EE364a: Convex Optimization I G. Ogut
August 15–18 2024

Final Exam

This is a 24 hour take-home final, timed by Gradescope. The Gradescope timer begins as soon as you request the exam.

You may use any books, notes, or computer programs, but you may not discuss the exam with anyone until August 18, after everyone has taken the exam. The only exception is that you can ask us for clarification, via the course staff email address. Please do not ask questions on Ed. We've tried pretty hard to make the exam unambiguous and clear, so we're unlikely to say much.

We will deduct points from long, needlessly complex solutions, even if they are correct. Our solutions are not long, so if you find that your solution to a problem goes on and on for many pages, you should try to figure out a simpler one. We expect neat, legible exams from everyone, including those enrolled CR/NC.

When a problem involves computation you must give all of the following: a clear discussion and justification of exactly what you did, the source code that produces the result, and the final numerical results or plots.

Files containing problem data can be found at

`https://github.com/cvxgrp/cvxbook_additional_exercises/tree/main/python`

Please respect the honor code. Although we allow you to work on homework assignments in small groups, you cannot discuss the final with anyone, at least until everyone has taken it.

All problems have equal weight. Some are (quite) straightforward. Others, not so much.

Be sure you are using the most recent version of CVXPY, which is 1.4. Check the messages on Ed often during the exam, just in case we need to send out an important announcement.

Some problems involve applications. But you do not need to know *anything* about the problem area to solve the problem; the problem statement contains everything you need.

The problems do *not* appear in order of increasing difficulty.

1. *A statistical model of seasonal shading.* Let $p_t \in \mathbf{R}_{++}$ denote the average power obtained from a photovoltaic (PV) system on day t, $t = 1, \ldots, T$. This power depends on the day of the year, which affects the sun's trajectory over the system, as well as whether or not there is shading from clouds on that day. We let $p_t^{\text{cs}} \in \mathbf{R}_{++}$ denote the average power that would be obtained from the system *if* there were no shading from clouds. (This can be estimated separately, and is called the *clear sky power*.) We define the shade fraction as $s_t = p_t/p_t^{\text{cs}}$, which is the fraction of the clear sky power that was obtained on day t. These numbers lie between zero (full shade) and one (clear sky). These shade fraction numbers are our given data.

   We model s_t as independent random variables with CDF $F(x) = x^{\gamma_t}$ for $x \in [0, 1]$, where $\gamma_t > 0$. (This is a special case of a beta distribution.) If $\gamma_t = 1$, then the shading on day t is uniform on $[0, 1]$; if $\gamma_t = 3$, then the shading on day t is more concentrated on high values than low ones.

   We will require that γ_t be 365-periodic, *i.e.*, it repeats every year, which means $\gamma_{t+365} = \gamma_t$ for $t = 1, \ldots, T - 365$. This means our model is specified by the 365 numbers $\gamma_1, \ldots, \gamma_{365}$. In addition we assume that γ_t varies smoothly over the year, which means that the Dirichlet energy (sum of differences squared),

$$D(\gamma) = (\gamma_2 - \gamma_1)^2 + (\gamma_3 - \gamma_2)^2 + \cdots + (\gamma_{365} - \gamma_{364})^2 + (\gamma_1 - \gamma_{365})^2,$$

   is small.

   We obtain our estimate of λ_t by minimizing $\ell(\gamma) + \omega D(\gamma)$, where ℓ is the negative log-likelihood, and ω is a positive hyperparameter. Note that here we use the total negative log-likelihood across the T data points, and not the average negative log-likelihood, which would divide this by t.

   (a) Explain how to formulate this as a convex problem. Justify any change of variables.

   (b) Carry out the method of part (a) on the data found in `seasonal_shading_data.py`, using hyperparameter value $\omega = 300$. This file contains the shade fraction for a residental PV system in California from July 1, 2015 to June 30, 2018 (with the leap year day February 29th 2016 removed so each year has 365 days). Plot γ_t versus t along with s_t. Plot the CDF of the estimated distribution for $t = 138$ and $t = 290$ (corresponding to November 15 and April 15). Briefly discuss the results.

1. $\ell(\gamma) = -\log p_\gamma(S_t)$

where $p_\gamma(x)$ is the likelihood function of S_t

$$p_{\gamma_t}(x) = \frac{d}{dx} F(x) = \gamma_t x^{\gamma_t}$$

$$p_\gamma(x) = \begin{cases} \gamma_t x^{\gamma_t} & \text{for } 0 \le x \le 1 \\ 0 & \text{otherwise} \end{cases}$$

thus $\ell(\gamma_t) = -\log p_{\gamma_t}(S_t)$

$$= -\log \gamma_t S_t^{\gamma_t}$$

$$= -(\log \gamma_t + \gamma_t \log S_t)$$

$\log \gamma_t$: concave in γ_t; $\gamma_t \log S_t$: affine; sum: concave; negative: convex

then, the convex optimization is

$$\text{minimize} \quad -\sum_{t=1}^{T}(\log \gamma_t + \gamma_t \log S_t) + 10\left(\sum_{t=1}^{364}(\gamma_{t+1} - \gamma_t)^2 + (\gamma_1 - \gamma_{365})^2\right)$$

$$\text{subject to} \quad \gamma_t = \gamma_{t+365} \quad \text{for } t = 1, \ldots, T - 365$$

2. *Smoothest ride through a set of green lights.* We will design the smoothest trajectory of a car moving along a road, making a sequence of green lights, with given initial and terminal conditions. Let $p_t \in \mathbf{R}$ denote the position along the road at time period $t = 0, 1, \ldots, T$. We are given the initial condition $p_0 = 0$ and the terminal condition $p_T = L$, where L is the total length of the route along the road.

   We define the speed as $s_t = p_{t+1} - p_t$, $t = 0, \ldots, T-1$, the acceleration as $a_t = s_{t+1} - s_t$, $t = 0, \ldots, T-2$, and the jerk (third derivative in continuous time) as $j_t = a_{t+1} - a_t$, $t = 0, \ldots, T-3$. Our objective is to minimize the mean square jerk of the trajectory, given by

$$J = \frac{1}{T-2} \sum_{t=0}^{T-3} j_t^2.$$

   We are given a minimum and maximum speed, *i.e.*, $S^{\min} \leq s_t \leq S^{\max}$, $t = 1, \ldots, T-1$. We have $S^{\min} > 0$, which means the car always moves forward, and never backs up. We assume that the car starts at rest, *i.e.*, $s_0 = 0$.

   The twist is that the car must pass through a given set of K green lights. Each one is specified by $l_k \in (0, L)$, the distance along the road to the stoplight, the time it turns green $g_k \in \{0, \ldots, T\}$, and the time it turns red $r_k \in \{0, \ldots, T\}$ (both g_k and r_k are integers). We have $r_k > g_k$, with $r_k - g_k$ the total time the kth stoplight is green.

   Now we explain what it means to make a green light. Note that p_t is an increasing sequence which starts at 0 and ends at L. Let us extend the sequence p to a piecewise linear function $\tilde{p}$ of the continuous variable $\tau \in [0, T]$. Thus $\tilde{p}_\tau$ is an increasing continuous function with $\tilde{p}_0 = 0$ and $\tilde{p}_T = L$. Let T_k be the unique $\tau \in (0, T)$ for which $\tilde{p}_\tau = l_k$, *i.e.*, the (continuous time) moment when the car arrives at the kth light. To say that we make the lights means

$$g_k \leq T_k \leq r_k, \quad k = 1, \ldots, K,$$

   i.e., the car arrives at the light no earlier than the light turns green, and no later than when it turns red.

   This is illustrated in the plot below. In this problem instance, we are given $T = 300$ s, $K = 5$, $L = 3000$ m, $S^{\min} = 4$ m/s, $S^{\max} = 16$ m/s, $l = (300, 825, 1620, 1900, 2800)$, $g = (10, 50, 100, 200, 240)$, and $r = (40, 80, 130, 230, 270)$. The plot shows the position of the car versus time.

   Remark. Note that for this example, the car doesn't minimize the jerk, but rather takes a suboptimal path to make the lights. It also violates the maximum speed constraint. Such a path is not smooth, and would result in an uncomfortable ride for the passengers.

minimize $\frac{1}{T-2}\sum_{t=0}^{T-3} j_t^2$

$j_t = a_{t+1} - a_t \qquad t = 0, \dots, T-3$

$a_t = s_{t+1} - s_t \qquad t = 0, \dots, T-2$

$s_t = p_{t+1} - p_t \qquad t = 0, \dots, T-1$

$s^{min} \le s_t \qquad p_0 = 0$

$s^{max} \ge s_t \qquad p_T = L$

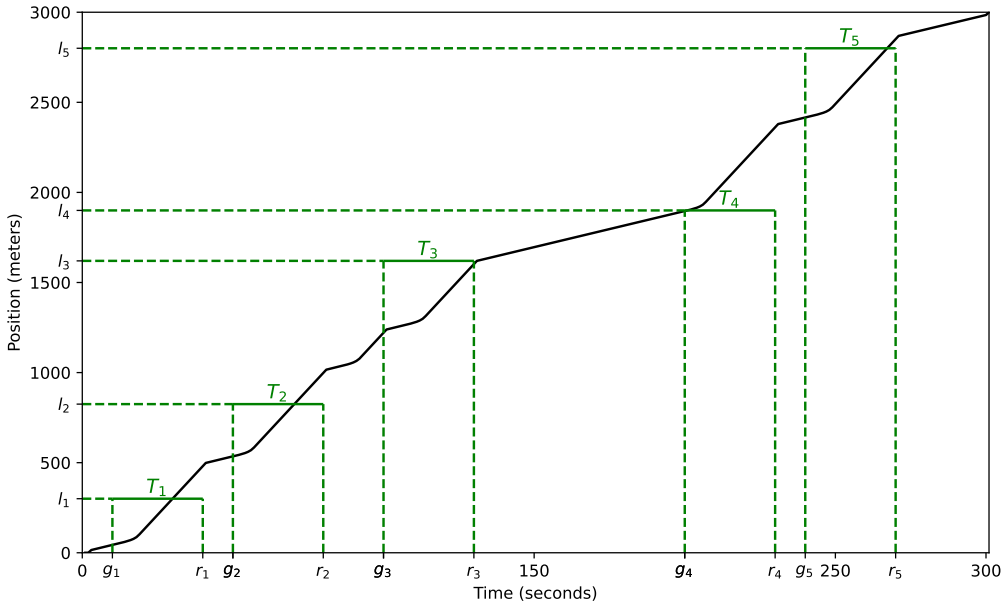
$$\begin{cases} g_k \le T_k \le r_k \\ \tilde{p}(T_k) = l_k \quad \text{for all } k \in (0,1,2,3) \end{cases}$$

Convert to convex constraints

$p_t \ge l_k \quad \text{for all } t \ge r_k$

redundant: $p_t \ge l_k \quad \text{for all } t \ge g_k$

$p_t \le l_k \quad \text{for all } t \le g_k$



(a) Explain how to transform to a convex or quasiconvex problem. Justify any change of variables or rewriting of the constraints. Be sure to identify the variables in your formulation.

(b) Carry out the solution with the given problem instance. Report the mean square jerk of the optimal trajectory. Round reported values to four decimal places. Code `smooth_ride_plot.py` generates a plot for you with the position versus time if you provide $p^\star$. Briefly compare the plot with the nonoptimal one above.

