1.1 **Gradient descent and line search.** Let $Q \in \mathbb{R}^{n\times n}$ be a symmetric positive-definite matrix, and $b \in \mathbb{R}^n$ be a given vector. Consider the quadratic optimization problem

$$\min_{x\in\mathbb{R}^n} \frac{1}{2}x^\mathsf{T}Qx - b^\mathsf{T}x. \qquad x \in \mathbb{R}^n$$

Let $f(x) := \frac{1}{2}x^\mathsf{T}Qx - b^\mathsf{T}x$, and denote the eigenvalues of Q as $\lambda_1, \ldots, \lambda_n$.

(a) Find the unique local minimum candidate $x^* \in \mathbb{R}^n$. Prove x^* is a global minimum.

Hint: Any twice-differentiable function f is strictly convex if the Hessian $\nabla^2 f(x)$ is positive-definite for all $x \in \mathbb{R}^n$.

(a)

https://en.wikipedia.org/wiki/Matrix_calculus see scaler by vector identity

$$\nabla_x f = Q^T x - b = Qx - b \quad (1)$$

$$\nabla_x^2 f = Q \quad (2)$$

Since Q is a symmetric pos-def matrix

$$\Rightarrow Q = Q^T \Rightarrow QQ^{-1} = I \Rightarrow Q(Q^T)^{-1} = I$$

$$(Q^T)^{-1} = Q^{-1}$$

$$\Rightarrow Q = PDP^{-1}$$

where $D = \begin{bmatrix} \lambda_1 & & 0 \\ & \lambda_2 & \\ 0 & & \ddots \\ & & & \lambda_n \end{bmatrix}$

$$\Rightarrow D^{-1} = \begin{bmatrix} 1/\lambda_1 & & 0 \\ & \ddots & \\ 0 & & 1/\lambda_n \end{bmatrix}$$

$\Rightarrow Q$ is invertible with $Q^{-1} = PD^{-1}P^{-1}$, which is positive definite as well (3)

$\Rightarrow \nabla^2 f = Q$ means $\nabla^2 f$ is positive definite as well (4)

given a local minimum x^*

$\nabla f = 0$

① + ③ $\Rightarrow$ $Qx^* - b = 0$

$Qx^* = b$

$x^* = Q^{-1}b$ ⑤

④ + ⑤ x^* is a strict local minimum for $f(x)$

$f(x^*) = \frac{1}{2}b^TQ^{-1T}QQ^{-1}b - b^TQ^{-1}b = \frac{1}{2}b^TQ^{-1T}b - b^TQ^{-1}b$

$= -\frac{1}{2}b^TQ^{-1}b$

because $\nabla^2 f = 0$ is positive definite

f is strictly convex

because ⑥ any local minimum of a convex function must be global minimum and ⑦ a strictly convex function has at most one global minimum

x^* is a global minimum.

https://en.wikipedia.org/wiki/Convex_function

https://math.stackexchange.com/questions/337090/if-f-is-strictly-convex-in-a-convex-set-show-it-has-no-more-than-1-minimum

(b) Show that, starting from any initial point $x^{(0)} \in \mathbb{R}^n$, Newton's method with constant step size $\eta = 1$ converges in one iteration to the optimal solution x^*. Hence, performing one step of Newton's method is equivalent to solving the linear system of equations $Qx = b$. What would be the downside of this solution method if n is large (e.g., $n \gg 10^4$) and the matrix Q has no particular structure?

For $x \in \mathbf{dom}\, f$, the vector

$$\Delta x_{\mathrm{nt}} = -\nabla^2 f(x)^{-1} \nabla f(x)$$

is called the *Newton step* (for f, at x). Positive definiteness of $\nabla^2 f(x)$ implies that

$$\nabla f(x)^T \Delta x_{\mathrm{nt}} = -\nabla f(x)^T \nabla^2 f(x)^{-1} \nabla f(x) < 0$$

unless $\nabla f(x) = 0$, so the Newton step is a descent direction (unless x is optimal). The Newton step can be interpreted and motivated in several ways.

$\nabla^2 f = Q$ is pos. definite

then $\Delta x_{nt} = -\nabla^2 f(x)^{-1} \nabla f(x)$ for any x

$x + \Delta x_{nt} = x - \nabla^2 f(x)^{-1} \nabla f(x)$

$\nabla_x f = Qx - b$ (a)①

$\nabla_x^2 f = Q$ (a)②

$= x - Q^{-1}(Qx - b)$

$= x - Ix - Q^{-1}b$

$= Q^{-1}b = x^*$ see (a)⑤

Downside when $n \gg 10^4$ and Q has no particular structure

→ we need to matrix inverse, which is expensive when the matrix is large if Q doesn't has any particular structure to make it easily invertible

(c) Let $S \in \mathbb{R}^{n\times n}$ be a symmetric matrix. By the Spectral Theorem, there exist an orthogonal matrix $U \in \mathbb{R}^{n\times n}$ and a diagonal matrix $\Sigma = \mathrm{diag}(\mu_1, \ldots, \mu_n)$ such that $S = U\Sigma U^\mathsf{T}$. Show $\|Sx\|_2 = \|\Sigma U^\mathsf{T} x\|_2$ for any $x \in \mathbb{R}^n$. Then show $\|\Sigma z\|_2 \leq \max_{i\in\{1,\ldots,n\}} |\mu_i| \|z\|_2$ for any $z \in \mathbb{R}^n$. Finally, conclude that $\|Sx\|_2 \leq \max_{i\in\{1,\ldots,n\}} |\mu_i| \|x\|_2$ for any $x \in \mathbb{R}^n$.

Hint: If $U \in \mathbb{R}^{n\times n}$ is an orthogonal matrix, then $\|Uy\|_2 = \|U^\mathsf{T} y\|_2 = \|y\|_2$ for any $y \in \mathbb{R}^n$.

https://en.wikipedia.org/wiki/Orthogonal_matrix

https://inst.eecs.berkeley.edu/~ee127/sp21/livebook/l_sym_sed.html

(1) WTS $\|Sx\|_2 = \|\Sigma U^T x\|_2$

$$\|Sx\|_2^2 = x^T S^T S x$$
$$= x^T (U\Sigma U^T)^T (U\Sigma U^T) x$$
$$= x^T (U\Sigma U^T)(U\Sigma U^T) x$$

because U is orthogonal matrix
$U^T U = I$

$$\|Sx\|_2^2 = x^T U \Sigma I \Sigma U^T x$$
$$= x^T (\Sigma U^T)^T (\Sigma U^T) x$$
$$= \|\Sigma U^T x\|_2^2$$

thus $\|Sx\|_2 = \|\Sigma U^T x\|_2$

(2) WTS $\|\Sigma z\|_2 \leq \max_{i\in\{1,\ldots,n\}} |\mu_i| \|z\|_2$

$$\|\Sigma z\|_2 = \sqrt{z^T \Sigma^T \Sigma z}$$

$$= \sqrt{\sum_{i=1}^n \mu_i^2 z_i^2} \leq \sqrt{\max_{i\in\{1,\ldots,n\}} |\mu_i|^2} \cdot \sqrt{\sum_{i=1}^n z_i^2} = \max_{i\in\{1,\ldots,n\}} |\mu_i| \cdot \|z\|_2$$

therefore $\|\Sigma z\|_2 \leq \max_i |\mu_i| \|z\|_2$ for any $z \in \mathbb{R}^n$

(3) WTS $\|Sx\|_2 \leq \max_i |\mu_i| \, \|z\|_2$

let $z = U^T x$

we've shown in (1)

$\|Sx\|_2 = \|\Sigma U^T x\|_2 = \|\Sigma z\|_2$

Since we've shown in (2) that

$\|\Sigma z\|_2 \leq \max_i |\mu_i| \, \|z\|_2$ for any $z \in \mathbb{R}^n$

(d) For any $\eta > 0$, show that the eigenvalues of the matrix $I - \eta Q$ are exactly $\{1 - \eta\lambda_i\}_{i=1}^n$.

Hint: Identify an orthonormal basis of vectors $\{v_i\}_{i=1}^n \subset \mathbb{R}^n$ such that $(I - \eta Q)v_i = (1 - \eta\lambda_i)v_i$ for each i.

Let eigenvalues of $I - \eta Q$ be $\varepsilon_1, \varepsilon_2, \ldots, \varepsilon_n$

$$(I - \eta Q)\, V = VD$$

$$(I - \eta Q)v_i - \varepsilon_i I v_i = 0$$

$$(I - \eta Q - \varepsilon_i I)v_i = 0$$

$$[(1 - \varepsilon_i)I - \eta Q]v_i = 0$$

$$[(1 - \varepsilon_i)I - \eta U \Sigma U^T]v_i = 0$$

$$[(1 - \varepsilon_i)UU^T - U(\eta\Sigma)U^T]v_i = 0$$

$$U[(1 - \varepsilon_i)I - \eta\Sigma]U^T = 0$$

Because Q is symmetric matrix
so is ηQ, and so is $I - \eta Q$

then $\exists$ orthogonal matrix V, and diagonal matrix D s.t.

$$(I - \eta Q)V = VD$$

$$I - \eta Q = VDV^T$$

where cols of V are eigenvec of $(I - \eta Q)$

the diagonal entries of D is given by e_i for $i \in [1, n]$

$$D = V^T (I - \eta Q) V$$
$$= V^T (U^T U - U^T \eta \Sigma U) V$$
$$= V^T U^T (I - \eta \Sigma) U V$$
$$= W^T (I - \eta \Sigma) W$$

where $W = UV$ is an orthogonal matrix

since I & Σ are diagonal matrices

the diagonal entries are the eigenvalues

$$e_i = 1 - \eta \lambda_i$$

(e) Consider the gradient descent update rule $x^{(k+1)} = x^{(k)} - \eta \nabla f(x^{(k)})$ at iteration $k \in \mathbb{N}_{\geq 0}$ with a constant step size $\eta > 0$. Define $\delta^{(k)} := \|x^{(k)} - x^*\|_2$ and $\gamma(\eta) := \max_{i \in \{1,\dots,n\}} |1 - \eta\lambda_i|$. Use an inductive argument to show $\delta^{(k)} \leq \gamma(\eta)^k \delta_0$ for all $k \in \mathbb{N}_{\geq 0}$.

$x \in \mathbb{R}$

WTS $\delta^{(k)} \leq \gamma(\eta)^k \delta_0$ for $k \in \mathbb{N}_{\geq 0}$

Let statement $P(k)$ be $\delta^{(k)} \leq \gamma(\eta)^k \delta_0$ for $k = k$

Base Case

First, $k = 0$, $P(0)$

$\delta^0 = \|x^{(0)} - x^*\|_2$

$\gamma(\eta)^0 \delta_0 = \delta_0$

thus $\delta^{(0)} \leq \delta^0$, i.e. $P(0)$ is true

$\nabla_x f = Qx - b$ (a)(1)

$\nabla_x^2 f = Q$ (a)(2)

Induction Spse $P(k)$ is true

we want to show $P(k+1)$: $\|x^{(k+1)} - x^*\|_2 \leq \gamma(\eta)^{k+1} \delta_0$ is also true

$P(k)$ true $\Rightarrow \delta^{(k)} = \|x^{(k)} - x^*\|_2 \leq \gamma(\eta)^k \delta_0$

for $k+1$ we have

$$\delta^{k+1} = \|x^{(k+1)} - x^*\|_2$$

$$= \|x^{(k)} - \eta \nabla f(x^{(k)}) - x^*\|_2$$

$$= \|x^{(k)} - \eta(Qx^{(k)} - b) - x^*\|_2$$

$$= \|x^{(k)} - \eta Q x^{(k)} + \eta b - x^*\|_2$$

$$= \|(I-\eta Q)x^{(k)} + \eta b - x^*\|_2$$

$$= \|U^T(I-\eta\Sigma)U x^{(k)} + \eta b - x^*\|_2 \quad ①$$

$$\eta b - x^* = \eta I b - Q^{-1}b \qquad x^* = Q^{-1}b$$

$$= \eta I b - U^T\Sigma^{-1}U b$$

$$= (\eta I - U\Sigma^{-1}U)b$$

$$= U^T(\eta I - \Sigma^{-1})U b$$

$$= U^T(\eta I - \Sigma^{-1})UQQ^{-1}b$$

$$= U^T(\eta I - \Sigma^{-1})UQ\, x^*$$

$$= U^T(\eta I - \Sigma^{-1})UU^T\Sigma U\, x^*$$

$$= U^T(\eta I - \Sigma^{-1})\Sigma U x^*$$

$$= U^T(\eta\Sigma - I)U x^*$$

$$= -U^T(I-\eta\Sigma)U x^*$$

then ① $= \|U^T(I-\eta\Sigma)U x^{(k)} - U^T(I-\eta\Sigma)U x^*\|_2$

$= \|U^T(I-\eta\Sigma)U(x^{(k)} - x^*)\|_2$

$= \|(I-\eta\Sigma)U^T(x^{(k)} - x^*)\|_2$ by problem (c)

$$\leq \underbrace{\max_{i \in \{1,\dots,n\}} |1-\eta\lambda_i|}_{\gamma(\eta)} \|x^{(k)} - x^*\| \quad \text{by problem (d)}$$

$$= \gamma(\eta)\,\gamma^k(\eta)\,\delta_0 = \gamma^{k+1}\delta_0 \quad \text{by inductive assumption}$$

thus we have proven that if
P(k) is true,
then P(k+1) is also true

thus $\delta^{(k)} \leq \gamma(\eta)^k \delta_0$ is true $\forall k \in \mathbb{N}_{\geq 0}$

