

```
In [ ]: import numpy as np
import cp as cp
import matplotlib.pyplot as plt
```

A6.2) Minimax rational fit to the exponential

```
In [4]: k = 201
t = [-3 + 6 * (i-1) / (k-1) for i in range(1, k+1)]
y = [np.exp(t[i]) for i in range(k)]
print(len(t), len(y))
```

```
201 201
```

```
In [ ]: eps = 1e-3

a0 = cp.Variable()
a1 = cp.Variable()
a2 = cp.Variable()
b1 = cp.Variable()
b2 = cp.Variable()

u = 1
l = 0
max_iters = np.ceil(np.log2((u-l) / eps)).astype(int)
print(f"max_iters: {max_iters}")
iter = 0
while u-l > eps and iter <= max_iters:
    iter += 1
    gamma = 0.5 * (u + l)

    constraints = []
    for i in range(k):
        p = a0 + a1*t[i] + a2*t[i]**2
        q = 1 + b1*t[i] + b2*t[i]**2

        constraints += [
            q >= eps,
            p - y[i]*q <= gamma * q,
            p - y[i]*q >= -gamma * q
        ]

    prob = cp.Problem(cp.Minimize(0), constraints)
    prob.solve(qcp=True)
    if prob.status == 'optimal':
        u = gamma
        print(f"iter: {iter}, u: {u}")
    else:
        l = gamma
```

```

max_iters: 10
iter: 1, u: 0.5
iter: 2, u: 0.25
iter: 3, u: 0.125
iter: 4, u: 0.0625
iter: 5, u: 0.03125
iter: 7, u: 0.0234375
C:\Users\kyler\AppData\Local\Temp\ipykernel_27864\842571652.py:30: UserWarning: Solution
may be inaccurate. Try another solver, adjusting the solver settings, or solve
with verbose=True for more information.
    prob.solve(qcp=True)
iter: 11, u: 0.02294921875

```

```

In [60]: gamma = u
print(f"gamma: {gamma}")
a0 = cp.Variable()
a1 = cp.Variable()
a2 = cp.Variable()
b1 = cp.Variable()
b2 = cp.Variable()

constraints = []

for i in range(k):
    p = a0 + a1*t[i] + a2*t[i]**2
    q = 1 + b1*t[i] + b2*t[i]**2

    constraints += [
        q >= eps,
        p <= (y[i] + gamma) * q,
        p >= (y[i] - gamma) * q,
    ]

prob = cp.Problem(cp.Minimize(0), constraints)
prob.solve()
prob.status

```

gamma: 0.02294921875

Out[60]: 'optimal'

```

In [63]: print(f"a0: {a0.value} \na1: {a1.value}, \na2: {a2.value}, \nb1: {b1.value}, \nb2:
a0: 1.0098151381385128
a1: 0.6123330423080701,
a2: 0.11355127297386064,
b1: -0.41441350842003266,
b2: 0.04845136294950195

```

A16.11) Control with Various Objectives

```

In [42]: n = 4
m = 2

A = np.array([
    [0.95, 0.16, 0.12, 0.01],

```

```

[-0.12,  0.98, -0.11, -0.03],
[-0.16,  0.02,  0.98,  0.03],
[-0. ,  0.02, -0.04,  1.03],
])

B = np.array([
[ 0.8 ,  0. ],
[ 0.1 ,  0.2],
[ 0. ,  0.8],
[-0.2 ,  0.1],
])

x_init = np.ones(n)

T = 100

