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)
```

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

    return x_traj, f_traj, df_traj
```

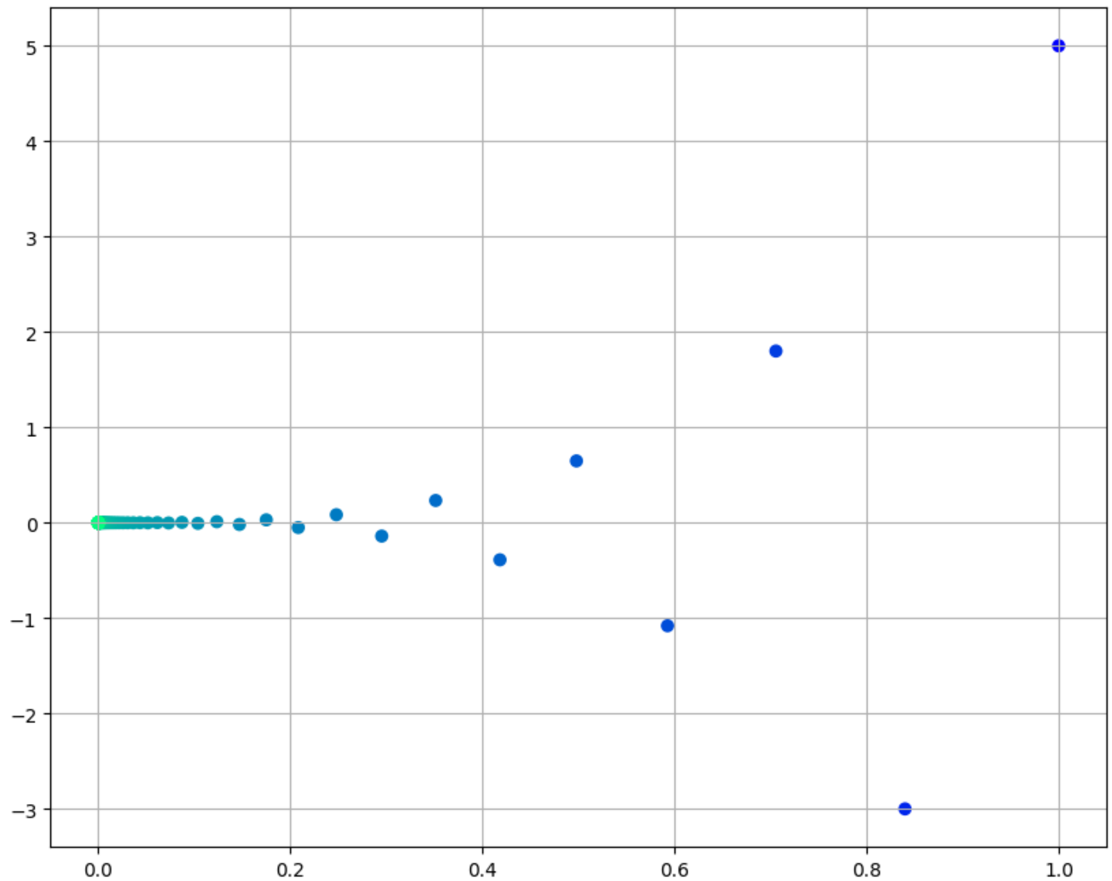
[8]:
```python
def plot_grad_desc(x_traj, f_traj, df_traj):
    fig, ax = plt.subplots(figsize=(10,8))

