2D Linear Inverted Pendulum-Based Three Steps Path Planning

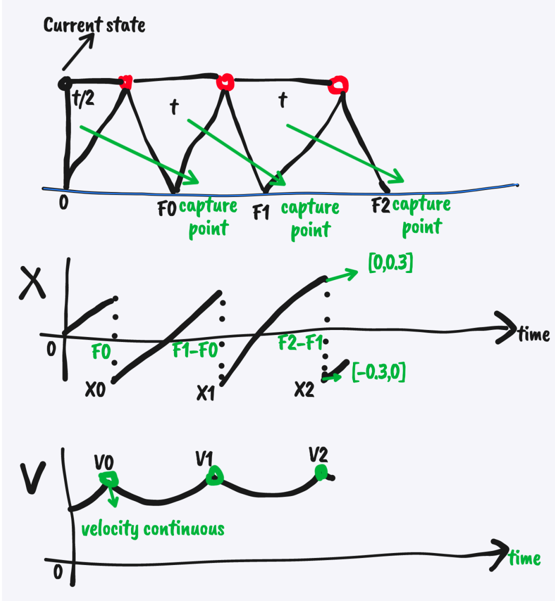
February 9, 2022

Contents

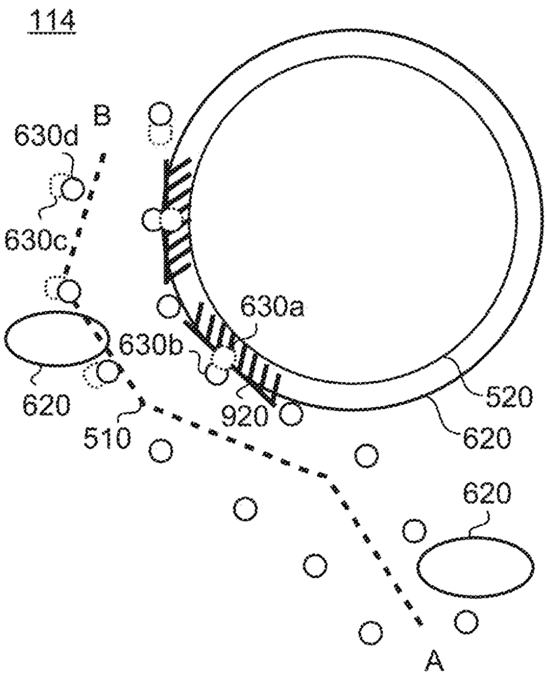
1 Variables

1. CoM Position $\boldsymbol{X}[\mathbf{3}] = \{X0, X1, X2\}$
2. CoM Velocity $\boldsymbol{V}[\mathbf{3}] = \{V0, V1, V2\}$
3. Foot Placement $\boldsymbol{F}[\mathbf{3}] = \{F0, F1, F2\}$

```
# ADD Variables
X = prog.NewContinuousVariables(3, "X")
V = prog.NewContinuousVariables(3, "V")
F = prog.NewContinuousVariables(3, "F")
```



tldraw



2 Constraints $g_i(x) \leq 0$

2.1 Dynamics

$\boldsymbol{x}(t) = \boldsymbol{x}(0)\cosh(t/T_c) + T_c\dot{\boldsymbol{x}}(0)\sinh(t/T_c)$

$\dot{\boldsymbol{x}}(t) = \boldsymbol{x}(0)/T_c\sinh(t/T_c) + \dot{\boldsymbol{x}}(0)\cosh(t/T_c)$

$T_c \equiv \sqrt{z/g}$

Note: t is a constant number.

```python
# Dynamics constraint
def constraint_Dynamics(z):  # z: X0 X1 X2 V0(3) V1 V2 F0(6) F1 F2
    [x_0f, v_0f] = predict(z[0], z[3], full_stance_time)
    [x_1f, v_1f] = predict(z[1], z[4], full_stance_time)
    return np.array(
        [
            (z[6] - 0) - (x_init - z[0]),
            z[3] - v_init,
            (z[7] - z[6]) - (x_0f - z[1]),
            z[4] - v_0f,
            (z[8] - z[7]) - (x_1f - z[2]),
            z[5] - v_1f,
        ]
    )


prog.AddConstraint(
    constraint_Dynamics,
    lb=np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0]),
    ub=np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0]),
    vars=[X[0], X[1], X[2], V[0], V[1], V[2], F[0], F[1], F[2]],
)
```

