This thesis was submitted to the Institute of Mechanism Theory, Machine Dynamics and Robotics

Development of a Motion Planner based on Nonlinear Optimization for Free-Floating Robots

Master Thesis

by:

Siddhant Vivek Kadwe B.E.

Matriculation number: 415706

supervised by:

Dr.-Ing. Roberto Lampariello
Mr. Carlo Weidemann MSc.

Examiner:

Univ.-Prof. Dr.-Ing. Dr. h. c. Burkhard Corves

Prof. Dr.-Ing. Mathias Hüsing

Aachen, 20 April 2023

Sperrvermerk

Diese Masterarbeit wurde am Institut für Robotik und Mechatronik des Deutschen Zentrums für Luft und Raum fahrt (DLR) angefertigt. Sie enthält daher firmeninterne und vertrauliche Informationen des DLR. Alle Rechte verbleiben beim DLR und die Arbeit bleibt Eigentum des DLR. Um diese Informationen zu schützen, unterliegt diese Arbeit einem unbefristeten Sperrvermerk.

Alle Informationen aus der Arbeit dürfen weder ganz noch teilweise ohne Genehmigung des DLR veröffentlicht oder vervielfältigt werden. Es wird auch darauf hingewiesen, dass keine Informationen an Dritte weitergegeben werden dürfen.

Betreuer

Dr.-Ing. Roberto Lampariello

Institut für Robotik und Mechatronik

Deutsches Zentrum für Luft- und Raumfahrt

Münchener Straße 20

82234 Weßling

roberto.lampariello@dlr.de

20 April 2023

Master Thesis

by Siddhant Vivek Kadwe B.E.
Matriculation number: 415706

Development of a Motion Planner based on Nonlinear Optimization for Free-Floating Robots

---

While the launch of spacecraft and satellites is increasing rapidly, the accumulation of debris in space creates an obstacle for future space missions. Free-floating robots are used to tackle the issue of grasping the tumbling, non-cooperative space debris and stabilizing them. Free-floating robots are manipulator arms attached to the base of an unactuated satellite. The core problem is calculating the robot's trajectory for grasping as the debris has unknown parameters (momentum, pose, et cetera) within a specific time window. The use of an optimization-based motion planner is considered in this research as it provides the criterion for robustness and stability. Trajectory planning is formulated as an Optimal Control Problem with various motion constraints and a cost function that has to be minimized. The drawback of this technique is the time consumption. Parallelization techniques, using multi-core CPU and GPU, will be analyzed to reduce the time. Comparison between different techniques for solution of the equation of motion for the non-holonomic system, optimization methodologies will also be carried out.

The methodology undertaken is as follows:
The Optimal Control problem is analyzed, and parts that could parallelize within the Optimal Control Framework will be determined. OpenMP-based parallel functions for CPU processing and CUDA-based parallel functions for GPU processing are to be developed. The speedup gain, if any, for the developed parallel functions will be examined. Different optimization methodologies will be implemented and compared. The trajectory optimization problem will be drawn, with new task space. The parallel routines developed, which provide a significant improvement over the state-of-the-art techniques, will be implemented for the motion planning problem. The statistical analysis of the results from the motion planner will be carried out.

Supervisor: Dr.-Ing. Roberto Lampariello
Mr. Carlo Weidemann MSc.

Eidesstattliche Versicherung

Siddhant Vivek Kadwe Matrikel-Nummer: 415706

Ich versichere hiermit an Eides Statt, dass ich die vorliegende Master Thesis mit dem Titel

Development of a Motion Planner based on Nonlinear Optimization for Free-Floating Robots

selbstständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Für den Fall, dass die Arbeit zusätzlich auf einem Datenträger eingereicht wird, erkläre ich, dass die schriftliche und die elektronische Form vollständig übereinstimmen. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Aachen, 20 April 2023

Siddhant Vivek Kadwe

Belehrung:

§ 156 StGB: Falsche Versicherung an Eides Statt

Wer vor einer zur Abnahme einer Versicherung an Eides Statt zuständigen Behörde eine solche Versicherung falsch abgibt oder unter Berufung auf eine solche Versicherung falsch aussagt, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.

§ 161 StGB: Fahrlässiger Falscheid; fahrlässige falsche Versicherung an Eides Statt

(1) Wenn eine der in den §§ 154 bis 156 bezeichneten Handlungen aus Fahrlässigkeit begangen worden ist, so tritt Freiheitsstrafe bis zu einem Jahr oder Geldstrafe ein.

(2) Straflosigkeit tritt ein, wenn der Täter die falsche Angabe rechtzeitig berichtigt. Die Vorschriften des § 158 Abs. 2 und 3 gelten entsprechend.

Die vorstehende Belehrung habe ich zur Kenntnis genommen:

Aachen, 20 April 2023

Siddhant Vivek Kadwe

The present translation is for your convenience only.
Only the German version is legally binding.

Statutory Declaration in Lieu of an Oath

Siddhant Vivek Kadwe Matriculation number: 415706

I hereby declare in lieu of an oath that I have completed the present Master Thesis entitled

Development of a Motion Planner based on Nonlinear Optimization for Free-Floating Robots

independently and without illegitimate assistance from third parties. I have used no other than the specified sources and aids. In case that the thesis is additionally submitted in an electronic format, I declare that the written and electronic versions are fully identical. The thesis has not been submitted to any examination body in this, or similar, form.

Aachen, 20 April 2023 Siddhant Vivek Kadwe

Official Notification:

Para. 156 StGB (German Criminal Code): False Statutory Declarations

Whosoever before a public authority competent to administer statutory declarations falsely makes such a declaration or falsely testifies while referring to such a declaration shall be liable to imprisonment not exceeding three years or a fine.

Para. 161 StGB (German Criminal Code): False Statutory Declarations Due to Negligence

(1)If a person commits one of the offences listed in sections 154 to 156 negligently the penalty shall be imprisonment not exceeding one year or a fine.
(2) The offender shall be exempt from liability if he or she corrects their false testimony in time. The provisions of section 158 (2) and (3) shall apply accordingly. I have read and understood the above official notification: :

Aachen, 20 April 2023 Siddhant Vivek Kadwe

Acknowledgments

I am pleased to take this opportunity to express my gratitude to the Institute of Robotics and Mechatronics at the German Aerospace Center (DLR), as well as to my supervisor, Dr.-Ing. Roberto Lampariello, for entrusting me with this opportunity. His invaluable guidance and knowledge have been instrumental in my growth, and I am deeply appreciative of everything he has taught me. Additionally, I wish to extend my sincere thanks to Dr. Prof. Matthias Gerdts for his contributions to my understanding of the project's mathematical aspects.

I am also grateful to Mr. Carlo Weidemann for his support and assistance throughout the completion of my thesis. He provided invaluable help with various tasks, including documentations at the university. Moreover, I would like to express my appreciation to Prof. Dr.-Ing. Dr. h. c. Burkhard Corves and Prof. Dr.-Ing. Mathias Hüsing for their exceptional course in Robotic Systems Engineering and for serving as my thesis examiners.

Lastly, I would like to thank my family members, Vivek Kadwe, Vaishali Kadwe, and Manas Kadwe, as well as my friends for their unwavering support and encouragement.

Siddhant Vivek Kadwe

20 April 2023

Contents

Formula symbols and indices

Optimal Control Symbols

t	s	Time
$\mathbf{x}$	–	General State Vector
$\mathbf{u}$	–	General Control Vector
$\mathbf{f}$	–	General System Dynamics
$\mathbf{h}$	–	Equality Constraints
$\mathbf{g}$	–	Inequality Constraints
$\boldsymbol{\psi}$	–	Boundary Conditions
$\mathbf{c}$	–	de Boor points
$\mathbf{B}$	–	Basis Function
$\mathbf{J}$	–	Cost Function Jacobian
$\mathbf{G}$	–	Inequality Constraint Jacobian
$\mathbf{H}$	–	Equality Constraint Jacobian
$\mathbf{S}$	–	Sensitivity Matrix

Robot Symbols

$\mathbf{r}$	m	Position Vector
$\mathbf{A}$	–	Orientation Matrix
$\mathbf{I}$	m	Position Vector between links
n	–	Number of links
$\mathbf{v}$	$\frac{\mathrm{m}}{\mathrm{s}}$	Linear Velocity
$\boldsymbol{\omega}$	rad/s	Angular Velocity

$\mathbf{q}$	rad	Joint Angles
$\boldsymbol{\tau}$	Nm	Joint Torques
$\mathbf{x}^e$	m	End Effector Pose
$\mathbf{x}^e_{des}$	m	Desired End Effector Pose
$\mathbf{x}^0$	m	Base Body Pose
$\mathbf{M}$	kgm^2	Inertia Matrix
$\mathbf{M}_b$	kgm^2	Base Body Inertia Matrix
$\mathbf{M}_{bm}$	kgm^2	Coupling Inertia Matrix between base and arm
$\mathbf{C}$	Nm/s	Coriolis and Centrifugal Matrix

List of abbreviations

General abbreviations

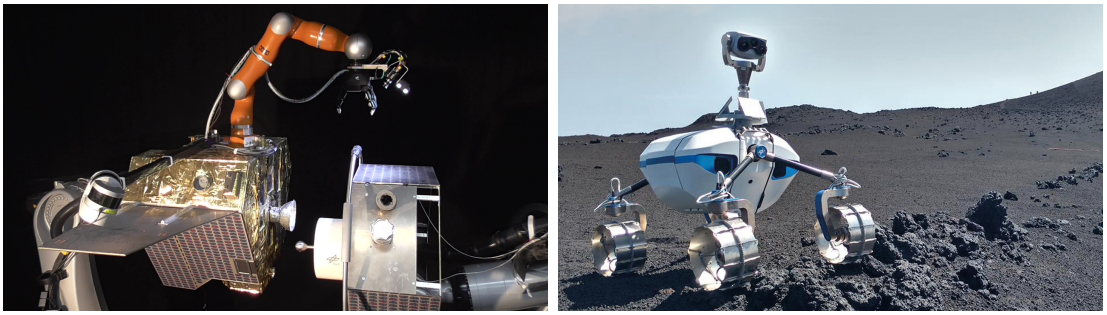
DLR	Deutsches Zentrum für Luft- und Raumfahrt
RM	Institute of Robotics and Mechatronics
OCP	Optimal Control Problem
NLP	Non-Linear Programming
CPU	Central Processing Unit
GPU	Graphical Processing Unit
API	Application Programming Interface
DAE	Differential and Algebraic Equation
ODE	Ordinary Differential Equation
DOCP	Discretized Optimal Control Problem
SQP	Sequential Quadratic Programming
GPGPU	General-Purpose computing on Graphics Processing Units
QP	Quadratic Programming
CUDA	Compute Unified Device Architecture
DH	Denavit-Hartenberg
CM	Center of Mass
DoF	Degrees of Freedom
DOPRI	Dormand-Prince Runge-Kutta
RHS	Right Hand Side
FD	Finite Difference
AD	Automatic Differentiation

1 Introduction

The German Aerospace Center, also known as Deutsches Zentrum für Luft- und Raumfahrt (DLR), is the national center for aerospace, energy, and transportation research in Germany. Founded in 1969, the DLR is responsible for designing, developing and testing new technologies and concepts for space exploration, aviation and transportation systems. With over 10,000 employees, the DLR is one of the largest and most advanced research institutions in the world, providing valuable insights and contributions to various national and international space projects.

The Institute of Robotics and Mechatronics (RM) is a leading research institution within the DLR. The RM focuses on developing advanced technologies for use in space exploration, aviation, and terrestrial applications. With a team of over 150 skilled researchers and scientists, the RM is one of the largest and most advanced institutions of its kind globally. The research activities of the RM span various topics, including intelligent mechatronics, autonomous systems, robotics for exploration, human-robot interaction, and virtual and augmented reality.

Space robotics is a growing field that aims to create intelligent and autonomous robots to operate in harsh environments in space. It can enhance human space missions, enable scientific investigations and exploration of remote locations.



