The Journal of Open Source Software

ManipulaPy: A GPU-Accelerated Python Framework for Robotic Manipulation, Perception, and Control

M. I. M. AboElNasr [1]

1 Universität Duisburg-Essen

DOI: 10.xxxxxx/draft

Software

- Review
- Repository
- Archive

Editor: Open Journals

Reviewers:

- @openjournals

Submitted: 01 January 1970
Published: unpublished

Summary

ManipulaPy is an open-source Python toolbox that stitches together the entire manipulation pipeline—from URDF parsing to GPU-accelerated dynamics, vision-based perception, planning and control—within a single API. Built on the Product-of-Exponentials model (Lynch & Park, 2017), PyBullet (Coumans & Bai, 2019), CuPy (Okuta et al., 2017) and custom CUDA kernels (Liang et al., 2018), the library lets researchers move from robot description to real-time control with up to **40** × faster inverse-dynamics on a 6-DOF UR5 than a NumPy baseline.

Statement of Need

Robotics research needs tight couplings of geometry, physics, vision and control. Existing stacks—MoveIt (Chitta et al., 2012), Orocos KDL (Smits, 2009) and the Python Robotics Toolbox (Corke & Haviland, 2021)—cover parts of this but require glue code or lack GPU paths. **ManipulaPy** instead:

- converts a URDF to PoE screws **and** realistic joint limits in one call,
- exposes CUDA kernels for time-scaling and (inverse) dynamics (Liang et al., 2018),
- pipes stereo vision through DBSCAN obstacle clustering into the planner[1],
- wraps PyBullet so cameras, planners and controllers stay synchronised at 1 kHz.

[1] Implementation mirrors the clustering in (Chu et al., 2021).

Library Architecture

- **urdf_processor.py** – URDF → (S_i, M, G_i) & limits → `SerialManipulator`, `ManipulatorDynamics`
- **kinematics.py** – PoE FK/IK + Jacobians
- **dynamics.py** – Mass matrix, Coriolis, gravity (GPU-optional)
- **path_planning.py** – CUDA cubic/quintic & SE(3) trajectories
- **control.py** – PD/PID, computed-torque, robust, adaptive controllers
- **vision.py / perception.py** – Stereo → depth → DBSCAN obstacles

- **singularity.py** – Jacobian condition, workspace Monte-Carlo

- **sim.py** – One-line PyBullet setup & loop

- **cuda_kernels.py** – Trajectory & dynamics kernels tuned for 256-thread blocks

- **utils.py** – Lie-group and SE(3) helpers

Theory Highlights

1. URDF Parsing

- The URDF is parsed using PyBullet's internal loader to access link names, joint types, limits, masses, and inertia tensors.
- The joint hierarchy is extracted as a tree of revolute/prismatic joints with parent-child link relations.

2. Screw Axis Extraction

- Each joint is converted into a **screw axis** $S_i \in \mathbb{R}^6$, using the joint's origin, axis, and type.

- The screw axis encodes both rotation and translation components:

$$S = \begin{bmatrix} \omega \\ v \end{bmatrix}, \quad \text{with } \omega = \text{axis}, \quad v = -\omega \times q$$

  where q is the joint position in the base frame.

3. Home Configuration Matrix M

- The default pose of the end-effector with all joint angles at zero is computed as a homogeneous transformation matrix $M \in SE(3)$.

- This serves as the base pose in PoE kinematics:

$$T(\theta) = e^{S_1\theta_1} \cdots e^{S_n\theta_n} M$$

  while the space Jacobian stacks each transformed screw axis

$$J(\theta) = \left[\mathrm{Ad}_{T_1} S_1, \ldots, S_n \right].$$

4. Inertial Property Extraction

- Each link's mass and spatial inertia tensor are wrapped into a 6×6 **spatial inertia matrix** G_i for use in dynamic calculations.
- These matrices are used to construct the mass matrix $M(\theta)$ and Coriolis/gravity terms.

$$M(\theta) = \sum_{i=1}^{n} \mathrm{Ad}_{T_i}^T \, G_i \, \mathrm{Ad}_{T_i}, \qquad \tau = M\ddot{\theta} + C(\theta, \dot{\theta}) + g(\theta).$$

5. Limit and Metadata Mapping

- PyBullet is queried to extract joint limits, damping/friction, and torque constraints.
- This metadata is stored in `robot_data` and injected into the planner and controller to enforce safety and realism.