```

```

In [43]: U = cp.Variable((m, T))
X = cp.Variable((n, T+1))
constraints = [X[:, 0] == x_init, X[:, -1] == np.zeros(n)]
for t in range(T):
    constraints += [X[:, t+1] == A @ X[:, t] + B @ U[:, t]]

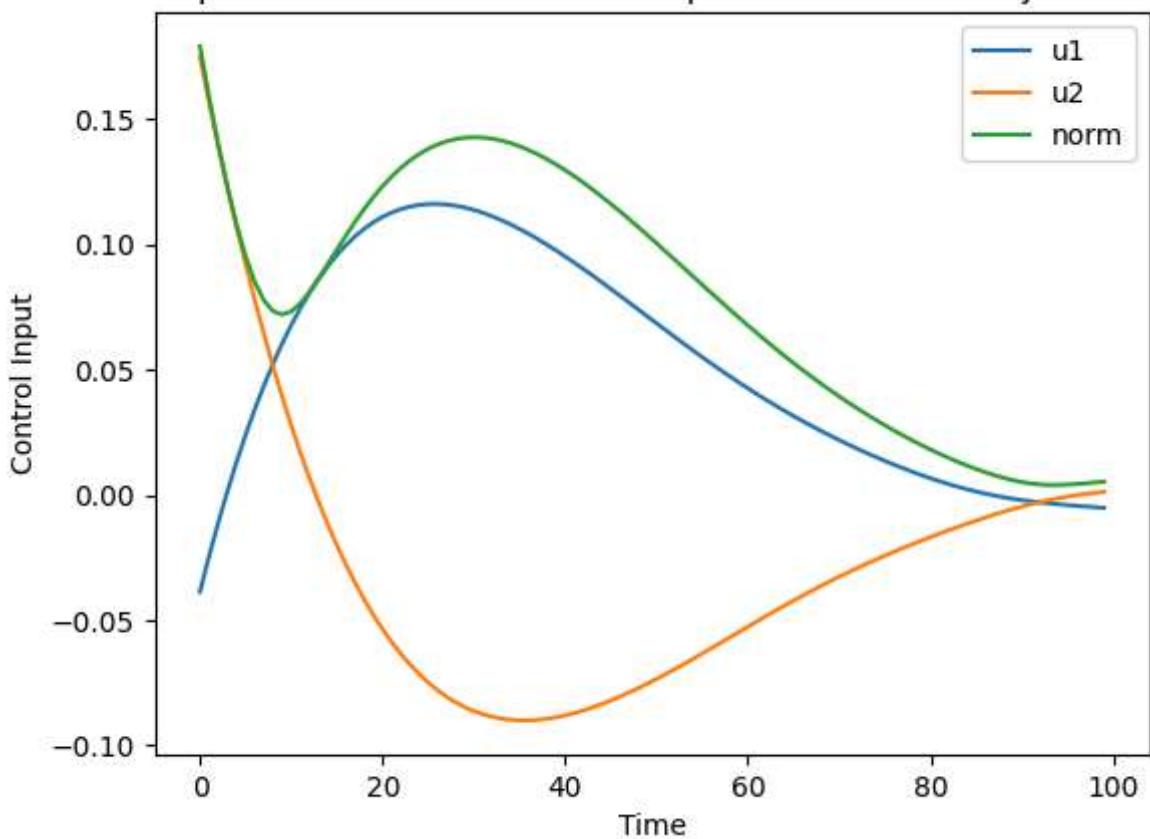
```

```

In [47]: obj16a = cp.Minimize(cp.sum(cp.sum_squares(U, axis=0)))
prob16a = cp.Problem(obj16a, constraints)
prob16a.solve()
plt.plot(U.value[0, :], label='u1')
plt.plot(U.value[1, :], label='u2')
plt.plot(np.linalg.norm(U.value, axis=0), label='norm')
plt.legend()
plt.xlabel('Time')
plt.ylabel('Control Input')
plt.title('Optimal Control with Sum of Squares of 2-Norm Objective')
plt.show()

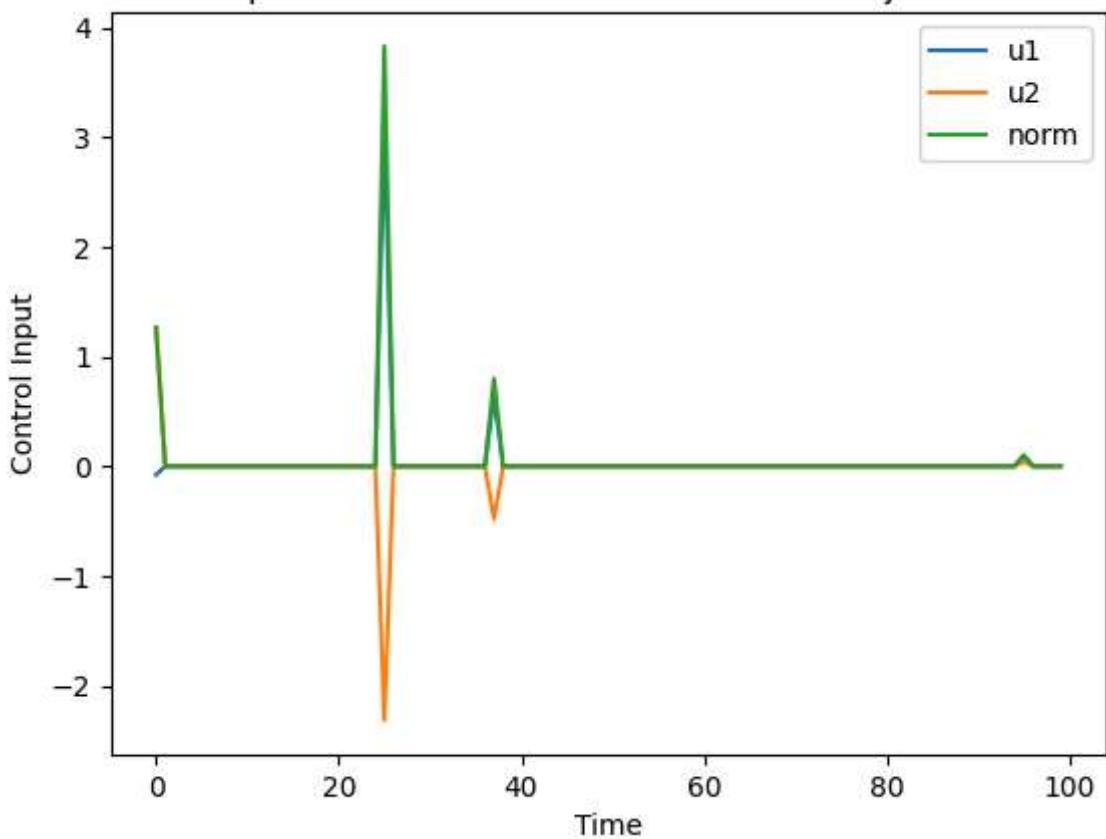
```

