how can i formulate mpc for a quadruped if my inputs are angles

To formulate an MPC for a quadruped robot using joint angles as inputs, you'll need to bridge kinematic planning with dynamic feasibility. Here's a structured approach:

1. Kinematic Model for Foot Placement

Define the relationship between joint angles and foot positions using forward kinematics. For each leg i:

$$\mathbf{p}_i = f_{\text{FK}}(\mathbf{q}_i)$$

where $\mathbf{q}_i = [q_1, q_2, q_3]^T$ are the joint angles, and $\mathbf{p}_i$ is the foot position in the body frame.
Example: For a 3-DOF leg, $\mathbf{p}_i$ depends on link lengths and trigonometric functions of $\mathbf{q}_i$.

2. Simplified Dynamic Model

Use a **single rigid body (SRB) model** to approximate the robot's dynamics. The body's acceleration depends on ground reaction forces (GRFs) $\mathbf{f}_i$ at each foot:

$$m\ddot{\mathbf{c}} = \sum_{i=1}^{4} \mathbf{f}_i + m\mathbf{g}, \quad \dot{\mathbf{L}} = \sum_{i=1}^{4} (\mathbf{r}_i \times \mathbf{f}_i)$$

where $\mathbf{r}_i = \mathbf{p}_i - \mathbf{c}$ is the foot position relative to the center of mass (CoM) $\mathbf{c}$.

3. Mapping Joint Angles to GRFs

Assume a spring-damper relationship between foot displacement and GRFs (e.g., virtual impedance control):

$$\mathbf{f}_i = -k_p(\mathbf{p}_i - \mathbf{p}_i^{\text{ref}}) - k_d\dot{\mathbf{p}}_i$$

Here, $\mathbf{p}_i^{\text{ref}}$ is the reference foot trajectory, and k_p/k_d are stiffness/damping gains. This allows GRFs to be indirectly controlled via joint angles[12] [13].

4. MPC Formulation

Objective Function

Minimize tracking errors for CoM position $\mathbf{c}$, orientation $\mathbf{R}$, and joint angles $\mathbf{q}$:

$$J = \sum_{k=1}^{N} \left(\|\mathbf{c}_k - \mathbf{c}_{\text{ref},k}\|^2 + \|\log(\mathbf{R}_k^T \mathbf{R}_{\text{ref},k})\|^2 + \|\mathbf{q}_k - \mathbf{q}_{\text{ref},k}\|^2 \right)$$

Constraints

- **Kinematic feasibility**: Foot positions must stay within workspace limits.
- **Friction cones**: GRFs must satisfy $\sqrt{f_x^2 + f_y^2} \leq \mu f_z$.
- **Joint limits**: $\mathbf{q}_{\min} \leq \mathbf{q} \leq \mathbf{q}_{\max}$.

State-Space Model

Discretize the SRB dynamics and linearize around the current state for a QP-compatible form[14] [13]:

$$\mathbf{x}_{k+1} = \mathbf{A}\mathbf{x}_k + \mathbf{B}\mathbf{u}_k + \mathbf{d}$$

where $\mathbf{u}_k = \mathbf{q}_k$ (joint angles) and $\mathbf{x} = [\mathbf{c}, \dot{\mathbf{c}}, \mathbf{R}, \boldsymbol{\omega}]^T$.

5. Implementation Steps

1. **Trajectory Generation**: Plan reference CoM $\mathbf{c}_{\text{ref}}$ and foot $\mathbf{p}_i^{\text{ref}}$ trajectories.
2. **QP Setup**: Use tools like `OSQP` or `qpOASES` to solve the MPC problem.
3. **Low-Level Control**: Send optimal $\mathbf{q}_k$ to joint-level PD controllers for tracking.

Example Code Snippet (Python)

```
# Simplified MPC loop for joint angles
for each MPC step:
    # Get current state (CoM, orientation, joint angles)
    x = get_robot_state()

    # Solve QP: minimize J subject to constraints
    q_opt = solve_mpc_qp(x, c_ref, R_ref, q_ref)

    # Send joint angles to PD controllers
    send_joint_commands(q_opt)
```

Key Considerations

- **Linearization**: Use variation-based linearization for rotation matrices to avoid singularities[14].
- **Real-Time Optimization**: Limit the prediction horizon (e.g., 10–20 steps) for tractability.
- **Hierarchical Control**: Combine with a low-level impedance controller for robust force tracking[12] [13].