(f) Consider gradient descent with exact line search. At each iteration k, denote the descent direction by $d^{(k)} := -\nabla f(x^{(k)})$ and the optimal step size by

$$\eta^{(k)} := \arg\min_{\eta \geq 0} f(x^{(k)} + \eta d^{(k)}).$$

Prove

$$\eta^{(k)} = \frac{\|d^{(k)}\|_2^2}{{d^{(k)}}^\top Q d^{(k)}}.$$

η that minimizes $f(x_k + \eta d_k)$ for $\eta \geq 0$

$$f(x) = \frac{1}{2} x^T Q x^T - b^T x$$

$$\nabla f = Q^T x - b$$

$$\nabla^2 f = Q$$

$$f(x^{(k)} + \eta d^{(k)}) = f(x^{(k)} - \eta \nabla f(x^{(k)}))$$

$$= \frac{1}{2} [x^{(k)} + \eta d^{(k)}]^T Q [x^{(k)} + \eta d^{(k)}] - b^T [x^{(k)} + \eta d^{(k)}]$$

$$\nabla f(x^{(k)} + \eta d^{(k)}) = Q^T (x^{(k)} + \eta d^{(k)}) - b = 0$$

$$\frac{\partial}{\partial \eta} f(x^{(k)} + \eta d^{(k)}) = \frac{\partial f(x^{(k)} + \eta d^{(k)})}{\partial (x^{(k)} + \eta d^{(k)})} \frac{\partial (x^{(k)} + \eta d^{(k)})}{\partial \eta}$$

$$= [Q^T (x^{(k)} + \eta d^{(k)}) - b]^T \cdot d^{(k)}$$

$$= (x^{(k)} + \eta d^{(k)})^T Q d^{(k)} - b^T \cdot d^{(k)}$$

$$\frac{\partial}{\partial \eta}\frac{\partial}{\partial \eta} f(x^{(k)}+\eta d^{(k)}) = \frac{\partial}{\partial (x^{(k)}+\eta d^{(k)})}\left(\frac{\partial}{\partial \eta} f(x^{(k)}+\eta d^{(k)})\right)\cdot \frac{\partial (x^{(k)}+\eta d^{(k)})}{\partial \eta}$$

$$= \left(Q \cdot d^{(k)}\right)^T \cdot d^{(k)}$$

$$= d^{(k)^T} \cdot Q \cdot d^{(k)} > 0$$

since Q is positive definite

$\frac{\partial}{\partial \eta} f(x^{(k)}+\eta d^{(k)}) = 0$ soln presents a global minimum by (a)

$$(x^{(k)}+\eta d^{(k)})^T Q d^{(k)} - b^T \cdot d^{(k)} = 0$$

$$x_k^T Q d_k + \eta d_k^T Q d_k - b d_k = 0$$

$$(x_k^T Q - b^T) d_k + \eta d_k^T Q d_k = 0$$

$$(Q^T x_k - b)^T d_k + \eta d_k^T Q d_k = 0$$

$$\nabla f(x_k)^T d_k + \eta d_k^T Q d_k = 0 \qquad \text{by definition}$$

$$-d_k^T \cdot d_k + \eta d_k^T Q d_k = 0$$

$$\eta = \frac{d_k^T \cdot d_k}{d_k^T Q d_k} = \frac{\|d_k\|_2^2}{d^T Q d_k}$$

(g) For $n = 2$ and $f(x) = \frac{1}{2}(x_1^2 + \gamma x_2^2)$ with $\gamma = 10$, what is the optimal solution x^*? Implement gradient descent with a constant step size and exact line search, starting from $x^{(0)} = (5, 1)$ and $x^{(0)} = (1, 5)$. What do you observe with exact line search? When does gradient descent begin to "zig-zag"? What issue do you observe with a constant step size? Repeat both experiments with $\gamma = 1$. Submit your plots.

$$\nabla f = \begin{bmatrix} x_1 \\ \gamma x_2 \end{bmatrix} = 0 \quad @ \; x^*$$

$$x^* = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$$f = \frac{1}{2} \begin{bmatrix} x_1 & x_2 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & \gamma \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

LQR as a QP. Consider the Linear Time-Invariant (LTI) dynamical system

$$x_{t+1} = Ax_t + Bu_t,$$

where $A \in \mathbb{R}^{n\times n}$ and $B \in \mathbb{R}^{n\times m}$ are given matrices, and $x_t \in \mathbb{R}^n$ and $u_t \in \mathbb{R}^m$ are the system state and applied control input, respectively, at time $t \in \mathbb{N}_{\geq 0}$.

Let $x_0 \in \mathbb{R}^n$ be the fixed initial state and $T \in \mathbb{N}$ be some time horizon. Our goal is to find a sequence of control inputs $u^* := (u_0^*, u_1^*, \ldots, u_{T-1}^*) \in \mathbb{R}^{mT}$ that minimizes the quadratic cost

$$J(u) := x_T^\mathsf{T} Q_T x_T + \sum_{t=0}^{T-1} \Big(x_t^\mathsf{T} Q x_t + u_t^\mathsf{T} R u_t \Big),$$

where $Q_T \in \mathbb{R}^{n\times n}$, $Q \in \mathbb{R}^{n\times n}$, and $R \in \mathbb{R}^{m\times m}$ are positive-definite matrices. Later, we will see how dynamic programming can be used to derive an elegant, recursive solution to this problem. For now, we study a convex least-squares formulation. Specifically, we reformulate the problem of minimizing $J(u)$ as

$$\min_{u\in\mathbb{R}^{mT}} \frac{1}{2} u^\mathsf{T} \tilde{Q} u - \tilde{b}^\mathsf{T} u,$$

where $u := (u_0, u_1, \ldots, u_{T-1}) \in \mathbb{R}^{mT}$ is the vector of stacked control inputs, $\tilde{Q} \in \mathbb{R}^{mT\times mT}$ is a positive-definite matrix, and $\tilde{b} \in \mathbb{R}^{mT}$.

(a) Write down $\tilde{Q}$ and $\tilde{b}$ in terms of Q_T, Q, R, A, B, and x_0.

$$J(u) = x_T^T Q_T x_T + \sum_{t=0}^{T-1} (x_t^T Q x_t + u_t^T R u_t)$$

$$x_0^T Q x_0 + u_0 R u_0 + x_1^T Q x_1 + u_1^T R u_1 + \cdots$$

$$x_{t+1} = Ax_t + Bu_t$$

Assume an optimal u^*

$$\Delta J = J(u^* + \Delta u) - J(u^*)$$
$$\approx J(u^*) + \nabla_u J \, \Delta u - J u^*$$
$$\approx \nabla_u J \, \Delta u = 0$$
$$\nabla_u J \geq 2 R^T u_t = 0$$

$$x_{t+1}^T Q x_{t+1} = (Ax_t + Bu_t)^T Q (Ax_t + Bu_t)$$
$$= (x_t^T A^T + u_t^T B^T) Q (Ax_t + Bu_t)$$
$$= x_t^T A^T Q A x_t + x_t^T A^T Q B u_t + u_t^T B^T Q A x_t + u_t^T B^T Q B u_t$$

$$\sum_{t=0}^{T-1} (x_t^T Q x_t + u_t^T R u_t)$$

$$= x_0^T Q x_0 + \sum_{t=1}^{T-1} (x_t^T A^T Q A x_t + x_t^T A^T Q B u_t + u_t^T B^T Q A x_t + u_t^T B^T Q B u_t)$$

$$x_{t+2}^T Q x_{t+2} = (A x_{t+1} + B u_{t+1})^T Q (A x_{t+1} + B u_{t+1})$$

$$= x_{t+1}^T A^T Q A x_{t+1} + x_{t+1}^T A^T Q B u_{t+1} + u_{t+1}^T B^T Q A x_{t+1} + u_{t+1}^T B^T Q B u_{t+1}$$

$$x_{t+1} = A x_t + B u_t$$

$$x_{t+2} = A x_{t+1} + B u_{t+1}$$

$$= A(A x_t + B u_t) + B u_{t+1}$$

$$= AA x_t + AB u_t + B u_{t+1}$$

$$x_{t+3} = A x_{t+2} + B u_{t+2}$$

$$= A^3 x_t + A^2 B u_t + AB u_{t+1} + B u_{t+2}$$

thus $X_{t+1} = \begin{bmatrix} \times \\ \\ \end{bmatrix}$

1.2 **LQR as a QP.** Consider the Linear Time-Invariant (LTI) dynamical system

$$x_{t+1} = Ax_t + Bu_t,$$

where $A \in \mathbb{R}^{n\times n}$ and $B \in \mathbb{R}^{n\times m}$ are given matrices, and $x_t \in \mathbb{R}^n$ and $u_t \in \mathbb{R}^m$ are the system state and applied control input, respectively, at time $t \in \mathbb{N}_{\geq 0}$.

Let $x_0 \in \mathbb{R}^n$ be the fixed initial state and $T \in \mathbb{N}$ be some time horizon. Our goal is to find a sequence of control inputs $u^* := (u_0^*, u_1^*, \ldots, u_{T-1}^*) \in \mathbb{R}^{mT}$ that minimizes the quadratic cost

$$J(u) := x_T^\mathsf{T} Q_T x_T + \sum_{t=0}^{T-1} \Big(x_t^\mathsf{T} Q x_t + u_t^\mathsf{T} R u_t \Big),$$

where $Q_T \in \mathbb{R}^{n\times n}$, $Q \in \mathbb{R}^{n\times n}$, and $R \in \mathbb{R}^{m\times m}$ are positive-definite matrices. Later, we will see how dynamic programming can be used to derive an elegant, recursive solution to this problem. For now, we study a convex least-squares formulation. Specifically, we reformulate the problem of minimizing $J(u)$ as

$$\min_{u\in\mathbb{R}^{mT}} \frac{1}{2} u^\mathsf{T} \tilde{Q} u - \tilde{b}^\mathsf{T} u,$$

where $u := (u_0, u_1, \ldots, u_{T-1}) \in \mathbb{R}^{mT}$ is the vector of stacked control inputs, $\tilde{Q} \in \mathbb{R}^{mT\times mT}$ is a positive-definite matrix, and $\tilde{b} \in \mathbb{R}^{mT}$.

(a) Write down $\tilde{Q}$ and $\tilde{b}$ in terms of Q_T, Q, R, A, B, and x_0.

LQR - Linear Quadratic Regulator

QP - Quadratic Programming

$$x_{t+1} = Ax_t + Bu_t$$

$$x_{t+2} = Ax_{t+1} + Bu_{t+1}$$

$$= A(Ax_t + Bu_t) + Bu_{t+1}$$

$$= AAx_t + ABu_t + Bu_{t+1}$$

$$x_{t+3} = Ax_{t+2} + Bu_{t+2}$$

$$= A^3 x_t + A^2 Bu_t + ABu_{t+1} + Bu_{t+2}$$

thus let $t = 0$ then

$$x_T = A^T x_0 + A^{T-1} Bu_0 + A^{T-2} Bu_1 + \cdots + ABu_{T-2} + Bu_{T-1}$$

for any $T \in [0, T]$

Let $u = \begin{bmatrix} u_0 \\ u_1 \\ \vdots \\ u_{T-1} \end{bmatrix}$, $X_T = \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{T-1} \\ x_T \end{bmatrix}$

only in terms of x_0

call this matrix C

$$X_T = \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{T-1} \\ x_T \end{bmatrix} = \begin{bmatrix} I \\ A \\ A^2 \\ \vdots \\ A^{T-1} \\ A^T \end{bmatrix} x_0 + \begin{bmatrix} 0 & & & \\ B & 0 & & \\ AB & B & 0 & \\ A^2B & AB & B & 0 \\ \vdots & \vdots & \vdots & \vdots \\ A^{T-1}B & A^{T-2}B & A^{T-3}B & \cdots & AB & B \end{bmatrix} \begin{bmatrix} u_0 \\ u_1 \\ \vdots \\ u_{T-1} \end{bmatrix} = Cx_0 + Du$$

call this matrix D

$$J(u) := x_T^\mathsf{T} Q_T x_T + \sum_{t=0}^{T-1} \left(x_t^\mathsf{T} Q x_t + u_t^\mathsf{T} R u_t \right),$$

call this matrix E

$$x_T^T Q_T x_T + \sum_{t=0}^{T-1} x_t^T Q x_t = \begin{bmatrix} x_0^T & x_1^T & \cdots & x_{T-1}^T & x_T^T \end{bmatrix} \begin{bmatrix} Q & & & & \\ & Q & & 0 & \\ & & Q & & \\ & & & \ddots & \\ & 0 & & & Q_T \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{T-1} \\ x_T \end{bmatrix}$$

$$= (x_0^T C^T + u^T D^T) \; E \; (Cx_0 + Du)$$

$$= x_0^T C^T E C x_0 + x_0^T C^T E D u + u^T D^T E C x_0 + u^T D^T E D u$$

$$\sum_{t=0}^{T-1} u_t^T R u_t$$

$$= \begin{bmatrix} u_0^T & u_1^T & \cdots & u_{T-1}^T \end{bmatrix} \begin{bmatrix} R & & & \\ & R & & \\ & & R & \\ & & & \ddots \\ & & & & R \end{bmatrix} \begin{bmatrix} u_0 \\ u_1 \\ \vdots \\ u_{T-1} \end{bmatrix}$$

call this F

$$= u^T F u$$

$x_0^T C^T E D u = u^T D^T E C x_0$ b/c they are scalars

$$J = x_0^T C^T E C x_0 + x_0^T C^T E D u + u^T D^T E C x_0 + u^T D^T E D u + u^T F u$$

$$= u^T (F + D^T E D) u + 2 x_0^T C^T E D u + x_0^T C^T E C x_0$$

as $x_0^T C^T E C x_0$ does not depend on u, it appears as a constant in minimization over u

thus it's sufficient to minimize cost function

$$J^* = u^T (F + D^T E D) u + 2 x_0^T C^T E D u$$
$$= u^T \tilde{Q} u - \tilde{b}^T u$$

where $\tilde{Q} = F + D^T E D$

$$\tilde{b} = -2D^T E^T C x_0 = -2D^T E^T C x_0$$

$$D = \begin{bmatrix} 0 & & & \\ B & 0 & & 0 \\ AB & B & 0 & \\ A^2B & AB & B & 0 \cdots \\ \vdots & \vdots & \vdots & \vdots \\ A^{T-1}B & A^{T-2}B & A^{T-3}B & \cdots \; AB \; B \; 0 \end{bmatrix} \qquad C = \begin{bmatrix} I \\ A \\ A^2 \\ \vdots \\ A^{T-1} \\ A^T \end{bmatrix}$$

$$E = \begin{bmatrix} Q & & & 0 \\ & Q & & \\ & & Q & \\ 0 & & & Q_T \end{bmatrix} \qquad F = \begin{bmatrix} R & & & \\ & R & & \\ & & R & \\ & & & R \end{bmatrix}$$

