Homework 1

Instructions

Download this jupyer notebook (button at the top of the page or download from the Github repository). Provide your answers as Markdown text, Python code, and/or produce plots as appropriate. The notebook should run all the cells in order without errors.
Submit both the `.ipynb` and a `.pdf` to Canvas.

Make sure the `.pdf` has all the relevant outputs showing. To save as `.pdf` you can first export the notebook as `.html`, open it in a browers and then "Print to PDF".

NOTE: As we will be sharing the files for peer grading, please keep your submission anonymous.

In [171…

```python
import abc
from typing import Callable
import jax
import jax.numpy as jnp
import matplotlib.pyplot as plt
import numpy as np
import functools
import cvxpy as cp
import math
from jax import grad, jit, vmap
```

Problem 1

In this problem, you will familiarize yourself with the basic operations with dynamical systems and useful JAX operations.

Given the following dynamics class below.

In [172…

```python
class Dynamics(metaclass=abc.ABCMeta):
    dynamics_func: Callable
    state_dim: int
    control_dim: int

    def __init__(self, dynamics_func, state_dim, control_dim):
        self.dynamics_func = dynamics_func
        self.state_dim = state_dim
        self.control_dim = control_dim

    def __call__(self, state, control, time=0):
        return self.dynamics_func(state, control, time)
```

(a) Setting up dynamics

Using the `Dynamics` class, construct the continuous time dynamics for the dynamically extended unicycle model.

$$\dot{\mathbf{x}} = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \\ \dot{v} \end{bmatrix} = \begin{bmatrix} v\cos\theta \\ v\sin\theta \\ \omega \\ a \end{bmatrix}, \qquad u = (\omega, a)$$

In [173…

```
def dynamic_unicycle_ode(state, control, time):
    x, y, theta, v = state
    omega = control[0]
    a = control[1]
    dxdt = v*jnp.cos(theta)
    dydt = v*jnp.sin(theta)
    dthetadt = omega
    dvdt = a

    return jnp.array([dxdt, dydt, dthetadt, dvdt])


state_dim = 4
control_dim = 2
continuous_dynamics = Dynamics(dynamic_unicycle_ode, state_dim, control_dim)
```

(b) Obtaining discrete-time dynamics

With the continuous time dynamics, we can obtain the discrete time dynamics by integrating over a time step Δt.

Implement both Euler integation and Runge-Kutta integration to obtain the discrete-time dynamics.

In [174…

```
def euler_integrate(dynamics, dt):
    # zero-order hold
    def integrator(x, u, t):
        dx = dynamics(x, u, t)
        return x + dt * dx
    return integrator


def runge_kutta_integrator(dynamics, dt=0.1):
    # zero-order hold
    def integrator(x, u, t):
        k1 = dynamics(x, u, t)
        k2 = dynamics(x + 0.5 * dt * k1, u, t + 0.5 * dt)
        k3 = dynamics(x + 0.5 * dt * k2, u, t + 0.5 * dt)
        k4 = dynamics(x + dt * k3, u, t + dt)
        return x + (dt / 6.0) * (k1 + 2*k2 + 2*k3 + k4)
    return integrator
```

In [175…

```
# example usage of the integrators

dt = 0.1 # timestep size

discrete_dynamics_euler = Dynamics(
    euler_integrate(continuous_dynamics, dt), state_dim, control_dim
)
discrete_dynamics_rk = Dynamics(
    runge_kutta_integrator(continuous_dynamics, dt), state_dim, control_dim
)
```

(c) Simulating dynamics

Simulate your dynamics over 5 seconds for different values of Δt and compare the trajectories.

Show on the same plot, the simulated trajectories for the following cases:

- Discrete-time dynamics with Euler integration, $\Delta t = 0.01$
- Discrete-time dynamics with Euler integration, $\Delta t = 0.5$
- Discrete-time dynamics with RK integration, $\Delta t = 0.01$
- Discrete-time dynamics with RK integration, $\Delta t = 0.5$

How does the choice of integration scheme and time step size influence the resulting trajectories? Feel free to try different time step values, but you don't need to submit plots of them.

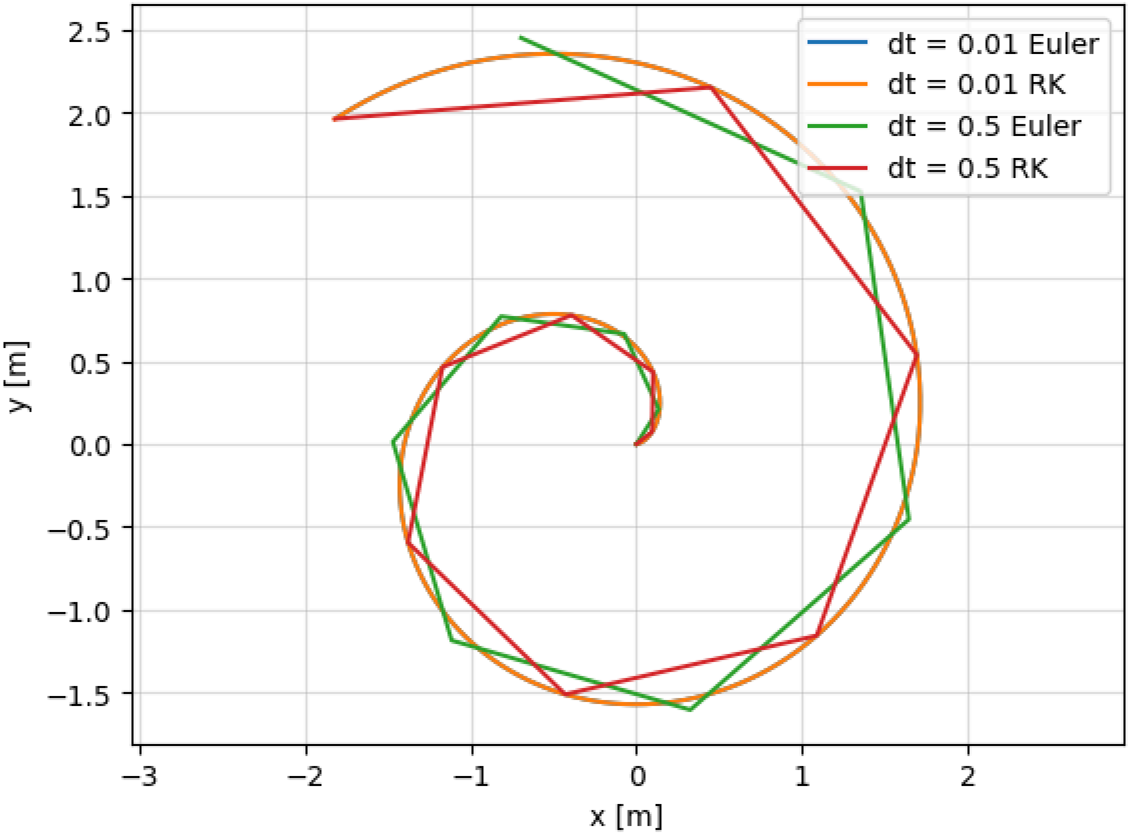
Smaller timesteps generally means a more accurate/smooth trajectory. Smaller timesteps also reveals the advantages of using a more robust integration scheme, as while Euler and RK results look identical at timestep = 0.5, the RK method is significantly more accurate at timestep = 0.01.