This approach enables dynamic locomotion while respecting joint angle constraints, leveraging simplified models for real-time performance.

⁂

<q>d</q> ?

The query "<q>d</q>" is ambiguous and lacks context. If you are referring to "d" in the context of robotics, it could represent various concepts depending on the specific domain:

Possible Interpretations:

1. **Joint Displacement (Prismatic Joint)**:
   In robotics, "d" often refers to the displacement parameter for prismatic joints in a robot's configuration. For example, in the Denavit-Hartenberg (DH) convention, "d" represents the distance along a joint axis [15] [16].
2. **State Space Dynamics**:
   "d" might represent a disturbance vector or offset in state-space models, such as $ \mathbf{d} $ in discrete-time system dynamics:
   $$x[n+1] = Ax[n] + Bu[n] + d$$
   Here, $ d $ accounts for external forces or model inaccuracies [17].
3. **Distance or Dimension**:
   In kinematics, "d" could refer to a spatial distance or dimensional parameter used in trajectory planning or workspace analysis [18] [19].
4. **Quaternion Derivative**:
   In attitude dynamics, "qd" may represent the derivative of a quaternion used to describe rotational motion [20].

If you provide more context (e.g., robotics terminology, mathematical modeling, or control systems), I can refine the explanation further.

⁂

how do i find a and b matrices for above

To find the **A** (state transition) and **B** (input control) matrices for the quadruped MPC formulation with joint angles as inputs, follow these steps:

1. Define State and Input Vectors

- **State vector** ($\mathbf{x}$):
  $$\mathbf{x} = \begin{bmatrix} \mathbf{c} \\ \dot{\mathbf{c}} \\ \boldsymbol{\theta} \\ \boldsymbol{\omega} \end{bmatrix}$$
  where:
  - $\mathbf{c}$: CoM position (3D)

- $\dot{\mathbf{c}}$: CoM velocity (3D)
  - $\boldsymbol{\theta}$: Body orientation (Euler angles or exponential coordinates, 3D)
  - $\boldsymbol{\omega}$: Angular velocity (3D)
- **Input vector** ($\mathbf{u}$):
  $$\mathbf{u} = \mathbf{q} = [q_1, q_2, \ldots, q_{12}]^T$$
  (12 joint angles for a quadruped with 3-DOF legs).

2. Linearize the Dynamics

The single rigid body (SRB) dynamics are nonlinear. Use **Taylor series expansion** to linearize around the current state $\mathbf{x}_0$ and input $\mathbf{u}_0$:

$$\dot{\mathbf{x}} \approx \underbrace{\frac{\partial \dot{\mathbf{x}}}{\partial \mathbf{x}}}_{\mathbf{A}_c}\mathbf{x} + \underbrace{\frac{\partial \dot{\mathbf{x}}}{\partial \mathbf{u}}}_{\mathbf{B}_c}\mathbf{u} + \text{constant terms}$$

Key Partial Derivatives:

- **CoM acceleration**:
  From $m\ddot{\mathbf{c}} = \sum \mathbf{f}_i + m\mathbf{g}$, compute $\frac{\partial \ddot{\mathbf{c}}}{\partial \mathbf{q}}$ using the chain rule:
  $$\frac{\partial \ddot{\mathbf{c}}}{\partial \mathbf{q}} = \frac{1}{m}\sum_{i=1}^{4} \frac{\partial \mathbf{f}_i}{\partial \mathbf{p}_i}\frac{\partial \mathbf{p}_i}{\partial \mathbf{q}}$$
  where $\mathbf{f}_i = -k_p(\mathbf{p}_i - \mathbf{p}_i^{\text{ref}}) - k_d\dot{\mathbf{p}}_i$ (spring-damper model).
- **Orientation dynamics**:
  For small angles, $\dot{\boldsymbol{\theta}} \approx \boldsymbol{\omega}$, and $\dot{\boldsymbol{\omega}} = \mathbf{I}^{-1}(\mathbf{r}_i \times \mathbf{f}_i - \boldsymbol{\omega} \times \mathbf{I}\boldsymbol{\omega})$.

