Delft University of Technology
Master's Thesis in Embedded Systems

Advanced Path Planning for a Neurosurgical Flexible Catheter

Improving the performance of sampling-based motion planning

Kemal Falatehan

embedded software

TU Delft Delft University of Technology

Advanced Path Planning for a Neurosurgical Flexible Catheter

Improving the performance of sampling-based motion planning

Master's Thesis in Embedded Systems

Embedded Software Section
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4, 2628 CD Delft, The Netherlands

Kemal Falatehan
kemal.falatehan@student.tudelft.nl

31st August 2012

Author
Kemal Falatehan (kemal.falatehan@student.tudelft.nl)
Title
Advanced Path Planning for a Neurosurgical Flexible Catheter
Subtitle
Improving the performance of sampling-based motion planning
MSc presentation
15 August 2012

Graduation Committee
Chair: Prof. dr. K.G.Langendoen, Faculty EEMCS, TU Delft
Dr. S.O. Dulman, Faculty EEMCS, TU Delft
Dr. F. Rodriguez y Baena, Department of Mechanical Engineering, Imperial College London

Abstract

Minimally invasive surgery and interventional techniques have successfully broadened the field of surgical applications over the last few decades. In comparison to the old-fashioned surgery techniques, minimally invasive surgery offers benefits to patients such as less invasive treatement, less pain to patients, better for aesthetic reasons and faster recovery time.

Minimally invasive surgery or sometimes is also called *keyhole surgery* or *laparoscopic surgery* is performed by the help of a small endoscopic camera and several long, thin, flexible instruments and through a small incision, these surgical tools are inserted into area of a body that requires treatment. However with the advance of this surgery, it also demands innovative procedures and instruments that could get the best out of this technique.

At Mechatronics in Medicine (MiM) Laboratory of Imperial College London, a neurosurgical steerable flexible probe (STING) that is used to access deep brain lesions through curved trajectories is currently being developed. The focus of the work is mainly on probe development, low- and high-level control and trajectory planning.

This thesis is related more to the work on trajectory planning. Some methods based on Rapidly-exploring Random Tree (RRT) algorithm have been developed to find a curvilinear path that can safely reach the target while maneuvering obstacles. However, due to the tendency of soft tissue to displace and deform, building an online control mechanism for the flexible probe is considered to be necessary to efficiently replan the calculated solution. Online control also allows physicians to interactively edit the planning environment in real-time by adding or removing obstacle definitions.

To move further toward online path exploration where performance has been the main issue of the previous implementations, the main focus of this thesis will be to investigate some ways to increase efficiency and performance of the trajectory planning. Some experiments have been thoroughly done to measure the performance of a well known sampling based path planning method, Reachability-Guided Rapidly-exploring Random Tree (RG-RRT).

The first step to improve the performance was to migrate from MATLAB to Python-C++ which yielded 12-13 times performance speedup. Besides taking a close look at the software implementation details, improving the

algorithm by implementing a *waypoint cache* [30] and exploiting some parallelization techniques have also been considered in this work. The parallelization techniques cover multi-core CPU (OR parallel, AND parallel, OR+AND parallel and Manager-Worker) and GPGPU techiques.

After the software implementation migration, RG-RRT with waypoint cache was experimentally able to reach 4 times performance speedup, while parallelization on multi-core CPU with AND parallel technique has shown the most significant result by obtaining approximately 5 times performance speedup. The other parallelization, which was done through the use of an NVIDIA CUDA-enabled GPU, has successfully obtained 10 times performance speedup. Despite its higher rate of performance speedup, later it was shown in Table 5.1 that GPGPU technique suffers the most from inefficiency due to I/O bottleneck that is caused by *device-host* memory transfer.

Imagination is more powerful even than knowledge Albert Einstein

Preface

This Master thesis is essentially the description of work that I have been doing during the past 6 months at the Mechatronics in Medicine Laboratory, Imperial College London. Coming from a Computer Science background, I consider being part of a multidisciplinary team a challenge. Besides, I have a very big interest in the area of robotic surgery applications since my university time.

Besides many other important topics in the area of intelligent robotics surgery domain, studies on path planning algorithms have drawn considerable attetion of academics and researchers. This trend is to some extend influenced by the advances of a surgery procedure that has become more common in the field of surgical applications, minimally invasive surgery. This Master thesis is more about the analysis of a well known sample based path planning technique, RG-RRT and how it will be suited in multi-core system environment. Experiments have been done in this area and followed by performance analysis for each delivered technique.

During the work of this project, a number of people have given a great deal of support. To start with, I would like to express my deepest gratitude to Dr. Ferdinando Rodriquez y Baena who has given me a priceless opportunity to become part of MiM lab members and also for his support and guidance throughout this project. My second gratitude goes to my parents, for their patience and support during the work of this project. I would also like to say many thanks to Nisa, who has given me support, spirits and priceless moments during the whole process. Last but not least, to all MiM lab members, Fangde, Stuart, Ryo and Matthew for providing their excellent suggestions and ideas on some difficult topics and some issues on the fields that I am not really familiar with, Robotics and Mechanical Engineering.

Kemal Falatehan

Delft, The Netherlands
31st August 2012

Contents

List of Figures

List of Tables

Acronyms

API Application Programming Interface

CUDA Compute Unified Device Architecture

CPU Central Processing Unit

ERRT Extended Rapidly-exploring Random Tree

GPU Graphics Processing Unit

GPGPU General-Purpose computation on Graphics Processing Units

GUI Graphical User Interface

ITK Insight Segmentation and Registration Toolkit

NUMA Non-Uniform Memory Access

MIM Mechatronics in Medicine

MIMD Multiple Instruction, Multiple Data

MIS Minimally Invasive Surgery

MISD Multiple Instruction, Single Data

OpenMP Open Multiprocessing

PRM Probabilistic Roadmap

RGRRT Reachability-Guided Rapidly-exploring Random Tree

RRM Reachability Roadmap

RRT Rapidly-exploring Random Tree

SIMD Single Instruction, Multiple Data

SISD Single Instruction, Single Data

SMA Shared Memory Architecture

SMP Symmetric Multi-Processor

STING Soft Tissue Intervention and Neurosurgical Guide

UMA Uniform Memory Access

VTK Visualization Toolkit

Chapter 1

Introduction

1.1 A Gentle Introduction to Minimally Invasive Surgery (MIS)

Minimally invasive surgery has been considered a major revolutionary breakthrough in robotic surgery. As the name might already suggest, minimally invasive surgery is less invasive than the open surgery and thus minimizing trauma to surrounding tissues, minimize scarring (up to 30%), shorten the hospital stays (up to 60% than the conventional surgery), much less pain and discomfort, less medication and eventually less post-surgical complications. The procedure is done by inserting a long, thin, flexible tube (trocar) with an attached camera that has a light source (usually endoscope (Fig. 1.1(a)) or laparoscope (Fig. 1.1(b))) at the end through small incisions (around 3/4 inch) in the skin. Images captured from the camera will then be projected onto monitors in the operating room where the surgeon will get an illuminated and magnified image (up to 100 times magnification) of an area of the body that needs treatment (pathology) [1]. When the surgeon needs to perform further surgical operations such as exploring, removing tissues, other instruments can then be inserted to the other openings. Being able to have a very detailed visualization will lead to much higher success rate and faster surgery time than the conventional surgery.

With all the advantages and the benefits of minimally invasive surgeries, not all surgeries can be done invasively. Sometimes it can even happen that after performing a thorough examination to the patient, the surgeon eventually decides not to perform minimally invasive surgery. This could happen due to the nature of the wounds such as the removal of cancer tumors that is more suited to the conventional surgery than minimally invasive surgery, some anatomy problems that arise just before the surgery takes place, for patients who have had previous open surgery in the upper or lower abdomen

parts and for patients who suffer from obesity.

Besides all the benefits of the minimally invasive surgery, there are also some difficulties in performing the surgery. They are [2]:

- restricted vision
- difficult handling of the instruments
- very restricted mobility
- difficult hand-eye coordination
- no tactile perception
- requires specialists and special training
- requires special instruments
- special instruments can be very expensive
- some procedures may take longer surgery time

There are two common procedures in minimally invasive surgery:

- Laparoscopy, medical procedures within the abdominal or pelvic (lower belly) cavities
- Hysteroscopy, medical procedures through vagina and cervix to examine uterus (womb)

The applications of the surgery might differ from one type of procedure to the others. A procedure like minimally invasive hemorrhoid procedure, can be performed with no incisions at all such as Hysteroscopy as the trocar can be inserted through the natural opening in the cervix (neck of the womb) from the vagina into the uterus and some procedures like minimally invasive colon surgery requires a slightly larger incision. With the advance of technology, there is even a procedure that requires only single entry incision point which is called Single Site Laparoscopy. Nowadays, minimally invasive can be performed in many human body parts [3] such as:

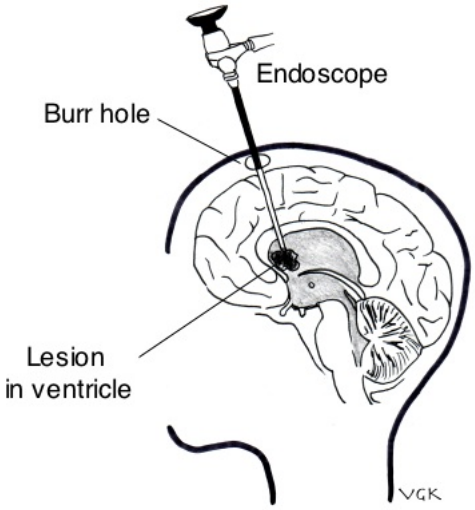
- Cardiac
- Gastrointestinal
- Gynaecological
- Neurological
- Orthopaedic

- Otorhinolaryngology
- Respiratory/Thoracic
- Urology
- Vascular

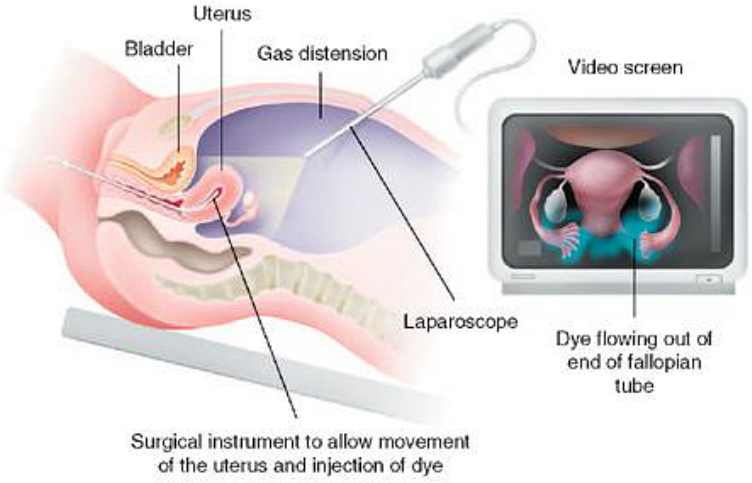
and surgical procedures that can be performed with minimally invasive surgery include:

- Breast biopsy
- Colon resection
- Cardiovascular surgery
- Fundoplication
- Gastric bypass
- Gynecological diagnosis and treatment
- Hemorrhoids
- Hernia repair
- Hysterectomy
- Nephrectomy
- Orthopedics

Although minimally invasive surgery has just been widely accepted as one of surgical procedures, it is not a new technique. It is in fact has been applied for more than 30 years. This slow progress is due to the nature of the procedure that is tightly related to the technology development. It is becoming more and more common and advanced with the development of ultrasmall video cameras and harmonic scalpels, which separate tissue by vibrating at 50.000 cycles per seconds [2]. Even until now, many researchers are still working to find new, innovative instruments that guarantee medical safety and efficiency.



(a) Endoscopic brain surgery or Neuroendoscopy



A laparoscopy allows careful inspection of the ovaries, uterus and fallopian tubes through a narrow 'telescope' with an attached video camera inserted into the abdomen.

(b) Laparoscopy

Figure 1.1: Minimally Invasive Surgeries

1.2 Development on Steerable Percutaneous Robotics

When minimally invasive surgery came to the surface for the first time, many surgeons tried to apply the technique to many surgical procedures. For many routine procedures, minimally invasive surgery is very effective, both for the surgeons and the patients. However to perform more delicate or complex medical procedures, precision became an inevitable issue as handling of the instruments turned out to be difficult and quite tedious. Comparing with the old conventional open surgery, minimally invasive surgery suffers more from this problem due to the small incisions. Robotically-assisted surgery was then invented to help surgeons in controlling the instruments. Instead of manually moving the instruments, robotically-assisted surgery overcomes the restricted mobility problems by performing the normal surgery movements whilst the robotics arms in fact carry out the movement using end-effectors and manipulators to do the actual surgery on the patient. Equipped with 3D monitors, the surgeon can perform the surgery conveniently, while still maintaining the precision, dexterity and control. To provide very detailed visualizations, due to the high sensitivity, high spatial resolution, excellent soft tissue contrast, and multi-planar volumetric imaging capabilities, MRI has been used together with 3D image viewer [4]. Besides MRI, percutaneous image-guided robotic system that employs ultrasound, CT guidance and X-ray have also been developed to provide smaller, simpler, more accurate and more cost-effective robotic systems.

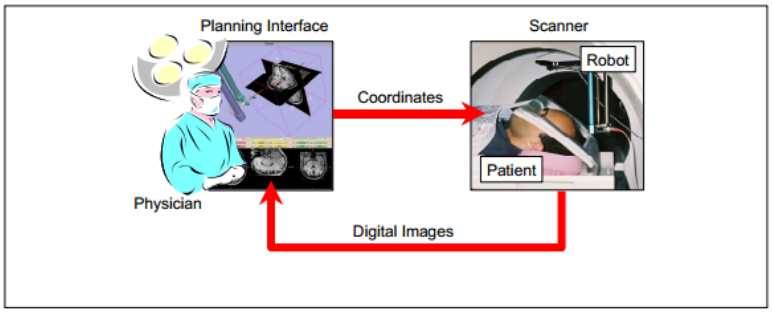


Figure 1.2: Image-guided intervention system with integrated planning interface and placement device

Inserting a straight, thick and rigid needle with a symmetric tip under visual feedback was later proven to yield inconsistent results and requires specialists or a long training. Moreover, some factors such as significant pressure on the tissue, errors during the insertion, tissue deformation and needle bending can degrade the accuracy significantly. When the target accuracy is very poor, it could harm the healthy and sensitive tissue or even cause haemorrhage

(bleeding) which can then lead to severe infections thus lengthening the surgery and recovery time. The bad news is, by having a very long and intensive training, clinicians will still not be able to guarantee the accuracy as they do not posses the control over the needle once it is inserted into the tissue. Due to this poor accuracy, medical dosage becomes inefficiently given to the target and consequently it could lead to negative side effects to the sorrounding healthy tissues. For these reasons, needle steering method using different needle tip shapes and sizes was developed to provide accuracy. Once the needle is inserted into the skin, the needle is steered towards the lesions through a combination of lateral and twisting motions.

To date, there are several needle steering methods that have been developed. The first one is by using a flexible asymmetric bevel tip needle. This technique relies on the significant correlations between the needle material elasticity, bevel tip angle and the linear and nonlinear tissue elasticity. Under these assumptions, needle trajectory path could then be envisaged [5]-[8]. The second method is to model the needle steering via Duty-Cycled Spinning [9]. This method includes the rotational speed and the spinning behaviour of the needle so that it can follow the curvilinear trajectory path through the tissue. Another method is based on a concentric combination of precurved elastic tubes. This method controls tip location and orientation by rotating and extending the tubes with respect to one another [10]. The last method is by using a flexible and steerable four-part probe, called STING, that is currently being developed at the Mechatronics in Medicine Laboratory, Imperial College London [11]. The background and the ideas behind STING will be covered in Chapter 2.

However, it was later discovered that due to errors during insertion, complex biomechanical interactions between the needle and the tissue and tissue deformation, manipulating the needle manually by twisting it turned out to be not a trivial thing to do. Robot-assisted needle steering was then invented to cope with those problems. This implies that the needle will be automatically guided to the target as it is inserted into the human tissue. As it later observed, avoiding obstacles turned out to be hassle as sorrounding tissues displace and deform when the needle is inserted into or retracted out of the tissue. Facing this fact, the needle has to be steered away from the sensitive parts or healthy tissues and planning algorithms that can take the needle to the intended target safely are required.

1.3 Path Planning for steerable needles

To reach the target safely with a better accuracy in a deformable tissue, many path planning algorithms have been developed. A path planning al-

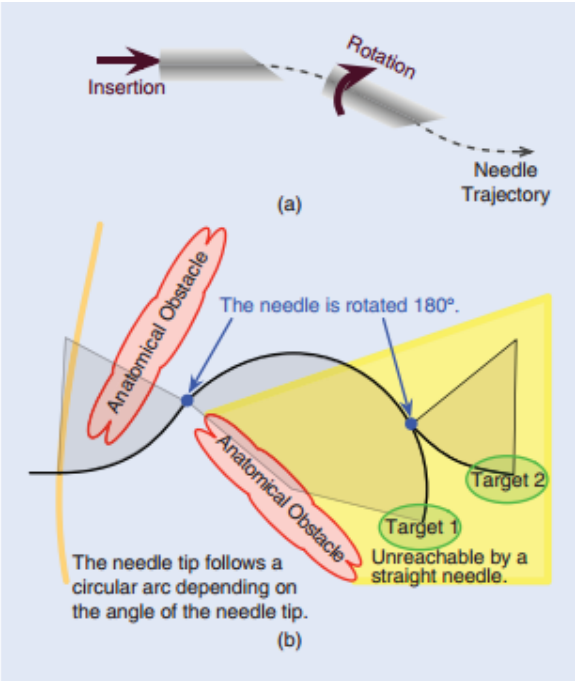


Figure 1.3: **(a)**The steering mechanism of asymmetric-tip needles [12] is due to forces between the tissue and needle tip that deflect the needle during insertion. Subsequent rotation of the needle shaft(from outsise the patient) reorients the tip so that further insertion deflects the tip in a new direction. The combination of these two control inputs, along with a flexible needle shaft, allows the needle to be steered inside the tissue. **(b)** A steerable needle can reach subsurface targets not accessible using conventional needles, and multiple targets can be reached without fully retracting the needle. Anatomical obstacles could include bones, vessels, nerves, and other structures that a needle might damage or not be able to penetrate [13]

gorithm requires both the initial needle and the target locations as well as the needle orientation as its inputs. It then calculates and creates a feasible and safe path (i.e. path that does not cross any sensitive and healthy tissues) in the free configuration space from the insertion point to the lesion.

1.4 Motivation and Objectives

Developing path planning algorithms in deformable and soft tissue have been so far quite challenging due to its uncertainty nature. Many methods have emerged such as tip-based steering, base manipulation and tissue manipulation to name a few. All of them have the same purpose, how to steer an asymmetric-tip needle safely from a given initial configuration to a given

goal configuration while minimizing the tissue damage. The work on this project is based on a well known sampling based path planning algorithm, RG-RRT (Reachability-Guided Rapidly-Exploring Random Tree). There has been some work on replanning with RRTs. However, to replan a path when infrequent changes occur in the configuration space could take some time thus becoming inefficient.