In [233…

```
def simulate(dynamics, initial_state, controls, dt):
    state = initial_state
    trajectory = [state]
    time = 0.0

    for u in controls:
        state = dynamics(state, u, time)
        trajectory.append(state)
        time += dt

    return jnp.stack(trajectory)


# code to loop over the different integrators and step sizes
# and plot the corresponding trajectories
```

```
initial_state = jnp.array([0.0, 0.0, 0.0, 0.0])
control = jnp.array([2.0, 1.0])  # constant control over the 5 second duration.
duration = 5.0
dts = [0.01, 0.5]

for dt in dts:
    num_steps = int(duration / dt)
    controls = [control] * num_steps

    # construct the discrete dynamics for given timestep
    discrete_dynamics_euler = Dynamics(
        euler_integrate(continuous_dynamics, dt), state_dim, control_dim
    )
    discrete_dynamics_rk = Dynamics(
        runge_kutta_integrator(continuous_dynamics, dt), state_dim, control_dim
    )

    # simulate dynamics
    xs_euler = simulate(discrete_dynamics_euler, initial_state, controls, dt)
    xs_rk = simulate(discrete_dynamics_rk, initial_state, controls, dt)

    # plot the trajectories
    plt.plot(xs_euler[:, 0], xs_euler[:, 1], label=f"dt = {dt} Euler")
    plt.plot(xs_rk[:, 0], xs_rk[:, 1], label=f"dt = {dt} RK")
    plt.legend()

plt.grid(alpha=0.4)
plt.axis("equal")
plt.xlabel("x [m]")
plt.ylabel("y [m]")
```

```
Out[233…   Text(0, 0.5, 'y [m]')
```



(d) `jax.vmap`

Suppose now that you want to simulate *many* trajectories. Rather than wrapping the `simulate` function in a for loop, we can use `jax.vmap` which is a *vectorize map* function, allowing us to apply a function, in this case `simulate` over multiple inputs.

An example usage of how to use the `jax.vmap` is shown below. Notice that we can specify which argument should be vectorized and along which dimension.

In [177…

```python
def foo(x, y, z):
    return x + y + z

N = 1000
x = jnp.array(np.random.randn(N))
y = jnp.array(np.random.randn(N))
z = jnp.array(np.random.randn(N))

xs = jnp.array(np.random.randn(N, N))
ys = jnp.array(np.random.randn(N, N))
zs = jnp.array(np.random.randn(N, N))

foo(x, y, z)  # non-vectorized version
# vectorized version for all inputs, 0 is the batch dimension for all inputs
jax.vmap(foo, in_axes=[0, 0, 0])(xs, ys, zs)

# x not batched, but ys and zs are with 0 as the batch dimension
jax.vmap(foo, in_axes=[None, 0, 0])(x, ys, zs)
```

```
# y not batched, but xs and zs are with 0 as the batch dimension
jax.vmap(foo, in_axes=[0, None, 0])(xs, y, zs)

# z not batched, but xs and ys are with 0 as the batch dimension
jax.vmap(foo, in_axes=[0, 0, None])(xs, ys, z)

# x and y not batched, but zs is with 0 as the batch dimension
jax.vmap(foo, in_axes=[None, None, 0])(x, y, zs)

# vectorized version for all inputs, batch dimension for xs is 1,
# while 0 is the batch dimension for yx and zs
jax.vmap(foo, in_axes=[1, 0, 0])(xs, ys, zs)
```

Out[177…

```
Array([[ 1.1073166 , -1.2638886 ,  1.6933036 , ...,  0.2811526 ,
        -1.1245667 ,  1.1898308 ],
       [-1.8935366 , -0.90015197,  0.13123298, ...,  1.279871  ,
        -1.0298936 ,  1.0070078 ],
       [ 3.0063605 , -0.02753484, -0.22629711, ..., -1.8671131 ,
         1.5921997 , -1.2252135 ],
       ...,
       [ 1.0368359 ,  0.17157444,  0.36409426, ..., -1.959691  ,
         0.56378484,  0.98173004],
       [-0.26755542, -2.422229  ,  2.686123  , ...,  1.3506397 ,
         2.7109694 , -0.67769927],
       [-0.5237786 , -1.9595429 ,  2.7321339 , ..., -1.4278572 ,
         2.1902704 ,  0.9672695 ]], dtype=float32)
```

Apply `jax.vmap` for the simulate function for the following batch of initial states and control inputs Choose $\Delta t = 0.1$. Use the following values and simulate multiple trajectories using the `jax.vmap` function.

In [178…

```
state_dim = continuous_dynamics.state_dim
control_dim = continuous_dynamics.control_dim
N = 1000
n_time_steps = 50
initial_states = jnp.array(np.random.randn(N, state_dim))
controls = jnp.array(np.random.randn(N, n_time_steps, control_dim))




trajs = jax.vmap(lambda init, u: simulate(runge_kutta_integrator(continuous_dynamic

# plot to visualize all the trajectories
```

(e) `jax.jit` (optional reading)

Bleh! You notice that it takes some time to run it. And if you increased the duration or number of trajectories to simulate, the computation would increase. If only we could compile the code to help reduce computation time. With JAX, you can! We can use the `jax.jit` function that performs just-in-time compilation. JAX will figure out the expected sizes of the input arrays and allocate memory based on that.

There are number of ways to just `jax.jit`, and it can get a bit tricky as your code becomes more complex. Best to read up the JAX documentation for more information. But for relatively simple functions, you can usually just apply `jax.jit` without any fuss, and get significant speedup in your code.

Uncomment the following cells to see the computational benefits of `jax.jit`.

In [179…

```
# # without jitting
# %timeit jax.vmap(simulate, in_axes=[None, 0, 0, None])(discrete_dynamics_rk, init
```

In [180…

```
# # method 1: directly apply jax.jit over the jax.vmap function
# # need to provide the static_argnums argument to the first argument since that is
# sim_jit = jax.jit(jax.vmap(simulate, in_axes=[None, 0, 0, None]), static_argnums=

# # time the run
# %timeit sim_jit(discrete_dynamics_rk, initial_states, controls, jnp.array(dt)).bl
```

In [181…

```
# # method 2: apply jax.jit over the simulate function and then apply jax.vmap
# sim_jit = jax.jit(simulate, static_argnums=0)
# sim_jit_vmap = jax.vmap(sim_jit, in_axes=[None, 0, 0, None])
# %timeit sim_jit_vmap(discrete_dynamics_rk, initial_states, controls, jnp.array(dt
```

In [182…

```
# # Method 3: apply jax.jit over the simulate function during function construction
# @functools.partial(jax.jit, static_argnames=("dynamics"))
# def simulate(dynamics, initial_state, controls, dt):
#     xs = [initial_state]
#     time = 0
#     for u in controls:
#         xs.append(dynamics(xs[-1], u, time))
#         time += dt
#     return jnp.stack(xs)

# sim_jit_vmap = jax.vmap(simulate, in_axes=[None, 0, 0, None])
# %timeit sim_jit_vmap(discrete_dynamics_rk, initial_states, controls, jnp.array(dt
```

Problem 2

We continue to consider the dynamically-extended unicycle model and investigate a way to linearize the dynamics around any state. First, we will perform the linearization analytically, and then leverage modern computation tools which will be incredibly helpful especially if the dynamics are complicated! We can efficiently compute gradients via automatic differentiation. JAX is an automatic differentiation library.

(a) Linearize dynamics analytically

Linearize the dynamics given in Problem 1 part (a) about a point $(\mathbf{x}_0, \mathbf{u}_0)$. That is, for linearized dynamics of the form $\dot{\mathbf{x}} \approx A\mathbf{x} + B\mathbf{u} + C$, give expressions for A, B, and C.

$A = \ldots$

$B = \ldots$

$C = \ldots$

Also code up your analytic expression in `linearize_unicycle_continuous_time_analytic`.