(a) On-Orbit Servicing Simulator (OOS-Sim)

(b) Lightweight Rover Unit (LRU)

Figure 1.1: Space Robots at Institute of Robotics and Mechatronics (RM), Source: [RM website]

The Department of *Autonomie und Fernprogrammierung* at RM is at the forefront of research and development of robotic technologies for space exploration. The group has a particular focus on on-orbit servicing, which involves the maintenance, refuelling, and repair of satellites and other space infrastructure. The group has developed a state-of-the-art On-Orbit Servicing Simulator Figure 1.1a, which provides a platform for the testing

and validation of new robotic servicing technologies. The *Mechatronische Systeme* group has developed the Lightweight Rover Unit (LRU) Figure 1.1b, which is a highly versatile and mobile robotic platform for lunar and planetary exploration. The LRU is capable of performing a wide range of tasks, including drilling, sampling, and reconnaissance, making it a valuable asset for future space missions.

Robotic manipulator arms mounted on a satellite are utilized for capturing a tumbling target in space and involve two phases: free-flying and free-floating [DP93]. The primary distinction between these phases is whether the satellite is actuated or not [PAM21]. The free-floating phase is activated to minimize the use of propellant by turning off the base satellite actuators.

Capturing targets with a free-floating robot requires planning its motion [PAM21]. This is generally formulated as an optimal control problem (OCP), also known as trajectory optimization [Bet98]. Such procedure provides high-quality solutions that can satisfy multiple performance criteria and nonlinear constraints, also generates smooth trajectories that are both continuous and differentiable. One approach to solving the OCP is to transcribe the problem to a Nonlinear Programming (NLP) problem, which can then be solved by finding the optimality conditions [Bet98]. The biggest drawback of this methodology is the time it takes to create a feasible solution, as multiple iterations are required to solve the NLP problem using the optimal control framework. It is crucial that the motion planning computation is completed within a fraction of the total time required for motion execution [Lam10], [LMO18].

Motivated by the time complexity challenge posed by the aforementioned OCP, the thesis conducts an analysis to evaluate the potential benefits of two novel computation methods. Through this analysis, the bottlenecks and segments within the OCP that could be parallelized are identified, ultimately pinpointing the sections that displayed particularly high time consumption:

- Calculating the Jacobians of cost function and constraint: 10 to 15 percent of the total time.
- Integrating the Robot Dynamics: 40 to 50 percent of the total time.

The scope of this thesis is to investigate various techniques for enhancing the time complexity and execution speed of motion planning using the recent advancements in CPU and GPU architectures to allow the parallelization of functions and routines. The efficacy of these techniques will be assessed and compared to determine the most appropriate solution. The developed methods will be applied to address offline and online motion planning

challenges for free-floating space robots, which are utilized for different purposes, including debris capturing in space.

The structure of this thesis is outlined as follows:

- Chapter 2 provides an overview of the current state-of-the-art motion planning techniques, as well as the laws, Application Programming Interfaces (APIs), and software used.
- In Chapter 3, the problem statement of the thesis is gradually developed, and two workloads that the thesis will address are formulated.
- Chapters 4 and 5 present the proposed solutions for the two workloads, and their experimental implementations and results are discussed in detail.
- Finally, Chapter 6 concludes the thesis by summarizing the findings and presenting potential directions for future research.

2 State of the Art

This chapter presents an overview about the state of the art methodologies and software in motion planning, optimal control, and parallel computation.

2.1 Motion Planning

A fundamental need in robotics is to have algorithms that convert high-level specifications of tasks from humans into low-level descriptions of how to move [LaV06]. Motion planning of robots is an essential research area in robotics that deals with developing algorithms for designing feasible and optimal trajectories for robots. Several approaches have been proposed to address this problem, including graph-based, sampling-based, and optimization-based methods. Graph-based methods represent the environment as a graph, where the nodes correspond to the robot's configuration and the edges represent the transitions between the configurations. The goal is to find a path through the graph that satisfies constraints and optimizes performance criteria. Sampling-based methods, on the other hand, generate a set of random configurations, which are tested for feasibility and optimality. These methods do not rely on the construction of an explicit graph and can handle high-dimensional configuration spaces. Optimization-based methods formulate the motion planning problem as an optimization problem, where the objective function is optimized subject to constraints. These methods can handle complex constraints and performance criteria, but the optimization problem's complexity can be a challenge.

2.1.1 Graph-based motion planning

Graph-based motion planning is a class of motion planning methods that rely on constructing an explicit graph of the configuration space, where each node represents a valid configuration and each edge represents a valid motion between two configurations. The most commonly used graph-based methods are Dijkstra's algorithm, A* algorithm, and their variations.

Dijkstra's algorithm [Dij59], also known as the shortest path algorithm, is a popular graph-based motion planning algorithm that finds the shortest path between two configurations. The algorithm iteratively explores the graph by selecting the node with the shortest distance from the start node and adding its neighbors to the search space. This process

continues until the goal node is reached, and the optimal path from the start to the goal configuration is obtained.

The A* algorithm [HNR68] is an extension of Dijkstra's algorithm that includes a heuristic function to guide the search towards the goal configuration. The heuristic function estimates the distance from a node to the goal configuration and helps to focus the search on the most promising nodes. A* algorithm is particularly effective in high-dimensional configuration spaces where the search space is large and the path cost can be reduced by considering the heuristic information.

Graph-based methods have several advantages, including optimality and completeness, meaning that they guarantee finding the optimal solution if one exists. However, they can be computationally expensive, particularly in high-dimensional configuration spaces, and they may not be suitable for systems with complicated dynamics and constraints.

2.1.2 Sampling-based motion planning

Sampling-based motion planning sample random robot configuration space to find a collision-free path between the start and goal configurations. The most commonly used sampling-based methods include probabilistic roadmaps (PRM) [KSL96], rapidly exploring random trees (RRT) [LaV98], and their variations [KWP11].

Probabilistic roadmaps (PRM) algorithm is a popular sampling-based motion planning algorithm that constructs a roadmap of the configuration space by randomly sampling configurations and connecting them to form a graph. The algorithm then searches for a path between the start and goal configurations by exploring the graph. PRM can be enhanced by using various sampling strategies, such as biased sampling towards narrow passages and high-cost regions, to improve the efficiency of the algorithm.

RRT algorithm and the variants are another popular sampling-based motion planning algorithm that constructs a tree of configurations by iteratively sampling a random configuration and adding it to the tree. The new configuration is connected to the nearest existing configuration in the tree, and this process continues until the goal configuration is reached. The RRT algorithm can be enhanced by using different strategies, such as goal biasing and intelligent sampling, to improve its efficiency and optimality.

The advantages of sampling-based methods include scalability, applicability to systems with complicated dynamics and constraints, and computational efficiency in high dimensional configuration spaces. However, they may not always guarantee finding an optimal solution and may require significant tuning of parameters to achieve satisfactory results.

2.1.3 Optimization-based motion planning

Optimization-based motion planning focuses on finding a solution that minimizes a cost function, which may represent the distance, time, energy, or any other performance criterion. The optimization-based motion planning methods can be formulated as nonlinear programming problems or linear programming problems, and the solution can be obtained by using gradient-based or gradient-free optimization techniques.

One of the most popular optimization-based motion planning methods is the trajectory optimization algorithm, which formulates the motion planning problem as a nonlinear programming problem and optimizes the trajectory subject to constraints such as dynamic constraints, collision avoidance, and boundary conditions. The solution to this problem can be obtained using numerical optimization techniques [Bet98], such as gradient-based optimization, and can produce high-quality trajectories that satisfy the motion constraints.

Other optimization-based motion planning methods include convex optimization-based methods, such as linear and quadratic programming, and non-convex optimization-based methods, such as global optimization and stochastic optimization. These methods can be used to find optimal solutions to motion planning problems that involve complex constraints and objectives, such as optimal path planning and motion planning under uncertainty.

Optimization-based motion planning find high-quality solutions that satisfy multiple performance criteria and constraints, as well as the ability to generate smooth and natural-looking trajectories. However, these methods can be computationally expensive and may require significant tuning of parameters to achieve satisfactory results [LMO18].

With the advantages of trajectory optimization, it is the most suitable motion planning method for free-floating robots. It can manage the complex dynamics and environments to produce optimal solutions. The next sections dives into the details of trajectory optimization and its solution methods.

2.2 Trajectory Optimization

The term trajectory optimization refers to a set of methods that are used to find the best choice of trajectory, typically by selecting the inputs to the system, known as controls, as functions of time [Kel17]. The problem of motion planning is framed as a nonlinear programming problem and the trajectory is optimized while taking into account certain constraints, such as dynamic constraints, collision avoidance, and boundary conditions.

Trajectory optimization can be used to plan the motion of a robotic system through complex environments while optimizing a performance metric such as energy consumption or execution time. To apply trajectory optimization, the motion planning problem is first formulated as an optimization problem with the objective function and constraints based on the specific requirements of the problem. The optimization problem can then be solved using numerical optimization techniques such as gradient-based optimization, which iteratively improves the solution by calculating the gradient of the objective function and adjusting the trajectory accordingly [Bet98].

Consider an objective function to be minimized with respect to (w.r.t) $\mathbf{x}$ and $\mathbf{u}$, the state and control trajectories respectively, whereas, t_0 and t_f are the initial and final time:

$$\min_{t_0, t_f, x(t), u(t)} J(t_0, t_f, \mathbf{x}(t_0), \mathbf{x}(t_f)) + \int_{\mathbf{t_0}}^{\mathbf{t_f}} (\omega(\tau, \mathbf{x}(\tau), \mathbf{u}(\tau)))d\tau \tag{2.1}$$

The objective function has 2 terms, $J(\cdot)$ is the Mayer term and the integral is known as the Langrange term. If both of these exists in the objective, then the complete form is known as Bolza form.

The objective function is minimized such that the constraints are satisfied such as the system dynamics ($\mathbf{f}$):

$$\dot{\mathbf{x}}(t) = \mathbf{f}(t, \mathbf{x}(t), \mathbf{u}(t)) \tag{2.2}$$

Path constraints ($\mathbf{h}$):

$$\mathbf{h}(t, \mathbf{x}(t), \mathbf{u}(t)) \leq 0 \tag{2.3}$$

Boundary constraints ($\mathbf{g}$):

$$\mathbf{g}(t_0, t_f, \mathbf{x}(t_0), \mathbf{x}(t_f)) \leq 0 \tag{2.4}$$

Path bounds on states where $\mathbf{x_{low}}$ and $\mathbf{x_{upp}}$ are the lower and upper bounds respectively:

$$\mathbf{x_{low}} \leq \mathbf{x}(t) \leq \mathbf{x_{upp}} \tag{2.5}$$

Path bounds on controls where $\mathbf{u_{low}}$ and $\mathbf{u_{upp}}$ are the lower and upper bounds respectively:

$$\mathbf{u_{low}} \leq \mathbf{u}(t) \leq \mathbf{u_{upp}} \tag{2.6}$$

Trajectory optimization has several advantages, including the ability to handle complex motion constraints and generate smooth and natural-looking trajectories. However, trajectory optimization can be computationally expensive and may require significant parameter tuning to achieve satisfactory results.

In recent years, researchers have proposed various extensions to the trajectory optimization framework to address specific challenges, such as incorporating uncertainty or multi-objective optimization. Examples of such extensions include chance-constrained trajectory optimization, multi-objective trajectory optimization, and learning-based trajectory optimization.

2.2.1 Optimal Control

Trajectory optimization can be solved using optimal control techniques by formulating the motion planning problem as an optimal control problem [Kel17]. The optimal control problem seeks to find a control policy that minimizes a cost function while satisfying the dynamics and constraint equations of the system.

The cost function can be defined based on various criteria such as energy consumption, time to complete the task, or trajectory smoothness. The dynamic and constraint equations describe the system's evolution over time and can be derived from the robot's kinematics and dynamics equations. One approach to solving the optimal control problem is through indirect methods, where the problem is first transformed into a boundary value problem, and then solved using numerical optimization techniques such as shooting methods or collocation methods.

2.2.2 OCPID-DAE1

Developed by Univ.-Prof. Dr. rer. nat. Matthias Gerdts of the *Universität der Bundeswehr, München*, the Optimal Control and Parameter Identification with Differential-Algebraic Equations of Index 1 (OCPID-DAE1) [Ger18] is a package developed in Fortran 90 to find numerical solution of the Optimal Control Problems (OCP).

