+1))
u = cp.Variable((m,T))

# Solve problem
objective = cp.Minimize(cp.sum(cp.norm(u,2,axis=0)))
const = [x[:,-1] == np.zeros(n)]
const.append(x[:,0] == x_init)
for t in range(1,T+1):
  const.append(x[:,t] == A @ x[:,t-1] + B @ u[:,t-1])
prob = cp.Problem(objective, const)
prob.solve()

# Plotting
plt.figure(figsize=(11,4))
plt.subplot(1,2,1)
plt.plot(u.value.T)
plt.ylabel("$u_t$", fontsize=14)
plt.xlabel("t")
plt.subplot(1,2,2)
plt.xlabel("t")
plt.plot(np.linalg.norm(u.value,axis=0),"m")
plt.ylabel("$||u_t||_2$", fontsize=14)
plt.tight_layout(pad = 2)
```



As you see the control input is sparse. Also both components are nonzero when the control is nonzero.

3. $max_{t=0,\ldots,T-1}||\mathbf{u}_t||_2$

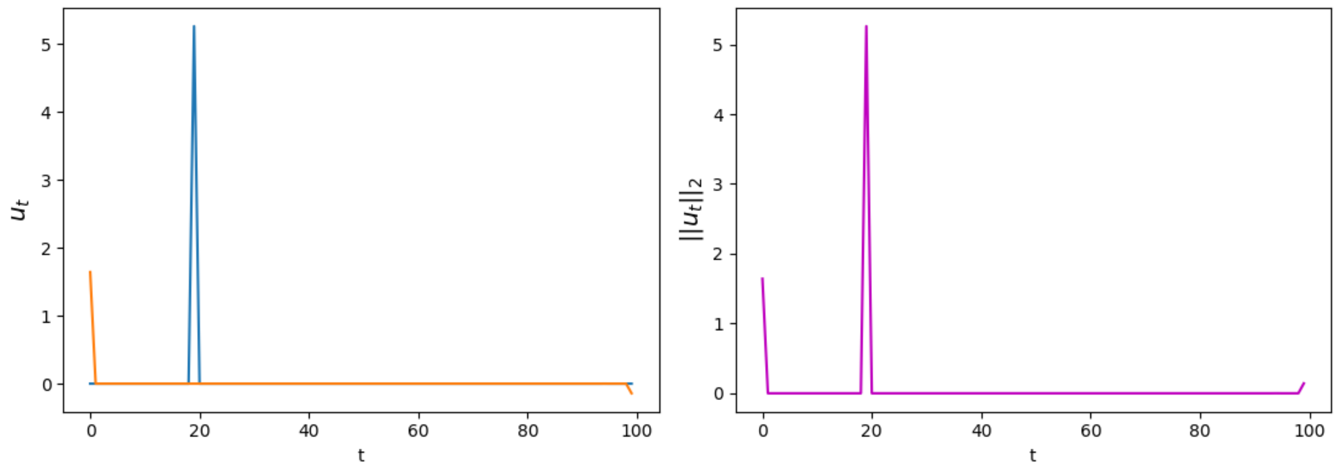
```
# Import required library
import numpy as np
import cvxpy as cp
import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec
from various_obj_regulator_data import *

# Initialize variable
x = cp.Variable((n,T+1))
u = cp.Variable((m,T))

# Solve problem
objective = cp.Minimize(cp.max(cp.norm(u,2,axis=0)))
const = [x[:,-1] == np.zeros(n)]
const.append(x[:,0] == x_init)
for t in range(1,T+1):
  const.append(x[:,t] == A @ x[:,t-1] + B @ u[:,t-1])
prob = cp.Problem(objective, const)
prob.solve()
# Plotting
plt.figure(figsize=(11,4))
plt.subplot(1,2,1)
plt.plot(u.value.T)
plt.ylabel("$u_t$", fontsize=14)
plt.xlabel("t")
plt.subplot(1,2,2)
plt.xlabel("t")
plt.plot(np.linalg.norm(u.value,axis=0),"m")
plt.ylabel("$||u_t||_2$", fontsize=14)
plt.ylim(ymin = 0, ymax = 0.12)
plt.tight_layout()
```



We know that the l_2 norm of the control input is constant. So the direction of the control input changes over time.

4. $\sum_{t=0}^{T-1} ||\mathbf{u}_t||_1$

```
# Import required library
import numpy as np
import cvxpy as cp
import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec
from various_obj_regulator_data import *

# Initialize variable
x = cp.Variable((n,T+1))
u = cp.Variable((m,T))

# Solve problem
objective = cp.Minimize(cp.sum(cp.norm(u,1,axis=0)))
const = [x[:,-1] == np.zeros(n)]
const.append(x[:,0] == x_init)
for t in range(1,T+1):
  const.append(x[:,t] == A @ x[:,t-1] + B @ u[:,t-1])
prob = cp.Problem(objective, const)
prob.solve()