    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 $\lambda = 10$

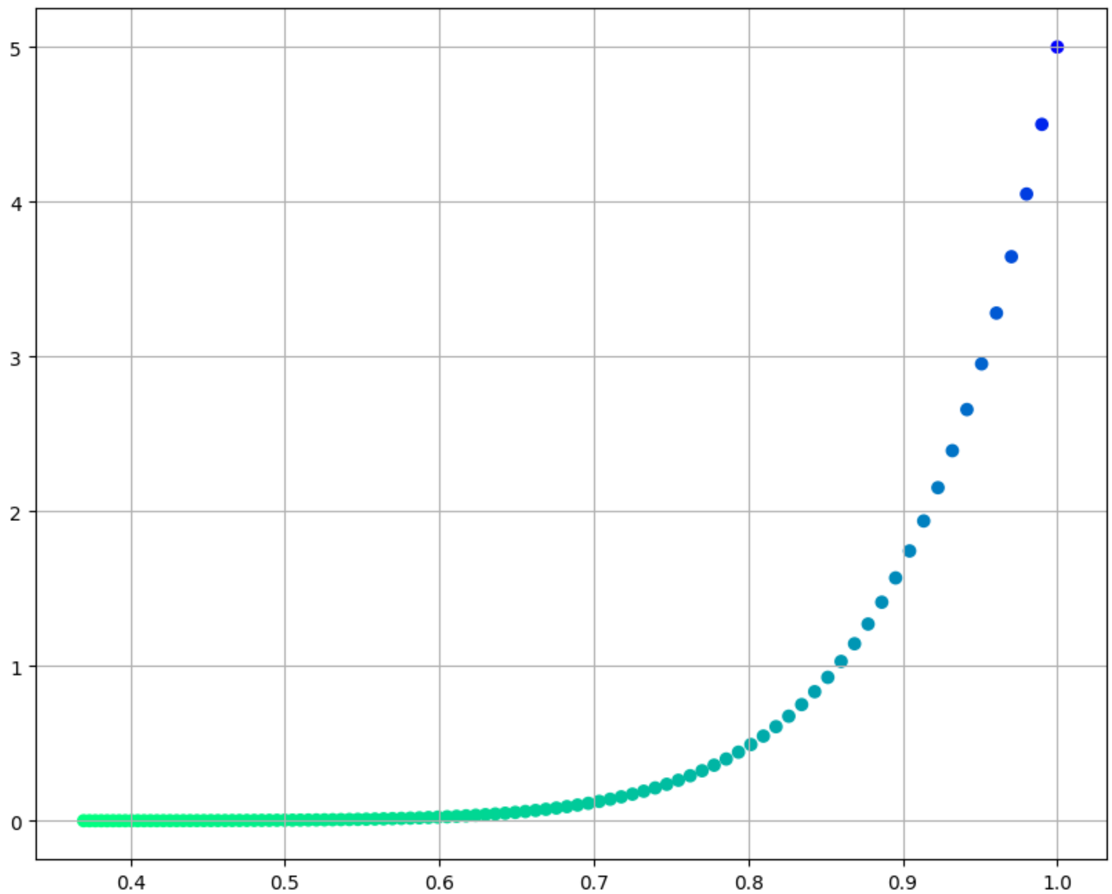
x0 = [1, 5]

- 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

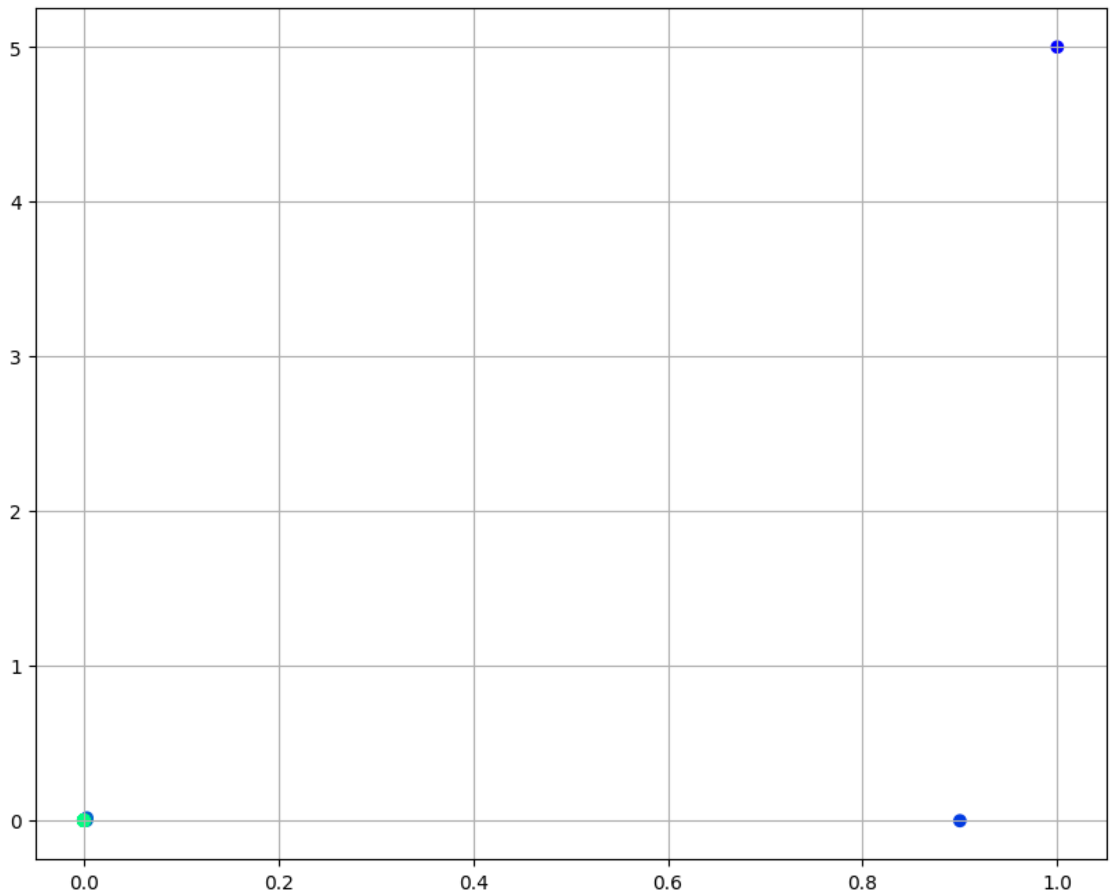
[9]:
```python
# For   > 0.1 see significant zig zag
# That is because   is big in the y direction

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)