At the *Mechatronics in Medicine Laboratory* of Imperial College London, a flexible bevel-tip needle for neurosurgery inspired by nature, called STING (Soft Tissue Intervention and Neurosurgical Guide) has been developed over the past few years. This flexible steerable probe is modelled as a non holonomic robot and is able to follow a curvilinear path in 2D planar cross-section of the human brain. This probe imposes mechanical and physcial constraints such as minimum radius of the curvature that can be achieved by the maximum steering offset of the probe, thickness or outer diameter of the probe and smooth as well as continuous shape of the curvature. The focus of this project is to improve the efficiency and the performance of RG-RRT for the flexible probe by switching to a compiled language, exploring another possibility on the algorithmic side and exploiting the power of parallelization on RG-RRT. Five different parallelization methods that takes advantage of a multi-core CPU and an NVidia CUDA-enabled GPU will be explored to facilate the speedup RG-RRT in dynamic environment.

1.5 Thesis Outline

The organization of this thesis can be described as follows:

- Chapter 2 presents the development on neurosurgical flexible probe, the ideas behind STING and explores different types of path planning algorithms in depth
- Chapter 3 presents the discussion on a sampling based path planning technique, RG-RRT and its variants. The discussion encompasses the issues, architectural design, implementation and system integration from software engineering perspective
- Chapter 4 presents the exploration on RG-RRT parallelization. Five methods will be presented and discussed in depth
- Chapter 5 presents the analysis and the discussion on the overall parallelization performance (a thorogh analysis will be given between two types of parallelization, CPU and GPGPU)
- Chapter 6 concludes the thesis and defines some possible improvements for future development

Chapter 2

Background and Literature Review

2.1 STING: Soft-Tissue Intervention and Neurosurgical Guide

Surgical interventions performed by minimally invasive surgery require high precision, accuracy and safety. Achieving these requirements cannot be done very easily with any conventional rigid instruments inside brain especially with smaller nerves and blood vessels compared to the other body parts. To address the problem of maneuverability, a biologically inspired soft-tissue intervention and neurosurgical guide or ***STING*** has been developed to access deep brain lesions through curved trajectories.

2.1.1 Bio-inspired Programmable Bevel Concept

Nature has inspired the development of a flexible steerable multi-segments probe, which allows the control of its approach angle by adjusting the steering offset between probe segment [7]. This mechanism mimics the 10 cm long ovipositor, or egg-laying organ of *Megarhyssa ichneumon*, a parasitic wasp that is commonly found in most of Europe, in the Australasian ecozone, in the Near East, in the Nearctic ecozone, in North Africa and in the Oriental ecozone. Instead of rotating the organ, the ovipositor reciprocally moves whilst injecting its eggs on larvae living in timber. Injecting eggs into solid element with this long and hollow organ is possible even without requiring to have instrinsic muscles due to the jagged surface on its tip and the reciprocal insertion mechanism. By doing this reciprocal movement, it will minimize its axial push force thus minimizing the risk of buckling consequently [14][15].
There are 4 segments in the probe, one is for medical procedures and the other is an electro-magnetic tracking sensor that is used to track and mon-

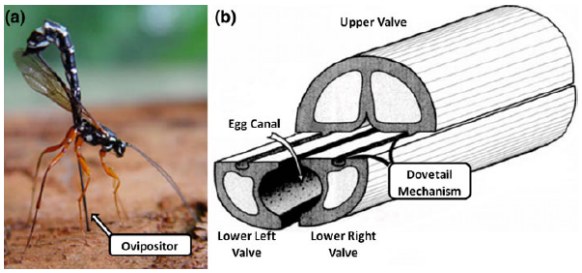


Figure 2.1: A giant wasp *Rhyssa persuasoria* and cross-section of its ovipositor [7]

itor probe location and orientation whilst being pushed through the tissue (Fig. 2.1). During the insertion, these segments will slide with respect to each other independently. The steering offset due to the slide, will reorient the angle of the tip thus controlling the radius of curvature. As a result, the probe can be easilly controlled to follow the continuous trajectory path [16].

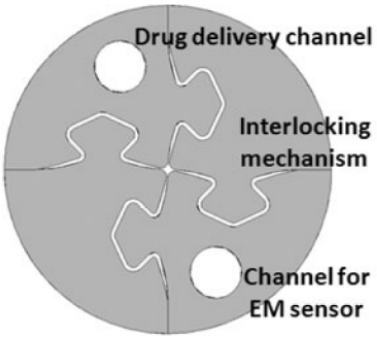


Figure 2.2: Different segments in the probe

2.1.2 Nonholonomic Modeling of Needle Steering

The kinematic model of needle steering can be described as a conventional unicycle model with a fixed steering angle [17], whereas in STING the radius of curvature is modeled as a function of steering offset between the interlocking segments. In this modeling perspective, STING kinematic model is comparable to that of a nonholonomic robot model.

2.1.3 Motion control of a Steerable Needle

The feedback control system of the flexible probe starts from the pre-operative diagnostic images (MRI or CT-based images). This input image together with the probe physical constraint (the minimum radius of curvature) will be forwarded to the 'high level controller'. After finding a safe and feasible

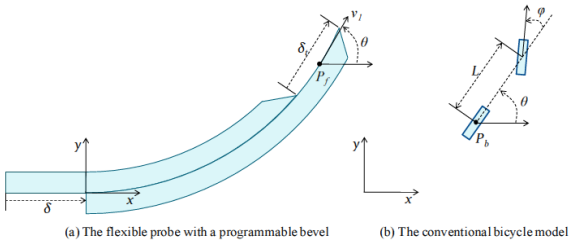


Figure 2.3: Kinematic model comparison between conventional bicycle and STING [17]

path from the initial configuration to the target configuration, the 'low level controller' will initiate motion signals to propel the flexible probe following the curvilinear trajectory. While pushing through the tissue, the electromagnetic tracking sensor located at the tip of the probe will send feedback control to the 'low level controller' to provide path following (Fig. 2.4).

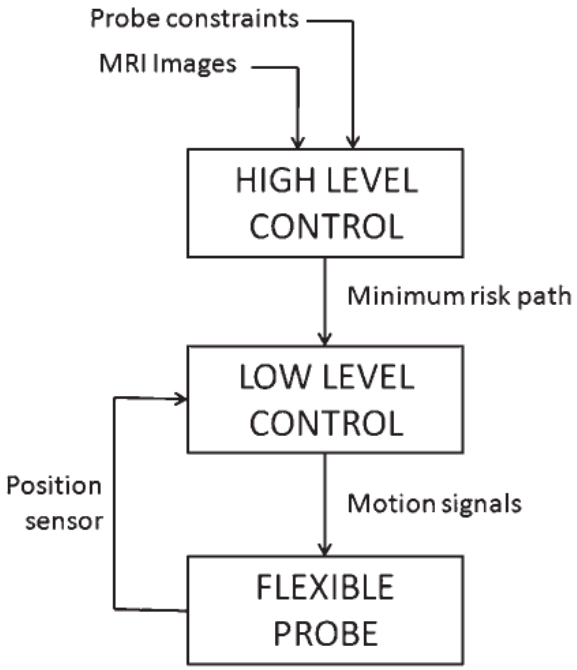


Figure 2.4: Probe System Architecture [11]

2.1.4 Risk-based Trajectory Planning

Although path planning algorithms are capable in finding a safe and feasible path from the initial configuration to the target configuration, to obtain an optimal path whilst minimizing the risk to the patient, risk-based planning

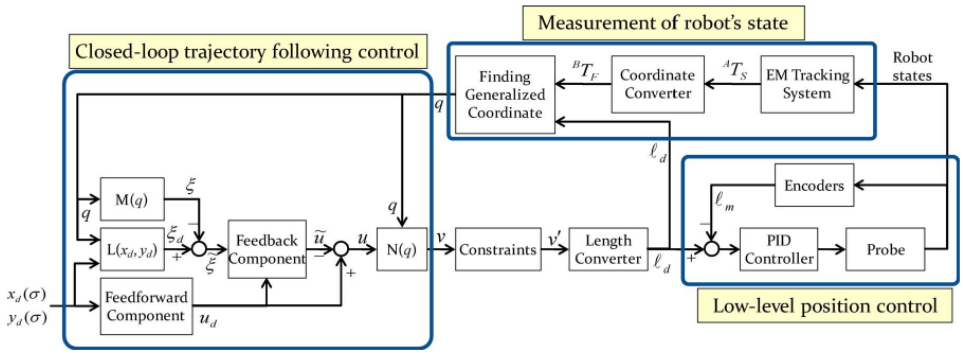


Figure 2.5:
Block diagram of the probe steering control Block diagram of the probe steering control strategy developed for the flexible probe. It includes a closed-loop trajectory controller, a low-level position controller for the robot actuators, and measuring blocks for the robot's state [18]

was introduced. By segmenting parts of the brain based on their risk values, a compromise path between its length and its risk can be obtained. This indicates that parts of the brain are not marked only by 'go' and 'no go' regions but are also classified according to its risk classification (Fig. 2.6).

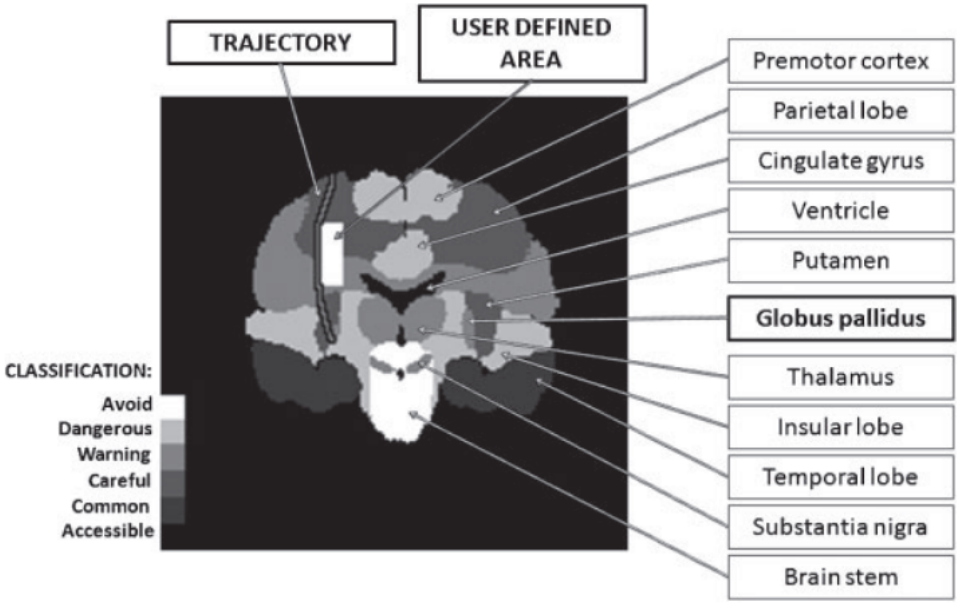


Figure 2.6: Brain Risk Classification

2.2 Path Planning Algorithms

Motion/path planning algorithms objective is to find a feasible path/motion from the initial configuration q_I [x_I, y_I, θ_I] to the goal configuration q_G [x_G, y_G, θ_G] (or one of its goal configurations) w.r.t its contraints. There two types of constaints, environmental constraints (e.g. obstacles) and mechanical/kinematics constraints of the robot (e.g. minimum radius of curvature, the smoothness and the continuity of its curvature).

In motion planning, vector space or the configuration of a system that is defined by a set of coordinates is called the *configuration space*. In his book [19], LaValle has defined the *configuration space* or often shortened as C_{space} as *"a set of possible transformations that could be applied to the robot"*. C_{space} consists of C_{obs} and C_{free} thus $C_{space} = C_{obs} \cup C_{free}$. In motion planning, C_{obs} is the area that will definitely block and prevent from reaching the target, therefore should be avoided. Whereas C_{free} is the area where a feasible path should be found starting from the q_I to the q_G. Fig. 2.7 depicts the basic motion planning that progress from q_I to q_G in C_{free} whilst manuevering obstacles (blocks of C_{obs}) at the same time.

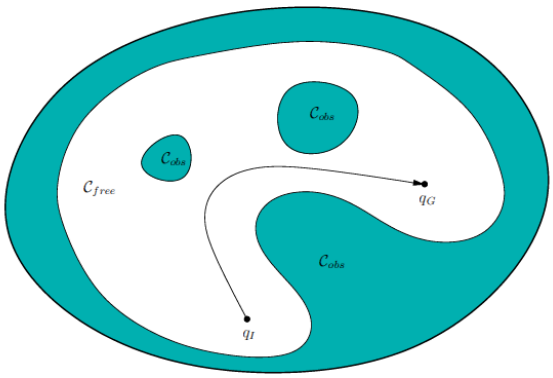


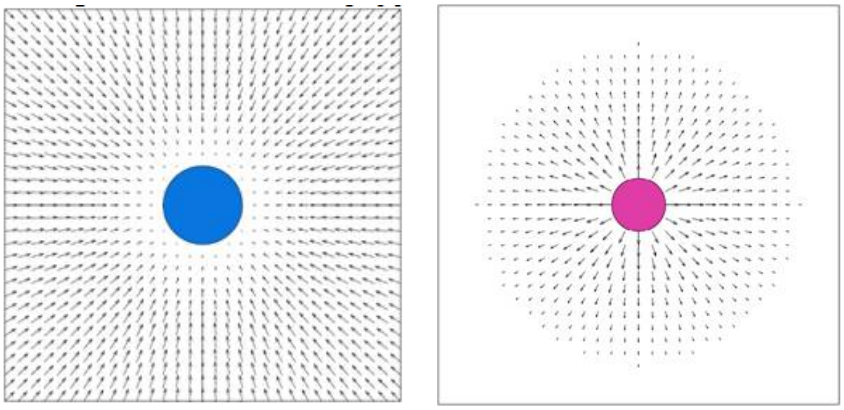
Figure 2.7: Basic Motion Planning [19]

2.2.1 Deterministic Path Planning Methods

Deterministic path planning falls under four different categories that will later be discussed. As the name implies, deterministic path planning behaves predictably given a particular set of inputs, initial and goal configurations. Some experiments on this type of algorithm such as Fast Marching and Gradient based planning algorithms have previously been performed [47] [48]. However, they gave unsatisfactory results as they do not guarantee to reach the goal position with curved edges and fail to bound the path's curvature.

2.2.1.1 Potential Field Method

Potential Field method was initially developed for online collission detection. Similar to machine learning, using this method the robot learns how to act given an observation of the environment. The robot itself is treated as a particle whilst the environment is represented as a potential field. This method is quite intuitive as it behaves exactly as nature. A feasible path can be found by creating an attractive field into the goal (Fig. 2.8(a)) and a repulsive field out of the obstacles (Fig. 2.8(b)) across the entire *configuration space*. To lead the robot to the goal while maneuvering obstacles at the same time, induced combined force by two types of fields (attractive and repulsive fields) that works on the robot has to be computed [20]. This method is very efficient as the motion of the robot is only calculated at its location at a moment in time and also easy to extend as the nature of potential field is additive. It has however some main drawbacks such as local minima trap situations (cyclic behaviour), no passage between closely spaced obstacles, oscillations in the presence of obstacles and oscillations in narrow passages [22].



(a) Attractive potential field [20] (b) Repulsive potential field [20]

Figure 2.8: Potential Fields

2.2.1.2 Roadmap Method

Roadmap method is basically a path planning algorithm that works based on graph model. A feasible path is found by connecting one vertex to another from the initial vertex to the goal vertex without having to pass through the obstacles. If there are more than one feasible paths can be found, shortest path algorithm can then be utilized to find the most optimal path. Some various types of roadmaps are the visibility graph, the Voronoi diagrams (retractions), the freeway net, and the silhouette (critical points).

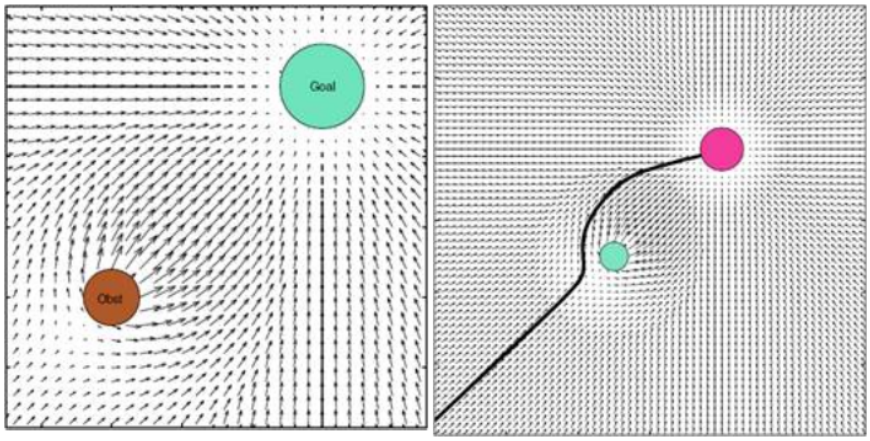


Figure 2.9: Combined Potential Fields [21]

2.2.1.3 Cell Decompositions

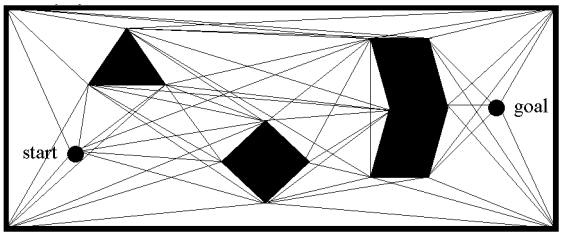
As the name suggests, Cell Decompositions method works by decomposing the *free space* from the intial configuration to the goal configuration into blocks of cells. A connectivity tree can further be determined based on the adjacency relationships between the cells, where the nodes represent the cells in the free space and the links between the nodes show that the corresponding cells are adjacent to each other. Finding a feasible path is then found by finding the shortest path or sequence of cells that connects the initial and the goal points [23]. There are two types of Cell Decompositions that are mostly used in basic motion planning, an exact cell decomposition and an approximate cell decomposition method.

2.2.1.4 Grid-based method

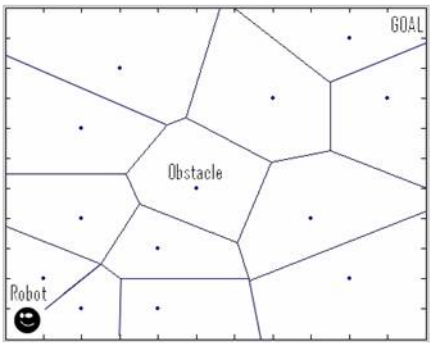
In two-dimensional planar model, the grid-based path planning method works by discretizing the *configuration space* into grid cells that are either blocked (C_{obs}) or unblocked (C_{free}). A feasible path is then calculated by finding the shortest distance in the unblocked area from the initial starting point to the goal point.