In [183…

```
def linearize_unicycle_continuous_time_analytic(state, control, time):
    '''
    Linearizes the continuous time dynamics of the dynamic unicyle using analytic e
    Inputs:
        state     : A jax.numpy array of size (n,)
        control   : A jax.numpy array of size (m,)
        time      : A real scalar

    Outputs:
        A : A jax.numpy array of size (n,n)
        B : A jax.numpy array of size (n,m)
        C : A jax.numpy array of size (n,1)
    '''
 # Get vals from state
    theta = state[2]
    v = state[3]

    A = jnp.array([
        [0, 0, -v *jnp.sin(theta), jnp.cos(theta)], #wrt xdot
        [0, 0, v *jnp.cos(theta), jnp.sin(theta)], #wrt ydot
        [0, 0, 0, 0], #wrt omega
        [0, 0, 0, 0] #wrt accel
    ])  #cols wrp x, y, theta, v


    B = jnp.array([
        [0, 0], #wrt xdor
        [0, 0], #wrt ydot
        [1, 0], #wrt omega
        [0, 1] #wrt accel
    ]) #cols wrt omega, accel

    C = jnp.array([0,0,0,0])

    return A, B, C
```

(b) Evaluate linearized dynamics (analytic)

Using your answer from 2(a), evaluate A, B, and C for $\mathbf{x}_0 = [0, 0, \frac{\pi}{4}, 2.]^T$ and $\mathbf{u}_0 = [0.1, 1.]^T$. Give your answer to 2 decimal places.

In [184…

```
x0 = np.transpose([0, 0, math.pi/4, 2])
u0 = np.transpose([0.1,1.])
```

```
time = .1
#print(linearize_unicycle_continuous_time_analytic(x0, u0, time))
```

(c) Linearize dynamics using JAX autodiff

Time to test out Jax's autodifferentiation capabilities! JAX has an Autodiff Cookbook that provides more details about the various autodiff functions, forward vs backward autodiff, jacboians, hessians, and so forth. You are strongly encouraged read through it.

Using Jax and its built-in `jax.jacobian` function, fill in the `linearize_autodiff` function that takes in a dynamics function, and a state and control to linearize about, and returns the A, B, and C matrices describing the linearized dynamics. Test your function using the continuous-time dynamics with $\mathbf{x}_0 = [0.0, 0.0, \frac{\pi}{4}, 2.0]^T$ and $\mathbf{u}_0 = [0.1, 1.0]^T$ and use the provided test code to verify that the outputs you get from your function are the same as the values you get from `linearize_unicycle_continuous_time_analytic`.

Side note for the curious: the `jnp.allclose` function tests if all corresponding elements of two arrays are within a small tolerance of each other. When working with finite-precision machine arithmetic, you can almost never test two numbers for exact equality directly, because different rounding errors in different computations very often result in very slightly different values even when the two calculations should theoretically result in the same number. For this reason, real numbers in software (which on almost all modern hardware are represented in IEEE 754 floating-point format) are usually considered to be equal if they are close enough that their difference could be reasonably explained by rounding errors.

In [185…

```
def linearize_autodiff(function_name, state, control, time):
    '''
    Linearizes the any dynamics using jax autodiff.
    Inputs:
        function_name: name of function to be linearized. Takes state, control, and
        state     : A jax.numpy array of size (n,); the state to linearize about
        control   : A jax.numpy array of size (m,); the control to linearize about
        time      : A real scalar; the time to linearize about

    Outputs:
        A : A jax.numpy array of size (n,n)
        B : A jax.numpy array of size (n,m)
        C : A jax.numpy array of size (n,1)
    '''

    func = lambda state, control, time: dynamic_unicycle_ode(state, control, time)
    A = jax.jacobian(func, argnums = 0)(state, control, time)
    B = jax.jacobian(func, argnums = 1)(state, control, time)
    C = jax.jacobian(func, argnums = 2)(state, control, time) #time does not affect


    #print(A,B,C)
    return A, B, C
```

In [186…

```
# test code:
state = jnp.array([0.0, 0.0, jnp.pi/4, 2.])
control = jnp.array([0.1, 1.])
time = 0.

A_autodiff, B_autodiff, C_autodiff = linearize_autodiff(continuous_dynamics, state,
A_analytic, B_analytic, C_analytic = linearize_unicycle_continuous_time_analytic(st

print('A matrices match:', jnp.allclose(A_autodiff, A_analytic))
print('B matrices match:', jnp.allclose(B_autodiff, B_analytic))
print('C matrices match:', jnp.allclose(C_autodiff, C_analytic))
```

```
A matrices match: True
B matrices match: True
C matrices match: True
```

(d) Linearize discrete-time dynamics

Assuming your answer from 2(c) matched 2(b) and that you are convinced of the power of automatic differentiation, use your `linearize_autodiff` function on `discrete_dynamics_euler` and `discrete_dynamics_rk` with $\mathbf{x}_0 = [0.0, 0.0, \frac{\pi}{4}, 2.0]^T$ and $\mathbf{u}_0 = [0.1, 1.0]^T$. (Imagine trying to differentiate the expressions analytically! It would be tedious!)

Let $\Delta t = 0.1$.

In [187…

```
state = jnp.array([0.0, 0.0, jnp.pi/4, 2.])
control = jnp.array([0.1, 1.])
time = 0.1

ARK_autodiff, BRK_autodiff, CRK_autodiff = linearize_autodiff(discrete_dynamics_rk,
ARK_analytic, BRK_analytic, CRK_analytic = linearize_unicycle_continuous_time_analy

AE_autodiff, BE_autodiff, CE_autodiff = linearize_autodiff(discrete_dynamics_euler,
AE_analytic, BE_analytic, CE_analytic = linearize_unicycle_continuous_time_analytic

print('A matrices match:', jnp.allclose(ARK_autodiff, AE_analytic))
print('B matrices match:', jnp.allclose(BRK_autodiff, BE_analytic))
print('C matrices match:', jnp.allclose(CRK_autodiff, CE_analytic))
```

```
A matrices match: True
B matrices match: True
C matrices match: True
```

(e) Applying `vmap` to linearize over multiple points

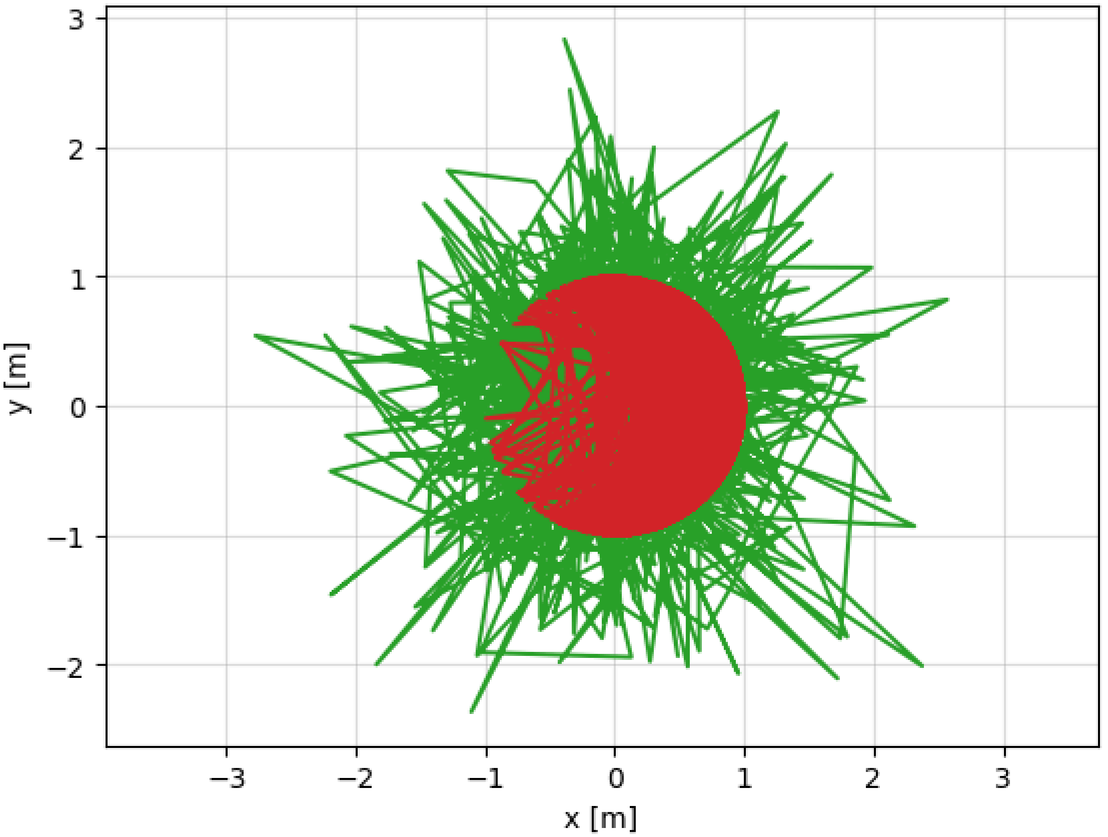
Now, try to linearize your dynamics over multiple state-control values using `vmap`!