3. Discretize the System

Convert continuous-time matrices $\mathbf{A}_c, \mathbf{B}_c$ to discrete-time $\mathbf{A}, \mathbf{B}$ using **zero-order hold** (ZOH) with timestep Δt:

$$\mathbf{A} = e^{\mathbf{A}_c \Delta t}, \quad \mathbf{B} = \int_0^{\Delta t} e^{\mathbf{A}_c \tau}\mathbf{B}_c \, d\tau$$

For small Δt, approximate:

$$\mathbf{A} \approx \mathbf{I} + \mathbf{A}_c \Delta t, \quad \mathbf{B} \approx \mathbf{B}_c \Delta t$$

4. Example: 2D Case

Assume planar motion (x, z, pitch) for simplicity:

- **State**: $\mathbf{x} = [c_x, c_z, \theta_y, \dot{c}_x, \dot{c}_z, \omega_y]^T$
- **Input**: $\mathbf{u} = [q_1, q_2, q_3, q_4]^T$ (hip and knee angles for 2 legs).

Linearized Matrices:

$$
\mathbf{A}_c = \begin{bmatrix} 0 & amp; 0 & amp; 0 & amp; 1 & amp; 0 & amp; 0 \\ 0 & amp; 0 & amp; 0 & amp; 0 & amp; 1 & amp; 0 \\ 0 & amp; 0 & amp; 0 & amp; 0 & amp; 0 & amp; 1 \\ 0 & amp; 0 & amp; 0 & amp; 0 & amp; 0 & amp; 0 \\ 0 & amp; 0 & amp; 0 & amp; 0 & amp; 0 & amp; 0 \\ 0 & amp; 0 & amp; 0 & amp; 0 & amp; 0 & amp; 0 \end{bmatrix}, \quad \mathbf{B}_c = \begin{bmatrix} 0 & amp; 0 & a \\ 0 & amp; 0 & a \\ 0 & amp; 0 & a \\ \frac{\partial \ddot{c}_x}{\partial q_1} & amp; \cdots & am \\ \frac{\partial \ddot{c}_z}{\partial q_1} & amp; \cdots & am \\ \frac{\partial \dot{\omega}_y}{\partial q_1} & amp; \cdots & am \end{bmatrix}
$$

5. Practical Implementation

1. **Symbolic Differentiation**: Use tools like `CasADi` or `SymPy` to compute $\mathbf{A}_c, \mathbf{B}_c$ automatically.
2. **Numerical Linearization**: Perturb states/inputs slightly and compute finite differences:
$$
\mathbf{A}_{ij} = \frac{f_i(\mathbf{x} + \delta \mathbf{e}_j, \mathbf{u}) - f_i(\mathbf{x}, \mathbf{u})}{\delta}
$$
3. **Code Example** (Python with CasADi):

```
import casadi as cs

# Define state and input variables
x = cs.MX.sym('x', 12)  # 12D state
u = cs.MX.sym('u', 12)  # 12D joint angles

# Define continuous-time dynamics x_dot = f(x, u)
f_continuous = ...  # SRB dynamics + kinematic model

# Compute Jacobians
A_c = cs.jacobian(f_continuous, x)
B_c = cs.jacobian(f_continuous, u)

# Discretize (Euler method)
dt = 0.1  # Timestep
A = cs.MX.eye(12) + A_c * dt
B = B_c * dt
```

Key Considerations:

- **Input Mapping**: Joint angles ($\mathbf{u}$) indirectly affect dynamics via GRFs. Ensure your $\mathbf{B}$ matrix accounts for the full chain: $\mathbf{q} \to \mathbf{p}_i \to \mathbf{f}_i \to \ddot{\mathbf{c}}, \dot{\boldsymbol{\omega}}$.
- **Iterative Linearization**: Recompute $\mathbf{A}, \mathbf{B}$ at each MPC iteration for accuracy.
- **Convexity**: Enforce linear constraints (e.g., friction cones) to maintain QP feasibility.

elaborate on that python code by defining f_continuous. Use the attached file as a reference for the kinematics

I'll analyze the provided `petoi_kinematics.py` code and define the continuous-time dynamics function `f_continuous` for MPC integration, focusing on the kinematic relationships.