2.2.2 Sampling-based Path Planning Methods

Sampling-based path planning is one of two main ways for addressing the motion planning problem by conducting a search by means of sampling points in the *configuration space*. Searching for a feasible path through sampling scheme is done by means of a collision detection module. In his book, LaValle [19] describes this collision detection module as a "black box" from the perspective of the motion planning algorithm. This implies



(a) Visibility graph [23]



(b) Voronoi diagram [24]

Figure 2.10: Roadmaps

that the collision detection module and the geometric models are loosely coupled(Fig. 2.11). The sampling-based path planning is considered to be *probabilistically complete*. This notion of *completeness* indicates that the probability of finding an existing solution converges to one as the number of search iterations approaches infinity. Therefore as long as a solution is still unresolvable, the algorithm will keep running until at one moment in time when it reaches the maximum iteration limit indicating that a solution cannot be found or when a feasible solution is eventually found.

Among many different types of sampling-based methods, one of the most widely-used motion planning algorithm was introduced by Kavraki et al. [51], under the name Probabilistic RoadMaps (PRMs). Not long after the emergence of PRMs, incremental sampling-based algorithms, such as the Rapidly-exploring Random Tree (RRT) algorithm [33] have also emerged.

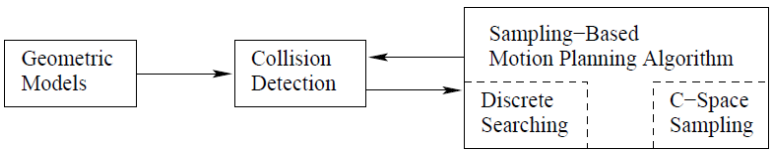


Figure 2.11: The sampling-based motion planning mechanism

2.2.2.1 PRM

The *Probabilistic Roadmap Planner* is one of sample-based motion planning methods that is designed to be a multi-query planner. The algorithm is suitable for a robot that has high degrees of freedom (dof) and consists of two phases, the *learning phase* and the *query phase* [25].

As it constructs a graph (*roadmap*) at this phase, the *learning phase* is often also called the *construction phase*. Each time a random configuration of the robot is sampled, it will further be fed to the collision detection module to check whether it is located in C_{obs} or in C_{free}. This procedure will run iteratively until all random configurations are correctly sampled in C_{free}. After this procedure is done, each of the configurations will be connected to k nearest neighbors or all neighbors with predetermined maximum distance using a fast local planner. To guarantee the validity of these edges, the collision detection module is once again called. For each of the valid edges, it will be added into the *roadmap* for later use.

In the second phase, the *query phase*, the start and goal configurations are added into the *roadmap* and a sequence of configurations connecting the start and goal configurations will be sought by the use of shortest path algorithm (Fig. 2.12).

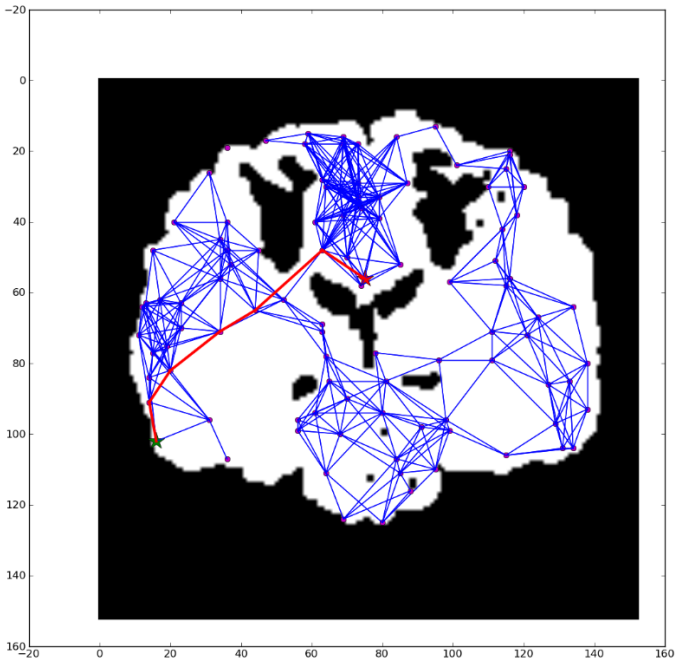


Figure 2.12: A simple Illustration of *Probabilistic Roadmap Planner* with A* shortest path query

2.2.2.2 RRT

A *Rapidly Exploring Random Tree* is a data structure and algorithm that is built incrementally and is suitable for nonconvex, high-dimensional search spaces (Fig. 2.13). On his website [26], Kuffner explains that this algorithm is particulary suited for path planning problems that involve obstacles and differential constraints (nonholonomic or kinodynamic). The algorithm can then be described in pseudocode as follows:

```
Algorithm BuildRRT
Input: Initial configuration q_init, number of vertices in RRT K,
incremental distance Δq
Output: RRT graph G
1. G.init(q_init)
2. for k = 1 to K
3.   q_rand ← RAND_CONF()
4.   q_near ← NEAREST_VERTEX(q_rand, G)
5.   q_new ← NEW_CONF(q_near, Δq)
6.   G.add_vertex(q_new)
7.   G.add_edge(q_near, q_new)
8. return G
```

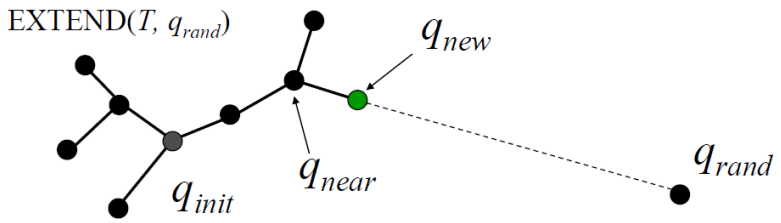


Figure 2.13: Motion Planning with RRTs [27]

In each iteration, a new random configuration (q_{rand}) is sampled by calling RAND_CONF(). Then in line 4 the algorithm tries to find the nearest neighbour (q_{near}) from that newly sampled configuration in the *configuration space*. In the subsequent line, once a q_{near} has been found, a new configuration (q_{new}) will then be calculated by moving incrementally with a constant distance, Δq from q_{near} toward the direction of q_{rand}. Finally, if the edge connecting q_{near} to q_{new} is valid, the new configuration and the new edge will then be added into the tree.

2.2.2.3 RRT-Connect

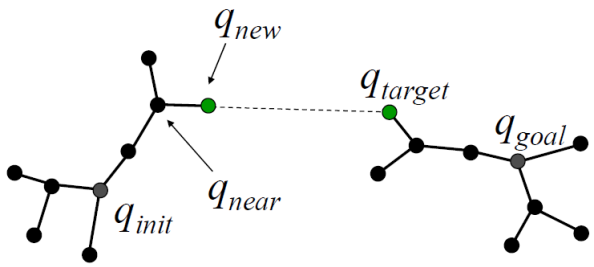
As basic RRT algorithm is more suitable for nonholonomic systems, RRT-connect subsequently developed to efficiently find a feasible path involving

holonomic systems with no differential constraints [27]. The principal mechanism of this algorithm is growing 2 trees at each end (q_I and q_G) and subsequently extending each of these trees toward one another until they have become connected and a solution is found. The algorithm can be described in pseudocode as follows [28]:

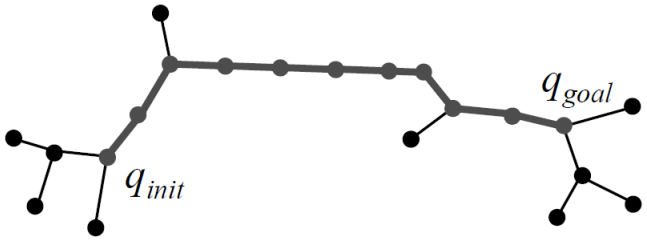
```
RRT_CONNECT_PLANNER(q_init, q_goal)
1. Tree_a.init(q_init);
2. Tree_b.init(q_goal);
3. for k = 1 to K do
4.    q_rand ← RANDOM_CONFIG();
5.    if not (EXTEND(Tree_a, q_rand)=Trapped) then
6.      if (CONNECT(Tree_b, q_new)=Reached) then
7.        Return PATH(Tree_a, Tree_b);
8.    SWAP(Tree_a, Tree_b);
9. Return Failure;

EXTEND(Tree, q)
10. q_near ← NEAREST_NEIGHBOR(q, Tree);
11. if NEW_CONFIG(q, q_near, q_new) then
12.    Tree.add_vertex(q_new);
13.    Tree.add_edge(q_near, q_new);
14.    if q_new = q then
15.      Return Reached;
16.    else
17.      Return Advanced;
18. Return Trapped;

CONNECT(Tree, q)
19. repeat
20.    S ← EXTEND(Tree, q);
21. until not (S = Advanced)
22. Return S;
```

In the pseudocode above, as indicated in line 1 and 2, the algorithm works by incrementally building two Rapidly-exploring Random Trees (RRTs) rooted at the initial and goal configurations ($Tree_a$ and $Tree_b$). In each iteration, while one tree is extended incrementally toward a random configuration (line 5), an attempt is made to connect the nearest vertex of the other tree to the new vertex (line 6). This CONNECT procedure will keep extending the branch (line 20) as long as adding a new collision-free branch is successful. The branch extension is directed by NEW_CONFIG procedure that makes

a motion toward q with some fixed incremental distance i.e. q_{new}. While extending the branch, three situations can occur: *Reached*, in which q is directly added to the RRT because it already contains a vertex between q_{near} and q_{new}; *Advanced*, in which a new vertex $q_{new} \neq q$ is added to the RRT; *Trapped*, in which the proposed new vertex is rejected because it does not lie in C_{free}. At the end of each iteration, shown in line 8, the roles are reversed by swapping the two trees. This procedure occurs iteratively until the maximum iteration count K is reached or when a path connecting both trees is found (Fig. 2.14(b)).



(a) RRT-Connect initial phase [27]



(b) RRT-Connect final phase [27]

Figure 2.14: RRT-Connect

Chapter 3

Reachability Guided Rapidly-exploring Random Tree Algorithms

3.1 Reachability Guided Rapidly-exploring Random Tree (RG-RRT)

RRTs in the past decades have been extensively used to solve many motion planning problems in the presence of differential or non-holonomic constraints. When an environment involves complicated kinodynamics constraints, RRTs turned out to be inefficient [29]. The obvious phenomenon of this inefficiency is a very slow progress toward the ultimate goal as it keeps on searching and extending in any random direction. Reachability Guided Rapidly-exploring Random Tree has been developed as it was based on the premise that sampling incurs much less time than the process of searching collision-free trajectories. It tackles the inefficiency problem by taking into account local reachability whilst expanding the tree. In other words, RG-RRT will throw away any randomly-sampled points that may have little or no effect toward the process of reaching the ultimate goal i.e. curvature bound constraint. Each time a new random configuration q_{sample} is sampled, reachability test will be performed to see whether this q_{sample} is within a reachable region from at least one existing point in the tree. As this assumption turns out to be not true, the q_{sample} will be discarded and another new random configuration will be sampled. This procedure will be performed iteratively until a collision-free trajectory connecting the initial configuration to the goal configuration is found or the algorithm has reached a maximum number of searching iterations bounding its execution time.

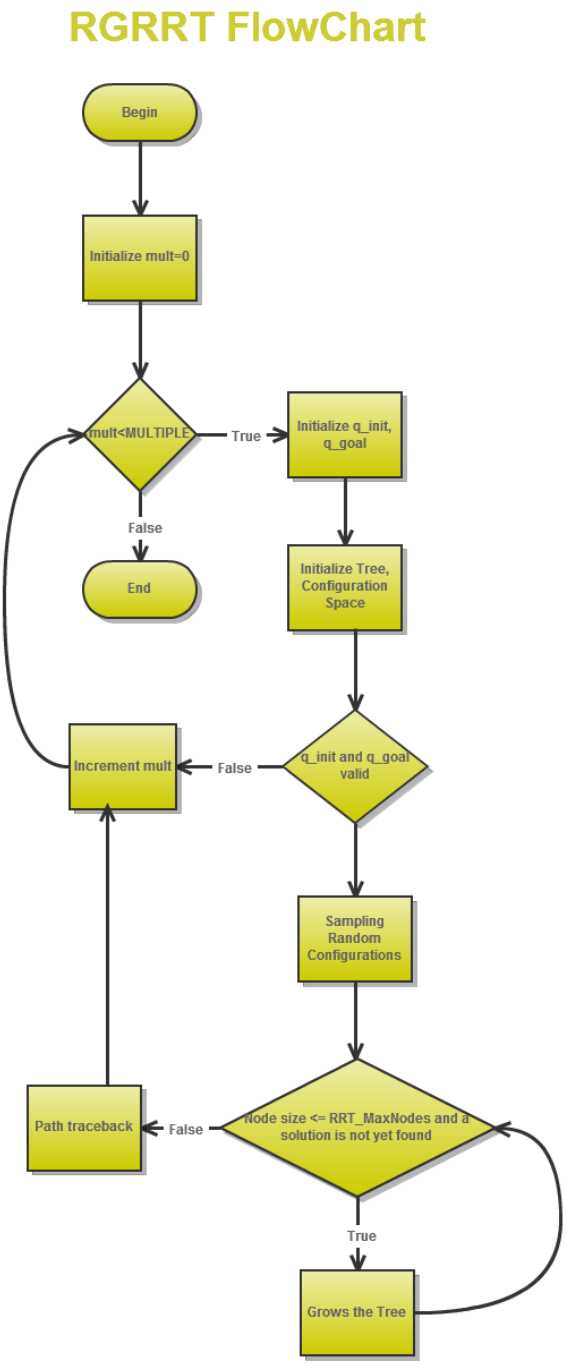


Figure 3.1: RG-RRT FlowChart

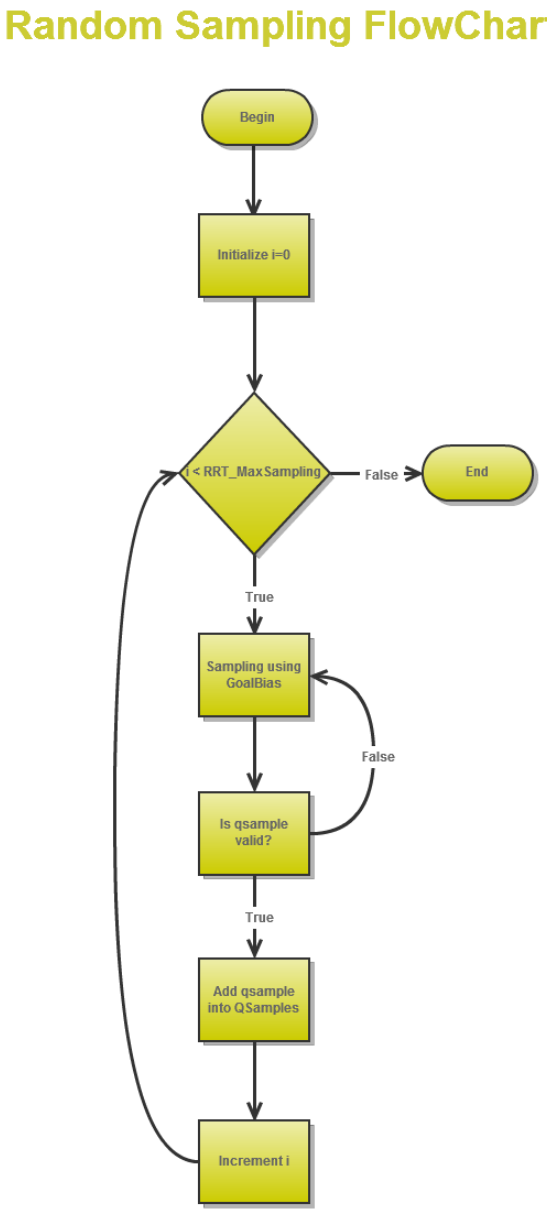


Figure 3.2: RandomSampling FlowChart

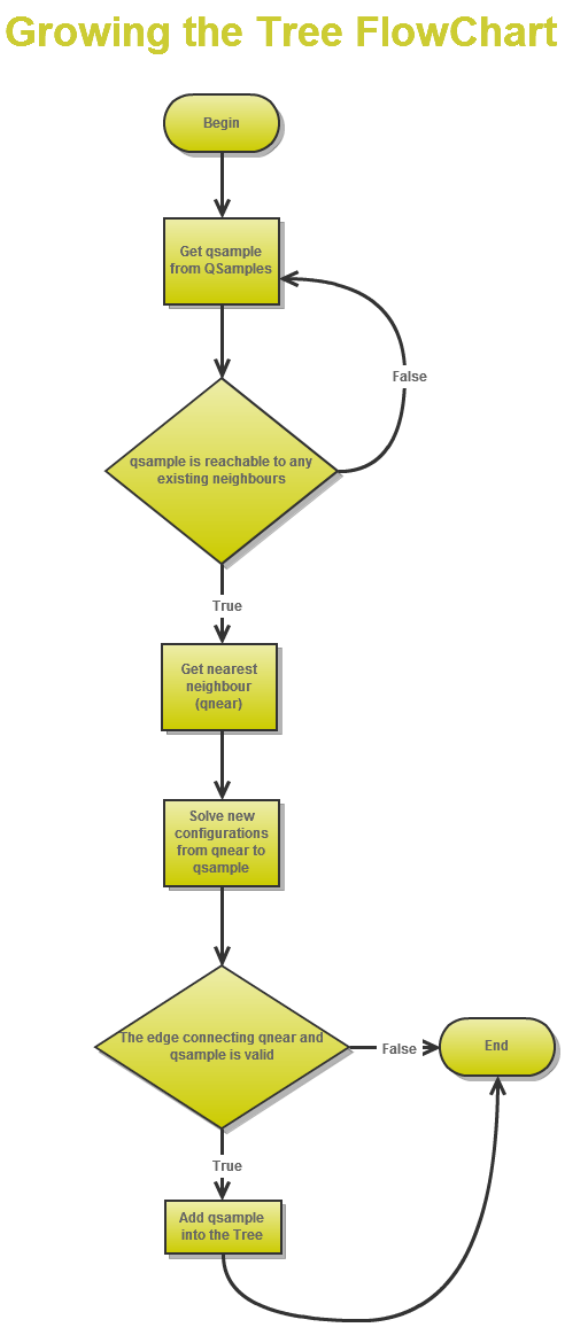


Figure 3.3: Growing Tree FlowChart

3.2 Extended Reachability Guided Rapidly-exploring Random Tree (ERG-RRT)

A non modified version of RG-RRT essentially works by expanding the tree with a combination of random exploration and biased motion toward the goal configuration. Expanding the tree always starts with an initial configuration. From this initial configuration, with probability p, it attempts to find the nearest point in the current tree and extend it toward the q_G and with probability 1 - p, it picks a random configuration [x, y] from the *configuration space*, finds the nearest point and extends the q_{near} to the q_{sample}. Therefore the *configuration space* exploration procedure can be described as follows:

```
function Explore (goal:state):state
 var p;
 p = UniformRandom in [0.0 .. 1.0];
 if p <= GoalBias then
  return goal;
 else
  return RandomState();
```

As can be seen above, during the exploration step the RG-RRTs expands the tree stochastically to any random direction or to its goal configuration. This procedure is performed iteratively until the goal configuration or the total number of nodes added into the tree has been reached.
As an extension to this, *waypoint cache* has been introduced [30] to speedup the convergence of the tree. *Waypoint cache* is essentially developed to limit the stochastic nature of nodes distribution. Instead of picking any new random configuration in the *configuration space*, the tree might also consider any successful configurations from its previous plans. By also taking into account the history from its previous plans, the tree will have the capability to "foresee" what might happen in the near future. The procedure in using *waypoint cache* can be depicted as follows:

```
function Explore (goal:state):state
 var p;
 p = UniformRandom in [0.0 .. 1.0];
 i = UniformRandom in [0 .. NumWayPoints-1];
 if p <= GoalBias then
  return goal;
 else if GoalBias < p <= GoalBias+WayPointBias then
  return WayPointCache[i];
 else
  return RandomState();
```

By storing the history of its previous plans, it possesses the knowledge of where a plan might happen again in the near future. Instead of having only two options to search and explore, now the algorithm may expand the tree stochastically to any new configuration in the *configuration space*, to its ultimate goal configuration or to any previous configuration stored in the *waypoint cache*. As can be seen above, with probability p $\leq$ GoalBias, a uniform configuration will be chosen; with probability p between the values of GoalBias and GoalBias+WayPointBias, a random waypoint is chosen; otherwise the goal configuration is then chosen as before.