3. *Constant current constant voltage charging of a Li-Ion battery.* We consider the charging of a Li-Ion battery over the time $t = 1, \ldots, T+1$ (in minutes). The charge in the battery at the beginning of time period t is denoted q_t, $t = 1, \ldots, T+1$. We have initial and terminal conditions $q_1 = Q^{\min}$ and $q_{T+1} = Q^{\max}$, where $Q^{\min} = 960$ and $Q^{\max} = 6300$ are the charge levels corresponding to fully discharged and fully charged, given in C (Coulombs). We let i_t, $t = 1, \ldots, T$, denote the charging current at time t, given in A (Amperes). We require $0 \leq i_t \leq I^{\max}$, where $I^{\max} = 1.5$ is the maximum allowed charging current. The charge and current are related as

$$q_{k+1} = q_k + hi_k, \quad k = 1, \ldots, T,$$

where $h = 60$ seconds converts current in A over one minute to C.

The battery has an open-circuit voltage that depends on the charge in it:

$$v_t^{\text{oc}} = a + \frac{b}{Q^{\text{crit}} - q_t}, \quad t = 1, \ldots, T+1,$$ convex, not affine

where $a = 3.40$, $b = 500$, and $Q_{\text{crit}} = 6925$ are parameters. The open circuit voltage has physical units of V (Volts), a is in V, b is in J (Joules), and Q^{crit} is in C.

The voltage of the battery (not to be confused with its open-circuit voltage) at time t is denoted v_t, $t = 1, \ldots, T$. It is given by $v_t = v_t^{\text{oc}} + Ri_t$, where $R = 0.4\ \Omega$ (Ohm) is the internal resistance. (In EE dialect, this model of a battery is a nonlinear capacitor in series with a resistance.) The battery voltage cannot exceed a given maximum value, *i.e.*, $v_t \leq V^{\max}$, $t = 1, \ldots, T$, with $V^{\max} = 4.22$ V.

The energy loss in charging the battery is given by

$$L = hR \sum_{t=1}^{T} i_t^2,$$

where $h = 60$ seconds converts power in W (Watts) for one minute into J. The total energy transferred into the battery in this exercise is $E = 20975$ J. The charging efficiency is then $(E - L)/E$.

(a) Formulate the problem of charging the battery with maximum efficiency as a convex optimization problem. The variables are $q_1, \ldots, q_{T+1}$ and $i_1, \ldots, i_T$.

(b) Carry out the method of part (a) for three different charging times: $T = 120$ (fast), $T = 180$ (normal), and $T = 240$ (slow). Give the charging efficiency for each of these charging times. Values should be rounded to two decimal places.

For each charging time, plot the current, charge, and voltage versus time t. (All together 9 plots.) Please use plotting code in `cccv_charging_plot.py`. You need to provide $i^\star, v^\star, q^\star$ for each charging time.

(a) maximize $\frac{E-L}{E}$ is the same as

minimize $L = hR\sum_{t=1}^{T} i_t^2$

$$q_1 = Q^{min}$$
$$q_{T+1} = Q^{max}$$

$$c = \frac{b}{Q^{crit} - q_t}$$

$$q_{t+1} = q_t + h i_t \qquad t = 1, \dots, T$$

□ $$v_t^{oc} = a + \frac{b}{Q^{crit} - q_t} \qquad t = 1, \dots, T$$

$$v_t = v_t^{oc} + R i_t \qquad t = 1, \dots, T$$

$$v_t \le V_{max} \qquad t = 1, \dots, T$$

$$i_t \le I_{max} \qquad t = 1, \dots, T$$

$$i_t \ge 0 \qquad t = 1, \dots, T$$

This is not a convex problem because constraint □ is not an affine equality

We consider relax □ to be an inequality constraint

$$v_t^{oc} \ge a + \frac{b}{Q^{crit} - q_t} \qquad t = 1, \dots, T$$

(c) The industry standard method for charging a Li-Ion battery is called *constant current constant voltage* (CCCV) charging. It consists of charging the battery with a constant current until some specified level of charge is achieved, and then switching to constant voltage charging until the battery is fully charged.

Give some brief remarks comparing the optimal charging you found above and the industry standard method.

Remarks.

- Don't worry about the physical units; we give them because the problem instance is realistic.
- The parameters given above are reasonable values for a so-called 18650 Li-Ion battery, which is a little bit larger than what you might have in your mobile phone.
- More sophisticated charging methods (*e.g.*, for electric vehicles) take into account battery temperature, and the problem of charging many diffferent battery cells.
- Please neglect potential solver warnings about numerical accuracy.

worst case

x

$a^T x + u \geq 0$

$\min(a^T x) = \min(a^T(-a)) = -a^T$

4. *Conditional Gaussian regression.* We wish to fit a model of the form $y \sim \mathcal{N}(\mu(x), \sigma^2(x))$, where $y \in \mathbf{R}$ is an outcome, given a feature vector $x \in \mathbf{R}^d$. We will assume that the features all lie in the interval $[-1, 1]$, *i.e.*, $\|x\|_\infty \leq 1$. We will use parametrization $\omega(x) = 1/\sigma(x)$, $\kappa(x) = \mu(x)/\sigma(x)$. We focus on linear regression, which means that $\omega(x) = \alpha^T x + u$ and $\kappa(x) = \beta^T x + v$, where $\alpha \in \mathbf{R}^d$, $\beta \in \mathbf{R}^d$ are vectors of parameters, and $u \in \mathbf{R}, v \in \mathbf{R}$ are offsets that together define our predictor. We require that $\omega \geq 0$ for all x with $\|x\|_\infty \leq 1$ (and not just the training data). We are given training data $x_1, \ldots, x_N \in \mathbf{R}^d$ and $y_1, \ldots, y_N \in \mathbf{R}$.

   (a) Formulate the problem of finding the maximum likelihood estimate of α, β, u, v given the training data as a convex optimization problem.

   (b) Carry out the method of part (a) on the data found in `cond_gauss_reg_data.py`. What are optimal $\alpha^\star, \beta^\star, u^\star, v^\star$? Report the mean and variance of the conditional distribution of the first outcome y_1 given x_1. Round reported values to two decimal places.

$\mu = \frac{\kappa}{\omega}$ $\quad \sigma^2 = \frac{1}{\omega}$

$z = y - \frac{\kappa}{\omega} = y - \frac{\beta^T x + v}{a^T x + u}$: not convex on a, β, v, u

maximize $p(z) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{z^2}{2\sigma^2}}$

$= \frac{1}{\sqrt{2\pi}} \frac{1}{\sigma} e^{-\frac{z^2}{2\sigma^2}}$

$= \frac{1}{\sqrt{2\pi}} w e^{-\frac{z^2 w^2}{2}}$

$= \frac{1}{\sqrt{2\pi}} w e^{-\frac{(zw)^2}{2}}$

$zw = yw - \kappa = y(a^T x + u) - \beta^T x - v$

$$\log p(x) = \log \frac{1}{\sqrt{2\pi}} + \log w + \left(-\frac{1}{2}\right)(x-w)^2$$

thus we can write the problem as

$$\text{maximize} \sum \log(a^T x + u) - \frac{1}{2}\left[y(a^T x + w) - \beta x - v\right]^2$$

$$a^T x + u \geq 0 \quad \text{for all } x \text{ s.t. } \|x\|_\infty \leq 1$$

because we require $a^T x + u \geq 0$ for all $\|x\|_\infty \leq 1$

then $a^T x \geq -u$

that is, $\{a^T x \mid \|x\|_\infty \leq 1\} \geq -u$

$$\Rightarrow \underbrace{\|a\|_2^2}_{\text{convex}} \geq u\|a\|_\infty \qquad = a^T \frac{-a}{\|a\|_\infty}$$

$$= -\frac{\|a\|_2^2}{\|a\|_\infty} \geq -u$$

$$a^T x + u \geq 0$$

$$a$$

maximize $\sum \log(w_i) - \frac{1}{2}\left[y_i w_i - \kappa_i\right]$

$$\kappa_i = \beta^T x_i + v$$

$$w_i = a^T x_i + u$$

$w \geq 0$ for all $\|x\|_\infty \leq 1$

we know $\inf\{a^T x + u \mid \|x\|_\infty \leq 1\} = -a^T a \cdot \frac{1}{c} + u$

where $x_{min} = -\frac{a}{c}$, $c = \|a\|_\infty$

we can then express $w \geq 0$ through

$$\|a\|_\infty \leq 1$$
$$-a^T a + u \geq 0$$

5. *Maximum entropy correction and completion of a correlation matrix.* We consider a correlation matrix $C \in \mathbf{S}^n_{++}$, which means $C_{ii} = 1$, $i = 1, \ldots, n$. We are given only some of the off-diagonal entries of C, and some of these data can be wrong. We denote these given values as C^{g}_{ij}, $(i,j) \in \mathcal{G}$, where $\mathcal{G} \subseteq \{1, \ldots, n\}^2$ is the set of indices of given off-diagonal values. Since C is symmetric, we have $(i,j) \in \mathcal{G}$ whenever $(j,i) \in \mathcal{G}$, and since only off-diagonal entries are given, we have $(i,i) \notin \mathcal{G}$.

   We will estimate C by modifying some of the entries of C^{g}, which incurs a cost

$$q(C) = \sum_{(i,j)\in\mathcal{G}} |C_{ij} - C^{\mathrm{g}}_{ij}|.$$

   In addition we wish to maximize $\log\det C$ (which is the entropy of a Gaussian distribution with covariance matrix C, up to a constant), which imposes the implicit constraint that $C \succ 0$. We minimize the weighted combination $-\log\det C + \lambda q(C)$, where λ is a positive hyperparameter.

   Problem
   Strictly inequality

   (a) Explain how to estimate C using convex optimization. Justify any change of variables.

   (b) Carry out the method of part (a) on the data

$$C^{\mathrm{g}} = \begin{bmatrix} 1 & -0.5 & ? & ? & 0.7 \\ -0.5 & 1 & ? & -0.6 & 0.8 \\ ? & ? & 1 & 0.3 & ? \\ ? & -0.6 & 0.3 & 1 & ? \\ 0.7 & 0.8 & ? & ? & 1 \end{bmatrix},$$

   where ? means $(i,j) \notin \mathcal{G}$. Use $\lambda = 10$. Report each entry of $C^\star$. Round reported values to two decimal places. How many of the given off-diagonal entries of C^{g} were modified?

minimize $\underbrace{-\log\det C}_{\text{Convex}} + \underbrace{\lambda \sum |C_{ij} - C^g_{ij}|}_{\text{Convex}}$

Convex

subject to $C \succ 0$ can be converted to $C \succeq I$

because

$z^T C z > 0$

$k z^T C z \geq z^T I z$

$k z^T C z - z^T I z \geq 0 \Rightarrow z^T (kC - I) z \geq 0$

Our objective is

$$\text{minimize } -\log\det C + \lambda \sum |C_{ij} - C^g_{ij}|$$

and it is homogenous, that is, let $C = \frac{\tilde{C}}{k}$, $C^g = \frac{\tilde{C}^g}{k}$

$$-\log\det C + \lambda \sum |C_{ij} - C^g_{ij}|$$

$$= -\log\det k\tilde{C} + \lambda \sum k |C_{ij} - C_{ij}|$$

$$= -\log k^n - \log\det \tilde{C} +$$

$$\text{minimize} \quad \underbrace{-\log\det C}_{\text{concave}} + \underbrace{\lambda \sum |C_{ij} - C^g_{ij}|}_{\text{convex}}$$

$$\underbrace{\qquad\qquad}_{\text{convex}}$$

subject to $C \succ 0 \Rightarrow C \succeq 0$ ← we relax and will find in (b) that this condition is not active

$$C_{ii} = 1 \quad \text{for } i \in \{1, \ldots, n\}$$

6. *Minimizing the number of actuators used in a control problem.* We consider a standard control problem with state $x_t \in \mathbf{R}^n$, $t = 0, \ldots, T$, input $u_t \in \mathbf{R}^m$, $t = 0, \ldots, T-1$, and dynamics $x_{t+1} = Ax_t + Bu_t$ for $t = 0, \ldots, T-1$. The dynamics matrix $A \in \mathbf{R}^{n \times n}$ and the input matrix $B \in \mathbf{R}^{n \times m}$ are given. We are given the initial state x_0 and the terminal state x_T, and seek $u_0, \ldots, u_{T-1}$ with $\|u_t\|_\infty \leq 1$. We call such an input sequence feasible.