(b) With this reformulation, implement the gradient descent algorithm of your choice to compute the optimal sequence of control inputs u^* for

$$Q_T = 10I_2, \quad Q = I_2, \quad R = I_1, \quad A = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}, \quad B = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \quad x_0 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \quad T = 20,$$

where I_n is the identity matrix with dimension n. What is the optimal cost $J(u^*)$?

Please see code.

1.3 Extremal curves. Given the functional

$$J(x) = \int_0^1 \left(\frac{1}{2}\dot{x}(t)^2 + 5x(t)\dot{x}(t) + x(t)^2 + 5x(t) \right) dt,$$

find an extremal curve $x^* : [0,1] \to \mathbb{R}$ that satisfies $x^*(0) = 1$ and $x^*(1) = 3$.

$g(x, \dot{x}, t) = \frac{1}{2}\dot{x}(t)^2 + 5x(t)\dot{x}(t) + x(t)^2 + 5x(t)$

Euler's Eqn States

$$\frac{d}{dx}g - \frac{d}{dt}\left(\frac{d}{d\dot{x}}g\right) = 0$$

$$\frac{d}{dx}g = 5\dot{x} + 2x + 5$$

$$\frac{d}{d\dot{x}}g = \dot{x} + 5x$$

$$\frac{d}{dt}\left(\frac{d}{d\dot{x}}g\right) = \ddot{x} + 5\dot{x}$$

Plugging in Euler's Eqn, we get

$$5\dot{x} + 2x + 5 - (\ddot{x} + 5\dot{x}) = 0$$
$$2x + 5 - \ddot{x} = 0$$

let $y = 2x + 5$, $\dot{y} = 2\dot{x}$, $\ddot{y} = 2\ddot{x}$

$$\Rightarrow \quad y - \frac{1}{2}\ddot{y} = 0$$

Guess a solution of the form

$$y = ke^{ct}$$

$$ke^{ct} - \frac{1}{2}kc^2e^{ct} = 0$$

$$1 - \frac{1}{2}c^2 = 0$$

$$c^2 = 2$$

$$c = \pm\sqrt{2} \Rightarrow y = k_1e^{\sqrt{2}t} + k_2e^{-\sqrt{2}t}$$

$$x = \frac{1}{2}(y-5) = \frac{1}{2}k_1e^{\sqrt{2}t} + \frac{1}{2}k_2e^{-\sqrt{2}t} - \frac{5}{2}$$

$$= k_3e^{\sqrt{2}t} + k_4e^{-\sqrt{2}t} - \frac{5}{2}$$

Since $x(0) = 1$, $x(1) = 3$

$$\Rightarrow \quad k_3 + k_4 - \frac{5}{2} = 1 \Rightarrow k_3 + k_4 = \frac{7}{2}$$

$$k_4 = \frac{7}{2} - k_3$$

$$k_3e^{\sqrt{2}} + k_4e^{-\sqrt{2}} - \frac{5}{2} = 3$$

$$\Rightarrow k_3e^{\sqrt{2}} + \left(\frac{7}{2} - k_3\right)e^{-\sqrt{2}} = \frac{11}{2}$$

$$k_3\left(e^{\sqrt{2}} - e^{-\sqrt{2}}\right) = \frac{11}{2} - \frac{7}{2}e^{-\sqrt{2}}$$

$$k_3 = \frac{11/2 - 7/2e^{-\sqrt{2}}}{e^{\sqrt{2}} - e^{-\sqrt{2}}} = \frac{1}{2}\frac{11 - 7e^{-\sqrt{2}}}{e^{\sqrt{2}} - e^{-\sqrt{2}}}$$

$$k_4 = \frac{7}{2} - \frac{11/2 - 7/2e^{-\sqrt{2}}}{e^{\sqrt{2}} - e^{-\sqrt{2}}} = \frac{1}{2}\frac{7e^{-\sqrt{2}} - 11}{e^{\sqrt{2}} - e^{-\sqrt{2}}}$$

Soln: $x = \frac{1}{2}\frac{11 - 7e^{-\sqrt{2}}}{e^{\sqrt{2}} - e^{-\sqrt{2}}}e^{\sqrt{2}t} + \frac{1}{2}\frac{7e^{-\sqrt{2}} - 11}{e^{\sqrt{2}} - e^{-\sqrt{2}}}e^{-\sqrt{2}t} - \frac{5}{2}$

Zermelo's ship. Zermelo's ship must travel through a region of strong currents. The position of the ship is denoted by $(x(t), y(t)) \in \mathbb{R}^2$. The ship travels at a constant speed $v > 0$, yet its heading $\theta(t)$ can be controlled. The current moves in the positive x-direction with speed $w(y(t))$. The equations of motion for the ship are

$$\dot{x}(t) = v\cos\theta(t) + w(y(t))$$
$$\dot{y}(t) = v\sin\theta(t)$$

We want to control the heading $\theta(t)$ such that the ship travels from a given initial position $(x(t_0), y(t_0)) = (x_0, y_0)$ to the origin $(0, 0)$ in minimum time.

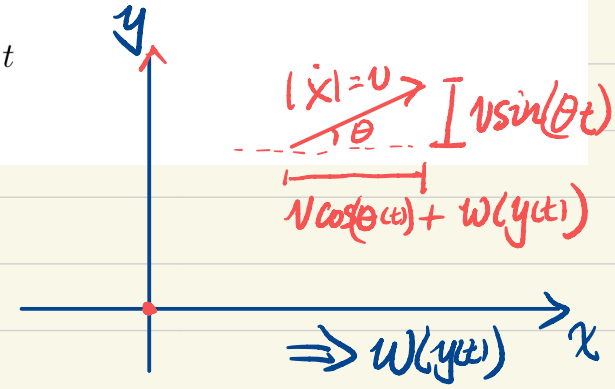
(a) Suppose $w(y(t)) = \frac{v}{h}y(t)$, where $h > 0$ is a known constant. Show that an optimal control law $\theta^*(t)$ must satisfy a linear tangent law of the form

$$\tan\theta^*(t) = \alpha - \frac{v}{h}t$$

for some constant $\alpha \in \mathbb{R}$.

Let $w(y(t)) = \frac{v}{h}y(t)$

Let $X(t) = \begin{bmatrix} x(t) \\ y(t) \end{bmatrix}$



$$\dot{X}(t) = f(x, y, t) = \begin{bmatrix} v\cos\theta(t) + w(y(t)) \\ v\sin\theta(t) \end{bmatrix}$$

we know $X(0) = \begin{bmatrix} x_0 \\ y_0 \end{bmatrix}$ $X(t_f) = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$

minimize $J(t_f) = \frac{1}{2}at_f^2$

thus $h = \frac{1}{2}at_f^2$, $g = 0$

$$h = \frac{1}{2}at_f^2 \Rightarrow \frac{dh}{dt_f} = at_f$$

let $p = \begin{bmatrix} p_1 \\ p_2 \end{bmatrix}$ be the Lagrangian/costate vector

$$H = g + p^T f$$
$$= p^T f$$
$$= p_1 V\cos\theta(t) + p_1 w(y(t)) + p_2 V\sin\theta(t)$$
$$= p_1 V\cos\theta(t) + p_1 \frac{V}{2} y(t) + p_2 V\sin\theta(t)$$

$$\dot{x} = \frac{\partial H}{\partial p} = \begin{bmatrix} V\cos\theta(t) + \frac{V}{2} y(t) \\ V\sin\theta(t) \end{bmatrix}$$

$$\begin{bmatrix} \dot{p}_1 \\ \dot{p}_2 \end{bmatrix} = \dot{p} = -\frac{\partial H}{\partial x} = -\begin{bmatrix} 0 \\ p_1 \frac{V}{2} \end{bmatrix} \Rightarrow \begin{cases} \dot{p}_1 = 0 \Rightarrow p_1 = c_1 \\ \dot{p}_2 = -p_1 \frac{V}{2} \Rightarrow p_2 = -p_1 \frac{V}{2} t + c_2 \\ \qquad = -c_1 \frac{V}{2} t + c_2 \end{cases}$$

$$0 = \frac{\partial H}{\partial \theta} = -p_1 V\sin\theta(t) + p_2 V\cos\theta(t)$$
$$p_1(t)\sin\theta(t) = p_2(t)\cdot\cos\theta(t)$$
$$p_2(t) = p_1(t)\frac{\sin\theta(t)}{\cos\theta(t)} \Rightarrow \frac{p_2}{p_1} = \tan\theta(t)$$
$$\dot{p}_2 = \dot{p}_1 \tan\theta(t) + p_1 \sec\theta(t)$$
($\dot{p}_2 \to -p_1\frac{V}{2}$, $\dot{p}_1 \to 0$)

$$P_1(t)\sin\theta(t) = P_2(t)\cdot\cos\theta(t)$$

$$C_1\sin\theta(t) = -C_1\frac{v}{2}t\cos\theta(t) + C_2\cos\theta(t)$$

$$\tan\theta(t) = \frac{C_2}{C_1} - \frac{v}{2}t$$

Zermelo's ship. Zermelo's ship must travel through a region of strong currents. The position of the ship is denoted by $(x(t), y(t)) \in \mathbb{R}^2$. The ship travels at a constant speed $v > 0$, yet its heading $\theta(t)$ can be controlled. The current moves in the positive x-direction with speed $w(y(t))$. The equations of motion for the ship are

$$\begin{aligned} \dot{x}(t) &= v\cos\theta(t) + w(y(t)) \\ \dot{y}(t) &= v\sin\theta(t) \end{aligned}.$$

We want to control the heading $\theta(t)$ such that the ship travels from a given initial position $(x(t_0), y(t_0)) = (x_0, y_0)$ to the origin $(0, 0)$ in minimum time.

(a) Suppose $w(y(t)) = \frac{v}{h}y(t)$, where $h > 0$ is a known constant. Show that an optimal control law $\theta^*(t)$ must satisfy a linear tangent law of the form

$$\tan\theta^*(t) = \alpha - \frac{v}{h}t$$

for some constant $\alpha \in \mathbb{R}$.

(b) Suppose $w(y(t)) \equiv \beta$ for some constant $\beta > 0$. Derive an expression for the optimal transfer time $t_1^* - t_0$.