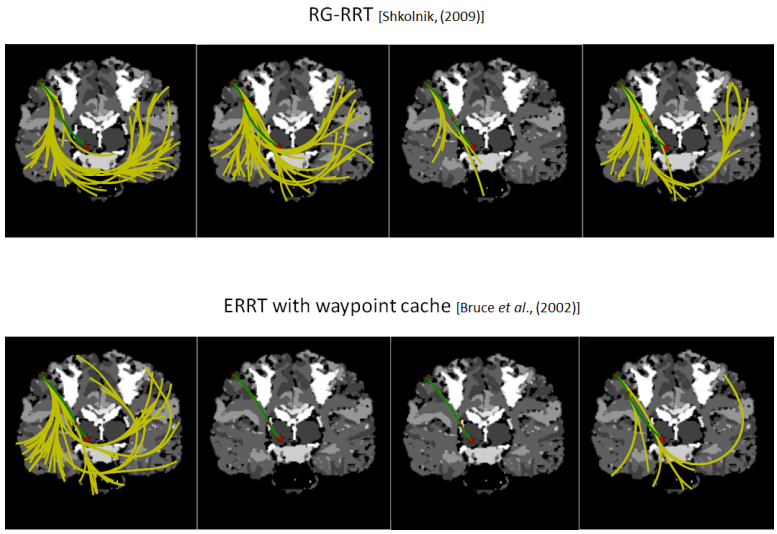


Figure 3.4: RG-RRT and RG-RRT with *Waypoint cache* comparison

3.3 Software Engineering Issues

3.3.1 MATLAB or C++

As we might already know, Computer Science recognizes two big main streams in programming language, compiled language and interpreted language. The true difference between these two streams lies in different conceptions of running applications. In compiled languages, the source code will be transformed into executable code that is specific to the hardware architecture (machine type) and software framework (operating system). Thus parsing and execution occurs in two distinct steps. Yet in interpreted language, only one step remains i.e. parsing and execution occur at the same time. The source code will be 'indirectly' executed or interpreted line-by-line by an interpreter at run time. Some advantages that can be drawn from the interpreted language include its relative ease of programming and

debugging, less memory consumption, cross-platform compatibility, ease of deployment and maintenance and JIT's compilers runtime optimization [31]. Besides its benefits, the different approach of execution will eventually have a small impact on its performance. It becomes evident as some interpreted languages incur from poor performance compared to executing executable code directly targeted on the host CPU. Apart from gaining speed performance, compiled language also possesses flexibility in distributing stand-alone executables. There are also some programming languages that possess a hybrid approach, half compiled and half interpreted. Like Java and C# for instance, the source code is pre-compiled into the intermediate code and executed at run time by its virtual machine i.e. Java VM and Common Language Runtime.

MATLAB is one of interpreted languages that is widely used in engineering which has gained its prominence due to its ease of usability and comes with sophistiated libraries for numerical computing, graphical simulation and symbolic computing capabilities. MATLAB turns out to be very useful especially for numerical calculations that involve matrix operations. However due to its execution approach nature, MATLAB might perform much slower than a compiled language. This is the point where C++ equipped with its optimal calculation and high speed floating point computation comes into play. However it is not a trivial thing to produce visual effects in C++. There are however some cross-platform application frameworks such as Qt that could be used to develop application software with a graphical user interface (GUI) yet it does not provide any morphological tools that can be applied to image segmentation, non-linear filtering, pattern recognition and image analysis. Even if it provides these capabilities, linking to a very big library with small need of capabilities (plotting and image filtering functionalities) may hurt the application performance as a whole.

To make worth the trade-off and gain benefits from these two types of programming languages, interfacing an interpreted language to a compiled language seems to be quite promising. As a start-up, the languages that will be used have to be properly defined, what to use and how it can be used. In this project, C++ and Python are the languages of my choice. Together with C, C++ has mainly been used for developing application programs due to its speed and power. Compared to C, C++ provides more options to interface with other high level languages such as Python. Basically both MATLAB and Python are considered as interpreted languages. As in MATLAB, Python also supports data processing and plotting functionalities through the use of NumPy, SciPy and Matplotlib packages. While Python is totally free and a general-purpose language, MATLAB is a proprietary scripting language attached to its numerical computing environment that is developed and maintained by its developer, MathWorks. It is indeed true that most

scripting languages are much simpler than general-purpose languages but not to Python. Besides possessing more meaningful syntax, it is also inherently object oriented. As it is inherently object oriented, each of member variables and methods could be encapsulated from the outside world as much as other object oriented languages do such as C++, C#, Java etc. In short, by having C++ and Python both combined, high performance algorithm with excellent prototyping and plotting functionalities can be guaranteed.

3.3.2 System Requirements

The neurosurgical flexible probe ***(STING)*** is modelled as a nonholonomic robot and can be steered in two dimensions. System requirements that have to be considered while developing the path planning are [48]:

- **Mechanical constraints:** The flexible probe imposes a minimum radius of curvature r_{min} constraint on the path. There are also some constraints that restrict the flexible probe to change its curvature drastically, in order to have a continuity and boundedness constraint on the derivative of the curvature.
- **Inputs for the path planner:** The set of inputs for the planner are:
  - The pre-operative CT/MRI images that will later be converted into configuration space by considering also thickness of the flexible probe
  - The initial or entry configuration (position, orientation)
  - The goal configuration
- **Outputs for the path planner:**
  - A curvilinear trajectory path that is bounded by constraints on the minimum radius of curvature, curvature and its derivative continuity
  - A cost value of the generated path, reflecting its intersection with different risk areas on the map

3.3.3 System Design

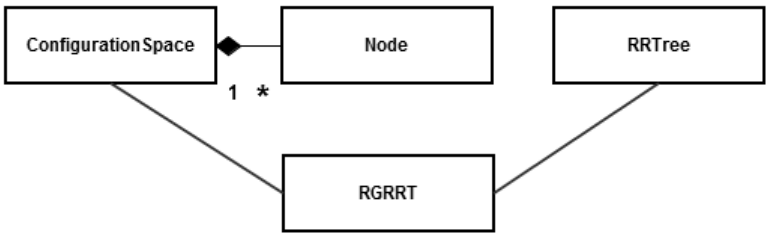


Figure 3.5: RG-RRT Class diagram

While Fig. 3.5 depicts a simple system architecture of RG-RRT, Fig. 3.6 - Fig. 3.9 show the member of ConfigurationSpace class, RGRRT class, RRTree struct and Node class. ConfigurationSpace *class* represents the *configuration space* of the physical system, which in the case of neurosurgery is the brain. The class has the full knowledge of the configuration map that

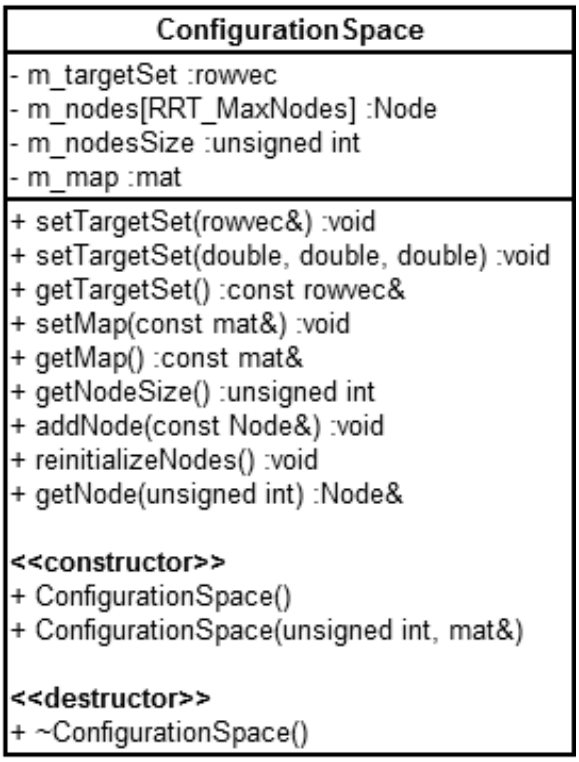


Figure 3.6: ConfigurationSpace class

will be explored by the flexible probe including the ultimate target configuration. This class has a composition type of association that is depicted as a filled diamond and a solid line with the Node *class*. This relationship imposes that the ConfigurationSpace *class* owns each of Node instances built in the tree. The algorithmic procedure lies on the RGRRT *class*. Besides having the sole responsibility of everything that is related to the algorithm (i.e. sampling random configurations, checking the reachability of points, validating edges, building new configurations), RGRRT *class* also provides helper functions for building as well as verifying and validating the tree.

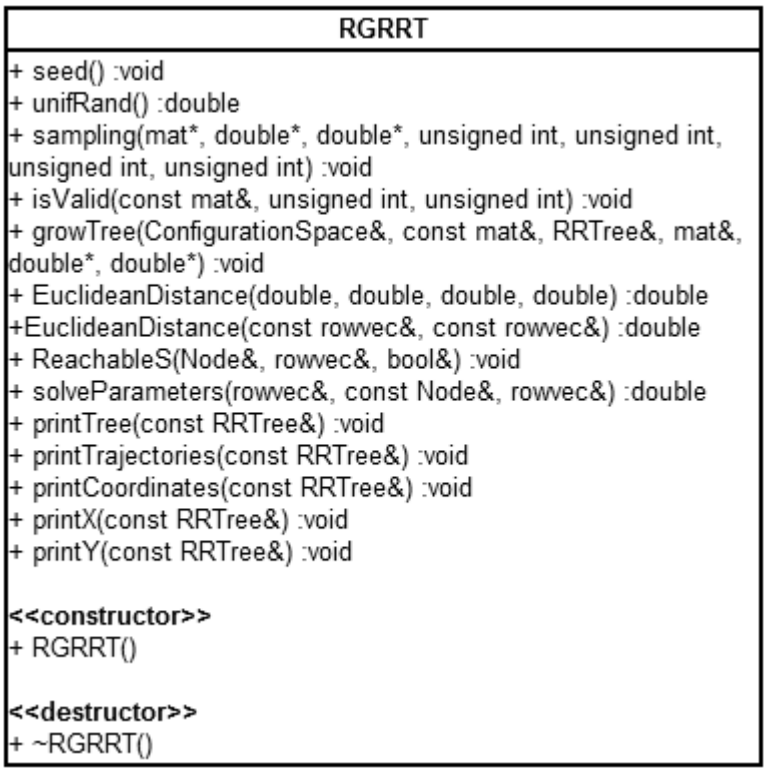


Figure 3.7: RGRRT class

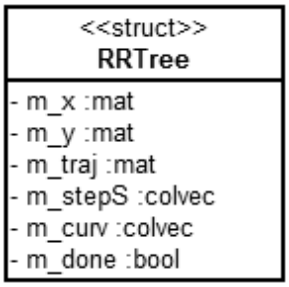


Figure 3.8: RRTree class

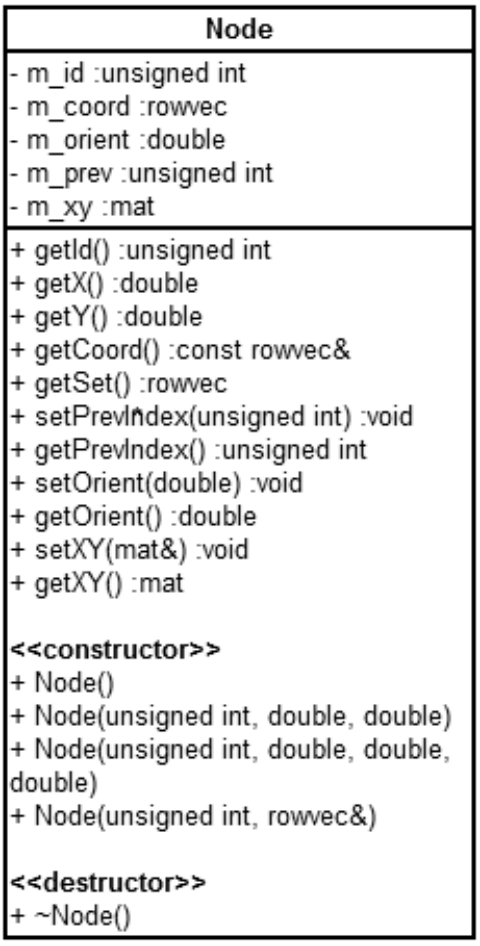


Figure 3.9: Node class

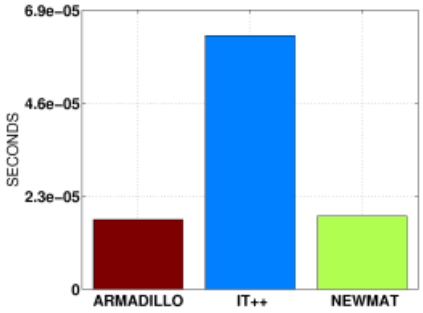
3.3.4 System Integration

RGRRT algorithm will be implemented in C++ while any visual effects (e.g. plotting, image processing) will be implemented in Python. To easily integrate the algorithmic side and the viewer side, the algorithmic side will be built into a shared library that will then be loaded dynamically by the application at run time. Loading dynamic link library into Python will be done using Ctypes. Ctypes is a foreign function library for Python that provides C compatible data types, and allows calling functions in dynamic-link libraries (DLLs) or any shared libraries. A snippet of the Python code can be seen as follows:

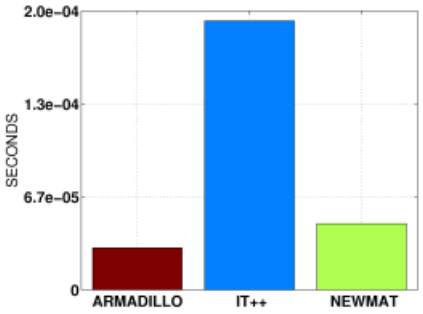
```
import ctypes as ct
dll = ct.cdll.LoadLibrary('RG-RRT.dll')
dll.rg_rrt()
```

Loading RG-RRT dynamic link library can easily be done by invoking **LoadLibrary()** and specifying the name of the shared library as its argument. After the shared library is successfully loaded, it has full access to all attributes and functions that reside in the library. Accessing these attributes and functions from loaded Dlls can be as easy as accessing them as attributes of Dll objects.

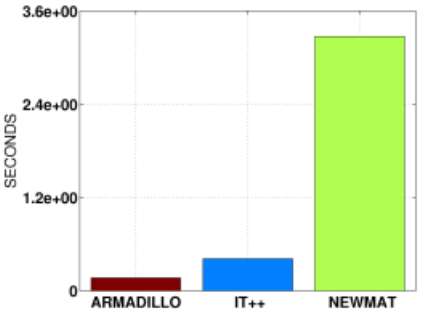
To support matrix and vector operations in C++, **Armadillo** C++ linear algebra library is considered to use for this project. One of primary reasons to use this library is due to its seamless integration with C++ and syntax similarity to MATLAB. Hence many integer, floating point and complex numbers are supported, as well as many math functions. Armadillo supports optimal and high-speed computations due to its integration with LAPACK and multi-threaded MKL or ACML libraries.



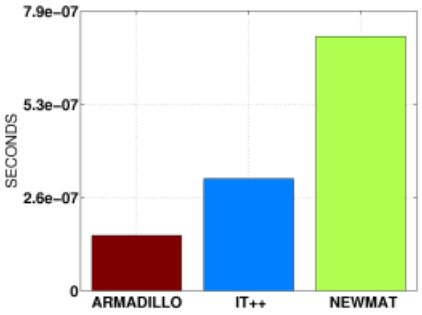
(a) Two matrices addition [32]



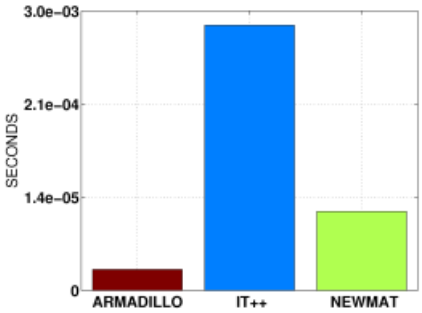
(b) Four matrices addition [32]



(c) Four matrices multiplication [32]



(d) Submatrix manipulation [32]



(e) Multi-operation expression [32]

Figure 3.10: C++ matrix libraries speed comparison

3.4 Parallelization

RG-RRT has gained its popularity among any other planning algorithms particulary for solving single-query motion planning problems. RG-RRT could solve a wide range of motion planning problems including holonomic, non-holonomic, kinodynamic, or kinematic closure constraints [33] - [35]. As it becomes more and more popular, the urge to optimize the algorithm has also become apparent. There are some advance improvements that have been developed to increase its efficiency and speed up the path exploration such as by adapting tuning of the sampling domain [37], increasing the efficiency of the nearest neighbour [38] or implementing gap reduction techniques [36]. With the advance of computing technology, the emergence of multi-core processors and the evolution of highly parallel, multithreaded, manycore processor with very high memory bandwidth, some parallelization techniques have been explored in recent times. Some of these parallelization techniques will be covered in the proceeding chapters.

