AMS516 Homework 1. Continuous-time stochastic linear quadratic regulators (LQR)

Juan Perez Osorio

1. An infinite-horizon discounted stochastic LQR problem

Consider the following continuous-time dynamics:

$$dx_t = (Ax_t + Bu_t)dt + Gdw_t$$

where:

- $x_t \in \mathbb{R}^n$ is the state vector
- $u_t \in \mathbb{R}^m$ is the control input
- w_t is a standard r-dimensional Wiener process
- $G \in \mathbb{R}^{n \times r}$ is the noise gain matrix

The discounted quadratic cost is given by:

$$J(x_0; u) = \mathbb{E}\left[\int_0^\infty e^{-\rho t}(x_t^\intercal Q x_t + u_t^\intercal R u_t)dt\right]$$

with:

- Discount rate $\rho > 0$
- $Q \geq 0$ (positive semidefinite)
- $R \geq 0$ (positive semidefinite)

The goal is to find optimal feedback control $u_t = -Kx_t$ minimizing J.

The Hamilton-Jacobi-Bellman (HJB) equation for the discounted cost is:

$$\rho V(x) = \min_u \left\{ x^\intercal Q x + u^\intercal R u + \nabla V(x)^\intercal (Ax + Bu) + \frac{1}{2}\mathrm{tr}(GG^\intercal \nabla^2 V) \right\}$$

The value function $V(x)$ and the optimal control u have the form:

$$V(x) = x^\intercal P x + c$$

$$u = -R^{-1}B^\intercal P x$$

Moreover, P satisfies the discounted continuous algebraic Riccati equation (CARE)

$$A^\intercal P + PA - PBR^{-1}B^\intercal P + Q - \rho P = 0$$

and c is given by:

$$c = \frac{\mathrm{tr}(GG^T P)}{\rho}.$$

Consider the problem with the following parameters:

$$\rho = 0.1, \quad A = \begin{bmatrix} 0 & 1 \\ -2 & -3 \end{bmatrix}, \quad B = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \quad G = \begin{bmatrix} 0.1 & 0 \\ 0 & 0.1 \end{bmatrix}, \quad Q = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

(a). Use the analytical result above to compute the value function $V(x)$ and optimal control u

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
from scipy.linalg import solve_continuous_are, solve_lyapunov, eigvals, norm
```

In [2]:
```
rho: float = 0.1

A = np.array([[0, 1], [-2, -3]])
B = np.array([[0], [1]])
G = np.array([[0.1, 0], [0, 0.1]])
Q = np.array([[1, 0], [0, 1]])
R = np.array([[1]])
```

We can see that when checking the library for the function `solve_continuos_are`, we found that it solves the continuos-time algebraic Riccati equation (CARE):

$$A^H X + XA - XBR^{-1}B^H X + Q = 0 \tag{1}$$

But we now that $P = X$ satisfies the discounted continuous algebraic Riccati equation (CARE):

$$A^\intercal P + PA - PBR^{-1}B^\intercal P + Q - \rho P = 0$$

So we need to turn our last equation into the form of (1) so we can apply the scipy method. To do so we manipulate the expression in the following way

$$
\begin{aligned}
A^\intercal P + PA - PBR^{-1}B^\intercal P + Q - \rho P &= 0 \\
A^\intercal P + PA - PBR^{-1}B^\intercal P + Q &= \rho P \\
A^\intercal P + PA - PBR^{-1}B^\intercal P + Q &= \frac{1}{2}\rho(IP + PI) \\
\left(A^\intercal P - \frac{1}{2}\rho IP\right) + \left(PA - \frac{1}{2}\rho PI\right) - PBR^{-1}B^\intercal P + Q &= 0 \\
\left(A^\intercal - \frac{1}{2}\rho I\right)P + P\left(A - \frac{1}{2}\rho I\right) - PBR^{-1}B^\intercal P + Q &= 0
\end{aligned}
$$

Letting $A_{\text{disc}} = A - \frac{1}{2}\rho I$, we can see that the previous equation reduces to

$$A_{\text{disc}}^\intercal P + PA_{\text{disc}} - PBR^{-1}B^\intercal P + Q = 0$$

we can see that our final expression is now in the same form as (1) and hence, we can apply the scipy method

In [3]:
```
I = np.identity(A.shape[0])

# Modify A so that we can use the solve_continuos_are method from scipy
A_discounted = A - (1/2) * rho * I

# Solve continuous-time algebraic Riccati equation (CARE)
P = solve_continuous_are(a = A_discounted, b = B, q = Q, r = R)
```

The optimal gain matrix K is defined to be

$$K = R^{-1}B^\intercal P$$

so we have that

$$u = -Kx$$

In [4]:
```
# Optimal gain matrix
K = np.linalg.inv(R) @ B.T @ P
```

Replacing this in the time dynamics we have

$$
\begin{aligned}
dx_t &= (Ax_t + Bu_t)dt + Gdw_t \\
&= (Ax_t - BKx_t)dt + Gdw_t \\
&= (A - BK)x_t dt + Gdw_t \\
&= (A_{\text{closed}})x_t dt + Gdw_t
\end{aligned}
$$

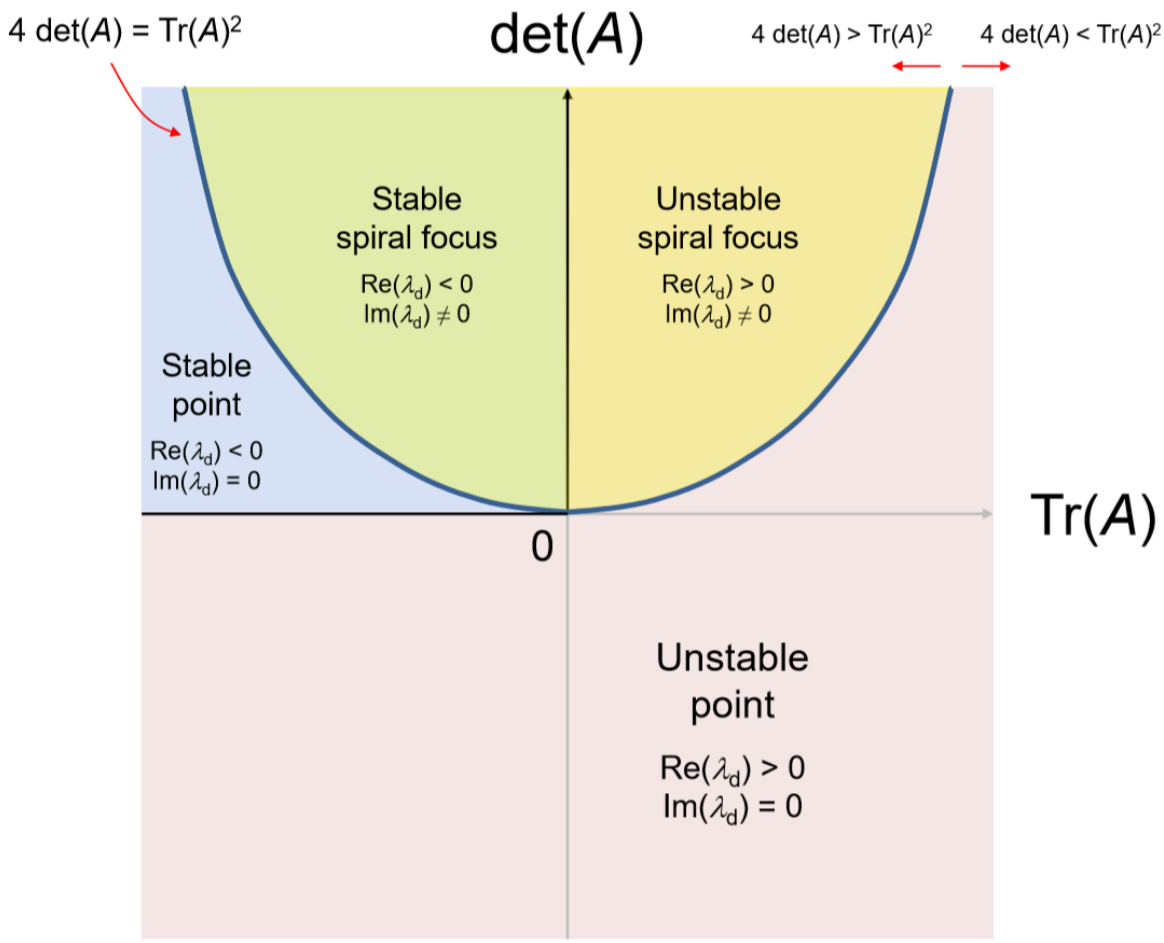
Where we define A_{closed} to be the closed loop matrix and it describes the dynamics of a system with feedback control law, i.e., the closed-loop system is obtained by substituting the optimal control u into the dynamics of the system.

A nice way to checking the stability of the system is by checking the eigenvalues of the closed loop matrix

> we can summarize how the two-dimensional linear dynamical system's stability depends on $\mathrm{Tr}(A)$ and $det(A)$ in a simple diagram as shown

In [4]:
```
from IPython.display import Image
Image('images/stability.png')
```

Out[4]:



Hence we have to check that all the eigenvalues of the closed loop matrix are negative so that our system is stable; we do this procedure to confirm that the solution P from the (discounted) CARE produces a stabilizing controller K. If any eigenvalue had a positive real part, the controller would be useless as the state would blow up.

In [5]:
```
# Stability analysis
A_cl = A - B @ K
eig_A_cl = eigvals(A_cl)

print("\nEigenvalues of A_cl:")
print(eig_A_cl)
```

```python
# Check if all real parts are negative (system is stable)
is_stable = np.all(np.real(eig_A_cl) < 0)
print(f"Is the closed-loop system stable? {is_stable}")
```

```
Eigenvalues of A_cl:
[-0.98838969+0.j -2.23620796+0.j]
Is the closed-loop system stable? True
```

Let's define $X_t = \mathbb{E}[x_t x_t^T]$. This is the covariance matrix of the state at time t. Its evolution is governed by the SDE. Itô's Lemma tells us how to find the differential $d(x_t x_t^T)$.

$$\begin{aligned} d(x_t x_t^\intercal) =& [d(x_t)]\, x_t^\intercal + x_t \left[d(x_t^\intercal)\right] + [d(x_t)] \left[d(x_t^\intercal)\right] \\ =& [A_{\text{closed}} x_t dt + G dw_t]\, x_t^\intercal + x_t \left[(A_{\text{closed}} x_t dt + G dw_t)^\intercal\right] + [d(x_t)] \left[d(x_t^\intercal)\right. \end{aligned}$$

but we know that

$$[d(x_t)][d(x_t)]^\intercal = (G dw_t)(G dw_t)^\intercal = G(dw_t dw_t^\intercal)G^\intercal = G(I dt)G^\intercal = GG^\intercal dt$$

Hence, we have

$$d(x_t x_t^\intercal) = (A_{\text{closed}} x_t dt + G dw_t)\, x_t^\intercal + x_t (A_{\text{closed}} x_t dt + G dw_t)^\intercal + GG^\intercal dt$$

We want an equation for $dX_t = d(\mathbb{E}[x_t x_t^\intercal]) = \mathbb{E}[d(x_t x_t^\intercal)]$.

- $\mathbb{E}[G dw_t x_t^\intercal] = 0$ and $\mathbb{E}[x_t dw_t^\intercal G^\intercal] = 0$ because the current state x_t is uncorrelated with the future noise dw_t.
- $\mathbb{E}[A_{cl} x_t x_t^\intercal dt] = A_{cl}\mathbb{E}[x_t x_t^\intercal] dt = A_{cl} X_t dt$
- $\mathbb{E}[x_t x_t^\intercal A_{cl}^\intercal dt] = X_t A_{cl}^\intercal dt$
- $\mathbb{E}[GG^\intercal dt] = GG^\intercal dt$

Hence we have that

$$dX_t = \mathbb{E}[d(x_t x_t^\intercal)] = A_{\text{closed}} X_t dt + X_t A_{\text{closed}}^\intercal dt + GG^\intercal dt$$

$$\frac{dX_t}{dt} = A_{\text{closed}} X_t + X_t A_{\text{closed}}^\intercal + GG^\intercal$$

At steady-state, the system's statistical properties don't change anymore. This means the covariance matrix X_t settles to a constant matrix Σ, so its derivative becomes zero $dX_t/dt = 0$. Hence, at steady state we have that

$$A_{\text{closed}} \Sigma + \Sigma A_{\text{closed}}^\intercal + GG^\intercal = 0$$

This is the famous **Lyapunov equation**. To solve it we use the `solve_lyapunov` from the scipy library

In [6]:
```python
# Steady state covariance
GGt = G @ G.T