```

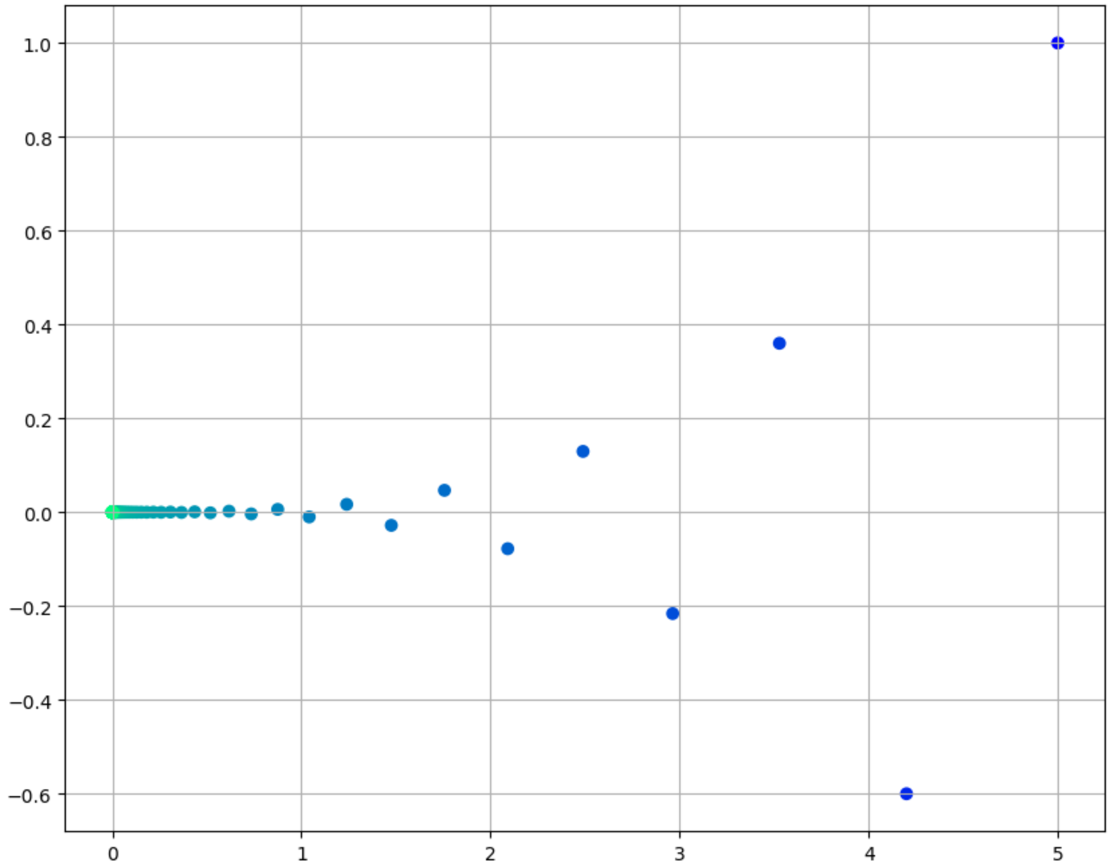


x0 = [5, 1]

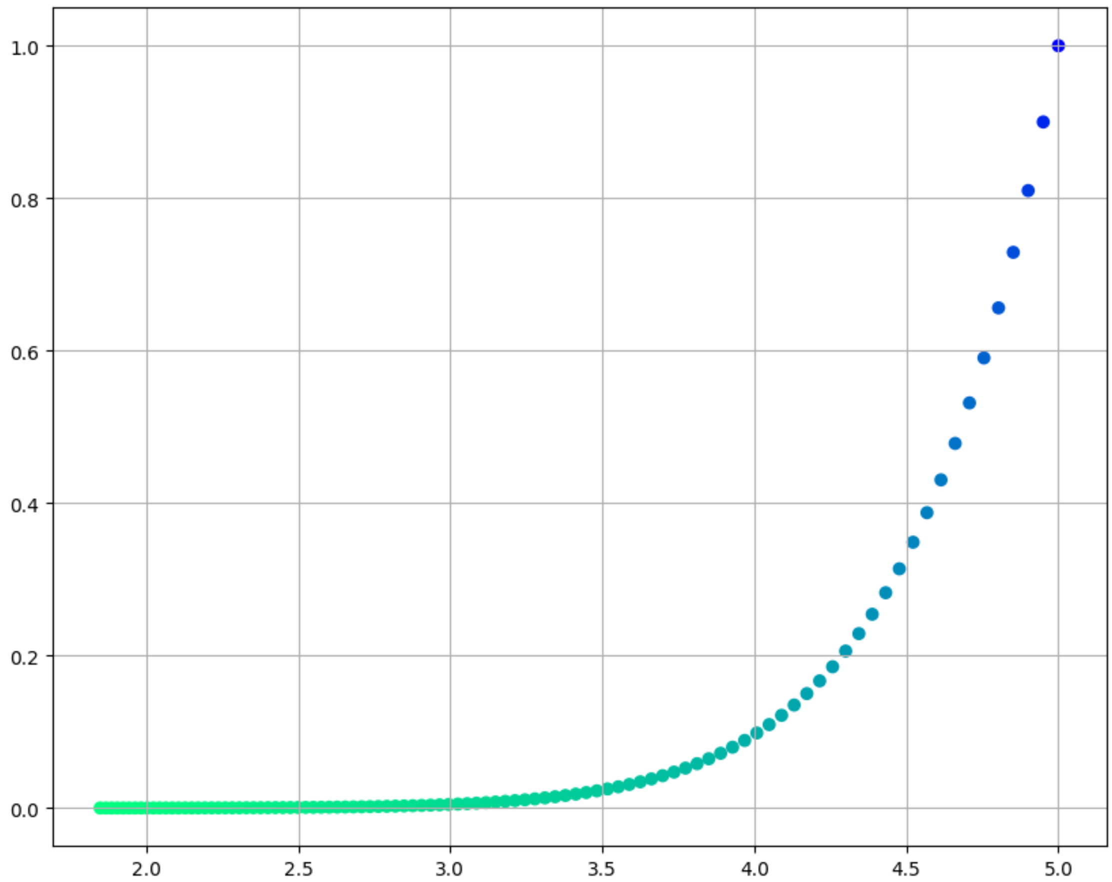
- For constant > 0.1 see significant zig zag that is because is big in the y direction, and it becomes easy to overstep in the y direction
- For constant <= 0.1, No zig zag for small step size, slow convergence as a result