Let $X(t) = \begin{bmatrix} x(t) \\ y(t) \end{bmatrix}$

$$\dot{X}(t) = f(x, y, t) = \begin{bmatrix} v\cos\theta(t) + \beta \\ v\sin\theta(t) \end{bmatrix}$$

Minimize $J = \int_0^{t_f} dt \Rightarrow h = 0 \quad g = 1$

$$H = g + p^T \cdot f = 1 + p_1(t)v\cos\theta(t) + p_1\beta + p_2 v\sin\theta(t)$$

$$\dot{p} = -\frac{\partial H}{\partial x} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \Rightarrow \begin{matrix} \dot{p}_1 = 0 \\ \dot{p}_2 = 0 \end{matrix} \Rightarrow \begin{matrix} p_1 = \bar{p}_1 \\ p_2 = \bar{p}_2 \end{matrix}$$

$$0 = \frac{\partial H}{\partial \theta} = -p_1 v\sin\theta(t) + p_2 v\cos\theta(t)$$

$$\tan\theta(t) = +\frac{p_2}{p_1}$$

$$\theta(t) = \tan^{-1}\frac{+p_2}{p_1}$$

$$\left.\begin{aligned} 0 &= (v\cos\theta + \beta)t_f + x_0 \\ 0 &= (v\sin\theta)t_f + y_0 \end{aligned}\right\} \Rightarrow \quad \sin\theta = -\frac{y_0}{vt_f}$$

$$\cos\theta = \sqrt{1-\sin^2\theta}$$

$$= \sqrt{1 - \frac{y_0^2}{v^2 t_f^2}}$$

$$= \sqrt{\frac{v^2t_f^2 - y_0^2}{v^2t_f^2}}$$

$$= \frac{\sqrt{v^2t_f^2 - y_0^2}}{vt_f}$$

$$(v\cos\theta + \beta)t_f + x_0 = \left(v \cdot \frac{\sqrt{v^2t_f^2 - y_0^2}}{vt_f} + \beta\right)t_f + x_0 = 0$$

$$\Rightarrow \sqrt{v^2t_f^2 - y_0^2} + \beta t_f + x_0 = 0$$

Solving for t_f, we get

$$t_f = \frac{x\beta \pm \sqrt{v^2x^2 + v^2y^2 - y^2\beta^2}}{v^2 - \beta^2}$$

we note that if $\lim_{\beta \to 0}$

$$\lim_{\beta \to 0} \frac{x\beta - \sqrt{v^2x^2 + v^2y^2 - y^2\beta^2}}{v^2 - \beta^2} = \frac{-v\sqrt{x^2+y^2}}{v^2} = -\frac{\sqrt{x^2+y^2}}{v}$$

implies a negative time that is infeasible

whereas

$$\lim_{\beta \to 0} \frac{x\beta + \sqrt{v^2x^2 + v^2y^2 - y^2\beta^2}}{v^2 - \beta^2} = \frac{v\sqrt{x^2+y^2}}{v^2} = \frac{\sqrt{x^2+y^2}}{v}$$

therefore $t_f = \dfrac{x\beta + \sqrt{v^2x^2 + v^2y^2 - y^2\beta^2}}{v^2 - \beta^2}$

HW1_code

April 20, 2024

[1]:
```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import jax, jax.numpy as jnp
```

0.1 Problem 1

[2]:
```
  = 10
Q = np.array([
    [1, 0],
    [0,  ]
])



f = lambda x: 1/2 * x.T @ Q @ x

df_dx = jax.grad(f)
```

[3]:
```
f(np.array([0., 1.]).T)
```

[3]:
```
5.0
```

[4]:
```
df_dx(jnp.array([0., 1.]).T)
```

```
/home/qdeng/.pyenv/versions/3.12.1/envs/AA203/lib/python3.12/site-
packages/jax/_src/xla_bridge.py:262: RuntimeWarning: Device 0 has CUDA compute
capability 5.0 which is lower than the minimum supported compute capability 5.2.
See https://jax.readthedocs.io/en/latest/installation.html#nvidia-gpu for more
details
  warnings.warn(
```

[4]:
```
Array([ 0., 10.], dtype=float32)
```

[5]:
```
jax.grad(f)(jnp.array([0., 1.]).T)
```

[5]:
```
Array([ 0., 10.], dtype=float32)
```

[ ]:

[6]:
```
def grad_desc( ,  , x0, n_iter=1e2):
    # Define Q
    Q = np.array([
        [1, 0],
        [0,  ]
    ])

    # Define f
    f = lambda x: 1/2 * x.T @ Q @ x

    # Take gradient with Jax
    df_dx = jax.grad(f)

    # Initiate variables for logging
    x_traj = []
    f_traj = []
    df_traj = []
    x_next = x0

    f_best = np.inf
    x_best = None

    for _ in range(int(n_iter)):
        # Log x, f, df
        x_traj.append(x_next)

        f_next = f(x_next)
        f_traj.append(f_next)

        df_next = df_dx(x_next)
        df_traj.append(df_next)

        # Record the x_best and f_best by comparing with the previous best f
        if f_best>f_next:
            f_best = f_next
            x_best = x_next

        # Perform gradient descent
        x_next = x_next -   * df_dx(x_next)

    x_traj = jnp.array(x_traj)
    f_traj = jnp.array(x_traj)
    df_traj = jnp.array(x_traj)

    return x_traj, f_traj, df_traj
```

[7]:
```
def grad_desc_optimal_step( , x0, n_iter=1e2):
    # Define Q
    Q = np.array([
        [1, 0],
        [0,  ]
    ])

    # Define f
    f = lambda x: 1/2 * x.T @ Q @ x

    # Take gradient with Jax
    df_dx = jax.grad(f)

    # Initiate variables for logging
    x_traj = []
    f_traj = []
    df_traj = []
    x_next = x0

    f_best = np.inf
    x_best = None

    for _ in range(int(n_iter)):
        # Log x, f, df
        x_traj.append(x_next)

        f_next = f(x_next)
        f_traj.append(f_next)

        df_next = df_dx(x_next)
        df_traj.append(df_next)

        # Record the x_best and f_best by comparing with the previous best f
        if f_best>f_next:
            f_best = f_next
            x_best = x_next

        # Calculate the optimal step based on the derived formula
        d = -df_next
          = d.T@d / (d.T@Q@d)

        # Perform gradient descent
        x_next = x_next -   * df_dx(x_next)

    x_traj = jnp.array(x_traj)
```

```
    f_traj = jnp.array(x_traj)
    df_traj = jnp.array(x_traj)

    return x_traj, f_traj, df_traj
```

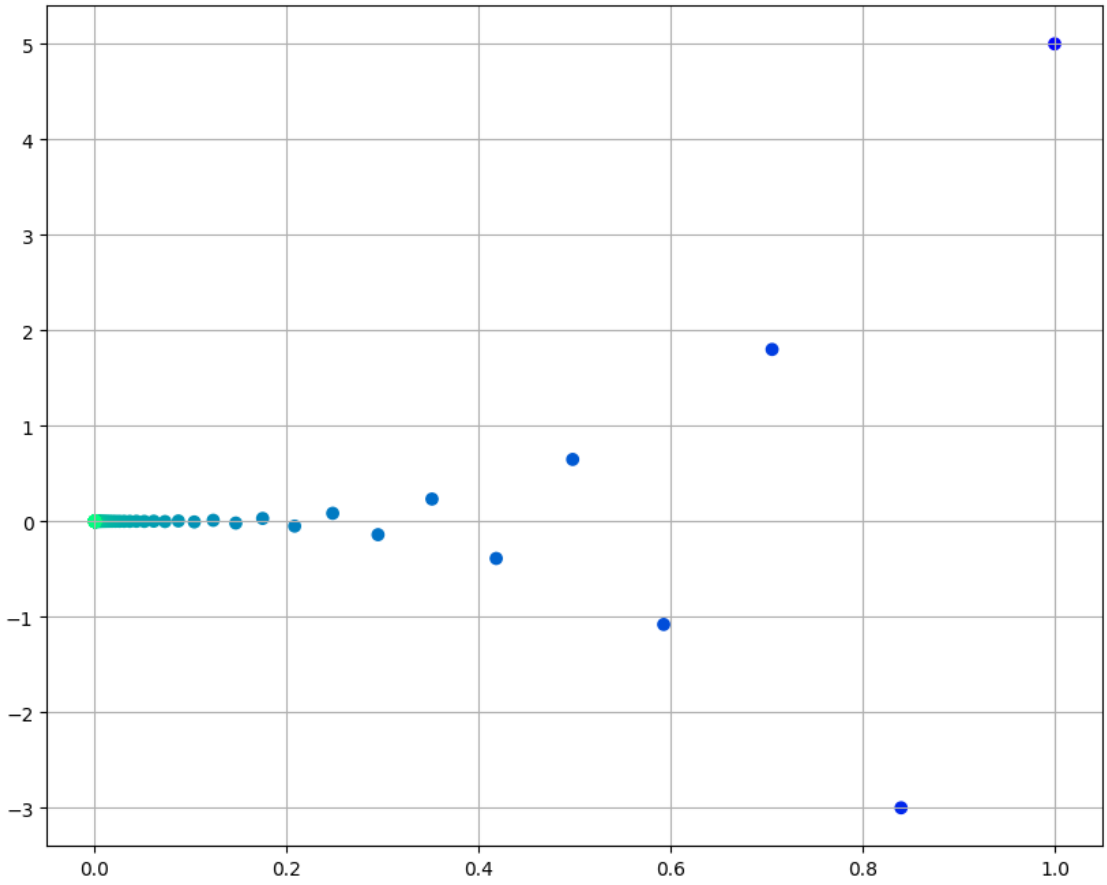
[8]:
```
def plot_grad_desc(x_traj, f_traj, df_traj):
    fig, ax = plt.subplots(figsize=(10,8))

    ax.scatter(
        x_traj[:,0],
        x_traj[:,1],
        c=np.log(np.linspace(1, len(x_traj[:,0]), len(x_traj[:,0]))),
        cmap='winter'
    )

    ax.grid()
    return fig, ax
```

0.1.1 $\lambda = 10$

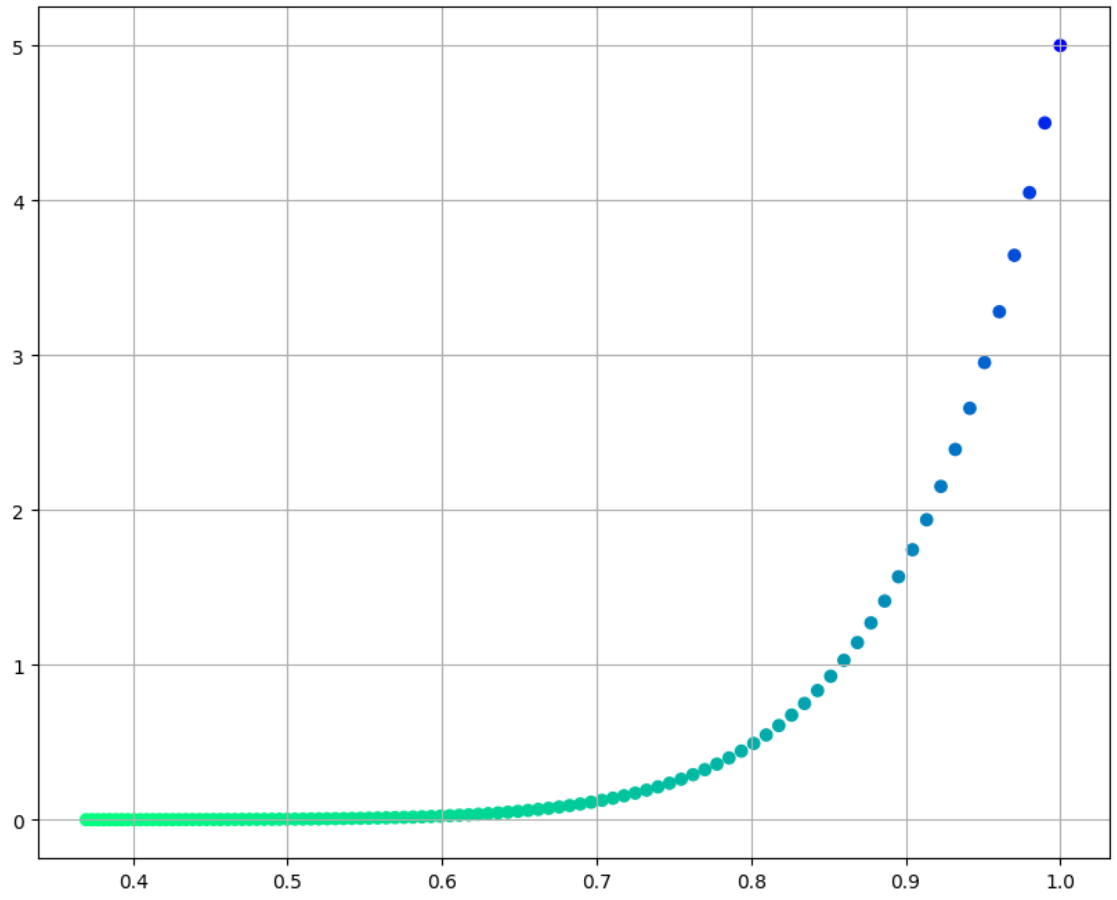
x0 = [1, 5]

- For > 0.1 see significant zig zag that is because is big in the y direction, and it becomes easy to overstep in the y direction
- For <= 0.1, No zig zag for small step size, slow convergence as a result
- With the optimal step size, we avoid the zig zag behavior in this particular caseand also converge much more quickly