# Solves the continuous Lyapunov equation AX + XA^H = Q
S = solve_lyapunov(A_cl, -GGt)

# The discounted value constants are given by
c = np.trace(GGt @ P) / rho

# Example: initial state and optimal discounted cost
x0 = np.array([1.0, 0])
J0 = x0.T @ P @ x0 + c

print('rho = ', rho)
print('P = \n', P)
print('K = \n', K)
print('Sigma = \n', S)
print('Value constant c = ', c)
print('J(x0) = ', J0)
```

```
rho =  0.1
P =
 [[1.14817499 0.2102449 ]
 [0.2102449  0.22459765]]
K =
 [[0.2102449  0.22459765]]
Sigma =
 [[ 0.00954679 -0.005     ]
 [-0.005       0.00497774]]
Value constant c =  0.13727726484530767
J(x0) =  1.2854522598310418
```

(b). Show that the optimal control u in (3) is given by $u^* = -R^{-1}B^\intercal \nabla V$. Plug u^* into (3), we obtain a differential equation of V. Use the PINN method to solve the obtained equation for V and u. Denote your solution by $\hat{V}$ and $\hat{u}$.

We know that the HJB equation for the discounted cost is

$$\rho V(x) = \min_u \left\{ x^\intercal Q x + u^\intercal R u + \nabla V(x)^\intercal (Ax + Bu) + \frac{1}{2}\mathrm{tr}(GG^\intercal \nabla^2 V) \right\}$$

as we can see, not all terms inside the minimization objective contain u, hence, we can split this in two parts

$$\rho V(x) = \min_u \{u^\intercal R u + \nabla V(x)^\intercal B u\} + \left[x^\intercal Q x + \nabla V(x)^\intercal A x + \frac{1}{2}\mathrm{tr}(GG^\intercal \nabla^2 V) \right.$$

Since we want to find the u that minimizes the expression, we can focus solely on the part that depends on u, let's define this part by f

$$f(u) = u^\intercal R u + \nabla V(x)^\intercal B u$$

To find the minimum of the function $g(u)$ with respect to the vector u, we take its gradient (derivative with respect to u) and set it equal to zero. Computing the Gradient $\nabla_u f(u)$ we obtain that

$$\nabla_u f(u) = (R + R^\intercal)u + B^\intercal \nabla V(x)$$

But since R is symmetric we have that

$$\nabla_u f(u) = 2Ru + B^\intercal \nabla V(x) = 0$$

$$2Ru = -B^\intercal \nabla V(x)$$

$$u^* = -\frac{1}{2} R^{-1} B^\intercal \nabla V(x)$$

Plugging this into the HJB equation for the discounted cost we can notice that

$$\nabla V(x)^\intercal B u^* = \nabla V(x)^T B(-R^{-1}B^T \nabla V) = -\frac{1}{2} \nabla V^\intercal B R^{-1} B^\intercal \nabla V$$

$$\begin{aligned} u^{*\intercal} R u^* &= \left(-\frac{1}{2} R^{-1} B^\intercal \nabla V\right)^\intercal R \left(-\frac{1}{2} R^{-1} B^\intercal \nabla V\right) \\ &= \frac{1}{4} \nabla V^\intercal B (R^{-1})^\intercal R R^{-1} B^\intercal \nabla V \\ &= \frac{1}{4} \nabla V^\intercal B R^{-1} B^\intercal \nabla V \end{aligned}$$

Now we plug these back in. Notice that

$$(u^*)^\intercal R u^* + \nabla V(x)^\intercal B u^* = \frac{1}{4} \nabla V^\intercal B R^{-1} B^\intercal \nabla V - \frac{1}{2} \nabla V^\intercal B R^{-1} B^\intercal \nabla V = -\frac{1}{4}$$

Therefore we have that the HJB equation for the discounted cost takes the form

$$\rho V(x) = x^\intercal Q x + \nabla V(x)^\intercal A x - \frac{1}{4} \nabla V^\intercal B R^{-1} B^\intercal \nabla V + \frac{1}{2} \mathrm{tr}(G G^\intercal \nabla^2 V)$$

So, we can define our loss function to be

$$L_{\text{physics}} = \left| \rho V(x) - x^\intercal Q x - \nabla V(x)^\intercal A x + \frac{1}{4} \nabla V(x)^\intercal B R^{-1} B^\intercal \nabla V(x) - \frac{1}{2} \mathrm{tr}(G \right.$$

Now we're going to set up the PINN in torch

1. Import required libraries

In [7]:
```
import torch
import torch.nn as nn
from torch.utils.data import DataLoader, TensorDataset
```

```
import torch.optim as optim
import matplotlib as mpl
```

In [8]:
```
plt.rcParams.update({
    "text.usetex": True,
    "font.family": "sans-serif",
    "font.sans-serif": "Helvetica",
})
```

In [9]:
```
# Set device: allows us to specify whether computations should occur on the
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Test to see where the tensor was created
tensor_on_device = torch.rand(5, device=device)
print(f"Tensor created directly on {tensor_on_device.device}")

# Convert the system parameters to torch tensors
rho: float = 0.1
A = torch.tensor([[0.0, 1.0], [-2.0, -3.0]], device = device)
B = torch.tensor([[0.0], [1.0]], device = device)
G = torch.tensor([[0.1, 0.0], [0.0, 0.1]], device = device)
Q = torch.tensor([[1.0, 0.0], [0.0, 1.0]], device = device)
R = torch.tensor([[1.0]], device = device)
R_inv = torch.linalg.inv(R)

# Pre-compute useful matrices
B_Rinv_BT = B @ R_inv @ B.T
GGt = G @ G.T
```

```
Tensor created directly on cpu
```

In [10]:
```
torch.cuda.is_available()
```

Out[10]:
```
False
```

Neural network definition:

In [24]:
```
class ValueNetwork(nn.Module):
    def __init__(self, input_dim=2, hidden_dim=64, out_dim=1, num_layers=4):
        super(ValueNetwork, self).__init__()

        # Define the input layers
        layers = []
        # Input layer
        layers.append(nn.Linear(input_dim, hidden_dim))
        layers.append(nn.Tanh())

        # Hidden layers
        for _ in range(num_layers - 1):
            layers.append(nn.Linear(hidden_dim, hidden_dim))
            layers.append(nn.Tanh())

        # Output layer (value function V(x))
        layers.append(nn.Linear(hidden_dim, out_dim))
```

```
        # Define the sequential network
        self.network = nn.Sequential(*layers)

    # Define the mathematical transformation from input to output
    def forward(self, x):
        return self.network(x)
```

PINN Loss Function

In [25]:

```
class HJB_PINN:
    def __init__(self, model, A, B, Q, R, R_inv, GGt, rho):
        self.model = model
        self.A = A
        self.B = B
        self.Q = Q
        self.R = R
        self.R_inv = R_inv
        self.GGt = GGt
        self.rho = rho
        self.B_Rinv_BT = B @ R_inv @ B.T

    def compute_gradients(self, x):
        '''Compute V, ∇V, and ∇²V using automatic differentiation'''
        x.requires_grad_(True)

        # Compute V(x)
        V = self.model(x)

        # Compute ∇V(x) (gradient of V with respect to x)
        grad_V = torch.autograd.grad(V.sum(), x, create_graph=True)[0]

        # Compute ∇²V (Hessian)
        hessian_V = []
        for i in range(x.shape[1]):
            grad_V_i = grad_V[:, i]    # i-th component of the gradient
            hessian_i = torch.autograd.grad(grad_V_i.sum(), x, create_graph=
            hessian_V.append(hessian_i.unsqueeze(1))

        # Stack to get full Hessian: [batch_size, n, n]
        hessian_V = torch.cat(hessian_V, dim=1)

        return V, grad_V, hessian_V

    def physics_loss(self, x):
        '''Compute the HJB PDE residual'''
        V, grad_V, hessian_V = self.compute_gradients(x)

        # HJB PDE terms
        term1 = torch.einsum('bi,ij,bj->b', x, self.Q, x).unsqueeze(1)  # xᵀ
        term2 = torch.einsum('bi,ij,bj->b', grad_V, self.A, x).unsqueeze(1)

        # GGᵀ∇²V for each sample in batch
        GGt_hessian = torch.einsum('ij,bjk->bik', self.GGt, hessian_V)

        # Sum of diagonal elements for each batch (Trace)
```

```
        trace_GGt_hessian = torch.einsum('bii->b', GGt_hessian).unsqueeze(1)
        term3 = 0.5 * trace_GGt_hessian  # ½tr(GGᵀ∇²V)

        # Nonlinear term: -1/4(∇Vᵀ B R⁻¹ Bᵀ ∇V)
        quad_form = torch.einsum('bi,ij,bj->b', grad_V, self.B_Rinv_BT, grad
        term4 = -0.25 * quad_form

        # PDE residual: ρV - [xᵀQx + ∇VᵀAx + ½tr(GGᵀ∇²V) - ¼(∇Vᵀ B R⁻¹ Bᵀ ∇V
        pde_residual = self.rho * V - (term1 + term2 + term3 + term4)

        return torch.mean(pde_residual**2)
```

Training setup and data generation

In [26]:
```
# Generate 1000 points on the domain. From [-2, 2]
def generate_training_data(num_domain=1000, x_range=(-2, 2)):
    '''Generate collocation points for training'''
    # Domain points (random sampling) [Generates random numbers between -2 a
    x_domain = torch.rand(num_domain, 2) * (x_range[1] - x_range[0]) + x_ran

    return x_domain.to(device)

# Initialize the model and trainer
model = ValueNetwork().to(device)
pinn = HJB_PINN(model, A, B, Q, R, R_inv, GGt, rho)

# Generate the training data
x_domain = generate_training_data()

# Optimizer
optimizer = torch.optim.Adam(model.parameters(), lr=5e-3)
scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer, patience=500, fa
```

Training loop

In [27]:
```
def train_pinn(pinn, x_domain, epochs = 5000, print_every = 500):

    losses = []

    for epoch in range(epochs + 1):
        optimizer.zero_grad()

        # Physics loss (PDE residual)
        loss_pde = pinn.physics_loss(x_domain)

        # Total loss
        total_loss = loss_pde

        total_loss.backward()
        optimizer.step()
        scheduler.step(total_loss)

        losses.append(total_loss.item())

        if epoch % print_every == 0:
```

```
            print(f'Epoch {epoch}, Loss: {total_loss.item():.6f}, PDE: {loss

    return losses

# Train the model
print("Start training...")
losses = train_pinn(pinn, x_domain, epochs=5000)
```

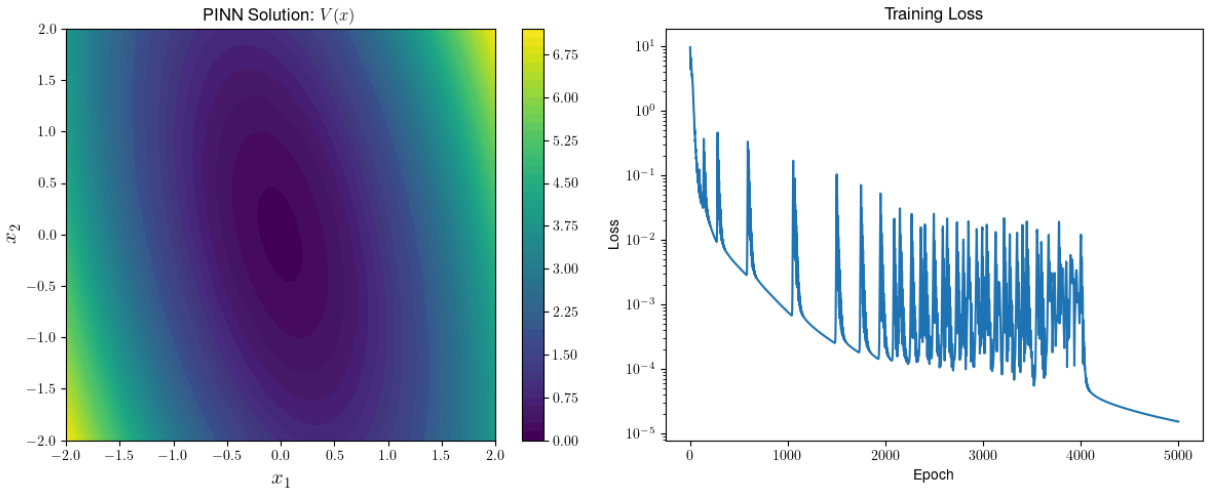
```
Start training...
Epoch 0, Loss: 9.720836, PDE: 9.720836
Epoch 500, Loss: 0.003733, PDE: 0.003733
Epoch 1000, Loss: 0.000757, PDE: 0.000757
Epoch 1500, Loss: 0.085265, PDE: 0.085265
Epoch 2000, Loss: 0.000385, PDE: 0.000385
Epoch 2500, Loss: 0.000530, PDE: 0.000530
Epoch 3000, Loss: 0.003502, PDE: 0.003502
Epoch 3500, Loss: 0.000166, PDE: 0.000166
Epoch 4000, Loss: 0.009402, PDE: 0.009402
Epoch 4500, Loss: 0.000023, PDE: 0.000023
Epoch 5000, Loss: 0.000015, PDE: 0.000015
```

Visualization and analysis

In [28]:

```
def analyze_solution(model):
    """Compare PINN solution with analytical solution"""
    # Create test grid
    x1 = torch.linspace(-2, 2, 50)
    x2 = torch.linspace(-2, 2, 50)
    X1, X2 = torch.meshgrid(x1, x2, indexing='ij')
    # Define pairs of coordinated in the grid
    x_test = torch.stack([X1.ravel(), X2.ravel()], dim=1).to(device)

    # PINN prediction
    with torch.no_grad():
        V_pinn = model(x_test).cpu().numpy()

    # Reshape for plotting
    V_pinn_grid = V_pinn.reshape(50, 50)

    # Plot results
    plt.figure(figsize=(12, 5), dpi=100)

    plt.subplot(1, 2, 1)
    contour = plt.contourf(X1.numpy(), X2.numpy(), V_pinn_grid, levels=50)
    plt.colorbar(contour)
    plt.xlabel(r'$x_1$', fontsize=15)
    plt.ylabel(r'$x_2$', fontsize=15)
    plt.title(r'PINN Solution: $V(x)$')

    plt.subplot(1, 2, 2)
    plt.semilogy(losses)
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.title('Training Loss')

    plt.tight_layout()
```

```
    plt.show()

    return V_pinn_grid

# Analyze the trained model
V_pinn = analyze_solution(model)
```



(c). Compare your results in (a) and (b). Report the ℓ_1 norm of the absolute error $e_1 := \|V(x) - \hat{V}(x)\|_1$ and the relative error $e_2 := \|V(x)/\hat{V}(x) - 1\|_1$ for V.

In [29]:
```
def compare_solutions(model, P_analytical, c_analytical, K_analytical, x_rar
    """Compare PINN solution with analytical solution"""

    # Create test grid
    x1 = np.linspace(x_range[0], x_range[1], grid_points)
    x2 = np.linspace(x_range[0], x_range[1], grid_points)

    # Make a grid of X1 and X2 coordinates
    X1, X2 = np.meshgrid(x1, x2, indexing='ij')
    # Merge both grids in coordinate pairs, shape = (2500, 2) = (gridpoints²
    x_test_np = np.stack([X1.ravel(), X2.ravel()], axis=1)
    x_test_torch = torch.tensor(x_test_np, dtype=torch.float32, device=devic

    # Analytical solutions
    V_analytical = np.array([x.T @ P_analytical @ x + c_analytical for x in
    u_analytical = -K_analytical @ x_test_np.T  # u = -Kx

    # PINN predictions
    with torch.no_grad():
        V_pinn = model(x_test_torch).cpu().numpy().flatten()

    # Compute PINN control (u = -½R⁻¹Bᵀ∇V)
    u_pinn = []
    grad_V_pinn = []
    for x in x_test_np:
        x_tensor = torch.tensor(x, dtype=torch.float32, device=device).unsqu
        x_tensor.requires_grad_(True)
        V = model(x_tensor)
        grad_V = torch.autograd.grad(V, x_tensor, create_graph=False)[0]
        u = -0.5 * (R_inv @ B.T @ grad_V.T).T
```

```
        u_pinn.append(u.cpu().numpy().flatten())
        grad_V_pinn.append(grad_V.cpu().numpy().flatten())

    u_pinn = np.array(u_pinn).T  # Shape: (1, n_points)
    grad_V_pinn = np.array(grad_V_pinn)

    # Reshape for plotting
    V_analytical_grid = V_analytical.reshape(grid_points, grid_points)
    V_pinn_grid = V_pinn.reshape(grid_points, grid_points)
    V_error_grid = np.abs(V_pinn_grid - V_analytical_grid)

    u_analytical_grid = u_analytical.reshape(1, grid_points, grid_points)
    u_pinn_grid = u_pinn.reshape(1, grid_points, grid_points)
    u_error_grid = np.abs(u_pinn_grid - u_analytical_grid)

    return {
        'x1': X1, 'x2': X2,
        'V_analytical': V_analytical_grid, 'V_pinn': V_pinn_grid, 'V_error':
        'u_analytical': u_analytical_grid, 'u_pinn': u_pinn_grid, 'u_error':
        'grad_V_pinn': grad_V_pinn
    }

def plot_comparison(comparison_results):
    """Create comprehensive comparison plots"""

    X1, X2 = comparison_results['x1'], comparison_results['x2']

    fig = plt.figure(figsize=(20, 15), dpi=300)

    # 1. Value Function Comparison

    # Analytical solution for V
    plt.subplot(3, 4, 1)
    contour = plt.contourf(X1, X2, comparison_results['V_analytical'], level
    plt.colorbar(contour)
    plt.xlabel(r'$x_1$', fontsize=15)
    plt.ylabel(r'$x_2$', fontsize=15)
    plt.title(r'Analytical $V(x)$')

    # PINN solution for V
    plt.subplot(3, 4, 2)
    contour = plt.contourf(X1, X2, comparison_results['V_pinn'], levels=50)
    plt.colorbar(contour)
    plt.xlabel(r'$x_1$', fontsize=15)
    plt.ylabel(r'$x_2$', fontsize=15)
    plt.title(r'PINN $\hat{V}(x)$')

    # Mean absolute error (V_analytical - V_pinn)
    plt.subplot(3, 4, 3)
    contour = plt.contourf(X1, X2, comparison_results['V_error'], levels=50)
    plt.colorbar(contour)
    plt.xlabel(r'$x_1$', fontsize=15)
    plt.ylabel(r'$x_2$', fontsize=15)
    plt.title(r'Absolute Error $|V - \hat{V}|$')

    # Scatter plot of V_analytical vs V_pinn
```

```
    plt.subplot(3, 4, 4)
    plt.scatter(comparison_results['V_analytical'].ravel(),
                comparison_results['V_pinn'].ravel(), alpha=0.5, s=1)
    plt.plot([comparison_results['V_analytical'].min(), comparison_results['
             [comparison_results['V_analytical'].min(), comparison_results['
    plt.xlabel(r'Analytical $V(x)$')
    plt.ylabel(r'PINN $\hat{V}(x)$')
    plt.title(r'$V(x)$ Correlation')
    plt.axis('Equal')

    # 2. Control Policy Comparison

    # Analytical solution for u
    plt.subplot(3, 4, 5)
    contour = plt.contourf(X1, X2, comparison_results['u_analytical'][0], le
    plt.colorbar(contour)
    plt.xlabel(r'$x_1$', fontsize=15)
    plt.ylabel(r'$x_2$', fontsize=15)
    plt.title(r'Analytical $u(x)$')

    # PINN solution for u
    plt.subplot(3, 4, 6)
    contour = plt.contourf(X1, X2, comparison_results['u_pinn'][0], levels=5
    plt.colorbar(contour)
    plt.xlabel(r'$x_1$', fontsize=15)
    plt.ylabel(r'$x_2$', fontsize=15)
    plt.title(r'PINN $\hat{u}(x)$')

    # Mean absolute error (u_PINN - u_analytic)
    plt.subplot(3, 4, 7)
    contour = plt.contourf(X1, X2, comparison_results['u_error'][0], levels=
    plt.colorbar(contour)
    plt.xlabel(r'$x_1$', fontsize=15)
    plt.ylabel(r'$x_2$', fontsize=15)
    plt.title(r"Absolute Error $\left|u - \hat{u}\right|$")

    # Scatter plot of u_analytical vs u_pinn
    plt.subplot(3, 4, 8)
    plt.scatter(comparison_results['u_analytical'][0].ravel(),
                comparison_results['u_pinn'][0].ravel(), alpha=0.5, s=1)
    plt.plot([comparison_results['u_analytical'][0].min(), comparison_result
             [comparison_results['u_analytical'][0].min(), comparison_result
    plt.xlabel(r'Analytical $u(x)$')
    plt.ylabel(r'PINN $\hat{u}(x)$')
    plt.title(r'u(x) Correlation')
    plt.axis(r'equal')

    # 3. 3D Surface Plots

    # 3D plot V(x) analytical
    ax = fig.add_subplot(3, 4, 9, projection='3d')
    ax.plot_surface(X1, X2, comparison_results['V_analytical'], cmap='viridi
    ax.set_title(r'Analytical $V(x)$ - 3D')
    ax.set_xlabel(r'$x_1$', fontsize=15)
    ax.set_ylabel(r'$x_2$', fontsize=15)
```

```
    # 3D plot V(x) PINN
    ax = fig.add_subplot(3, 4, 10, projection='3d')
    ax.plot_surface(X1, X2, comparison_results['V_pinn'], cmap='viridis', al
    ax.set_title(r'PINN $\hat{V}(x)$ - 3D')
    ax.set_xlabel(r'$x_1$', fontsize=15)
    ax.set_ylabel(r'$x_2$', fontsize=15)

    # 3D plot MAE (V_PINN - V_analytic)
    ax = fig.add_subplot(3, 4, 11, projection='3d')
    ax.plot_surface(X1, X2, comparison_results['V_error'], cmap='hot', alpha
    ax.set_title(r'V(x) Error - 3D')
    ax.set_xlabel(r'$x_1$', fontsize=15)
    ax.set_ylabel(r'$x_2$', fontsize=15)

    plt.tight_layout()
    plt.show()

    return None
```

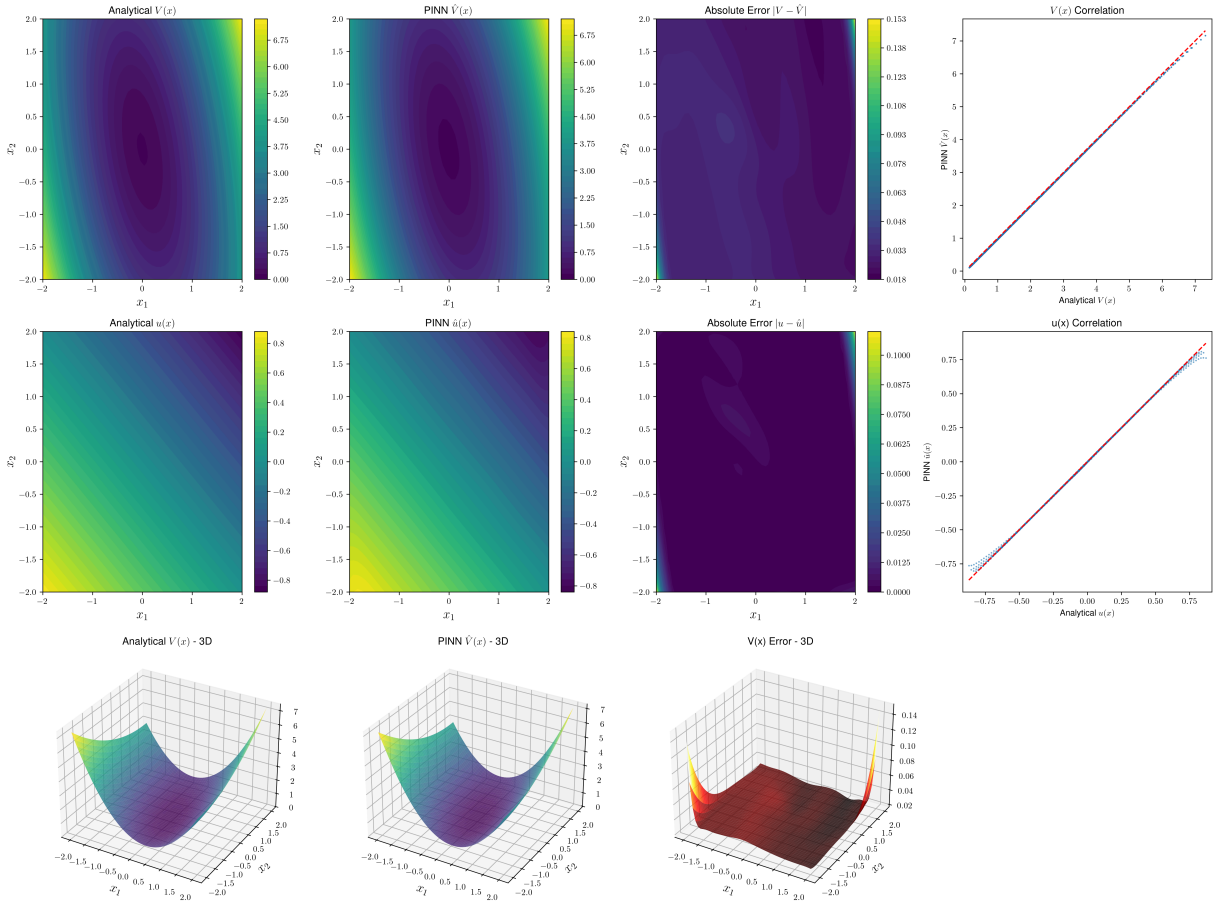
In [30]:

```
def compute_metrics(comparison_results):
    """Compute quantitative comparison metrics"""

    V_analytical_flat = comparison_results['V_analytical'].ravel()
    V_pinn_flat = comparison_results['V_pinn'].ravel()
    u_analytical_flat = comparison_results['u_analytical'][0].ravel()
    u_pinn_flat = comparison_results['u_pinn'][0].ravel()

    # Value function metrics

    # Mean absolute error
    V_mae = np.mean(np.abs(V_pinn_flat - V_analytical_flat))    # L1 mean ab
    # Relative error
    V_rel_err = np.mean(np.abs(V_pinn_flat / V_analytical_flat - 1))
    # Root mean squared error
    V_rmse = np.sqrt(np.mean((V_pinn_flat - V_analytical_flat)**2))
    # Maximum error
    V_max_error = np.max(np.abs(V_pinn_flat - V_analytical_flat))
    # Coefficient of determination
    V_r2 = 1 - np.sum((V_analytical_flat - V_pinn_flat)**2) / np.sum((V_anal

    # Control policy metrics
    u_mae = np.mean(np.abs(u_pinn_flat - u_analytical_flat))
    u_rel_err = np.mean(np.abs(u_pinn_flat / u_analytical_flat - 1))
    u_rmse = np.sqrt(np.mean((u_pinn_flat - u_analytical_flat)**2))
    u_max_error = np.max(np.abs(u_pinn_flat - u_analytical_flat))
    u_r2 = 1 - np.sum((u_pinn_flat - u_analytical_flat)**2) / np.sum((u_anal

    metrics = {
        'V_MAE': V_mae,
        'V_REL_ERR': V_rel_err,
        'V_RMSE': V_rmse,
        'V_Max_Error': V_max_error,
        'V_R2': V_r2,
        'u_MAE': u_mae,
        'u_REL_ERR': u_rel_err,
        'u_RMSE': u_rmse,
```

```
        'u_Max_Error': u_max_error,
        'u_R2': u_r2
    }

    print("QUANTITATIVE COMPARISON METRICS")
    print(f"Value Function V(x):")
    print(f"  MAE: {V_mae:.6f}")
    print(f"  MRE: {V_rel_err:.6f}")
    print(f"  RMSE: {V_rmse:.6f}")
    print(f"  Max Error: {V_max_error:.6f}")
    print(f"  R² Score: {V_r2:.6f}")
    print(f"Control Policy u(x):")
    print(f"  MAE: {u_mae:.6f}")
    print(f"  MRE: {u_rel_err:.6f}")
    print(f"  RMSE: {u_rmse:.6f}")
    print(f"  Max Error: {u_max_error:.6f}")
    print(f"  R² Score: {u_r2:.6f}")

    return metrics
```

In [31]:
```
# From the solved CARE analytically, we have that
P_analytical = P
K_analytical = K
c_analytical = c

# Compare solutions
print("\nComparing PINN with Analytical Solution...")
comparison_results = compare_solutions(model, P_analytical, c_analytical, K_
metrics = compute_metrics(comparison_results)
plot_comparison(comparison_results)
```

```
Comparing PINN with Analytical Solution...
QUANTITATIVE COMPARISON METRICS
Value Function V(x):
  MAE: 0.029689
  MRE: 0.032343
  RMSE: 0.030941
  Max Error: 0.150267
  R² Score: 0.999609
Control Policy u(x):
  MAE: 0.002100
  MRE: 0.026292
  RMSE: 0.007121
  Max Error: 0.108425
  R² Score: 0.999614
```



Notebook

October 20, 2025

0.1 2. Stochastic optimal stopping LQR

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
from scipy.linalg import solve_continuous_are, solve_lyapunov, eigvals, norm,␣
↪solve_continuous_lyapunov
import torch
import torch.nn as nn
from torch.utils.data import DataLoader, TensorDataset
import torch.optim as optim
```

[2]:
```
plt.rcParams.update({
    "text.usetex": True,
    "font.family": "sans-serif",
    "font.sans-serif": "Helvetica",
})
```

Consider a continuous-time risk-sensitive LQR problem, in which dynamics of x follows:

$$dx_t = (Ax_t + Bu_t)dt + Gdw_t$$

where $x_t \in \mathbb{R}^n$, $u_t \in \mathbb{R}^m$, w_t is a standard r-dimensional Wiener process and $G \in \mathbb{R}^{n\times r}$.

The stochastic optimal stopping LQR problem is to minimize the cost function:

$$J(u,\tau) = \mathbb{E}\left[\int_0^\tau e^{-\rho t}\left(x(t)^T Q x(t) + u(t)^T R u(t)\right) dt + e^{-\rho\tau} x(\tau)^T S x(\tau)\right]$$

by choosing optimal control u_t and an optimal stopping time τ. The value function $V(x)$ satisfies the stochastic HJB variational inequality:

$$\min\left\{\rho V(x) - \min_u\left[x^TQx + u^TRu + \nabla V(x)^T(Ax+Bu) + \frac{1}{2}\mathrm{Tr}(GG^T\nabla^2 V(x))\right], V(x) - x^TSx\right\} = 0$$

The solution to the above problem is:

$$V(x) = \begin{cases} x^TPx, & x \in \text{ the continuation region } C = \mathbb{R}^m \quad S \\ x^TSx, & x \in \text{ the stopping region } S \end{cases}$$

This means that, for $x \in C$, apply $u^*(x)$, and for $x \in S$, stop immediately: $\tau = 0$.

In the continuation region C, the optimal control is linear:

$$u^*(x) = -R^{-1}B^T Px$$

and the matrix P satisfies the stochastic Riccati equation:

$$A^T P + PA - PBR^{-1}B^T P + Q - \rho P + PGG^T P = 0$$

The stopping region is given by:

$$S = \{x \in \mathbb{R}^n : x^T Px \leq x^T Sx\}$$

The boundary of S which solves the equation $x^T Px \leq x^T Sx$ is called the stopping boundary. Since the stopping boundary is a function of x, we denote it by $b(x)$.

Consider the problem with parameters:

$$\rho = 0.1, \quad A = \begin{bmatrix} 0 & 1 \\ -2 & -3 \end{bmatrix}, \quad B = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \quad G = \begin{bmatrix} 0.1 & 0 \\ 0 & 0.1 \end{bmatrix}, \quad Q = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad R = [1]$$

and $S = I_2$, solve the following problem:

(a) Use the analytical result above to compute the value function $V(x)$, optimal control u, and the stopping boundary $b(x)$.

To find the value function $V(x)$ we need to solve first the stochastic Riccati equation to obtain P

[3]:
```
# System parameters
rho = 0.1
A = np.array([[0, 1], [-2, -3]])
B = np.array([[0], [1]])
G = np.array([[0.1, 0], [0, 0.1]])
Q = np.array([[1, 0], [0, 1]])
R = np.array([[1]])
S = np.eye(2)  # S = I_2

GGt = G @ G.T
R_inv = np.linalg.inv(R)

# Solve Riccati equation

def solve_stochastic_riccati(A, B, R, Q, G, rho,
                             tol=1e-9, maxiter=50, verbose=False):
    '''Solves the stochastic Riccati equation iteratively
    Inputs: A, B, Q, G are numpy arrays; R is assumed symmetric posdef scalar/
 ↪matrix
    '''
```

```
    n = A.shape[0]
    I = np.eye(n)
    A_discounted = A - 0.5 * rho * I

    # Initial guess: solve CARE ignoring the P GG^T P term
    try:
        Pk = solve_continuous_are(A_discounted, B, Q, R)
    except Exception:
        # fallback initial guess
        Pk = Q.copy()

    def F(P):
        return (A_discounted.T @ P + P @ A_discounted
                - P @ B @ np.linalg.inv(R) @ B.T @ P
                + Q + P @ G @ G.T @ P)

    for k in range(maxiter):
        Fk = F(Pk)
        err = norm(Fk, ord='fro')
        if verbose:
            print(f"iter {k}: ||F(Pk)||_F = {err:.3e}")
        if err < tol:
            print(f"Converged after {k + 1} iterations")
            return Pk

        # Build closed-loop matrix for linearization
        BRB = B @ np.linalg.inv(R) @ B.T   # B R^{-1} B^T
        Acl = A_discounted - BRB @ Pk + (G @ G.T) @ Pk

        # Solve Lyapunov: Acl^T Δ + Δ Acl = -F(Pk)
        # scipy solver expects A X + X A^T = Q; we adapt by transposing
        # Solve (Acl^T) Δ + Δ (Acl) = -Fk  =>  (Acl) Δ^T + Δ^T (Acl)^T = -Fk^T
        # but we can use solve_continuous_lyapunov on Acl and right-hand side:
        # solve_continuous_lyapunov(Acl.T, -Fk) returns Δ
        try:
            Delta = solve_continuous_lyapunov(Acl.T, -Fk)
        except Exception as e:
            raise RuntimeError("Lyapunov solve failed in Newton step: " +␣
↪str(e))

        Pk = Pk + Delta

    raise RuntimeError("Newton iteration did not converge within maxiter")

P = solve_stochastic_riccati(A, B, R, Q, G, rho, verbose=True)
P_analytical = P
```

```
iter 0: ||F(Pk)||_F = 1.425e-02
```

```
iter 1: ||F(Pk)||_F = 8.643e-06
iter 2: ||F(Pk)||_F = 2.949e-12
Converged after 3 iterations
```

[4]:
```
# We can check that indeed our solution is optimal:
A.T @ P + P @ A - P @ B @ R_inv @ B.T @ P + Q - rho * P + P @ GGt @ P
```

[4]:
```
array([[-2.54698519e-12, -1.01155716e-12],
       [-1.01155672e-12, -4.01755777e-13]])
```

We know that in the continuation region C, our optimal solution is given by

$$u^*(x) = -R^{-1}B^T Px$$

which is equivalent to

$$u^*(x) = -Kx$$

[5]:
```
K = np.linalg.inv(R) @ B.T @ P
K
```

[5]:
```
array([[0.21316998, 0.22563788]])
```

[6]:
```
# Stopping region: {x: x Px   x Sx} = {x: x (P-S)x   0}
P_minus_S = P - S
print(f"P - S = \n{P_minus_S}")

eigvals = np.linalg.eigvals(P_minus_S)
print(f"Eigenvalues of (P-S): {eigvals}")
```

```
P - S =
[[ 0.15725402  0.21316998]
 [ 0.21316998 -0.77436212]]
Eigenvalues of (P-S): [ 0.20371405 -0.82082215]
```