Optimal Control with Sum of Squares of 2-Norm Objective



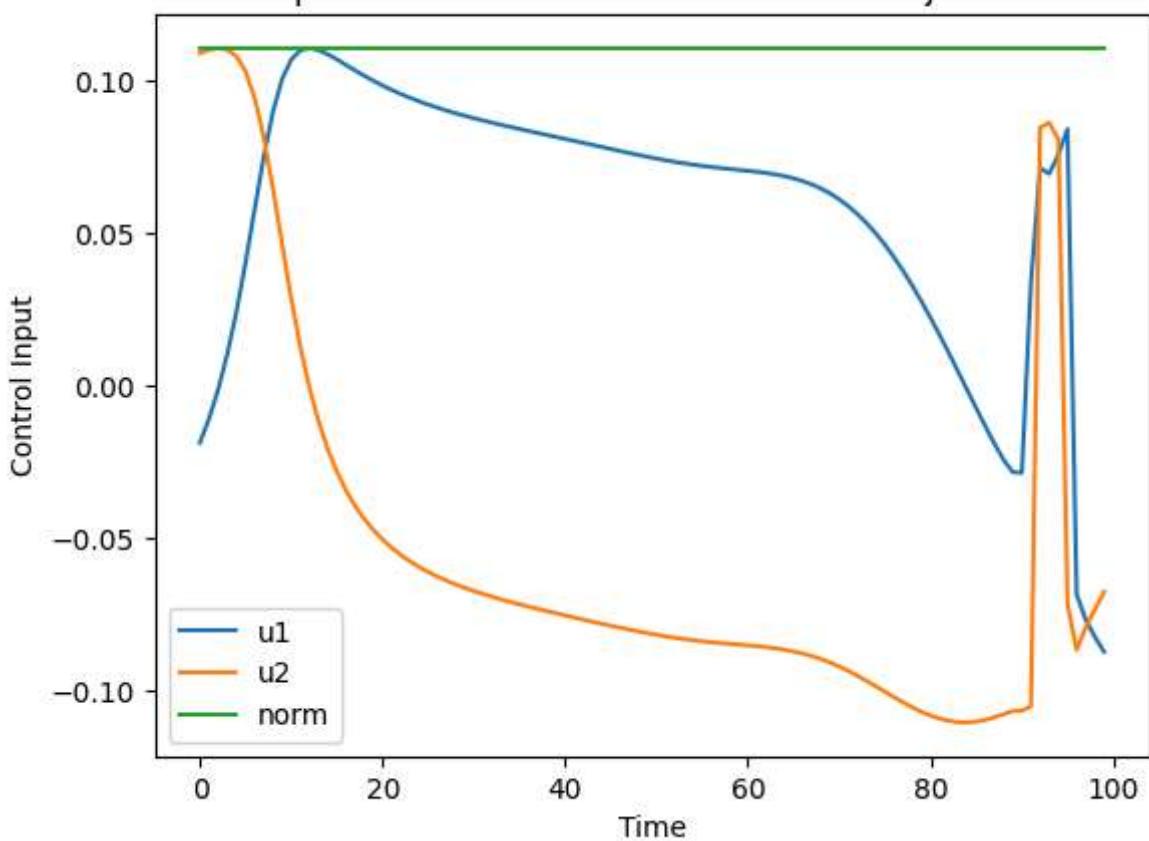
```
In [51]: obj16b = cp.Minimize(cp.sum(cp.norm(U, p=2, axis=0)))
prob16b = cp.Problem(obj16b, constraints)
prob16b.solve()
plt.plot(U.value[0, :], label='u1')
plt.plot(U.value[1, :], label='u2')
plt.plot(np.linalg.norm(U.value, axis=0), label='norm')
plt.legend()
plt.xlabel('Time')
plt.ylabel('Control Input')
plt.title('Optimal Control with Sum of 2-Norm Objective')
plt.show()
```