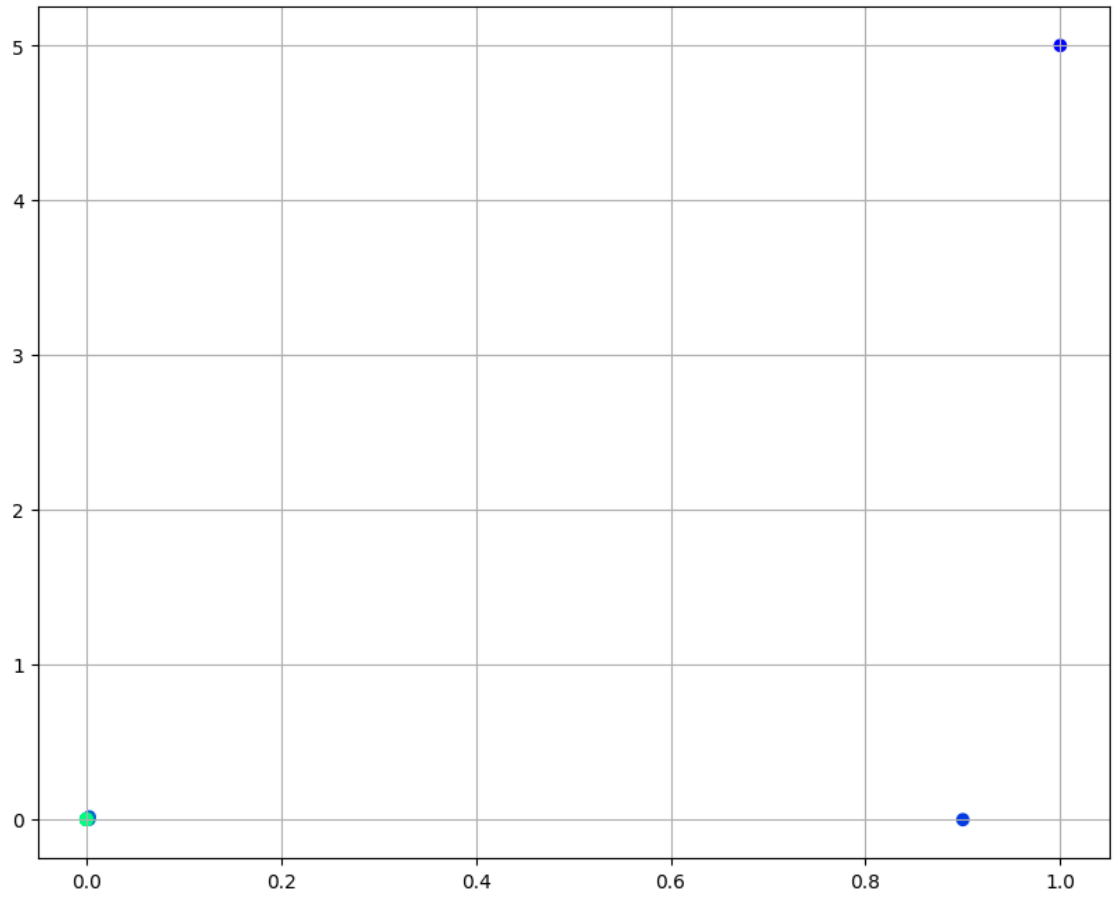
[9]:
```
# For  > 0.1 see significant zig zag
# That is because  is big in the y direction

x_traj, f_traj, df_traj = grad_desc(
     =10,
     =0.16,
    x0=jnp.array([1.,5.]).T,
    n_iter=1e2
)

##### Plotting #####
fig, ax = plot_grad_desc(x_traj, f_traj, df_traj)
```



[10]:
```
# No zig zag for small step size, slow convergence as a result

x_traj, f_traj, df_traj = grad_desc(
     =10,
     =0.01,
    x0=jnp.array([1.,5.]).T,
    n_iter=1e2
)

##### Plotting #####
fig, ax = plot_grad_desc(x_traj, f_traj, df_traj)
```



[11]:
```
# With the optimal step size, we avoid the zig zag behavior
# and also converge much more quickly

x_traj, f_traj, df_traj = grad_desc_optimal_step(
     =10,
    x0=jnp.array([1.,5.]).T,
    n_iter=1e2
)

##### Plotting #####
fig, ax = plot_grad_desc(x_traj, f_traj, df_traj)
```

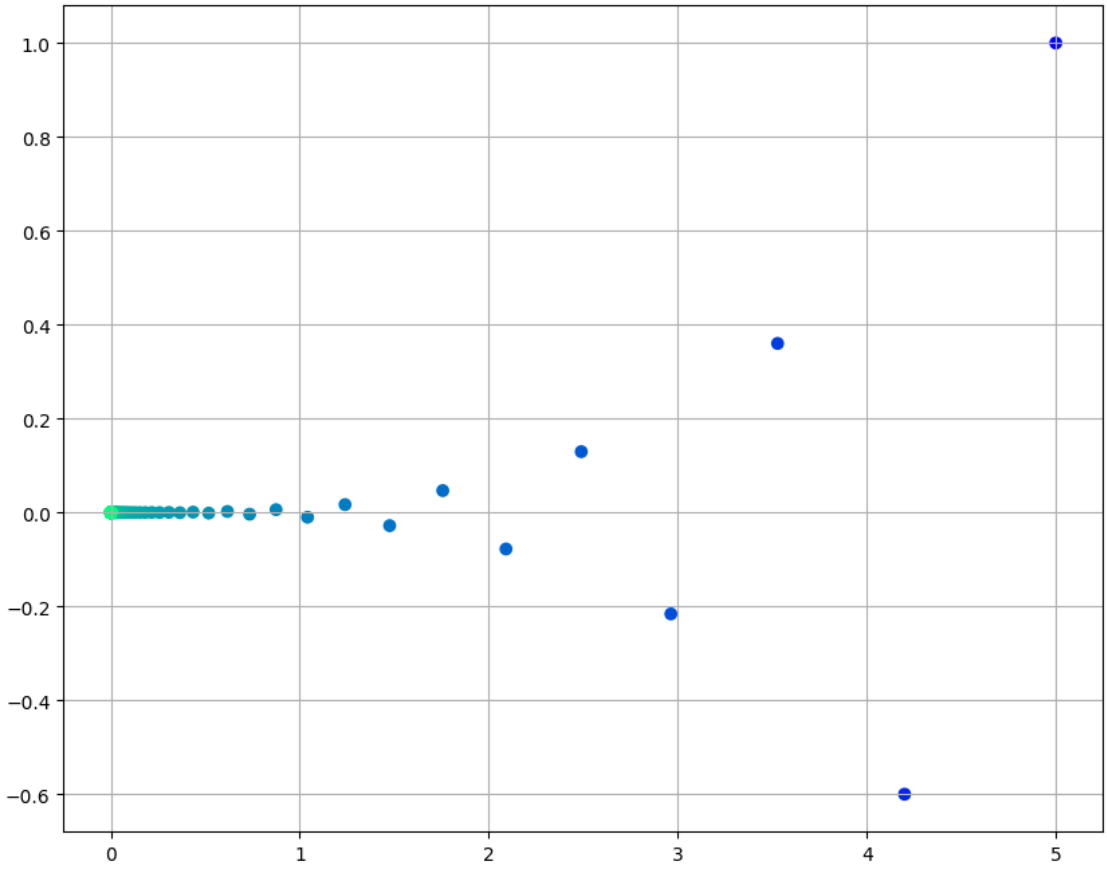


x0 = [5, 1]

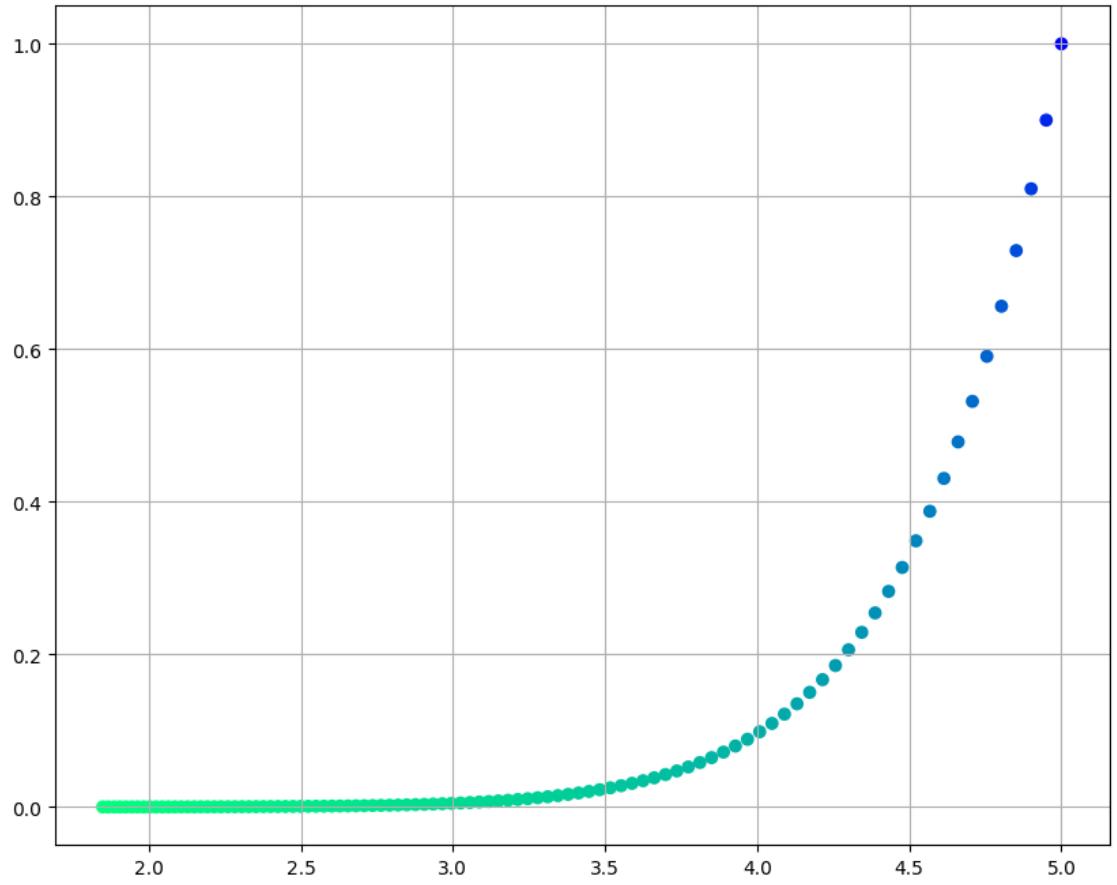
- For constant > 0.1 see significant zig zag that is because is big in the y direction, and it becomes easy to overstep in the y direction
- For constant <= 0.1, No zig zag for small step size, slow convergence as a result
- With the optimal step size, it is interesting that we still observe a zig-zag pattern in the trajectory (why?)

[12]:
```
# For  > 0.1 see significant zig zag
# That is because  is big in the y direction

x_traj, f_traj, df_traj = grad_desc(
    =10,
    =0.16,
    x0=jnp.array([5.,1.]).T,
    n_iter=1e2
)

##### Plotting #####
```

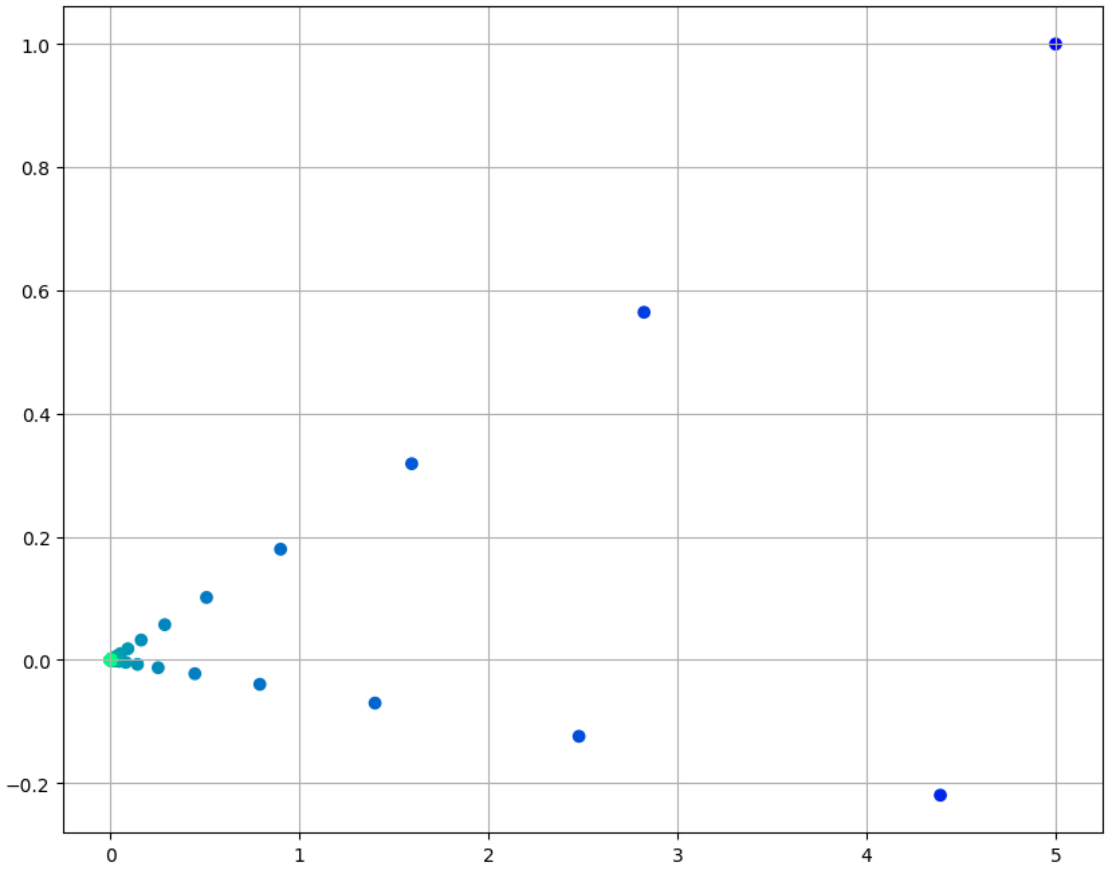
```
fig, ax = plot_grad_desc(x_traj, f_traj, df_traj)
```