The goal is to minimize the Objective Function subject to the Implicit Differential Equation (DAE), Boundary Conditions, State Constraints, and the Box Constraints. $\zeta_i \in [t_0, t_f]$; $i = 1, \cdots, L$; $L \in \mathbb{N}_0$. H is used for model parameter identification problems. $\mathbf{x}(\cdot) : [t_0, t_f] \rightarrow \mathbb{R}^{n_x}$ is the state variable, $\mathbf{u}(\cdot) : [t_0, t_f] \rightarrow \mathbb{R}^{n_u}$ is the control variable, and $p \in \mathbb{R}^{n_p}$. The problem formulation can be defined as follow:

Objective Function:

$$\varphi(\mathbf{x}(t_0), \mathbf{x}(t_f), t_f, p) + \sum_{\mathbf{n=1}}^{\infty} (H(\zeta_i, \mathbf{x}(\zeta_i), \mathbf{u}(\zeta_i), p)) \tag{2.7}$$

Implicit Differential Equation (DAE):

$$\mathbf{f}(t, \mathbf{x}(t), \dot{\mathbf{x}}(t), \mathbf{u}(t), p) = 0 \tag{2.8}$$

Boundary Conditions:

$$\underline{\Psi} \leq \Psi(t_0, t_f, \mathbf{x}(t_0), \mathbf{x}(t_f), \mathbf{u}(t_0), \mathbf{u}(t_f), p) \leq \bar{\Psi} \tag{2.9}$$

State Constraints:

$$\underline{g} \leq \mathbf{g}(t, \mathbf{x}(t), \mathbf{u}(t), p) \leq \bar{g} \tag{2.10}$$

Box Constraints:

$$\underline{\mathbf{u}} \leq \mathbf{u}(t) \leq \bar{\mathbf{u}}, \quad \underline{p} \leq p \leq \bar{p} \tag{2.11}$$

The time is bounded $t_0 \leq t \leq t_f$. The equation Eq.(2.8) can be designed in two ways:

- Ordinary Differential Equations (ODEs):

  $$\mathbf{M}(t, \mathbf{x}(t), \mathbf{u}(t), p) \cdot \dot{\mathbf{x}}(t) - \mathbf{f}(t, \mathbf{x}(t), \mathbf{u}(t), p) = 0 \tag{2.12}$$

  with non-singular matrix $\mathbf{M}(\cdot)$

- Index-1 Differential Algebraic Equations (DAEs)

  $$\mathbf{F}(t, \mathbf{x_d}(t), \mathbf{y}(t), \dot{\mathbf{x_d}}(t), \mathbf{u}(t), p) = 0 \tag{2.13}$$

The state $x = (x_d(t), y) \in \mathbb{R}^{n_x}$ is separated as differential variables $x_d \in \mathbb{R}^{n_x - n_y}$ and algebraic variables $y \in \mathbb{R}^{n_y}$

As time from t_0 to t_f is infinite, thus are the control inputs to the optimal control problem. To solve this problem, the OCP is to be discretized and transformed into a finite-dimensional nonlinear optimization problem.

- Control Discretization: This discretization is achieved by approximating the control $\mathbf{u}$ by the function:

$$\mathbf{u_N}(t) = \mathbf{u_N}(t; c_1, \cdots, c_{N+k-2}) := \sum_{\mathbf{n=1}}^{\mathbf{N+k-2}} (\mathbf{c}_i \cdot \mathbf{B}_{ik}(t)) \tag{2.14}$$

  and discretizing the time as:

$$\mathbb{R}_u := t_1, t_2, \cdots, t_N, \quad N \geq 2 \tag{2.15}$$

  The vector $\mathbf{c}_i \in \mathbb{R}^{n_u}$, $i = 1, \cdots, N + k - 2$ are the de Boor points and the basis function $B_{ik}(\cdot)$ are the B-splines of order k.

- State Approximation: The differential equation (2.8) can be approximated by a suitable integration scheme such as Runge Kutta methods which gives $x_{\text{app}}(\cdot)$

$$\mathbf{x}_{app}(t_{i+1}) = \mathbf{x}_{app}(t_i) + h_i \cdot f(t_i, \mathbf{x}_{app}(t_i), \mathbf{u}(t_i)), \quad \mathbf{x}_{app}(t_0) = \mathbf{x}(t_0) \tag{2.16}$$

The Discretized Optimal Control Problem (DOCP) can then be formulated as:

Objective Function

$$\varphi(\mathbf{x}_{app}(t_0), \mathbf{x}_{app}(t_f), t_f, p) + \sum_{\mathbf{n=1}}^{\mathbf{L}} (H(\zeta_i, \mathbf{x}_{app}(\zeta_i), \mathbf{u}_N(\zeta_i), p)) \tag{2.17}$$

Discretized Boundary Conditions

$$\underline{\Psi} \leq \Psi(t_0, t_f, \mathbf{x}_{app}(t_0), \mathbf{x}_{app}(t_f), \mathbf{u}_N(t_0), \mathbf{u}_N(t_f), p) \leq \bar{\Psi} \tag{2.18}$$

Discretized State Constraints

$$\underline{g} \leq \mathbf{g}(t_i), \mathbf{x}_{app}(t), \mathbf{u}_N(t), p) \leq \bar{g} \tag{2.19}$$

Box Constraints

$$\underline{u} \leq \mathbf{c}_i \leq \bar{u}, \quad \underline{p} \leq p \leq \bar{p}, \quad i = 1, \cdots, N + k - 2 \tag{2.20}$$

To evaluate the problem, the Objective Function, Differential Equation, State Constraints, Boundary Conditions, Initial estimate of state, and the Initial estimate of the de Boor points are to be implemented as routines in Fortran. The function *OCPIDDAE* is then called with all the necessary initial parameters such as the initial and final time, upper and lower bounds of the constraints, and the de Boor grid. DOCP is then solved by Sequential Quadratic Programming (SQP) method implemented in the sqpfiltertoolbox, also developed by Univ.-Prof. Dr. rer. nat. Matthias Gerdts.

2.3 Parallel Computing

Parallel computing is a computing technique that involves executing multiple tasks simultaneously by breaking them into smaller parts and assigning each part to a separate processor or computing device. This approach allows for faster and more efficient processing of large and complex data sets, and has become increasingly popular in recent years due to advancements in hardware and software technologies.

Parallel computing can be implemented in a variety of ways, including using multiple cores on a single processor, using multiple processors on a single machine, or using a cluster of machines connected via a network. Each approach has its own strengths and weaknesses, and the choice of which approach to use will depend on factors such as the size of the data set, the complexity of the task, and the available hardware and software resources.

One of the key challenges of parallel computing is ensuring that the different parts of the task are properly synchronized and coordinated. This requires careful planning and design, as well as the use of specialized programming tools and techniques. Additionally, parallel computing can also be more complex and difficult to debug than traditional serial processing, which can make it more challenging for developers and researchers to work with.

As hardware and software technologies continue to advance, it is likely that parallel computing will become even more important in the years to come, enabling researchers and developers to tackle increasingly complex and demanding tasks with greater efficiency and speed. The three main laws that governs the parallel computing framework are described below. Parallel computing can be achieved by CPU multi-threading or General-Purpose

computing on Graphics Processing Units (GPGPU) which are then introduced later in this chapter.

2.3.1 Moore's Law

In 1965, Intel co-founder Gordon Moore predicted that the number of transistors on a chip would double roughly every two years, with a minimal rise in cost [Moo65]. This law has held true for several decades, and has been a driving force behind the rapid advances in computing technology over the past few decades.

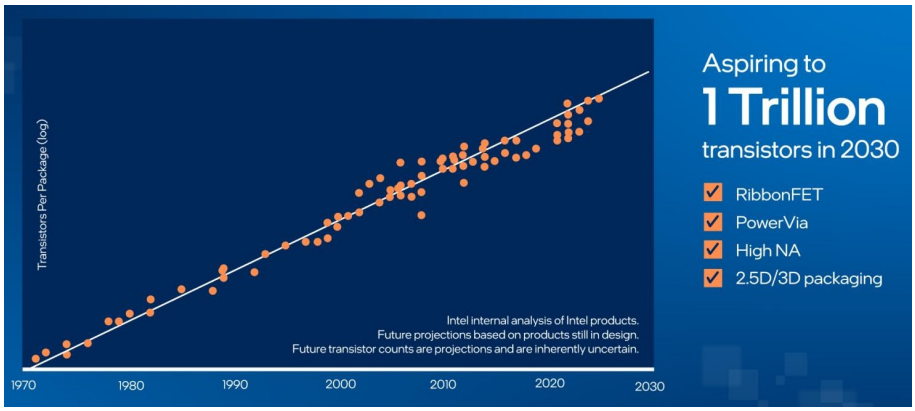


Figure 2.1: Moore's Law, Source: [Intel Website]

Moore's Law Figure 2.1 has been the guiding principle behind the development of the semiconductor industry and has allowed the technology industry to push the limits of computing power. It has led to a range of technological advancements, from smaller and more powerful computers to the development of mobile devices and the Internet of Things. However, the continued adherence to Moore's Law has faced challenges in recent years. The laws of physics dictate that it becomes increasingly difficult to pack more transistors onto a microchip as the size of the transistors approaches the atomic scale. As a result, the rate of increase in computing power has slowed down and the cost of developing new technologies has increased.

Despite these challenges, researchers and engineers have continued to push the limits of computing power, exploring new technologies such as quantum computing and neuromorphic computing. These emerging technologies offer new ways to overcome the limitations of traditional computing and could potentially usher in a new era of computing power.

2.3.2 Amdahl's Law

Amdahl's Law [Amd67] is a fundamental concept in parallel computing that describes the potential speedup that can be achieved when a program is executed on a parallel processing system. The law was formulated by Gene Amdahl, a computer architect and designer, in 1967.

The law states that the maximum possible speedup of a program running on a parallel system is limited by the portion of the program that cannot be parallelized, or the "serial fraction". This means that even if a program is run on an infinitely large number of processors, there is a limit to how much faster the program can be executed if a portion of the program cannot be parallelized.

Mathematically, Amdahl's Law can be expressed as:

$$S(N) = 1/(1 - p + (p/N)) \tag{2.21}$$

where, S is the speedup gain, p is the proportion of the program that can be parallelized, N is the number of processors, and $(1 - p)$ is the proportion of the program that cannot be parallelized.

From the equation, it is clear that as the number of processors (N) increases, the speedup of the program becomes more limited by the serial fraction of the program. This means that the benefits of parallel processing diminish as more processors are added, and there is a point beyond which additional processors do not result in significant performance gains.

Amdahl's Law highlights the importance of identifying the portion of a program that cannot be parallelized and optimizing it for maximum efficiency. It also emphasizes the need for careful analysis and planning when developing parallel programs, as well as the importance of choosing the right balance between the number of processors and the amount of parallelization in order to achieve maximum performance.

2.3.3 Gustafson's Law

Gustafson's Law [Gus88] is a complementary concept to Amdahl's Law, developed by John Gustafson in 1988. While Amdahl's Law emphasizes the limitations of parallel processing due to the serial fraction of a program, Gustafson's Law focuses on the potential benefits of parallel processing when scaling up the size of a problem.

Gustafson's Law states that the speedup of a program running on a parallel system can be increased by scaling up the size of the problem to be solved, rather than just increasing the number of processors. This means that as the size of the problem increases, more processing power can be used to solve the problem faster.

Gustafson's Law can be denoted mathematically as:

$$S(N) = N + (1 - N) * p \tag{2.22}$$

where, S is the speedup gain, N is the number of processors and p is the proportion of the program that can be parallelized.

From the equation, it is clear that as the size of the problem increases (i.e., as $p \to 1$), the speedup of the program becomes more dependent on the number of processors. This means that the benefits of parallel processing increase as the size of the problem scales up, and there is no limit to the potential speedup that can be achieved.

2.3.4 GPGPU

General-purpose computing on graphics processing units (GPGPU) refers to the use of a graphics processing unit (GPU) to perform computations that are typically performed by the central processing unit (CPU). GPUs were originally designed for rendering graphics in video games and other visual applications, but their massively parallel architecture makes them well-suited for certain types of scientific, engineering, and financial computations.