6. Model Object Output

- Finally, two primary Python objects are constructed:
  - `SerialManipulator`: A pure kinematic model with screw axes and link transforms.
  - `ManipulatorDynamics`: A dynamic model with mass, inertia, and external force computations.

These are returned via:

```
robot = urdf_proc.serial_manipulator
dynamics = urdf_proc.dynamics
```

This one-call setup bridges URDF semantics with analytical modeling, enabling immediate simulation, control, and planning.

CUDA Acceleration

Custom CUDA kernels optimize critical operations:

- **Trajectory Kernel**: Computes joint paths with cubic/quintic scaling
- **Forward Dynamics Kernel**: Solves equations of motion in parallel
- **Inverse Dynamics Kernel**: Calculates required torques from accelerations
- **Cartesian Trajectory Kernel**: Generates SE(3) trajectories with rotation interpolation

These kernels are optimized for 256-thread blocks, reducing trajectory generation latency.

Minimal Example

```python
from ManipulaPy import urdf_processor, path_planning, control, sim
import numpy as np

# build model & CUDA-ready dynamics
proc   = urdf_processor.URDFToSerialManipulator("xarm.urdf")
robot  = proc.serial_manipulator
dyn    = proc.dynamics
ctrl   = control.ManipulatorController(dyn)

# 45° joint ramp
Tf, N  = 3.0, 300
goal   = np.deg2rad([45]*6)
traj   = path_planning.TrajectoryPlanning(robot, "xarm.urdf",
        dyn, proc.robot_data["joint_limits"]).joint_trajectory(
        np.zeros(6), goal, Tf, N, method=3)

# PyBullet sim with PD control
simu   = sim.Simulation("xarm.urdf", proc.robot_data["joint_limits"])
simu.initialize_robot()
Kp, Kd = np.full(6, 80.0), np.full(6, 8.0)
simu.run_controller(ctrl, traj["positions"], traj["velocities"],
        traj["accelerations"], g=[0,0,-9.81], Ftip=np.zeros(6),
        Kp=Kp, Ki=np.zeros(6), Kd=Kd)
```

Acknowledgements

Work supported by **Universität Duisburg-Essen** and inspired by *Modern Robotics* (Lynch & Park, 2017), PyBullet (Coumans & Bai, 2019), and Ultralytics YOLO (**?**) projects.

References

Chitta, S., Sucan, I. A., & Cousins, S. (2012). MoveIt!: An overview. *IEEE Robotics & Automation Magazine*. https://moveit.ros.org/

Chu, Y., Wang, L., & Zhang, M. (2021). An approach to boundary detection for 3-d point clouds based on DBSCAN clustering. *Pattern Recognition*, *124*, 108431. https://doi.org/10.1016/j.patcog.2021.108431

Corke, P., & Haviland, J. (2021). Robotics, vision and control: Fundamental algorithms in MATLAB, Python and Julia. *Robots, Autonomous Systems*, *1*. https://petercorke.com/books/robotics-vision-control-python-the-practice-of-robotics-vision/

Coumans, E., & Bai, Y. (2019). *PyBullet: Physics simulation for games, robotics, and machine learning*. http://pybullet.org

Liang, H., Du, X., & Xiao, J. (2018). GPU-based high-performance robot dynamics computation. *IEEE Int. Conf. On Robotics and Automation (ICRA)*, 2396–2402.

Lynch, K. M., & Park, F. C. (2017). *Modern robotics: Mechanics, planning, and control*. Cambridge University Press. http://modernrobotics.org

Okuta, R., Unno, Y., Nishino, D., Hido, S., & Loomis, C. (2017). CuPy: A NumPy-compatible library for NVIDIA GPU calculations. *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Thirty-First Annual Conference on Neural Information Processing Systems (NeurIPS)*. https://cupy.dev

Smits, R. (2009). The kinematics and dynamics library (KDL). *OROCOS Project*. https://www.orocos.org/kdl.html