2.2 CoP Offsets (Kinematics)

$X0 \in [-0.3, 0]$

$X0_{final} \in [0, 0.3]$

$X1 \in [-0.3, 0]$

$X1_{final} \in [0, 0.3]$

$X2 \in [-0.3, 0]$

```python
# Kinematics constraint
def constraint_CoP(z):  # z: X0 X1 X2 F0 F1 F2
    return np.array([z[0], z[1], z[2], z[4] - z[3] + z[1], z[5] - z[4]


prog.AddConstraint(
    constraint_CoP,
    lb=np.array([-0.3, -0.3, -0.3, 0, 0]),
    ub=np.array([0.0, 0.0, 0.0, 0.3, 0.3]),
    vars=[X[0], X[1], X[2], F[0], F[1], F[2]],
)
```

2.3 Feet Keep-Out Areas (Convex Decomposition)

$F1 \in [0.4, 0.45]$

```
prog.AddConstraint(lambda z: np.array([z[0]]), lb=[0.40], ub=[0.45], va
```

3 Costs $\min_x f(x)$

3.1 Reference

$cost_{reference} = \sum_i (F_i - F_{reference})^2 + \sum_i (V_i - V_{reference})^2$

```
prog.AddQuadraticErrorCost(
    Q=np.identity(6),
    x_desired=np.array(
        [
            reference_step_length,
            reference_step_length * 2,
            reference_step_length * 3,
            reference_com_velocity,
            reference_com_velocity,
            reference_com_velocity,
        ]
    ),
    vars=[F[0], F[1], F[2], V[0], V[1], V[2]],
)
```

3.2 Keep-Out Areas Margin

$cost_{margin} = \sum_i (F_1 - MiddleSafetyPoint)^2$

```
weight_margin = 5
prog.AddCost(weight_margin * (0.425 - F[1]) ** 2)
```

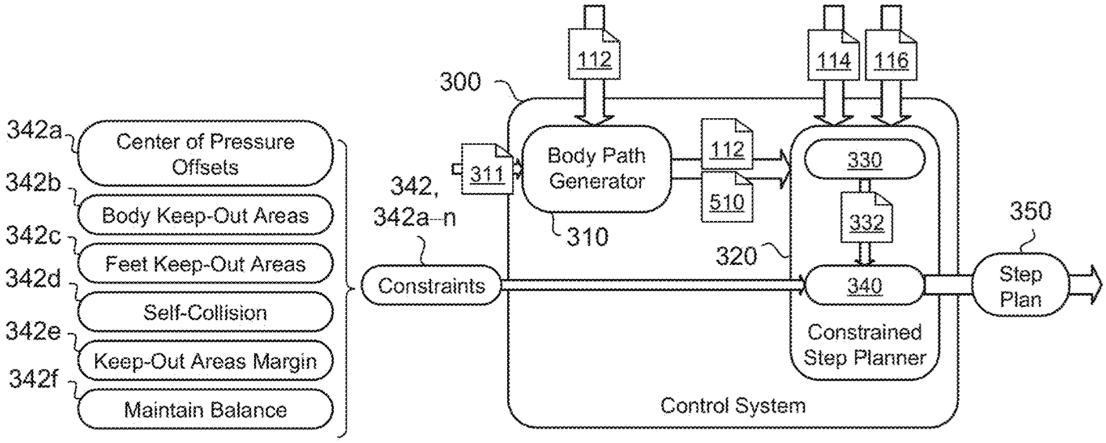
3.3 Maintain Balance

$cost_{balance} = \sum_i (F_i - CapturePoint_i)^2$

```
def cost_Capture(z):  # X0 X1 X2 V0(3) V1 V2 F0(6) F1 F2
    f0_error = (z[6] - 0) - (x0 + v0 * Tc)
    f1_error = (z[7] - z[6]) - (z[0] + z[3] * Tc)
    f2_error = (z[8] - z[7]) - (z[1] + z[4] * Tc)
    return f0_error ** 2 + f1_error ** 2 + f2_error ** 2


prog.AddCost(cost_Capture, vars=[X[0], X[1], X[2], V[0], V[1], V[2], F[
```

4 Solve




```
reference_step_length:  0.24888888888888885
Success? True
X     =   [−0.12227655 −0.10308764 −0.07743495]
V     =   [0.79075813 0.73742488 0.76698752]
F     =   [0.23719403 0.44008937 0.63306645]
cost =    0.055843215451975264
solver:   SNOPT/fortran
```