Optimal Control with Sum of 2-Norm Objective



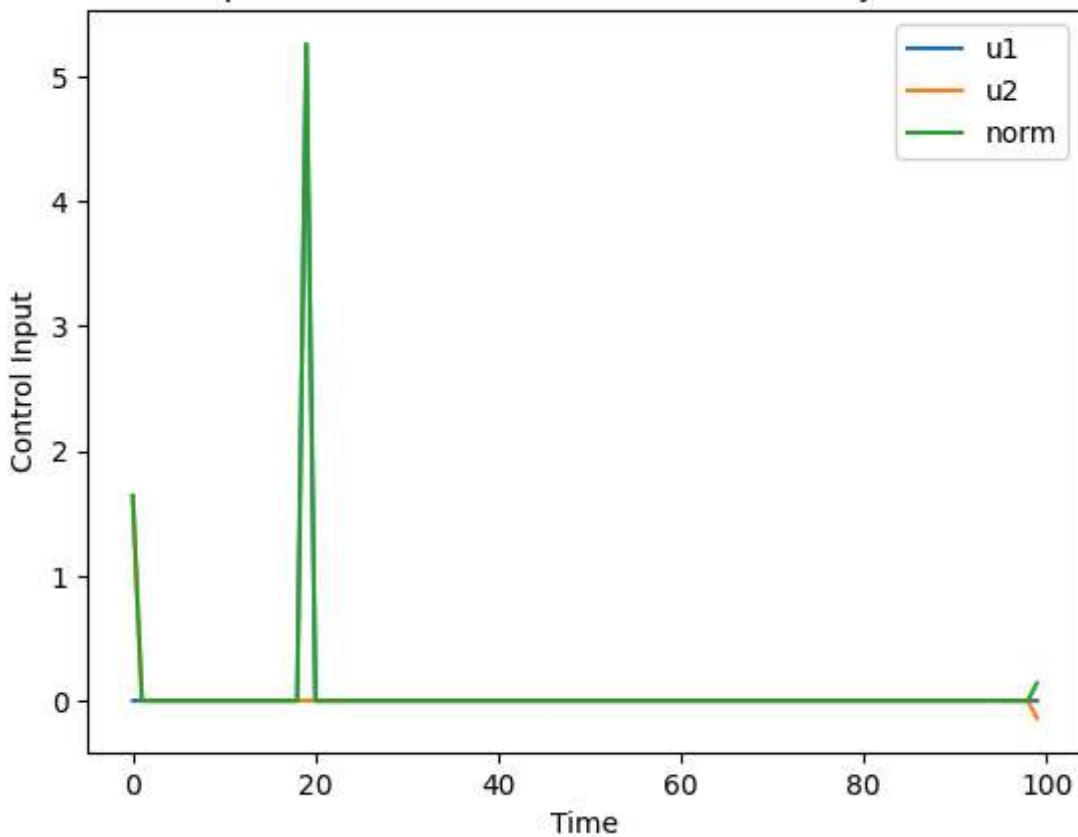
```
In [52]: obj16c = cp.Minimize(cp.max(cp.norm(U, p=2, axis=0)))
prob16c = cp.Problem(obj16c, constraints)
prob16c.solve()
plt.plot(U.value[0, :], label='u1')
plt.plot(U.value[1, :], label='u2')
plt.plot(np.linalg.norm(U.value, axis=0), label='norm')
plt.legend()
plt.xlabel('Time')
plt.ylabel('Control Input')
plt.title('Optimal Control with Max of 2-Norm Objective')
plt.show()
```

Optimal Control with Max of 2-Norm Objective



```
In [54]: obj16d = cp.Minimize(cp.sum(cp.norm(U, p=1, axis=0)))
prob16d = cp.Problem(obj16d, constraints)
prob16d.solve()
plt.plot(U.value[0, :], label='u1')
plt.plot(U.value[1, :], label='u2')
plt.plot(np.linalg.norm(U.value, axis=0), label='norm')
plt.legend()
plt.xlabel('Time')
plt.ylabel('Control Input')
plt.title('Optimal Control with Sum of 1-Norm Objective')
plt.show()
```

Optimal Control with Sum of 1-Norm Objective



- A) We see that there are fairly few large control inputs and inputs are smooth across timesteps, as using the 2-norm and squaring the magnitudes before summing heavily penalizes large control effort.
- B) We see that summing over the control efforts has a sparsifying effect like the L1-norm, with short bursts of control effort over a mostly 0 trajectory.
- C) Taking the max over all control inputs results in a constant amount of control being applied over the entire trajectory, since anything less than or equal to the max is not penalized.
- D) Because we are using the 1-norm, we have sparse control input with many zeros and many relatively large control inputs as well.