Chapter 4

Parallel RG-RRT

4.1 Introduction to Parallel Computing

The very first microprocessor created was designed for the US Navy F14A "TomCat" fighter jet from 1968 - 1970. Microprocessors were not commercially available on the market until 1971 when Intel Corporation released **Intel 4004**, a 4-bit central processing unit (CPU). With instruction set of 46 instructions and the maximum clock speed of 740 kHz, Intel 4004 was built into Busicom 141-PF printing calculator. By incorporating a micropressor, this calculator was capable in doing some fairly simple calculations. However, with the advance of computing technology and the increase of device complexity, the demands of having more and more processing power and bandwidth were inevitable. To incorporate many capabilities onto a chip, integrating more and more number of transistors on a chip are required. These course of events have been widely known as the **Moore's Law**. Gordon Moore, Intel co-founder has predicted nearly 40 years ago that transistor density on integrated circuits doubles about every two years, as can be seen in Fig. 4.1. Through the years, his prediction has proven to be valid and in 2012, a home personal computer or PC might have an equivalent processing and computing power of 2000's medium scale movie industry render-farm. In short, microprocessors have now become smaller, denser and more powerful.

Most of today's high-performance microprocessors employ *out-of-order execution*, *on-chip chaching*, *prefetching* microarchitecture techniques to reduce memory latency at the cost of integrating more transistors on a chip. Besides integrating more transistors, chip manufacturers also seek some novel ways to increase clock rates (Fig. 4.2) and to exploit instruction-level parallelism. Unfortunately everything does not come for free. Later it was found out that that by increasing the clock rates, it will also significantly increase the power consumptions. As the power consumptions increase, CPU's were hitting the

power wall and high-frequency micro-architectures are not very suitable for many of the low-power design techniques that had been invented to deal with the power issue. Facing this fact, chip manufacturers have turned their focus more on the instructions per clock cycle, the other metric that determines the CPU performance. To effectively increase the instructions per clock cycle, they try to improve the efficiency of the logic architecture that is rearranging the flow of data in a CPU so that more work can be done without increasing the clock rate.

To increase the CPU performance even further, parallel computing turned out to be necessary. The parallelization can be achieved by taking full benefits of having more transistors on a single die. Apart from the performance boost, having more cores will also reduce energy dissipation. Execution time and cost can be described as follows:

```
Execution time = (total work)/(aggregate speed)
* Serial execution time(Ts) = Ws/V(M)
* Parallel execution time(Tp) = Wp/((pV(M/p))

Cost = (number of processors)x(execution time)
* Serial cost = Cs = Ts = Ws/V(M)
* Parallel cost = Cp = pTp = Wp/((pV(M/p))
```

```
Power = (C x V x V x F)/4 Performance = Cores x F
Capacitance Voltage Frequency
```

By integrating more transistors onto a chip, it will increase the capacitance which eventually can increase the performance, whereas by adding more cores into the die it may reach the same performance yet merely $\frac{1}{4}$ power required. Due to these facts, there was massive transition from single core to multi core processors done by semiconductor chip manufacturers. Following the hardware transition, more and more multithreaded applications are created in order to utilize the computing power of multi core processors.

To be able to create an effective and efficient parallel algorithm, some factors have to be considered. They are:

- *Load balance*: Work distribution among multiple processors or any other computing resources. By maximizing the load balance, it will minimize the response time, maximize the throughput and avoid the overload.
- *Concurrency*: Ability to work on computations or calculations simultaneously

- *Overhead*: Excess or additional need of resources (i.e. time, bandwidth, energy) that are required during the execution

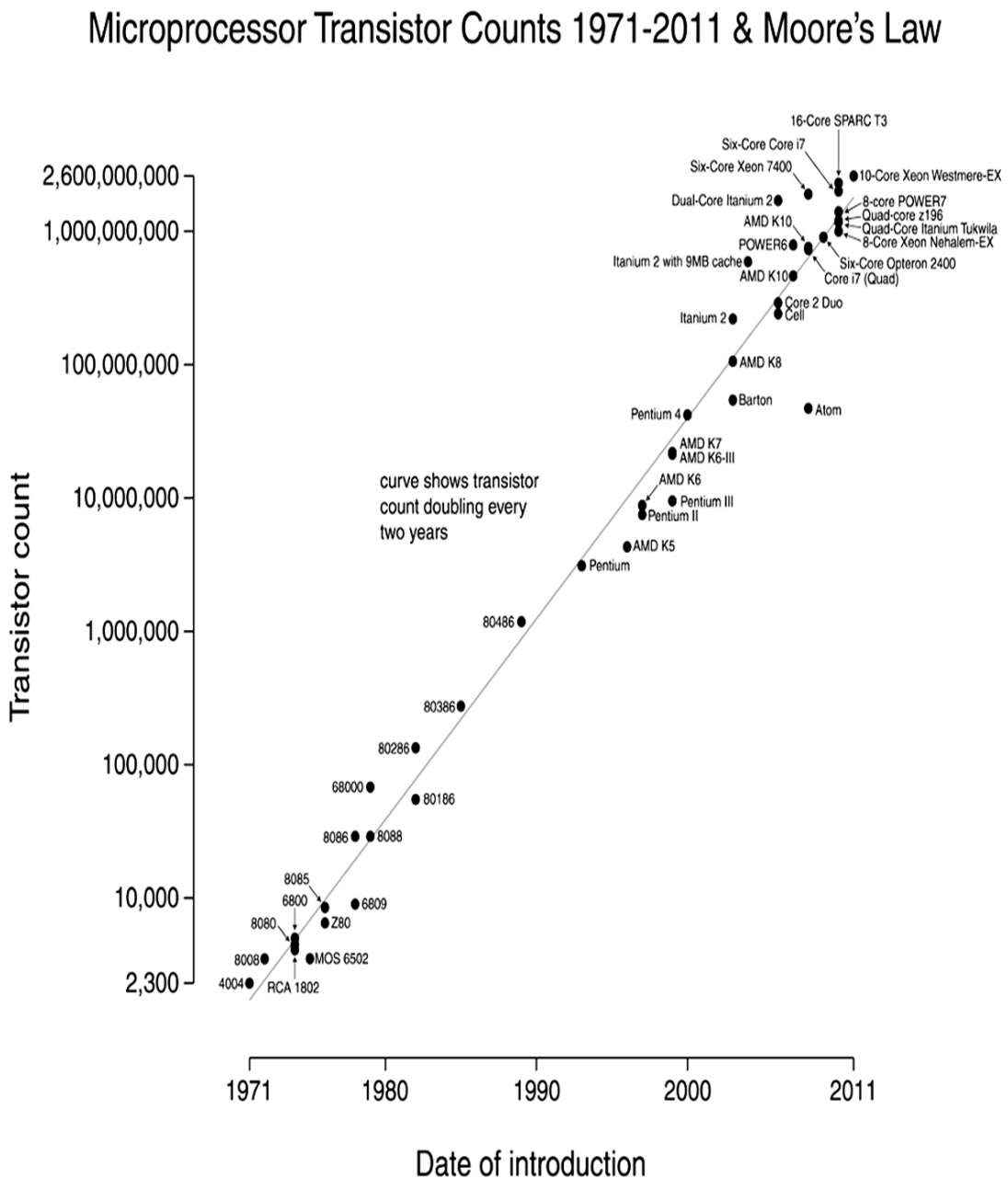


Figure 4.1: The exponential growth with transistor count doubling every two years hence conforming to the Moore's Law [40]

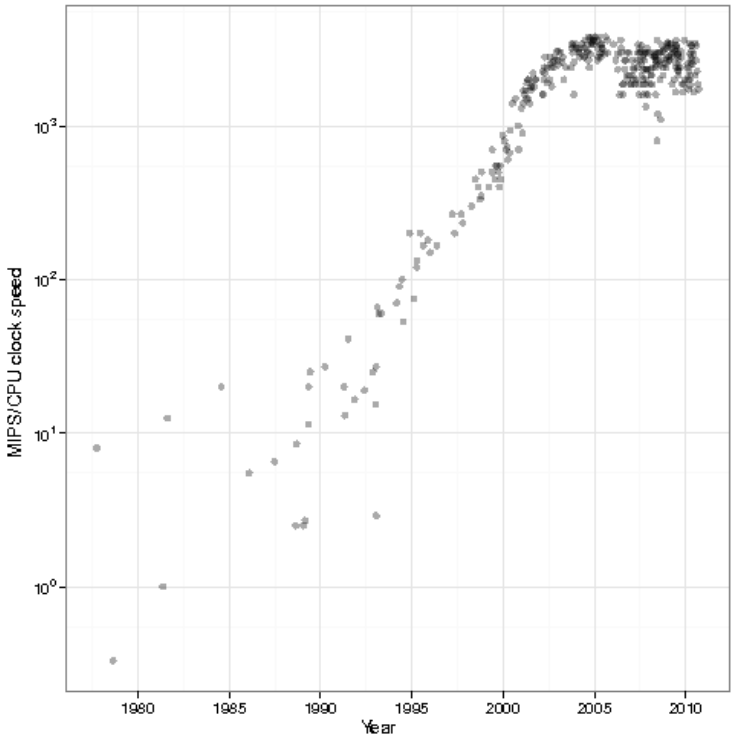


Figure 4.2: The increase of clock rates [41]

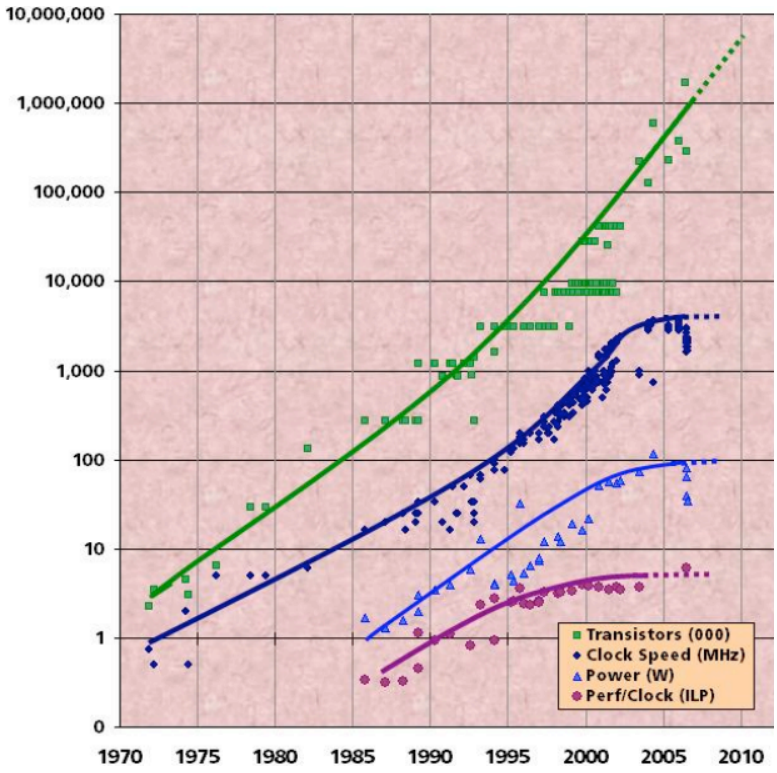


Figure 4.3: Current semiconductor technology trend [41]

4.2 CPU Parallelization

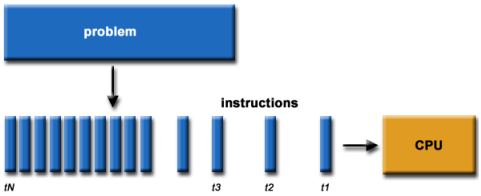
In good old days where everything is done serially, a problem is divided into series of small problems where these small problems will then be executed by the CPU as a discrete series of instructions one at a time. An illustration of this process of execution can be seen in Fig. 4.4(a) where a big chunk of "problem" is broken down into series of instructions from *t1* until *tN*. The CPU will first take *t1* to be processed. Once the first execution has been completed, the CPU will then process *t2* and sequentially taking down *tNth* until the whole series of instructions execution is completed. Different from its counterpart, *parallel computing* relies on its computing power resources to execute series of instructions simultaneously. As it is illustrated in Fig. 4.4(b), a big chunk of problem is divided into small yet independent problems that can be executed concurrently. These independent problems will then be broken down into series of instructions that later be fed into different CPUs. Among different CPUs, each part of these instructions can be executed simultaneously. By executing instructions simultaneously, computational problem can be processed more effectively.

In this section of the report, we are going to discuss some CPU parallelization techniques on RG-RRT algorithm. Some early work has been done previously on implementing CPU parallel versions of the RRT algorithm, such as [42], [44], [49] and [50]. However, all these implementations focus on a distributed memory model rather than on a shared memory model of parallel computation. Eventhough a distributed memory model offers high scalability, yet computing clusters are expensive and specialized. On the other hand, a shared memory programs are usually shorter and easier to understand as it does not have to involve message passing communication system thus giving more transparent process-to-process communication.

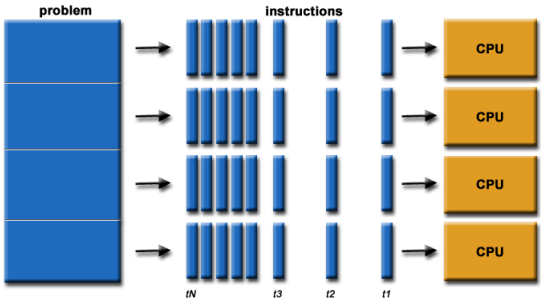
As it has been covered in 3.3.2, ***(STING)*** imposes some mechanical constraints that must not be violated in order to function properly, thus the focus of this work is not only finding the best solution to improve the performance of the RG-RRT algorithm but also improving the performance of the flexible probe with its inherent constraints.

4.2.1 OpenMP Programming Model

OpenMP is an Application Programming Interface specifically designed for multi-threaded applications developed in shared memory architecture [43]. OpenMP library supports parallel programming in C/C++ and Fortran on all architectures, including Unix and Windows NT platforms. As OpenMP programs are multi-threaded applications, parallelism is done exclusively through the use of threads. A *thread* is a lightweight process that can be



(a) Serial Computing [39]



(b) Parallel Computing [39]

Figure 4.4: Two basic types of computation

scheduled by an operating system. The life of a thread strongly depends on a process. In other words of saying, threads are even located within resources of processes. When a process "goes out of scope", the corresponding thread belongs to that process will cease to exist.

The openmp specification defines a set of pragmas. These pragmas are compiler directives that specify how to process the block of code that follows the directive. The omp pragma that is most common to find is the *#pragma omp parallel*. This pragma denotes a parallel region. Meaning to say that the block of code that follows this pragma will be executed parallely. Besides this parallel pragma, there are many other omp pragmas such as *#pragma omp section*, *#pragma omp master*, *#pragma omp barrier*, *#pragma omp single*, *#pragma omp critical* to name a few.

As OpenMP is a parallel programming model specifically designed for *Shared Memory Architecture*, keeping shared data integrity is unquestionably important. OpenMP therefore provides some Synchronization Constructs. They are:

- Barrier Construct : defines a waiting/pending mechanism at a specific point in the code for all threads before proceeding further

- Ordered Construct : defines a sequential order of execution within a parallel loop

- Critical Construct : defines a *critical section* with the intention of ensuring only one thread has an access to the same shared data at a given moment in time.
- Atomic Construct : defines a *one thread-execution* section similar to Critical construct. In contrast to the others, this construct only applies to a single assignment statement instead of a block of code
- Locks : defines a lock mechanism for a particular block of code. Similar to Critical and Atomic constructs but with greater flexibility
- Master Construct : defines a block of code that is executed only by the master thread

Thread-based parallelism seems to be relatively easy compared to any other parallelism methods. However, there are some caveats that have to be considered :

- Cost of starting a thread or process
- Cost of communicating shared data
- Cost of synchronizing
- Extra computation cost

4.2.2 OR Parallel RG-RRT

As we have covered so far, the exploration of RG-RRT has stochastic nature. At one moment in time, the route could be different with other time. OR Parallel exploits the stochastic nature of RG-RRT[42]. In multicore system, when each core is responsible in expanding a random tree, each of them is very likely to produce different results. One of them will find a solution earlier than the rest. As soon as a solution is found, a global variable *stopCondition* indicating that the quest should be stopped will immediately be set to true. A snippet of OR Parallel algorithm can be described as follows:

```
OR Parallel RG-RRT
Input : Initial configuration q_init, the configuration space C
Output : tree
1. tree ← initTree(q_init)
2. Q_Rand[] ← sampleRandomConfigurations(C)
3. i ← 0
4. while not stopCondition and nodeSize ≤ MaxNodes
5.    q_rand ← Q_Rand[i]
6.    q_near ← findNearestNeighbor(tree, q_rand)
7.    q_new ←extend(q_near, q_rand)
8.    if valid(q_near, q_rand)
9.      stopCondition ← true
10.   i++
```

As indicated in line 2 after the tree initialization takes place, *sampleRandomConfigurations* procedure generates all random configurations before even starts expanding the tree. This is done to make the RG-RRT gains full benefits from parallelism. The space exploration in fact begins in line 4 when entering the while loop. However before entering the while loop, it should first verify that the number of nodes added in the tree (*nodeSize*) is still less than the maximum number of nodes (*MaxNodes*) allowed. After the verification, instead of generating a random number, line 5 shows that a random number *q_rand* is taken out from previously generated *Q_Rand* array. Similar to the basic RRT, finding a nearest neighbor, finding a new configuration (*q_new*), and validating the connecting edge between *q_near* and *q_rand* will be subsequently done. Once a solution is found, the corresponding thread will set *stopCondition* to true indicating the other independent processes to stop expanding its own trees and discard the results. Eventhough *OR Parallel RG-RRT* can perform better than running the algorithm serially, its implementation is only limited by the performance of a single processor [42]. The faster execution times cannot be slower than running it in a single processor and it is almost impossible to have all processes finish exactly at the same time hence it is an effective method to avoid long time execution.

```
1. #pragma omp parallel num_threads(8) default(shared)
2. firstprivate(space) private(tree)
```

As it was previously discussed in section 4.2.1, openmp defines a set of pragmas to specify how to process a block of code that follows the directive. This means, before entering the planner procedure one must first specify how to act upon a block of code by specifying an omp pragma. In the code snippet above, it uses *#pragma omp parallel* to indicate that 8 threads will be executed simultaneously with all variables shared by default. The clause

private is used to indicate that each thread owns its own copy of the variable listed (i.e. tree) whereas **firstprivate** adds an additional functionality that the variable (i.e. space) will retain its original value of initialization existing before entering the **parallel** construct (line 1).