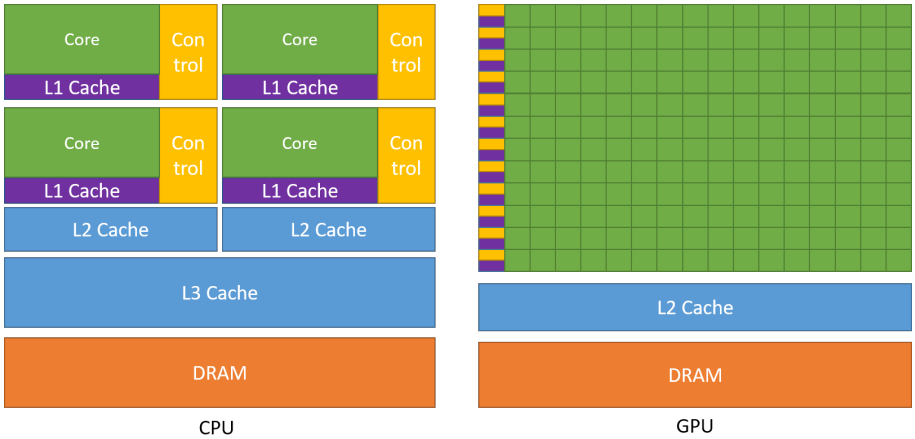


Figure 2.2: GPU vs CPU architecture, Source: [CUDA Programming Guide]

GPUs consist of many processing cores, which can perform computations simultaneously. While CPUs typically have a few cores optimized for serial execution, GPUs have hundreds or thousands of cores optimized for parallel execution Figure 2.2. This allows GPUs to

process large amounts of data in parallel, which can result in significant speedup over traditional CPU-based computations.

To utilize the power of GPUs for general-purpose computing, specialized programming models and languages have been developed. One popular programming model is CUDA (Compute Unified Device Architecture) [CUDA Programming Guide], which is a parallel computing platform and programming model developed by Nvidia for use with their GPUs. CUDA allows programmers to write code in a C-like language and to execute it on Nvidia GPUs. Other popular programming models for GPGPU include OpenCL [Khronos Group OpenCL], which is an open standard for heterogeneous computing, and DirectCompute [Direct Compute], which is a Microsoft API for parallel computing on GPUs.

GPGPU has been used in a wide range of applications, including scientific simulations, data analytics, machine learning, and finance. For example, GPGPU has been used to accelerate the training of neural networks, which is a computationally-intensive task that requires many matrix operations. GPGPU has also been used to accelerate simulations of fluid dynamics, which require the solution of large systems of partial differential equations.

2.3.5 Compute Unified Device Architecture (CUDA)

CUDA is a parallel computing platform and programming model developed by NVIDIA for accelerating computing performance on GPUs. CUDA enables developers to write programs in C, C++, and Fortran that can run on NVIDIA GPUs, allowing for massive parallelism that can speed up computations by orders of magnitude [CUDA Programming Guide].

One of the key concepts in CUDA programming is the concept of kernels. Kernels are small, highly parallel functions that are executed on the GPU. They are written in C/C++ and are called from the host code. When a kernel is called, it is executed by a large number of threads on the GPU, with each thread executing the same code on different data.

Algorithm 1 is a small example of CUDA pseudo code that demonstrates the use of kernels. The three arrays A, B, and C are first allocated on host (CPU) memory and then arrays A and B are initialized with some data. We then allocate memory for these arrays on the device (GPU) and the data is copied from the host to the device. The block size and grid size for the kernel launch are set after which the kernel *addArrays* with the specified block and grid size, passing in the device pointers to arrays A, B, and C, as well as the size of the arrays is launched. The kernel adds corresponding elements of A and B and stores the

result in C. Finally, the data is copied from the device to the host using and the memory is freed from the host and device. CUDA can be used for a wide variety of applications, including scientific simulations, computer vision, machine learning, and more.

Algorithm 1 CUDA Array Addition

procedure AddArray
$i \leftarrow Block.ID.x * Block.Dim.x + Thread.ID.x$
if $i < N$ **then**
$C[i] \leftarrow A[i] + B[i]$
end if
end procedure

procedure main
$N \leftarrow 1000$
$A \leftarrow array[N] * sizeof(float)$
$B \leftarrow array[N] * sizeof(float)$
$C \leftarrow array[N] * sizeof(float)$
$dA \leftarrow array[N] * sizeof(float)$
$dB \leftarrow array[N] * sizeof(float)$
$dC \leftarrow array[N] * sizeof(float)$

for $i = 0$ to $N - 1$ **do**
$A[i] \leftarrow i$
$B[i] \leftarrow N - i$
end for

$(dA \leftarrow A)$: memory copy from host to device
$(dB \leftarrow B)$: memory copy from host to device

$blockSize \leftarrow 256$
$gridSize \leftarrow \frac{(N+blockSize-1)}{blockSize}$

call $addArrays <<< gridSize, blockSize >>> (dA, dB, dC, N)$

$(C \leftarrow dC)$: memory copy from device to host
end procedure

2.3.6 CPU Multithreading

CPU multi-threading is a technique used to improve the performance of applications that can be parallelized. With multi-threading, multiple threads of execution can be created to run concurrently on a CPU, allowing for parallel processing of tasks.

One popular library for implementing multi-threading in C/C++ is OpenMP. OpenMP provides a set of compiler directives and library functions that allow developers to easily create parallel applications for multi-core CPUs.

Algorithm 2 OpenMP Array Sum

```
procedure MAIN
    num_threads ← 4
    sum ← 0

    #pragma omp parallel num_threads
    thread_id ← omp_get_thread_num()
    thread_sum ← 0
    for i = thread_id to 100 with a step of num_threads do
        thread_sum ← thread_sum + i
    end for

    #pragma omp critical
    sum ← sum + thread_sum
end procedure
```

Algorithm 2 is a small example of OpenMP pseudo code that demonstrates the use of multithreading. In this example, the number of threads is set to 4. The *#pragma_omp_parallel* directive is used to create a parallel region, indicating that the following code should be executed concurrently by the specified number of threads. Each thread gets its own ID using *omp_get_thread_num*(). Each thread then calculates its own sum over a portion of the loop, adding every *num_threads* element. The results are then accumulated into a shared variable sum using the *#pragma_omp_critical* directive to ensure that only one thread at a time is accessing this shared resource, to avoid race conditions. Finally, the value of sum is printed to the console.

OpenMP handles the details of creating and managing the threads, making it easier for developers to write parallel code. With OpenMP, developers can focus on the parallelism of their code, while the library handles the details of scheduling and synchronization.

OpenMP can be used for a wide variety of applications, including scientific simulations, image processing, and more. With the power of OpenMP, developers can take advantage of the full computing power of multi-core CPUs, improving the performance of their applications.

2.4 Computing Hardware

The results in this thesis are computed using multiple machines for GPU and CPU computations, as given in Table 2.1 and Table 2.2.

Table 2.1: GPU Machines

Machine Name	GPU	CUDA version	Driver version
125	GTX 1080 TI	11.4	470.161.03
12	Titan V	11.4	470.161.03
16	RTX 2080 TI	11.4	470.161.03

Table 2.2: CPU Machines

Machine Name	CPU(s)	S:C:T	Processors
181	10	1:4:2	Xeon(R) CPU E5-1620 v3
125	20	1:10:2	Xeon(R) CPU E5-2687W v3
231	56	1:7:1	Xeon(R) CPU E5-2680 v4

3 Problem Statement

In this chapter, the thesis explores the kinematics and dynamics of free-floating robots, as well as delve into the original motion planning problem.

3.1 Free Floating Robots

This section presents the free-floating mode of robot manipulators that are mounted on non-actuated satellite bases. The mode excludes any external actions, and the robot's dynamic behavior is solely determined by the orbital free multi-body dynamics.

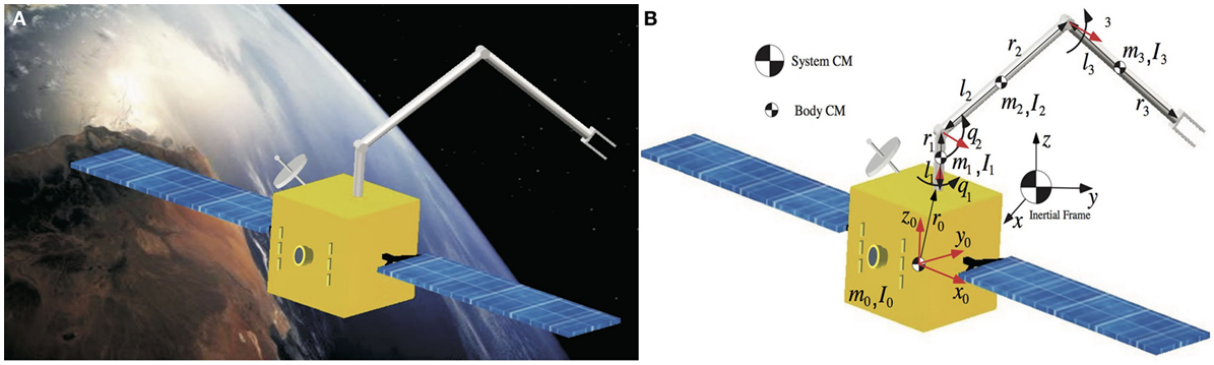


Figure 3.1: Free Floating Robot, [NP17]

The free-floating mode can be achieved by deactivating all spacecraft thrusters, which will prevent the system's center of mass from translating. In this mode, the spacecraft will translate and rotate in response to the manipulator motions. To maintain the spacecraft attitude during manipulator motions and avoid communication loss with ground stations, as well as solar panel disorientation, momentum control devices (MCDs) such as reaction wheels or momentum gyros are used. While the system's center of mass remains stationary, these MCDs actively control the spacecraft attitude [PAM21]. If MCDs are used, then the mode is referred to as partial free-floating. Both the free-floating and partial free-floating modes are preferred during grasping, as they eliminate sudden motions due to thrusters and conserve propellant and power.

3.1.1 Geometry

In robotics, it is typical to first describe the structure of the robot in terms of its kinematics. Each body is then associated with a reference frame denoted by $(O^i, \mathbf{e}^i)$. As seen in Figure 3.2, the free-floating robot also has a base body with a reference frame denoted by

$(O^O, \mathbf{e}^O)$. The frames for the robot links are attached at the respective joints in terms of the Denavit and Hartenberg (DH) notation. The end-effector also has a reference frame denoted by $(O^e, \mathbf{e}^e)$.

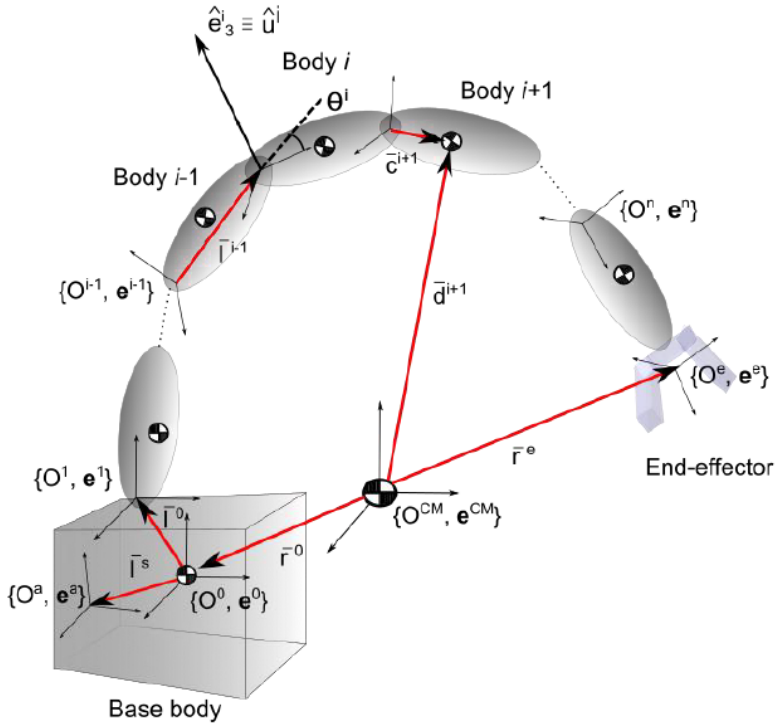


Figure 3.2: Robot Geometry

The Center of Mass (CM) of the complete system has a reference frame denoted by $(O^{CM}, \mathbf{e}^{CM})$ and is also an inertial frame $(O^I, \mathbf{e}^I)$ for the free-floating robot. The motion of the whole multi-body system is restricted around the CM.