In [188…

```
key = jax.random.PRNGKey(42)  # Set a fixed seed
n_samples = 1000
state_dim = 4  # 4-dimensional state
ctrl_dim = 2  # 2-dimensional control

time = 0.0
random_states = jax.random.normal(key, shape=(n_samples, state_dim))
random_controls = jax.random.normal(key, shape=(n_samples, ctrl_dim))

trajs = jax.vmap(lambda init, u: linearize_autodiff(discrete_dynamics_rk, init, u,

plt.plot(trajs[0][:, 0], trajs[0][:, 1])
plt.grid(alpha=0.4)
plt.axis("equal")
plt.xlabel("x [m]")
plt.ylabel("y [m]")
```

Out[188…

```
Text(0, 0.5, 'y [m]')
```



Problem 3 Unconstrained optimization

The goal of this problem is to introduce some basic optimization theory and understanding how automatic differentition can be used for solving unconstrained optimization problems (and essentially powering deep learning libraries under the hood).

In this problem, we will implement a simple version of the gradient descent algorithm to solve an unconstrained optimization problem. Specifically, you will learn about the log-barrier method which is used to cast a *constrained* optimization problem into an unconstrained one which can be easily solved using gradient descent.

Some background on optimization

In mathematics, *optimization* is the process of finding which value of an argument to a function makes the output of that function as high or as low as possible. In other words, given a function $f(x)$ (called the *cost function* or *objective function*), which value of x makes $f(x)$ as high (or as low) as possible?

In controls, we often consider the objective as a *cost* function (e.g., total fuel or control effort, distance traveled) and thus aim to make $f(x)$ as *low* as possible; in other words, we often only consider the problem of minimizing $f(x)$. While in reinforcement learning, typically the objective is viewed as a *reward*, and therefore the aim is to maximize $f(x)$. But note that this is just the same thing as minimizing $-f(x)$, so maximization and minimization actually turn out to be basically the same mathematical problem, but the convention depends on the community.

Optimization is very useful in engineering; for example, if x is a variable that somehow represents the design of an airplane (maybe it's a vector of design parameters such as wing aspect ratio, engine thrust, cargo space, etc), and if we can find a function $f(x)$ that computes the fuel consumption of the final airplane as a function of the design parameters, than finding x which minimizes $f(x)$ amounts to designing the most fuel-efficient airplane possible under the design assumptions. We often denote the value of x which optimizes $f(x)$ as x^*, and the optimal value of $f(x)$ is then $f(x^*)$.

You may remember from undergrad calculus that we can minimize $f(x)$ by taking its derivative $f'(x)$, setting $f'(x^*) = 0$, and solving for $^*x$. If $f(x)$ is scalar-valued but has a vector-valued input, we would generalize this to taking the *gradient* $\nabla f(x)$, setting to zero, and solving for x^*. However, in many practical applications $\nabla f(x)$ is far too complicated to solve $\nabla f(x^*) = 0$ analytically; just think how complicated our fuel consumption function in the example above would be. Thus, in practice we often use iterative numerical optimization algorithms such as *gradient descent*.

Gradient descent (for a scalar function of a single scalar argument) works as follows. First, assume some initial guess x_0. Then, iteratively improve successive guesses x_k like so:

$$x_{k+1} = x_k - \alpha \nabla f(x_k),$$

where α is some small positive fixed "step size" parameter which we choose. This has the effect of moving our successive guesses "down the hill" of the function $f(x)$. This process is repeated until successive iterates converge to some specified convergence tolerance ϵ:

$$|x_{k+1} - x_k| < \epsilon \implies \text{stop iterating.}$$

The final x iterate we take to be our optimal x^*.

You may notice that your solution, or whether you may converge, depends on your choice of step size. There are techniques to choose and adapt the step size. We won't discuss this further in the course, but ADAM is a popular approach.

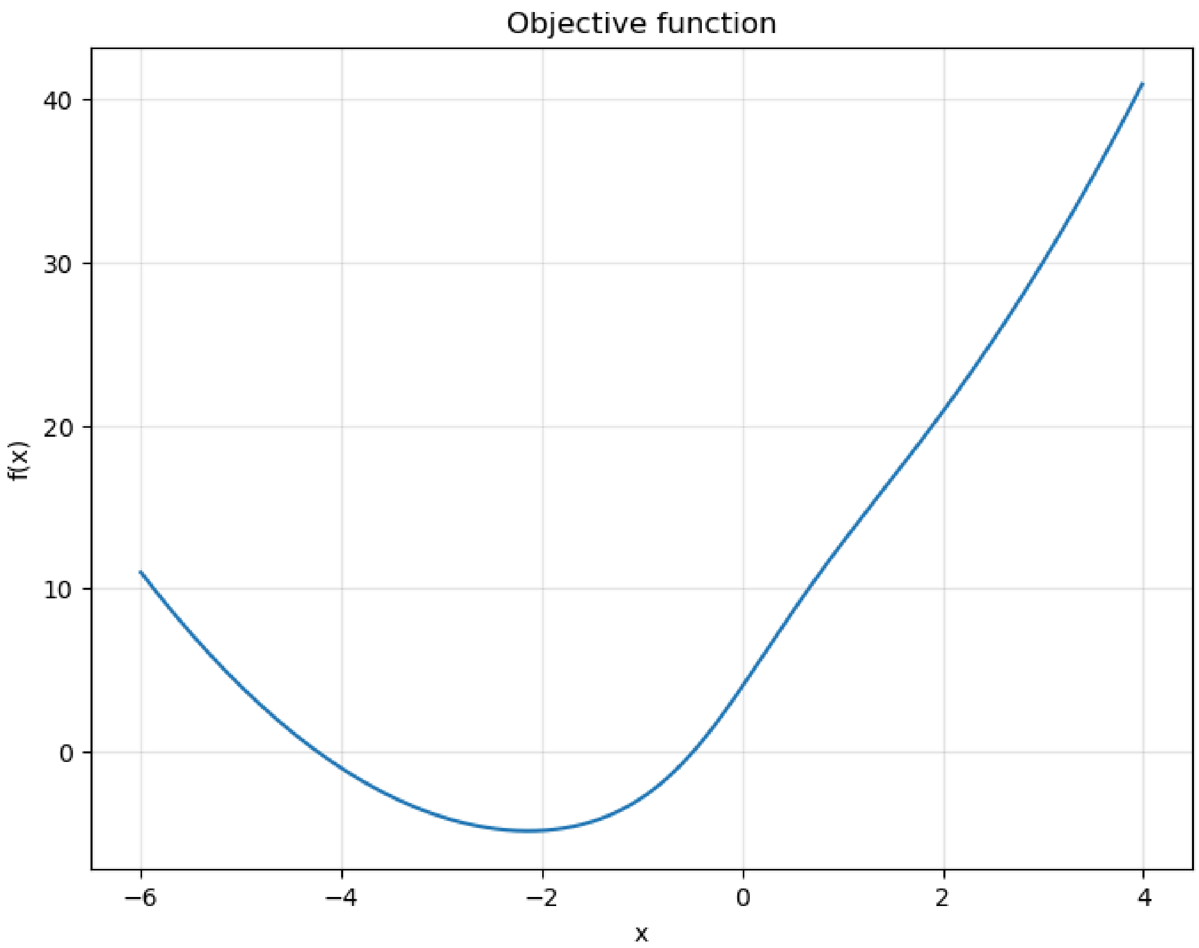
(a) Gradient descent on unconstrained optimization problem