   We say that actuator i is not used if $(u_t)_i = 0$, $t = 0, \ldots, T-1$. The problem is to determine a feasible input trajectory that minimizes the number of actuators used.

   (a) Explain how to solve this problem exactly using convex optimization. Your method can involve solving a modest number of convex optimization problems. You can assume that m is small enough that 2^m is considered modest.

   (b) Carry out the procedure in part (a) with the following data: $T = 20$, $n = 3$, $m = 4$,

$$x_0 = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, \qquad x_T = \begin{bmatrix} 1 \\ 2 \\ -1 \end{bmatrix},$$

   and

$$A = \begin{bmatrix} 1 & 0.1 & 0.0 \\ 0.0 & 0.9 & -0.1 \\ -0.1 & 0 & 0.9 \end{bmatrix}, \qquad B = \begin{bmatrix} 0.9 & -0.4 & 0.0 & 0.6 \\ 2.0 & 0.7 & 0.2 & -0.4 \\ 0.0 & 0.2 & -0.3 & 1.7 \end{bmatrix}.$$

   Report the minimum number of actuators used, and a set of indices of actuators that achieves this minimum number.

(a) We create 2^m B_i matrices

where for B_i where $i = 1c_1 + 2c_2 + 4c_3 + 8c_4$

we copy B to B_i but make column j all zero $c_j \in \{1, 0\}$

then we solve

minimize 1

$$x_{t+1} = Ax_t + B_i u_t$$
$$x_0 = x_0$$
$$x_T = x_T$$
$$\|u_t\|_\infty \leq 1$$

the feasible B_i with the most column set to zero tell us which actuators we can get rid of

7. *Optimal diagonal preconditioner for a PD matrix.* Let $A \in \mathbf{S}^n_{++}$. We seek a diagonal matrix D with positive diagonal entries that minimizes the condition number of DAD,

$$\mathbf{cond}(DAD) = \frac{\lambda_{\max}(DAD)}{\lambda_{\min}(DAD)}.$$

We will simply assume that an optimal D exists. Note also that if D is optimal, so is αD for any $\alpha > 0$. This optimization problem arises in numerical methods where D represents a diagonal preconditioner, but you don't need to know this to solve this problem.

(a) Explain how to use convex optimization to find an optimal diagonal preconditioner for A. You will receive half credit for a valid quasiconvex formulation.

(b) Carry out the procedure from part (a) for the problem instance with

$$A = \begin{bmatrix} 0.2 & -0.2 & 0.6 & -0.6 \\ -0.2 & 0.4 & -1.4 & 1.3 \\ 0.6 & -1.4 & 5.2 & -4.7 \\ -0.6 & 1.3 & -4.7 & 4.4 \end{bmatrix}.$$

Give the diagonal entries of $D^\star$. Report the optimal condition number $\mathbf{cond}(D^\star A D^\star)$ and the condition number of the original matrix, $\mathbf{cond}(A)$.

Minimize $\dfrac{\lambda_{max}(DAD)}{\lambda_{min}(DAD)}$ $\quad A = U^T S U$

subject to $D_{ij} = 0$ for $i \neq j$

$D \succ 0$

$\dfrac{\lambda_{max}(DAD)}{\lambda_{min}(DAD)} \leq \beta$

$\lambda_{max}(DAD) - \beta \lambda_{min}(DAD) \leq 0$ the β-sublevel set of $cond(DAD)$ is quasiconvex

we can then convert this into a feasibility problem

$$\begin{aligned} &\text{minimize} && 0 \\ &\text{subject to} && \lambda_{max}(DAD) - \beta\lambda_{min}(DAD) \leq 0 \\ & && D_{ij} = 0 \\ & && D \succ 0 \end{aligned}$$

Because αD is also optimal when D is optimal for $\alpha > 0$ and D is homogeneous in all expression we can scale D arbitrarily such that

$$D \succ 0 \Rightarrow D - I \succeq 0$$

$$z^T \tilde{D} z > 0$$

$$k z^T \tilde{D} z \geq z^T I z$$

$$k z^T \tilde{D} z - z^T I z \geq 0 \Rightarrow z^T (k\tilde{D} - I) z \geq 0$$

$$D = k\tilde{D}$$

We use bisection to find β^* starting with a feasible $\beta = u$ and infeasible $\beta = l$

if $\beta = \frac{u+l}{2}$ is feasible

$$u_{next} = \frac{u+l}{2}$$

if $\beta = \frac{u+l}{2}$ is infeasible

$$l_{next} = \frac{u+l}{2}$$

we iterate until convergence

$$\begin{bmatrix} d_1 & & 0 \\ & d_2 & \\ 0 & & d_3 \\ & & & d_4 \end{bmatrix} \underbrace{\begin{bmatrix} 1 & 1 & 1 & 1 \\ a_{i1} & a_{i2} & a_{i3} & a_{i4} \\ 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} d_1 & & 0 \\ & d_2 & \\ & & d_3 \\ 0 & & & d_4 \end{bmatrix}}$$

$$M = \begin{bmatrix} d_1 & d_2 & d_3 & d_4 \end{bmatrix} \begin{bmatrix} a_{i1}d_1 & a_{i2}d_2 & a_{i3}d_3 & a_{i4}d_4 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} a_{11}d_1^2 & a_{12}d_1d_2 \\ & \\ & \end{bmatrix}$$

$\lambda(AD) -$

$M_{ij} = a_{ij} d_i d_j$

$$\leq \lambda_{max}(D)\lambda_{max}(AD) - \beta\lambda_{min}(D)\lambda_{min}(AD) \leq 0$$

$$\lambda_{max}(DAD) - \beta\lambda_{min}(DAD) \leq \lambda_{max}(D)\lambda_{max}(A)\lambda_{max}(D) - \beta\lambda_{min}(D)\lambda_{min}(A)\lambda_{min}(D) \leq 0$$

$$\lambda_{max}(A)\lambda_{max}^2(D) - \beta\lambda_{min}(A)\lambda_{min}^2(D) \leq 0$$

$$\|DAD\|_2 \|D^{-1}A^{-1}D^{-1}\|_2$$

$$= \sqrt{\lambda_{max}(DA^TDDAD)} \cdot \sqrt{\lambda_{max}(D^{-1}A^{-1}D^{-1}D^{-1}AD^{-1})}$$

$$\frac{\lambda^2_{max}(A)}{\lambda^2_{min}(A)} = \frac{\lambda_{max}(A^TA)}{\lambda_{min}(A^TA)}$$

$$I_n \preceq DA^TDDAD \preceq aI_n$$

$$D^{-1} \preceq ADDAD \preceq aD^{-1}$$

$$(D^{-1})^2 \preceq ADDA \preceq a(D^{-1})^2$$

$$\tilde{D} = D^2$$

7. *Optimal diagonal preconditioner for a PD matrix.* Let $A \in \mathbf{S}^n_{++}$. We seek a diagonal matrix D with positive diagonal entries that minimizes the condition number of DAD,

$$\mathbf{cond}(DAD) = \frac{\lambda_{\max}(DAD)}{\lambda_{\min}(DAD)}.$$

We will simply assume that an optimal D exists. Note also that if D is optimal, so is αD for any $\alpha > 0$. This optimization problem arises in numerical methods where D represents a diagonal preconditioner, but you don't need to know this to solve this problem.

(a) Explain how to use convex optimization to find an optimal diagonal preconditioner for A. You will receive half credit for a valid quasiconvex formulation.

Minimize $\quad \mathbf{cond}(DAD)$

$$\mathbf{cond}(DAD) = \sqrt{\mathbf{cond}(DA^TDDAD)}$$

$$I_n \le DA^TDDAD \le kI_n$$

$$\tilde{D} = DD \qquad \tilde{D}^{-1} = D^{-1}D^{-1}$$

$$\tilde{D}^{-1} \le A^T\tilde{D}A \le k\tilde{D}^{-1}$$

$$\begin{cases} A^T\tilde{D}A \ge \tilde{D} \\ k\tilde{D}^{-1} \ge A^T\tilde{D}A \\ k \ge 0 \\ \tilde{D} \ge I \end{cases}$$

7. *Optimal diagonal preconditioner for a PD matrix.* Let $A \in \mathbf{S}^n_{++}$. We seek a diagonal matrix D with positive diagonal entries that minimizes the condition number of DAD,

$$\mathbf{cond}(DAD) = \frac{\lambda_{\max}(DAD)}{\lambda_{\min}(DAD)}.$$

We will simply assume that an optimal D exists. Note also that if D is optimal, so is αD for any $\alpha > 0$. This optimization problem arises in numerical methods where D represents a diagonal preconditioner, but you don't need to know this to solve this problem.

(a) Explain how to use convex optimization to find an optimal diagonal preconditioner for A. You will receive half credit for a valid quasiconvex formulation.

$= \|A^{1/2}D\|^2$ $= \|A^{\frac{1}{2}}D\|^2$

$\log \lambda_+(DAD) - \log \lambda_-(DAD)$

$\text{inv}(DAD) = D^{-1}A^{-1}D^{-1}$

λ_-

8. *Control variable selection via SDP.* Let $x \sim \mathcal{N}(0, \Sigma)$ be a multivariate Gaussian random variable, with a known $\Sigma \in \mathbf{S}^n_{++}$. We want to create *control variables* $\tilde{x} \sim \mathcal{N}(0, \tilde{\Sigma})$, and consider the joint variable $\chi = (x, \tilde{x}) \in \mathbf{R}^{2n}$. Here, $\chi \sim \mathcal{N}(0, \Lambda)$ is jointly Gaussian, with $\Lambda \in \mathbf{S}^{2n}_{++}$. We wish to find Λ such that the following properties hold:

   - $x = (\chi_1, \ldots, \chi_n) \sim N(0, \Sigma)$.
   - χ_i and χ_{i+n} are exchangeable. This means we can swap x_i and $\tilde{x}_i$ for $i = 1, \ldots, n$ without changing the distribution of χ and have the same Λ after the exchange.
   - χ_i and χ_{i+n} are as negatively correlated as possible for $i = 1, \ldots, n$. Precisely, we want to minimize $\sum_{i=1}^n \Lambda_{i(n+i)}$.