Key Components from Provided Code

1. **Kinematic Parameters**:

```
self.body_length = 10.5  # cm (a)
self.body_width = 10     # cm (b)
self.leg_length = 4.6    # cm (c)
self.foot_length = 4.6   # cm (d)
```

2. **Transformation Matrices**:

```
# Body-fixed frame to leg base frames
self.T01_front = np.array([[cg, sg, -self.a/2*cg], [-sg, cg, self.a/2*sg], [0, 0, 1]])
self.T01_back = np.array([[cg, sg, self.a/2*cg], [-sg, cg, -self.a/2*sg], [0, 0, 1]])
```

3. **Inverse Kinematics**:

```
def leg_ik(self, xz: np.array):
    # Returns joint angles (α, β) from foot positions
```

Continuous-Time Dynamics Formulation

```
import casadi as cs

def create_dynamics_function(petoi: PetoiKinematics):
    # State: [c_x, c_z, θ, ẋ, ż, ω] (CoM position, pitch angle, velocities)
    x = cs.MX.sym('x', 6)

    # Input: Joint angles [α1, β1, ..., α4, β4]
    u = cs.MX.sym('u', 8)

    # Extract body orientation from state
    theta = x[^6_2]
    cg, sg = cs.cos(theta), cs.sin(theta)

    # Forward kinematics for all feet
    foot_positions = []
    for i in range(4):
        alpha = u[i*2]
        beta = u[i*2+1]
```

```
        # From leg_ik inverse kinematics, reversed
        L = petoi.leg_length + petoi.foot_length * cs.sin(beta)
        x_foot = L * cs.sin(alpha)
        z_foot = -L * cs.cos(alpha)

        # Transform to body frame
        if i in [0, 3]:  # Front legs
            foot_pos = cs.mtimes(petoi.T01_front, cs.vertcat(x_foot, z_foot, 1))
        else:  # Back legs
            foot_pos = cs.mtimes(petoi.T01_back, cs.vertcat(x_foot, z_foot, 1))

        foot_positions.append(foot_pos[:2])  # Use x,z coordinates

    # Single Rigid Body Dynamics (SRBD)
    m = 1.0  # Robot mass [kg]
    g = 9.81  # Gravity [m/s²]

    # Sum of ground reaction forces (spring-damper approximation)
    sum_f = cs.MX.zeros(2)
    for fp in foot_positions:
        # Virtual spring-damper model
        f_x = -10.0 * (fp[^6_0] - x[^6_0]) - 1.0 * x[^6_3]
        f_z = -10.0 * (fp[^6_1] - x[^6_1]) - 1.0 * x[^6_4] + m*g
        sum_f += cs.vertcat(f_x, f_z)

    # Dynamics equations
    dxdt = cs.vertcat(
        x[^6_3],  # ẋ
        x[^6_4],  # ż
        x[^6_5],  # ω
        sum_f[^6_0]/m,  # ẍ
        sum_f[^6_1]/m,  # z̈
        0.0  # Simplified angular dynamics
    )

    return cs.Function('f_continuous', [x, u], [dxdt])
```

Implementation Notes

1. **State Vector**:

   ```
   x = [c_x, c_z, θ, ẋ, ż, ω]
   - c_x: CoM horizontal position
   - c_z: CoM vertical position
   - θ: Body pitch angle
   - ẋ, ż: CoM velocities
   - ω: Angular velocity
   ```

2. **Key Kinematic Relationships**:
   - Uses transformation matrices `T01_front/T01_back` from provided code
   - Implements inverse of `leg_ik` for forward kinematics
   - Includes body orientation in transformation calculations

3. **Dynamic Assumptions**:
    - Simple spring-damper model for ground reaction forces
    - Single rigid body approximation for dynamics
    - Planar motion (sagittal plane) simplification