Consider the objective function $f(x) = (x+2)^2 + 5\tanh(x)$, defined in code and plotted below. The value of x which minimizes $f(x)$ is $x^* = -2.13578$, and $f(x^*) = -4.84389$. In the next code cell, fill out the `minimize_with_gradient_descent` function; you can use Jax's autodiff capabilities to compute $f'(x)$. The starter code in that cell will use your gradient descent function with an initial guess of $x = 5$, a step size of 0.1, and a convergence tolerance of 10^{-8} to minimize $f(x)$; verify that your algorithm does indeed find the correct minimizing x^* and minimal value $f(x^*)$.

Fun fact: with $f(x) = (x+2)^2 + 5\tanh(x)$, then $f'(x)$ is a transcendental function, and $f'(x) = 0$ *can't* be solved algebraically. If your gradient descent code worked, congratulations! You've written code that numerically solved a math problem in milliseconds that can't be solved analytically at all.

In [189…

```
def f(x):
    return (x + 2)**2 + 5*jnp.tanh(x)

args = np.arange(-6,4,0.01)
plt.figure(figsize=(8,6))
plt.plot(args, f(args))
plt.xlabel('x')
plt.ylabel('f(x)')
plt.title('Objective function')
plt.grid(alpha=0.3)
plt.show()
```

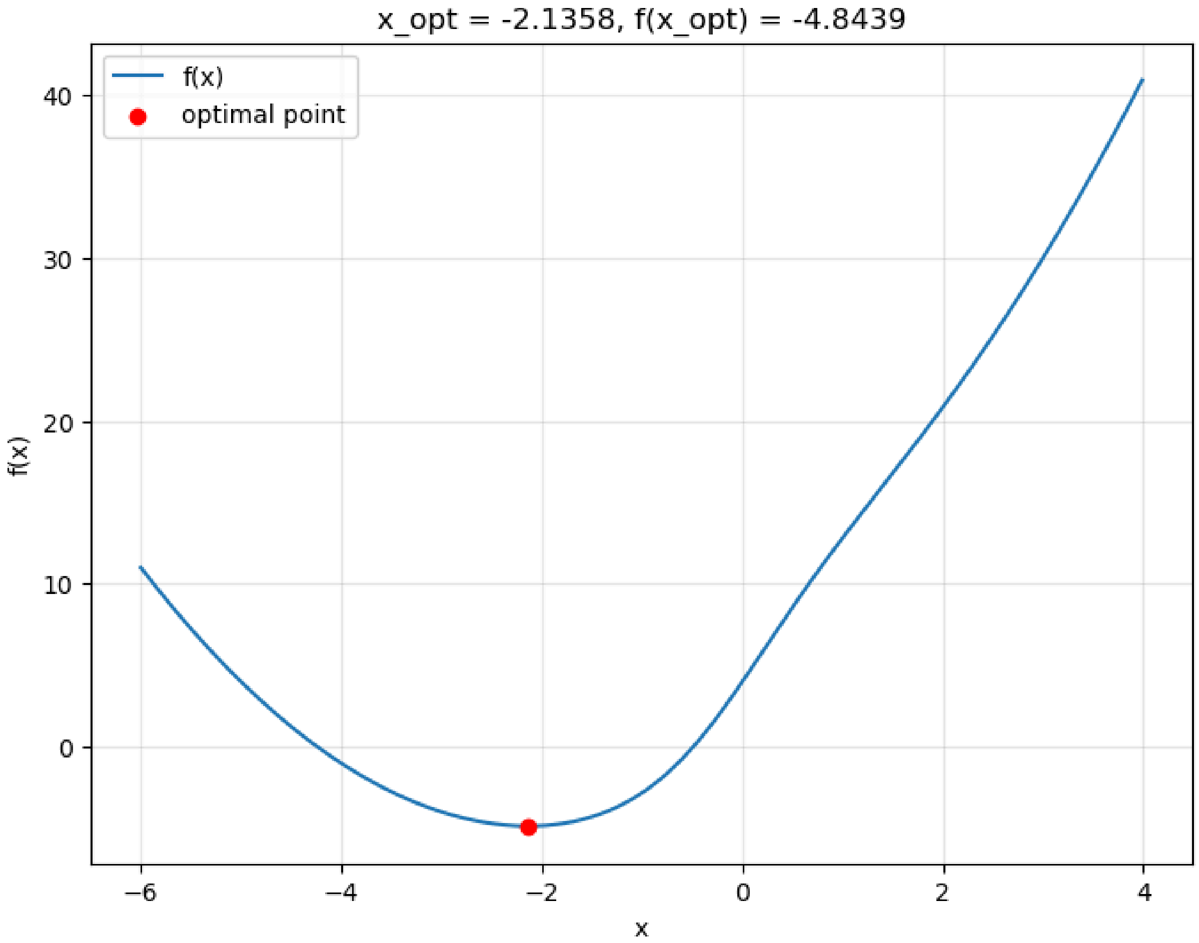


In [190…

```python
def minimize_with_gradient_descent(func, initial_guess, step_size, convergence_tol=
    '''
    Minimizes a scalar function of a single variable.
    Inputs:
        func                : name of function to be optimized. Takes initial_guess a
        initial_guess       : a real number
        convergence_tol     : convergence tolerace; when current and next guesses of
                              together than this, algorithm terminates and returns cu

    Outputs:
        cur_x : current best estimate of x which minimizes f(x)
    '''

    next_x = cur_x = initial_guess  #init
    current_tol = convergence_tol
    deriv_func = grad(func)
    while current_tol >= convergence_tol:

        next_x = cur_x - step_size * deriv_func(cur_x)
        current_tol = abs(cur_x - next_x) # calculate new tol
        cur_x = next_x # update curr

    return cur_x

x_opt = minimize_with_gradient_descent(f, 5.0, 0.1)
```

```
# output and plot:
print('optimal x:', x_opt)
print('optimal value of f(x):', f(x_opt))

args = np.arange(-6,4,0.01)
plt.figure(figsize=(8,6))
plt.plot(args, f(args), label='f(x)')
plt.scatter(x_opt, f(x_opt), zorder=2, color='red', label='optimal point')
plt.title('x_opt = {:.4f}, f(x_opt) = {:.4f}'.format(x_opt, f(x_opt)))
plt.grid(alpha=0.3)
plt.xlabel('x')
plt.ylabel('f(x)')
plt.legend()
plt.show()
```

```
optimal x: -2.135782
optimal value of f(x): -4.8438854
```



(b) Applying log-barrier for solving constrained optimization problems

Now suppose we want to add a *constraint* to our optimization problem; that is, a restriction on the set of x values we'll consider acceptable. In particular, in this case we'll say that we only want values of x such that, for some specified *constraint function* $g(x)$, we have $g(x) < 0$; this is called an *inequality constraint*.

The gradient descent algorithm described above has no way to enforce a constraint like this; x is allowed to wander wherever the gradient takes it. Thus we must modify the algorithm to allow it to find constrained optima under inequality constraints. There are several ways to do this, but in this problem we will be using the *log-barrier method*.