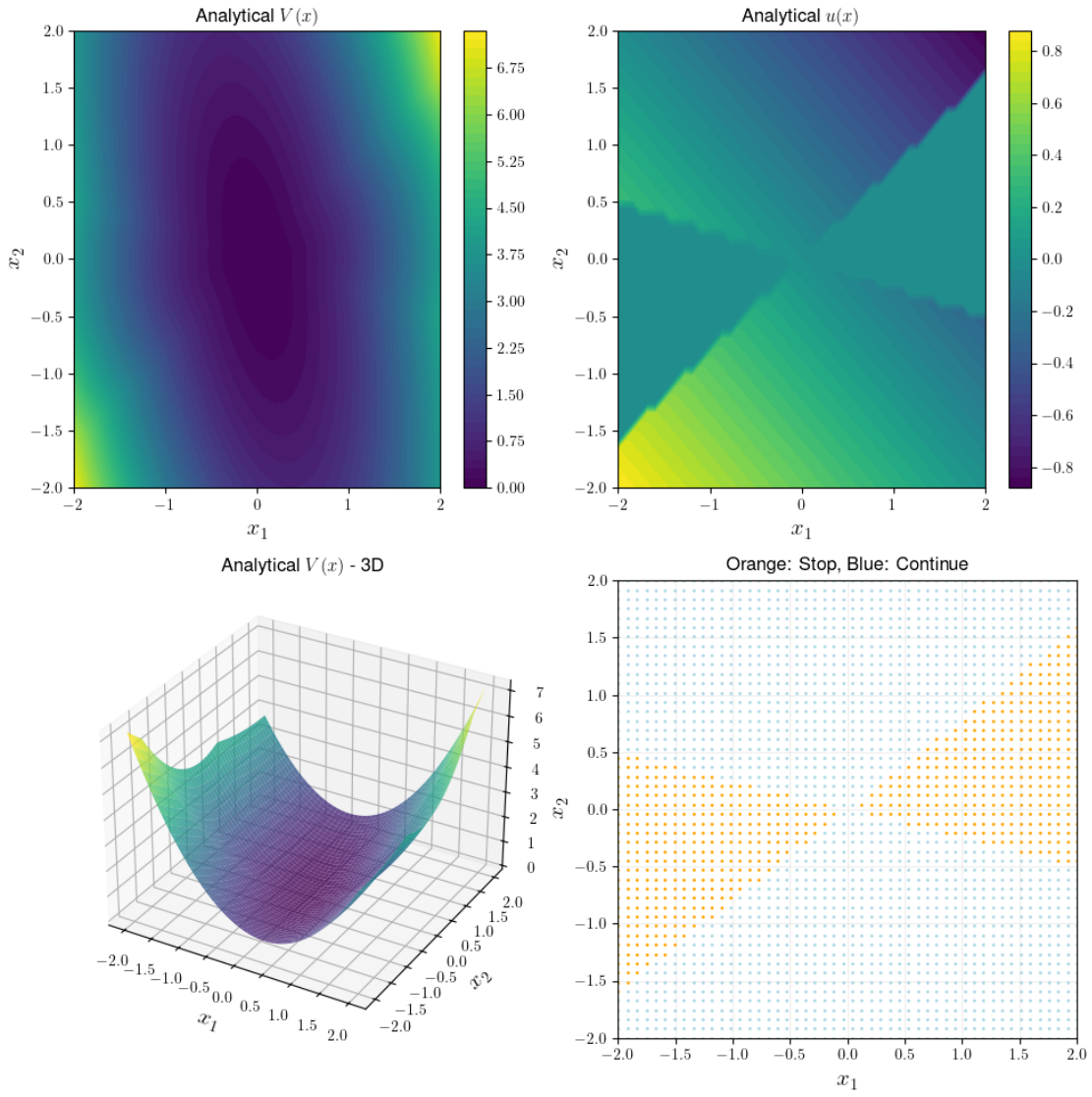
[7]:
```
def exact_solution(P_analytical, S, K_analytical, x_range=(-2, 2),␣
↪grid_points=50):
    """Find analytical solution for each point in the domain"""

    # Create test grid
    x1 = np.linspace(x_range[0], x_range[1], grid_points)
    x2 = np.linspace(x_range[0], x_range[1], grid_points)

    # Make a grid of X1 and X2 coordinates
    X1, X2 = np.meshgrid(x1, x2, indexing='ij')
    # Merge both grids in coordinate pairs, shape = (2500, 2) = (gridpoints², 2)
    x_test_np = np.stack([X1.ravel(), X2.ravel()], axis=1)
```

```
    # Analytical solutions and continuation / stopping regions
    V_analytical = []
    u_analytical = []
    continuation_region = []
    stopping_region = []

    for x in x_test_np:
        V_continue = x.T @ P_analytical @ x
        V_stop = x.T @ S @ x

        V_x = min(V_continue, V_stop)
        V_analytical.append(V_x)

        # Check if the point is in the stopping region
        if (V_continue < V_stop):
            continuation_region.append(x)
            u_analytical.append((-K_analytical @ x) * np.ones_like(x))  # u =␣
↪-Kx
        # Continuation region
        else:
            stopping_region.append(x)
            u_analytical.append(np.zeros_like(x))

    # Reshape for plotting
    V_analytical_grid = np.array(V_analytical).reshape(grid_points, grid_points)
    u_analytical_grid = np.array(u_analytical).T.reshape(2, grid_points,␣
↪grid_points)

    return {
        'x1': X1, 'x2': X2,
        'V_analytical': V_analytical_grid,
        'u_analytical': u_analytical_grid,
        'Regions': [continuation_region, stopping_region]
    }

def plot_solution(exact_results, P, S):
    """Create comprehensive comparison plots"""

    X1, X2 = exact_results['x1'], exact_results['x2']

    fig = plt.figure(figsize=(10, 10))

    # 1. Value Function Comparison

    # Analytical solution for V
    plt.subplot(2, 2, 1)
    contour = plt.contourf(X1, X2, exact_results['V_analytical'], levels=50)
```

```
    plt.colorbar(contour)
    plt.xlabel(r'$x_1$', fontsize=15)
    plt.ylabel(r'$x_2$', fontsize=15)
    plt.title(r'Analytical $V(x)$')

    # 2. Control Policy Comparison

    # Analytical solution for u
    plt.subplot(2, 2, 2)
    contour = plt.contourf(X1, X2, exact_results['u_analytical'][0], levels=50)
    plt.colorbar(contour)
    plt.xlabel(r'$x_1$', fontsize=15)
    plt.ylabel(r'$x_2$', fontsize=15)
    plt.title(r'Analytical $u(x)$')

    # 3. 3D Surface Plots

    # 3D plot V(x) analytical
    ax = fig.add_subplot(2, 2, 3, projection='3d')
    ax.plot_surface(X1, X2, exact_results['V_analytical'], cmap='viridis',␣
↪alpha=0.8)
    ax.set_title(r'Analytical $V(x)$ - 3D')
    ax.set_xlabel(r'$x_1$', fontsize=15)
    ax.set_ylabel(r'$x_2$', fontsize=15)

    # 4. Decision region

    ax = fig.add_subplot(2, 2, 4)
    continue_points = np.array(exact_results['Regions'][0])
    stop_points = np.array(exact_results['Regions'][1])

    # Scatter plot the regions
    ax.scatter(stop_points[:, 0], stop_points[:, 1],
            c='orange', s=1, alpha=1, label='Stop', marker='o')
    ax.scatter(continue_points[:, 0], continue_points[:, 1],
            c='lightblue', s=1, alpha=1, label='Continue')


    ax.set_title('Orange: Stop, Blue: Continue')
    ax.set_xlabel(r'$x_1$', fontsize=15)
    ax.set_ylabel(r'$x_2$', fontsize=15)
    ax.set_xlim(-2, 2)
    ax.set_ylim(-2, 2)
    ax.grid(True, alpha=0.2)

    plt.tight_layout()
    plt.show()
```

```
    return None
```

[8]:
```
exact_solution = exact_solution(P, S, K)
plot_solution(exact_solution, P, S)
```



(b) Use the PINN method to solve the obtained equation for $V(x)$, u and $b(x)$. Denote your solution by $\hat{V}(x)$ and $\hat{u}(x)$. (Hint: Construct two networks for $V(x)$ and $b(x)$, respectively).

[9]:
```
# Common parameters from the homework
rho = 0.1
A = torch.tensor([[0., 1.], [-2., -3.]])
B = torch.tensor([[0.], [1.]])
```

```
G = torch.tensor([[0.1, 0.], [0., 0.1]])
Q = torch.tensor([[1., 0.], [0., 1.]])
R = torch.tensor([[1.]])
GG_T = G @ G.T
```

[10]:
```
# Set default device and data type
DEVICE = 'cuda' if torch.cuda.is_available() else 'cpu'
DTYPE = torch.float32
print(f"Using device: {DEVICE}")

# Common parameters from the homework
rho = 0.1
A = torch.tensor([[0., 1.], [-2., -3.]], dtype=DTYPE, device=DEVICE)
B = torch.tensor([[0.], [1.]], dtype=DTYPE, device=DEVICE)
G = torch.tensor([[0.1, 0.], [0., 0.1]], dtype=DTYPE, device=DEVICE)
Q = torch.tensor([[1., 0.], [0., 1.]], dtype=DTYPE, device=DEVICE)
R = torch.tensor([[1.]], dtype=DTYPE, device=DEVICE)
GG_T = G @ G.T

# PINN training parameters
N_train = 2000
learning_rate = 5e-3
epochs = 5000

# Helper function to compute the Hessian for a batch of inputs
def hessian(output, inputs):
    """Computes the Hessian of a scalar-valued function for a batch of inputs.
    """
    # output shape: (N, 1), inputs shape: (N, 2)
    grad_output = torch.autograd.grad(outputs=output.sum(), inputs=inputs,␣
    ↪create_graph=True)[0]

    hess = []
    for i in range(inputs.shape[1]):
        # grad of each component of the grad_output
        grad_of_grad_i = torch.autograd.grad(outputs=grad_output[:, i].sum(),␣
    ↪inputs=inputs, create_graph=True)[0]
        hess.append(grad_of_grad_i)

    return torch.stack(hess, dim=2) # Result shape: (N, 2, 2)
```

```
Using device: cpu
```

[14]:
```
# Analytical Solution
A_np, B_np, Q_np, R_np = A.cpu().numpy(), B.cpu().numpy(), Q.cpu().numpy(), R.
    ↪cpu().numpy()
A_tilde = A_np - (rho / 2) * np.eye(A_np.shape[0])
```

```
P_analytical = torch.tensor(P_analytical, dtype=DTYPE, device=DEVICE)
S = torch.eye(2, dtype=DTYPE, device=DEVICE)
print("Analytical P matrix for Problem 2:\n", P_analytical.numpy())

def V_analytical_2(x):
    return torch.einsum('bi,ij,bj->b', x, P_analytical, x)

# (b) PINN Method
# Define the PINN model for V(x)
class ValueNetwork(nn.Module):
    def __init__(self):
        super(ValueNetwork, self).__init__()
        self.net = nn.Sequential(
            nn.Linear(2, 64), nn.Tanh(),
            nn.Linear(64, 64), nn.Tanh(),
            nn.Linear(64, 64), nn.Tanh(),
            nn.Linear(64, 64), nn.Tanh(),
            nn.Linear(64, 1)
        )
    def forward(self, x):
        return self.net(x)

V_model_2 = ValueNetwork().to(DEVICE)
optimizer_2 = torch.optim.Adam(V_model_2.parameters(), lr=learning_rate)

# Training loop
domain_points = torch.rand((N_train, 2), device=DEVICE) * 4 - 2 # Points in␣
 ↪[-2, 2]
domain_points.requires_grad = True

for epoch in range(epochs):
    optimizer_2.zero_grad()

    V = V_model_2(domain_points)
    grad_V = torch.autograd.grad(V.sum(), domain_points, create_graph=True)[0]
    hess_V = hessian(V, domain_points)

    # Hamiltonian for the continuation region
    hamiltonian = (torch.einsum('bi,ij,bj->b', domain_points, Q, domain_points)␣
 ↪# x'Qx
                   - 0.25 * torch.einsum('bi,ij,bj->b', grad_V, B @ torch.
 ↪inverse(R) @ B.T, grad_V)
                   + torch.einsum('bi,bi->b', grad_V, domain_points @ A.T) #␣
 ↪gradV'Ax
                   + 0.5 * torch.einsum('bii->b', GG_T @ hess_V)) # 1/2 Tr(GG'␣
 ↪Hessian)
```

```
    # Variational inequality terms
    f1 = rho * V.squeeze() - hamiltonian
    f2 = V.squeeze() - torch.einsum('bi,ij,bj->b', domain_points, S,␣
 ↪domain_points)

    loss_pde = torch.mean((f1 * f2)**2)
    loss_ineq = torch.mean(torch.relu(-f1)**2 + torch.relu(f2)**2)
    loss = loss_pde + loss_ineq

    loss.backward()
    optimizer_2.step()

    if epoch % 500 == 0:
        print(f"Problem 2 - Epoch {epoch}: Loss = {loss.item()}")
```

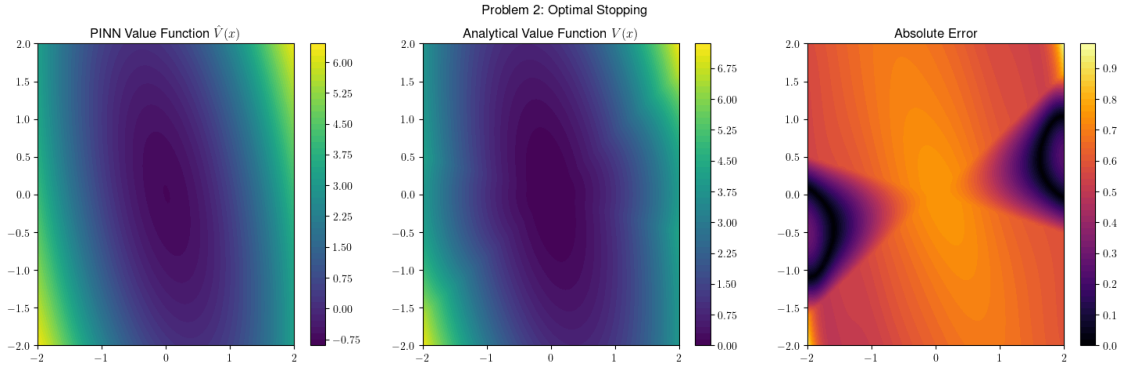
```
Analytical P matrix for Problem 2:
 [[1.157254   0.21316999]
 [0.21316999 0.22563788]]
Problem 2 - Epoch 0: Loss = 222.18482971191406
Problem 2 - Epoch 500: Loss = 0.07166951149702072
Problem 2 - Epoch 1000: Loss = 0.03133003041148186
Problem 2 - Epoch 1500: Loss = 0.012628571130335331
Problem 2 - Epoch 2000: Loss = 0.007192113436758518
Problem 2 - Epoch 2500: Loss = 0.022839773446321487
Problem 2 - Epoch 3000: Loss = 0.0032205763272941113
Problem 2 - Epoch 3500: Loss = 0.013518402352929115
Problem 2 - Epoch 4000: Loss = 0.005636159330606461
Problem 2 - Epoch 4500: Loss = 0.004486419260501862
```

(c) Compare your results in (a) and (b).