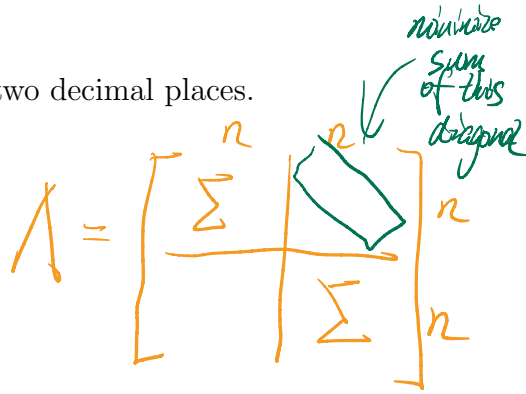
   Remark. This procedure is also known as the *model-x Gaussian knockoffs* procedure. Using properties of conditional Gaussians, we can sample knockoffs $\tilde{x}$ for observed data x. These knockoffs can be used for evaluating feature importances (even when valid p-values cannot be obtained) and controlling the false discovery rate. You do not need to know this for the problem.

   (a) Explain how you can find Λ by solving a semidefinite program (SDP).

   (b) Carry out the procedure outlined in (a) for the problem instance with $n = 3$ and

$$\Sigma = \begin{bmatrix} 4.9 & -3.8 & 1.4 \\ -3.8 & 3.8 & -1.9 \\ 1.4 & -1.9 & 2.5 \end{bmatrix}.$$

   Report $\Lambda^\star_{i(n+i)}$ for $i = 1, 2, 3$. Round reported values to two decimal places.



Minimize $\sum_{i=1}^{n} \Lambda_{i(n+i)}$

$\Lambda \succeq 0 \Rightarrow \Lambda^T = \Lambda$

$x = [y_1\ y_2 \cdots y_n\ \tilde{y}_1\ \tilde{y}_2 \cdots \tilde{y}_n]$

$Cov(\chi) = Cov\left(\begin{bmatrix} 0 & I \\ I & 0 \end{bmatrix} \chi\right) \Rightarrow \Lambda$

$Cov(\chi) = \begin{bmatrix} 0 & I \\ I & 0 \end{bmatrix} Cov(\chi) \begin{bmatrix} 0 & I \\ I & 0 \end{bmatrix} \Rightarrow \quad \Lambda \begin{bmatrix} 0 & I \\ I & 0 \end{bmatrix} = \begin{bmatrix} 0 & I \\ I & 0 \end{bmatrix} \Lambda$

(a) Our SPD problem is

$$\text{Minimize} \quad \sum_{i=1}^{n} \Lambda_{i(n+i)}$$

- $x = (\chi_1, \ldots, \chi_n) \sim N(0, \Sigma)$. $\Rightarrow \Lambda[:n, :n] = \Sigma$

- χ_i and χ_{i+n} are exchangeable. This means we can swap x_i and $\tilde{x}_i$ for $i = 1, \ldots, n$ without changing the distribution of χ and have the same Λ after the exchange.

$\Rightarrow \Lambda[:n, :n] = \Sigma = \Lambda[-n:, -n:]$

Because every single χ_i and χ_{i+n} is swappable the correlation is the same for (χ_i, χ_j) and (χ_{i+n}, χ_j)

but since we want to minimize the diagonal entries of the off diagonal block, we introduce a $n \times n$ diagonal matrix δ to let them vary

$\Rightarrow \Lambda[:n, -n:] = \Lambda[-n:, :n] = \Sigma + \text{diag}(\delta)$

$\Rightarrow \Lambda \succeq 0$

Final_SM24

August 17, 2024

[1]:
```
import numpy as np
import cvxpy as cp
import matplotlib.pyplot as plt
import sys, os
import seaborn as sns
sys.path.insert(0, os.path.abspath('/home/qdeng/Github/
 ↪cvxbook_additional_exercises/python'))
```

[2]:
```
cp.version.full_version
```

[2]:
```
'1.5.2'
```

0.1 Problem 1 – Completed

[3]:
```
from seasonal_shading_data import *
```

[4]:
```
T = s.shape[0]
n = 365 # Periodicity
omega = 300 # hyperparameter
T
```

[4]:
```
1095
```

[5]:
```
gamma = cp.Variable(T)

obj = (
    -1 * cp.sum(
        [cp.log(gamma[t])+gamma[t]*np.log(s[t]) for t in range(T)]
    )
    + omega * cp.sum(
        [cp.square(gamma[t+1]-gamma[t]) for t in range(n)]
    )
    + omega * cp.square(gamma[365]-gamma[0])
)

obj
```

[5]:
```
Expression(CONVEX, UNKNOWN, ())
```

[6]:
```
constraints = []
constraints += [gamma[t] == gamma[t+n] for t in range(T-n)]
```

[7]:
```
prob = cp.Problem(
    cp.Minimize(obj),
    constraints
)
prob.solve()
```

```
/home/qdeng/.pyenv/versions/3.12.1/envs/cvx/lib/python3.12/site-
packages/cvxpy/problems/problem.py:158: UserWarning: Objective contains too many
subexpressions. Consider vectorizing your CVXPY code to speed up compilation.
  warnings.warn("Objective contains too many subexpressions. "
```

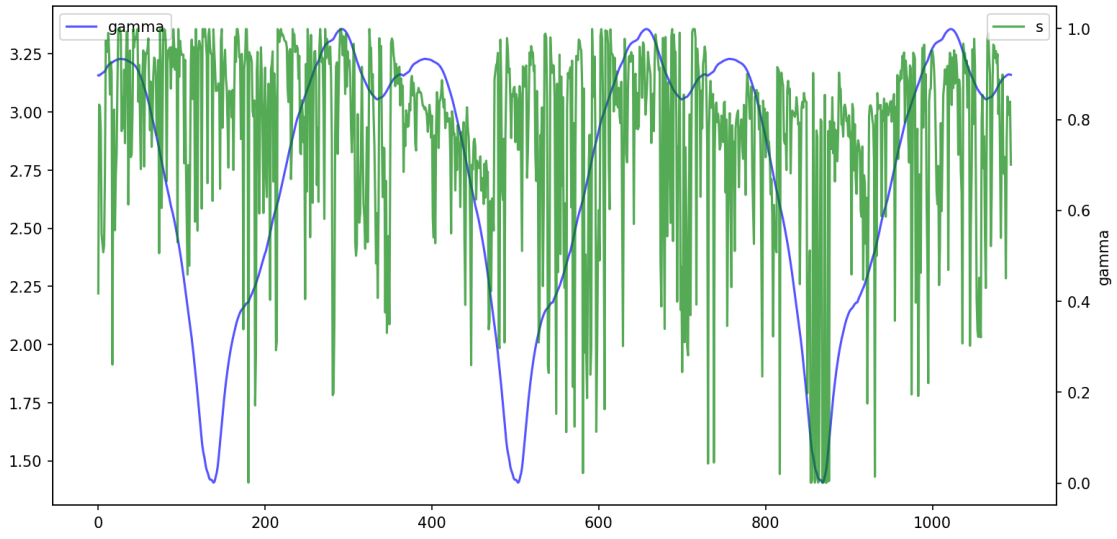
[7]:
```
np.float64(-4.579298006620498)
```

[8]:
```
fig, ax = plt.subplots(figsize=(12,6), dpi=150)
ax.plot(range(T),gamma.value,'b',label='gamma',alpha=2/3)


ax2 = ax.twinx()
ax2.plot(range(T),s,'g',label='s',alpha=2/3)
ax2.set_ylabel('s')

ax.legend(loc=2)
ax2.legend(loc=1)
ax2.set_ylabel('gamma')
# ax.grid()
```

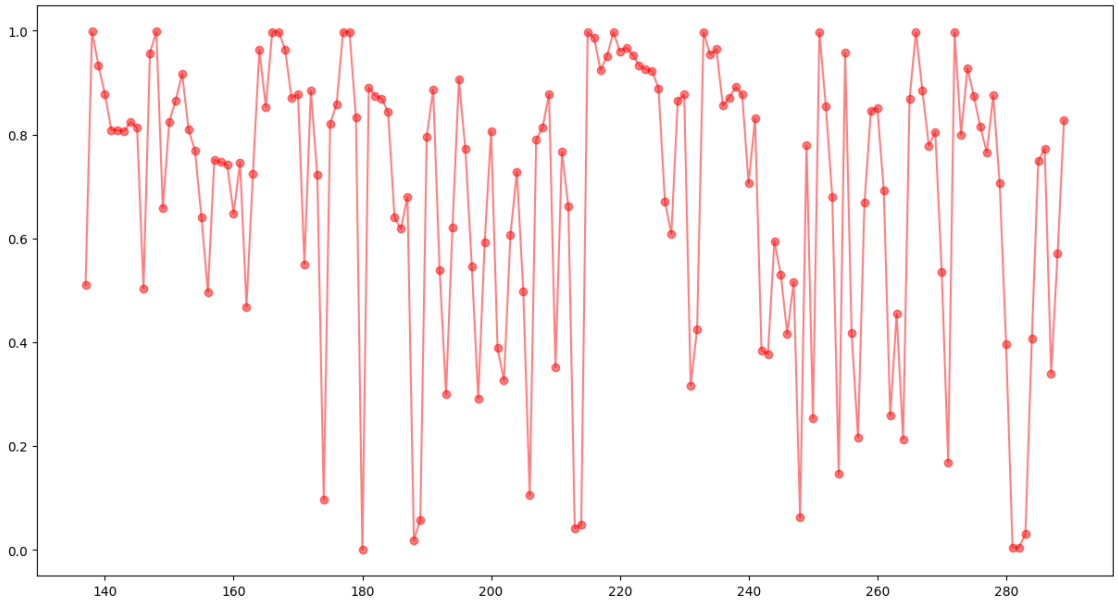
[8]:
```
Text(0, 0.5, 'gamma')
```



[9]:
```
T_plt = np.arange(138-1, 290, 1) # Zero indexing in Python and 1-indexing in␣
↪Problem
T_plt

fig, ax = plt.subplots(figsize=(15,8))
ax.plot(T_plt,
        [s[t]**gamma.value[t] for t in T_plt],
         'ro-',label='gamma',alpha=1/2)
```

[9]:
```
[<matplotlib.lines.Line2D at 0x73d04ecd9640>]
```

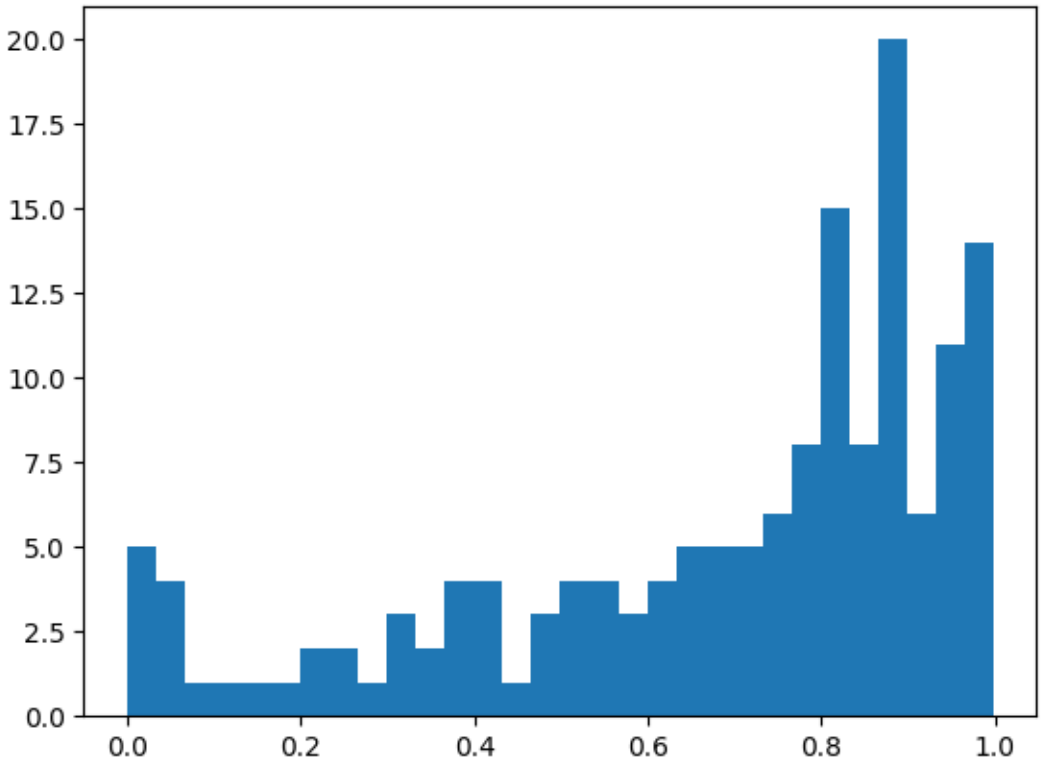


[10]:
```
fig, ax = plt.subplots()

ax.hist([s[t]**gamma.value[t] for t in T_plt],bins=30)
```