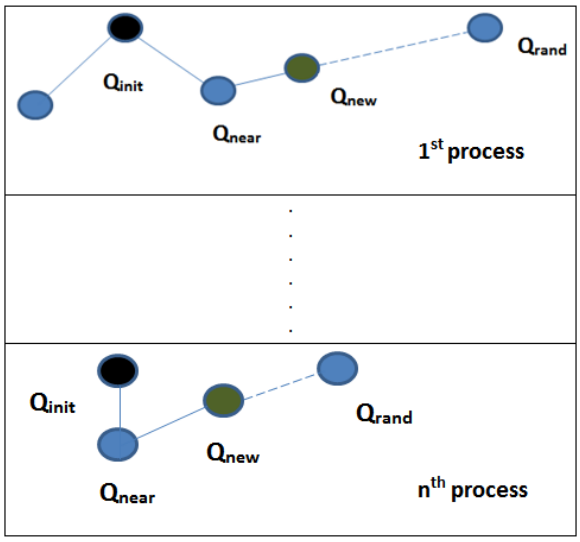


Figure 4.5: *OR Parallel* RG-RRT

4.2.3 AND Parallel RG-RRT

Different from *OR Parallel* where each core/process tries to expand its tree in its own way, in *AND Parallel* implementation, each process cooperatively and collaboratively builds the tree until a solution is found. Due to each process contribution, the tree will grow faster than running it only in one process. However, this method cannot guarantee the short execution times as sometimes the growth of the tree is not correctly directed toward the exterior or even trapped in local minima. A code snippet below indicates that 8 threads will be run simultaneously and collaboratively to build the tree and the clause *shared* indicates that there is **only one copy** of data shared among different processes.

```
#pragma omp parallel num_threads(8) default(shared)
```

This method also possesses an Achilles heel when not wisely used. As the *tree* and the *configuration space* are now shared, proper synchronization must be carefully considered to keep the integrity of the data especially when more than one process has access to the same resource. Code snippet below illustrates the use of Critical construct in RG-RRT. Here Critical

construct is exclusively used for a block of code starting from the new node addition until the goal verification. This synchronization block is required as the *configuration space*, the *tree* and *stopCondition* are shared among different threads.

```
#pragma omp critical
{
 if(!stopCondition)
 {
  // add the new node into the space
  p_space.addNode(qnew);

  // test if the new point is in the goal region
  if(p_space.getTargetSet().at(0) == qsample(0) &&
  p_space.getTargetSet().at(1) == qsample(1))
  {
   p_tree.m_done = stopCondition = true;
  }
 }
}
```

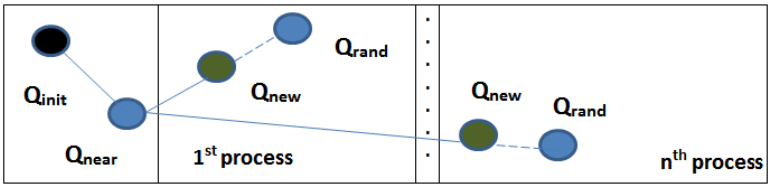


Figure 4.6: *AND Parallel* RG-RRT

4.2.4 AND-OR Parallel RG-RRT

In *OR Parallel*, the minimum execution time cannot be improved as its performance is only limited by the performance of a single processor. However it does minimize the maximum execution times as the faster execution times cannot be slower than running it in a single processor. In *AND parallel*, due to processors collaboration, the solution will be found faster than running it only in a single processor. However care has to be taken as proper synchronization must take place to avoid conflicts. Combining these two methods, might yield in minimizing the maximum execution times yet gaining performance speed-up. In my implementation, as I am using a quad-core processor with Hyper-threading (thus 8 threads available in total), I have divided them into two separate groups, 4 threads running for each group.

AND Parallel is run within each group, and *OR Parallel* is run between these two separate groups. Thus processes in the same group will collaboratively expand the tree and as soon as a group finds a solution, it will notify the other group to stop searching for a solution.

4.2.5 Manager-Worker RG-RRT

In *Manager-Worker* implementation, a classical approach to perform a *functional decomposition* has been taken. Similar to [44], in this *functional decomposition*, tasks are divided based on the ones that require knowledge of the tree and those that do not require to know the existence of the tree. In Fig. 4.7, it can clearly be observed that the tasks of the Manager involve initializing the *tree*, sampling random configurations, verifying the random configurations reachability from the flexible probe and finding nearest neighbours. What the Workers do are generating new configurations and ensuring the validity of the edge connecting q_{near} to q_{rand}. When the edge is valid, then it is the Manager task to add the new node into the *tree*.

Code snippet below describes briefly how the algorithm is carried out. There are three global lists (i.e. qnodeData, qrandData, qnearData) used as means of communication between the Manager and the Workers. As previous algorithms, *stopCondition* indicates whether a solution has been found. When it has not found a solution, the Manager checks whether qnodeData list is not empty. If it is not, it means there are some qnodes waiting to be inserted into the *tree*. Then it samples a new random configuration, finds the nearest neighbour and adds both of the variables into its corresponding lists, to be further processed by the Workers. Any idle workers will check whether there are any q_{rand}'s and q_{near}'s to be processed. If there are any, a worker will initiate to fetch the data from the lists and calculate its corresponding q_{node}. Once the validity of the q_{node} is guaranteed, it adds the newly constructed q_{node} into the list, to be further inserted into the *tree* by the Manager.

```
while(!stopCondition)
{
  if processID = Master then
  {
    while(qnodeData is !empty)
    {
      #pragma omp critical(qnodeData)
      {//get qnode from the list}
      if(!stopCondition)
      {//add qnode into the space}
    }

    qrand = samplingRandomConfiguration()
    qnear = nearestNeighbour()

   //add qrand and qnear into the list
  }

  else then
  {
    if(qrandData is !empty and qnearData is !empty)
    {
      #pragma omp critical(qrandData)
      {//get qrand from the list}
      #pragma omp critical(qnearData)
      {//get qnear from the list}
      empty = false
    }

    if(!empty)
    {
      qnode = solveConfiguration()
      if(valid)
      {
        #pragma omp critical(qnodeData)
        {//add qnode into the list}
      }
    }
  }
}
```

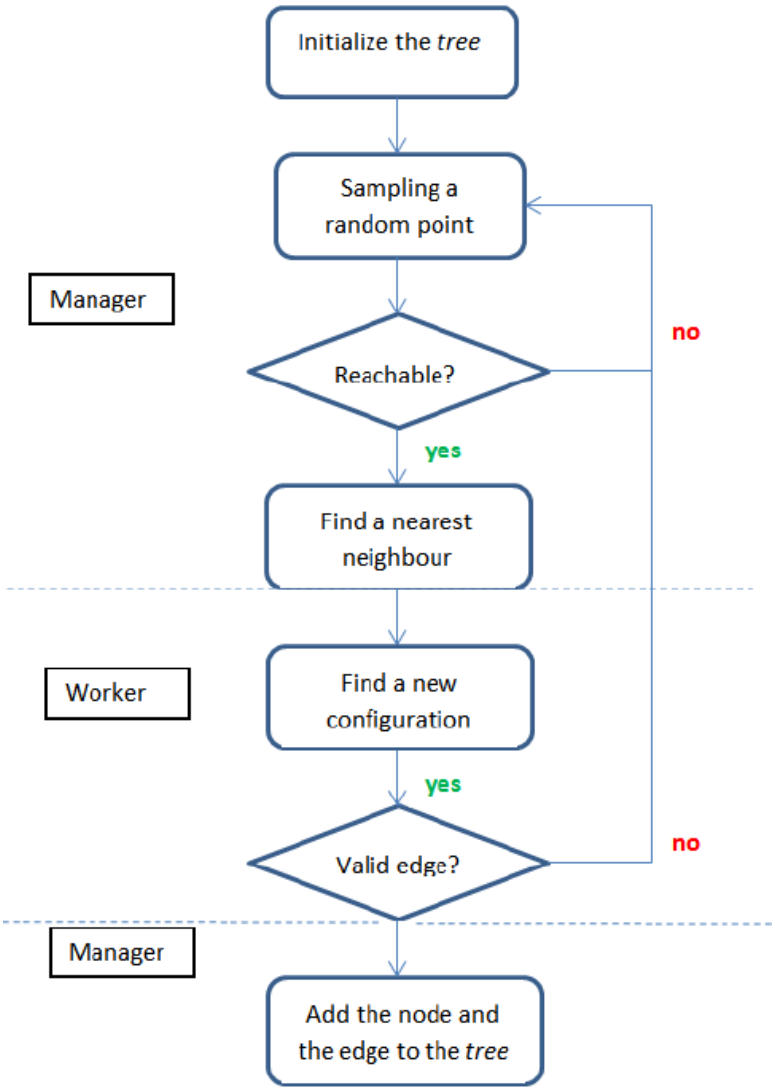


Figure 4.7: *Manager-Worker Parallel* RG-RRT

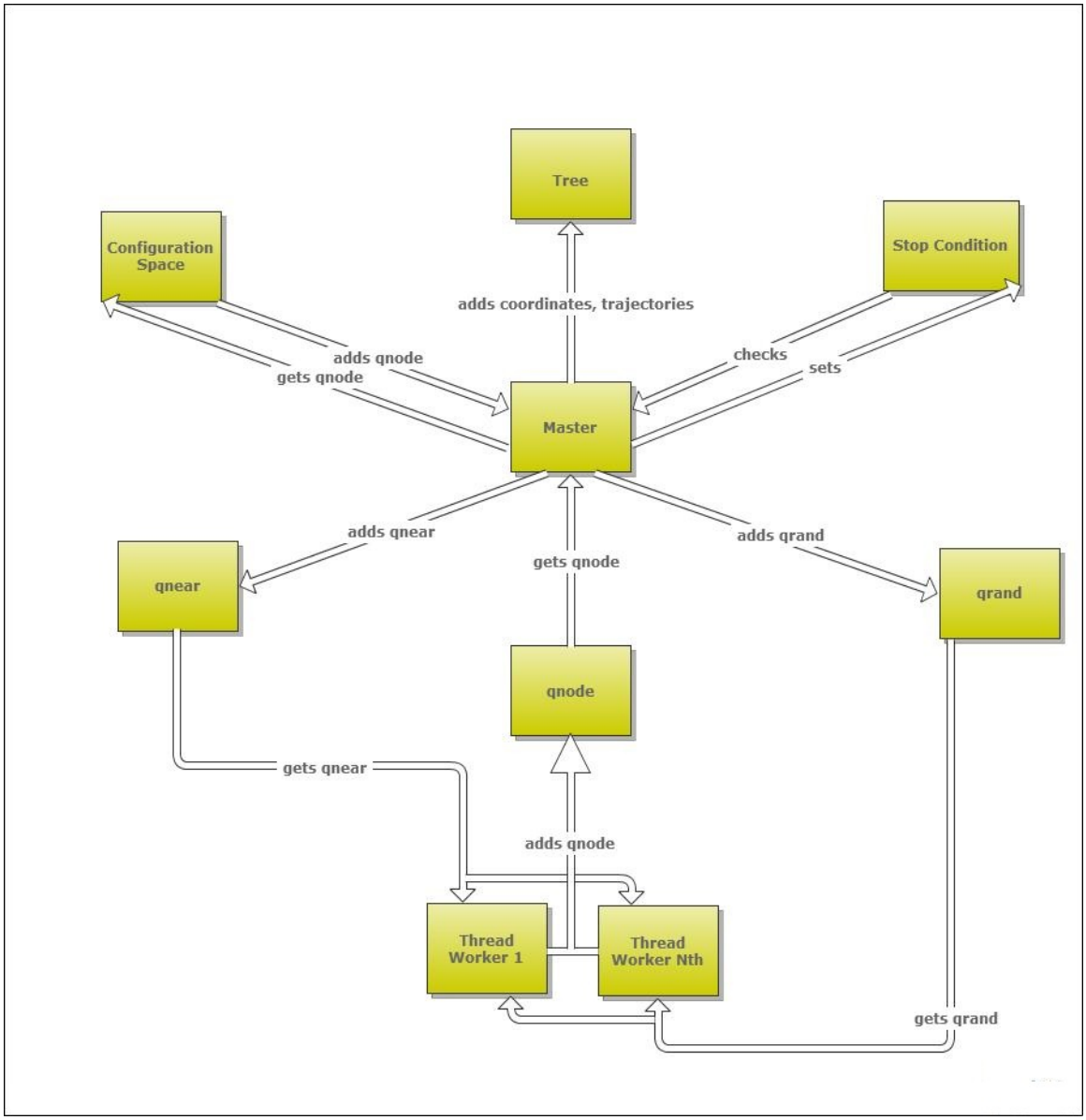
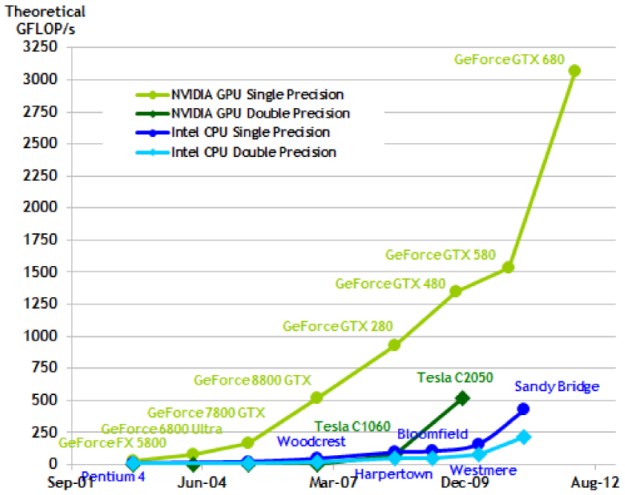


Figure 4.8: *Manager-Worker* Logic Diagram RG-RRT

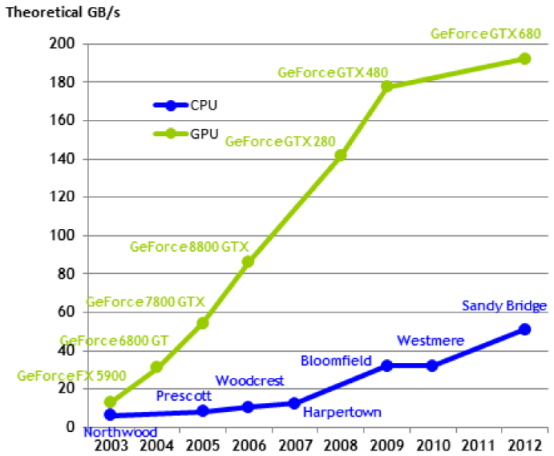
4.3 GPU Parallelization

Graphical Processing Units (GPUs) have been studied in recent years by many researchers to be able to surpass the high-speed performance of multi-core processors. Researchers believe that this is possible due to the number of transistors and less caches and flow control that a GPU does have compared to a CPU (Fig. 4.9(a)). GPUs also possess higher memory bandwidth than CPUs which leads to very short memory access times (Fig. 4.9(b)). Besides these facts, the GPU also supports much more data-parallel programming computations with high speed of arithmetic calculations compared to the CPU. In other words of saying, the GPU is more suited to parallelism than the CPU.

There has been some earlier work done by J. Bialkowski et al. in parallelizing RRT in the GPU [52]. However, the focus of the work was only to massively parallelize collision-checking procedure, a procedure that is considered to be the most computationally expensive procedure in the sampling-based motion planning algorithms. As the nature of the algorithm is stochastic, there is a high possibility that many sampling numbers are required to be generated. By also parallelizing the generation of random numbers, we could get the most benefit out of GPGPU parallelism.



(a) Floating-Point Operations per Second[45]



(b) Memory Bandwidth[45]

Figure 4.9: CPU vs. GPU

4.3.1 NVidia CUDA

NVidia CUDA is a parallel computing and programming model invented by NVIDIA in 2006 that supports data and task parallelism on a GPU. CUDA supports a few extensions of high-level programming languages, application programming interfaces such C, C++, FORTRAN, DirectCompute, OpenCL, OpenACC.

4.3.2 CUDA Programming Model

The smallest process in CUDA is done via threads. In fact, CUDA exploits GPU parallelism using threads. CUDA Programmers are able to execute portions of code in GPU by running kernel functions. A kernel is essentially a function callable from *the host* (as we refer to the CPU and the system memory) and executed on *the device* (as we refer to the GPU and its memory). Once a programmer runs a kernel from the host, the kernel will then be simultaneously run by threads assigned to that specific kernel.

CUDA allows programmers to divide a problem into smaller sub-problems that can further be solved simultaneously by blocks of threads. Within each of these sub-problems, independent sub-sub-problems can then be solved parallely by all threads belonging to that specific block. Fig. 4.10 depicts the inner structure within a kernel function. Grid in CUDA is organized as a two dimensional array of blocks and within each block, thread blocks are organized into a one-dimensional, two-dimensional, or even three-dimensional array.

4.3.3 CUDA Memory Hierarchy

The largest volume of memory available in a GPU is called **global memory**. Although it has the biggest capacity than any other types of memory residing in a GPU, it is however the slowest type of memory next to **local memory**. Besides having a smaller capacity, another difference between these two types of memory is the accessibility. Global memory can be accessed by both the host and the device however local memory is only attached to a thread. This implies that each thread owns a copy of its local memory and it is not shared among any other threads.

There are two types of memory in CUDA that provide high-speed memory access. They are **registers** and **shared memory**. Registers are automatic variables that reside within a kernel function. The memory size of registers are very limited and overuse of them will degrade the performance as compilers might allocate these variables in the local memory. As in the case of the local memory, despite its high-speed memory access, registers are local

to threads. To share data among threads residing in the same block, shared memory is therefore commonly used as shown in Fig. 4.11.

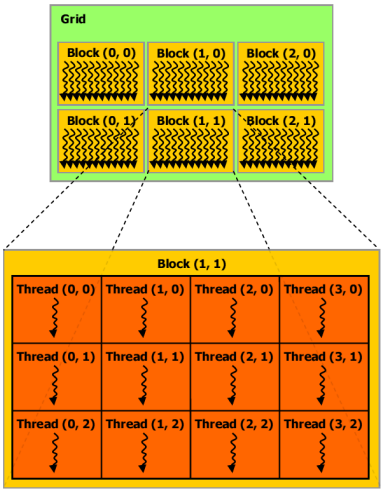


Figure 4.10: Grid of Thread Blocks[45]

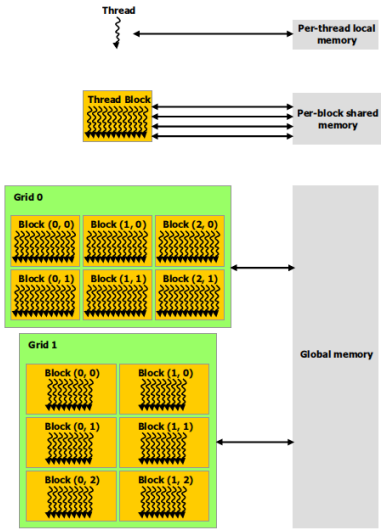


Figure 4.11: Memory Hierarchy[45]

4.3.4 RG-RRT on CUDA

In this project, we will focus more on two small procedures that make up the whole RG-RRT algorithm namely, **randomSampling** and **checkObstacle**. These two procedures were chosen as they both with a bit of tweaking can be *massively parallelized*. In this chapter I will highlight some portions of the code that need to be adjusted in order to achieve massively parallelizable RG-RRT.