[15]:
```
# Comparison
xx, yy = np.meshgrid(np.linspace(-2, 2, 100), np.linspace(-2, 2, 100))
x_grid = torch.tensor(np.vstack([xx.ravel(), yy.ravel()]).T, dtype=DTYPE,␣
 ↪device=DEVICE)

V_pinn_2 = V_model_2(x_grid).detach().cpu().numpy().squeeze()
V_true_2_cont = V_analytical_2(x_grid).detach().cpu().numpy()

# Determine stopping region
is_stopping = (torch.einsum('bi,ij,bj->b', x_grid, S - P_analytical, x_grid) <=␣
 ↪0).cpu().numpy()
V_true_2 = V_true_2_cont
V_true_2[is_stopping] = torch.einsum('bi,ij,bj->b', x_grid[is_stopping], S,␣
 ↪x_grid[is_stopping]).cpu().numpy()

abs_error_l1 = np.mean(np.abs(V_pinn_2 - V_true_2))
rel_error_l1 = np.mean(np.abs(V_pinn_2 / V_true_2 - 1.0))
```

```
print(f"\nProblem 2 - L1 Absolute Error: {abs_error_l1:.4e}")
print(f"Problem 2 - L1 Relative Error: {rel_error_l1:.4e}\n")

# Plotting
fig, axes = plt.subplots(1, 3, figsize=(18, 5))
im1 = axes[0].contourf(xx, yy, V_pinn_2.reshape(100, 100), 50, cmap='viridis')
axes[0].set_title(r'PINN Value Function $\hat{V}(x)$'); fig.colorbar(im1,␣
 ↪ax=axes[0])
im2 = axes[1].contourf(xx, yy, V_true_2.reshape(100, 100), 50, cmap='viridis')
axes[1].set_title(r'Analytical Value Function $V(x)$'); fig.colorbar(im2,␣
 ↪ax=axes[1])
im3 = axes[2].contourf(xx, yy, np.abs(V_pinn_2 - V_true_2).reshape(100, 100),␣
 ↪50, cmap='inferno')
axes[2].set_title('Absolute Error'); fig.colorbar(im3, ax=axes[2])
plt.suptitle("Problem 2: Optimal Stopping")
plt.show()
```

```
Problem 2 - L1 Absolute Error: 5.6506e-01
Problem 2 - L1 Relative Error: 2.9130e+00
```



(d) Explain why it is optimal to stop immediately at time 0 for

$$S = \begin{bmatrix} 2.1482 & 0.2102 \\ 0.2102 & 1.2246 \end{bmatrix}$$

and why it should never stop for

$$S = \begin{bmatrix} 0.1482 & 0.2102 \\ 0.2102 & -0.7754 \end{bmatrix}$$

We know that the stopping region is given by

$$S = \{x \in \mathbb{R}^n : x^T P x \leq x^T S x\}$$

but this is equivalent to

$$S = \{x \in \mathbb{R}^n : x^T (P - S) x \leq 0\}.$$

Hence we only need to analyze the matrix $P - S$.

[16]:
```
P_analytical = P_analytical.cpu().numpy()
S_stop = np.array([[2.1482, 0.2102], [0.2102, 1.2246]])
S_never = np.array([[0.1482, 0.2102], [0.2102, -0.7754]])

print("=== Analysis of Special Cases ===")

# For S_stop: Check if P - S is negative definite
P_minus_S_stop = P_analytical - S_stop
eigvals_stop = np.linalg.eigvals(P_minus_S_stop)
print(f"S_stop case - Eigenvalues of (P-S): {eigvals_stop}")
print("Interpretation: Both eigenvalues are NEGATIVE → P-S is negative␣
 ↪definite")
print("→ x (P-S)x  0 for ALL x → Entire space is stopping region")
print("→ Optimal to stop immediately everywhere")

print("\n" + "="*60 + "\n")

# For S_never: Check if P - S is positive definite
P_minus_S_never = P_analytical - S_never
eigvals_never = np.linalg.eigvals(P_minus_S_never)
print(f"S_never case - Eigenvalues of (P-S): {eigvals_never}")
print("Interpretation: Both eigenvalues are POSITIVE → P-S is positive␣
 ↪definite")
print("→ x (P-S)x  0 for ALL x → Entire space is continuation region")
print("→ Never optimal to stop")
```

```
=== Analysis of Special Cases ===
S_stop case - Eigenvalues of (P-S): [-0.98996555 -0.99994259]
Interpretation: Both eigenvalues are NEGATIVE → P-S is negative definite
→ x (P-S)x  0 for ALL x → Entire space is stopping region
→ Optimal to stop immediately everywhere

============================================================

S_never case - Eigenvalues of (P-S): [1.01003445 1.00005741]
Interpretation: Both eigenvalues are POSITIVE → P-S is positive definite
→ x (P-S)x  0 for ALL x → Entire space is continuation region
→ Never optimal to stop
```

We can see that in the first case, our matrix $P - S$ has negative eigenvalues, hence, we should stop immediately since the entire space is the stopping region. Similarly, in the second case, the matrix $P - S$ is positive definite, which means that all the space is a continuation space, hence we should never stop.

This notebook was converted with convert.ploomber.io

Notebook

October 20, 2025

0.1 3. A Markov jump (or switching) stochastic LQR

[2]:
```
import torch
import torch.nn as nn
import numpy as np
from scipy.linalg import solve_continuous_are, solve_lyapunov
import matplotlib.pyplot as plt
import torch.optim as optim
```

[3]:
```
plt.rcParams.update({
    "text.usetex": True,
    "font.family": "sans-serif",
    "font.sans-serif": "Helvetica",
})
```

(a) Design a PINN algorithm to solve the HJB systems.

[4]:
```
DEVICE = 'cuda' if torch.cuda.is_available() else 'cpu'
DTYPE = torch.float32
```

[5]:
```
# Common parameters from the homework
rho = 0.1
A = torch.tensor([[0., 1.], [-2., -3.]], dtype=DTYPE, device=DEVICE)
B = torch.tensor([[0.], [1.]], dtype=DTYPE, device=DEVICE)
G = torch.tensor([[0.1, 0.], [0., 0.1]], dtype=DTYPE, device=DEVICE)
Q = torch.tensor([[1., 0.], [0., 1.]], dtype=DTYPE, device=DEVICE)
R = torch.tensor([[1.]], dtype=DTYPE, device=DEVICE)
GG_T = G @ G.T

# PINN training parameters
N_train = 2000
learning_rate = 5e-3
epochs = 5000
```

[6]:
```
# Helper function to compute the Hessian for a batch of inputs
def hessian(output, inputs):
    """Computes the Hessian of a scalar-valued function for a batch of inputs.
    """
```

```
    # output shape: (N, 1), inputs shape: (N, 2)
    grad_output = torch.autograd.grad(outputs=output.sum(), inputs=inputs,␣
 ↪create_graph=True)[0]

    hess = []
    for i in range(inputs.shape[1]):
        # grad of each component of the grad_output
        grad_of_grad_i = torch.autograd.grad(outputs=grad_output[:, i].sum(),␣
 ↪inputs=inputs, create_graph=True)[0]
        hess.append(grad_of_grad_i)

    return torch.stack(hess, dim=2) # Result shape: (N, 2, 2)
```

[ ]:
```
class ValueNetwork(nn.Module):
    def __init__(self):
        super(ValueNetwork, self).__init__()
        self.net = nn.Sequential(
            nn.Linear(2, 64), nn.Tanh(),
            nn.Linear(64, 64), nn.Tanh(),
            nn.Linear(64, 64), nn.Tanh(),
            nn.Linear(64, 1)
        )
    def forward(self, x):
        return self.net(x)
```

[12]:
```
# Analytical Solutions and PINN Implementation
P1_analytical = torch.tensor([[1.1527, 0.2149], [0.2149, 0.2295]], dtype=DTYPE,␣
 ↪device=DEVICE)
P2_analytical = torch.tensor([[1.2002, 0.2643], [0.2643, 0.2807]], dtype=DTYPE,␣
 ↪device=DEVICE)
print("Analytical P1 matrix:\n", P1_analytical.cpu().numpy())
print("Analytical P2 matrix:\n", P2_analytical.cpu().numpy())

def V_analytical_3(x, P):
    return torch.einsum('bi,ij,bj->b', x, P, x)

A2 = torch.tensor([[0., 1.], [-1.5, -2.5]], dtype=DTYPE, device=DEVICE)
lambda12, lambda21 = 0.5, 0.3

V1_model_3 = ValueNetwork().to(DEVICE)
V2_model_3 = ValueNetwork().to(DEVICE)
params = list(V1_model_3.parameters()) + list(V2_model_3.parameters())
optimizer_3 = torch.optim.Adam(params, lr=learning_rate)
scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer_3, patience=500,␣
 ↪factor=0.5)

# Training loop
```

```python
for epoch in range(epochs):
    domain_points = torch.rand((N_train, 2), device=DEVICE) * 4 - 2
    domain_points.requires_grad = True

    optimizer_3.zero_grad()

    V1, V2 = V1_model_3(domain_points), V2_model_3(domain_points)
    grad_V1 = torch.autograd.grad(V1.sum(), domain_points, create_graph=True)[0]
    hess_V1 = hessian(V1, domain_points)
    grad_V2 = torch.autograd.grad(V2.sum(), domain_points, create_graph=True)[0]
    hess_V2 = hessian(V2, domain_points)

    # Hamiltonians
    ham1 = (torch.einsum('bi,ij,bj->b', domain_points, Q, domain_points)
            - 0.25 * torch.einsum('bi,ij,bj->b', grad_V1, B @ torch.inverse(R)␣
 ↪@ B.T, grad_V1)
            + torch.einsum('bi,bi->b', grad_V1, domain_points @ A.T)
            + 0.5 * torch.einsum('bii->b', GG_T @ hess_V1))
    ham2 = (torch.einsum('bi,ij,bj->b', domain_points, Q, domain_points)
            - 0.25 * torch.einsum('bi,ij,bj->b', grad_V2, B @ torch.inverse(R)␣
 ↪@ B.T, grad_V2)
            + torch.einsum('bi,bi->b', grad_V2, domain_points @ A2.T)
            + 0.5 * torch.einsum('bii->b', GG_T @ hess_V2))

    # HJB residuals
    f1 = rho * V1.squeeze() - (ham1 + lambda12 * (V2.squeeze() - V1.squeeze()))
    f2 = rho * V2.squeeze() - (ham2 + lambda21 * (V1.squeeze() - V2.squeeze()))
    loss = torch.mean(f1**2) + torch.mean(f2**2)

    loss.backward()
    optimizer_3.step()
    scheduler.step(loss)

    if epoch % 500 == 0:
        print(f"Problem 3 - Epoch {epoch}: Loss = {loss.item()}")
```

```
Analytical P1 matrix:
 [[1.1527 0.2149]
 [0.2149 0.2295]]
Analytical P2 matrix:
 [[1.2002 0.2643]
 [0.2643 0.2807]]
Problem 3 - Epoch 0: Loss = 19.28597640991211
Problem 3 - Epoch 500: Loss = 0.02344566583633423
Problem 3 - Epoch 1000: Loss = 0.006729764398187399
Problem 3 - Epoch 1500: Loss = 0.003459033090621233
Problem 3 - Epoch 2000: Loss = 0.0008236168650910258
```

```
Problem 3 - Epoch 2500: Loss = 0.0007361177704297006
Problem 3 - Epoch 3000: Loss = 0.0007562467362731695
Problem 3 - Epoch 3500: Loss = 0.0180111825466156
Problem 3 - Epoch 4000: Loss = 0.002064173575490713
Problem 3 - Epoch 4500: Loss = 0.002267175354063511
```