In this method, we construct from the objective function $f(x)$ a different objective function $\phi(x)$ that has the following properties:

- When x is such that $g(x)$ is far away from 0, then $\phi(x)$ must approximate $f(x)$, so that minimizing $\phi(x)$ approximately minimizes $f(x)$.
- $\phi(x)$ must grow to infinity as $g(x)$ approaces 0. This will prevent x from crossing over the boundary of $g(x) < 0$.

To accomplish these goals, we construct $\phi(x)$ like so:

$$\phi(x) = f(x) - t\ln(-g(x)),$$

where t is a weighting parameter that we choose.

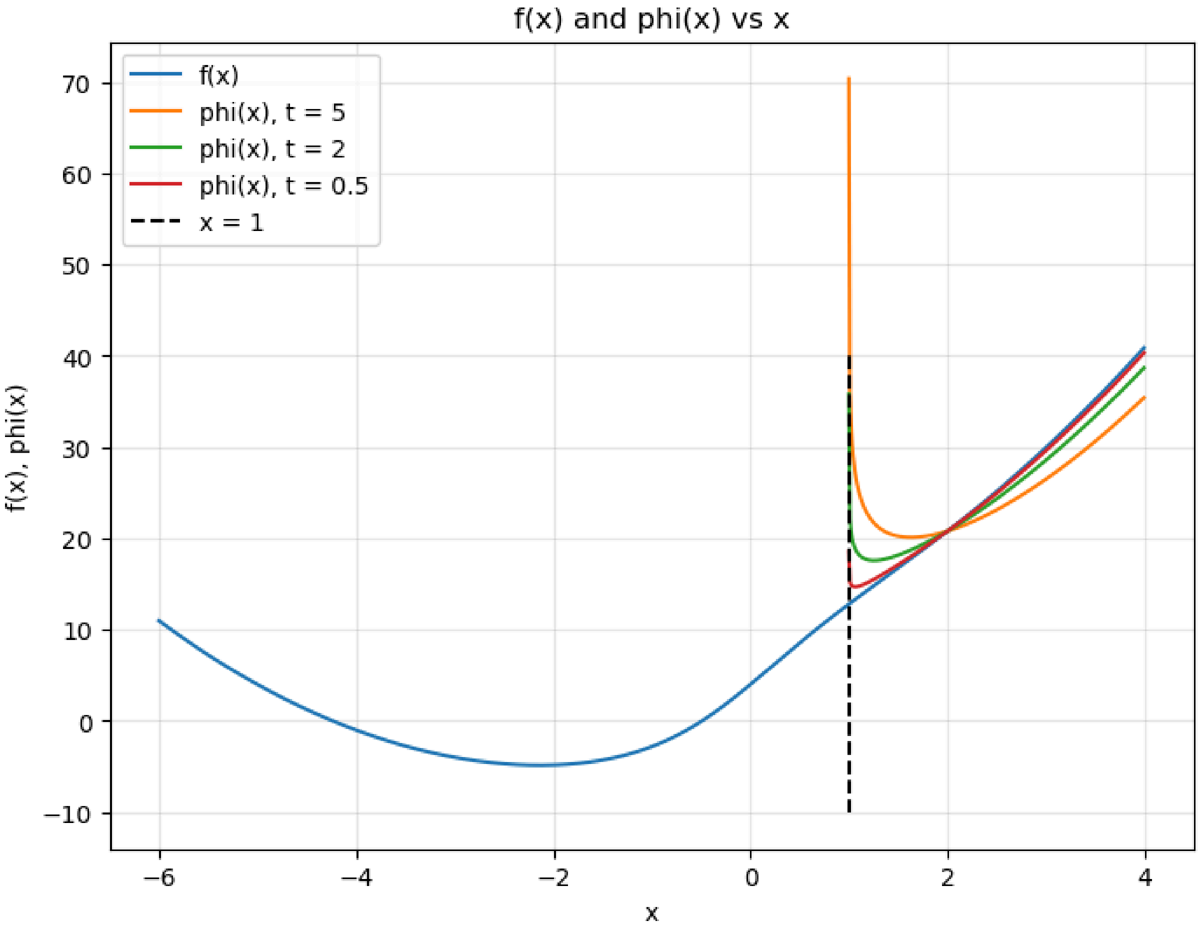
Suppose we want to minimize the same objective function as before, $f(x) = (x+2)^2 + 5\tanh(x)$, but now we want a constraint $x > 1$. In the code block below, fill out the functions `g` and `phi`. The starter code will plot $f(x)$, as well as $\phi(x)$ for $t = 0.5,\ 2,\ 5$. Comment on how $\phi(x)$ changes with changing t. Note that since $\ln(y)$ is not defined (in the real numbers) for $y \leq 0$, the domain of $\phi(x)$ is restricted to $x > 1$.

Hint: for your `phi` function to work well with the next parts of this problem, be sure to use `jnp.log` instead of `np.log`.

In [191…

```python
# fill out g(x) so that the statement g(x) < 0 is equivalent to the statement x > 1
def g(x):
    return 1 - x

def phi(f, x, g, t):
    '''
    Computes phi(x).
    Inputs:
        f  : name of f(x) function; takes x as input
        x  : variable to be optimized
        g  : constraint function; we want to ensure g(x) <= 0
        t  : log-barrier weighting parameter

    Outputs:
        phi(x)
    '''

    phi_x = f(x) - t*jnp.log(-g(x))

    return phi_x
```

```
x_upper = 4
dx = 0.01
f_x_domain = np.arange(-6, x_upper, dx)
phi_x_domain = np.arange(1.00001, x_upper, dx)

plt.figure(figsize=(8,6))
plt.plot(f_x_domain, f(f_x_domain), label='f(x)')
plt.plot(phi_x_domain, phi(f, phi_x_domain, g, 5), label='phi(x), t = 5')
plt.plot(phi_x_domain, phi(f, phi_x_domain, g, 2), label='phi(x), t = 2')
plt.plot(phi_x_domain, phi(f, phi_x_domain, g, 0.5), label='phi(x), t = 0.5')
plt.vlines(1, -10, 40, linestyles='dashed', label='x = 1', color='black')
plt.xlabel('x')
plt.grid(alpha=0.3)
plt.ylabel('f(x), phi(x)')
plt.title('f(x) and phi(x) vs x')
plt.legend(loc='upper left')
# plt.ylim(-10, 40)
plt.show()
```



(c)

Now, we can use our `minimize_with_gradient_descent` function to minimize $\phi(x)$ instead of $f(x)$. Do so below, for a few different values of t, and comment on the results. The true constrained optimal point is $x^* = 1$, $f(x^*) = 12.808$.

- Does the log-barrier method find the true constrained optimal point?

The lower the barrier weight, the closer we get to the barrier. However, if the step size isnt sufficently small, the barrier could be passed and the program will run indefinitely, unable to return to the safe area.

- How does the point found by the log-barrier method change as you change t?

As t decreases, the point found by the log-barrier method will get closer and closer to the boundary.That is why, as you stated, a small t-value needs a small step-size, as there is greater risk of passing the barrier. High t-values are less risky of passing the barrier, and in term are less accurate/farther away from the barrier, if that is where the true value lies.

You may find that as you decrease t, you also need to decrease the gradient descent step size to get good behavior (in particular, to avoid "overshooting" the constraint and finding an x^* estimate which does not respect the constraint). Also, for the log-barrier method to work, your initial guess must itself respect the constraint (in this case, your initial guess must be greater than one).

Hint: the `minimize_with_gradient_descent` function requires an objective function that takes only x as an argument: $f(x)$. But our `phi` Python function defined above takes `f`, `x`, `g`, and `t`; you may find Python's "lambda function" capabilities useful or the `functools.partial`.