4.3.4.1 Parallel Random Sampling

Generating random configurations can be done much quicker when it entails parallelism. In the *non-parallelized* version of RG-RRT, a random configuration is sampled each time the *search tree* is expanded incrementally. A code snippet can be seen as follows:

```
//assigning a random value as the x-coordinate
new_point(0) = (col-1)*unifRand();
//assigning a random value as the y-coordinate
new_point(1) = (row-1)*unifRand();

while(!isValid(space.getMap(), new_point))
{
 new_point(0) = (col-1)*unifRand();
 new_point(1) = (row-1)*unifRand();
}
```

However by implementing this routine as above, there will be no space for the algorithm to exploit parallelism. To be able to massively parallelizing the **randomSampling()**, all random configurations will have to be generated at once. By generating them at once, one thread is exclusively responsible in running **unifRand()** until a pair of valid points is successfully found. As it has been discussed in the preceding chapter, a kernel function has to be written in order to run code in the device from the host.

```
// generate random numbers
generate_kernel<<nBlocks, nThreads>>(arg1, arg2, ...)
```

The kernel description above indicates that the kernel will be called nBlocks $*$ nThreads times simulatenously. All sampled random points will then be stored in the global memory so that the host can later access them. The kernel structure will be similar to the *non-parallelized* version of the algorithm, only that now it will be massively parallelized.

```
__global__ void
setup_kernel(arg1, arg2, ...)
{
 // thread's id
 const unsigned int tid = threadIdx.x +
 blockIdx.x * blockDim.x;
 if(tid < len)
 {
  // each thread gets same seed,
  // a different seq number, no offset
  curand_init(..);
 }
}

__global__ void
generate_kernel(arg1, arg2, ...)
{
 // thread's id
 const unsigned int tid = threadIdx.x +
 blockIdx.x * blockDim.x;
 if(tid < len)
 {
  // copy state to register for efficiency
  curandState localState = state[tid];

  float p = curand_uniform(&localState);
  const float RRT_GoalBias = 0.1;

  if(p <= RRT_GoalBias)
  {
    // expanding the tree toward goal configurations
  }
  else
  {
   // expanding the tree to a random configuraton
  }

  // copy state back to global memory
  state[tid] = localstate;
 }
}
```

Generating random number uniformly in the device can be done as easy as in the host with **curand_uniform()**. This function returns a sequence of pseudorandom floats uniformlny distributed between 0.0 and 1.0 [46]. Prior calling **curand_uniform()**, the quasirandom number generator state has to be initialized. The initialization is done by calling **curand_init()**. The only shortcoming of generating random number in the device lies on **curand_init()**. Calls to this function are much slower than generating the random number itself that will eventually degrade the system performance. Different seeds for each thread and a constant sequence number of 0 can be used to tackle the performance issue. Another trick that I have done is to initialize only some of random generator states and call them iteratively as follows:

```
unsigned int multiple = 300;
unsigned int nseeds = len/multiple;

// declare a pointer to curandState
curandStateXORWOW_t * devStates;

// initialization of the random generator
setup_kernel<<<nBlocks, nThreads>>>(devStates, 0
, nseeds);

for(int i=0; i<multiple; ++i)
{
  // generate random numbers
  generate_kernel<<<nBlocks, nThreads>>>(xy_d+nseeds*i
  , nseeds, devStates, ...)
}
```

4.3.4.2 Parallel Collision Detection

Collision detection in the *non-parallelized* version of RG-RRT is done by iteratively checking pixel by pixel until it encounters an obstacle.

```
while i < noElems and valid
{

 if it exceeds the image dimension then
  valid = false

 else if it encounters an obstacle then
  valid = false

  ++i
}
```

To have them parallelized, each thread will be responsible in verifying one pixel. For instance if there are 3000 pixels to be verified, then 3000 threads will also be created. However to make them optimally efficient, not only one arc line will be verified at a time but a few of them. A care has to be taken in determining the number of arc lines as when it becomes too many, it will hamper the *tree* progression. I found that 11 arc lines would be a reasonable amount as I experienced computation inefficiency when it is below 8 and sluggish *tree* progression when it is above 11. These 11 arc lines are then combined into one long arc line that will be verified in the kernel function.

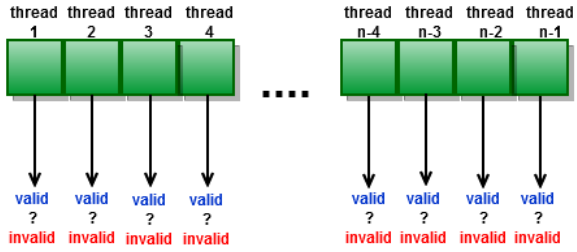


Figure 4.12: Collision Detection

```
__global__ void
checkEdge_kernel(arg1, arg2, ...)
{
 // thread's id
 const unsigned int tid = threadIdx.x +
 blockIdx.x * blockDim.x;
 if(tid < len)
 {
  // store to registers x and y
  float x = xy[tid].x;
  float y = xy[tid].y;
  x = ceil(x);
  y = ceil(y);

  if(y>=nrows || y<0 || x<0 || x>=ncols)
  {
   invalids[tid] = 1;
  }
  if(tex2D(tex, x+0.5f, y+0.5f) == 0)
  {
   invalids[tid] = 1;
  }
 }
}
```

Chapter 5

Results and Discussion

5.1 MATLAB vs. C++

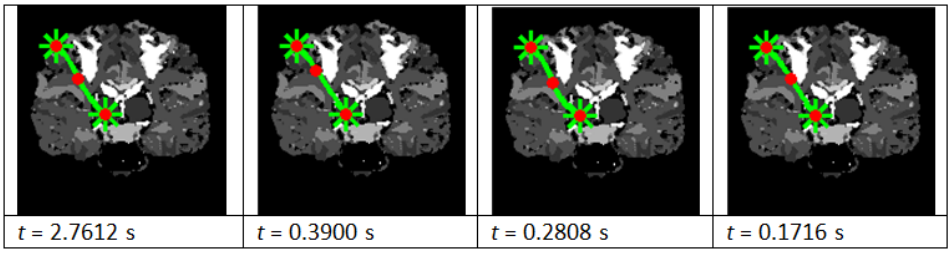


Figure 5.1: RG-RRT in MATLAB

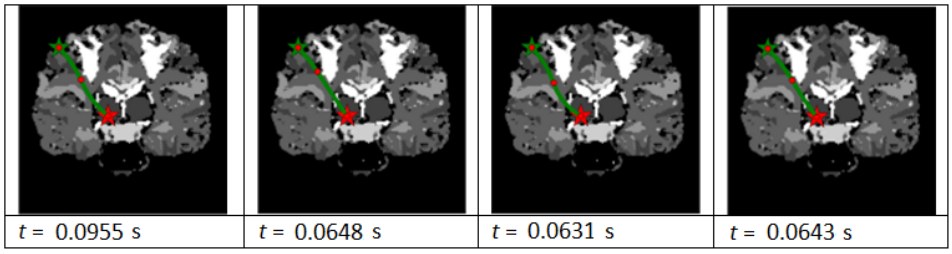


Figure 5.2: RG-RRT in C++ and Python

Fig. 5.1 and Fig. 5.2 are provided to show the performance difference between two distinct implementations. It clearly shows that the hybrid approach (i.e. compiled and interpreted language) between C++ and Python indicates better performance than running the application in MATLAB. To have a fair comparison, the corresponding paths have also been deterministically selected. We could expect the RG-RRT implementation in C++ and Python runs **12 - 13 times** faster than its MATLAB implementation.

5.2 RG-RRT vs. Extended RG-RRT

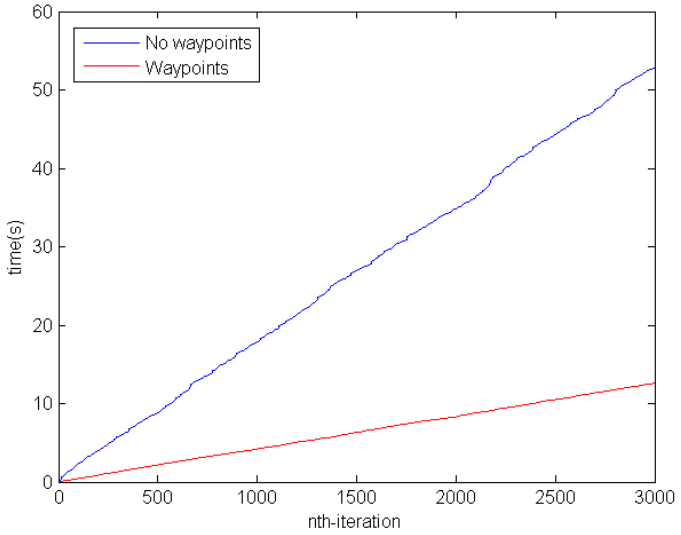


Figure 5.3: The performance with(red) and without(blue) waypoints

Fig. 5.3 clearly shows how RG-RRT with *waypoint cache* might perform compared to the original RG-RRT. As the algorithm always starts from having only an initial configuration, both of the first images naturally will show only difference that is caused by its stochastic nature. Both of them attempted to explore the tree stochastically without having any guidance. Starting from the second image, RG-RRT with *waypoint cache* shows quite a significant difference comparing the basic RG-RRT as it receives "hints" from the first iteration solution. However this will not always be the case. If the first iteration fails the converge, neither performs well as waypoints cannot help by offering "hints" from previous solutions.

I ran 3000 iterations to get a clear idea of how ERG-RRT might yield a better result compared to its counterpart, RG-RRT. As shown in Fig. 5.4, the first 5 iterations does not indicate the superiority of ERG-RRT. This may happen as both algorithms are stochastic or it can also be caused by adding extra code for storing and retrieving the *waypoint cache*. In the first 100 iterations, the original RG-RRT still outperforms the extended version, however this time the performance difference is no longer huge. In the 500 iterations, the efficiency of getting "hints" from the previous plans has increased that the ERG-RRT now outperforms the RG-RRT. The situation has not changed much in the 1000 iterations until in the 3000 iterations where there is a huge gap between the RG-RRT and the ERG-RRT.

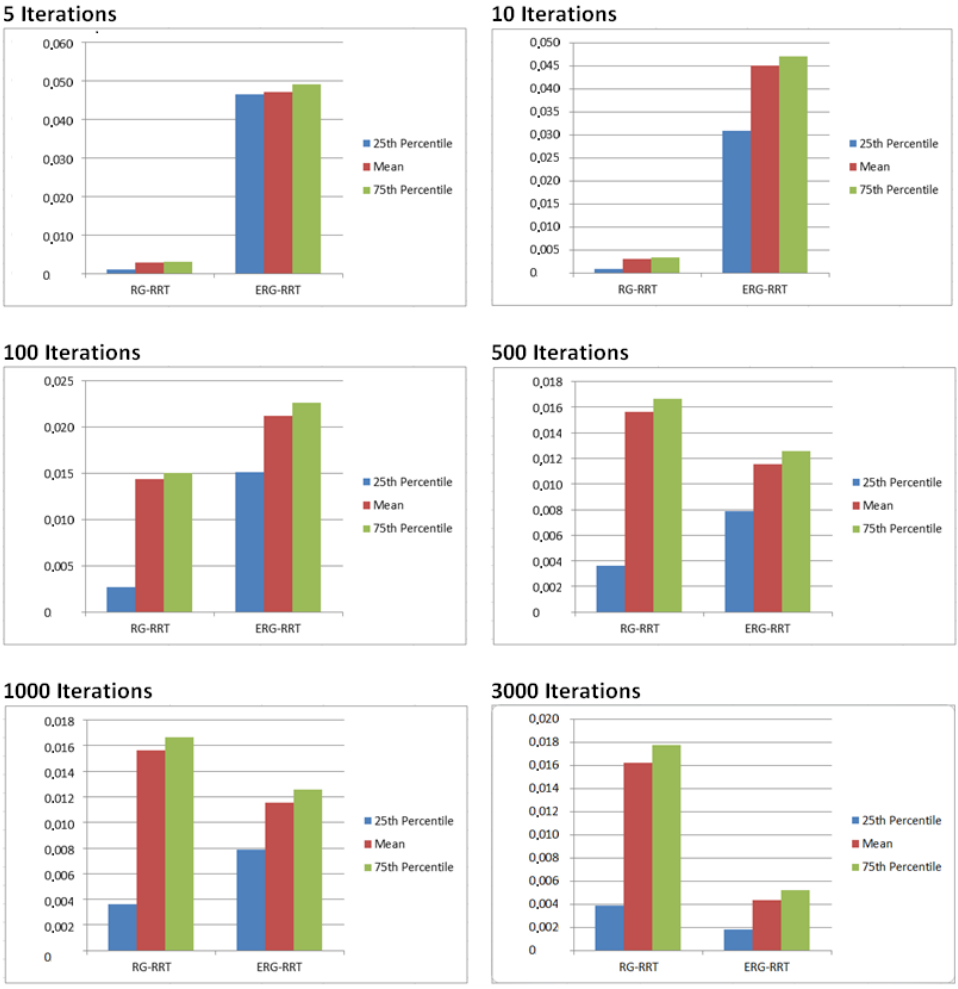


Figure 5.4: 25th percentile, mean, 75th percentile of RG-RRT and ERG-RRT among different iterations

5.3 Comparison and Analysis of GPGPU and Parallel Computing on Multi-Core CPU

5.3.1 Experiment Settings

System settings used in comparing the performance of the CPU implemetation with the GPU implementation of RG-RRT algorithm are as follows:

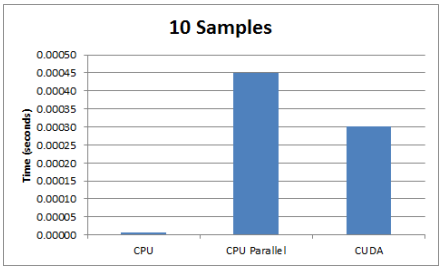
1. CPU: Intel Core i7-2630QM, 2.0GHz
2. Memory: 6GB
3. GPU: NVIDIA GeForce GT 540M, CUDA version is 2.1
4. System: Windows 7 Premium

Application settings used to effectively explore the configuration space are as follows:

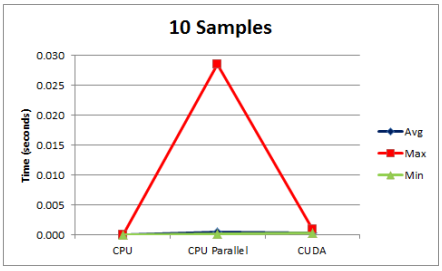
1. Input: 1530 x 1530 pixels pre-operative image
2. Number of random numbers: 100.000

5.3.2 Sampling Performance Comparison

In this section, some statistics on sampling performance are given. The statistics start from a small amount of samples and then gradually increasing to a big amount of samples. Generating 10 samples (Fig. 5.5(a)) seems very inefficient and ineffective compared with its parallelized versions. The inefficiency running in many threads parallely with only small number of iterations originate from the cost of creating threads and the work-sharing between these threads. In addition to CUDA, another cost also comes from the data copy and transfer between the *host* and the *device*. However in the first graph the cost of data copy and transfer is relatively small compared to its high speed processing power that CUDA possesses. This explains why CUDA is still faster than CPU parallel at this point. When performing 100 samples generation, the parallelized versions still take more time than its serialized version, but now CUDA performs a bit worse than CPU parallel. This small jump is caused by data copy and transfer that take more time than before. Generating 1000 samples indicates that parallelized versions perform now much better than running it serially. However still at this phase, CUDA performs a little bit slower that its counter part, CPU parallel. This small difference becomes even less (slightly at the same level) when we increase the number of samples to 10000. When it reaches 100000 samples, the difference between methods now becomes obvious as it is now very expensive to generate samples serially and CUDA has shown quite a significant speedup compared with the other two methods.

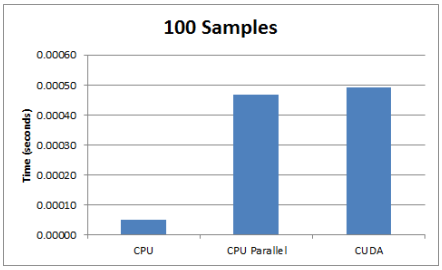


(a) Average execution times

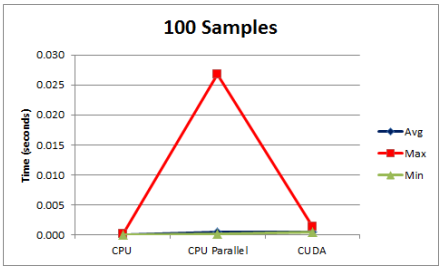


(b) Average, Max, Min execution times

Figure 5.5: 10 Samples Comparison



(a) Average execution times



(b) Average, Max, Min execution times

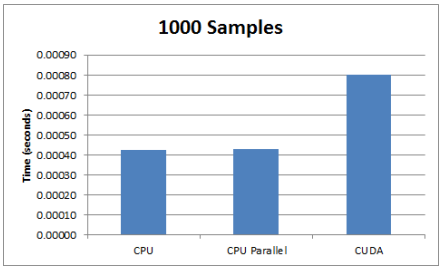
Figure 5.6: 100 Samples Comparison

5.3.2.1 Overall Performance Comparison

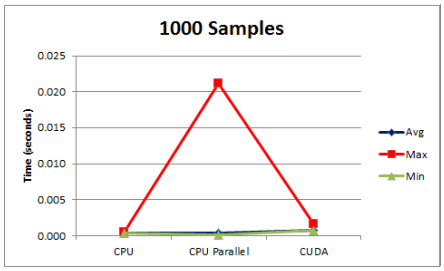
Experimental results in Table 5.1 and visually represented in Fig. 5.10 were performed with the configured experiment settings above and an input brain image with dimension of 1530 x 1530 px. The graph clearly shows that all parallel methods outperform its serial computation.

The first method, OR parallel, as expected has obtained a slightly better result than the original version with only 1.31 times speedup gain. This result is fairly comprehensible as OR parallel will not improve the minimum execution time but in fact it merely helps reducing the maximum time. AND parallel to some extent managed to break the gap even further by obtaining the ratio up to approximately 5 times more than its original version. This result can be explained as in AND parallel, threads work collaboratively with the others to find one single solution. The only main consideration of this method is the time for (explicit or implicit) synchronizations between all the threads. Synchronizations are required as it shares its resources among all threads. Thus data integrity and coherence must be really taken care of.