[13]:
```
# No zig zag for small step size, slow convergence as a result
x_traj, f_traj, df_traj = grad_desc(
     =10,
     =0.01,
    x0=jnp.array([5.,1.]).T,
    n_iter=1e2
)

##### Plotting #####
fig, ax = plot_grad_desc(x_traj, f_traj, df_traj)
```



[14]:
```
# With the optimal step size, we avoid the zig zag behavior
# and also converge much more quickly

x_traj, f_traj, df_traj = grad_desc_optimal_step(
     =10,
    x0=jnp.array([5., 1.]).T,
    n_iter=1e2
)

##### Plotting #####
fig, ax = plot_grad_desc(x_traj, f_traj, df_traj)
```



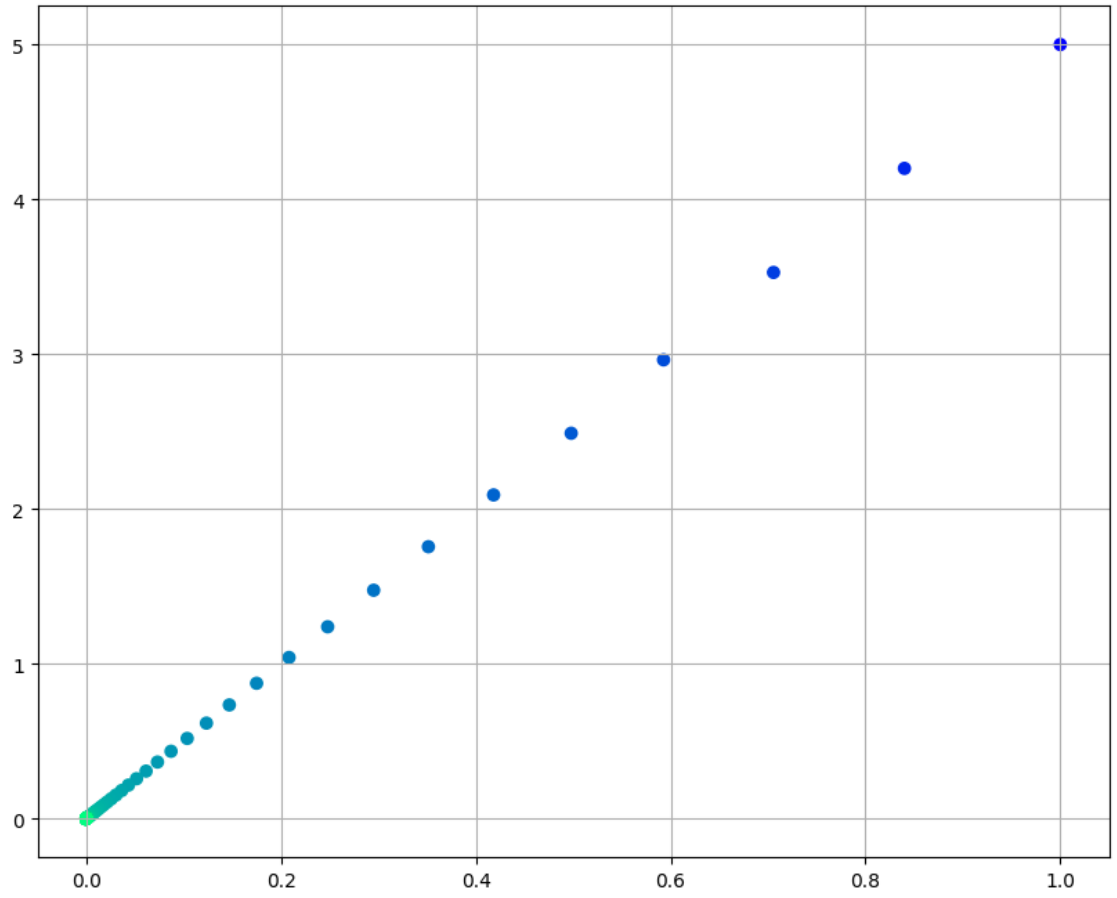
0.1.2 $\lambda = 1$

x0 = [1, 5]

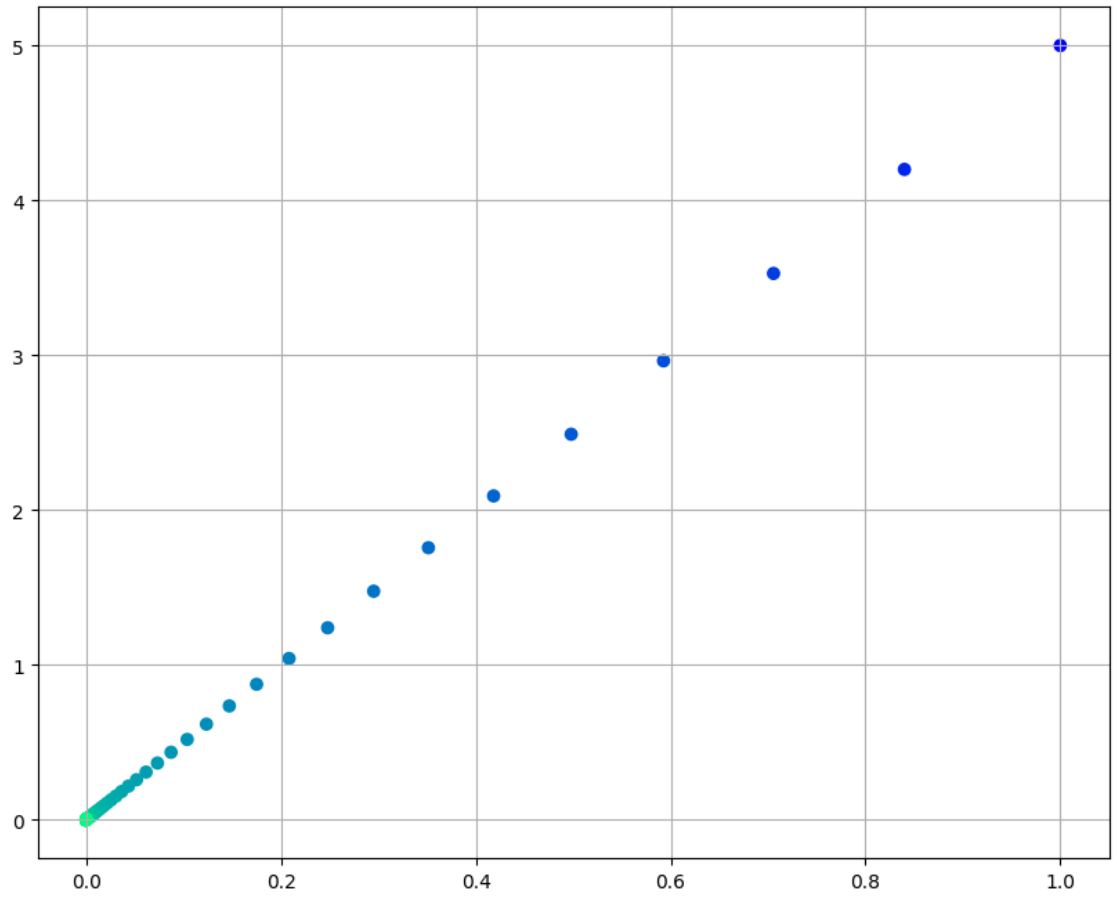
- For constant , we don't see zig zag behaviors across the board. It is due to the fact that f scales equally in x and y directions, such that the gradient descent points to the minimum to begin with.
- With the optimal step size, we approach the minimum in one step.

[15]:
```
# No zig zag, a straight line to zero
# because f scale equally in both x and y direction

x_traj, f_traj, df_traj = grad_desc(
     =1,
     =0.16,
    x0=jnp.array([1.,5.]).T,
    n_iter=1e2
)

##### Plotting #####
```

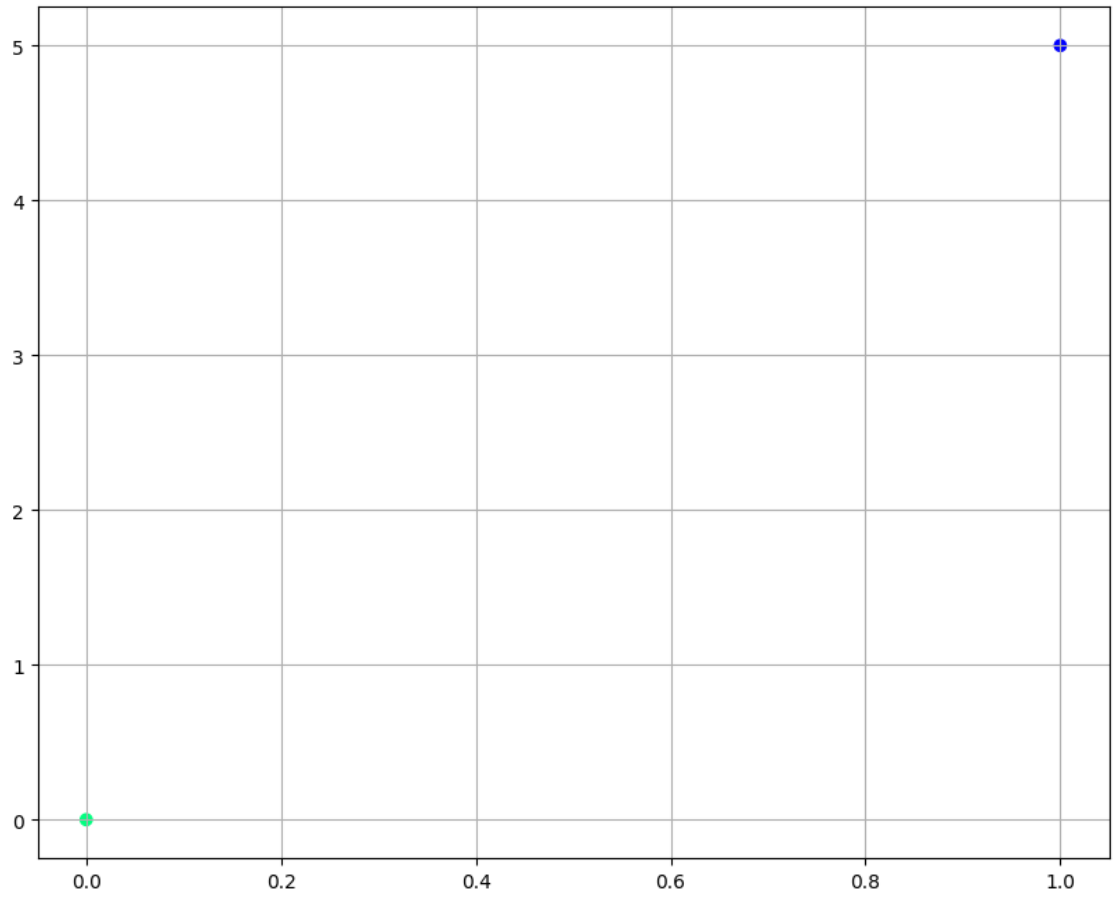
```
fig, ax = plot_grad_desc(x_traj, f_traj, df_traj)
```



[16]:
```
# No zig zag, a straight line to zero
# because f scale equally in both x and y direction

x_traj, f_traj, df_traj = grad_desc(
     =1,
     =0.16,
    x0=jnp.array([1.,5.]).T,
    n_iter=1e2
)

##### Plotting #####
fig, ax = plot_grad_desc(x_traj, f_traj, df_traj)
```