- With the optimal step size, it is interesting that we still observe a zig-zag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

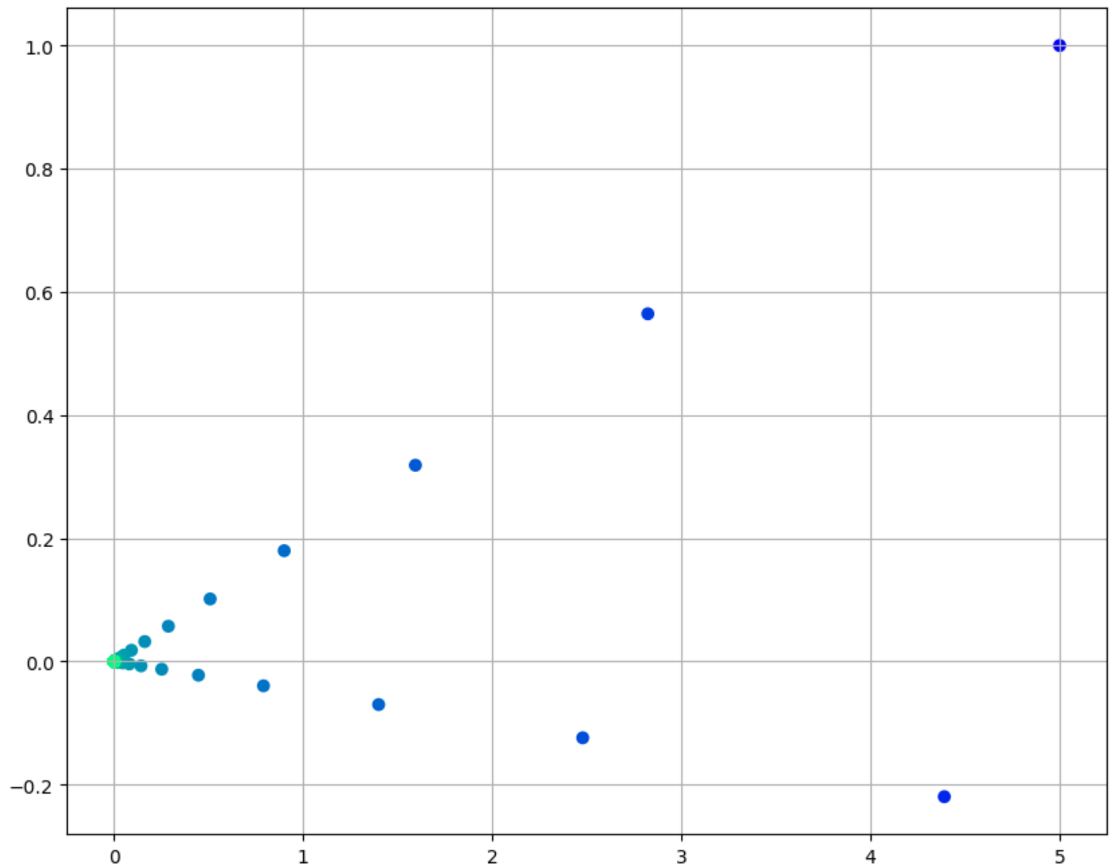
```
fig, ax = plot_grad_desc(x_traj, f_traj, df_traj)
```



[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)
```



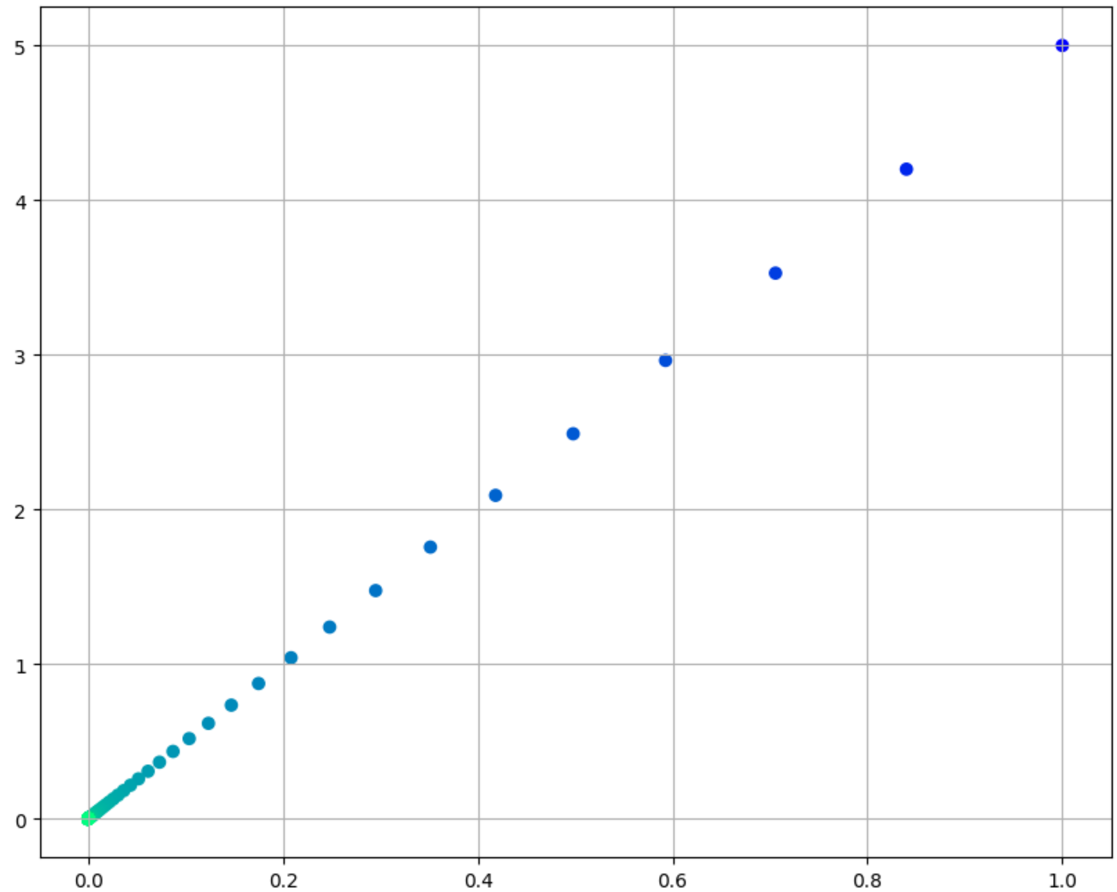