# Plotting
plt.figure(figsize=(11,4))
plt.subplot(1,2,1)
plt.plot(u.value.T)
plt.ylabel("$u_t$", fontsize=14)
plt.xlabel("t")
plt.subplot(1,2,2)
plt.xlabel("t")
plt.plot(np.linalg.norm(u.value,axis=0),"m")
plt.ylabel("$||u_t||_2$", fontsize=14)
plt.tight_layout()
```



The control input is sparse. Also in different times the different components are nonzero.

5. Portfolio optimization

1. First part

Maximize $\quad \mu^T w - \gamma \max w^T \Sigma^{[k]} w$
Subject to $\quad \mathbf{1}^T w = 1$

The above optimization problem is equivalent to the following problem.

Minimize $\quad -\mu^T w + \gamma t$
Subject to $\quad \mathbf{1}^T w = 1$
. $\quad w^T \Sigma^{[k]} w \leq t \quad \text{for } k = 1, \ldots, M$

Suppose λ and ν be a Lagrange multiplier. So we have:

$$L(w, t, \lambda, \nu) = -\mu^T w + \gamma t + \sum_{k=1}^{M} \lambda_k (w^T \Sigma^{[k]} w - t) + \nu \, (\mathbf{1}^T w - 1)$$

So the KKT conditions are:

$-\mu + \sum_{k=1}^{M} 2\lambda_k \Sigma^{[k]} w + \nu = 0$
$\gamma - \sum_{k=1}^{M} 2\lambda_k = 0$
$\mathbf{1}^T w = 1$
$w^T \Sigma^{[k]} w \leq t \quad \text{for } k = 1, \ldots, M$
$\gamma \succeq 0$
$\lambda_k (w^T \Sigma^{[k]} w - t) \quad \text{for } k = 1, \ldots, M$

Also for the problem

Maximize $\quad \mu^T w - \sum_{k=1}^{M} \gamma_k \, w^T \Sigma^{[k]} w$
Subject to $\quad \mathbf{1}^T w = 1$

is equal to

Minimize $\quad -\mu^T w + \sum_{k=1}^{M} \gamma_k \, w^T \Sigma^{[k]} w$
Subject to $\quad \mathbf{1}^T w = 1$

and Lagrange function is

$$L(w,\alpha) = -\mu^T w + \sum_{k=1}^{M} \gamma_k \, w^T \Sigma^{[k]} w + \alpha(\mathbf{1}^T w - 1)$$

So KKT condition are:

$$\mathbf{1}^T w = 1$$
$$-\mu + \alpha\,\mathbf{1} + \sum_{k=1}^{M} 2\gamma_k \Sigma^{[k]} w = 0$$

If $(w^\star, t^\star, \lambda^\star, \nu^\star)$ be optimal for the first optimization, by choosing $\gamma_k = \lambda_k$ we can say $(w, \alpha) = (w^\star, \nu^\star)$ satisfy and $w^\star$ is a optimal point for main optimization problem.

2. Second part

```
# Import required library
import numpy as np
import cvxpy as cvx
from multi_risk_portfolio_data import *

# Initialize variable
w = cvx.Variable(n)
t = cvx.Variable()

# Solve problem
risks = [cvx.quad_form(w, Sigma) for Sigma in (Sigma_1, Sigma_2, Sigma_3, Sigma_4,
                                           Sigma_5, Sigma_6)]
risk_constraints = [risk <= t for risk in risks]
prob = cvx.Problem(cvx.Maximize(w.T @ mu - gamma * t), risk_constraints + [cvx.sum(w) ==
                                           1])
prob.solve()

# Print output
print("weights:")
print("\n".join(["{}".format(weight) for weight in w.value]))
print("worst case : ", t.value)
print("risk values:")
print("\n".join(["{}".format(risk.value) for risk in risks]))
print("gamma_k values:")
print("\n".join(["{}".format(risk.dual_value) for risk in risk_constraints]))
```

```
weights:
0.4247382049873743
0.6642699652212044
-0.11469037220485757
1.3805550930331925
1.4242285197751403
-1.527064897348089
-0.6140154499013144
-0.4987908148392034
-0.25406875723047384
0.11483850850702625
worst case : 0.12188147875724109
risk values:
0.12188147878073646
0.0845435093507211
0.08247154824172492
0.12188147874793029
0.12188147884781253
0.12188147867241997
```
γ_k values:
```
0.29231555
3.35675336e-10
3.53686168e-10
0.46580236
0.14230459
0.0995775
```