In [234…

```
# ========================== hint: lambdas ==============================

def hint_func(arg1, arg2, arg3):
    return arg1 + 2 * arg2 + arg3

def hint_func_caller(func, arg):
    return func(arg)

foo = 42
bar = 100

lambda x: hint_func(foo, x, bar) # this essentially defines a function of x only, t

# hint_func_caller expects a function that takes only one argument. But we can use
# "prepopulate" all but one argument of hint_func, "turning it into" an argument of

hint_func_caller(lambda x: hint_func(foo, x, bar), 5) # this will work and give 152

# ^^^^^^^^^^^^^^^^^^^^^^^^^^ hint: lambdas ^^^^^^^^^^^^^^^^^^^^^^^^^^

# ========================== hint: functools.partial ==============================

new_func = functools.partial(hint_func, foo, arg3=bar) # this is equivalent to lamb
new_func(5) # this will give 152

# OR

new_func = functools.partial(hint_func, arg1=foo, arg3=bar) # this is equivalent to
```

```
new_func(arg2=5) # this will give 152

# ^^^^^^^^^^^^^^^^^^^^^^^^^^ hint: functools.partial ^^^^^^^^^^^^^^^^^^^^^^^^^^

# f, x, g, t
x = 5.0
t = .001 #barrier weight
step_size = .001
#new_func = functools.partial(minimize_with_gradient_descent, func=f, initial_guess
#print(new_func(step_size=t))

new_phi = lambda x: phi(f, x, g, t)
xbar_opt = minimize_with_gradient_descent(new_phi, initial_guess= x, step_size=step
print(xbar_opt, f(xbar_opt))
```

```
-2.1358607 -4.8438864
```

Problem 4 (`cvxpy`)

In this problem we will explore the basics of `cvxpy`, a Python package for solving convex optimization problems. `cvxpy` has a good tutorial here, so read that page before proceeding with this problem (the section on "parameters", while useful, is not important for this homework, so consider that section optional for now).

Consider a vector variable $x = [x_1, x_2, x_3]^T$. Use `cvxpy` to compute the minimizer of the following objective function

$$x_1^2 + 2x_2^2 + 3.5x_3^2,$$

subject to the constraint

$$\begin{bmatrix} 0.707 & 0.707 & 0 \\ -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{bmatrix} x \le \begin{bmatrix} 2 \\ -1 \\ -1 \\ -3 \end{bmatrix}.$$

Print the optimal x and the optimal value of the objective function.

Notice how much easier it is to use `cvxpy` than to write our own optimization algorithm from scratch!

In [202…

```
A = np.array([
    [0.707, 0.707, 0],
    [-1, 0, 0],
    [0, -1, 0],
    [0, 0, -1]
    ])

b = np.array([2, -1, -1, -3])
X = cp.Variable(3)
```

```
constraints = [A @ X <= b]

# Minimizes objective function within Problem
prob = cp.Problem(cp.Minimize(X[0]**2 + 2 * X[1]**2 + 3.5 * X[2]**2), constraints)
prob.solve()
print(X.value, prob.value)
```

```
[1. 1. 3.] 34.5
```

Problem 5 (Control Barrier Function and CBF-QP)

In this problem, using the concepts and code you have developed in the previous problems, you will implement a CBF-QP safety filter. We will restrict ourselves to a simple 1D system so that you can verify your results analytically if needed, but note that the theory extends to higher dimensional problems, and try to keep your code general so that it could still work on a different system and choice of CBF.

Consider the following 1D single integrator dynamics $\dot{x} = u$ and let $b(x) = x^2 - 1$.

(a) Computing the Lie derivative

What is the expression for $\nabla b(x)^T f(x, u)$?

$\nabla b(x) = 2x$

$f(x, u) = u$

$\nabla b(x)^T f(x, u) = 2x * u$

(b) Solving the CBF-QP

Suppose that your desired control is $u_{\text{des}} = 0.5$, which is to move in the positive x-direction at constant velocity. But the safety filter enforces that $x^2 \geq 1$, i.e., keep a 1 unit away from the origin.

The CBF safety filter essentially chooses a control input that is as close to u_{des} as possible while satisfying the CBF inequality constraint.

$$u_{\text{safe}}(x) = \underset{u}{\operatorname{argmax}} \|u - u_{\text{des}}\|_2^2 \;\; \text{subj. to} \;\; \nabla b(x)^T f(x, u) \geq -\alpha(b(x))$$

This is referred to as the CBF-QP since it is a quadratic program (quadratic objective and linear constraints). For simplicity, we assume there are no other constraints on controls.

Let $\alpha(z) = az$, $a = 0.5$ Use `cvxpy` to solve the CBF-QP for $x = -3$, $x = -2$ and $x = -1.1$.

Report the corresponding safe control values.

Hint: You may find the `Parameter` variable in `cvxpy` helpful. It allows you to update certain parameters without needing to reconstruct the optimization each time.

In [204…

```
x = cp.Parameter(nonpos = True)
u = cp.Variable()

u_des = 0.5
a = 0.5 #constraint decentivization

b = lambda x_val: x_val**2 - 1 #b(x) equation
alpha_func = lambda b_val: a * b_val

constraint = [2 * x * u >= -alpha_func(b(x))]

prob = cp.Problem(cp.Minimize((u - u_des)**2), constraint)

x_vals = [-3, -2, -1.1]

for ele in x_vals:
    x.value = ele
    prob.solve()
    print(x.value, u.value)
```

```
-3 0.5000000000000001
-2 0.375
-1.1 0.04772727272727264
```

(c) Applying the CBF safety filter

Now, simulate the system starting from $x = -5$ and with $u_{\text{des}} = 0.5$, but the CBF safety filter is applied. Plot the state and control sequence for $a = 2$, $a = 1$, $a = 0.5$ and $a = 0.1$ where $\alpha(z) = az$. Comment on how the trajectory changes as a changes. What is your interpretation of a?

It appears that the larger "a" is, the more agressive the filter is. With higher a values, the change in the control value is sharper and the state values beeline toward the safe region. Higher a values mean the system will prioritize moving toward the safe region versus the desired input.