A17.18) Option Price Bonds

```
In [33]: m = 200
S = np.linspace(0.5, 2, m)
S0 = 1
r = 1.05

prices = [1, 1, 0.06, 0.03, 0.02, 0.01]

P = np.vstack([
```

```

r * np.ones(m),                      # risk-free
S,                                     # stock
np.maximum(S - 1.1, 0),                # call 1.1
np.maximum(S - 1.2, 0),                # call 1.2
np.maximum(0.8 - S, 0),                # put 0.8
np.maximum(0.7 - S, 0)                 # put 0.7
]).T
collar_P = np.minimum(1.15, np.maximum(S, 0.9))

```

In [37]:

```

q = cp.Variable(m)
constraints = [ P.T @ q == prices, q >= 0 ]
lower = cp.Problem(cp.Minimize(collar_P @ q), constraints)
lower.solve()
print(f"lower bound on arbitrage-free collar price: {lower.value}")

```

lower bound on arbitrage-free collar price: 0.9849999998988971

In [38]:

```

upper = cp.Problem(cp.Maximize(collar_P @ q), constraints)
upper.solve()
print(f"upper bound on arbitrage-free collar price: {upper.value}")

```

upper bound on arbitrage-free collar price: 1.0173313491887297

A20.10) Optimal operation of a microgrid

In [56]:

```

# This script generates data for the micro grid optimization problem
# WARNING: it imports cppy as cp to generate some data, so after running it
# you might want to re-import cppy if you use it under a different name
# same for numpy and matplotlib.pyplot

PLOT_FIGURES = True # True to plot figures, false to suppress plots

N = 96 # Number of periods in the day (so each interval 15 minutes)

# Convenience variables for plotting
fig_size = (14, 3)
xtick_vals = [0, 8, 16, 24, 32, 40, 48, 56, 64, 72, 80, 88, 96]
xtick_labels = (
    "0:00",
    "2:00am",
    "4:00am",
    "6:00am",
    "8:00am",
    "10:00am",
    "12:00pm",
    "2:00pm",
    "4:00pm",
    "6:00pm",
    "8:00pm",
    "10:00pm",
    "12:00am",
)
)

#####
# Price data generation - price values and intervals based off of PG&E Time Of Use
#####

```

```

partial_peak_start = 34 # 08:30
peak_start = 48 # 12:00
peak_end = 72 # 18:00 (6:00pm)
partial_peak_end = 86 # 21:30 (9:30pm)

off_peak_inds = np.concatenate(
    [np.arange(partial_peak_start), np.arange(partial_peak_end, N)])
)
partial_peak_inds = np.concatenate(
    [np.arange(partial_peak_start, peak_start), np.arange(peak_end, partial_peak_end)])
)
peak_inds = np.arange(peak_start, peak_end)

# rates in $ / kWh
off_peak_buy = 0.14
partial_peak_buy = 0.25
peak_buy = 0.45

# Rate cuts from buy prices to get sell prices
off_peak_perc_cut = 0.20
partial_peak_perc_cut = 0.12
peak_perc_cut = 0.11

off_peak_sell = (1 - off_peak_perc_cut) * off_peak_buy
partial_peak_sell = (1 - partial_peak_perc_cut) * partial_peak_buy
peak_sell = (1 - peak_perc_cut) * peak_buy

# Combine the buy and sell prices into the price vectors
R_buy = np.zeros(N)
R_buy[off_peak_inds] = off_peak_buy
R_buy[partial_peak_inds] = partial_peak_buy
R_buy[peak_inds] = peak_buy

R_sell = np.zeros(N)
R_sell[off_peak_inds] = off_peak_sell
R_sell[partial_peak_inds] = partial_peak_sell
R_sell[peak_inds] = peak_sell

# Plot the prices
if PLOT_FIGURES:
    plt.figure(figsize=fig_size)
    plt.plot(R_buy, label="Buy Price")
    plt.plot(R_sell, label="Sell Price")
    plt.legend()
    plt.ylabel("Price ($/kWh)")
    plt.title("Energy Prices ($/kWh)", fontsize=19)
    plt.xticks(xtick_vals, xtick_labels)
    plt.show()

#####
# Solar data generation
#####
# Just something simple: a shifted cosine wave, squared to smooth edges, peak at no
shift = N / 2
p_pv = np.power(np.cos((np.arange(N) - shift) * 2 * np.pi / N), 2)