0.1.2 $\lambda = 1$

x0 = [1, 5]

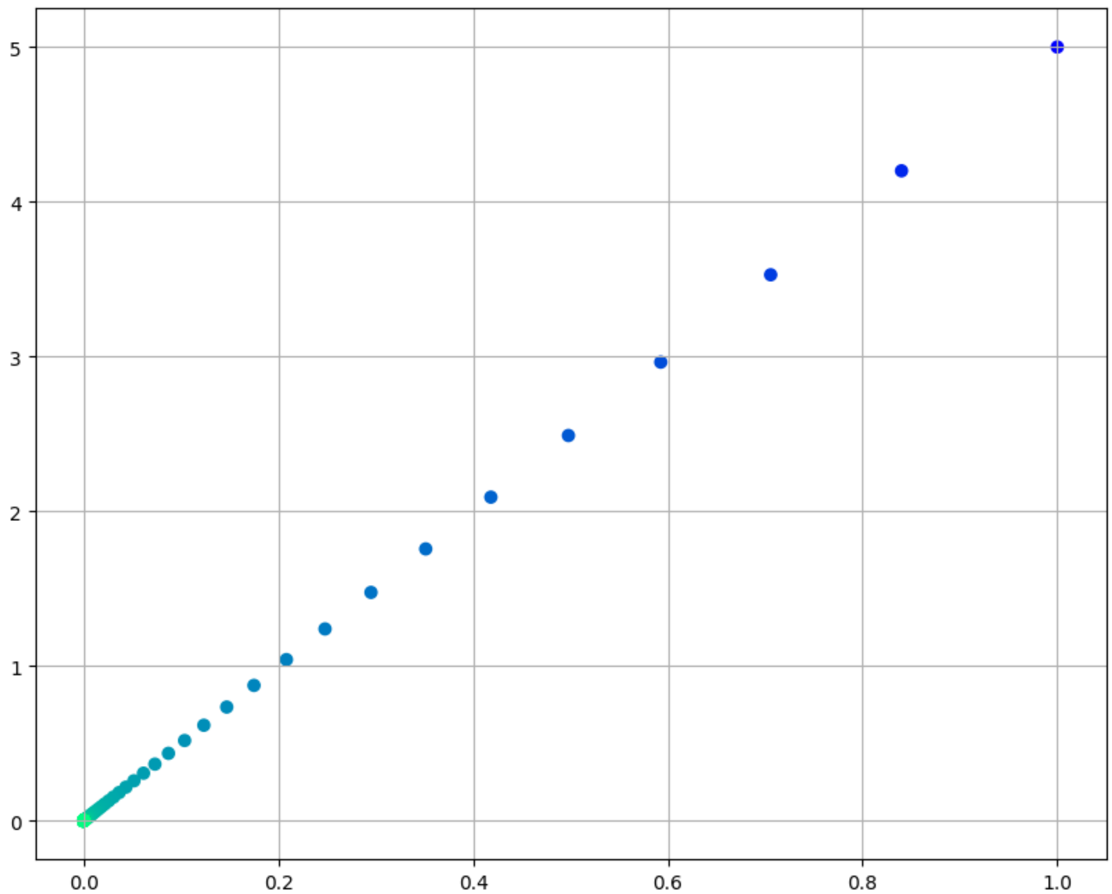
- For constant , we don't see zig zag behaviors across the board. It is due to the fact that f scales equally in x and y directions, such that the gradient descent points to the minimum to begin with.
- With the optimal step size, we approach the minimum in one step.