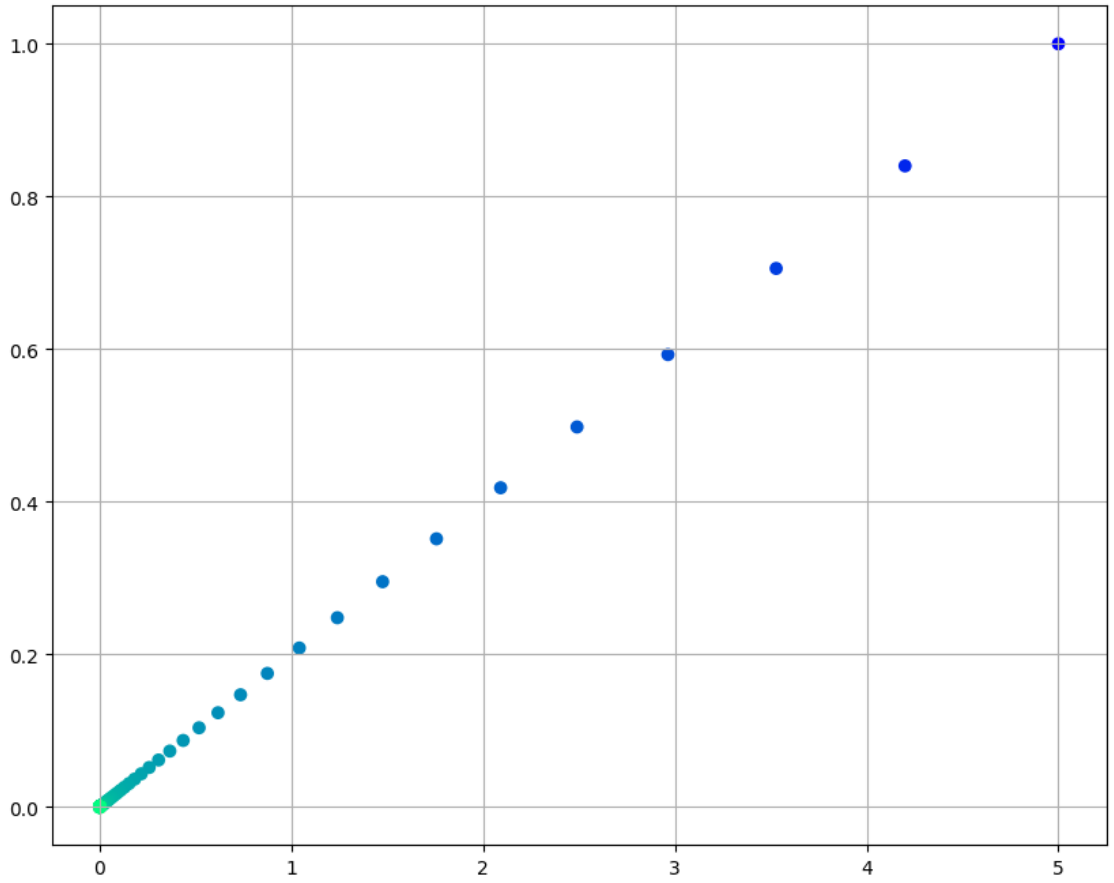
[17]:
```
# With the optimal step size, we avoid the zig zag behavior
# and also converge much more quickly

x_traj, f_traj, df_traj = grad_desc_optimal_step(
     =1,
    x0=jnp.array([1.,5.]).T,
    n_iter=1e2
)

##### Plotting #####
fig, ax = plot_grad_desc(x_traj, f_traj, df_traj)
```



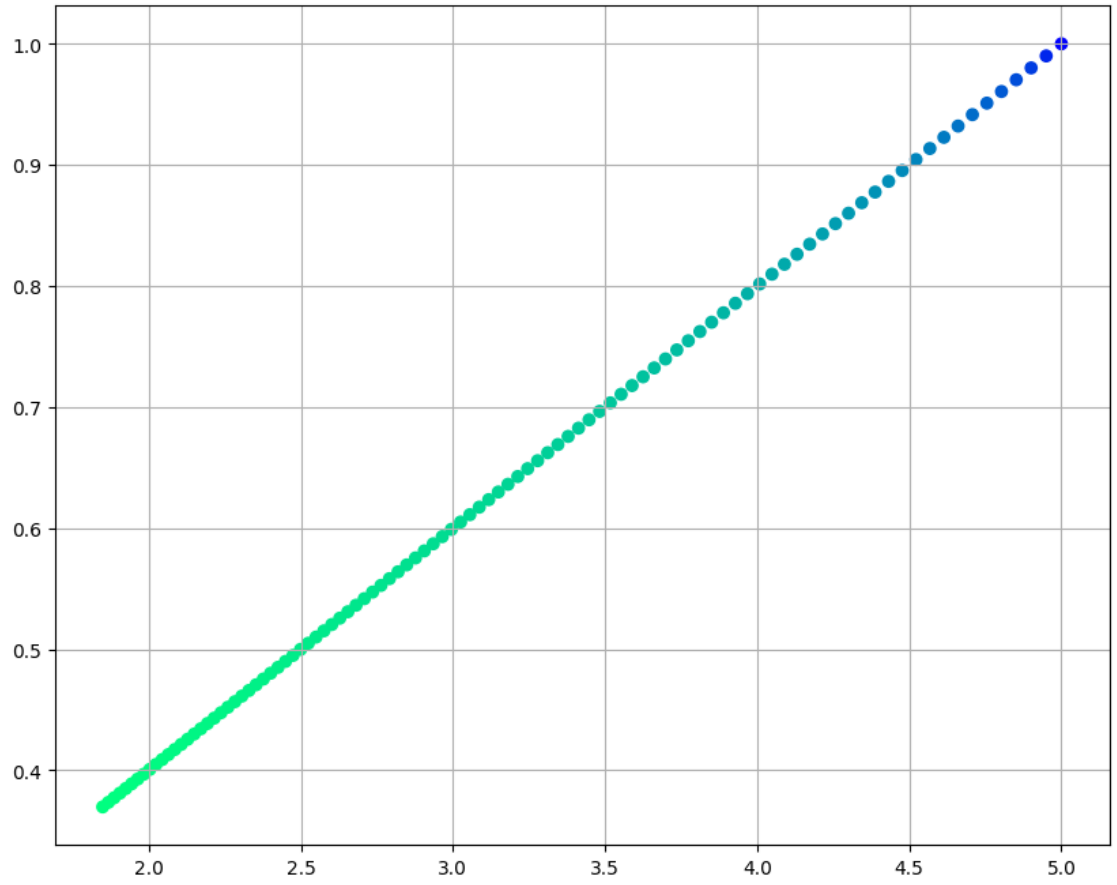
x0 = [5, 1]

- For constant , we don't see zig zag behaviors across the board. It is due to the fact that f scales equally in x and y directions, such that the gradient descent points to the minimum to begin with.
- With the optimal step size, we approach the minimum in one step.

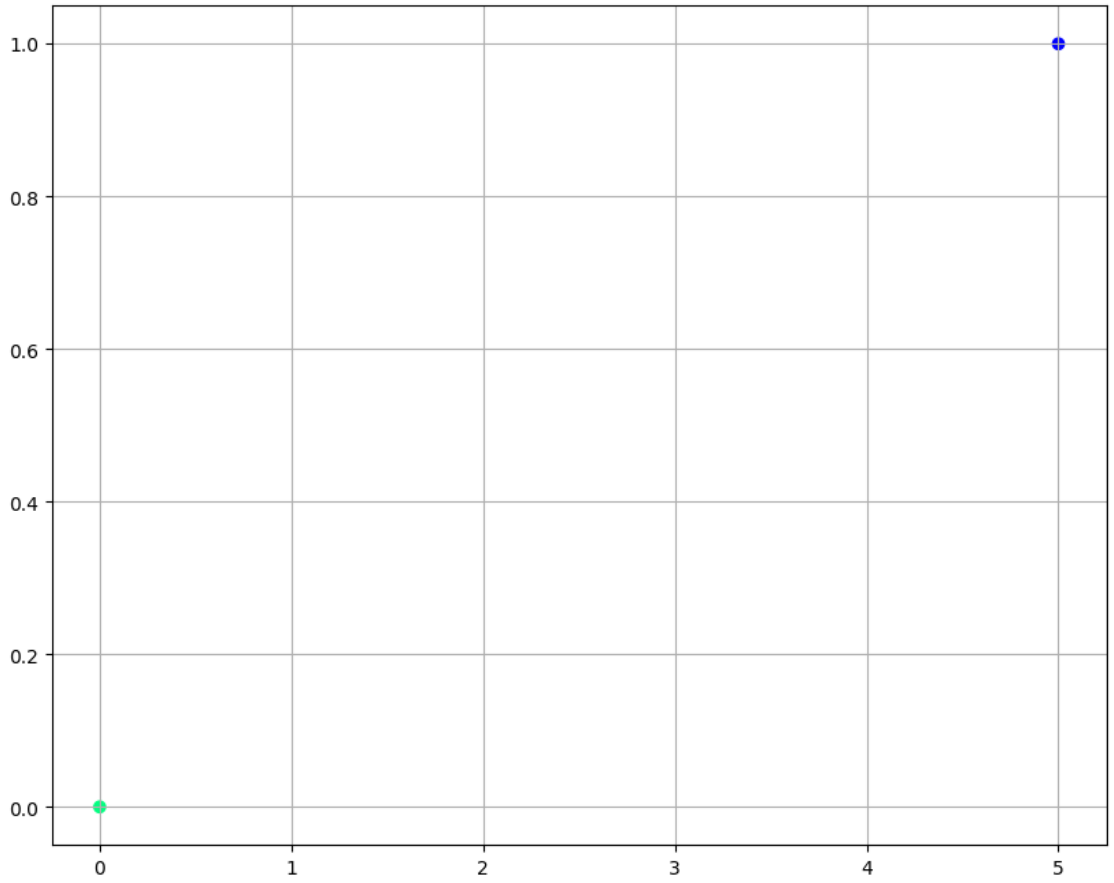
[18]:
```
# For   > 0.1 see significant zig zag
# That is because   is big in the y direction

x_traj, f_traj, df_traj = grad_desc(
     =1,
     =0.16,
    x0=jnp.array([5.,1.]).T,
    n_iter=1e2
)

##### Plotting #####
fig, ax = plot_grad_desc(x_traj, f_traj, df_traj)
```



[19]:
```
# No zig zag for small step size, slow convergence as a result

x_traj, f_traj, df_traj = grad_desc(
     =1,
     =0.01,
    x0=jnp.array([5.,1.]).T,
    n_iter=1e2
)

##### Plotting #####
fig, ax = plot_grad_desc(x_traj, f_traj, df_traj)
```



[20]:
```
# With the optimal step size, we avoid the zig zag behavior
# and also converge much more quickly

x_traj, f_traj, df_traj = grad_desc_optimal_step(
     =1,
    x0=jnp.array([5.,1.]).T,
    n_iter=1e2
)

##### Plotting #####
fig, ax = plot_grad_desc(x_traj, f_traj, df_traj)
```



0.2 Problem 2

[21]:
```
A = np.array([
    [1, 1],
    [0, 1],
])
B = np.array([[0, 1]]).T
x0 = np.array([[1, 0]]).T
T = 20
Q_T = 10 * jnp.eye(2)
Q = 1 * jnp.eye(2)
R = 1 * jnp.eye(1)
```

[22]:
```
A
```

[22]:
```
array([[1, 1],
       [0, 1]])
```

[23]:
```
arr = np.arange(0, T,)
arr_encoded = np.zeros((arr.size, arr.max()+1), dtype=int)
arr_encoded[np.arange(arr.size),arr] = 1.

F = []

# Create a diagonal block matrix whose diagonal entries are R
for row in arr_encoded:
    block_row = []
    for col in row:
        block_row.append(R*col)
    F.append(block_row)

F = jnp.block(F)

F.shape,F
```

[23]:
```
((20, 20),
 Array([[1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
         0., 0., 0., 0.],
        [0., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
         0., 0., 0., 0.],
        [0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
         0., 0., 0., 0.],
        [0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
         0., 0., 0., 0.],
        [0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
         0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
         0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
         0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0.,
         0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0.,
         0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0.,
         0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0.,
         0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0.,
         0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0.,
         0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0.,
         0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 0.,
```

```
        0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1.,
        0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        1., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 1., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 1., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 1.]], dtype=float32))
```

[24]:
```
arr = np.arange(0, T+1,)
arr_encoded = np.zeros((arr.size, arr.max()+1), dtype=int)
arr_encoded[np.arange(arr.size),arr] = 1.
E = []

# Create a diagonal block matrix with all Q
for row in arr_encoded:
    block_row = []
    for col in row:
        block_row.append(Q*col)
    E.append(block_row)

# Replace the last diagonal with Q_T
block_row = []
for col in arr_encoded[-1]:
    block_row.append(Q_T*col)
E[-1] = block_row
E = jnp.block(E)

E.shape, E
```

[24]:
```
((42, 42),
 Array([[ 1.,  0.,  0., …,  0.,  0.,  0.],
        [ 0.,  1.,  0., …,  0.,  0.,  0.],
        [ 0.,  0.,  1., …,  0.,  0.,  0.],
        …,
        [ 0.,  0.,  0., …,  1.,  0.,  0.],
        [ 0.,  0.,  0., …,  0., 10.,  0.],
        [ 0.,  0.,  0., …,  0.,  0., 10.]], dtype=float32))
```

[25]:
```
D = []

# Block Matrix with T+1 Block Rows and T Block Columns
for row in np.arange(0,T+1):
    block_row = []
```

```
    for col in np.arange(0,T):
        if row>col:
            block_row.append(np.linalg.matrix_power(A,int(row-col-1))@B*1)
        else:
            # Make sure row_num==col_num and everything above is all zero
            block_row.append(B*0)
    D.append(block_row)
D = jnp.block(D)
D.shape, D
```