3.1.2 Kinematics

In this section, the thesis examines the kinematic properties of free-floating robots based on the geometry developed in Section 3.1.1. The position vector of the base body from the CoM is $\mathbf{r}^0 \in \mathbb{R}^3$. The position of the (i+1)'th link and CM of (i+1)'th link from link i are ${}^{i+1}\mathbf{I}^i \in \mathbb{R}^3$ and ${}^i\mathbf{c}^{i+1} \in \mathbb{R}^3$, respectively.

The pose - position ($\mathbf{r}^i \in \mathbb{R}^3$) and orientation matrix ($\mathbf{A}^{i,I} \in \mathbb{R}^{3\times3}$) of the i'th link in the inertial frame $(O^I, \mathbf{e}^I)$ is as follows:

$$\mathbf{r}^i = \mathbf{r}^0 + \mathbf{A}^{I,0} \cdot {}^1\mathbf{I}^0 + \sum_{\mathbf{j=1}}^{\mathbf{i}} (\mathbf{A}^{I,j} \cdot {}^J\mathbf{I}^{J-1}) \tag{3.1}$$

$$\mathbf{A}^{i,I} = \mathbf{A}^{i,i-1} \cdots \mathbf{A}^{1,0} \cdot \mathbf{A}^{0,I} \tag{3.2}$$

Here, $0 \leq i \leq n$, where n+1 is the total number of bodies. The Degrees of Freedom (DoF) of the system are denoted by the joint angles $\mathbf{q} \in \mathbb{R}^n$.

- Forward Kinematics:

  The end effector pose is calculated with the help of forward kinematics.

  $$\mathbf{r}^e = \mathbf{r}^0 + \mathbf{A}^{I,0} \cdot {}^1\mathbf{I}^0 + \sum_{j=1}^{n} (\mathbf{A}^{I,j}(q_j) \cdot {}^j\mathbf{I}^{j-1}) \tag{3.3}$$

  $$\mathbf{A}^{e,I} = \mathbf{A}^{n,n-1}(q_{n-1}) \cdots \mathbf{A}^{1,0} \cdot \mathbf{A}^{0,I} \tag{3.4}$$

  It is important to note that the calculations for a robot's forward kinematics depend on both the robot's design parameters and the measured joint configuration. However, these values are never known with complete accuracy, which means that the computed values for forward kinematics will always differ slightly from the actual values. Although these differences may be small, they are still significant and cannot be ignored.

- Differential Kinematics:

  Differential kinematics can be used to calculate the end-effector velocities (linear $\mathbf{v}^e \in \mathbb{R}^3$ and angular $\boldsymbol{\omega}^e \in \mathbb{R}^3$) of the robot. The base body's linear and angular velocities are denoted by $\mathbf{v}^0 \in \mathbb{R}^3$ and $\boldsymbol{\omega}^0 \in \mathbb{R}^3$, respectively.

  $$\mathbf{v}^e = \mathbf{v}^0 + \sum_{i=0}^{n} (\boldsymbol{\omega}^i \cdot {}^{i+1}\mathbf{I}^i) \tag{3.5}$$

  with $\boldsymbol{\omega}^i = \boldsymbol{\omega}^0 + \sum_{j=1}^{i} (\boldsymbol{\omega}^{j(j-1)}) = \boldsymbol{\omega}^0 + \sum_{j=1}^{i} (\dot{q}_j \cdot \boldsymbol{u}^j)$

  $$\boldsymbol{\omega}^e = \boldsymbol{\omega}^0 + \sum_{i=0}^{n} (\dot{q}_i \cdot \boldsymbol{u}^i) \tag{3.6}$$

$\boldsymbol{u}^j$ is the unit vector in the direction of the axis of rotation of joint j. Equations (3.5) and (3.6) is then expressed in matrix form:

$$\dot{\mathbf{x}}^e = \mathbf{J}_b \cdot \dot{\mathbf{x}}^0 + \mathbf{J} \cdot \dot{\mathbf{q}} \tag{3.7}$$

With $\dot{\mathbf{x}}^e = [\mathbf{v}^{eT} \quad \boldsymbol{\omega}^{eT}]^T \in \mathbb{R}^6$, $\dot{\mathbf{x}}^0 = [\mathbf{v}^{0T} \quad \boldsymbol{\omega}^{0T}]^T \in \mathbb{R}^6$. Here, $\mathbf{J}$ is the geometric Jacobian of the robot and $\mathbf{J}_b$ is the base Jacobian. $\mathbf{E}_3 \in \mathbb{R}^{3\times3}$ is an Identity matrix.

$$\mathbf{J}_b = \begin{bmatrix} \mathbf{E}_3 & -\mathbf{r}^{e,0} \\ 0 & \mathbf{E}_3 \end{bmatrix} \tag{3.8}$$

3.1.3 Dynamics

The momentum of a free-floating robotic system remains conserved when the external forces acting on it are zero. The equation of motion for a free-floating system can be obtained using the Lagrangian approach [DP93].

The equation of motion for free-floating system can be derived from the free-flying system with $\mathbf{F}_0$ are the forces acting on the base body and $\mathbf{F}_e$ are the forces when robot interacts with the environment, $\boldsymbol{\tau}$ are the torques acting on the joints:

$$\mathbf{M} \cdot \begin{bmatrix} \ddot{\mathbf{x}}^0 \\ \ddot{\mathbf{q}} \end{bmatrix} + \mathbf{C} \cdot \begin{bmatrix} \dot{\mathbf{x}}^0 \\ \dot{\mathbf{q}} \end{bmatrix} = \begin{bmatrix} \mathbf{F}_0 \\ \boldsymbol{\tau} \end{bmatrix} + \begin{bmatrix} \mathbf{J^T}_b \\ \mathbf{J^T} \end{bmatrix} \mathbf{F^e} \tag{3.9}$$

$\mathbf{M} = \begin{bmatrix} \mathbf{M}_b & \mathbf{M}_{bm} \\ \mathbf{M^T}_{bm} & \mathbf{M}_m \end{bmatrix}$ the inertia matrix, $\mathbf{C} = \begin{bmatrix} \mathbf{C}_b & \mathbf{C}_{bm} \\ \mathbf{C}_{mb} & \mathbf{C}_m \end{bmatrix}$ the Coriolis and centrifugal matrix.

For a free-floating system, there are no external forces on the robot and thus, $\mathbf{F}_0 = 0$ (conservation of momentum) and $\mathbf{F}_e = 0$. The Equation (3.9) then becomes [YT93]:

$$\ddot{\mathbf{x}}^0 = -\mathbf{M}_b^{-1}\mathbf{M}_{bm}\ddot{\mathbf{q}} - \mathbf{M}_b^{-1}\mathbf{C}_b\dot{\mathbf{x}}^{\mathbf{0}} - \mathbf{M}_b^{-1}\mathbf{C}_{bm}\dot{\mathbf{q}} \tag{3.10}$$

$$\boldsymbol{\tau} = \mathbf{M}^*\ddot{\mathbf{q}} + \mathbf{C}^*\dot{\mathbf{q}} \tag{3.11}$$

$$\mathbf{M}^* = \mathbf{M}_m - \mathbf{M}_{bm}^T \mathbf{M}_b^{-1} \mathbf{M}_{bm} \tag{3.12}$$

$$\mathbf{C}^* = \mathbf{C}_m - \mathbf{M}_{bm}^T \mathbf{M}_b^{-1} \mathbf{C}_{bm} - (\mathbf{C}_{mb} - \mathbf{M}_{bm}^T \mathbf{M}_b^{-1} \mathbf{C}_b) \mathbf{M}_b^{-1} \mathbf{M}_{bm} \tag{3.13}$$

Equations (3.5) and (3.6) reveal that the end-effector's linear and angular velocities are functions of the joint rates and the base body's angular velocity. It has been shown that the base body's angular velocity also depends on the joint rates [Pap90]. This dependence can be numerically integrated to obtain the base body orientation, which is dependent on the path taken in joint space. Since the end-effector's position in the inertial frame is a function of the base body orientation, the end-effector's position is also a function of the path taken. However, as the calculation of the base body orientation cannot be done analytically and is path dependent, it introduces a non-holonomic characteristic to the free-floating system.

3.2 Original Motion Planning Problem

To reduce the risk of failure and increase the safety, it is important to choose control algorithms that accurately satisfy the constraints for motion planning. This gives rise to a nonlinear constrained control problem. For simplicity, consider the original point-to-point motion control problem where the end effector at time $t_0 \in \mathbb{R}$ is at pose $\mathbf{x}^e(0) = [\mathbf{r}^{eT}(0) \quad \boldsymbol{\psi}^{eT}(0)]$ and at time $t_f \in \mathbb{R}$ is at pose $\mathbf{x}_{des}^e(t_f) = [\mathbf{r}_{des}^{eT}(t_f) \quad \boldsymbol{\psi}_{des}^{eT}(t_f)]$. The planning is defined next [LMO18]:

The cost function taken into consideration is to minimize the mechanical energy

$$J = \min_{\dot{q}(t), \boldsymbol{\tau}(t)} \int_{t_0}^{t_f} (\boldsymbol{\tau}^T(t) \cdot \dot{\mathbf{q}})^2 dt \tag{3.14}$$

The free-floating robot momentum conservation equations yield [YWH16]:

$$\dot{\mathbf{x}}^0(t) = -\mathbf{M_b^{-1}} \mathbf{M_{bm}} \cdot \dot{\mathbf{q}}(t) \tag{3.15}$$

$\mathbf{M_b}$, $\mathbf{M_{bm}}$ are the inertia matrices for the base body and the coupling between the base and arm respectively. The end-effector pose $\mathbf{x}^e(t_f)$ and velocity $\dot{\mathbf{x}}^e(t_f)$ at the final time

should reach a desired pose $\mathbf{x}^e_{des}(t_f)$ with a desired velocity $\dot{\mathbf{x}}^e_{des}(t_f)$, which results in an equality constraints:

$$\mathbf{x}^e(t_f) = \mathbf{x}^e_{des} \tag{3.16}$$

$$\dot{\mathbf{x}}^e(t_f) = \dot{\mathbf{x}}^e_{des} \tag{3.17}$$

and the boundary conditions that need to be satisfied are:

$$\mathbf{q}_{min}(t) \leq \mathbf{q}(t) \leq \mathbf{q}_{max}(t) \tag{3.18}$$

$$\dot{\mathbf{q}}_{min}(t) \leq \dot{\mathbf{q}}(t) \leq \dot{\mathbf{q}}_{max}(t) \tag{3.19}$$

$$\boldsymbol{\tau}_{min}(t) \leq \boldsymbol{\tau}(t) \leq \boldsymbol{\tau}_{max}(t) \tag{3.20}$$

[DP93] states that the free-floating system is non-holonomic. To solve the continuous time OCP, the thesis use the single shooting transcription method to reduce it to a discretized version. Controls are discretized using B-splines and Equation (3.15) is approximated using an integration scheme like Runge-Kutta. The time grid is divided into N via-points, and de Boor points c_i are vectors in $\mathbb{R}^n$, where n is the number of available joints. B-spline basis functions $B_{ik}(\cdot)$ of order k are used, where $(1 \leq i \leq N+k-1)$ as mentioned in Section 2.2.2.

Thus, the DOCP is:

$$\min_{\mathbf{c}} J(t_f, \mathbf{q}(t, \mathbf{c}), \boldsymbol{\tau}(t)) \tag{3.21}$$

subject to

$$\mathbf{g}_i(\mathbf{q}(t, \mathbf{c}), \boldsymbol{\tau}(\boldsymbol{t})) = 0 \tag{3.22}$$

$$\mathbf{h}_i(\mathbf{q}(t, \mathbf{c}), \boldsymbol{\tau}(t)) \leq 0 \tag{3.23}$$

and $\boldsymbol{\tau}(t)$ is solved by the equation in (3.11):

$$\boldsymbol{\tau}(t) = \mathbf{M}^* \cdot \ddot{q}(t, \mathbf{c}) + \mathbf{C}^* \cdot \dot{q}(t, \mathbf{c}) \tag{3.24}$$

This discretized problem is solved using the Sequential Quadratic Programming method which requires the Jacobians of the cost function (J) and the constraints (g and h).