[10]:
```
(array([ 5.,  4.,  1.,  1.,  1.,  1.,  2.,  2.,  1.,  3.,  2.,  4.,  4.,
         1.,  3.,  4.,  4.,  3.,  4.,  5.,  5.,  5.,  6.,  8., 15.,  8.,
        20.,  6., 11., 14.]),
 array([2.84897304e-07, 3.32782113e-02, 6.65561377e-02, 9.98340640e-02,
        1.33111990e-01, 1.66389917e-01, 1.99667843e-01, 2.32945770e-01,
        2.66223696e-01, 2.99501622e-01, 3.32779549e-01, 3.66057475e-01,
        3.99335401e-01, 4.32613328e-01, 4.65891254e-01, 4.99169181e-01,
        5.32447107e-01, 5.65725033e-01, 5.99002960e-01, 6.32280886e-01,
        6.65558813e-01, 6.98836739e-01, 7.32114665e-01, 7.65392592e-01,
        7.98670518e-01, 8.31948444e-01, 8.65226371e-01, 8.98504297e-01,
```

```
       9.31782224e-01, 9.65060150e-01, 9.98338076e-01]),
<BarContainer object of 30 artists>)
```



Comments: - the optimal objective value of the problem is -4.579 - Over the course of three years, we observe higher gamma around the summer months (around July), and lower around winter month (around December, that is July + ~150 day) - The variation in gamma following the pattern in s with periodicity constraints. While the first year did not see a low shade fraction in the winter months, the drop in winter months for the second and third years have a significant impact on our estimation of gamma and result in low gamma, which correponds to our interpretation that a lower gamma means the shading factor is getting more likely than higher shade factors.

0.2 Problem 2 – Completed

[3]:
```
from smooth_ride_plot import *
```

[4]:
```
T = 301  # Total time periods (in seconds)
K = 5  # Number of green lights
L = 3000  # Total length of the route (in meters)
g = [10,50,100,200,240]# Time green light turns on
r = [40,80,130,230,270]# Time green light turns off
l = [300,825,1620,1900,2800]# Positions of the lights (in meters)
S_min = 4.0  # Minimum speed (m/s)
```

```
S_max = 16.0  # Maximum speed (m/s)
```

[5]:
```
p = cp.Variable(T)
s = cp.Variable(T-1)
a = cp.Variable(T-2)
j = cp.Variable(T-3)

# obj = 1/(T-2)*cp.sum_squares(j)
obj = 1/(T-2)*cp.sum([cp.square(j_i) for j_i in j])

obj
```

[5]:
```
Expression(CONVEX, NONNEGATIVE, ())
```

[6]:
```
constraints = []
constraints += [j[t]==a[t+1]-a[t] for t in range(T-3)]
constraints += [a[t]==s[t+1]-s[t] for t in range(T-2)]
constraints += [s[t]==p[t+1]-p[t] for t in range(T-1)]
constraints += [p[0]==0]
constraints += [p[-1]==L]
constraints += [s[0]==0]
constraints += [s[i+1]-S_min>=0 for i in range(T-2)]
constraints += [s-S_max<=0]

# Red Light constraints
for k in range(K):
    constraints += [
        p[t]>=l[k] for t in np.arange(r[k], T)
    ]

# Green Light constraints
for k in range(K):
    constraints += [
        p[t]<=l[k] for t in np.arange(0,g[k])
    ]
```

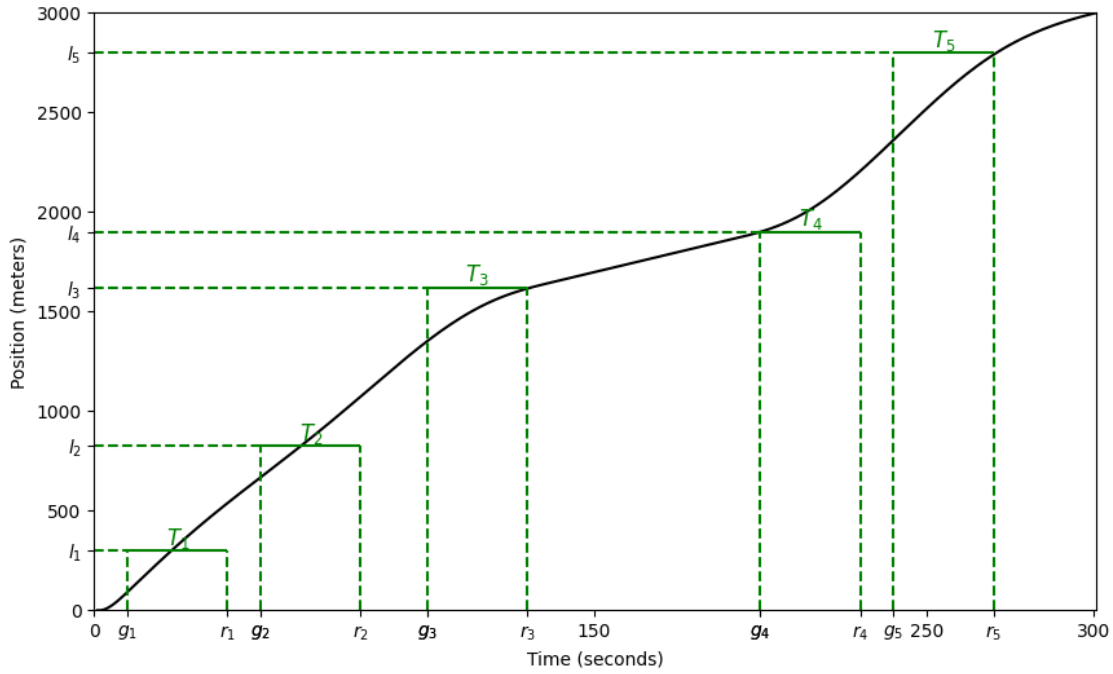
[7]:
```
prob = cp.Problem(
    cp.Minimize(obj),
    constraints
)
prob.solve('CLARABEL')
```

[7]:
```
np.float64(0.0069189503127390925)
```

[8]:
```
prob.is_dcp()
```

[8]:
```
True
```

[9]:
```
plot_trajectory(p.value, l, g, r)
```



Comment: - Compare to the piecewise linear trajectory shown in the problem prompt which violates Maximum Speed and ignore the jerkiness of the ride, the optimal trajectory that minimizes the jerk looks very much like a smoothened version of it – in the sense that, while we cross Light 1 and 2 in the middle of the green, we really waited until the last possible second to cross Light 3 and 5, and crossed Light 4 at the earliest possible instant.

0.3 Problem 3 – Completed

[3]:
```
from cccv_charging_plot import *
```

[4]:
```
Q_max = 6300
Q_min = 960
R = 0.4
a = 3.4
b = 500
Q_crit = 6925
I_max = 1.5
V_max = 4.22
E = 20975
T_fast = 120
T_normal = 180
T_slow = 240
h = 60
```

[5]:
```
results = []

for T in [T_fast, T_normal, T_slow]:
    i = cp.Variable(T)
    q = cp.Variable(T+1)

    v_oc = cp.Variable(T)
    v = cp.Variable(T)

    obj = h*R*cp.sum([cp.square(i_i) for i_i in i])

    constraints = []
    constraints += [q[0]==Q_min]
    constraints += [q[-1]==Q_max]
    constraints += [q[t+1] == q[t] + h*i[t] for t in range(T)]
    constraints += [v[t] == v_oc[t] + R*i[t] for t in range(T)]
    constraints += [v - V_max<=0]
    constraints += [i - I_max<=0]
    constraints += [i >= 0]
    constraints += [v_oc[t] >= a + b * cp.inv_pos(Q_crit-q[t]) for t in␣
 ↪range(T)] #??? make this DCP

    prob = cp.Problem(
        cp.Minimize(obj),
        constraints
    )

    prob.solve()

    results.append(i.value)
    results.append(v.value)
    results.append(q.value)
```

```
/home/qdeng/.pyenv/versions/3.12.1/envs/cvx/lib/python3.12/site-
packages/cvxpy/problems/problem.py:1407: UserWarning: Solution may be
inaccurate. Try another solver, adjusting the solver settings, or solve with
verbose=True for more information.
  warnings.warn(
```