[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 #####
```

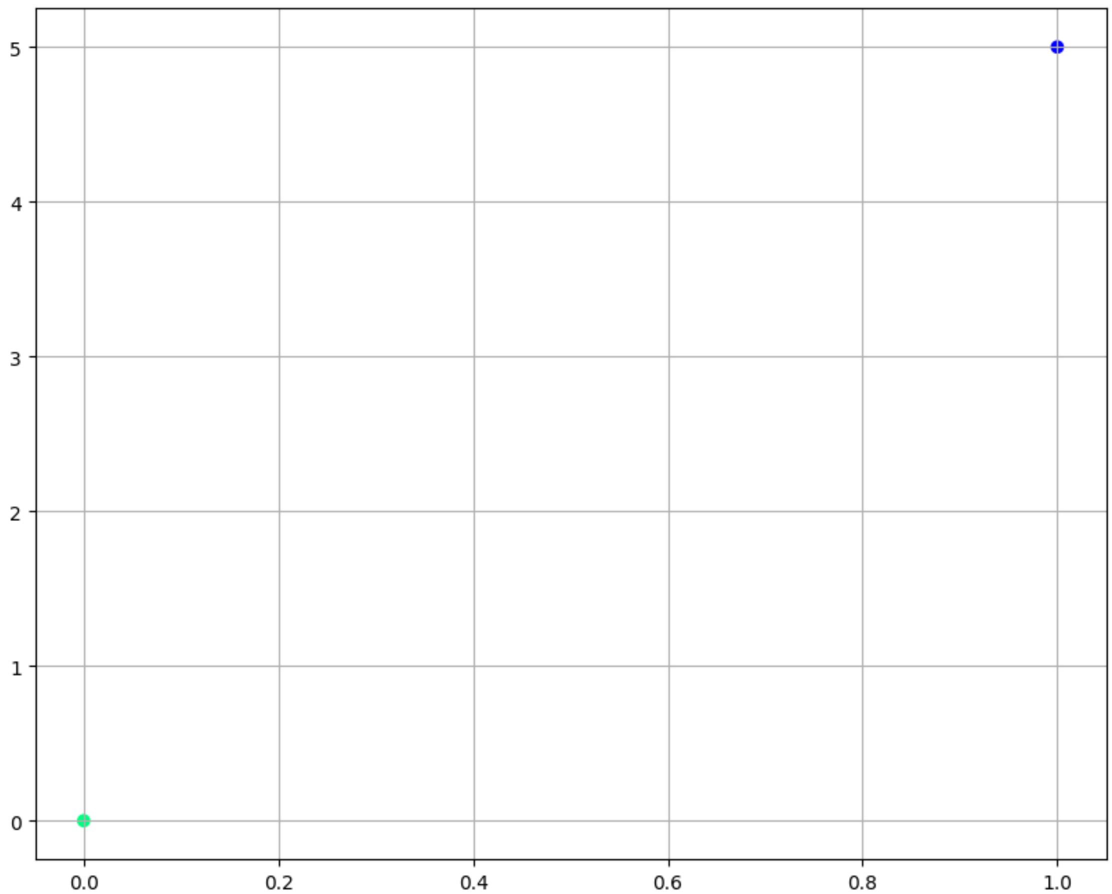
```
fig, ax = plot_grad_desc(x_traj, f_traj, df_traj)
```




```
[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)
```



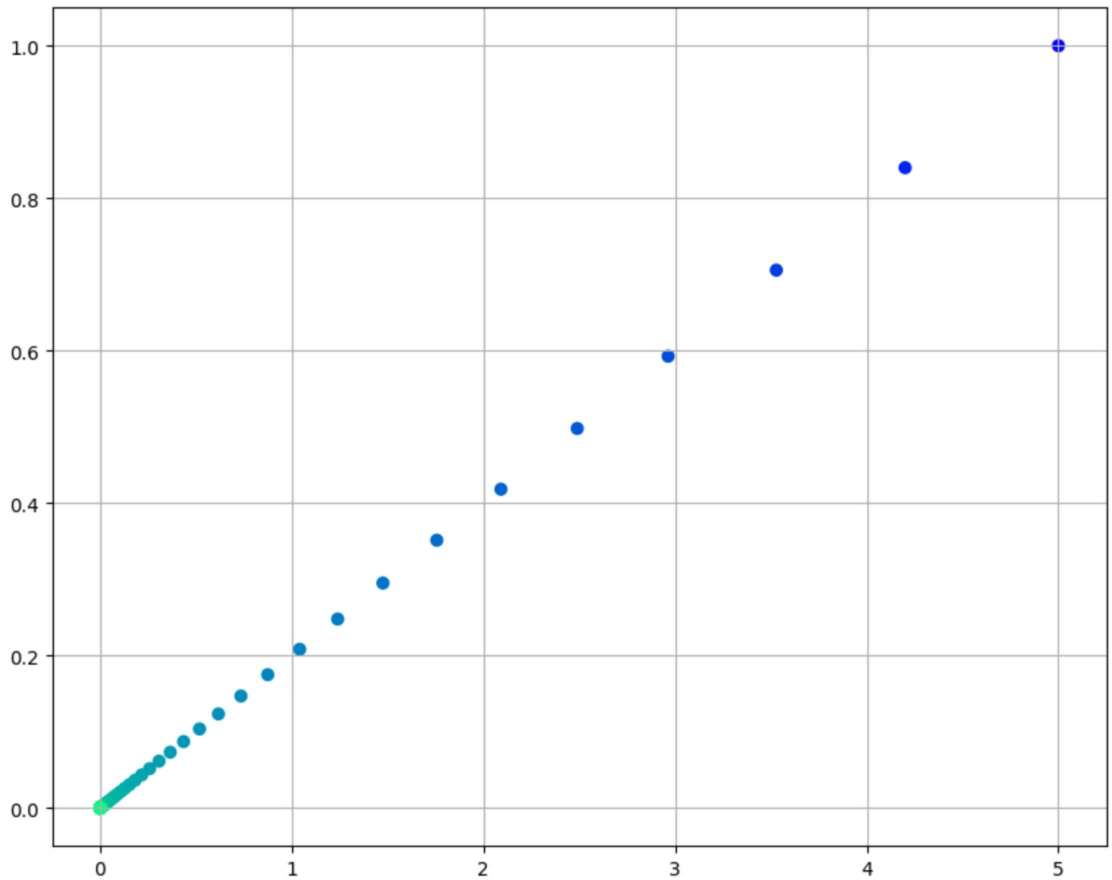
[17]:
```