[25]:
```
((42, 20),
 Array([[ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
          0,  0,  0,  0],
        [ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
          0,  0,  0,  0],
        [ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
          0,  0,  0,  0],
        [ 1,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
          0,  0,  0,  0],
        [ 1,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
          0,  0,  0,  0],
        [ 1,  1,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
          0,  0,  0,  0],
        [ 2,  1,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
          0,  0,  0,  0],
        [ 1,  1,  1,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
          0,  0,  0,  0],
        [ 3,  2,  1,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
          0,  0,  0,  0],
        [ 1,  1,  1,  1,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
          0,  0,  0,  0],
        [ 4,  3,  2,  1,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
          0,  0,  0,  0],
        [ 1,  1,  1,  1,  1,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
          0,  0,  0,  0],
        [ 5,  4,  3,  2,  1,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
          0,  0,  0,  0],
        [ 1,  1,  1,  1,  1,  1,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
          0,  0,  0,  0],
        [ 6,  5,  4,  3,  2,  1,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
          0,  0,  0,  0],
        [ 1,  1,  1,  1,  1,  1,  1,  0,  0,  0,  0,  0,  0,  0,  0,  0,
          0,  0,  0,  0],
        [ 7,  6,  5,  4,  3,  2,  1,  0,  0,  0,  0,  0,  0,  0,  0,  0,
          0,  0,  0,  0],
        [ 1,  1,  1,  1,  1,  1,  1,  1,  0,  0,  0,  0,  0,  0,  0,  0,
          0,  0,  0,  0],
```

```
[ 8,  7,  6,  5,  4,  3,  2,  1,  0,  0,  0,  0,  0,  0,  0,  0,
  0,  0,  0,  0],
[ 1,  1,  1,  1,  1,  1,  1,  1,  1,  0,  0,  0,  0,  0,  0,  0,
  0,  0,  0,  0],
[ 9,  8,  7,  6,  5,  4,  3,  2,  1,  0,  0,  0,  0,  0,  0,  0,
  0,  0,  0,  0],
[ 1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  0,  0,  0,  0,  0,  0,
  0,  0,  0,  0],
[10,  9,  8,  7,  6,  5,  4,  3,  2,  1,  0,  0,  0,  0,  0,  0,
  0,  0,  0,  0],
[ 1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  0,  0,  0,  0,  0,
  0,  0,  0,  0],
[11, 10,  9,  8,  7,  6,  5,  4,  3,  2,  1,  0,  0,  0,  0,  0,
  0,  0,  0,  0],
[ 1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  0,  0,  0,  0,
  0,  0,  0,  0],
[12, 11, 10,  9,  8,  7,  6,  5,  4,  3,  2,  1,  0,  0,  0,  0,
  0,  0,  0,  0],
[ 1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  0,  0,  0,
  0,  0,  0,  0],
[13, 12, 11, 10,  9,  8,  7,  6,  5,  4,  3,  2,  1,  0,  0,  0,
  0,  0,  0,  0],
[ 1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  0,  0,
  0,  0,  0,  0],
[14, 13, 12, 11, 10,  9,  8,  7,  6,  5,  4,  3,  2,  1,  0,  0,
  0,  0,  0,  0],
[ 1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  0,
  0,  0,  0,  0],
[15, 14, 13, 12, 11, 10,  9,  8,  7,  6,  5,  4,  3,  2,  1,  0,
  0,  0,  0,  0],
[ 1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
  0,  0,  0,  0],
[16, 15, 14, 13, 12, 11, 10,  9,  8,  7,  6,  5,  4,  3,  2,  1,
  0,  0,  0,  0],
[ 1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
  1,  0,  0,  0],
[17, 16, 15, 14, 13, 12, 11, 10,  9,  8,  7,  6,  5,  4,  3,  2,
  1,  0,  0,  0],
[ 1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
  1,  1,  0,  0],
[18, 17, 16, 15, 14, 13, 12, 11, 10,  9,  8,  7,  6,  5,  4,  3,
  2,  1,  0,  0],
[ 1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
  1,  1,  1,  0],
[19, 18, 17, 16, 15, 14, 13, 12, 11, 10,  9,  8,  7,  6,  5,  4,
  3,  2,  1,  0],
[ 1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
```

```
       1,  1,  1,  1]], dtype=int32))
```

[26]:
```
# Create block matrix C, that is a column stack of A raise to sequential powers
C = np.block([[np.eye(2)@np.linalg.matrix_power(A,i)] for i in range(T+1)])

C
```

[26]:
```
array([[ 1.,  0.],
       [ 0.,  1.],
       [ 1.,  1.],
       [ 0.,  1.],
       [ 1.,  2.],
       [ 0.,  1.],
       [ 1.,  3.],
       [ 0.,  1.],
       [ 1.,  4.],
       [ 0.,  1.],
       [ 1.,  5.],
       [ 0.,  1.],
       [ 1.,  6.],
       [ 0.,  1.],
       [ 1.,  7.],
       [ 0.,  1.],
       [ 1.,  8.],
       [ 0.,  1.],
       [ 1.,  9.],
       [ 0.,  1.],
       [ 1., 10.],
       [ 0.,  1.],
       [ 1., 11.],
       [ 0.,  1.],
       [ 1., 12.],
       [ 0.,  1.],
       [ 1., 13.],
       [ 0.,  1.],
       [ 1., 14.],
       [ 0.,  1.],
       [ 1., 15.],
       [ 0.,  1.],
       [ 1., 16.],
       [ 0.,  1.],
       [ 1., 17.],
       [ 0.,  1.],
       [ 1., 18.],
       [ 0.,  1.],
       [ 1., 19.],
       [ 0.,  1.],
```

```
        [ 1., 20.],
        [ 0.,  1.]])
```

[27]:
```
# Calculate b based on definition
b_ = -2*D.T@E@C@x0
b_
```

[27]:
```
Array([[-722.],
       [-666.],
       [-612.],
       [-560.],
       [-510.],
       [-462.],
       [-416.],
       [-372.],
       [-330.],
       [-290.],
       [-252.],
       [-216.],
       [-182.],
       [-150.],
       [-120.],
       [ -92.],
       [ -66.],
       [ -42.],
       [ -20.],
       [   0.]], dtype=float32)
```

[28]:
```
# Calculate Q based on definition
Q_ = F + D.T@E@D
Q_
```

[28]:
```
Array([[5749., 5386., 5025., 4666., 4310., 3958., 3611., 3270., 2936.,
        2610., 2293., 1986., 1690., 1406., 1135.,  878.,  636.,  410.,
         201.,   10.],
       [5386., 5054., 4719., 4386., 4055., 3727., 3403., 3084., 2771.,
        2465., 2167., 1878., 1599., 1331., 1075.,  832.,  603.,  389.,
         191.,   10.],
       [5025., 4719., 4414., 4106., 3800., 3496., 3195., 2898., 2606.,
        2320., 2041., 1770., 1508., 1256., 1015.,  786.,  570.,  368.,
         181.,   10.],
       [4666., 4386., 4106., 3827., 3545., 3265., 2987., 2712., 2441.,
        2175., 1915., 1662., 1417., 1181.,  955.,  740.,  537.,  347.,
         171.,   10.],
       [4310., 4055., 3800., 3545., 3291., 3034., 2779., 2526., 2276.,
        2030., 1789., 1554., 1326., 1106.,  895.,  694.,  504.,  326.,
         161.,   10.],
```

```
[3958., 3727., 3496., 3265., 3034., 2804., 2571., 2340., 2111.,
 1885., 1663., 1446., 1235., 1031.,  835.,  648.,  471.,  305.,
  151.,   10.],
[3611., 3403., 3195., 2987., 2779., 2571., 2364., 2154., 1946.,
 1740., 1537., 1338., 1144.,  956.,  775.,  602.,  438.,  284.,
  141.,   10.],
[3270., 3084., 2898., 2712., 2526., 2340., 2154., 1969., 1781.,
 1595., 1411., 1230., 1053.,  881.,  715.,  556.,  405.,  263.,
  131.,   10.],
[2936., 2771., 2606., 2441., 2276., 2111., 1946., 1781., 1617.,
 1450., 1285., 1122.,  962.,  806.,  655.,  510.,  372.,  242.,
  121.,   10.],
[2610., 2465., 2320., 2175., 2030., 1885., 1740., 1595., 1450.,
 1306., 1159., 1014.,  871.,  731.,  595.,  464.,  339.,  221.,
  111.,   10.],
[2293., 2167., 2041., 1915., 1789., 1663., 1537., 1411., 1285.,
 1159., 1034.,  906.,  780.,  656.,  535.,  418.,  306.,  200.,
  101.,   10.],
[1986., 1878., 1770., 1662., 1554., 1446., 1338., 1230., 1122.,
 1014.,  906.,  799.,  689.,  581.,  475.,  372.,  273.,  179.,
   91.,   10.],
[1690., 1599., 1508., 1417., 1326., 1235., 1144., 1053.,  962.,
  871.,  780.,  689.,  599.,  506.,  415.,  326.,  240.,  158.,
   81.,   10.],
[1406., 1331., 1256., 1181., 1106., 1031.,  956.,  881.,  806.,
  731.,  656.,  581.,  506.,  432.,  355.,  280.,  207.,  137.,
   71.,   10.],
[1135., 1075., 1015.,  955.,  895.,  835.,  775.,  715.,  655.,
  595.,  535.,  475.,  415.,  355.,  296.,  234.,  174.,  116.,
   61.,   10.],
[ 878.,  832.,  786.,  740.,  694.,  648.,  602.,  556.,  510.,
  464.,  418.,  372.,  326.,  280.,  234.,  189.,  141.,   95.,
   51.,   10.],
[ 636.,  603.,  570.,  537.,  504.,  471.,  438.,  405.,  372.,
  339.,  306.,  273.,  240.,  207.,  174.,  141.,  109.,   74.,
   41.,   10.],
[ 410.,  389.,  368.,  347.,  326.,  305.,  284.,  263.,  242.,
  221.,  200.,  179.,  158.,  137.,  116.,   95.,   74.,   54.,
   31.,   10.],
[ 201.,  191.,  181.,  171.,  161.,  151.,  141.,  131.,  121.,
  111.,  101.,   91.,   81.,   71.,   61.,   51.,   41.,   31.,
   22.,   10.],
[  10.,   10.,   10.,   10.,   10.,   10.,   10.,   10.,   10.,
   10.,   10.,   10.,   10.,   10.,   10.,   10.,   10.,   10.,
   10.,   11.]], dtype=float32)
```

[29]:
```
F.shape
```

[29]: (20, 20)

[30]:
```
D.shape
```

[30]: (42, 20)

[31]:
```
E.shape
```

[31]: (42, 42)

[32]:
```
g = lambda u: (1/2*u.T@Q_@u - b_.T@u)[0,0]

dg = jax.grad(g)

dg(np.ones((20,1)))
```

[32]:
```
Array([[52888.],
       [49791.],
       [46696.],
       [43605.],
       [40521.],
       [37448.],
       [34391.],
       [31356.],
       [28350.],
       [25381.],
       [22458.],
       [19591.],
       [16791.],
       [14070.],
       [11441.],
       [ 8918.],
       [ 6516.],
       [ 4251.],
       [ 2140.],
       [  201.]], dtype=float32)
```

[33]:
```
def grad_desc_optimal_step( , x0, n_iter=1e2):
    # Define Q
    Q = np.array([
        [1, 0],
        [0,  ]
    ])


    f = lambda x: 1/2 * x.T @ Q @ x
```

```
    # Take gradient with Jax
    df_dx = jax.grad(f)

    # Initiate variables for logging
    x_traj = []
    f_traj = []
    df_traj = []
    x_next = x0

    f_best = np.inf
    x_best = None

    for _ in range(int(n_iter)):
        # Log x, f, df
        x_traj.append(x_next)

        f_next = f(x_next)
        f_traj.append(f_next)

        df_next = df_dx(x_next)
        df_traj.append(df_next)

        # Record the x_best and f_best by comparing with the previous best f
        if f_best>f_next:
            f_best = f_next
            x_best = x_next

        # Calculate the optimal step based on the derived formula
        d = -df_next
          = d.T@d / (d.T@Q@d)

        # Perform gradient descent
        x_next = x_next -   * df_dx(x_next)

    x_traj = jnp.array(x_traj)
    f_traj = jnp.array(x_traj)
    df_traj = jnp.array(x_traj)

    return x_traj, f_traj, df_traj
```

[34]:
```
def grad_desc_optimal_step(u0, max_iter=1e4, thres_error=1e-5):
    # Define g
    g = lambda u: (1/2*u.T@Q_@u - b_.T@u)[0,0]

    # Take gradient with Jax
    dg = jax.grad(g)
```

```
    # Initiate variables for logging
    u_traj = []
    g_traj = []
    dg_traj = []
    u_next = u0

    g_best = np.inf
    u_best = None

    for _ in range(int(max_iter)):
        # Log u, g, dg
        u_traj.append(u_next)

        g_next = g(u_next)
        g_traj.append(g_next)

        if abs(g_next-g_best) < thres_error:
            break

        dg_next = dg(u_next)
        dg_traj.append(dg_next)

        # Record the u_best and g_best by comparing with the previous best g
        if g_best>g_next:
            g_best = g_next
            u_best = u_next

        d = -dg_next
          = d.T@d / (d.T@Q_@d)

        # Perform gradient descent
        u_next = u_next -   * dg_next

    u_traj = jnp.array(u_traj)
    g_traj = jnp.array(u_traj)
    dg_traj = jnp.array(u_traj)

    return u_traj, g_traj, dg_traj, u_best, g_best
```

[35]:
```
u_traj, g_traj, dg_traj, u_best, g_best = grad_desc_optimal_step(np.
↪ones((20,1)))
```

[36]:
```
u_best, g_best
```

```
[36]: (Array([[-0.6459    ],
              [-0.01319054],
              [ 0.2105224 ],
              [ 0.22274862],
              [ 0.15479854],
              [ 0.07829656],
              [ 0.02269153],
              [-0.00692374],
              [-0.01626588],
              [-0.01354499],
              [-0.00559433],
              [ 0.00302509],
              [ 0.00942469],
              [ 0.01150498],
              [ 0.00768338],
              [-0.00224895],
              [-0.01512608],
              [-0.02204243],
              [-0.00752994],
              [ 0.04628965]], dtype=float32),
       Array(-53.97545, dtype=float32))
```