[6]:
```
constraints[0].dual_value
```

[6]:
```
0.3036175337862522
```

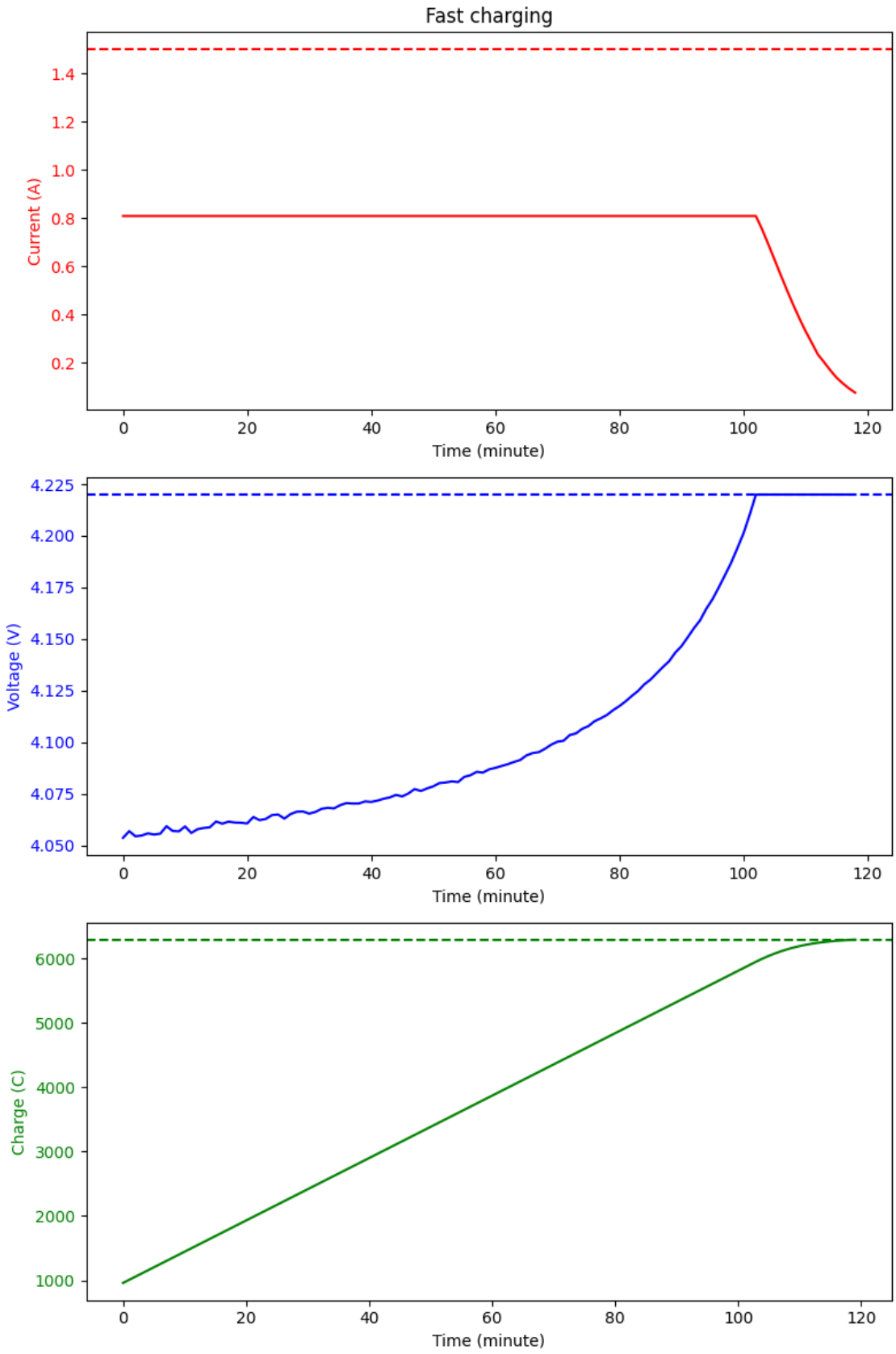
[7]:
```
cp.inv_pos(Q_crit-q[0])
```

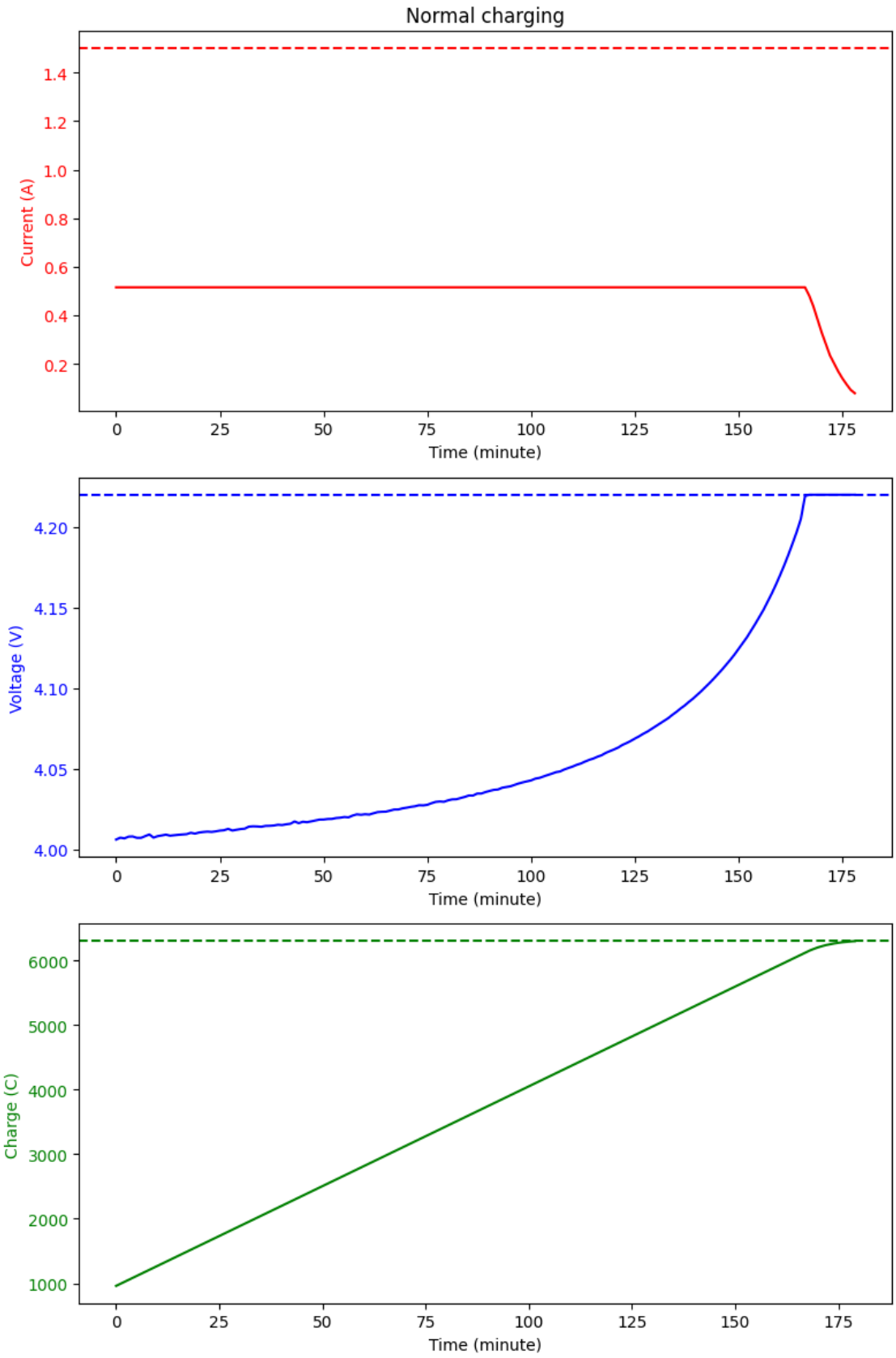
[7]:
```
Expression(CONVEX, NONNEGATIVE, ())
```

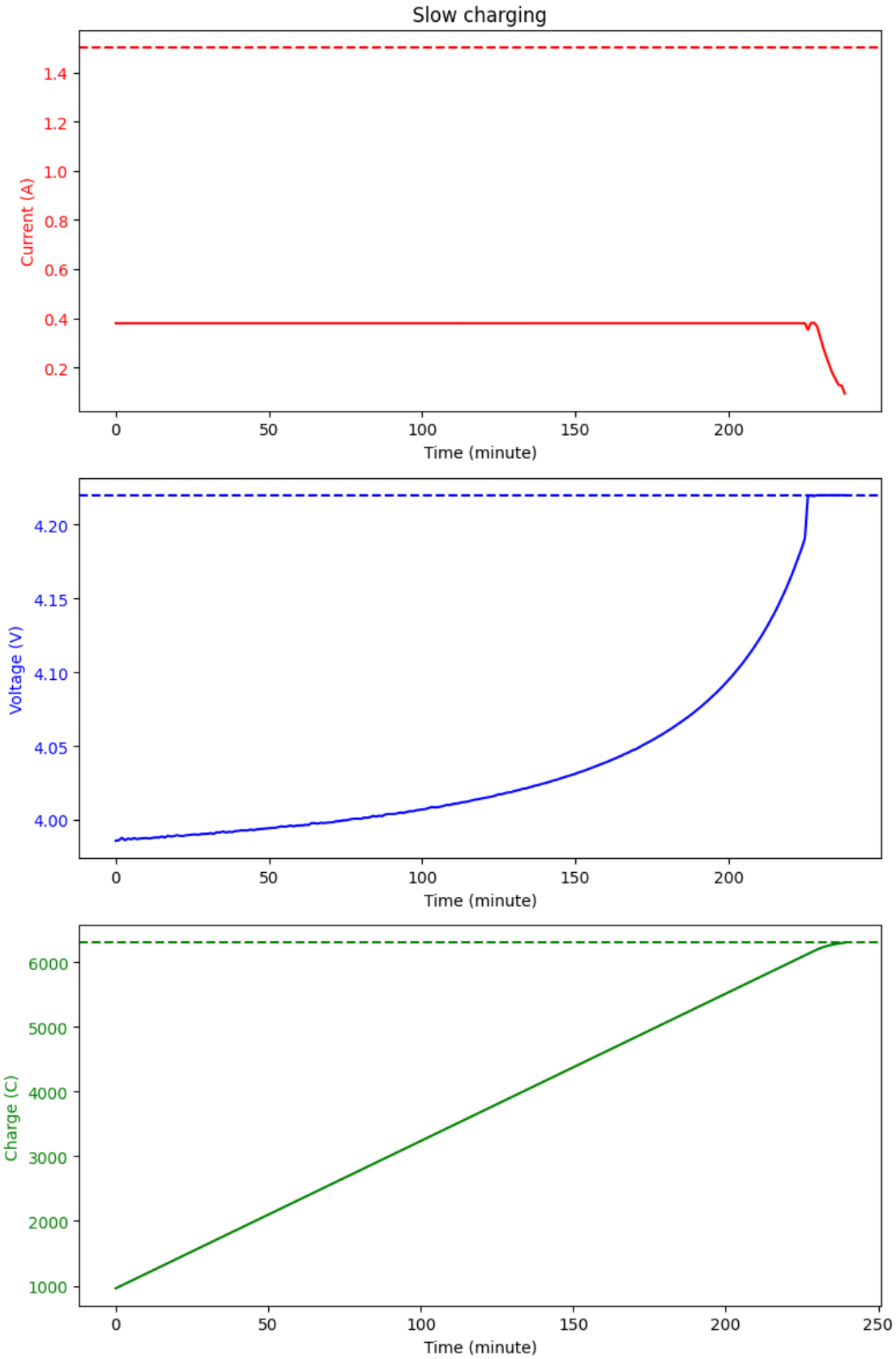
[8]:
```
len(results)
```

[8]: 9

[9]:
```
#Fast charging
# i_fast = i.value#np.zeros((T_fast,))
# v_fast = v.value#np.zeros((T_fast,))
# q_fast = q.value#np.zeros((T_fast,))
# #Normal charging
# i_normal = np.zeros((T_normal,))
# v_normal = np.zeros((T_normal,))
# q_normal = np.zeros((T_normal,))
# #Slow charging
# i_slow = np.zeros((T_slow,))
# v_slow = np.zeros((T_slow,))
# q_slow = np.zeros((T_slow,))
i_fast, v_fast, q_fast, i_normal, v_normal, q_normal, i_slow, v_slow, q_slow =␣
 ↪results
#Example usage
plot_charging(i_fast, v_fast, q_fast, i_normal, v_normal, q_normal, i_slow,␣
 ↪v_slow, q_slow, I_max, V_max, Q_max)
```


Fast charging
Current (A)
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0
20
40
60
80
100
120
Time (minute)
Voltage (V)
4.225
4.200
4.175
4.150
4.125
4.100
4.075
4.050
0
20
40
60
80
100
120
Time (minute)
Charge (C)
6000
5000
4000
3000
2000
1000
0
20
40
60
80
100
120
Time (minute)


Normal charging
Current (A)
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0
25
50
75
100
125
150
175
Time (minute)
Voltage (V)
4.20
4.15
4.10
4.05
4.00
0
25
50
75
100
125
150
175
Time (minute)
Charge (C)
6000
5000
4000
3000
2000
1000
0
25
50
75
100
125
150
175
Time (minute)


Slow charging
Current (A)
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0
50
100
150
200
Time (minute)
Voltage (V)
4.20
4.15
4.10
4.05
4.00
0
50
100
150
200
Time (minute)
Charge (C)
6000
5000
4000
3000
2000
1000
0
50
100
150
200
250
Time (minute)

Part (c) Comment: - The industry Standard CCCV charging is exactly what the solved optimal charging strategy is above. Across all three charging speed, we see a charging profile that consists of a constant current charging for the first majority amount of charging and switching to constant voltage but variable current charging once the maximum allowed voltage has been reached.