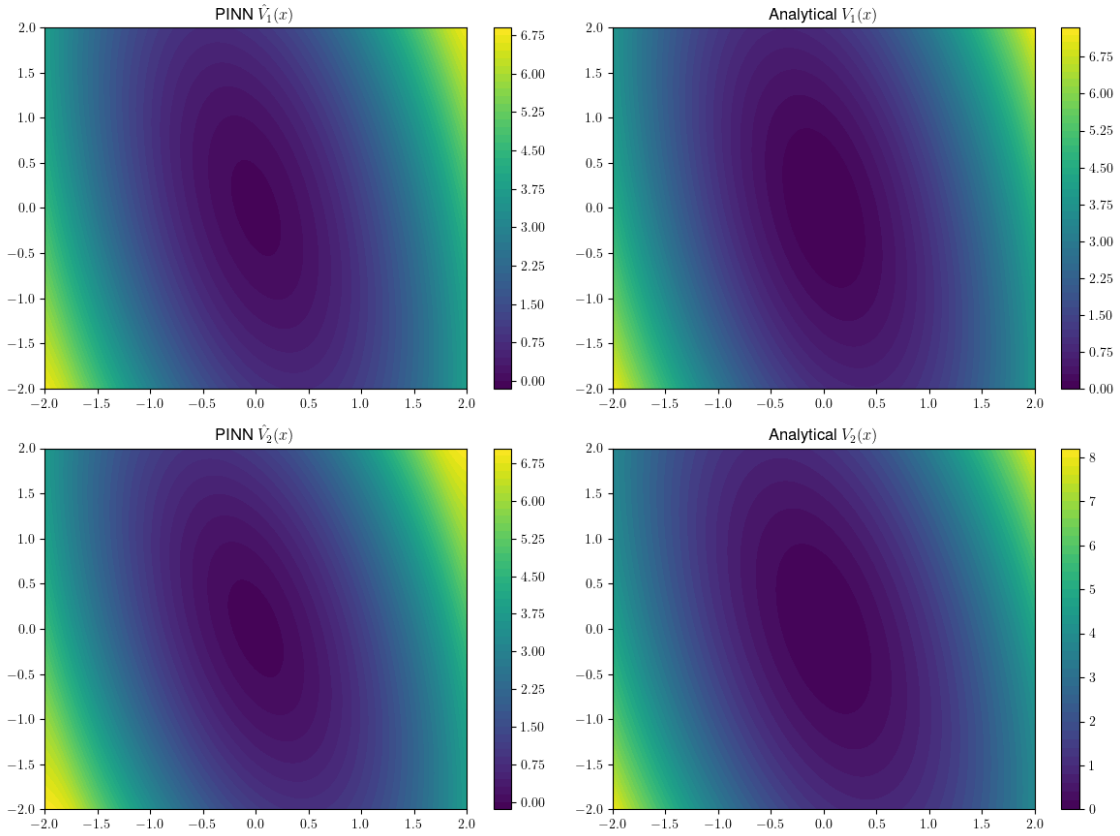
(b) For the given parameters, use your PINN algorithm to solve the problem. Compare your result with the one obtained by analytical solutions.

[13]:
```
# Comparison
xx, yy = np.meshgrid(np.linspace(-2, 2, 100), np.linspace(-2, 2, 100))
x_grid = torch.tensor(np.vstack([xx.ravel(), yy.ravel()]).T, dtype=DTYPE,␣
 ↪device=DEVICE)

V1_pinn = V1_model_3(x_grid).detach().cpu().numpy().squeeze()
V2_pinn = V2_model_3(x_grid).detach().cpu().numpy().squeeze()
V1_true = V_analytical_3(x_grid, P1_analytical).detach().cpu().numpy()
V2_true = V_analytical_3(x_grid, P2_analytical).detach().cpu().numpy()

rel_error_V1 = np.linalg.norm(V1_pinn - V1_true) / np.linalg.norm(V1_true)
rel_error_V2 = np.linalg.norm(V2_pinn - V2_true) / np.linalg.norm(V2_true)
print(f"\nProblem 3 - Relative L2 Error for V1: {rel_error_V1:.4e}")
print(f"Problem 3 - Relative L2 Error for V2: {rel_error_V2:.4e}\n")

# Plot 1: Original 2D Contour Plots
fig, axes = plt.subplots(2, 2, figsize=(12, 10))
im1 = axes[0, 0].contourf(xx, yy, V1_pinn.reshape(100, 100), 50, cmap='viridis')
axes[0, 0].set_title(r'PINN $\hat{V}_1(x)$'); fig.colorbar(im1, ax=axes[0, 0])
im2 = axes[0, 1].contourf(xx, yy, V1_true.reshape(100, 100), 50, cmap='viridis')
axes[0, 1].set_title(r'Analytical $V_1(x)$'); fig.colorbar(im2, ax=axes[0, 1])
im3 = axes[1, 0].contourf(xx, yy, V2_pinn.reshape(100, 100), 50, cmap='viridis')
axes[1, 0].set_title(r'PINN $\hat{V}_2(x)$'); fig.colorbar(im3, ax=axes[1, 0])
im4 = axes[1, 1].contourf(xx, yy, V2_true.reshape(100, 100), 50, cmap='viridis')
axes[1, 1].set_title(r'Analytical $V_2(x)$'); fig.colorbar(im4, ax=axes[1, 1])
plt.suptitle("Problem 3: Markov Jump LQR (2D Contours)", fontsize=16)
plt.tight_layout(rect=[0, 0.03, 1, 0.95])
plt.show()

# Plot 2: New 3D Surface Plots for Comparison
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm

# Create a new figure for the 3D plots
fig_3d = plt.figure(figsize=(16, 7), dpi=300)
fig_3d.suptitle("Problem 3: 3D Surface Comparison", fontsize=16)

# Subplot for V1
ax1 = fig_3d.add_subplot(1, 2, 1, projection='3d')
```

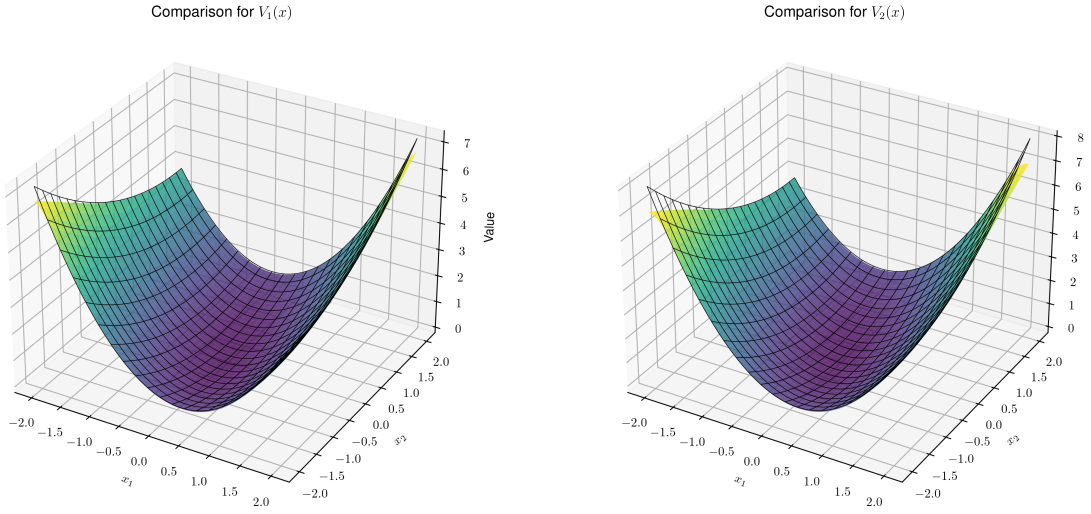
```
# Plot the analytical solution as a wireframe
ax1.plot_wireframe(xx, yy, V1_true.reshape(100, 100), color='black', rstride=5,␣
 ↪cstride=5, linewidth=0.5, label=r'Analytical $V_1(x)$')
# Plot the PINN solution as a surface
surf1 = ax1.plot_surface(xx, yy, V1_pinn.reshape(100, 100), cmap=cm.viridis,␣
 ↪alpha=0.8, label=r'PINN $\hat{V}_1(x)$')
ax1.set_title(r'Comparison for $V_1(x)$')
ax1.set_xlabel(r'$x_1$')
ax1.set_ylabel(r'$x_2$')
ax1.set_zlabel('Value')

# Subplot for V2
ax2 = fig_3d.add_subplot(1, 2, 2, projection='3d')
# Plot the analytical solution as a wireframe
ax2.plot_wireframe(xx, yy, V2_true.reshape(100, 100), color='black', rstride=5,␣
 ↪cstride=5, linewidth=0.5, label=r'Analytical $V_2(x)$')
# Plot the PINN solution as a surface
surf2 = ax2.plot_surface(xx, yy, V2_pinn.reshape(100, 100), cmap=cm.viridis,␣
 ↪alpha=0.8, label=r'PINN $\hat{V}_2(x)$')
ax2.set_title(r'Comparison for $V_2(x)$')
ax2.set_xlabel(r'$x_1$')
ax2.set_ylabel(r'$x_2$')
ax2.set_zlabel('Value')

plt.show()
```

```
Problem 3 - Relative L2 Error for V1: 1.7217e-02
Problem 3 - Relative L2 Error for V2: 2.7033e-02
```

Problem 3: Markov Jump LQR (2D Contours)



Problem 3: 3D Surface Comparison



This notebook was converted with convert.ploomber.io

Notebook

October 20, 2025

0.1 4. Risk sensitive LQR

[8]:
```
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
from scipy.linalg import solve_continuous_are, solve_lyapunov
import matplotlib.pyplot as plt
import numpy as np
```

(a) Use the analytical result to compute the value function $V(x)$ and optimal control u for parameters given by (6).

(b) Use the PINN method to solve the HJB equation (12) for V and u. Denote your solution by $\hat{V}$ and $\hat{u}$.

(c) Compare your results in (a) and (b)

[ ]:
```
DEVICE = 'cuda' if torch.cuda.is_available() else 'cpu'
DTYPE = torch.float32
print(f"Using device: {DEVICE}")

# Common parameters
rho = 0.1
A = torch.tensor([[0., 1.], [-2., -3.]], dtype=DTYPE, device=DEVICE)
B = torch.tensor([[0.], [1.]], dtype=DTYPE, device=DEVICE)
G = torch.tensor([[0.1, 0.], [0., 0.1]], dtype=DTYPE, device=DEVICE)
Q = torch.tensor([[1., 0.], [0., 1.]], dtype=DTYPE, device=DEVICE)
R = torch.tensor([[1.]], dtype=DTYPE, device=DEVICE)
GG_T = G @ G.T

# PINN training parameters
N_train = 2000
learning_rate = 5e-3
epochs = 5000

# Helper function to compute the Hessian
def hessian(output, inputs):
```

```
    grad_output = torch.autograd.grad(outputs=output.sum(), inputs=inputs,␣
 ↪create_graph=True)[0]
    hess = []
    for i in range(inputs.shape[1]):
        grad_of_grad_i = torch.autograd.grad(outputs=grad_output[:, i].sum(),␣
 ↪inputs=inputs, create_graph=True)[0]
        hess.append(grad_of_grad_i)
    return torch.stack(hess, dim=2)

# PINN Model Definition
class ValueNetwork(nn.Module):
    def __init__(self):
        super(ValueNetwork, self).__init__()
        self.net = nn.Sequential(
            nn.Linear(2, 64), nn.Tanh(),
            nn.Linear(64, 64), nn.Tanh(),
            nn.Linear(64, 64), nn.Tanh(),
            nn.Linear(64, 64), nn.Tanh(),
            nn.Linear(64, 1)
        )
    def forward(self, x):
        return self.net(x)

# --- Problem 4: Risk-Sensitive LQR ---
print("--- Solving Problem 4: Risk-Sensitive LQR ---")

def solve_rsare_iteratively(theta, max_iters=100, tol=1e-8):
    """Solves the risk-sensitive algebraic Riccati equation."""
    A_np, B_np, Q_np, R_np = A.cpu().numpy(), B.cpu().numpy(), Q.cpu().numpy(),␣
 ↪R.cpu().numpy()
    n = A_np.shape[0]
    R_inv_np = np.linalg.inv(R_np)
    GG_T_np = GG_T.cpu().numpy()

    P_k = solve_continuous_are(A_np, B_np, Q_np, R_np)
    A_lyap = A_np.T - (rho / 2.0) * np.eye(n)

    for i in range(max_iters):
        # This implementation uses the correct RSARE derived from the HJB␣
 ↪equation (12)
        Q_tilde = -(Q_np - P_k @ B_np @ R_inv_np @ B_np.T @ P_k
                    + theta * 2 * P_k @ GG_T_np @ P_k)

        P_k_plus_1 = solve_lyapunov(A_lyap, Q_tilde)

        if np.linalg.norm(P_k_plus_1 - P_k) < tol:
```

```
            print(f"RSARE solver converged in {i+1} iterations for␣
↪theta={theta}.")
            return P_k_plus_1
        P_k = P_k_plus_1
    print("RSARE solver did not converge.")
    return P_k

def run_problem_4_for_theta(theta):
    print(f"\n--- Running for theta = {theta} ---")

    # (a) Analytical Solution
    P_analytical_4_np = solve_rsare_iteratively(theta)
    P_analytical_4 = torch.tensor(P_analytical_4_np, dtype=DTYPE, device=DEVICE)
    print(f"Analytical P_theta matrix for theta={theta}:\n", P_analytical_4_np)

    def V_analytical_4(x):
        return torch.einsum('bi,ij,bj->b', x, P_analytical_4, x)

    def u_analytical_4(x):
        K = torch.inverse(R) @ B.T @ P_analytical_4
        return -torch.einsum('ij,bj->b', K, x)

    # (b) PINN Method
    V_model_4 = ValueNetwork().to(DEVICE)
    optimizer_4 = torch.optim.Adam(V_model_4.parameters(), lr=learning_rate)
    scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer_4, patience=500,␣
↪factor=0.5)

    for epoch in range(epochs):
        domain_points = torch.rand((N_train, 2), device=DEVICE) * 4 - 2
        domain_points.requires_grad = True
        optimizer_4.zero_grad()

        V = V_model_4(domain_points)
        grad_V = torch.autograd.grad(V.sum(), domain_points,␣
↪create_graph=True)[0]
        hess_V = hessian(V, domain_points)
        gradV_Ax_term = torch.einsum('bi,bi->b', grad_V, domain_points @ A.T)

        hamiltonian = (torch.einsum('bi,ij,bj->b', domain_points, Q,␣
↪domain_points)
                       - 0.25 * torch.einsum('bi,ij,bj->b', grad_V, B @ torch.
↪inverse(R) @ B.T, grad_V)
                       + gradV_Ax_term
                       + 0.5 * torch.einsum('bii->b', GG_T @ hess_V))

        risk_term = theta * torch.einsum('bi,ij,bj->b', grad_V, GG_T, grad_V)
```

