SLIRP.py (Serial Link Integrated Robot Physics) Design Document

Index

Overview

The SLIRP class models a serial-link manipulator with an arbitrary number of revolute joints. Its primary capabilities include:

1. Symbolic definition of kinematic variables.
2. Forward kinematics for pose (position/orientation), velocity, and acceleration.
3. Computation of link center of mass (COM) positions.
4. End-effector position and orientation (pose).
5. Substitution between joint angles (θ) and generalized coordinates (q).
6. Computation of the Jacobian matrix.
7. Computation of kinetic and potential energy, and the inclusion of non-conservative forces.
8. (Stretch Goal) Incorporation of constraints using Lagrange multipliers.
9. Ability to output the Equations of Motion (EOM) in either generalized coordinates or workspace coordinates.

By achieving these objectives, SLIRP serves as a foundation for advanced robot analysis, simulation, and control development.

Key Design Goals

1. **Arbitrary Number of Links**: The class must dynamically handle any number of links.
2. **Symbolic Computation**: Use Sage Math for symbolic variables for all parameters to allow for analytical solutions and manipulations with minimal time complexity.
3. **Forward Kinematics**: Provide methods to compute end-effector pose, velocity, and acceleration from given joint coordinates and their derivatives.
4. **Energy and Forces**: Compute kinetic and potential energy
5. **Constraints and Lagrange Multipliers (Stretch)**: Symbolically incorporate constraints into the system's equations, enabling the solution of constrained dynamics.
6. **Multi-Coordinate Output**: Generate EOM in both joint (generalized) coordinates and workspace coordinates.
7. **User-Friendliness and Extensibility**: Offer clear documentation, a user manual, and a structure that enables users to easily extend functionality.

Class Structure and Responsibilities

Class Initialization

__init__(self, num_links)

- Inputs: num_links (integer)
- Actions:
    1. Store num_links.
    2. Define symbolic time variable.
    3. Initialize lists and dictionaries for link lengths, masses, angles, and generalized coordinates.
    4. Call methods to establish variables, compute COMs, define end-effector, set substitution dictionaries, and compute Jacobian.

Key Attributes:

- *self.num_links*: Number of links in the manipulator.
- *self.time*: Symbolic time variable.
- *self.link_lengths, self.link_masses:* Lists of symbolic variables.
- *self.thetas, self.qs, self.qdots*: Joint angles and generalized coordinates.
- self.link_COMs: Positions of link centers of mass.
- self.end_effector: [x, y, θ] or [x, y] depending on chosen representation.
- self.qidot_COMS: Substituted expressions for end-effector velocities/accelerations.
- self.theta_to_q_subs, self.q_to_theta_subs: Dictionaries for substitution.
- self.J: The Jacobian matrix.

Establishing Link Variables

establish_link_variables()

- Defines symbolic variables L_i, theta_i, q_i, q_i_dot for each link.
- Stores these in lists for later use.

Calculating COM Positions

calculate_COM_positions()

- Uses link_lengths and thetas to compute each link's COM position.
- Stores [x_COM_i, y_COM_i] for each link in self.link_COMs.

Defining the End-Effector Position

define_end_effector()

- Computes the end-effector's [x, y, θ] by summing the contributions of all links.

Forward Kinematics for Velocities and Accelerations

New Methods:

- ### compute_velocities():
  Uses q_i and q_i_dot along with self.J to compute end-effector linear and angular velocities.
  Output: Symbolic expressions for end-effector velocity.
- ### compute_accelerations():
  Differentiates velocities again or uses q_i_dot and second derivatives q_i_ddot to obtain end-effector accelerations.
  Output: Symbolic expressions for end-effector acceleration.

These methods rely on the Jacobian and time derivatives of joint variables.

Generalized Coordinates Substitution

compute_generalized_coordinates()

- Applies theta_to_q_subs to rewrite end-effector and COM positions in terms of q instead of theta.
- Prepares the system for downstream computations like deriving EOM.

Computing the Jacobian

compute_jacobian()

- Differentiates end-effector coordinates with respect to generalized coordinates to form self.J.