0.4 Problem 4 – Completed

[3]:
```
from cond_gauss_reg_data import *
```

[4]:
```
T, n = x.shape
x.shape, y.shape
```

[4]:
```
((1000, 10), (1000,))
```

[5]:
```
omega = cp.Variable(T)
kappa = cp.Variable(T)
alpha = cp.Variable(n)
beta = cp.Variable(n)
u = cp.Variable()
v = cp.Variable()
```

[6]:
```
omega == alpha@x.T+u
```

[6]:
```
Equality(Variable((np.int32(1000),), var1), Expression(AFFINE, UNKNOWN,
(np.int32(1000),)))
```

[7]:
```
kappa == beta@x.T+v
```

[7]:
```
Equality(Variable((np.int32(1000),), var2), Expression(AFFINE, UNKNOWN,
(np.int32(1000),)))
```

[8]:
```
cp.log(omega)-1/2*(cp.multiply(y, omega)-kappa)
```

[8]:
```
Expression(CONCAVE, UNKNOWN, (np.int32(1000),))
```

[9]:
```
cp.multiply(y, omega)
```

[9]:
```
Expression(AFFINE, UNKNOWN, (np.int32(1000),))
```

[10]:
```
cp.square(cp.multiply(y, omega)-kappa)
```

[10]:
```
Expression(CONVEX, NONNEGATIVE, (np.int32(1000),))
```

[11]:
```
obj = cp.sum(
    cp.log(omega)-1/2*cp.square(cp.multiply(y, omega)-kappa)
```

```
)
obj
```

[11]:
```
Expression(CONCAVE, UNKNOWN, ())
```

[12]:
```
constraints = []
constraints += [omega == alpha@x.T+u]
constraints += [kappa == beta@x.T+v]
constraints += [-cp.norm2(alpha)+u>=0]
constraints += [cp.norm_inf(alpha)<=1]
# constraints += [alpha@x[i,:]+u>=0 for i in range(T)]
# constraints += [omega>=0]
```

[16]:
```
prob = cp.Problem(
    cp.Maximize(obj),
    constraints
)
prob.solve()
```

[16]:
```
np.float64(400.59972701237064)
```

[17]:
```
sigma = 1/omega.value
mu = np.multiply(omega.value, kappa.value)
```

[18]:
```
print('Distribution Mean of y1 given x1: %.2f'%mu[0]**2)
print('Distribution Variance of y1 given x1: %.2f'%sigma[0]**2)
```

```
Distribution Mean of y1 given x1: 1.51
Distribution Variance of y1 given x1: 0.16
```

0.5 Problem 5 – Completed

[3]:
```
n = 5
lamb = 10
C = cp.Variable((n,n))

Cg = np.array(
[
    [1, -0.5, 0, 0, 0.7],
    [-0.5, 1, 0, -0.6, 0.8],
    [0, 0, 1, 0.3, 0],
    [0, -0.6, 0.3, 1, 0],
    [0.7, 0.8, 0, 0, 1],
]
)
```

```
Cg==Cg.T
```

[3]:
```
array([[ True,  True,  True,  True,  True],
       [ True,  True,  True,  True,  True],
       [ True,  True,  True,  True,  True],
       [ True,  True,  True,  True,  True],
       [ True,  True,  True,  True,  True]])
```

[4]:
```
Cg!=0
```

[4]:
```
array([[ True,  True, False, False,  True],
       [ True,  True, False,  True,  True],
       [False, False,  True,  True, False],
       [False,  True,  True,  True, False],
       [ True,  True, False, False,  True]])
```

[5]:
```
Cg[Cg!=0]
```

[5]:
```
array([ 1. , -0.5,  0.7, -0.5,  1. , -0.6,  0.8,  1. ,  0.3, -0.6,  0.3,
        1. ,  0.7,  0.8,  1. ])
```

[6]:
```
obj = -cp.log_det(C) + lamb*cp.norm1(cp.hstack([C[Cg!=0]-Cg[Cg!=0]]))
obj
```

[6]:
```
Expression(CONVEX, UNKNOWN, ())
```

[7]:
```
constraints = []
constraints += [C>>0]
constraints += [C[i,i]==1 for i in range(n)]
```

[8]:
```
prob = cp.Problem(
    cp.Minimize(obj),
    constraints
)
prob.solve()
```

[8]:
```
np.float64(14.152968271939406)
```

[9]:
```
constraints[0].dual_value
```

[9]:
```
array([[ 1.65250690e-15,  0.00000000e+00, -1.46062354e-16,
         1.16849883e-15,  7.78999222e-16],
       [ 0.00000000e+00,  1.65250690e-15, -3.89499611e-16,
        -3.89499611e-16,  1.16849883e-15],
       [-1.46062354e-16, -3.89499611e-16,  6.05919196e-15,
         1.94749805e-15, -2.28831021e-15],
       [ 1.16849883e-15, -3.89499611e-16,  1.94749805e-15,
```

```
 -5.50835632e-16, -7.78999222e-16],
[ 7.78999222e-16,  1.16849883e-15, -2.28831021e-15,
 -7.78999222e-16,  2.75417816e-15]])
```

[10]:
```
np.round(C.value, 2)
```

[10]:
```
array([[ 1.  , -0.48,  0.09,  0.29,  0.48],
       [-0.48,  1.  , -0.18, -0.6 ,  0.48],
       [ 0.09, -0.18,  1.  ,  0.3 , -0.09],
       [ 0.29, -0.6 ,  0.3 ,  1.  , -0.29],
       [ 0.48,  0.48, -0.09, -0.29,  1.  ]])
```

[11]:
```
Cg
```

[11]:
```
array([[ 1. , -0.5,  0. ,  0. ,  0.7],
       [-0.5,  1. ,  0. , -0.6,  0.8],
       [ 0. ,  0. ,  1. ,  0.3,  0. ],
       [ 0. , -0.6,  0.3,  1. ,  0. ],
       [ 0.7,  0.8,  0. ,  0. ,  1. ]])
```

Comment - 6 of the off diagonal entries from C_g were modified 1. Cg(1,2) = -0.5 -> C(1,2) = -0.48 2. Cg(2,1) = -0.5 -> C(2,1) = -0.48 3. Cg(1,5) = 0.7 -> C(1,5) = 0.48 2. Cg(5,1) = 0.7 -> C(5,1) = 0.48 3. Cg(2,5) = 0.8 -> C(2,5) = 0.48 2. Cg(5,2) = 0.8 -> C(5,2) = 0.48

0.6 Problem 6 – Completed

[3]:
```
T = 20
n = 3
m = 4

x0 = np.array([0,0,0])
xT = np.array([1, 2, -1])

A = np.array([
    [1, 0.1, 0],
    [0, 9,-0.1],
    [-0.1, 0, 0.9]
])

B = np.array(
    [
        [0.9, -0.4, 0, 0.6],
        [2.0, 0.7, 0.2, -0.4],
        [0.0, 0.2, -0.3, 1.7]
    ]
)
```

```
list_bin = []
for i in range(2**m):
    bin_index = f'{i:04b}'
    list_bin.append(bin_index)

bool_index = [[True if digit=='1' else False for digit in num] for num in␣
 ↪list_bin]

list_B = []
for i_list in bool_index:
    print(i_list)
    B_i = B.copy()
    B_i[:, i_list] = 0

    list_B.append(B_i)
```

```
[False, False, False, False]
[False, False, False, True]
[False, False, True, False]
[False, False, True, True]
[False, True, False, False]
[False, True, False, True]
[False, True, True, False]
[False, True, True, True]
[True, False, False, False]
[True, False, False, True]
[True, False, True, False]
[True, False, True, True]
[True, True, False, False]
[True, True, False, True]
[True, True, True, False]
[True, True, True, True]
```

[4]:
```
list_feasible = []
for Bi in list_B:
    x = cp.Variable((n,T))
    u = cp.Variable((m,T-1))

    obj = 0

    constraints = []
    constraints += [x[:,0]==x0]
    constraints += [x[:,-1]==xT]
    constraints += [cp.norm_inf(u[:, t])<=1 for t in range(T-1)]
    constraints += [x[:,t+1]==A@x[:,t]+Bi@u[:,t] for t in range(T-1)]

    prob = cp.Problem(
```

```
        cp.Minimize(obj),
        constraints
    )

    if prob.solve()==0:
        list_feasible.append(1)
    else:
        list_feasible.append(0)
```

[5]:
```
list_feasible
```

[5]:
```
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0]
```

[6]:
```
np.array(list_feasible)
```

[6]:
```
array([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0])
```

[7]:
```
np.sum(np.array(bool_index), axis=1)#[list_feasible]
```

[7]:
```
array([0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4])
```

[8]:
```
np.array(bool_index)[(np.array(list_feasible))&(np.sum(np.array(bool_index),␣
 ↪axis=1)==3)==1,:]
```

[8]:
```
array([[False,  True,  True,  True],
       [ True, False,  True,  True],
       [ True,  True,  True, False]])
```

Comment: - The minimum number of actuators need to achieve a trajectory is 1 - This can be achieved by 1. Having only the first (0) actuator: u_0 used, u_1 u_2 u_3 unused 2. Having only the second (1) actuator: u_1 used, u_0 u_2 u_3 unused 3. Having only the fourth (3) actuator: u_3 used, u_0 u_1 u_2 unused

[9]:
```
# Verification for only first actuator
x = cp.Variable((n,T))
u = cp.Variable((m,T-1))

obj = 0

constraints = []
constraints += [x[:,0]==x0]
constraints += [x[:,-1]==xT]
constraints += [cp.norm_inf(u[:, t])<=1 for t in range(T-1)]
constraints += [x[:,t+1]==A@x[:,t]+B@u[:,t] for t in range(T-1)]

constraints += [u[[1,2,3],:]==0]
```

```
prob = cp.Problem(
    cp.Minimize(obj),
    constraints
)

prob.solve()
```

[9]: 0.0

[10]:
```
# Verification for only second actuator
x = cp.Variable((n,T))
u = cp.Variable((m,T-1))

obj = 0

constraints = []
constraints += [x[:,0]==x0]
constraints += [x[:,-1]==xT]
constraints += [cp.norm_inf(u[:, t])<=1 for t in range(T-1)]
constraints += [x[:,t+1]==A@x[:,t]+B@u[:,t] for t in range(T-1)]

constraints += [u[[0,2,3],:]==0]

prob = cp.Problem(
    cp.Minimize(obj),
    constraints
)

prob.solve()
```

[10]: 0.0

[11]:
```
# Verification for only fourth actuator
x = cp.Variable((n,T))
u = cp.Variable((m,T-1))

obj = 0

constraints = []
constraints += [x[:,0]==x0]
constraints += [x[:,-1]==xT]
constraints += [cp.norm_inf(u[:, t])<=1 for t in range(T-1)]
constraints += [x[:,t+1]==A@x[:,t]+B@u[:,t] for t in range(T-1)]

constraints += [u[[0,1,2],:]==0]

prob = cp.Problem(
```

```
    cp.Minimize(obj),
    constraints
)

prob.solve()
```