As seen in the introduction, there are two workloads which takes high execution time which are as follows:

3.2.1 Calculation of Jacobians

To solve the nonlinear programming problem arising from the optimal control framework, the Jacobians of the constraints and the objective function are required. As the number of constraints exceeds the number of optimization variables [Lam10], the sensitivity based approach and finite difference approach are the most suitable [Ger12].

3.2.2 Computation of Robot Dynamics

To calculate the derivatives as discussed in the previous section, the constraints need to be evaluated. Determining the constraints involves the core robot calculation, which requires numerically integrating Equation (3.15). This is computationally expensive. For computing multiple queries using an offline motion planner to create a database for warm starting of online motion planner, the numerical integration can be parallelized for all the queries using GPGPU computation.

The next chapters deal with novel methodologies for the above workloads and provide solutions with better time complexities.

4 Computation of Robot Dynamics

An offline motion planner is used to create trajectories for various final end-effector poses and velocities. These trajectories can then be used to warm-start an online (real-time) motion planner. Giving good initial guesses to the optimization planner is of utmost importance, and the feasibility and optimality of the solution depends on the initial guess. With a bad initial guess, the motion planner could reach an unfavorable locally optimal solution or simply not converge. Thus, generating feasible warm-starting solutions for an online motion planner with an offline motion planner will guarantee a robust and near-global optimal solution for the problem. As solving the OCP is time-consuming, such multiple queries could be parallelized using either CPU multi-threading or GPU parallelization. This chapter focuses on the parallelization of the integration of robot dynamics Equation 3.15.

The ideology behind this is motivated by [CEK16], where the humanoid motion planning is solved as a semi-infinite optimization program, and the core robot and constraints computations are calculated using GPUs. However, they do not consider the flight mode where the system becomes non-holonomic. The challenge in a non-holonomic system is the integration of the system dynamics within the complete time frame. A single instance of the integration cannot be parallelized. Whereas in the time frame where contact is present (no flight phase - holonomic system), the computation of end-effector pose is an analytical equation and thus can be parallelized on the discretized time steps.

The authors in [YWP16] observed a significant speedup in computing forward dynamics using different algorithms on the Nvidia CUDA GPU platform, especially for a high number of links. [YWP18] leveraged sparsity within the joint-space inertia matrix to compute forward dynamics on GPU. [PNB22] developed a library for forward and inverse dynamics computation with gradients, resulting in a 2.5x speedup compared to multi-threaded CPU implementation, which is available as open-source software. [LVY13] implemented the humanoid robot dynamics using CPU multi-threading. However, none of the research presented a method for free-floating robot dynamics.

This chapter presents a novel method to compute the integration and the dynamics of the free-floating robot system in parallel for multiple queries. The parallelization is accomplished with CPU multi-threading as well as with the use of GPU. Although a single query's execution on a GPU is slower than on a CPU, parallelizing thousands of queries on a GPU provides a significant speedup compared to sequential execution on a CPU.

4.1 Methodology

In OCP, the systems dynamics are represented by Ordinary Differential Equations (ODE). To find solutions of such an ODE, numerical integration techniques are developed such as one-step methods, multi-step methods, or extrapolation methods. The most common methods are the one-step methods and particularly the Runge-Kutta methods. The Runge-Kutta methods use a higher order approximation in each step, thus reducing the error. The basic principle within the Runge-Kutta methods is as follows [Ger12].

For a general ODE initial value problem, where $\mathbf{z} \in \mathbb{R}^{n_z}$ is the solution vector and $\mathbf{f}$ is the ODE:

$$\dot{\mathbf{z}}(t) = \mathbf{f}(t, \mathbf{z}(t)) \tag{4.1}$$

To find the values of $\mathbf{z}$ which satisfies the ODE from the initial time t_0 to final time t_f. The infinite time interval from t_0 to t_f is discretized by a grid in N intervals:

$$\mathbb{G}_N := \{t_0 < t_1 < \cdots < t_{N-1} < t_N = t_f\} \tag{4.2}$$

Consider the integral representation of Equation (4.1):

$$\mathbf{z}(t_{i+1}) - \mathbf{z}(t_i) = \int_{t_i}^{t_{i+1}} \mathbf{f}(t, \mathbf{z}(t))dt \tag{4.3}$$

Numerically integrating Equation (4.1) provides approximate values of $\mathbf{z}$. Let the approximation of $\mathbf{z}$ at time t_{i+1} be $\mathbf{z}_N(t_{i+1})$. Then $\mathbf{z}_N(t_{i+1})$ can be represented with the Runge Kutta scheme as:

$$\mathbf{z}_N(t_{i+1}) := \mathbf{z}_N(t_i) + h_i \cdot \sum_{j=1}^{s}(b_j \cdot k_j) \tag{4.4}$$

$$k_j = f(t_i + (c_j \cdot h_i), \mathbf{z}_N(t_i) + h_i \cdot \sum_{l=1}^{s}(a_{jl}, k_l(t_i, \mathbf{z}_N(t_i); h_i)) \tag{4.5}$$

where $s \in \mathbb{N}$ are the s stages of the Runge Kutta method, h_i is the step size. b_j, c_j, a_{jl} are the coefficients such that $[a_{jl}]$ is the Runge-Kutta matrix, and b_j, c_j are the weights and nodes respectively. The typical arrangement of this data is within a mnemonic tool commonly referred to as a Butcher table. The step size h_i can stay fixed or can be adjusted to increase the efficiency of the algorithm, depending on the accuracy required for the solution.

c_1	$a_{11}\ a_{12} \cdots a_{1s}$
c_2	$a_{21}\ a_{22} \cdots a_{2s}$
$\vdots$	$\vdots\ \vdots\ \ddots\ \vdots$
c_s	$a_{s1}\ a_{s2} \cdots a_{ss}$
	$b_1\ b_2 \cdots b_s$

Table 4.1: General Butcher Table

Dormand-Prince Runge-Kutta algorithm (DOPRI5) [DP80] is a seven stage method and computes six function evaluation per step. These six evaluations calculate the fourth and fifth order accurate solutions. The difference between these solutions is taken as the error of the fourth order solution. It is the default method within MATLAB and GNU Octave for the function call "*ode45*" [ODE45]. The seven stages are as follows:

$$k_1 = f(t_n, y_n)$$

$$k_2 = f(t_n + \frac{h}{5}, y_n + \frac{h}{5}k_1)$$

$$k_3 = f(t_n + \frac{3h}{10}, y_n + \frac{3h}{40}k_1 + \frac{9h}{40}k_2)$$

$$k_4 = f(t_n + \frac{4h}{5}, y_n + \frac{44h}{45}k_1 - \frac{56h}{15}k_2 + \frac{32h}{9}k_3)$$

$$k_5 = f(t_n + \frac{8h}{9}, y_n + \frac{19372h}{6561}k_1 - \frac{25360h}{2187}k_2 + \frac{64448h}{6561}k_3 - \frac{212h}{729}k_4)$$

$$k_6 = f(t_n + h, y_n + \frac{9017h}{3168}k_1 - \frac{355h}{33}k_2 + \frac{46732h}{5247}k_3 + \frac{49h}{176}k_4 - \frac{5103h}{18656}k_5)$$

$$k_7 = f(t_n + h, y_n + \frac{35h}{384}k_1 + \frac{500h}{1113}k_3 + \frac{125h}{192}k_4 - \frac{2187h}{6784}k_5 + \frac{11h}{84}k_6)$$

4.2 Implementation

This section demonstrates the parallelization of multiple queries for integrating the system dynamics of the free floating robot. The dynamics as developed in Section 3.1.3 are used within the system dynamics (ODE) of the OCP in Section 3.2.

To compute multiple instances of the integration and dynamics, the thesis implemented the solver using Boost Odeint (version 1.81) [Odeint] and OpenMP API 5.2 [OpenMP] in C++.

In Algorithm 3, the inputs are the current position and orientation of the base body (*pose*), the current velocity of the base body (*vel*), the joint angles (q), joint velocities ($\dot{q}$), and joint torques (τ). The rhs vector is the right-hand side of the ODE. All vectors are concatenations of all the values for each instance. The total number of threads is created according to the number of instances to be computed.

Algorithm 3 CPU Parallelization of Integration

1: **for** $threadID \leftarrow 0$ to $numThreads$ **do**
2: **for** $t \leftarrow t_0$ to t_f **do**
3: $state \leftarrow (pose, vel, q, \dot{q}, \tau)$
4: Update: $rhs \leftarrow f(state, t)$
5: $state \leftarrow$ Integrate(rhs, $state$, t_0, t_f)
6: **end for**
7: $(pose, vel, q, \dot{q}, \tau) \leftarrow state$
8: **end for**

Algorithm 4 computes the integration and robot dynamics on GPU, and uses Boost Odeint (version 1.81) [Odeint], ArrayFire [ArrayFire], and Thrust CUDA (version 11.7) [Thrust]. Input parameters include base body pose (*pose*) and velocity (*vel*), joint angles (q), velocities ($\dot{q}$) and torques (τ). Data is transferred between host and device with $d \leftarrow h$ and $h \leftarrow d$ notation. The rhs vector, representing the right-hand side of the ODE, and integration occur on the GPU.

Algorithm 4 GPU Parallelization of Integration

1: **for** $t \leftarrow t_0$ **to** t_f **do**
2: $(d \leftarrow h)$: $state \leftarrow (pose, vel, q, \dot{q}, \tau)$
3: CUDA: $rhs \leftarrow f(state, t)$
4: CUDA: $state \leftarrow$ Integrate(rhs, $state$, t_0, t_f)
5: **end for**
6: $(h \leftarrow d)$: $(pose, vel, q, \dot{q}, \tau) \leftarrow state$

4.3 Results

While computing in parallel on the CPU, it is necessary to take care of keeping the code thread-safe, managing memory access between each thread, allocating memory for shared and private variables for threads, avoiding race conditions, synchronizing threads, load balancing, and minimizing overhead. As shown in Figure 4.1 (machine 181, Table 2.2),

for a final time of 10 seconds, a maximum speedup of approximately 7.5x, and for 50 seconds, a speedup of approximately 7.6x is achieved. The x-axis represents the number of threads initiated.

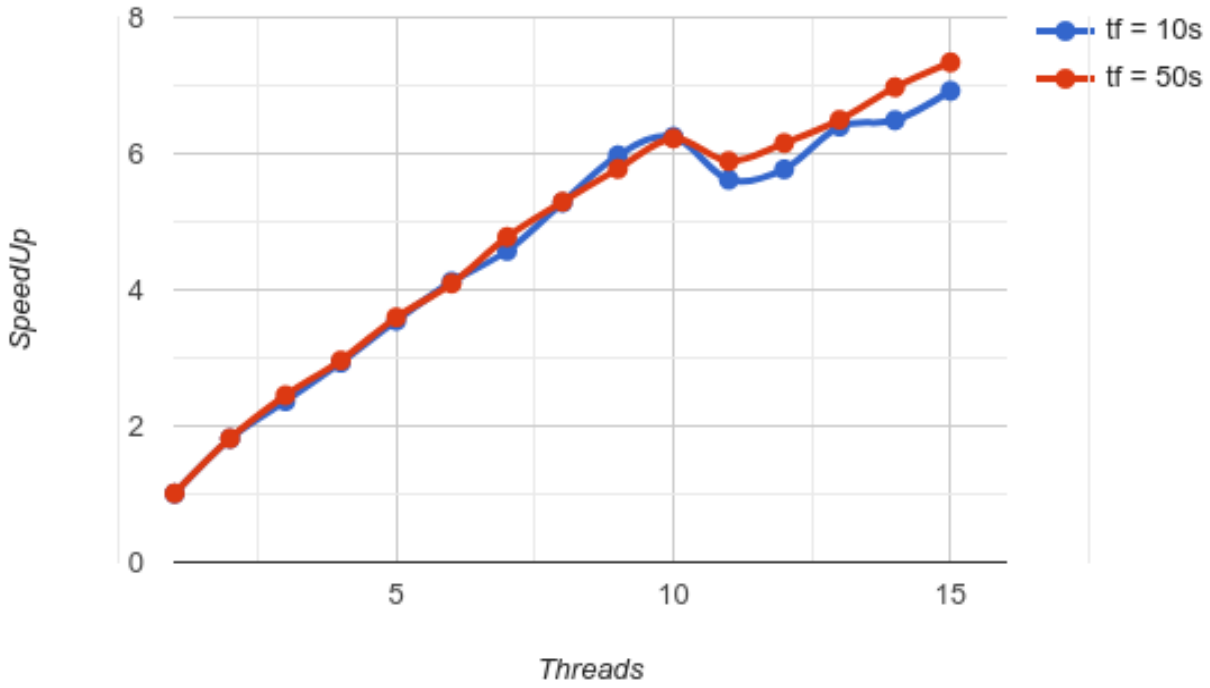


Figure 4.1: Speed Up vs Number of Threads for Machine 181

The speedup at 11 threads decreases compared to 10 threads because the OpenMP API initiates a maximum of 8 threads for two steps. For the second step, only one thread has a workload, which increases the computation time. In Figure 4.2 (machine 231, Table 2.2), the maximum speedup achieved is more than 15x when 56 threads are initiated. As the number of threads increases, the acceleration in speed reduces due to the additional time needed for thread initialization overhead and thread synchronization.