# With the optimal step size, we avoid the zig zag behavior
# and also converge much more quickly

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)
```



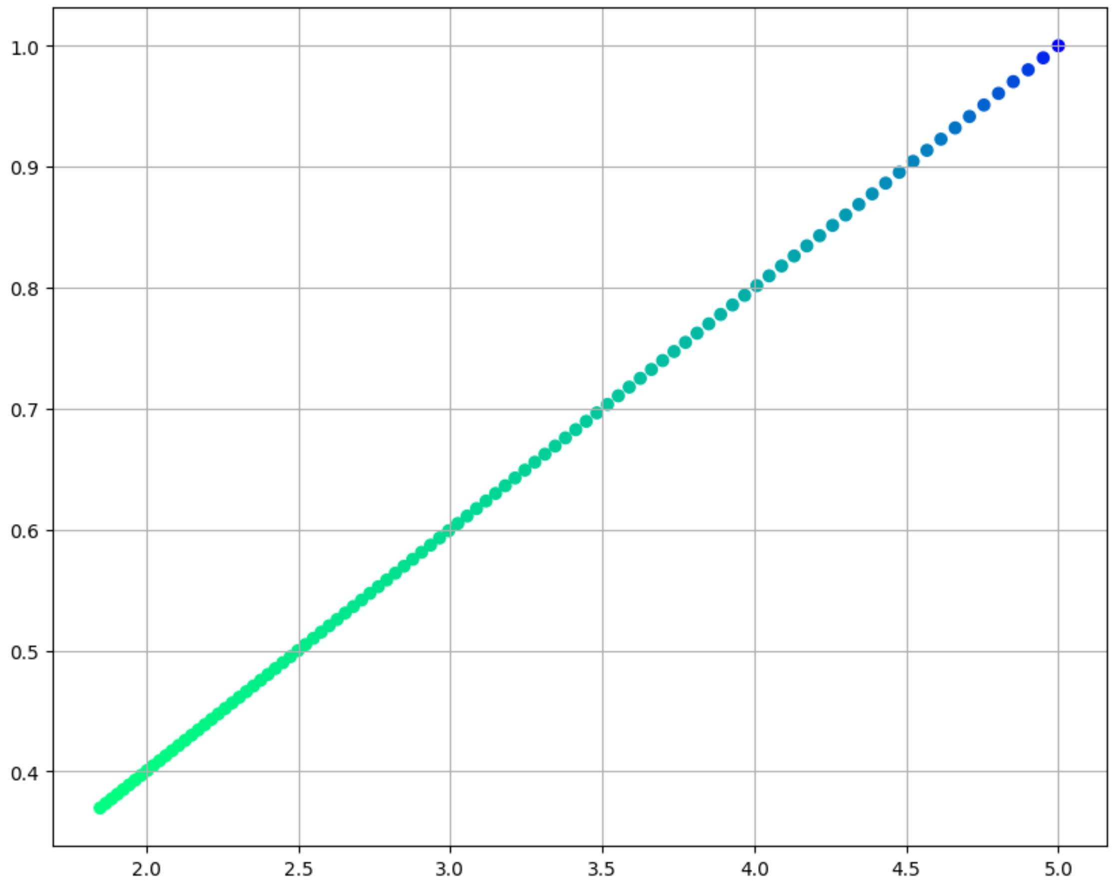
x0 = [5, 1]

- 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.

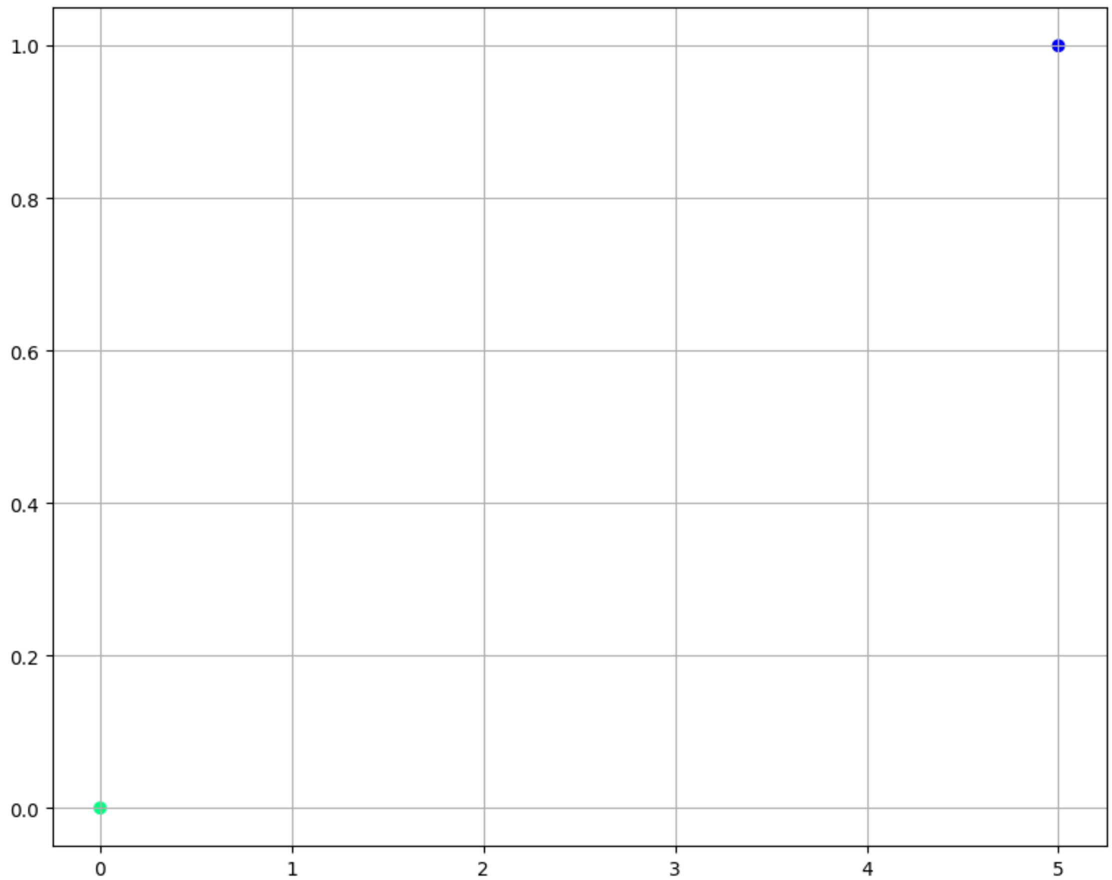
[18]:
```
# For   > 0.1 see significant zig zag
# That is because   is big in the y direction

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]:
```python
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., 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.]], 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]:
```python
# 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))
```