Energy Computations

compute_kinetic_energy():

- Uses joint velocities, link masses, and possibly moment of inertia (if defined later) to compute the total kinetic energy T.

compute_potential_energy():

- Uses link_COMs, masses, and gravity to compute gravitational potential energy U.

compute_non_conservative_forces():

- Symbolically define non-conservative forces (e.g., friction, external loads).
- These forces will appear in the equations of motion but not derived from a potential.

These computations allow the user to derive the Lagrangian L = T - U and subsequently deriving the EOM.

Constraints and Lagrange Multipliers (Stretch Goal)

Incorporate_constraints(constraints):

- Accepts a set of constraint equations involving q, qdot, and possibly t.
- Introduces Lagrange multipliers symbolically to enforce these constraints.
- Modifies the EOM derivation to include these constraints.

Equations of Motion in Various Coordinates

derive_EOM_in_generalized_coordinates():

- Uses Lagrange's equations: $dtd(\partial \dot{q} i \partial L) - \partial q i \partial L = Qnc$
- Produces equations of motion in terms of: $q, \dot{q}, and\ \ddot{q}$

transform_EOM_to_workspace_coordinates():

- Applies kinematic transformations (via the Jacobian) to convert the EOM from joint space to workspace coordinates.
- Output: EOM expressed in terms of end-effector position/orientation and their derivatives.

Data Structures

- **Sage.Matrix**: For link lengths, masses, angles, coordinates, and COM positions.
- **Dictionaries**: For substitution mappings between theta and q.

- **Symbolic Variables**: For all computations (position, velocity, acceleration, energies, EOM).

Extending the Class

- Inertia and Complex Mass Distributions: Add definitions for moments of inertia and compute more general kinetic energies.
- Additional Non-Conservative Forces: Introduce functions for damping, friction models, or control inputs.
- Redundant/Parallel Mechanisms: Extend methods to handle non-serial kinematic chains or closed-loop constraints.
- Numeric Integration and Simulation: Add methods to evaluate expressions numerically and run simulations over time.
- Visualization of user defined System and Simulations: Add methods to plot and visualize the pose and motion of the system given a desired end effector position.

User Manual

Installation

Requirements: Python environment with Sage Math.

Basic Usage

Initialize the Robot:

$$Import\ SLIRP.py$$

$$robot\ =\ SLIRP(num\_links = 3)$$

Set Link Lengths and Masses:

Assign numeric values after initialization

$$robot.link\_lengths[0] \; = \; 1.0$$

$$robot.link\_lengths[1] \; = \; 0.5$$

$$robot.link\_lengths[2] \; = \; 0.75$$

$$robot.link\_masses[0] \; = \; 2.0$$

$$robot.link\_masses[1] \; = \; 1.5$$

$$robot.link\_masses[2] \; = \; 1.0$$

Compute End-Effector Pose:

$$pose \; = \; robot.end\_effector \;\; \# \, [x\_end, y\_end, theta\_end]$$

Compute Jacobian:

$$J \; = \; robot.J$$

Forward Kinematics for Velocity/Acceleration:

$$v\_ee \; = \; robot.compute\_velocities() \quad \# \, Returns\ symbolic\ velocity\ expressions$$

$$a\_ee \; = \; robot.compute\_accelerations() \;\; \# \, Returns\ symbolic\ acceleration\ expressions$$

Energy Calculations:

$$T = robot.compute\_kinetic\_energy()$$

$$U = robot.compute\_potential\_energy()$$

Equations of Motion:

$$EOM\_q = robot.derive\_EOM\_in\_generalized\_coordinates()$$

$$EOM\_workspace = robot.transform\_EOM\_to\_workspace\_coordinates(EOM\_q)$$

How to Extend the Code

Add Inertial Properties:
Add additional attributes and methods to compute inertia matrices and incorporate them into kinetic energy calculations.

Include Control Inputs:
Add methods to represent joint torques or external forces, and include them as non-conservative forces.

Numerical Evaluation:
Write helper methods to substitute numeric values into symbolic expressions for simulation.