```

```

scale_factor = 35
p_pv = p_pv * scale_factor
p_pv = np.maximum(p_pv, 0)
p_pv[: int(shift / 2)] = 0
p_pv[-int(shift / 2) :] = 0

# Plot it
if PLOT_FIGURES:
    plt.figure(figsize=fig_size)
    plt.plot(p_pv)
    plt.title("PV Curve (kW)", fontsize=19)
    plt.ylabel("Power (kW)")
    plt.xticks(xtick_vals, xtick_labels)
    plt.show()

#####
# Load Data Generation (using cp)
#####
# Fit a curve to some handpicked points and constrain the end points
# to match and the derivative at the end to be the same at the beginning

# points to fit to
points = [
    [0, 7],
    [10, 8],
    [20, 10],
    [28, 15],
    [36, 21],
    [45, 23],
    [52, 21],
    [56, 18],
    [60, 22.5],
    [66, 24.3],
    [70, 25],
    [73, 24],
    [83, 19],
    [95, 7],
]
points = np.array(points, dtype=int)

# Formulate an optimization problem that minimizes the error of
# the fit while also minimizing the 2nd order difference of the function
p_fit = cp.Variable(N)
obj_val = 0
# Add periodicity constraints
constr = [p_fit[0] == p_fit[-1]] # Constraint the end points to match
constr += [
    (p_fit[1] - p_fit[0]) == (p_fit[-1] - p_fit[-2])
] # constraint deriv at endpoints to match
# Loss for fitting the data points
for pt in points:
    obj_val += cp.square(p_fit[pt[0]] - pt[1])

# Loss for 2nd order smoothness (weight parameter chosen from weight twiddling)

```

```

for i in range(N):
    obj_val += 100 * cp.square(p_fit[(i + 1) % N] - 2 * p_fit[i] + p_fit[(i - 1) %

obj = cp.Minimize(obj_val)
prob = cp.Problem(obj, constr)
prob.solve()
p_fit = p_fit.value
p_ld = p_fit

# Plot the curve
if PLOT_FIGURES:
    plt.figure(figsize=fig_size)
    plt.plot(p_fit)
    # plt.plot(points[:,0], points[:,1], 'o') # For plotting the interpolation p
    plt.title("Load Curve (kW)", fontsize=19)
    plt.ylabel("Power (kW)")
    plt.xticks(xtick_vals, xtick_labels)
    plt.show()

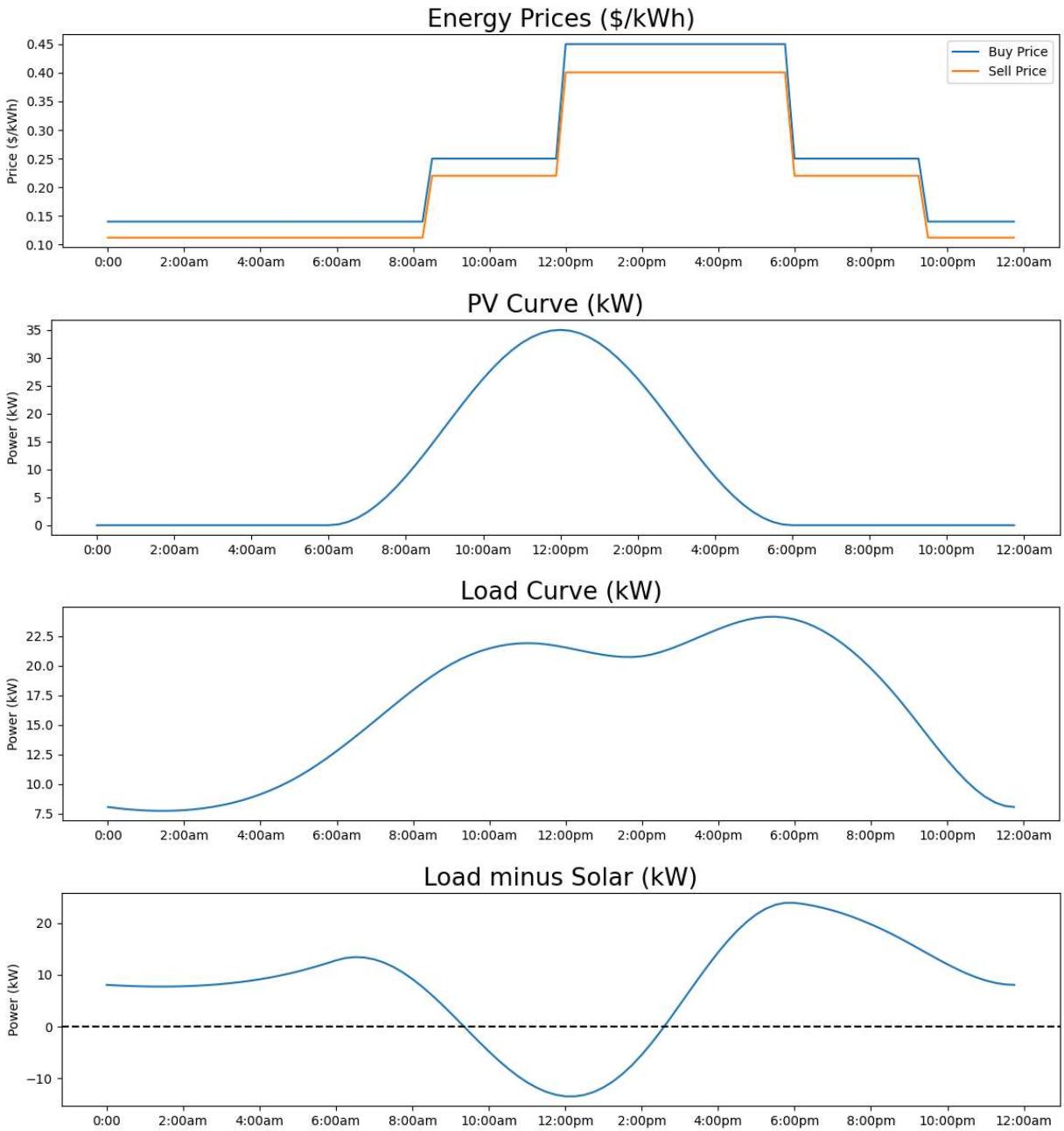
# For reference, plot the net Load curve: Load - solar
# Aside: the shape of this curve is referred to as "the duck curve"
# by some people in energy (since the shape looks like a duck on profile), and
# is a result of renewable generation displacing load demand in the day.
if PLOT_FIGURES:
    plt.figure(figsize=fig_size)
    plt.plot(p_ld - p_pv)
    plt.axhline(0, color="black", linestyle="--")
    plt.title("Load minus Solar (kW)", fontsize=19)
    plt.ylabel("Power (kW)")
    plt.xticks(xtick_vals, xtick_labels)
    plt.show()

#####
# Battery and Grid Line Constraint Values
#####
# Max charge and discharge rates
D = 10 # Max discharge rate (kW)
C = 8 # Max charge rate (kW)
Q = 27 # Max energy (kWh)

"""
Final list of values generated:

N (scalar): number of intervals we split the day into
R_buy (vector, $/kWh): prices one can buy energy at from grid in given interval
R_sell (vector, $/kWh): prices one can sell energy at to grid in given interval
p_pv (vector, kW): power generated by solar
p_ld (vector, kW): power demands of load
D (scalar, kW): max discharge rate of battery
C (scalar, kW): max charge rate of battery
Q (scalar, kWh): max energy of battery
"""