Similar to parallelization with CPU multi-threading, this thesis developed and implemented a novel method for the integration and robot dynamics computation on GPUs with the machines presented in Table 2.1. The challenges encountered with GPU parallelization are different from those with CPU parallelization, including synchronization, overhead, thread divergence, and others. After computation, it was found that one instance of the integration on the GPU is approximately 250x slower than the serial CPU version.

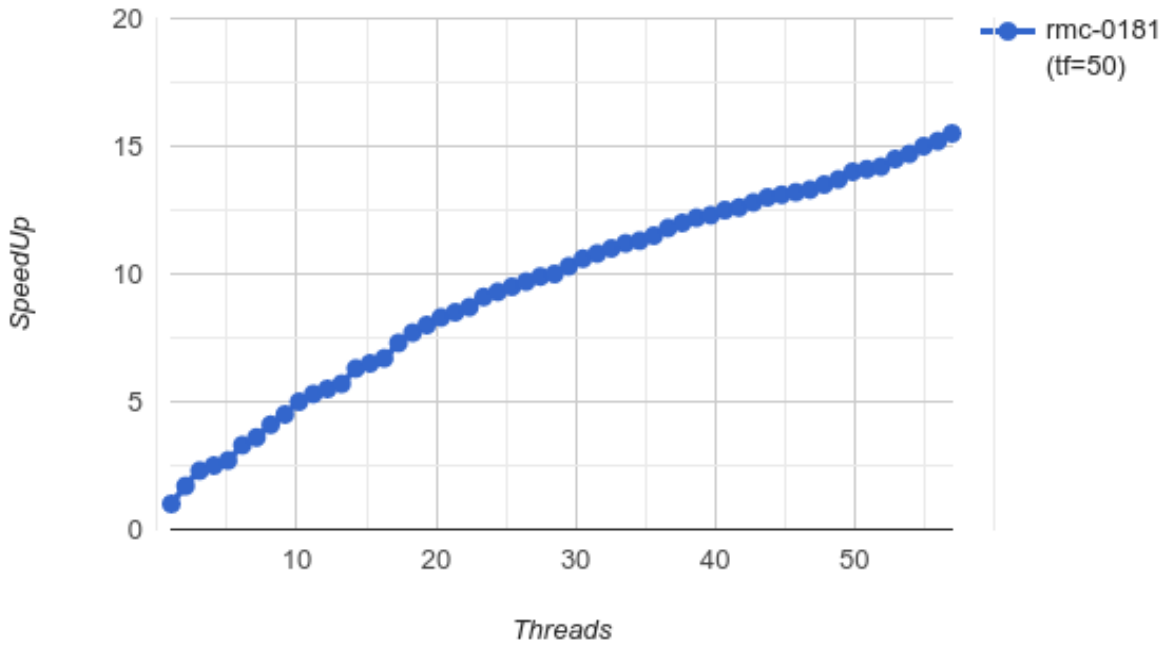


Figure 4.2: SpeedUp vs Number of Threads for Machine 231

5 Computation of Jacobians

As shown in Section 3.2.1, to compute the Jacobians of the constraints, the derivatives of the constraints needs to be evaluated. From the various methods, this thesis focuses on sensitivity based approach and finite difference approach.

5.1 Methodology

This section presents the methodology for both the sensitivity based approach and the finite difference method, along with the mathematical background required for understanding these techniques. The section aims to provide a comprehensive understanding of these methods and their application.

5.1.1 Sensitivity-based approach

In the reduced discretization in Section 3.2, the system dynamics $\mathbf{f}$ are solved using the one-step method and are not imposed as equality constraints. It is exploited that the state $\mathbf{x}_k$ at time t_k depends implicitly on the initial value $\mathbf{x}_0 \in \mathbb{R}^{n_x}$ and the control values $\mathbf{u} \in \mathbb{R}^{n_u}$ up to time point t_{k-1}:

$$\mathbf{x}_k = \mathbf{X}_k(\mathbf{x}_0, \mathbf{u}_0, \cdots, \mathbf{u}_{k-1}), \quad k = 1, \cdots, n \tag{5.1}$$

The states can be recursively solved using the non-linear functions $\mathbf{X}_k$:

$$\mathbf{x}_1 = \mathbf{x}_0 + h_0 \cdot \mathbf{f}(t_0, \mathbf{x}_0, \mathbf{u}_0) := \mathbf{X}_1(\mathbf{x}_0, \mathbf{u}_0) \tag{5.2}$$

$$\mathbf{x}_2 = \mathbf{x}_1 + h_1 \cdot \mathbf{f}(t_1, \mathbf{X}_1(\mathbf{x}_0, \mathbf{u}_0), \mathbf{u}_0) := \mathbf{X}_2(\mathbf{x}_0, \mathbf{u}_0, \mathbf{u}_1) \tag{5.3}$$

$$\mathbf{x}_N = \mathbf{x}_{N-1} + h_{N-1} \cdot \mathbf{f}(t_{N-1}, \mathbf{X}_{N-1}(\mathbf{x}_0, \mathbf{u}_0, \cdots, \mathbf{u}_{N-2}), \mathbf{u}_{N-1}) \tag{5.4}$$

$$\mathbf{x}_N := \mathbf{X}_N(\mathbf{x}_0, \mathbf{u}_0, \mathbf{u}_1, \cdots, \mathbf{u}_{N-1}) \tag{5.5}$$

The derivatives of $\mathbf{X}$ w.r.t. $\mathbf{x}_0$ and c_i (de Boor of controls using B-splines as described in Section 2.2.2) are:

$$\mathbf{X}'_{k,\mathbf{x}_0} = \frac{\partial \mathbf{X}'_k}{\partial \mathbf{x}_0} \tag{5.6}$$

$$\mathbf{X}'_{k,c_i} = \begin{bmatrix} \frac{\partial \mathbf{X}'_k}{\partial \mathbf{c}_0} & \frac{\partial \mathbf{X}'_k}{\partial \mathbf{c}_1} & \dots & \frac{\partial \mathbf{X}'_k}{\partial \mathbf{c}_N} \end{bmatrix} \tag{5.7}$$

The derivatives of the objective function J with respect to the initial $\mathbf{x}_0 = \mathbf{x}(t_0)$ and final $\mathbf{x}_N = \mathbf{x}(t_N)$ values of the state variables in the OCP are represented by J'_{x_0} and J'_{x_N} respectively, and let $\mathbf{z} = (\mathbf{x}_0, \mathbf{c})$. The Jacobian of the objective function $\mathbf{J}'(\mathbf{z})$ is defined in [Ger12]:

$$\mathbf{J}'(\mathbf{z}) = \begin{bmatrix} J'_{x_0} + J'_{x_N} \cdot \mathbf{X}'_{N,x_0} & [J'_{x_N} \cdot \mathbf{X}'_{N,c_i}] \end{bmatrix} \tag{5.8}$$

The derivatives of the inequality constraints $\mathbf{g}$ with respect to the state variables at time t_i, where $0 \leq i \leq N$, are denoted by $\mathbf{g_x}'[t_i]$. The Jacobian of the inequality constraints $\mathbf{G}'(\mathbf{z})$ is defined in [Ger12]:

$$\mathbf{G}'(\mathbf{z}) = \begin{bmatrix} \mathbf{g_x}'[t_0] & [\mathbf{0}] \\ \mathbf{g_x}'[t_1] \cdot \mathbf{X}'_{1,x_0} & [\mathbf{g_x}'[t_1] \cdot \mathbf{X}'_{1,c_i}] \\ \vdots & \vdots \\ \mathbf{g_x}'[t_N] \cdot \mathbf{X}'_{N,x_0} & [\mathbf{g_x}'[t_N] \cdot \mathbf{X}'_{N,c_i}] \end{bmatrix} \tag{5.9}$$

The derivatives of the equality constraints $\mathbf{h}$ with respect to the initial $\mathbf{x}_0$ and final $\mathbf{x}_N$ values of the state variables in the OCP are represented by $\mathbf{h}'_{x_0}$ and $\mathbf{h}'_{x_N}$ respectively. The Jacobian of the equality constraints $\mathbf{H}'(\mathbf{z})$ is defined in [Ger12]:

$$\mathbf{H}'(\mathbf{z}) = \begin{bmatrix} \mathbf{h}'_{x_0} + \mathbf{h}'_{x_N} \cdot \mathbf{X}'_{N,x_0} & [\mathbf{h}'_{x_N} \cdot \mathbf{X}'_{N,c_i}] \end{bmatrix} \tag{5.10}$$

The sensitivity matrices are $\mathbf{S}_k \in \mathbb{R}^{n_x \cdot (n_x + N \cdot n_u)}$:

$$\mathbf{S}_k := \frac{\partial \mathbf{X}_k(\mathbf{z})}{\partial z} \tag{5.11}$$

The computation of the sensitivity differential equations $\mathbf{S}'(t)$ involves the derivatives of $\mathbf{f}$ with respect to $\mathbf{x}$ and $\mathbf{u}$, which are denoted as $\mathbf{f}'_x$ and $\mathbf{f}'_u$, respectively:

$$\mathbf{S}'(t) = \mathbf{f}'_x(t, \mathbf{X}(t;z), \mathbf{u}(t;z)) \cdot \mathbf{S}(t) + \mathbf{f}'_u(t, \mathbf{X}(t;z), \mathbf{u}(t;z)) \cdot \frac{\partial u}{\partial z}(t,z) \tag{5.12}$$

Integrating Equation (5.12) will result in the sensitivities Equation (5.11). This integration can be coupled with integrating the robot dynamics Equation (3.15) using the numerical integration method of *DOPRI5* as described in Section 4.1.

5.1.2 Finite Differences-based approach

Another approach to calculate the Jacobian matrices is the finite difference (FD) method [Ger12]. FD is a class of numerical methods to solve differential equations. The time is discretized and the value at these discreet points is approximated by finite differences. FD is approximated using the Taylor series expansion. However, this method lacks precision, which can hinder convergence. FD can be very computationally expensive [CEK16]. The Jacobians in Equations (5.8), (5.9) and (5.10) can be also described as:

$$\mathbf{J}' = \begin{bmatrix} \frac{d(\varphi)}{dc_0} & \frac{d(\varphi)}{dc_1} \dots \frac{d(\varphi)}{dc_N} \end{bmatrix} \tag{5.13}$$

$$\mathbf{G}' = \begin{bmatrix} \frac{d(\varphi_0)}{dc_0} & \frac{d(\varphi_1)}{dc_0} & \dots & \frac{d(\varphi_N)}{dc_0} \\ \vdots & \vdots & \vdots & \vdots \\ \frac{d(\varphi_0)}{dc_N} & \frac{d(\varphi_1)}{dc_N} & \dots & \frac{d(\varphi_N)}{dc_N} \end{bmatrix} \tag{5.14}$$

$$\mathbf{H}' = \begin{bmatrix} \frac{d(h_0)}{dc_0} & \frac{d(h_1)}{dc_0} & \dots & \frac{d(h_N)}{dc_0} \\ \vdots & \vdots & \vdots & \vdots \\ \frac{d(h_0)}{dc_N} & \frac{d(h_1)}{dc_N} & \dots & \frac{d(h_N)}{dc_N} \end{bmatrix} \tag{5.15}$$

5.2 Implementation

The sensitivity differential equation (5.12) indicates that each column can be computed in parallel. This can be achieved by utilizing CPU multi-threading and creating threads based on the number of optimization parameters ($\in \mathbb{R}^{(N+k-2)\cdot n}$) and the total number of cores available on the machines used for computation.