OR+AND parallel method in the graph performs comparably fast with AND

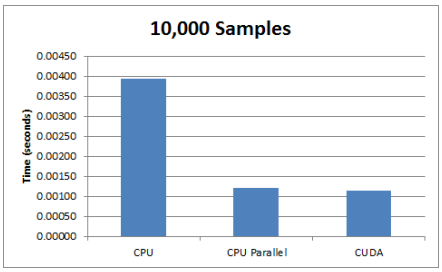


(a) Average execution times

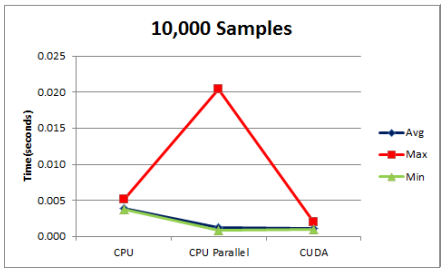


(b) Average, Max, Min execution times

Figure 5.7: 1000 Samples Comparison



(a) Average execution times



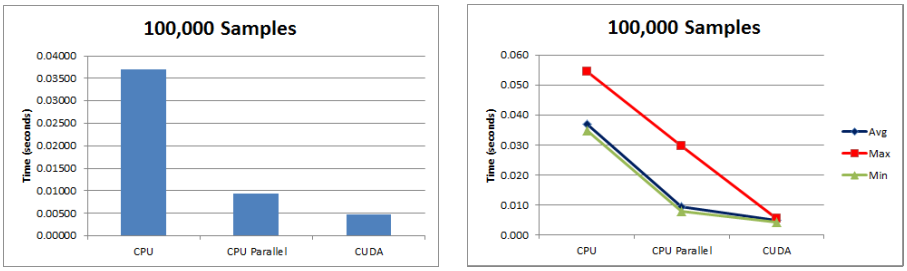
(b) Average, Max, Min execution times

Figure 5.8: 10000 Samples Comparison

parallel hence we cannot decide which one is faster than the other one. Manager-Worker method performs quite poorly compared to AND-based techniques due to idle time it suffers when the *Manager* or the *Worker* has to go through waiting time for data to be processed. The idle time has been primarily a bottleneck in this method especially when there is an unbalanced workload among different processes.

Table 5.1: Average execution times for different RG-RRT methods with 100.000 samples

	Serial	OR	AND	OR+AND	Manager-Worker	CUDA
Avg Time (s)	0.1605	0.1226	0.0323	0.0347	0.0636	0.01764
Speedup	N/A	1.31	4.97	4.63	2.52	9.10
Improvement	N/A	0.2361	0.7988	0.7838	0.6037	0.8901
Efficiency	N/A	0.1636	0.6211	0.5782	0.3154	0.0091



(a) Average execution times (b) Average, Max, Min execution times

Figure 5.9: 100000 Samples Comparison

5.3.3 Discussion

In this section of the report, some experiment results have been given both on the CPU and the GPU parallelization techniques. The evaluation of the parallel execution performance is measured with respect to speedup, performance improvement and efficiency with reference to both sequential and parallel processing time [53].

Speedup measurement indicates how much a parallel algorithm is faster than a corresponding sequential algorithm. The calculation is done based on equation:

```
Speedup =
sequential(time)/parallel(time)
```

The performance improvement however depicts relative improvement of the parallel system over the sequential process. The calculation is done based on equation:

```
Performance Improvement =
(sequential(time) - parallel(time))/sequential(time)
```

The last evaluation measurement is its efficiency. It is used to indicate how well-utilized the processors are in solving the problem, compared to how much effort is wasted in communication and synchronization. The equation for calculating efficiency is:

```
Efficiency =
sequential(time) / (no-procs x parallel(time))
```

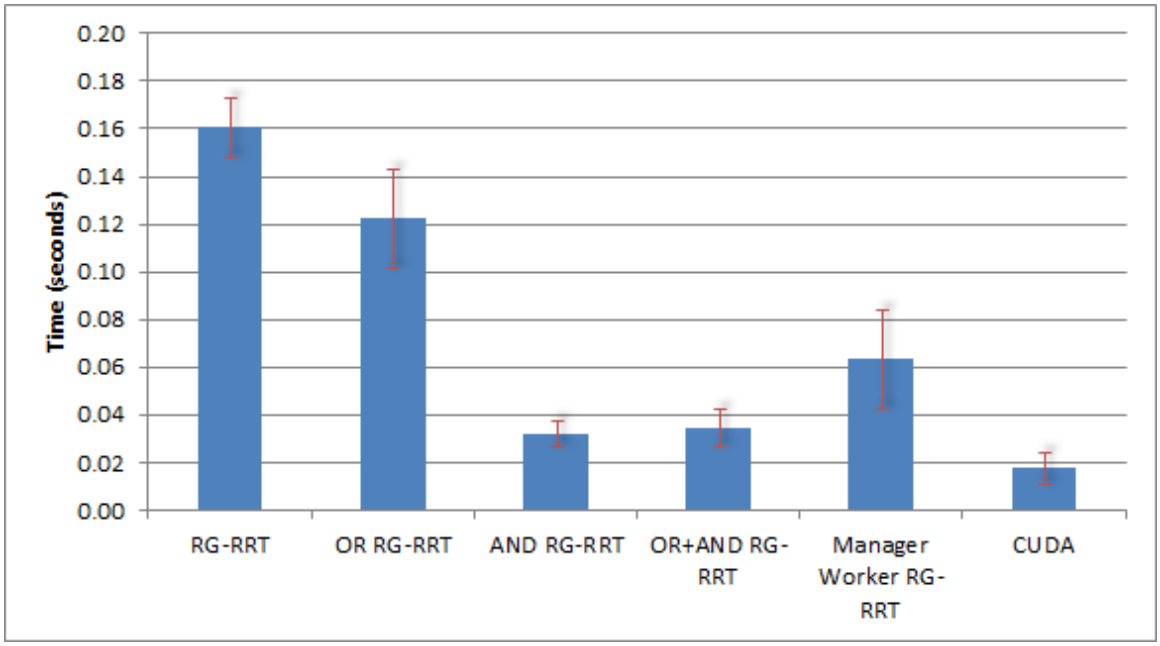


Figure 5.10: Average execution times for different RG-RRT methods

Based on results shown in Table 5.1, massively parallelizing random numbers generation and collision-checking in CUDA enabled-GPUs has proven to give the most significant speedup and performance improvement compared to the other CPU parallel techniques. However, in terms of efficiency, AND parallel performs much better than the GPGPU technique. In fact, all CPU parallelization techniques perform with much greater efficiency than running it in the GPU. This great loss in efficiency is due to the I/O bottleneck that involves *device-host* memory transfer as it was covered in the previous chapter.

I/O problem has always been a bottleneck in GPGPU technique. The problem becomes more obvious when handling small data size. As we could observe in Fig. 5.5(a) - 5.9(a), the GPGPU technique performance is gradually rising with the increase of sample size. Similar to the sample size, the bigger the image size, the more efficient the collision-checking will take place as more data will then have to be processed in the GPU. From this observation, I can conclude that when only dealing with small data size (size $\leq$ 100.000), it is more advisable to use CPU parallel rather than GPGPU technique. This is done to avoid the *device-host* memory transfer overhead that might beat the performance gain. However when data size is bigger than 100.000, parallel computation in the GPUs might give better results as the bottleneck will be relatively small compared to the performance gain.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

Performance issue of RG-RRT path planning of the neurosurgical flexible probe has been the main focus in this thesis. It mainly covers the implementation consideration of the algorithm until the parallelism exploitation both in CPU and in GPU. The RG-RRT tree exploration was done entirely in C++ and the image related issues ranging from displaying until processing the input image has been thoroughly done in Python. Implementation in C++ and Python has proven to be faster and much more efficient than running it in MATLAB. The perfomance speedup that it has successfully gained is around 12 - 13 faster than its MATLAB version. This significant performance gain was later brought even futher by implementing some parallelization techniques both in CPU and in GPU.

In total there were 5 parallelization techniques covered in this graduation project, 4 methods were done in CPU multi-core architecture and 1 method was done in CUDA-enabled GPU. Those 4 methods that were performed in CPU were OR parallel, AND parallel, OR+AND parallel and Manager-Worker. Among these 4 methods, AND parallel has shown the most significant result that could run up to 5 times faster than the original C++ and Python version. Based on some further experiments, results have indicated that when the algorithm was run in CUDA-enabled GPU, it could even reach up to 10 times faster than its original version. Thus by implementing the RG-RRT in C++ and then parallelizing it with CUDA, it could reach approximately 120 times faster than the original MATLAB version. It brings us finally to a conclusion that RG-RRT is a parallelizable algorithm and can be applied for massive parallelism implementation in the GPU despite its low efficiency compared to the CPU parallelization techniques.

6.2 Future Work

Although the primary goal of this project which was increasing the RG-RRT path planning efficiency has been successfully accomplished yet there are still many rooms of improvement. Currently steering the flexible proble is only limited to the 2D CT/MRI brain images, allowing it to explore in the 3D configuration space would be very useful because 3D view can be very detailed as it stores more information than 2D thus providing much more accuracy. In the current implementation, the focus is merely the rapid exploration of RG-RRT and not finding the most optimal path over a roadmap. Hence an exploration to other algorithms that guarantee path optimality such as RRT* and RRM could further be done for the use of neurosugical flexible probe. Last but not least, increasing the efficiency and effectiveness of the algorithm can be considered as the first step toward real-time RG-RRT planning. Some algorithms that provide fast yet accurate replanning could also be explored.

Bibliography

[1] *eurosh Center for Minimally Invasive Endoscopic*. Available at: `http://brainsurgeryprocedures.com` (Accessed: 03 July 2012)

[2] *UHS : Minimally Invasive Surgery*. Available at: `http://www.uhs.net/minimally-invasive-surgery` (Accessed: 03 July 2012)

[3] *Minimally Invasive Surgery*. Available at: `http://www.patient.co.uk/doctor/Minimally-Invasive-Surgery.htm` (Accessed: 03 July 2012)

[4] S. P. DiMaio, S.Pieper, K.Chinzei, G.Fichtinger, and R. Kikinis, "Robot-assisted Percutaneous Intervention in Open-MRI" in *5th Interventional MRI Symposium*, 2004

[5] Webster R, "Design and mechanics of continuum robots for Surgery", John Hopkins University, 2007

[6] Webster RJ, Memisevic J, Okamura AM, "Design considerations for robotic needle steering" in *IEEE international conference on robotics and automation*, Barcelona, Spain, Apr Washington DC, pp 3588-3594

[7] Misra S, Reed KB, Douglas AS, Ramesh KT, Okamura AM, "Needle-tissue interaction forces for bevel-tip steerable needles" in *2nd IEEE RAS & EMBS international conference on biomedical robotics and biomechatronics, BioRob*, 2008, pp 224-231

[8] Misra S, Reed KB, Douglas AS, Ramesh KT, Okamura AM, "Observations and models for needle-tissue interactions" in *IEEE international conference on robotics and automation, Kobe international conference center*, pp 2687-2692

[9] Minhas D, Engh JA, Fenske MM, Riviere C, "Modeling of needle steering via duty-cycled spinning" in *Proceedings of the 29th annual international conference of the IEEE engineering in medicine and biology society*

[10] Dupont PE, Lock J, Itkowitz B, Butler E, "Design and control of concentric-tube robots" in *IEEE Trans Robot*, 26(2):209-225

[11] L. Frasson, S. Y. Ko, A. Turner, T. Parittotokkaporn, J. F. Vincent, and F. R. y Baena, "Sting: a soft-tissue intervention and neurosurgical guide to access deep brain lesions through curved trajectories", *Proceedings of the Institution of Mechanical Engineers, Part H: Journal of Engineering in Medicine*, vol. vol. 224, pp.775-788, 2010

[12] R. J. Webster, III, J. S. Kim, N. J. Cowan, G. S. Chirikjian, and A. M. Okamura, "Nonholonomic modeling of needle steering" in *Int. J. Robot. Res.*, vol. 25, no. 5-6, pp. 509-525, 2006

[13] Kyle B. Reed, Ann Majewicz, Vinutha Kallem, Ron Alterovitz, Ken Golberg, Noah J. Cowan, and Allison M. Okamura, "Robot-Assisted Needle Steering" in *IEEE Robotics & Automation*, December 2011

[14] Quicke, D. L. J, "Ovipositor mechanics of the braconine wasp genus Zaglyptogastra and the ichneumonid genus Pristomerus", *J. Nat. Hist.*, 1991, 25, 971-977

[15] Quicke, D. L. J. and Fitton, M. G, "Ovipositor steering mechanisms in parasitic wasps of the families Gasteruptiidae and Aulacidae (Hymenoptera),*Proc. R. Soc. Lond., B*, 1995, 261, 99-103

[16] Luca Frasson, Francesco Ferroni, Seong Young Ko, Gorkem Dogangil and Ferdinando Rodriguez y Baena, "Experimental evaluation of a novel steerable probe with a programmable bevel tip inspired by nature", 2012, Pages:1-9, ISSN:1863-2483

[17] S. Ko, B. Davies, F. Rodriguez y Baena, *et al.*, "Two-dimensional needle steering with a programmable bevel inspired by nature: Modeling preliminaries" in *Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference*, pp. 2319-2324, IEEE

[18] Seong Young Ko, Luca Frasson, Ferdinando Rodriguez y Baena, "Closed-Loop Planar Motion Control of a Steerable Probe With a Programmable Bevel Inspired by Nature" in *Robotics, IEEE Transactions*, 2011, pp. 970-983

[19] S. M. LaValle, "Planning Algorithms". Cambridge, U.K.: Cambridge University Press, 2006. Available at: `http://planning.cs.uiuc.edu/`

[20] Hani Safadi, *McGill University School of Computer Science : Local Path Planning Using Virtual Potential Field*. Available at: `http://www.cs.mcgill.ca/~hsafad/robotics/index.html` (Accessed: 10 July 2012)

[21] Michael A. Goodrich, *Potential Fields Tutorial*. Available at: `http://borg.cc.gatech.edu/ipr/files/goodrich_potential_fields.pdf` (Accessed: 11 July 2012)

[22] Y.Koren, J. Borenstein, "Potential Field Methods and Their Inherent Limitations for Mobile Robot Navigation" in *Proceedings of the IEEE Conference on Robotics and Automation*, Sacramento, California, April 7-12, 1991, pp. 1398-1404

[23] Eric Roberts, "Motion Planning in Robotics" : Stanford University. Available at: `http://www-cs-faculty.stanford.edu/~eroberts/courses/soco/projects/1998-99/robotics/basicmotion.html` (Accessed: 11 July 2012)

[24] "Introduction to Voronoi Diagrams". Available at: `http://dasl.mem.drexel.edu/Hing/VoronoiTutorial.htm` (Accessed: 11 July 2012)

[25] *Kavraki Lab, Physical and Biological Computing.* Available at: `http://www.kavrakilab.org/robotics/prm.html` (Accessed: 12 July 2012)

[26] James Kuffner, *The RRT Page.* Available at: `http://msl.cs.uiuc.edu/rrt/` (Accessed: 12 July 2012)

[27] Howie Choset, *Robotic Motion Planning: RRT's.* Available at: `http://www.cs.cmu.edu/~motionplanning/lecture/lec20.pdf` (Accessed: 12 July 2012)

[28] James J. Kuffner, Jr, Steven M. LaValle, "RRT-Connect: An Efficient Approach to Single-Query Path Planning" in *Int'l Conf. on Robotics and Automation (ICRA 2000)*, IEEE

[29] P. Cheng, "Sampling-based motion planning with differential constraints," Ph.D. dissertation, University of Illinois, Urbana-Champaign, Urbana, IL, August 2005

[30] J. Bruce, M. Veloso, "Real-Time Randomized Path Planning for Robot Navigation" in *IEEE international conference on intelligence robots and systems*, 2002, pp 2383-2388 vol.3

[31] "Information systems management" IT essays. Available at: `http://www.raulvm.com/information-systems.php/2010/12/09/interpreted-vs-compiled-languages`

[32] *Armadillo C++ linear algebra library.* Available at: `http://arma.sourceforge.net/` (Accessed: 21 July 2012)

[33] S. M. LaValle and J. J. Kuffner, "Rapidly-exploring random trees: progress and prospects," in *Algorithmic and Computational Robotics: New Directions.* A K Peters, 2001, pp. 293-308

[34] _____, "Randomized kinodynamic planning" in *Int. J. Robot. Research*, vol. 20, no. 5, 2001

[35] J. Cortes and T., "Sampling-baesd motion planning under kinematic loop-closure constraints" in *Algorithmic Foundations of Robotics VI.*, Springer-Verlag, 2005, pp. 75-90

[36] P.Cheng, E.Frazzoli, and S. M. LaValle, "Improving the performance of sampling-based planners by using a symmetry-exploiting gap reduction algorithm" in *Proc. IEEE ICRA*, 2004

[37] L. Jaillet, A. Yershova, S. M. LaValle, and T. Simon, "Adaptive tuning of the sampling domain for dynamic-domain RRTs" in *Proc. IEEE/RSJ Int. Conf. Intell. Robot. Syst.*, 2005

[38] A. Yershova and S. M. LaValle, "Improving motion planning algorithms by efficient nearest-neighbor searching" in *IEEE Trans. Robot.*, vol. 23, no. 1, 2007

[39] Blaise Barney, *Lawrence Livermore National Laboratory : Introduction to Parallel Computing*. Available at: `https://computing.llnl.gov/tutorials/parallel_comp` (Accessed: 22 July 2012)

[40] `http://www.wikipedia.org` (Accessed: 22 July 2012)

[41] *Electrical Engineering and Computer Sciences Department, University of California Berkeley : Parallel Programming for Multicore*. Available at: `http://www.cs.berkeley.edu/~yelick/cs194f07/` (Accessed: 23 July 2012)

[42] Iker Aguinaga, Diego Borro, Luis Matey, "Parallel RRT-based path planning for selective disassembly planning" in *Int J Adv Manuf Technol*, (2008) 36:1221-1233 DOI 10.1007/s00170-007-0930-2

[43] `http://openmp.org/wp/` (Accessed: 25 July 2012)

[44] Didier Devaurs, Thierry Simon and Juan Corts, "Parallelizing RRT on Distributed-Memory Architectures" in *IEEE International Conference on Robotics and Automation*, May 9-13, 2011, Shanghai, China

[45] NVIDIA CUDA^{TM}: NVIDIA CUDA C Programming Guide, Version 4.2

[46] CUDA Toolkit 4.2: CURAND Guide

[47] C. Caborni, *Curvilinear Path Planning for a Steerable Flexible Neurosurgical Probe*. Politecnico di Milano, A.A, MSc Thesis 2010-11

[48] S. Bano, *Continuous Curvature Path Planning for a Neurosurgical Flexible Probe*. Erasmus Mundus in Vision and Robotics (VIBOT), MSc Thesis 2011

[49] S. Carpin and E. Pagello, "On parallel RRTs for multi-robot systems," in *Proc. 8th Conf. Italian Association for Artificial Intelligence*, Pisa, Italy, Sep 2003

[50] S. Sengupta, "A parallel randomized path planner for robot navigation," *International Journal of Advanced Robotic Systems*, vol. 3, pp. 256-266, Sep 2006

[51] L. E. Kavraki, P. Svestka, J. C. Latombe, and M. H. Overmars, "Probabilistic roadmaps for path planning in high-dimensional configuration spaces," *IEEE Transactions on Robotics and Automation*, vol. 12, no. 4, pp. 566-580, 1996

[52] J. Bialkowski, S. Karaman, and E. Frazzoli, "Massively parallelizing the *RRT* and the *RRT**," *IROS, 2011 IEEE/RSJ International Conference*, pp. 3513-3518, 2011

[53] N. Haron, R. Ami, I. A.Aziz, L. T. Jung and S. R. Shukri, Parallelization of Edge Detection Algorithm using MPI on Beowulf Cluster. *Innovations in Computing Sciences and Software Engineering.* 2010