Usage Example with MPC

```
# Initialize kinematics model
petoi = PetoiKinematics(render_mode=None)
f_continuous = create_dynamics_function(petoi)

# CasADi-based MPC setup
opti = cs.Opti()
x = opti.variable(6, N+1)  # State trajectory
u = opti.variable(8, N)    # Control trajectory

for k in range(N):
    # Euler discretization
    opti.subject_to(x[:,k+1] == x[:,k] + dt*f_continuous(x[:,k], u[:,k]))
```

This formulation connects the provided kinematic code with dynamic constraints needed for MPC, enabling whole-body control using joint angles as inputs while considering body orientation and ground contact forces.

⁂

import mujoco

import mujoco.viewer
import numpy as np
import time
import matplotlib.pyplot as plt
import matplotlib
matplotlib.use("TkAgg") # or "Qt5Agg", depending on your setup

import os

from skills import balance, wkf, bk
from PID_Controller import PID_Controller
from petoi_kinematics import PetoiKinematics
import casadi as cs

def create_dynamics_function(petoi: PetoiKinematics):
# State: [c_x, c_z, θ, ẋ, ż, ω] (CoM position, pitch angle, velocities)
x = cs.MX.sym('x', 6)

```
# Input: Joint angles [α1, β1, ..., α4, β4]
u = cs.MX.sym('u', 8)

# Extract body orientation from state
theta = x[2]
cg, sg = cs.cos(theta), cs.sin(theta)

# Forward kinematics for all feet
foot_positions = []
for i in range(4):
    alpha = u[i*2]
    beta = u[i*2+1]

    # From leg_ik inverse kinematics, reversed
    L = petoi.leg_length + petoi.foot_length * cs.sin(beta)
    x_foot = L * cs.sin(alpha)
    z_foot = -L * cs.cos(alpha)

    # Transform to body frame
    if i in [0, 3]:  # Front legs
        foot_pos = cs.mtimes(petoi.T01_front, cs.vertcat(x_foot, z_foot, 1))
    else:  # Back legs
        foot_pos = cs.mtimes(petoi.T01_back, cs.vertcat(x_foot, z_foot, 1))

    foot_positions.append(foot_pos[:2])  # Use x,z coordinates

# Single Rigid Body Dynamics (SRBD)
m = 1.0  # Robot mass [kg]
g = 9.81  # Gravity [m/s²]

# Sum of ground reaction forces (spring-damper approximation)
sum_f = cs.MX.zeros(2)
for fp in foot_positions:
    # Virtual spring-damper model
    f_x = -10.0 * (fp[0] - x[0]) - 1.0 * x[3]
    f_z = -10.0 * (fp[1] - x[1]) - 1.0 * x[4] + m*g
    sum_f += cs.vertcat(f_x, f_z)

# Dynamics equations
dxdt = cs.vertcat(
    x[3],  # ẋ
    x[4],  # ż
    x[5],  # ω
    sum_f[0]/m,  # ẍ
    sum_f[1]/m,  # z̈
    0.0  # Simplified angular dynamics
)

return cs.Function('f_continuous', [x, u], [dxdt])
```

pid_controller = PID_Controller("urdf/bittle.xml")
f_continuous = create_dynamics_function(petoi)

Compute Jacobians

A_c = cs.jacobian(f_continuous, x)
B_c = cs.jacobian(f_continuous, u)

Discretize (Euler method)

dt = 0.1 # Timestep
A = cs.MX.eye(12) + A_c * dt
B = B_c * dt

Define state and input variables

x = cs.MX.sym('x', 12) # 12D state
u = cs.MX.sym('u', 12) # 12D joint angles

kp = 6e4
ki = 5e2
kd = 5e1
dt = 1e-3
num_timesteps = int(10e4)
with mujoco.viewer.launch_passive(pid_controller.model, pid_controller.data) as viewer:
if os.name == 'nt':
import ctypes
hwnd = ctypes.windll.user32.GetForegroundWindow()
ctypes.windll.user32.ShowWindow(hwnd, 3) # SW_MAXIMIZE = 3
#print("Stand")
#execute(balance,num_timesteps,dt, kp,ki,kd, plotty=True)
# print("Walk Forward")
# pid_controller.execute(wkf,num_timesteps,dt,kp,ki,kd, viewer=viewer, plotty = True)
print("Walk Backward")
pid_controller.execute(bk,num_timesteps,dt,kp,ki,kd, viewer=viewer, plotty=True)