Algorithm 5 requires the derivative of the system dynamics with respect to the state $\mathbf{f}'_x(\cdot)$, the derivative of the system dynamics with respect to the control $\mathbf{f}'_u(\cdot)$, control input $\mathbf{u}$, parameter vector $\mathbf{z}$, start and end times t_0 and t_f, and initial sensitivity matrix $\mathbf{S}(t_0)$ as inputs. The finite-difference method is employed to obtain the values of $\mathbf{f}'_x(\cdot)$ and $\mathbf{f}'_u(\cdot)$.

Algorithm 5 Compute Sensitivity Matrices

1: **while** $t < t_f$ **do**
2: Compute: $\frac{\partial u}{\partial z}(t, z)$
3: $\mathbf{S}'(t) \leftarrow \mathbf{f}'_x \cdot \mathbf{S}(t) + \mathbf{f}'_u \cdot \frac{\partial u}{\partial z}(t, z)$
4: Integrate $\mathbf{S}'(t)$ in Algorithm 1 or Algorithm 2
5: $t \leftarrow t + \Delta t$
6: **end while**

The derivative of a function $f(x)$ at x, with step size h, can be computed by Algorithm 6 that utilizes the central finite differences formula. To parallelize the computation of Jacobian matrix elements, OpenMP API [OpenMP] is employed. The function $f(x)$ serves as the constraints in the OCP.

Algorithm 6 Central Finite Differences

1: $f'(x) \leftarrow \frac{f(x+h)-f(x-h)}{2h}$

5.3 Results

Setting the various tolerances of the OCP solver with accuracy is crucial when computing Finite Differences for the OCP. In this thesis, the central finite difference method is implemented to solve the motion planning problem, with 100 random starts for the OCP, with each start needing to reach 100 different desired end-effector pose and velocity configurations, amounting to 10,000 unique instances. The sensitivity-based approach provides an average speedup of approximately 2 times faster than using FD, with a solution accuracy ranging around 10^{-8}. In contrast, the accuracy achieved with FD varied from 10^{-5} to 10^{-7}. Apart from the accuracy, the solutions - generated trajectories for both methods are similar. The success rate for the sensitivity-based approach is approximately 98 percent for the 10,000 queries, while for the finite difference-based method, it is approximately 95 percent.

Although parallelizing the computation of sensitivities resulted in a speedup of 1.2 to 1.5 times, using Finite Differences caused a slowdown of approximately 1.1 to 1.3 times. Calculating FD or sensitivities at each iterations takes a few microseconds. The bottlenecks occur due to the large number of QP solutions in the SQP, which leads to the bottleneck in calculating the sensitivities or FDs at each iteration. However, sensitivities offer superior

execution speed and solution accuracy in both serial and parallel algorithms compared to Finite Differences. Thus, it is recommended to use sensitivity differential equations for computing Jacobian matrices instead of Finite Differences, despite the complexity of implementation.

6 Conclusion

The present study has explored methodologies to accelerate the motion planning for free-floating robots through GPU and CPU parallelization. By examining the point-to-point motion, bottlenecks within the OCP were identified and analyzed. The computation of robot dynamics takes up to 40 percent of the total computation time and calculating the Jacobians takes up to 15 percent. It was found that parallelizing the robot dynamics computation on CPU resulted in a maximum speedup of 15x, while computing it using GPU showed that one iteration is 250x slower than the CPU iteration for the given hardware.

The results obtained from parallelizing the robot dynamics and its integration on GPU and CPU can be applied in various ways. The CPU parallelization may be used with a parallel CPU implementation of the NLP solver and the constraints computation using the MPI framework. Otherwise, if the NLP solver can also be implemented on GPUs, the complete pipeline may be computed using GPGPU. While multiple shooting is an alternative for using GPU parallelization for the different phases within the shooting nodes, it becomes less desirable when hundreds of phases (given a 250x speed-down for one iteration, at least 250 phases would be required to reach a break-even) are required to achieve a significant speedup.

Regarding Jacobian computation, it was found that the sensitivity-based approach is more suitable than the finite difference based approach as it is faster (almost 1.5x) and produces more accurate results. The results are also applicable for more complex robotic systems such as humanoids, where the number of state and control variables are high, and there are more constraints to be satisfied.

Although parallelizing with CPU did not result in significant gains, this study provides a foundation for future research to enhance planning efficiency. The future work can include implementing the findings in both free-floating and humanoid robotic systems, and examining practical applications such as automatic differentiation (AD) for computing the Jacobians. AD has already been quite useful in various other fields, including machine learning, finance, and physical modeling. It can also be parallelized and is already implemented in libraries such as TensorFlow and PyTorch.

The study's results are relevant for developing more efficient motion planning algorithms for robotic systems. The findings contribute to understanding the challenges and opportunities of parallelizing motion planning. This information can be useful in different

domains such as orbital robotics and humanoids. Additionally, the study provides valuable insights into the limitations of current approaches and potential avenues for future research.

Although this study demonstrated parallelization's potential to improve motion planning efficiency, future research needs to address several limitations. For example, the study only focused on classical control methodologies for free-floating robots. Future research could examine how machine learning could apply these findings.

Overall, this master's thesis lays a foundation for further research to enhance motion planning efficiency for free-floating robotic systems. The findings can benefit researchers and practitioners interested in faster computation of motion planning.

Bibliography

[Amd67] Amdahl, G. M.
Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities
In: Proceedings of the April 18-20, Spring Joint Computer Conference (, 1967), pp. 483–485.

[ArrayFire] *ArrayFire*
https://arrayfire.com/ (visited on 04/07/2022).

[Bet98] Betts, J.
Survey of Numerical Methods for Trajectory Optimization
In: Journal of Guidance, Control, and Dynamics, 21 (1998) 2.

[CEK16] Chrétien, B.; Escande, A.; Kheddar, A.
GPU Robot Motion Planning using Semi-Infinite Nonlinear Programming
In: IEEE Transactions on Parallel and Distributed Systems, 27 (2016) 10.

[CUDA Programming Guide] *GPUs vs CPUs*
https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html (visited on 04/07/2022).

[Dij59] Dijkstra, E. W.
A note on two problems in connexion with graphs
In: Numerische mathematik, 1 (1959) 1, pp. 269–271.

[Direct Compute] *DirectCompute*
https://developer.nvidia.com/directcompute (visited on 04/07/2022).

[DP80] Dormand, J.; Prince, P.
A family of embedded Runge-Kutta formulae
In: Journal of Computational and Applied Mathematics, 6 (1980) 1.

[DP93] Dubowsky, S.; Papadopoulos, E.
Kinematics, dynamics, and control of free-flying and free-floating space robotic systems
In: Robotics and Automation, IEEE Transactions on, 9 (1993).

[Ger12] Gerdts, M.
Optimal Control of ODEs and DAEs
De Gruyter, 2012.

[Ger18] Gerdts, M.
User's Guide: OCPID-DAE1 - Optimal Control and Parameter Identification with Differential-Algebraic Equations of Index 1
In: Ingenieur Mathematik (, 2018).

[Gus88] Gustafson, J. L.
Reevaluating Amdahl's Law
In: Commun. ACM, Association for Computing Machinery, 31 (1988) 5, pp. 532–533.

[HNR68] Hart, P.; Nilsson, N.; Raphael, B.
A Formal Basis for the Heuristic Determination of Minimum Cost Paths
In: IEEE Transactions on Systems Science and Cybernetics, 4 (1968) 2, pp. 269–271.

[Intel Website] *Moore's Law*
https://www.intel.com/content/www/us/en/newsroom/opinion/moore-law-now-and-in-the-future.html (visited on 04/07/2022).

[Kel17] Kelly, M.
An Introduction to Trajectory Optimization: How to Do Your Own Direct Collocation
In: SIAM Review, 59 (2017) 4, pp. 849–904.

[Khronos Group OpenCL] *OpenCL*
https://www.khronos.org/opencl/ (visited on 04/07/2022).

[KSL96] Kavraki, L.; Svestka, P.; Latombe, J.-C.; Overmars, M.
Probabilistic Roadmaps for Path Planning in High-Dimensional Configuration Spaces

In: Robotics and Automation, IEEE Transactions, 12 (1996) 4, pp. 566–580.

[KWP11] Karaman, S.; Walter, M. R.; Perez, A.; Frazzoli, E.; Teller, S.
Anytime Motion Planning using the RRT-star
In: International Conference on Robotics and Automation (, 2011).

[Lam10] Lampariello, R.
Motion Planning for the On-orbit Grasping of a Non-cooperative Target Satellite with Collision Avoidance
In: International symposium on Artificial Intelligence, Robotics and Automation in Space, 10 (2010).

[LaV06] LaValle, S. M.
Planning Algorithms
Cambridge: Cambridge University Press, 2006.

[LaV98] LaValle, S. M. et al.
Rapidly-exploring random trees: A new tool for path planning
In: Technical Report. Computer Science Department, Iowa State University (, 1998).

[LMO18] Lampariello, R.; Mishra, H.; Oumer, N.; Schmidt, P.; De Stefano, M.; Albu-Schäffer, A.
Tracking Control for the Grasping of a Tumbling Satellite With a Free-Floating Robot
In: IEEE Robotics and Automation Letters, 3 (2018) 2.

[LVY13] Lengagne, S.; Vaillant, J.; Yoshida, E.; Kheddar, A.
Generation of whole-body optimal dynamic multi-contact motions
In: The International Journal of Robotics Research (, 2013).

[Moo65] Moore, G.
Cramming more components onto integrated circuits
In: IEEE Solid-State Circuits Society Newsletter, 11 (1965) 3.

[NP17] Nanos, K.; Papadopoulos, E. G.
On the Dynamics and Control of Free-floating Space Manipulator Systems in the Presence of Angular Momentum
In: Frontiers in Robotics and AI, 4 (2017).

[ODE45] *ODE45*
https://de.mathworks.com/help/matlab/ref/ode45.html (visited on 04/07/2022).

[Odeint] *Odeint*
https://www.boost.org/doc/libs/1_81_0/libs/numeric/odeint/doc/html/index.html (visited on 04/07/2022).

[OpenMP] *OpenMP*
https://www.openmp.org/ (visited on 04/07/2022).

[PAM21] Papadopoulos, E.; Aghili, F.; Ma, O.; Lampariello, R.
Robotic Manipulation and Capture in Space: A Survey
In: Frontiers in Robotics and AI, 8 (2021).

[Pap90] Papadopoulos, E.
On the dynamics and control of space manipulators
In: Dept. Mech. Eng., MIT, Cambridge, MA, Ph.D dissertation (, 1990).

[PNB22] Plancher, B.; Neuman, S. M.; Bourgeat, T.; Kuindersma, S.; Devadas, S.; Reddi, V. J.
GRiD: GPU-Accelerated Rigid Body Dynamics with Analytical Gradients
In: International Conference on Robotics and Automation (ICRA) (, 2022).

[RM website] *Space Robots at RM DLR*
https://www.dlr.de/rm/desktopdefault.aspx/tabid-12513/ (visited on 04/06/2022).

[Thrust] *Thrust*
https://docs.nvidia.com/cuda/thrust/ (visited on 04/07/2022).

[YT93] Y., X.; T., K.
Space Robotics: Dynamics and Control
In: Kluwer Academic Publishers (, 1993).

[YWH16] Yoshida, K.; Wilcox, B.; Hirzinger, G.; R., L.
Space Robotics - Springer Handbook of Robotics
Springer International Publishing, 2016.

[YWP16] Yang, Y.; Wu, Y.; Pan, J.
Novel GPU-based Parallel Implementation Scheme and Performance Analysis of Robot Forward Dynamics Algorithms
In: Computing Research Repository (, 2016).

[YWP18] Yang, Y.; Wu, Y.; Pan, J.
Unified GPU-Parallelizable Robot Forward Dynamics Computation Using Band Sparsity
In: IEEE Robotics and Automation Letters (, 2018).

List of Tables

List of Figures