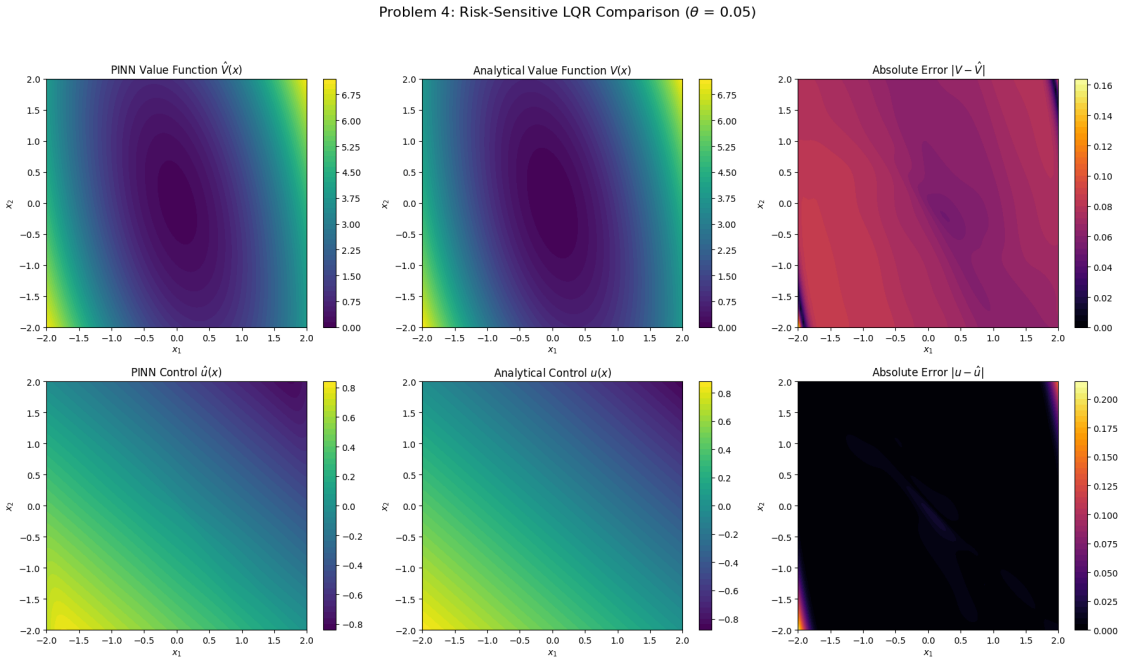
```
        f = rho * V.squeeze() - (hamiltonian + risk_term)
        loss = torch.mean(f**2)

        loss.backward()
        optimizer_4.step()
        scheduler.step(loss)

        if epoch % 1000 == 0:
            print(f"Problem 4 (theta={theta}) - Epoch {epoch}: Loss = {loss.
↪item()}")

    # (c) Comparison
    xx, yy = np.meshgrid(np.linspace(-2, 2, 100), np.linspace(-2, 2, 100))
    x_grid = torch.tensor(np.vstack([xx.ravel(), yy.ravel()]).T, dtype=DTYPE,␣
↪device=DEVICE, requires_grad=True)

    # Calculate V and u for both analytical and PINN methods
    V_true = V_analytical_4(x_grid).detach().cpu().numpy()
    u_true = u_analytical_4(x_grid).detach().cpu().numpy()

    V_pinn_raw = V_model_4(x_grid)
    V_pinn = V_pinn_raw.detach().cpu().numpy().squeeze()

    grad_V_pinn = torch.autograd.grad(V_pinn_raw.sum(), x_grid)[0]
    u_pinn = 0.5 * (-torch.inverse(R) @ B.T @ grad_V_pinn.T).T.detach().cpu().
↪numpy().squeeze()

    # Report L1 errors for V [cite: 115]
    error_v_abs = np.abs(V_pinn - V_true)
    error_v_rel = np.abs(V_pinn / V_true - 1.0)
    e1 = np.mean(error_v_abs)
    e2 = np.mean(error_v_rel)
    print(f"\nProblem 4 (theta={theta}) - L1 Absolute Error (e1): {e1:.4e}")
    print(f"Problem 4 (theta={theta}) - L1 Relative Error (e2): {e2:.4e}\n")

    # Generate Plots
    fig, axes = plt.subplots(2, 3, figsize=(18, 11))
    fig.suptitle(rf'Problem 4: Risk-Sensitive LQR Comparison ($\theta$ =␣
↪{theta})', fontsize=16)

    # --- Row 1: Value Function V(x) ---
    im1 = axes[0, 0].contourf(xx, yy, V_pinn.reshape(100, 100), 50,␣
↪cmap='viridis')
    axes[0, 0].set_title(r'PINN Value Function $\hat{V}(x)$'); fig.
↪colorbar(im1, ax=axes[0, 0])
```

```
    im2 = axes[0, 1].contourf(xx, yy, V_true.reshape(100, 100), 50,␣
↪cmap='viridis')
    axes[0, 1].set_title(r'Analytical Value Function $V(x)$'); fig.
↪colorbar(im2, ax=axes[0, 1])

    im3 = axes[0, 2].contourf(xx, yy, error_v_abs.reshape(100, 100), 50,␣
↪cmap='inferno')
    axes[0, 2].set_title(r'Absolute Error $|V - \hat{V}|$'); fig.colorbar(im3,␣
↪ax=axes[0, 2])

    # --- Row 2: Optimal Control u(x) ---
    im4 = axes[1, 0].contourf(xx, yy, u_pinn.reshape(100, 100), 50,␣
↪cmap='viridis')
    axes[1, 0].set_title(r'PINN Control $\hat{u}(x)$'); fig.colorbar(im4,␣
↪ax=axes[1, 0])

    im5 = axes[1, 1].contourf(xx, yy, u_true.reshape(100, 100), 50,␣
↪cmap='viridis')
    axes[1, 1].set_title(r'Analytical Control $u(x)$'); fig.colorbar(im5,␣
↪ax=axes[1, 1])

    error_u_abs = np.abs(u_pinn - u_true)
    im6 = axes[1, 2].contourf(xx, yy, error_u_abs.reshape(100, 100), 50,␣
↪cmap='inferno')
    axes[1, 2].set_title(r'Absolute Error $|u - \hat{u}|$'); fig.colorbar(im6,␣
↪ax=axes[1, 2])

    for ax_row in axes:
        for ax in ax_row:
            ax.set_xlabel(r'$x_1$'); ax.set_ylabel(r'$x_2$')

    plt.tight_layout(rect=[0, 0.03, 1, 0.95])
    plt.show()

# --- Run for both values of theta ---
run_problem_4_for_theta(theta=0.05)  # Risk-averse
run_problem_4_for_theta(theta=-0.01) # Risk-seeking
```

```
Using device: cpu
--- Solving Problem 4: Risk-Sensitive LQR ---

--- Running for theta = 0.05 ---
RSARE solver converged in 8 iterations for theta=0.05.
Analytical P_theta matrix for theta=0.05:
 [[1.14907136 0.21053332]
 [0.21053332 0.22470021]]
Problem 4 (theta=0.05) - Epoch 0: Loss = 10.218717575073242
```

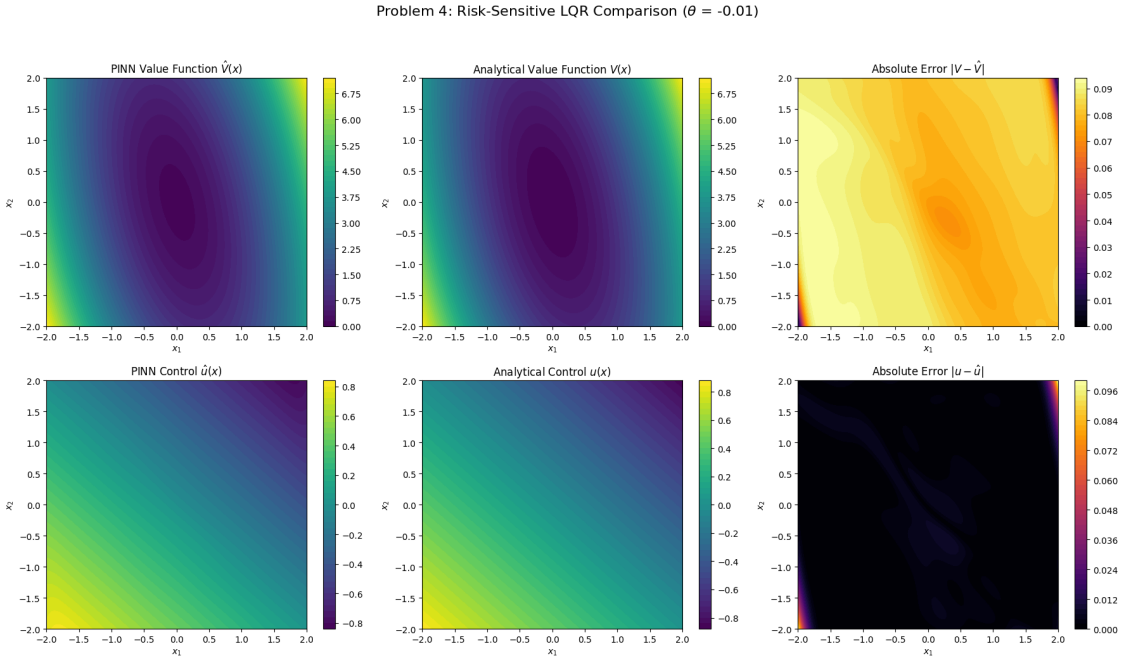
```
c:\Users\Jujop\Documents\GitHub\ams-516\.venv\Lib\site-
packages\torch\optim\lr_scheduler.py:1340: UserWarning: Converting a tensor with
requires_grad=True to a scalar may lead to unexpected behavior.
Consider using tensor.detach() first. (Triggered internally at C:\actions-runner
\_work\pytorch\pytorch\pytorch\torch\csrc\autograd\generated\python_variable_met
hods.cpp:836.)
  current = float(metrics)
```

```
Problem 4 (theta=0.05) - Epoch 1000: Loss = 0.001956452149897814
Problem 4 (theta=0.05) - Epoch 2000: Loss = 0.0014607745688408613
Problem 4 (theta=0.05) - Epoch 3000: Loss = 0.001658852444961667
Problem 4 (theta=0.05) - Epoch 4000: Loss = 2.936747478088364e-05

Problem 4 (theta=0.05) - L1 Absolute Error (e1): 7.0550e-02
Problem 4 (theta=0.05) - L1 Relative Error (e2): 2.4235e-01
```




```
--- Running for theta = -0.01 ---
RSARE solver converged in 8 iterations for theta=-0.01.
Analytical P_theta matrix for theta=-0.01:
 [[1.14799602 0.21018732]
 [0.21018732 0.22457718]]
Problem 4 (theta=-0.01) - Epoch 0: Loss = 9.905126571655273
Problem 4 (theta=-0.01) - Epoch 1000: Loss = 0.004311292432248592
Problem 4 (theta=-0.01) - Epoch 2000: Loss = 0.00031027194927446544
Problem 4 (theta=-0.01) - Epoch 3000: Loss = 0.0001475278550060466
```

```
Problem 4 (theta=-0.01) - Epoch 4000: Loss = 5.097033135825768e-05

Problem 4 (theta=-0.01) - L1 Absolute Error (e1): 8.3390e-02
Problem 4 (theta=-0.01) - L1 Relative Error (e2): 3.0640e-01
```



This notebook was converted with convert.ploomber.io