```



```
Out[56]: '\nFinal list of values generated:\n\nN (scalar): number of intervals we split the day into\nR_buy (vector, $/kWh): prices one can buy energy at from grid in given interval\nR_sell (vector, $/kWh): prices one can sell energy at to grid in given interval\np_pv (vector, kW): power generated by solar\np_ld (vector, kW): power demands of load\nD (scalar, kW): max discharge rate of battery\nC (scalar, kW): max charge rate of battery\nQ (scalar, kWh): max energy of battery\n'
```

In order to have a convex objective, we can rearrange the objective to be:

$$1/4(R^{buy})^T p^{grid} + 1/4(R_{sell} - R^{buy})^T (p^{grid} - (p^{grid})_+)$$

```
In [72]: p_grid = cp.Variable(N)
p_batt = cp.Variable(N)
q = cp.Variable(N) # battery state
obj20 = cp.Minimize(0.25 * R_buy.T @ p_grid + 0.25 * (R_sell.T - R_buy.T) @ (p_grid
constr20 = [p_ld == p_pv + p_grid + p_batt,
            q >= 0,
            q <= Q,
```

```

        q[0] == q[-1] - 0.25 * p_batt[-1],
        p_batt >= -C,
        p_batt <= D
    ]
    for i in range(N-1):
        constr20 += [q[i+1] == q[i] - 0.25 * p_batt[i]]
prob20 = cp.Problem(obj20, constr20)
prob20.solve()

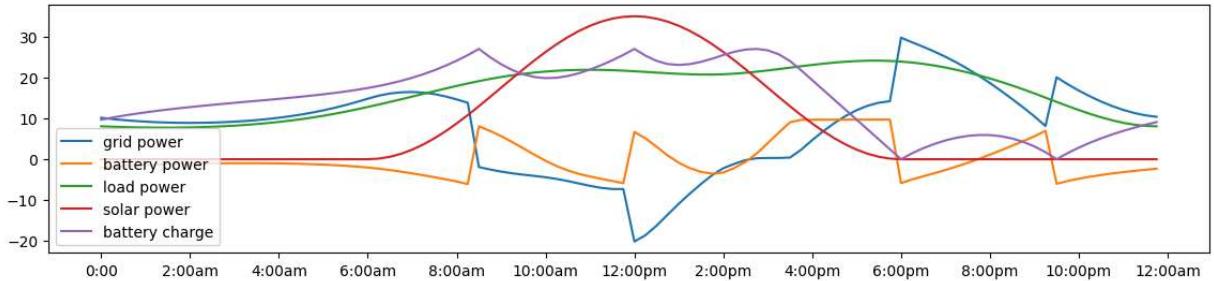
```

Out[72]: np.float64(32.76732513646789)

```

In [73]: plt.figure(figsize=fig_size)
plt.plot(p_grid.value, label='grid power')
plt.plot(p_batt.value, label='battery power')
plt.plot(p_ld, label='load power')
plt.plot(p_pv, label='solar power')
plt.plot(q.value, label='battery charge')
plt.xticks(xtick_vals, xtick_labels)
plt.legend()
plt.show()

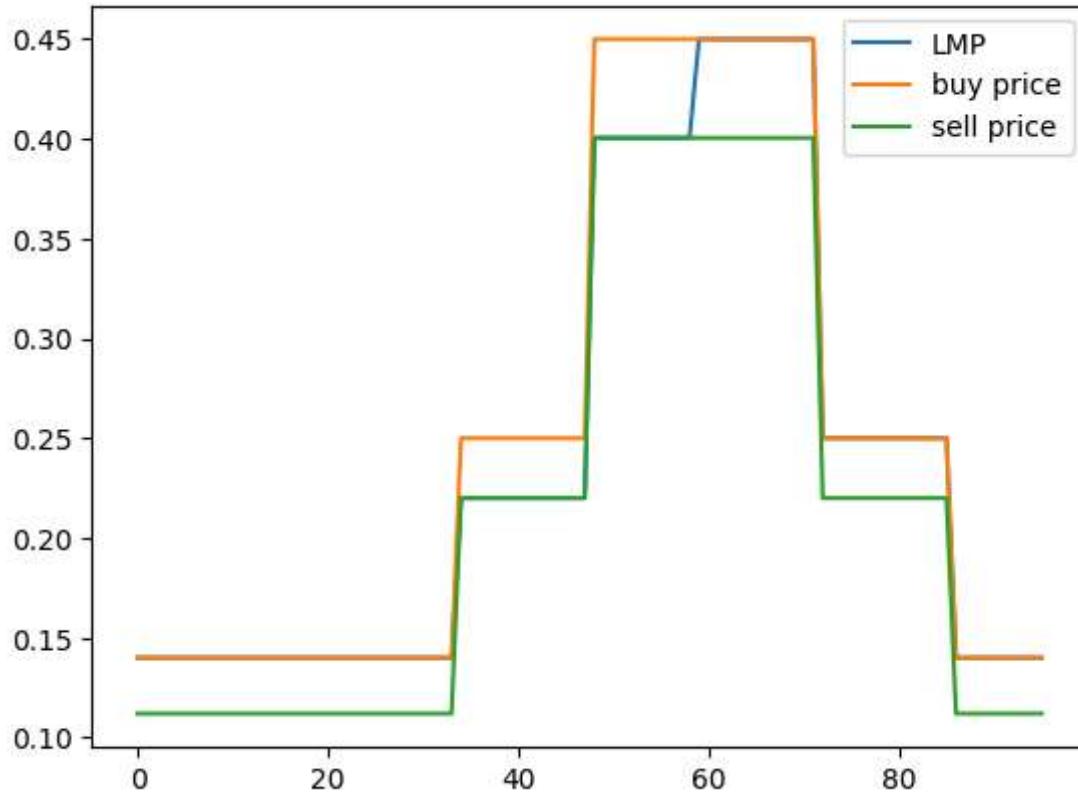
```



```

In [ ]: lmp = -4 * prob20.constraints[0].dual_value
plt.plot(lmp, label='LMP')
plt.plot(R_buy, label='buy price')
plt.plot(R_sell, label='sell price')
plt.plot
plt.legend()
plt.show()

```



We see that the dual variable approximates the buy and sell prices, switching between the min and max of the prices.

```
In [77]: print(f"Load cost: {lmp @ p_ld:.2f}")
print(f"Battery cost: {lmp @ p_batt.value:.2f}")
print(f"Grid cost: {lmp @ p_grid.value:.2f}")
print(f"PV value: {lmp @ p_pv:.2f}")
```

```
Load cost: 430.35
Battery cost: 33.48
Grid cost: 131.07
PV value: 265.80
```

We see that this satisfies the power balance constraint.