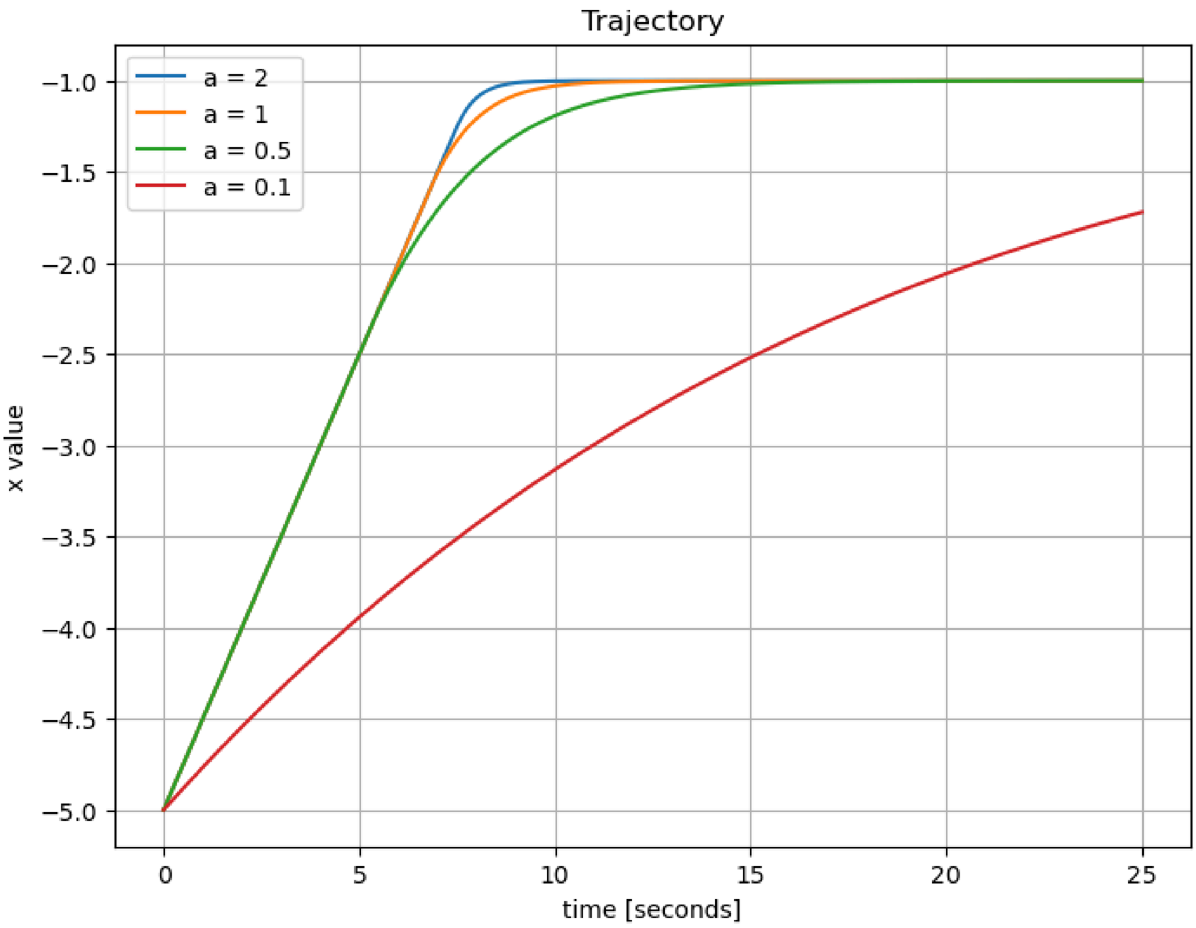
Use $\Delta t = 0.05$ and simulate for 500 steps.

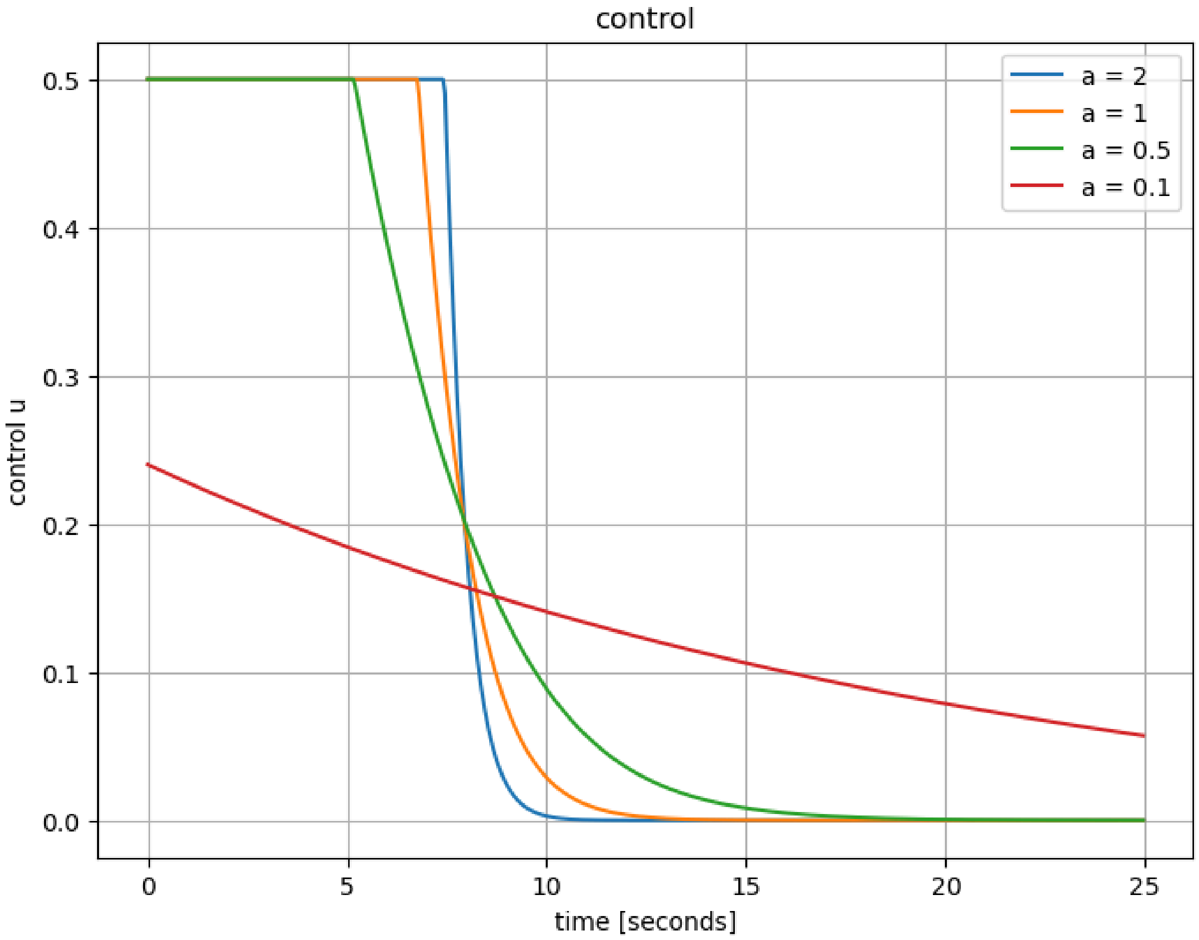
Note that the CBF theory is for continuous-time dynamics, but when we simulate, we use discrete-time dynamics. There are some practical issues that we need to be careful about (see this paper for more details), but right now, let's stick to using a reasonably small time step.

In [228…

```
u_des = .5
dt = 0.05
steps = 500
x0 = -5
a_vals = [2, 1, 0.5, 0.1]
```

```
trajs = {} #dictionary


for a in a_vals:
    x_param = cp.Parameter(nonpos=True)  # x must be nonpositive for CBF
    u = cp.Variable()

    x_vals = [x0]
    u_vals = []

    for i in range(steps):
        x_param = cp.Parameter(nonpos=True)
        u = cp.Variable()

        # iterate through the x_vals
        x_val = x_vals[-1]
        x_param.value = x_val

        b_func = cp.square(x_param) - 1
        alpha_func = a * b_func
        constraint = [2 * x_param * u >= -alpha_func]

        prob = cp.Problem(cp.Minimize((u - u_des)**2), constraint)
        prob.solve()

        u_val = u.value #store u_vals for
        u_vals.append(u_val)

        x_next = x_val + dt * u_val #get next x_val, Euler int.
        x_vals.append(x_next)

    trajs[a] = (np.array(x_vals), np.array(u_vals)) #storing/connecting the x_vals

# Plot results
time = np.arange(steps + 1) * dt #time steps 0 through 5.00s

# Plot x(t)
plt.figure(figsize=(8,6))
for a in a_vals:
    plt.plot(time, trajs[a][0], label=f'a = {a}')
    plt.ylabel('x value')
    plt.xlabel('time [seconds]')
    plt.title('Trajectory')
    plt.legend()
    plt.grid(True)
plt.show()


# Plot u(t)
plt.figure(figsize=(8,6))
for a in a_vals:
    plt.plot(time[:-1], trajs[a][1], label=f'a = {a}')
    plt.ylabel('control u')
    plt.xlabel('time [seconds]')
    plt.title('control')
    plt.legend()
    plt.grid(True)
plt.show()



    # simulate system, during which the CBF-QP safety filter is applied.
    # loop over various values of a.
```




control
control u
time [seconds]
0.0
0.1
0.2
0.3
0.4
0.5
0
5
10
15
20
25
a = 2
a = 1
a = 0.5
a = 0.1