fix this

Here's the corrected code with explanations of the fixes:

```
import mujoco
import mujoco.viewer
import numpy as np
import time
import matplotlib.pyplot as plt
import matplotlib
matplotlib.use("TkAgg")  # or "Qt5Agg", depending on your setup
import os
import casadi as cs  # Requires installation: pip install casadi

# Fix 1: Add proper imports and initialize PetoiKinematics correctly
from petoi_kinematics import PetoiKinematics
```

```
# Fix 2: Initialize PetoiKinematics before use
petoi = PetoiKinematics(render_mode=None)

# Fix 3: Correct state dimensions (6D instead of 12D)
def create_dynamics_function(petoi: PetoiKinematics):
    # State: [c_x, c_z, θ, ẋ, ż, ω] (6 dimensions)
    x = cs.MX.sym('x', 6)

    # Input: Joint angles [α1, β1, ..., α4, β4] (8 dimensions)
    u = cs.MX.sym('u', 8)

    # Extract body orientation from state
    theta = x[^7_2]
    cg, sg = cs.cos(theta), cs.sin(theta)

    # Forward kinematics for all feet
    foot_positions = []
    for i in range(4):
        alpha = u[i*2]
        beta = u[i*2+1]

        L = petoi.leg_length + petoi.foot_length * cs.sin(beta)
        x_foot = L * cs.sin(alpha)
        z_foot = -L * cs.cos(alpha)

        if i in [0, 3]:  # Front legs
            foot_pos = cs.mtimes(petoi.T01_front, cs.vertcat(x_foot, z_foot, 1))
        else:  # Back legs
            foot_pos = cs.mtimes(petoi.T01_back, cs.vertcat(x_foot, z_foot, 1))

        foot_positions.append(foot_pos[:2])

    # Single Rigid Body Dynamics
    m = 1.0  # Robot mass [kg]
    g = 9.81  # Gravity [m/s²]

    sum_f = cs.MX.zeros(2)
    for fp in foot_positions:
        f_x = -10.0 * (fp[^7_0] - x[^7_0]) - 1.0 * x[^7_3]
        f_z = -10.0 * (fp[^7_1] - x[^7_1]) - 1.0 * x[^7_4] + m*g
        sum_f += cs.vertcat(f_x, f_z)

    dxdt = cs.vertcat(
        x[^7_3],  # ẋ
        x[^7_4],  # ż
        x[^7_5],  # ω
        sum_f[^7_0]/m,  # ẍ
        sum_f[^7_1]/m,  # z̈
        0.0  # Simplified angular dynamics
    )

    return cs.Function('f_continuous', [x, u], [dxdt])

# Fix 4: Use correct dimensions for Jacobian calculation
f_continuous = create_dynamics_function(petoi)
```

```
x = cs.MX.sym('x', 6)  # Match dynamics function state dimension
u = cs.MX.sym('u', 8)  # Match dynamics function input dimension

A_c = cs.jacobian(f_continuous(x, u), x)
B_c = cs.jacobian(f_continuous(x, u), u)

# Fix 5: Proper discretization with correct matrix dimensions
dt = 0.1  # Timestep
A = cs.MX.eye(6) + A_c * dt  # 6x6 identity matrix
B = B_c * dt

# Fix 6: Initialize PID controller with proper path
# Ensure "urdf/bittle.xml" exists in your project directory
pid_controller = PID_Controller("urdf/bittle.xml")

# Fix 7: Use correct controller parameters
kp = 6e4
ki = 5e2
kd = 5e1
dt = 1e-3
num_timesteps = int(10e4)

# Fix 8: Add proper exception handling for Mujoco
try:
    with mujoco.viewer.launch_passive(pid_controller.model, pid_controller.data) as viewe
        if os.name == 'nt':
            import ctypes
            hwnd = ctypes.windll.user32.GetForegroundWindow()
            ctypes.windll.user32.ShowWindow(hwnd, 3)

        print("Walk Backward")
        pid_controller.execute(
            bk, num_timesteps, dt, kp, ki, kd,
            viewer=viewer, plotty=True
        )
except Exception as e:
    print(f"Error in main execution: {str(e)}")
```