[11]: 0.0

0.7 Problem 7 –Attempt 1

[3]:
```
A = np.array(
    [
        [0.2, -0.2, 0.6, -0.6] ,
        [-0.2, 0.4, -1.4, 1.3],
        [0.6, -1.4, 5.2, -4.7],
        [-0.6, 1.3, -4.7, 4.4]
    ]
)

n, _ = A.shape

I = np.eye(n)
```

[4]:
```
beta = 0.5
D = cp.Variable((n,n))

obj = 0

constraints = []
constraints += [D[i,j]==0 for i in range (n) for j in range(n) if i!=j]
constraints += [D-I>>0]
constraints += [cp.lambda_max(D)-beta*cp.lambda_min(D)<=0]

prob = cp.Problem(
    cp.Minimize(obj),
    constraints
)

prob.solve()
```

[4]: inf

[5]:
```
A
```

[5]:
```
array([[ 0.2, -0.2,  0.6, -0.6],
       [-0.2,  0.4, -1.4,  1.3],
       [ 0.6, -1.4,  5.2, -4.7],
```

```
        [-0.6,  1.3, -4.7,  4.4]])
```

[6]:
```
U, S, Vh = np.linalg.svd(A)
# U==Vh.T
U.T
```

[6]:
```
array([[-0.088678  ,  0.19660857, -0.71878621,  0.6609294 ],
       [ 0.84455772, -0.1993116 , -0.41296455, -0.27650935],
       [-0.47078505,  0.09100114, -0.54757315, -0.6857433 ],
       [ 0.23920682,  0.95568758,  0.11388731, -0.12833952]])
```

[7]:
```
A_half = np.diag(np.sqrt(S))@U.T
A_half.T@A_half
```

[7]:
```
array([[ 0.2, -0.2,  0.6, -0.6],
       [-0.2,  0.4, -1.4,  1.3],
       [ 0.6, -1.4,  5.2, -4.7],
       [-0.6,  1.3, -4.7,  4.4]])
```

[8]:
```
cp.square(cp.norm(A_half@D))
```

[8]:
```
Expression(CONVEX, NONNEGATIVE, ())
```

[9]:
```
cp.square(cp.norm(A_half@cp.inv_pos(D)))
```

[9]:
```
Expression(UNKNOWN, NONNEGATIVE, ())
```

[10]:
```
lambda_A_max = max(S)
lambda_A_min = min(S)
```

[30]:
```
u = 1000
l = 0

while (u-l)>1e-6:
    beta = (u+l)/2
    # print(beta)
    D = cp.Variable((n,n))

    obj = 0

    constraints = []
    constraints += [D[i,j]==0 for i in range (n) for j in range(n) if i!=j]
    constraints += [D-I>>0]
    constraints += [cp.lambda_max(A@D)-beta*cp.lambda_min(A@D)<=0]
    # constraints += [cp.lambda_max(D@A@D)-beta*cp.lambda_min(D@A@D)<=0] ###␣
↪CVXPY does not accept Matrix Quad Form
    # constraints += [(cp.square(cp.norm(A_half@D))-beta<=0)]
```

```
    prob = cp.Problem(
        cp.Minimize(obj),
        constraints
    )

    val = prob.solve()

    if val==np.inf:
        l = beta
    else:
        u = beta


# Do a final solve with u such that we report a feasible D
beta = u
D = cp.Variable((n,n))

obj = 0

constraints = []
constraints += [D[i,j]==0 for i in range (n) for j in range(n) if i!=j]
constraints += [D-I>>0]
constraints += [cp.lambda_max(A@D)-beta*cp.lambda_min(A@D)<=0]
# obj = 0

# constraints = []
# constraints += [D[i,j]==0 for i in range (n) for j in range(n) if i!=j]
# constraints += [D-I>>0]
# # constraints += [Y==A@D]
# # constraints += [M[i,j] >= A[i,j]*cp.geo_mean(D[i,i], D[j,j]) for i in range␣
 ↪(n) for j in range(n)]
# # constraints += [cp.lambda_max(D@A@D)-beta *cp.lambda_min(D@A@D)<=0]
# # constraints += [cp.square(cp.lambda_max(D))-beta *cp.square(cp.
 ↪lambda_min(D))<=0]
# constraints += [(cp.square(cp.norm(A_half@D))-beta<=0)]

prob = cp.Problem(
    cp.Minimize(obj),
    constraints
)
prob.solve()
```

[30]:
```
0.0
```

[31]:
```
cp.square(cp.max(D))#-beta *cp.square(cp.norm_inf(-D))
```

[31]:
```
Expression(UNKNOWN, NONNEGATIVE, ())
```

[32]:
```
A@cp.square(D)@A
```

[32]:
```
Expression(UNKNOWN, UNKNOWN, (np.int32(4), np.int32(4)))
```

[33]:
```
u, l, beta
```

[33]:
```
(224.08816777169704, 224.08816684037447, 224.08816777169704)
```

[35]:
```
eigval_A = np.linalg.eigvals(A)
cond_A = np.max(eigval_A) / np.min(eigval_A)
cond_A
```

[35]:
```
np.float64(1170.1104124302037)
```

[36]:
```
eigval_DAD = np.linalg.eigvals(D.value@A@D.value)
cond_DAD = np.max(eigval_DAD) / np.min(eigval_DAD)
cond_DAD
```

[36]:
```
np.float64(1170.1082748362933)
```

[37]:
```
D.value
```

[37]:
```
array([[ 9.99954321e-01,  5.14407986e-08, -3.03613969e-09,
         2.37509480e-10],
       [ 1.29185768e-07,  9.99953084e-01,  1.94706254e-08,
        -2.14011441e-08],
       [-9.56315948e-08,  8.87066798e-08,  9.99952288e-01,
         1.42542573e-08],
       [ 1.04730558e-07, -9.52184333e-08,  1.35969837e-08,
         9.99952297e-01]])
```

0.8 Problem 7 –Attempt 2

[ ]:
```
d
```

[133]:
```
A = np.array(
    [
        [0.2, -0.2, 0.6, -0.6] ,
        [-0.2, 0.4, -1.4, 1.3],
        [0.6, -1.4, 5.2, -4.7],
        [-0.6, 1.3, -4.7, 4.4]
    ]
)

n, _ = A.shape
```

```
I = np.eye(n)
```

[157]:
```
d = cp.Variable(n)

i = 0
j = 1
d[i]*A[i,j]*d[j]
```

[157]:
```
Expression(UNKNOWN, UNKNOWN, ())
```

[150]:
```
beta = 10000
D = cp.Variable((n,n))

obj = 0

constraints = []
constraints += [D[i,j]==0 for i in range (n) for j in range(n) if i!=j]
constraints += [D>>I]
# constraints += [beta >= 0]
constraints += [A.T@D@A >= D]
constraints += [beta*D >= A.T@D@A ]

prob = cp.Problem(
    cp.Minimize(obj),
    constraints
)

prob.solve()
```

[150]:
```
inf
```

[151]:
```
constraints
```

[151]:
```
[Equality(Expression(AFFINE, UNKNOWN, ()), Constant(CONSTANT, ZERO, ())),
 Equality(Expression(AFFINE, UNKNOWN, ()), Constant(CONSTANT, ZERO, ())),
 Equality(Expression(AFFINE, UNKNOWN, ()), Constant(CONSTANT, ZERO, ())),
 Equality(Expression(AFFINE, UNKNOWN, ()), Constant(CONSTANT, ZERO, ())),
 Equality(Expression(AFFINE, UNKNOWN, ()), Constant(CONSTANT, ZERO, ())),
 Equality(Expression(AFFINE, UNKNOWN, ()), Constant(CONSTANT, ZERO, ())),
 Equality(Expression(AFFINE, UNKNOWN, ()), Constant(CONSTANT, ZERO, ())),
 Equality(Expression(AFFINE, UNKNOWN, ()), Constant(CONSTANT, ZERO, ())),
 Equality(Expression(AFFINE, UNKNOWN, ()), Constant(CONSTANT, ZERO, ())),
 Equality(Expression(AFFINE, UNKNOWN, ()), Constant(CONSTANT, ZERO, ())),
 Equality(Expression(AFFINE, UNKNOWN, ()), Constant(CONSTANT, ZERO, ())),
 Equality(Expression(AFFINE, UNKNOWN, ()), Constant(CONSTANT, ZERO, ())),
 PSD(Expression(AFFINE, UNKNOWN, (np.int32(4), np.int32(4)))),
```

```
 Inequality(Variable((np.int32(4), np.int32(4)), var26393)),
 Inequality(Expression(AFFINE, UNKNOWN, (np.int32(4), np.int32(4))))]
```

0.9 Problem 8 – Completed

[3]:
```
sigma = np.array(
    [
        [4.9, -3.8, 1.4],
        [-3.8, 3.8, -1.9],
        [1.4, -1.9, 2.5]
    ]
)

n = 3

sigma==sigma.T
```

[3]:
```
array([[ True,  True,  True],
       [ True,  True,  True],
       [ True,  True,  True]])
```

[4]:
```
np.linalg.svd(sigma)
```

[4]:
```
SVDResult(U=array([[-0.69882028, -0.46563551,  0.54298599],
       [ 0.63207126, -0.04660389,  0.7735076 ],
       [-0.33486734,  0.88374863,  0.32688257]]), S=array([9.00790176,
1.86255367, 0.32954456]), Vh=array([[-0.69882028,  0.63207126, -0.33486734],
       [-0.46563551, -0.04660389,  0.88374863],
       [ 0.54298599,  0.7735076 ,  0.32688257]]))
```

[9]:
```
lamb = cp.Variable((2*n, 2*n))
delta = cp.Variable(n)
```

[10]:
```
obj = cp.sum(
    cp.hstack(
        [lamb[i,n+i] for i in range(n)]
    )
)
```

[11]:
```
constraints = []
constraints += [lamb[:n,-n:]==sigma+cp.diag(delta)]
constraints += [lamb[-n:,:n]==sigma+cp.diag(delta)]
constraints += [lamb[:n,:n]==sigma]
constraints += [lamb[-n:,-n:]==sigma]
constraints += [lamb>>0]
# constraints += [W@lambda == W@lambda]
```

[12]:
```
prob = cp.Problem(
    cp.Minimize(obj),
    constraints
)

prob.solve()
```

[12]:
```
np.float64(7.899909909236449)
```

[13]:
```
np.round(lamb.value,2)
```

[13]:
```
array([[ 4.9, -3.8,  1.4,  3.7, -3.8,  1.4],
       [-3.8,  3.8, -1.9, -3.8,  3.8, -1.9],
       [ 1.4, -1.9,  2.5,  1.4, -1.9,  0.4],
       [ 3.7, -3.8,  1.4,  4.9, -3.8,  1.4],
       [-3.8,  3.8, -1.9, -3.8,  3.8, -1.9],
       [ 1.4, -1.9,  0.4,  1.4, -1.9,  2.5]])
```

[14]:
```
for i in range(n):
    print(f'lambda[{i},{i+n}] %.2f'%lamb.value[i, i+n])
```

```
lambda[0,3] 3.70
lambda[1,4] 3.80
lambda[2,5] 0.40
```

[ ]:
```

```