Key fixes made:

1. **Dimension Corrections**: Matched state (6D) and input (8D) dimensions throughout
2. **Matrix Sizes**: Corrected identity matrix size for discretization (6×6 instead of 12×12)
3. **Initialization Order**: Properly initialized `PetoiKinematics` before creating dynamics function
4. **Path Handling**: Added notes about URDF file location requirements
5. **Error Handling**: Added try/except blocks for robust execution
6. **Package Installation**: Added comment about required CasADi installation

To run this successfully:

1. Install required packages:

```
pip install casadi mujoco mujoco-viewer
```

2. Ensure the URDF file exists at `urdf/bittle.xml`
3. Verify Mujoco license setup if using proprietary version

⁂

implement MPC using the "bk" skill as a reference trajectory

To implement Model Predictive Control (MPC) using the "bk" walking backward skill as a reference trajectory, here's a structured approach leveraging your existing codebase:

1. MPC Setup with CasADi

```
# MPC Parameters
N = 10  # Prediction horizon
Q = cs.diag([1e3, 1e3, 1e2, 1e2, 1e2, 1e2])  # State tracking weights
R = cs.diag([1e-3]*8)  # Control effort weights

# Create MPC optimizer
opti = cs.Opti()
X = opti.variable(6, N+1)  # State trajectory
U = opti.variable(8, N)    # Control trajectory
```

2. Reference Trajectory Handling

```
# Load "bk" skill and convert to radians
bk_ref = np.deg2rad(np.array(bk))
ref_traj = bk_ref[:, [3,7,0,4,2,6,1,5]]  # Reorder to match actuator_nums
```

3. MPC Constraints Formulation

```
# Initial state constraint
opti.subject_to(X[:,0] == current_state)

# Dynamics constraints
for k in range(N):
    opti.subject_to(X[:,k+1] == X[:,k] + f_continuous(X[:,k], U[:,k])*dt)

# Joint angle limits (converted from "bk" skill)
opti.subject_to(opti.bounded(-np.pi/2, U, np.pi/2))
```

4. Cost Function Design

```
total_cost = 0
for k in range(N):
    # State tracking cost (CoM + orientation)
    state_error = X[:,k] - ref_traj[k%len(ref_traj), :6]
    total_cost += state_error.T @ Q @ state_error

    # Control smoothing cost
    if k &gt; 0:
        control_diff = U[:,k] - U[:,k-1]
        total_cost += control_diff.T @ R @ control_diff

opti.minimize(total_cost)
```

5. Real-Time MPC Loop

```
# Create solver instance
opts = {'print_time': 0, 'error_on_fail': False}
opti.solver('qpoases', opts)

# Main control loop
for i in range(num_timesteps):
    # Get current state from sensors
    orientation, gyro = pid_controller.get_imu_readings()
    current_state = np.concatenate([pid_controller.data.qpos[:2],
                                   [orientation[^8_2]],  # θ from quaternion
                                   pid_controller.data.qvel[:2],
                                   [gyro[^8_2]]])       # ω_z

    # Solve MPC problem
    sol = opti.solve()

    # Apply first control input
    optimal_angles = sol.value(U[:,0])
    for j, num in enumerate(actuator_nums):
        pid_controller.data.ctrl[actuator_to_ctrl[actuator_map[num]]] = optimal_angles[j]

    # Shift trajectory
    opti.set_initial(X[:,:-1], sol.value(X[:,1:]))
    opti.set_initial(X[:,-1], sol.value(X[:,-1]))
    opti.set_initial(U[:,:-1], sol.value(U[:,1:]))
```

6. Key Integration Points

1. **State Estimation**:
    - Use IMU data for CoM orientation (θ) and angular velocity (ω)
    - Compute CoM position via forward kinematics from joint sensors
2. **Actuator Mapping**: