KTH Numerical Analysis
and Computer Science

Autonomous Path Planning and Real-Time Control – a Solution to the Narrow Passage Problem for Path Planners and an Evaluation of Real-Time Linux Derivatives for Use in Robotic Control

Daniel Aarno

TRITA-NA-E04006

NADA

Numerisk analys och datalogi
KTH
100 44 Stockholm

Department of Numerical Analysis
and Computer Science
Royal Institute of Technology
SE-100 44 Stockholm, Sweden

Autonomous Path Planning and Real-Time Control – a Solution to the Narrow Passage Problem for Path Planners and an Evaluation of Real-Time Linux Derivatives for Use in Robotic Control

Daniel Aarno

TRITA-NA-E04006

Master's Thesis in Computer Science (20 credits)
at the School of Electrical Engineering,
Royal Institute of Technology year 2004
Supervisor at Nada was Henrik Christensen
Examiner was Henrik Christensen

Abstract

This thesis consists of two parts. The first part is concerned with the *narrow passage problem* of path planners and the second part is concerned with the *evaluation of real-time Linux derivatives* in the context of robotic control.

The first part presents a new variation of the *probabilistic roadmap method* (PRM). The new planner is called artificial potential biased PRM (APBPRM) and uses an artificial potential function to bias the distribution of nodes in order to increase the node density in difficult regions of the configuration space. Further the planner presented in this thesis uses a lazy evaluation scheme that makes it suitable for single query path planning problems.

The second part presents a *study and comparison of three real-time Linux derivatives*, RT-Linux, RTAI and KURT. The operating systems are evaluated for their suitability as a robotic control system in an academic environment. The evaluation considers real-time performance (scheduling jitter), ease of installation and use, and the richness of the associated API. Finally some preliminary results of a control system implemented on RTAI and used to control a Puma 560 robotic manipulator are presented.

Autonom vägplanering och realtidsstyrning

En lösning till problemet med trånga passager för vägplanerare och en studie av realtidsoperativsystem, som bygger på Linux, för realtidsstyrning

Sammanfattning

Den första delen behandlar de problem som uppstår då en automatisk vägplanerare (eng. path planner) ska ta sig igenom trånga passager. Den andra delen utvärderar lämpligheten för styrning av robotar hos realtidsoperativsystem som bygger på Linux.

Den första delen handlar om en ny variant av den sk *probabilistic roadmap method* (PRM), en vanlig metod som används för automatisk vägplannering. Denna nya variant kallas för artificial potential biased PRM (APBPRM) och använder en artificiell potentialfunktion för att vikta distributionen av noder så att nodtätheten ökar i svåra områden av konfigurationsrymden. Planeraren som beskrivs i den här rapporten använder en precis-i-tid (eng. just in time, lazy) metod för att kontrollera kollisioner längs vägarna, vilket gör den lämplig för enstaka vägplaneringsproblem, dvs problem där omgivningen ofta ändras.

Den andra delen är en *studie av, och jämförelse mellan tre realtidsoperativsystem som bygger på Linux*, RT-Linux, RTAI och KURT. Operativsystemen utvärderas för deras lämplighet att användas för att styra robotar i en universitetsmiljö. Utvärderingen inehåller aspekter så som realtidsprestanda (schemaläggningsfluktuationer), installations- och användarvänlighet, samt innehållet i medföljande API. Slutligen ges några preliminära resultat för ett reglersystem implementerat under RTAI. Reglersystemet används för att styra en Puma 560 robotarm.

Foreword

This thesis is part of a M. Sc. project in computer science at the school of Electrical Engineering, Royal Institute of Technology. The M. Sc. project was carried out at the Center for Autonomous Systems (CAS) during 2002-2003. CAS is a research center that is a part of NADA at the Royal Institute of Technology (KTH) in Stockholm, Sweden. The center does research in (semi-) autonomous systems including mobile robot systems for manufacturing and domestic applications.

The original goal of this project was to evaluate real-time Linux derivatives for use in robotic control and investigate the possibility of using the reinforcement learning (RL) paradigm to perform path planning. While doing research on path planning I was intrigued by the apparent success of the Lazy PRM planner described in the excellent work done by Robert Bohlin [1], which I can recommend anyone interested in the subject to read. Also I realized that PRM planners, such as that described by Bohlin, would experience difficulties if the configuration space contained narrow passages. Feeling that helping to solve the narrow passage problem of PRM planners would be a better contribution to the field of robotics than a new way of doing path planning, I modified the topic of the project to include a study of the narrow passage problem with PRM planners. Because of the time limits associated with this project I had to abandon the investigation of using the RL paradigm to do path planning.

Contents

List of Figures

Chapter 1

Introduction and Motivation

Robotic appliances are gradually becoming a part of our everyday lives. It can be envisaged that, besides providing services such as assistance to elderly and disabled people, there will come a time when robotic appliances are a general utility to humans both at the workplace and in their homes. If these systems are to operate in a complex and unpredictable environment, such as a domestic or office one, it is crucial that they can perform *motion planning* in these environments. In order to perform motion planning it is important to have a robust an reliable underlying *control system* as well as a capable *path planner*. This project includes a study of *real-time Linux derivatives* for use as a manipulator control system, as well as the construction of a *path planner* targeted at working in *complex environments containing narrow passages*.

1.1 Robot Path Planning

Autonomous path planning addresses the problem of finding collision-free paths for moving objects (robots) among obstacles [1]. The path planning problem has been proved to be a hard one and although complete algorithms exists their high complexity precludes any useful applications [2]. This result has led to the development of heuristic algorithms.

Probabilistic roadmap methods[1] (PRMs) have been successfully used to solve difficult path planning problems, but *their efficiency is disappointing when the free space contains narrow passages* [3]. This thesis presents a new sampling scheme, that aims to increase the probability of finding paths through narrow passages. This new scheme could be useful for planning in both industrial environments and for service robots operating in a domestic environment.

[1] See section 2.3.1

1.1.1 Planner Requirements

The time required for planning should be related to the difficulty of the planning task, i.e. a simple path in an uncluttered environment should be found quickly while a difficult path in an more complicated environment may require more time [1].

Likewise, the planning time should be related to the desired *quality* of the solution. The quality of the solution is difficult to quantify (see [1] for further discussion), but in general short paths in configuration space are preferred.

There is an obvious trade-off between planning time and solution quality, and ideally the planner should be easily tuned to a "goodness" measure allowing it to function in different situations where the planning requirements are different. Consider a welding robot that will perform the same operation over and over again for say a year. It is clear that the quality of the solution is the most important thing here since the same solution will be repeated many times. However, if the task is for a robotic arm to help a disabled person pick up an object in a changing environment the planning time is most important. This is because it is not necessary to find the best solution as long as the problem is solved the planner has performed well.

1.2 Real-Time Control

Control of a robot manipulator requires real-time capabilities of the underlying operating system to ensure adequate control performance in terms of safety and trajectory tracking. This is typically achieved using proprietary operating systems such as QNX, OS9 or VxWorks. A disadvantage of such operating systems is that the underlying code in general is not publicly distributed which is a significant obstacle for academic research. Consequently, it is of interest to determine if open-source[2] operating systems available under open-source licenses, such as GPL or LGPL, can be used for the control of a manipulator.

This project contains a study of the characteristics of real-time Linux derivatives for control of a robot manipulator. The evaluation of potential systems considers real-time performance, ease of installation and use, and the richness of the associated API.

1.3 Outline of This Thesis

This thesis consists of two parts, that can be read independently. The first part deals with the narrow passage problem of some path planners (PRMs in particular) and the second part deals with the evaluation of real-time Linux derivatives for the purpose of manipulator control.

Chapter 2 defines the path planning problem and introduces two well known heuristic methods, the probabilistic roadmap method (PRM) (section 2.3), and a method that uses gradient descent on artificial potential functions (section 2.4).

[2]Open Source Initiative (OSI) http://www.opensource.org

Chapter 3 introduces a new probabilistic roadmap method, using a biased sampling scheme to deal with the narrow passage problem.

Chapters 4 and 5 deal with the implementation and evaluation of the planner described in chapter 3.

Chapter 6 provides a summary of results and conclusions for the first part of this thesis as well as some suggestions for future work.

Chapter 7 describes the fundamental differences between real-time operating systems and "normal" time-sharing operating systems (such as UNIX and Linux). Further it discusses methods for evaluating a real-time operating system.

Chapter 8 describes the architectures of the three operating systems that are evaluated (RT-Linux, RTAI and KURT).

Chapter 9 describes the implementation of the tests performed as well as a discussion about the suitability of the three different real-time Linux derivatives. Finally a brief description of the control system used to control a PUMA 560 robotic arm is given.

Chapter 10 provides a summary of results and conclusions for the second part of this thesis.

Part I

Artificial Potential Biased Probabilistic Roadmap Method - APBPRM

Chapter 2

Introduction to Robot Path Planning

Probabilistic roadmap methods (PRMs) (section 2.3.1) have been successfully used to solve difficult path planning problems, but *their efficiency is disappointing when the free space contains narrow passages* [3]. This thesis presents a new sampling scheme, that aim to increase the probability of finding paths through narrow passages. A *biased sampling scheme* is used to increase the distribution of nodes (milestones) in narrow regions of the free space. A partial computation of the *artificial potential field* (section 2.4) is used to bias the distribution of milestones.

2.1 The Path Planning Problem

This section will introduce the reader to the path planning problem and define some basic concepts, for a more in depth introduction to path planning see [4].

Consider a robot that is some kind of versatile mechanical device - for example, a wheeled or legged vehicle, a robotic (manipulator) arm or a robotic hand. This robot operates in (a subset of) the physical world. The subset of the physical world that some point on the robot can reach is known as the robot's *workspace*. This workspace may be populated by physical objects, such as chairs or automobiles, and is subject to the laws of physics. The robot can perform tasks by executing motions in the workspace and (possibly) interact with other objects.

Next consider a robot platform that can move freely in a closed space, e.g. a room, and is equipped with a manipulator (e.g. a robotic arm) that has the ability to pick up and release objects. One use for this robot could be a retrieve/deliver task, i.e. the robot is given a command to go somewhere, pickup an object and deliver it somewhere else. To accomplish this task the robot has to execute the following subtasks:

- Move from the initial location a to a location b where the object o is reachable by the manipulator's grasping device. While moving from a to b the robot may not collide with any obstacles present in the room.

- Once location b is reached the robot must change the configuration of the manipulator in order to place the gripper in suitable position for grasping o. While the manipulator is moving to the correct configuration no part of the manipulator must collide with any obstacles in the room.
- Now the object o must be grasped and held by the manipulator.
- Next the robot must move from b to a position c in the vicinity of where o should be delivered. This must also occur without hitting any obstacles.
- Now the manipulator must move to a new configuration in order to place o at c and finally the gripper can release o.

To accomplish the subtasks mentioned above the robot has to be able to, among other things, plan collision free paths for both the mobile platform and the manipulator in order to position the grasping device. Constructing such paths is the job of a *path planner*. Throughout the rest of this thesis it is assumed that the world accessible to the robot is completely known, thus greatly simplifying the problem of planning a path in a real environment (i.e. the world).

2.1.1 Mathematical Description of the Path Planning Problem

Let A denote an arbitrary robot, called an *agent*, consisting of one or more rigid bodies resulting in N degrees of freedom (dof) with a frame (coordinate-system) F_T attached to some specific point of A. The frame F_T, known as the *tool-frame,* is a local coordinate-system and it is fixed with respect to some point of A. Denote the workspace of A by $W \subset \mathbb{R}^3$, A is then said to *operate* in W. Further, attach to W a coordinate-system, or frame, F_W known as the *world frame*. The frame F_W is considered to be a fixed global reference frame.

One way of representing a configuration of A is by specifying a closed and bounded region $\Omega \subset W$ that contains A. Another way of representing a configuration of an agent with N dof is by assigning a scalar value to each dof (for instance the angle of a joint), resulting in a point $\mathbf{q} \in \mathbb{R}^N$. The subset of the N-fold product space $\mathbb{R}^N$ for which all kinematic (and other) constraints on the agent are met is called the *configuration space* of the agent [1].

Examine the simple manipulator in figure 2.1, it has two dof, the direction and the length of the manipulator. A specific configuration of the manipulator can be described in W by a closed and bounded region $\Omega \subset W$ that contains the manipulator, i.e. the image of the manipulator in W. An other way of representing a configuration of the manipulator is as a point $\mathbf{q}^c$ in the manipulators configuration space, or C-space. The configuration space C and workspace W of the manipulator in figure 2.1 are shown in figure 2.2.

If the geometry of the rigid bodies that make up A is known it is usually quite easy to calculate the image of A in W given a point $\mathbf{q}^c \in C$, this is denoted by $\mathbf{q}^c \curvearrowright \Omega \subset W$. A mapping from an image $\Omega \in W$ to a point in C is usually much more cumbersome. Thus from here on the configuration of A is considered to be

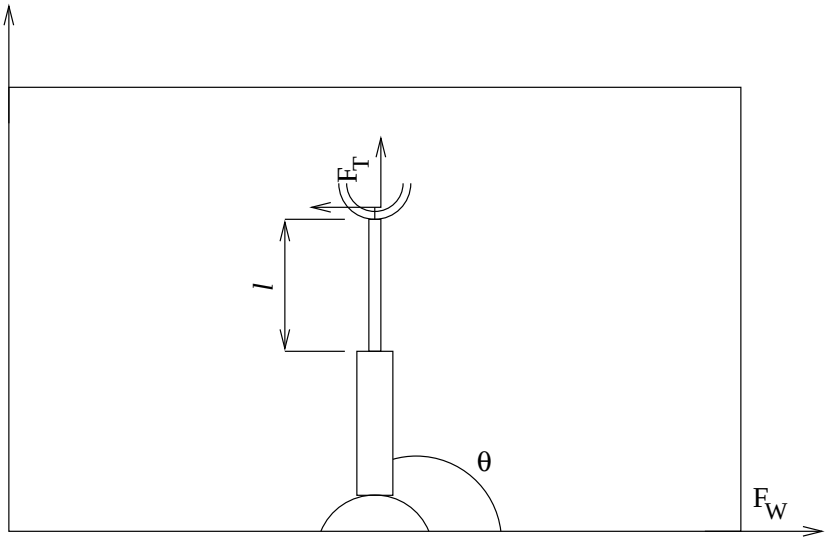


Figure 2.1. A simple 2 dof manipulator.

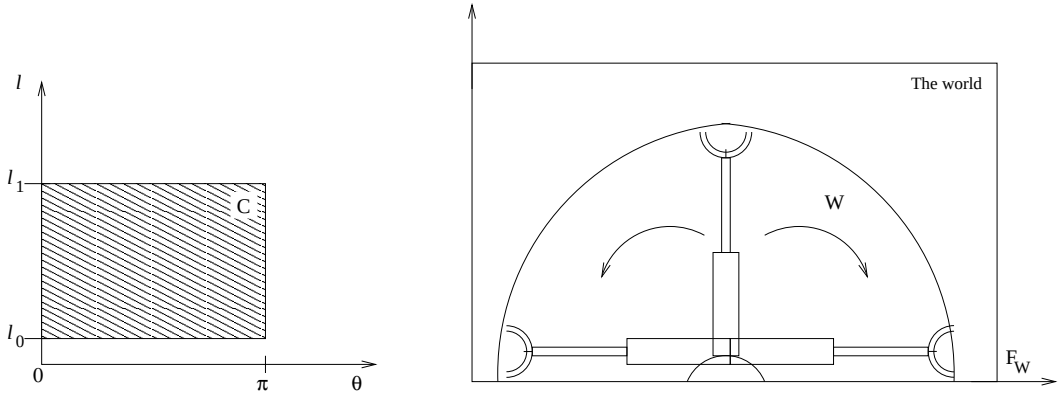


Figure 2.2. Left image shows the configuration space of the manipulator in figure 2.1. Right image shows the workspace of the manipulator in figure 2.1

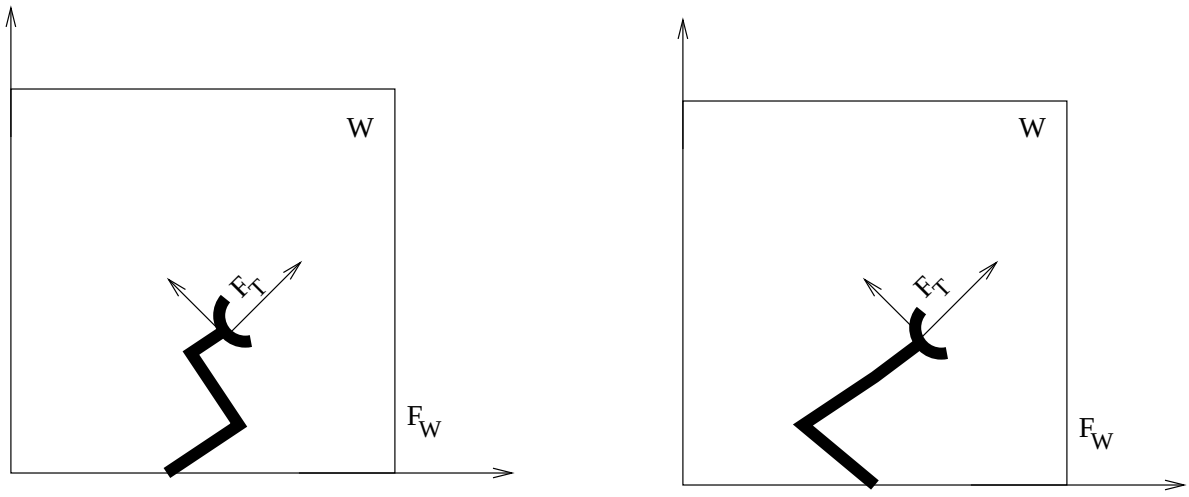


Figure 2.3. Two different configurations that place F_T (the gripper) in the same position with the same orientation relative F_W.

represented by a point in C. From here on it is assumed that *a goal configuration of A is given by the location and orientation of F_T relative F_W*, ignoring the actual configuration of the rigid bodies that make up A. Why this assumption is valid will be described later.

For every point $\mathbf{q}^c \in C$ there exists a unique mapping $\mathbf{q}^c \rightarrow \mathbf{q}^w \in \mathbb{R}^6$ that gives the location and orientation of F_T relative to F_W. This mapping is called the *forward-kinematics* (FK) of A and there exists a unique solution for all points in C for any physical agent. The inverse mapping $\mathbf{q}^w \rightarrow \mathbf{q}^c \in C$ is called the *inverse-kinematics* (IK) and is in general not trivial. In general, unique solutions to the IK mapping do not exist and the solutions may give rise to singularities. From here on it is assumed that the FK of A is known and the IK is not.

Consider the manipulator in figure 2.3 and note that the gripper is positioned in the same way in both the left and right figures. If the manipulator's task was to pick up an object located at the origin of the coordinate-system denoted by F_T both configurations would be acceptable. This is often the case in path planning, a frame F_T is attached to some specific point on the agent and the task of the path planner is to put F_T at a specific location and in a specific orientation relative to the a fixed frame F_W in the world. Figure 2.3 shows the relation between F_T and F_W. Throughout this thesis a goal configuration in W, denoted $\mathbf{q}^w_{goal}$, is a vector specifying the position and orientation of F_T relative to F_W, thus $\dim(\mathbf{q}^w_{goal}) \leq 6$. For an agent consisting of a single rigid body a configuration $\mathbf{q}^w$ in W completely specifies the closed and bounded region occupied by the agent in the workspace W.

Let $W_{free} \subset W$ denote the subset of W in which the agent may place a part of its body, i.e. the part of W not occupied by obstacles. The task of the path planner is to find a feasible and unobstructed path in C such that agent moves from a given

configuration $\mathbf{q}^c_{start}$ in C to a goal configuration $\mathbf{q}^c_{goal} \rightarrow \mathbf{q}^w_{goal} \in W$. With the definition of configuration space given above the whole of C may not be connected. Since the agent can only move between points in C that are connected the agent is limited to the subset $C_{con} \subset C$ that is the connected component containing $\mathbf{q}^c_{start}$ (see [5] for a definition of connected and connected component). Finally let C_{free} be the subset of C_{con} that maps all of its interior to W_{free}, that is $C_{free} \subset C_{con}$ and $\mathbf{q}_i \curvearrowright \Omega_i \subset W_{free}$ for all $\mathbf{q}_i \in C_{free}$.

Thus using the notation in this section the path planning problem is defined as follows:

Definition: The **path planning problem** is to connect, by a continuous path, a point $\mathbf{q}^c_{start} \in C_{free}$ to any point $\mathbf{q}^c_{goal} \in C_{free}$ that satisfies the condition: $\mathbf{q}^c_{goal} \rightarrow \mathbf{q}^w_{goal} \in W_{free}$ under the constraints of the FK of A.

2.2 Robot Path Planning

Several methods exists that can be used for automated path planning. The rest of this chapter describes two methods that have been successfully used to solve difficult path planning problems. The *probabilistic roadmap method* and gradient descent on an *artificial potential function*. Both methods have been successful in solving path planning problems and are directly related to the theory behind the biased sampling scheme presented in this thesis.

2.3 Roadmap Planners

Because the complexity of the path planning problem grows rapidly with the dimensionality of the configuration space, complete methods rapidly becomes useless in practice [2]. A set of planners, usually referred to as *roadmap planners* [1, 3, 6, 7, 8], overcomes this problem by distributing a set of nodes (milestones) in C_{con} and then connect them with the help of a simple local planner to form a roadmap (graph) in C_{con}. The planner then searches this roadmap for feasible paths from $\mathbf{q}^c_{start}$ to $\mathbf{q}^c_{goal}$.

2.3.1 The Probabilistic Roadmap Method - PRM

The probabilistic roadmap method (PRM) begins by randomly distributing a set of nodes, $\mathbf{Q} = \{\mathbf{q}_1, ..., \mathbf{q}_n\}$, in C. Some planners keep only those nodes that are in C_{free}, some keep all nodes or one could even use some other scheme for selecting which nodes to keep [3]. Next the planner connects all *adjacent*[1] nodes in $\mathbf{Q}$, using the local planner. Two nodes $\mathbf{q}_i$ and $\mathbf{q}_j$ are said to be adjacent if the (weighted) distance $\|\mathbf{q}_i - \mathbf{q}_j\|_{\mathbf{w}}$ is smaller than a given threshold δ.

[1] Some PRM planners instead connects the k nearest nodes, giving each node a fixed number of edges.

Definition: Two nodes $\mathbf{q}_i$ and $\mathbf{q}_j$ that belongs to C are said to be **adjacent** if the weighted distance $\|\mathbf{q}_i - \mathbf{q}_j\|_{\mathbf{w}} < \delta$, for a given δ. Where $\|\mathbf{q}_i - \mathbf{q}_j\|_{\mathbf{w}} = \sum_{k=1}^{\dim(C)} \sqrt{\left(\left(q_k^i - q_k^j\right)^2 \cdot w_k\right)}$. Where $w_k \in \mathbb{R}$ is a weight that biases the importance of movement along a specific dimension of C.

The local planner should be simple and fast, usually a straight line in C is used [1]. Such line connecting two points in the roadmap is known as an *edge*.

Once the roadmap is constructed it needs to be searched for valid paths from $\mathbf{q}_{start}^c$ to $\mathbf{q}_{goal}^c$. There exists two different approaches when searching the roadmap for a feasible path. The first approach begins by removing all the nodes and edges that are illegal. A node or an edge is illegal if it contains any point that is not in C_{free}. When all illegal nodes and edges have been removed the planner searches the remainder of the roadmap for the shortest path. The other approach begins by searching the roadmap for the shortest path, *assuming* all edges are legal, and when a path is found checks to see if it is a legal path. If the path was legal the planner is done, if not it removes the edge found to be illegal from the roadmap and tries again. This later approach is called Lazy PRM and was introduced in [1].

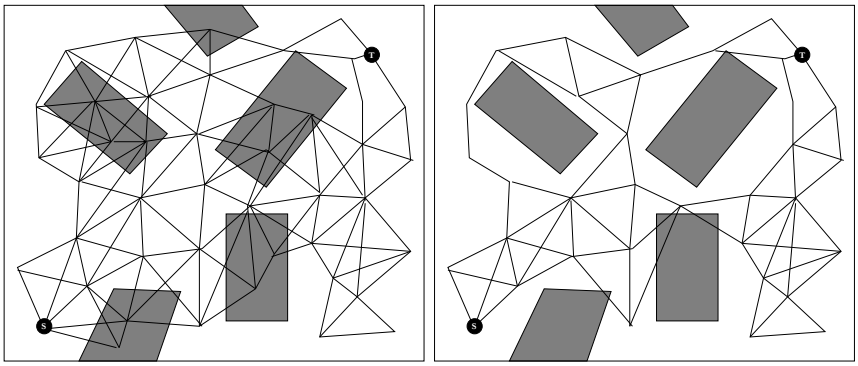
Figure 2.4 shows an illustration of how the PRM planner works. A point shaped agent must move from S to T without hitting any of the obstacles (marked in gray).

2.3.2 Enhancing the Roadmap

If the search for a path in C fails, i.e. the start and goal configurations becomes disconnected, the planner can enter an *enhancement step*. In this enhancement step more nodes are added to the roadmap according to some policy. The simplest form of enhancement simply distributes new nodes at random but better schemes exists. One common way is to distribute new nodes close to some number of nodes randomly selected among all existing nodes in $\mathbf{Q}$ according to some probability depending on the difficulty of connecting a node. For instance in [7] the probability of a node being selected is $\frac{1}{1+e}$, where e is the number of edges connected to that node.

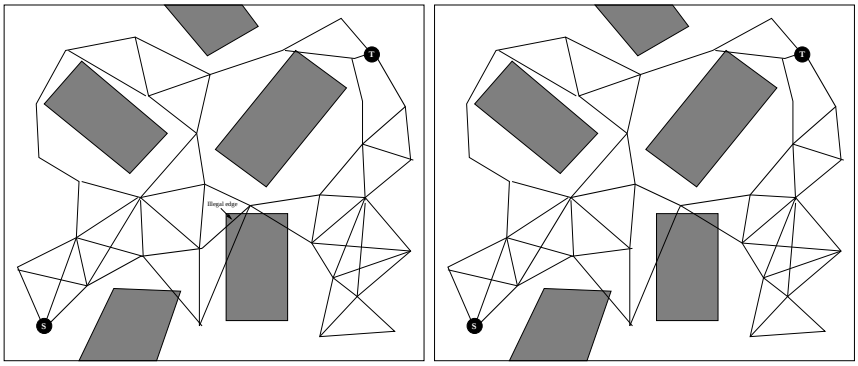
2.3.3 The Narrow Passage Problem with PRM

One problem with PRM methods is that they often fail to solve fairly simple problems when the agent must pass through a narrow passage [9, 3]. This is because most sampling strategies distributes nodes uniformly in C and thus narrow regions will contain few or no nodes which the local planner might not be able to connect to the rest of the roadmap. To overcome this problem one might distribute more nodes to increase the probability of nodes ending up in a narrow region. This approach has one major drawback, it increases the size of the roadmap. Having too many nodes in "open" areas of C is a waste of computational resources. The enhancement step described in section 2.3.2 can be used, especially with a good resampling strategy, to overcome this problem. As it will be described in section 3.2, relying on the



(a) The initial roadmap constructed by a simple PRM planner that uses uniform sampling and keeps all nodes.

(b) The planner in this thesis does not keep points that are not valid. It is clear that this can reduce the size of the roadmap significantly. The significance of this depends strongly on the ratio between free C-space and C-space occupied by obstacles.



(c) Once the roadmap is constructed it is searched for the shortest path. In this case the path is not valid since it contains an illegal edge.

(d) The illegal edge is removed and the roadmap is searched again. This time the planner finds a path connecting S and T.

Figure 2.4. Example of a PRM planning task

enhancement step has several drawbacks and methods that have a lower probability to enter an enhancement step is preferable.

2.4 Artificial Potential Functions

The use of artificial potential functions as a way of solving the path planning problem of section 2.1 is described in [10, 11, 12]. The key idea is to compute the solution of *Laplace's equation* (2.1) in a domain $\Psi \subset \mathbb{R}^N$.

$$\nabla^2\phi(\mathbf{x}) = \sum_{i=1}^{N} \frac{\partial^2\phi}{\partial x_i^2} = 0 \tag{2.1}$$

Laplace's equation is one of the most important equations in applied mathematics. It can be used to describe phenomena such as steady-state heat conductivity problems and the electric potential in dielectric mediums containing no electric charges [13]. More importantly Laplace's equation can be used to describe the potential of a particle in free space acted on only by gravitational forces [13], this is the property artificial potential methods try to mimic. Because of this property Laplace's equation is often referred to as the *potential equation*.

Since Laplace's equation describes a steady-state, no initial conditions must be satisfied. However, boundary value conditions must be met on the bounding surface $\partial\Psi$ of the region in which (2.1) is to be solved. To obtain a solution to (2.1) it is sufficient to express a boundary condition on *every point on the boundary* $\partial\Psi$ [13].

The Dirichlet and Neumann boundary conditions are the most commonly used boundary conditions when solving Laplace's equation. With the Dirichlet boundary condition all points on the boundary $\partial\Psi$ are held at a fixed value. Equation (2.2) shows a Dirichlet boundary condition with the potential held fixed at a constant c on every point on the boundary. Throughout this thesis a Dirichlet boundary condition will mean that (2.2) is satisfied.

$$\phi \mid_{\partial\Psi} = c,\ c \in \mathbb{R} \tag{2.2}$$

With the Neumann boundary condition the derivative of ϕ in the direction of the normal of the boundary $\partial\Psi$ is given on every point on the boundary $\partial\Psi$. Throughout this thesis a Neumann boundary condition will mean that (2.3) is satisfied.

$$\frac{\partial\phi}{\partial\mathbf{n}} \mid_{\partial\Psi} = 0, \text{ where } \mathbf{n} \text{ is the vector normal on the boundary } \partial\Psi. \tag{2.3}$$

2.4.1 Artificial Potential Fields for Path Planning

Solving (2.1) in C_{con}, with the potential ϕ held fixed at 1 (equation (2.2) with $c = 1$) on all points $\mathbf{q}^c \notin C_{free}$ and at -1 on all points $\mathbf{q}^c \to \mathbf{q}^w_{goal}$, will result in an artificial potential field ϕ_c in C_{con}. Performing *gradient descent* on ϕ_c will result in a path from the starting point $\mathbf{q}^c_{start}$ to a minima $\mathbf{q}^c_{min}$. If $\mathbf{q}^c_{min} \to \mathbf{q}^w_{goal}$ (a solution has

been found) the planner is done, however if $\mathbf{q}^c_{min} \nrightarrow \mathbf{q}^w_{goal}$ (a solution has not been found) $\mathbf{q}^c_{min}$ is a *local minima* that must be escaped from. The escape from a local minima can be achieved by means of a *random walk* or a *local search*. When the local minima has been escaped from, a new gradient descent may be performed and a new minima is located. This process can then be repeated until a solution has been found.

One drawback of the potential field approach is the creation of local minima that does not correspond to a goal configuration. If the world in which the agent operates is complicated, containing many objects or objects of complex shape, the agent may get stuck moving from one local minima to another not reaching the goal in acceptable time. Another problem is that as the dimensionality of C increases the time required to compute ϕ grows rapidly. The problem with high dimensional C may be circumvented by computing ϕ in W. Since $1 \leq \dim(W) \leq 3$, ϕ may always be computed in W in a relatively short time (compared to C that may have 10s of dimensions). Once the potential $\phi^w(\mathbf{x})$ is known in W the potential $\phi^c(\mathbf{q})$ for a specific location in C may be computed by summing the potential for all points in the region $\Omega \subset W$ occupied by A [12]. This relation is shown for the discrete case in (2.4).

$$\phi^c(\mathbf{q}) = \sum_{\forall \mathbf{x} \in \Omega} \phi^w(\mathbf{x}) \tag{2.4}$$

Equation (2.4) makes it possible to use potential fields for higher dimensions of C than would otherwise be feasible, however in order to perform gradient descent not only the potential of the current configuration must be known but rather the potential of all neighboring configurations. The time required to compute all those potentials will eventually grow to unacceptable values, still it is usually much better then computing ϕ in C explicitly.

The potential function can be computed numerically using standard finite difference methods (FDM) such as Gauss-Seidel iteration, Jacobi iteration or the Newton-Rhapson method [10]. Since Laplace's equation (2.1) can be used to describe voltages in a resistive grid, a resistive grid can be used to obtain an analog solution to ϕ in a matter of microseconds [10].

2.4.2 The Narrow Passage Problem with Artificial Potential Fields

Artificial potential field planners work well in relatively uncluttered workspaces, however if the agent has to move through a narrow passage artificial potential field planners, just as PRM planners, experience problems. This is because often the potential in a narrow passage will be high. If a local minima exists near the entrance of a narrow passage it is unlikely that the planner will be able to escape this minima through (or actually over) the high potential ridge in the narrow passage.

Chapter 3

Artificial Potential Biased PRM

Both PRM and artificial potential functions have been successfully used to solve numerous path planning problems, however both these methods often fail to solve problems when the agent has to pass through a narrow passage in C (section 2.3.3 and 2.4.2). In this section, a new method is proposed, artificial potential biased PRM (APBPRM). This method combines PRM and artificial potential functions into a new roadmap planner that is better in dealing with the narrow passage problem. This method uses a (partial) computation of the artificial potential field to bias the sampling of nodes in C, increasing the probability of distributing nodes in narrow regions of C.

3.1 Theory Behind APBPRM

When solving Laplace's equation (2.1) numerically, finite difference methods (FDM) [14] can be used. There exists some commonly used iterative FDMs, such as Jacobi iteration, Gauss-Seidel iteration, Crank-Nicolsons method or Successive Over-Relaxation (SOR) which can be used to solve Laplace's equation [10, 14]. For (2.1) these methods essentially replace each grid point's value with a weighted average of its neighbors. This is then repeated iteratively until a stable solution is found.

Instead of computing the solution of Laplace's equation, APBPRM uses the idea that while solving for the potential ϕ, iterative methods will in general cause the potential to *rise more rapidly in narrow regions*. An intuitive way to realize this is that a grid point that has a Manhattan distance of n to the closest boundary point will remain at (the initial) zero potential for the first n steps of the iterative computation. Grid points close to the boundary of Ψ can on the other hand be updated many times during the first n iterations and thus rise to a high potential. This is especially true for grid points surrounded by boundary points, such as grid points in narrow or concave regions of Ψ.

APBPRM computes ϕ_N, the first N steps of a solution to ϕ using FDM, and use it to bias the distribution of nodes in the roadmap. The node distribution scheme used by APBPRM works as follows:

First a set of nodes, $\mathbf{Q}_{rnd}$, is distributed at random uniformly throughout C, keeping only nodes that satisfy $\mathbf{q} \in C_{free}$ until $\mathbf{Q}_{rnd}$ contains M nodes. Then more nodes are randomly distributed in the same way but keeping only nodes that satisfy $\mathbf{q} \in C_{free}$ with a probability P given by equation (3.1) until the set of nodes $\mathbf{Q}_{apb}$ contains K nodes.

$$P = k_\phi \phi_N + k_r, \text{ where } k_\phi \text{ and } k_r \text{ are aribitrary constants } k_\phi,\ k_r \in \mathbb{R} \qquad (3.1)$$

Using the probability in (3.1) *all* nodes are kept with at least the probability k_r[1], and the probability of keeping a node is increased proportionally to ϕ_N. The set of nodes in the roadmap is finally constructed by combining the two sets of nodes to a new set $\mathbf{Q} = \mathbf{Q}_{apb} \cup \mathbf{Q}_{rnd}$ that form the nodes of the roadmap. This will result in *denser sampling of the C-space close to obstacle boundaries* and especially in narrow and concave regions. The idea that a denser distribution of nodes along C-space surfaces helps to guide the agent through narrow passages is also supported by [8].

3.1.1 Probabilistic Completeness

In this section the probabilistic completeness of APBPRM is discussed. A proof of the probabilistic completeness of PRM planners is given in [1].

Definition: A path planner is called **probabilistically complete** if the probability of solving any given problem approaches 1 as the planning time approaches ∞, provided a solution exists.

Because APBPRM is a simple extension to normal PRM it has the same probabilistic completeness as other PRM planners. This means that APBPRM will be able to solve any given problem for any given agent for which a solution exists, given sufficient running-time and that the agent is *locally controllable* [15]. The property of local controllability is further discussed in [15] and essentially means that an agent A always can move in a neighborhood of $\mathbf{q}$ for any given $\mathbf{q} \in C_{free}$. The definition of local controllability used in [15] is given below.

Definition: Given a robot A, let $\sum_A$be its control system. That is, $\sum_A$ describes the velocities that A can attain in C-space. For a configuration $\mathbf{q}$ of a robot A, the set of configurations that A can reach within time T is denoted by $A_{\sum_A}(\leq T, \mathbf{q})$. A is defined to be **locally controllable** if for any configuration $\mathbf{q} \in C_{free}$, $A_{\sum_A}(\leq T, \mathbf{q})$ contains a neighborhood of $\mathbf{q}$ for all $T > 0$.

3.2 Benefits of APBPRM

In this section some of the benefits of APBPRM over standard PRM, as well as some PRMs that have been modified to deal with the narrow passage problem, are described.

[1] Unless $k_r < 0$ which might be interesting to investigate in some high dimensional cases.

Because of the probabilistic completeness of PRM (section 3.1.1), it can solve any problem given that a sufficiently large number of enhancements are made to the roadmap. This fact implies that a biased sampling scheme might not be necessary, however this is not entirely true. One obvious reason to prefer a biased sampling scheme is that graph search time is reduced since the roadmap has to be rebuilt and searched again for every enhancement step. Also, the enhancement steps often tend to oversample the "open" regions of C, creating a roadmap with more nodes than are actually required to solve the problem (increasing search time). Another issue arises when there are *two (or more) ways to reach the target* and one way is shorter than the other but contains a narrow passage (figure 3.1) .

A normal PRM planner could probably find the "long way around" (dashed line in figure 3.1) fairly easy, given a suitable number of initial nodes in the roadmap. Since the planner found a solution to the path planning problem it will be satisfied and not go into an enhancement step. While this behavior might be acceptable under some circumstances it would be better if the planner would find its way through the narrow passage, taking the shorter path. There is, of course, a trade-off here. A real agent could probably travel with a higher speed if it decided to take the "long way around" because it would have to be less concerned with bumping in to the walls, thus the time required is not directly related to the length of the path. An agent choosing the short path might have to go very slowly to navigate the narrow passage while an agent taking the "long way around" could go at a much higher speed. If such considerations are taken in to aspect the function that measures the "goodness" of a path in C-space might have to be changed to penalize paths that are too close to obstacle borders. The graph searching algorithm should then optimize on this "goodness" function rather than the C-space distance.

From the planner's perspective it is better to find both paths and then choose the best, according to some metric, than to only find one path and not have the possibility of choosing the other path. The APBPRM would have a much higher probability of finding the narrow passage path (solid line in figure 3.1), given the same number of initial nodes, as the normal PRM for the same problem.

3.2.1 Other Schemes for Dealing with the Narrow Passage Problem

Other schemes already exists that deal with the narrow passage problem of PRM [3, 6, 8]. These usually work by first distributing nodes uniformly throughout C and then, using information attained from this sampling of C, enhance the roadmap. This enhancement is done in different ways.

In [6], if the roadmap is disconnected in a place where C_{free} is not, this place is considered to correspond to some narrow passage, and hence difficult region of C_{free}. Nodes in such regions are then *expanded.* Expanding a node $\mathbf{q}$ corresponds to adding more nodes in the neighborhood of $\mathbf{q}$. All nodes in the roadmap are given a positive weight $w(\mathbf{q})$ which is a heuristic measurement of the "difficulty" of the neighborhood of $\mathbf{q}$. Thus, $w(\mathbf{q})$ is larger whenever $\mathbf{q}$ is considered to be in a difficult region of C_{free}. With w normalized ($\sum_{\forall \mathbf{q}} w(\mathbf{q}) = 1$), nodes are repeatedly selected

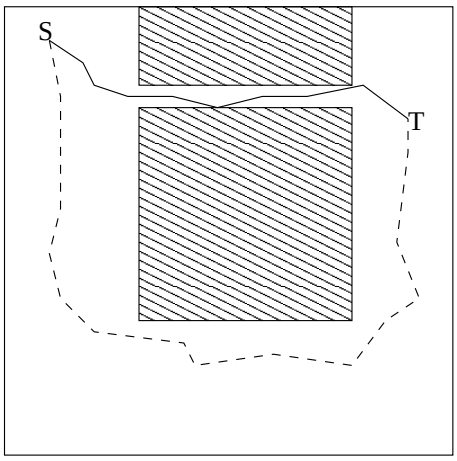


Figure 3.1. A point shaped agent planning a path from S to T in an environment containing a narrow passage. APBPRM yields the solid line path and standard PRM yields the dashed line path.

from the roadmap with probability $P(\mathbf{q} \text{ is selected}) = w(\mathbf{q})$ and then expanded. Several ways to define the heuristic $w(\mathbf{q})$ are given in [6]. One of these is similar to the method discussed in section 2.3.2, using a $w(\mathbf{q})$ that is inversely proportional to the number of neighbors. This method has the drawback that collision detection, roadmap construction and roadmap search have to be carried out several times.

In [8] an obstacle-based PRM (OBPRM) planner is considered. This planner tries to add sample points close to or on C-space obstacle surfaces, similar to APBPRM. The OBPRM described in [8] works in three steps. First there is the node generation step, in which nodes are distributed in C in a way that increases the node density at C-obstacle surfaces. This is accomplished by finding configurations $\mathbf{q}_i$ that intersect with C-obstacles (i.e. $\mathbf{q}_i \notin C_{free}$). From these configurations, "rays" are shot out in random directions and the bounding surface of the C-obstacle is located by means of binary search. The second step is the roadmap connection where several more powerful local planners are used. First, the simple straight line planner is used to connect the nodes in C, and then, in regions found to be difficult, more advanced (and hence slower) planners are used. In the third step, the more powerful planners may also add new nodes to the roadmap, increasing the connectivity of the roadmap. This OBPRM actually deals with a quite general path planning problem with obstacles and APBPRM could easily be incorporated into this general planner in the node distribution step, possibly reducing the amount of work that has to be done by the more advanced planners.

The planner in [3] uses the notion of *dilated free space* to increase the density of nodes near the boundary of C_{free}. This means that C_{free} is expanded, allowing the agent to "penetrate" some distance into obstacles. Nodes are then distributed in this dilated free space and nodes found to lie outside C_{free} is then "pushed" into C_{free} by a local resampling operation. This method would presumably fail given a task

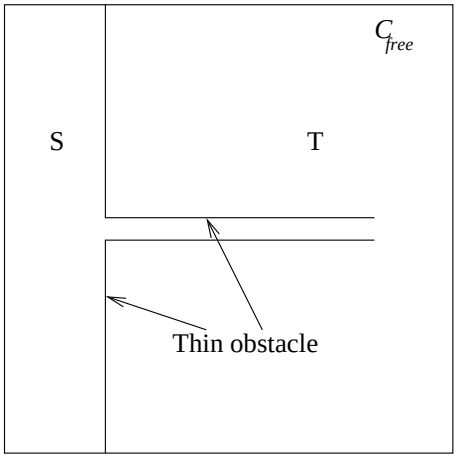


Figure 3.2. A situation where the use of dilated free space would fail because of thin obstacles. The agent must move from S to T and no expansion of C_{free} is possible.

where very thin objects are present, making it impossible to expand C_{free} (figure 3.2).

3.2.2 Computational Benefits

Using the potential function to bias sampling in a PRM planner could provide some computational benefits. First of all, since the APBPRM planner is less likely to go into the enhancement step, roadmap connection time and search time could be reduced. Since node distribution time usually requires a few percent of the total computation time and roadmap connection and search makes up the rest, it would be preferable to minimize the number of connections and searches made. Although APBPRM adds somewhat to the time required for distributing nodes it is presumably better than doing one or more extra connect/search steps.

Since APBPRM uses a partial solution of Laplace's equation to bias the search, one could easily imagine a scheme where a more accurate solution of Laplace's equation is computed (more steps in the FDM solution). This solution could then be used for gross motion planning or to guide the search algorithm when searching the roadmap, i.e. search "down-hill" first. Using this scheme would minimize the time "wasted" when computing the partial solution to Laplace's equation, however it is beyond the scope of this thesis to consider this.

3.3 Drawbacks of APBPRM

So, is APBPRM the "perfect" technique that will rid the world of the narrow passage problem? No, of course not. As often is the case with heuristic methods it is often quite easy to find problems where they do not work or produce strange results. What

APBPRM provides is (yet) another way of handling the narrow passage problem, increasing the set of "tools" available for people who construct motion planners to use in real world applications. As most other schemes for dealing with the narrow passage problem, APBPRM has some drawbacks.

Because the potential function is usually quite steep near obstacles the planner will tend to "crawl" near the edges of obstacles. While this is not an issue while planning for a massless agent in a completely known environment, it is when planning motions for real physical agents in an approximation of the real world. In the real world the agent should probably have some minimum clearance to the obstacles. Also the agent probably has to go more slowly when close to obstacle boundaries to avoid collisions. Often APBPRM generates paths where the agent crawls along the edges of obstacles.

Also the computation of the solution to Laplace's equation in $\mathbb{R}^3$ is quite expensive, limiting the usefulness of APBPRM in environments with many moving obstacles.

Chapter 4

Implementation

To test the theoretical foundation of APBPRM a sample PRM planner with support for artificial potential biased sampling was implemented using C++. Due to the time limitations associated with this work and the fact that the main purpose of the planner was to test the theory behind APBPRM some key features of the planner arose:

- The testing should occur in a simulated environment with virtual agents.
- To fully understand the impact of the biased sampling scheme a complete (and hence slower) graph searching algorithm should be used.
- The planning time can be allowed to be greater than what would be acceptable on a real system, however it must still be relatively efficient to allow simulations to complete in reasonable time (hours).
- The planner should be extensible. This means that it should be simple to adapt to new types of agents and it should be possible to continue developing the planner to the point where it can be used in a real system.

The extensibility of the planner is easily achieved by using the object oriented (OO) paradigm. The planner consists of three principal classes:

1. The `World` class contains information about a normalized 2D or 3D world. This information include obstacles, the goal and the partial solution to the potential function.
2. The `RoadMap` class maintains the roadmap graph, implements the local planner and provides means for searching and checking the roadmap.
3. The abstract `Agent` class specifies an interface for which each type of agent should have its own implementation. The `Agent` class provides agent specific knowledge, such as FK, collision checking and retrieval of the C-space potential.

The world is modeled as a uniform, variable resolution grid with the world coordinates normalized, i.e. $x, y, z \in [0, 1]$. A `World` object begins by loading a bitmap image representation of the world where the obstacles are marked by a 1 and the free space is marked by a 0. Once the world representation is loaded the `World` object computes and stores ϕ_N. The `World` class provides access to the partial potential for points in W (truncated to the nearest grid point) and a boolean function that tests if a point in W lies in W_{free}.

A `RoadMap` object is provided with a list of nodes and a start and goal configuration. A `RoadMap` object begins by building the roadmap graph. All nodes, including the start and goal nodes, are inserted in an array and are provided with a unique hash key for efficient reference. All nodes are provided with pointers to their adjacent nodes. The operation of building the graph is $\mathcal{O}(n \log(n))$, where n is the number of nodes in the roadmap. However building the roadmap is a parallel process and can thus take advantage of MP (**M**ulti **P**rocessor) machines. Once the graph is built the `RoadMap` object can be queried for a solution to the path planning problem. The graph is now searched for the shortest possible solution path using Dijkstra's algorithm [16]. Dijkstra's algorithm is $\mathcal{O}((n + e) \log(n))$ where e is the number of edges and n is the number of nodes in the roadmap [16]. Better algorithms that use a heuristic to guide the search, such as A* search [17], exists [1] but was not used because the behavior of a complete algorithm is easier to understand and analyze. Once a path is found it is checked to see if it is valid or not. The collision checking in [1] is used for high efficiency. If the path is valid the planner is done, if not the edges and nodes found to be illegal are removed and the graph is searched again. This is repeated until either a solution path is found or the goal and start configurations becomes disconnected. If the goal and start configurations becomes disconnected the planner reports failure. No enhancement step is implemented. The algorithm used by the path planner is shown in pseudo code in algorithm 1.

Algorithm 1 A single path planning query.

```
world←Load world from file
if(q_start or q_goal is not valid)
    return FAILURE

Compute potential for world
nodes ←Distribute nodes according to policy
Add nodes to roadmap

Build graph in roadmap

while(q_start and q_goal are connected)
    path←Shortest path from q_start to q_goal in roadmap
    if(path is collision free)
        return path
    remove illegal edge and/or node in path from roadmap
end while

return FAILURE
```

Chapter 5

Experimental Evaluation

To evaluate the performance of APBPRM six different path planning tasks were simulated. The worlds (W-space) in which the simulations took place are shown in figure 5.1. Due to time limitations all tasks involve a point shaped agent moving in the world ($C = W$). At the end of this chapter some preliminary results for a 5 dof planar arm are given.

5.1 Results for a Point Shaped Agent

To measure the performance of APBPRM vs PRM each path planning task was carried out 100 times with each method and the probability of reaching the goal without requiring an enhancement step was calculated. Also the average number of paths that where tested for collision was calculated. More detailed results, including graph build and search time can be found in appendix C.1. Each planning task was carried out with a different number of nodes in the roadmap. The results of these simulations can be seen in table 5.1-5.6 and in more detail in appendix C.1. In all simulations ϕ_{100} was used and nodes where kept with a probability $P = \phi_{100} + 0.1$ (equation (3.1)). The world was modeled as a 180×180 grid and the C-space distance was computed with the same weight in all dimensions. Nodes in the roadmap where considered adjacent if their C-space distance was less than $\frac{1}{\sqrt{20}}$ and the number of neighbors where limited to a maximum of 100.

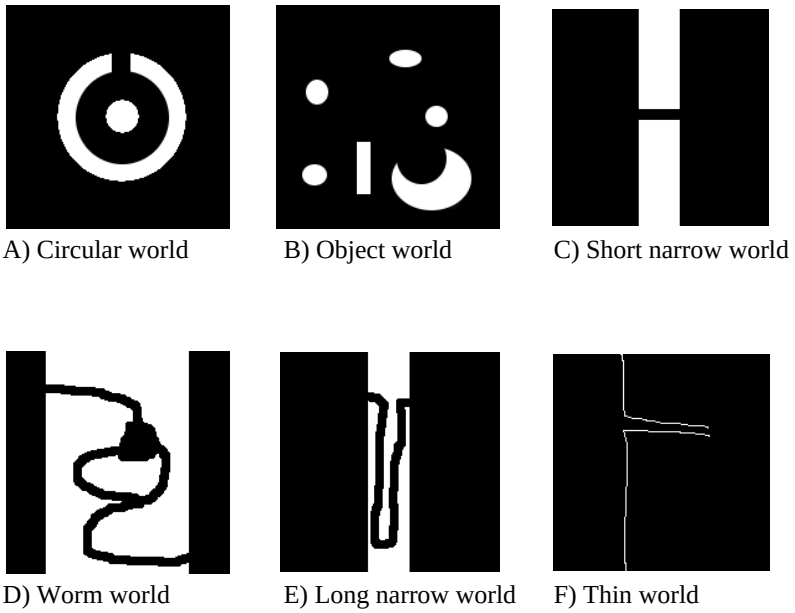


Figure 5.1. The different worlds used to evaluate the performance of APBPRM.

Nodes	Goal reached (%)	Paths tested
100	96	188
250	100	1197
500	100	3945
750	100	5764

Nodes	Goal reached (%)	Paths tested
100	73	45
250	99	330
500	100	1268
750	100	2606

Table 5.1. Simulation statistics for world A, $\mathbf{q}_{start} = (0.5, 0.0)$ and $\mathbf{q}_{goal} = (0.5, 0.35)$. Average values over 100 trials. Left with APB sampling and right with uniform sampling.

Nodes	Goal reached (%)	Paths tested
100	71	21
250	100	51
500	100	100
750	100	125

Nodes	Goal reached (%)	Paths tested
100	91	9
250	100	16
500	100	37
750	100	53

Table 5.2. Simulation statistics for world B, $\mathbf{q}_{start} = (0, 0)$ and $\mathbf{q}_{goal} = (1, 1)$. Average values over 100 trials. Left with APB sampling and right with uniform sampling.

Nodes	Goal reached (%)	Paths tested
100	69	21
250	97	146
500	100	440
750	100	597

Nodes	Goal reached (%)	Paths tested
100	41	5
250	81	18
500	97	59
750	100	115

Table 5.3. Simulation statistics for world C, $\mathbf{q}_{start} = (0, 0)$ and $\mathbf{q}_{goal} = (1, 1)$. Average values over 100 trials. Left with APB sampling and right with uniform sampling.

Nodes	Goal reached (%)	Paths tested
100	10	86
250	84	865
500	99	3352
750	100	5887

Nodes	Goal reached (%)	Paths tested
100	0	20
250	43	287
500	92	1357
750	100	2647

Table 5.4. Simulation statistics for world D, $\mathbf{q}_{start} = (0, 0)$ and $\mathbf{q}_{goal} = (0.95, 0.45)$. Average values over 100 trials. Left with APB sampling and right with uniform sampling.

Nodes	Goal reached (%)	Paths tested
100	1	362
250	68	2643
500	100	7505
750	100	11684

Nodes	Goal reached (%)	Paths tested
100	0	36
250	2	277
500	22	1305
750	61	2953

Table 5.5. Simulation statistics for world E, $\mathbf{q}_{start} = (0.2, 0.5)$ and $\mathbf{q}_{goal} = (0.8, 0.5)$. Average values over 100 trials. Left with APB sampling and right with uniform sampling.

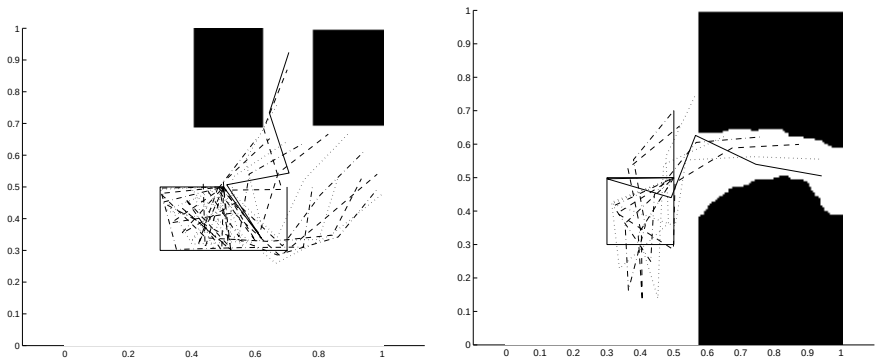
Nodes	Goal reached (%)	Paths tested
100	55	262
250	96	866
500	99	1739
750	100	2381

Nodes	Goal reached (%)	Paths tested
100	0	82
250	45	351
500	90	789
750	98	1069

Table 5.6. Simulation statistics for world F, $\mathbf{q}_{start} = (0.2, 0.5)$ and $\mathbf{q}_{goal} = (0.8, 0.5)$. Average values over 100 trials. Left with APB sampling and right with uniform sampling.

5.2 Preliminary Results for a 5 dof Planar Link Agent

This section shows some preliminary results of APBPRM when applied to a 5 dof planar arm. Because the planner uses a complete algorithm it is not possible to use the amount of nodes needed[1] to provide sufficient resolution in C-space, thus giving poor results. Also no investigation of the effect of the parameters k_ϕ and k_r in equation (3.1) has been done. Two different task where tested, the W-spaces of the tasks are shown in figure 5.2. The results of the tests are shown i table 5.2.



(a) (b)

Figure 5.2. Two tests with a 5 dof planar arm. The start and goal locations are shown (solid lines) as well as some intermediate positions.

A		
Planner	Goal reached (%)	Paths tested
APBPRM	78	14.9
PRM	69	5.0

B		
Planner	Goal reached (%)	Paths tested
APBPRM	30	18.0
PRM	23	3.3

Table 5.7. Planning results for a 5 dof planar arm in the two worlds from figure 5.2. Average results over 100 trials.

5.3 Summary and Analysis

The path planning problem is indeed a difficult one to which there is currently no single solution that is capable of efficiently dealing with all aspects of the problem. Rather an efficient path planner has to be built using a combination of various methods. Which methods to include in the path planner depends on the tasks the agent is to perform, as well as the environment in which it operates. Sometimes the

[1] Because the planning time would be too long

planner is in dire need of a good scheme for finding paths through narrow passages and sometimes the narrow passage problem is completely irrelevant to the planner.

Because APBPRM is an extension and combination of two existing methods it could easily be incorporated into existing schemes. Also there is a code base available, providing methods for computing Laplace's equation and performing graph search which can be utilized to quickly build an APBPRM planner.

The results given in chapter 5 shows that the APBPRM planner performs different under different circumstances. In relatively open regions, such as world A and B in figure 5.1, the use of the biased sampling scheme provides little or no improvement over a random sampling scheme. It can actually perform slightly worse than the random sampling scheme (world B) under some circumstances. This result is not very strange, consider a situation where the world is vast and contains a single relatively small object. Planning in such environment does not require special consideration of the narrow passage problem and thus a random scheme works well. In this case APBPRM will only "waste" nodes by putting them near the obstacle surface, where there is no actual use for a denser sampling. This, of course, is not the type of planning tasks the APBPRM planner was intended for. In complex planning tasks, especially those containing a single long narrow passage as well as a an extensive amount of open space, the APBPRM outperforms the standard PRM planner (world E).

The APBPRM method will (hopefully) provide a useful tool for dealing with the narrow passage problem. The method has both advantages and disadvantages compared to other methods that deal with the narrow passage problem. Which methods, or combination of methods, to use depends on the requirements of the agent and the environment in which it will operate.

Chapter 6

Results and Conclusions - APBPRM

It is clear from the results in chapter 5 that APBPRM can be used to increase the probability of finding passages through narrow passages for a point shaped agent. The success of APBPRM indicates that denser sampling around C-space obstacles surfaces aid the planner in narrow and cluttered regions.

No effort to incorporate the gradient of the potential function into the roadmap search has been done, however this is an interesting idea that deserves further investigation. The preliminary results for a 5 dof planar arm indicate that the success of APBPRM declines some with increased dimensionality of the C-space. This does not have to be true and also deserves further investigation. For instance the relatively poor performance reported in section 5.2 could be due to a bad selection of parameters in equation (3.1). If k_r is too high the sampling of the C-space will be too gross, i.e. nodes are kept too easy. Combined with a uniform random sampling it might even be interesting to evaluate the performance with $k_r < 0$.

It might be interesting to investigate other "potential functions" altogether, for instance it is possible do a simple search in the neighborhood of each point in W to determine the average distance to nearby objects and store such information as the "potential" of that point. Perhaps it is possible to find equations that will not be as steep near obstacles as the potential function is, such as a Gaussian or some type of linear function. All of this is beyond the scope of this thesis to consider. The artificial potential seems to work as expected by biasing the sampling of points near obstacle boundaries and in particular in narrow and concave regions.

6.1 Future Work

Some of the ideas presented in chapter 3 have not been implemented or tested. Among interesting future work would be to investigate the performance gained when adding biased sampling to existing PRM planners, using their more efficient search algorithms. Also the possibility to use the potential function as a heuristic in search algorithms such as A*-search would require further investigation. Finally the effect of the parameters k_ϕ and k_r in (3.1) needs to be evaluated. Perhaps the probability

given by equation (3.1) needs to be changed, it is probably not optimal to use a probability proportional to the potential function, rather some other relation, such as the power of the potential, could give better results. For instance using $P = k_\phi\sqrt{\phi} + k_r$ would, on average, distribute nodes farther away from C-space obstacle surfaces and thus lessen the tendency an agent has to "crawl" along the edges of C-space obstacles.

Part II

Real-Time Linux

Chapter 7

Introduction to Real-Time Operating Systems

Control of a robot manipulator requires real-time capabilities of the underlying operating system to ensure adequate control performance in terms of safety and trajectory tracking. This is typically achieved using proprietary operating systems such as QNX, OS9 or VxWorks. A disadvantage of such operating systems is that the underlying code in general is not publicly distributed which is a significant obstacle for academic research. Consequently, it is of interest to determine if open-source[1] operating systems (available under GPL or LGPL) can be used for the control of a manipulator. A typical example of such an operating system is GNU/Linux with real-time extensions. This project is a study of the characteristics of real-time UNIX derivatives for control of a robot manipulator. The study was carried out in the context of control of a PUMA 560 robotic arm, one of the most widely used manipulators for academic research. The evaluation of potential systems considered real-time performance, ease of installation and use, and the richness of the associated API.

To control the PUMA 560 manipulator a PID controller was implemented in a parallel M. Sc. project.

7.1 Real-Time Operating System Evaluation

In order to successfully chose an operating system suitable for implementing a PID (**P**roportional, **I**ntegral, **D**erivative) controller [18] there are basically two main aspects to consider. First there is *performance*. If the OS is not capable of handling the timing constraints that are needed to successfully execute the control task it is of little or no use. The second aspect to consider is "*usability*". The OS should be easy to understand, with a clear and simple API.

Evaluation of the performance is quite simple since it can be measured in terms of the *scheduling jitter* (section 7.4). The scheduling jitter gives a direct, numeric, measurement of the real-time performance of an OS.

[1] Open Source Initiative (OSI) http://www.opensource.org

The "usability" part presents more problems since there is no way to perform precise measurements that will show which OS is best suited. Rather feelings and impressions will have to do as the guide here. To improve the validity of the "usability" evaluation a table of *pro et contra* was constructed to have something to base the decision on.

7.2 Real-Time Operating Systems

A real-time system is a system that must guarantee a response to an external event within a given time [19]. Thus a real-time operating system (RTOS) must provide facilities to run a task to completion within a given time (usually micro- or milliseconds) after an external event was generated.

Real-time systems are usually divided in two groups, *hard* and *soft* real-time systems. Hard real-time systems are systems where deadlines must be met, *or else!* An example of such system could be an emergency shut-down procedure of a nuclear power-plant. In soft real-time systems the requirements are usually statistically defined. These requirements means the system is allowed to miss some (small) fraction of its deadlines. An example of such system could be a DV-player.

7.3 Real-Time Issues

Linux is a free UNIX [20] like operating system developed by Linus Torvalds and other developers around the world. As the number of features in Linux has grown since its initial release in 1991 its popularity as an alternative to UNIX has grown as well. Today Linux is used in numerous production systems around the world. Linux is basically a UNIX clone and UNIX was originally designed as a time-sharing OS [21, 20]. Linux retains the time-sharing nature of UNIX and, unfortunately, this clashes with that of the real-time system.

7.3.1 Time-Sharing vs Real-Time

The goal of a time-sharing OS is to allow several users to run several tasks concurrently on one processor, or in parallel on many processors [19]. If there is only one user at any time using the OS it is usually said to be *multitasking*. Multitasking is a special case of time-sharing and from now on the term time-sharing will be used to represent the feature where the OS allows several processes, belonging to one or more users, to appear to execute concurrently on a single (uni or multi processor) machine.

There are two different approaches to time-sharing, *cooperative* and *preemptive*. In systems using cooperative time-sharing the running process has complete control over the CPU and must voluntarily release the CPU to let other processes run. While this approach would make it quite easy to write real-time applications it has many drawbacks regarding time-sharing for which it was actually intended. Most modern OSs belong to the preemptive type and Linux is one of them. In a preemptive OS

a special task, called the *scheduler* is responsible for making the processes on the system appear to execute concurrently [20, 21]. The scheduler has the power to suspend any process at any time. For a more in-depth discussion about scheduling in UNIX and Linux see [21, 20]. Typically Linux runs its scheduler at fixed time intervals, however since each time the scheduler is run it introduces overhead, wasting the systems resources, the interval, or the *time-slice*, can not be too small. Typically the time-slice is in the order of 10 ms, however it is not uncommon for real-time systems to require scheduling accuracy down to a few μs. The standard Linux scheduler can not provide such fine resolution and is therefor useless for most real-time applications. Fortunately there are ways to overcome this problem, as will be shown when discussing the different real-time Linux derivatives in chapter 8.

7.3.2 More Real-Time Issues

When scheduling real-time process it is important to have a high timer resolution. If the scheduler is unable to determine, with enough resolution, the elapsed time since a periodic process was last invoked it will not be able to do a good job scheduling it. Assume a periodic task is set to run at a period of 1 ms, if the scheduler is only able to determine the time since the task was last executed at a resolution of 1 ms the actual period could be as large as 2 ms, twice the reference period. Another issue that makes Linux perform bad as a real-time OS is the fact that the Linux kernel itself is, in general, not preemptable. This is because the Linux kernel often disables interrupts as a mean of synchronization [21]. Disabling interrupts incur unpredictability in the system and makes it difficult, if not impossible, to understand exactly how and when the scheduler will actually run.

Some other issues to consider when making a real-time OS include virtual memory and cache problems. Virtual memory introduce unpredictability and comparatively large delays. Cache effects are not as severe but very hard to overcome [22]. See [22] for a more in-depth discussion about these real-time issues and how to measure the real-time performance.

7.4 Scheduling Jitter

When an external event is received it is common that a hardware interrupt is generated. This interrupt switches the CPU to protected mode and executes an interrupt-handler. Consider the special case when this external event is a signal from a timer used by the real-time scheduler to schedule a periodic process. Initially the timer is set to produce interrupts at a specific period P. An interrupt-handler is installed that runs the scheduler every time this interrupt is generated. Since the timer has a limited resolution the actual period between timer events will differ from P with some quantity δ_1. When the scheduler is invoked it has to do some work. This work may include storing the state of the CPU and saving the memory map among other things [21]. All this takes some amount of time δ_2 and contributes to the fact that

the *actual* period P_{true} will differ from P. Now take the difference between P_{true} and P and call this difference ΔP. This difference is known as the *scheduling jitter*.

Definition: Let $\Delta P = P - P_{true}$, where P is the theoretical periods reference value and P_{true} is the actual period. The time ΔP is then known as the **scheduling jitter**.

Measuring

One way to measure the scheduling jitter is to run a periodic task and print out a time-stamp every time the task i actually run [22]. Pseudo code for such application is given in algorithm 2. The `GetTime()` function reads the current time at a high precession, `Print()` prints a variable to screen and the variable `period` is the reference value for the scheduling period.

Algorithm 2 Measuring scheduling jitter. This algorithm can be used to print out the scheduling jitter of a periodic process.

```
wait for next period
oldTime ← GetTime()
while(still testing)
    wait for next period
    time ← GetTime()
    jitter ← period - (time - oldTime)
    oldTime ← time
    Print(jitter)
end while
```

This method of measuring the scheduling jitter has the drawback that it may report an incorrect value if `GetTime()` returns a result with too low accuracy. However, if the accuracy of `GetTime()` is high enough this method is very attractive since it does not require any external hardware and is easy to implement.

7.5 Usability

One important issue when selecting a real-time OS for use in academic research is that it should be easy to learn and master. Since UNIX and Linux is widely spread in academia a real-time UNIX derivative should be able to provide an advantage over other real-time operating-systems. Some of the key features that influences an operating systems usability for academic research are shown in table 7.1.

- **Ease of installation** - The system must install without requiring too many manual changes and tuning.
- **Documentation** - Poorly documented software is harder to learn and much more difficult to master. If a function is not documented well it might not perform the action the user intended. The function name is not always enough do determine the actions of that function.
- **API** - Not only must the API be well documented, as stated above, it is also an advantage if it is subject to some form of standard. The preferred standard in this case would be the POSIX (**P**ortable **O**peration **S**ystem **I**nterface) standard for real-time features in UNIX [21] defined by IEEE.
- **Interaction with Linux** - The advantage of being able to use normal Linux libraries and system calls when developing real-time software is obvious. It is also vital that good communications primitives exist between real-time tasks and the rest of the system.
- **Development and debugging tools** - The easier it is to develop and debug real-time software the more time the researcher can spend working on things actually related to the research topic.
- **Hardware support** - It is considered a good thing if the system is able to run on a variety of platforms. Embedded systems often use processors from other families than the Intel x86 such as the PowerPC or m68k. Since the evaluation has its base in real-time robotic control it might be advantageous to be able to implement a controller on low-power embedded controller instead of a desktop PC. Many ports of the source code may also imply that the code is well written and thoroughly investigated.

Table 7.1. Important usability issues of a real-time OS.

Chapter 8

The Different Operating Systems

This section describes the system architecture of the three different RTOSs tested. Comparisons with standard Linux will be made, key features will be dissected and the issues in section 7.3 will be addressed. The architecture of the standard Linux kernel is shown in figure 8.1. Note that interrupts are delivered directly to the kernel and loaded device drivers, also the user processes are restricted from hardware access in order to guarantee the stability of the system.

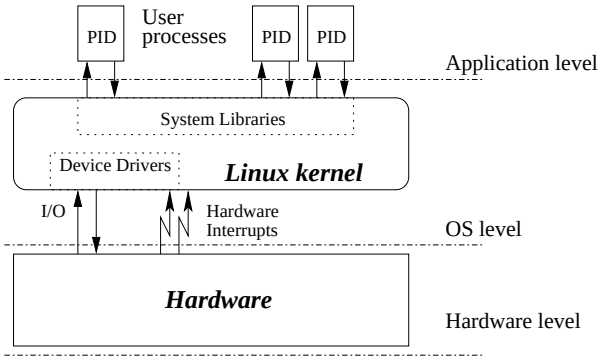


Figure 8.1. Architecture of the standard Linux kernel.

8.1 RT-Linux

RT-Linux was created and is maintained by Finite State Machine, Inc[1]. RT-Linux works by patching the standard Linux kernel with the real-time plug-in shown in figure 8.2. The real-time plug-in is in itself a small predictable operating system, with its own (real-time) scheduler and interrupt handlers. RT-Linux uses a slightly extended POSIX 1003.13 API and supports SMP machines. It is available for the x86, PowerPC and Alpha architectures and supports kernel level periodic and sporadic real-time tasks. RT-Linux also provides status information via the /`proc` portion

[1]Finite State Machine, Inc. http://www.fsmlabs.com

of the file system. Interprocess communication (IPC) is supported through either shared memory or FIFO (**f**irst-**i**n-**f**irst-**o**ut) buffers.

Interrupt Handling

Apart from installing the real-time plug-in, the RT-Linux patch also replaces all the calls in the Linux kernel that disables and enables interrupts as well as the return from interrupt instructions (the `cli, sti` and `iret` operations on the x86 architecture) by macros (`S_CLI`, `S_STI` and `S_IRET`). These macros call functions in the plug-in layer that emulates the interrupt controller hardware, they are described in [23]. Instead of disabling interrupts the plug-in layer resets a variable. When an interrupt occurs it is delivered to the plug-in (which is the actual OS kernel). If Linux has interrupts enabled (the variable is set) the interrupt is forwarded to the Linux kernel, however if the Linux kernel has interrupts disabled the interrupt is stored in a queue and not delivered to Linux until Linux re-enables interrupts. When Linux re-enables interrupts all interrupts in the queue are delivered to Linux. Since these interrupts do not originate directly from the hardware, but from a intermediate software emulation layer (the plug-in), they are known as *software interrupts* [23]. The reason for using software interrupts is that the developers of RT-Linux thought that it would not be feasible to change the Linux kernel to the extent required to eliminate any unpredictability originating from disabling interrupts [24].

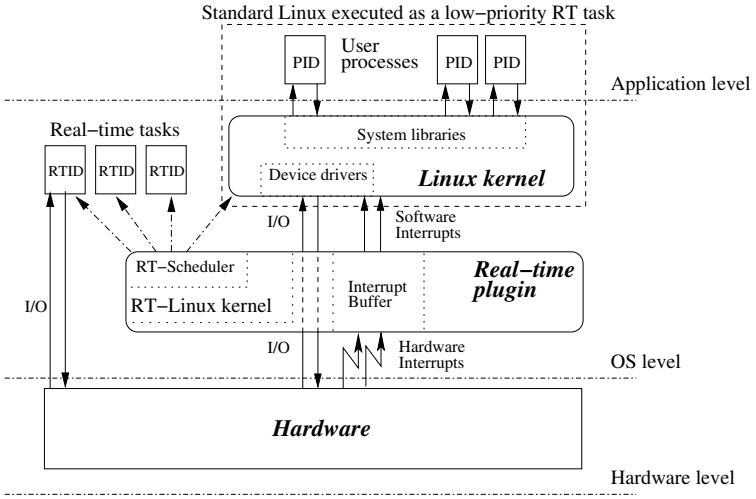


Figure 8.2. Architecture of the RT-Linux kernel.

Real-Time Tasks

A real-time task in RT-Linux is created as a loadable Linux kernel module (a piece of code that can be dynamically linked with the kernel code). All kernel modules provide an `init_module()` function that is invoked when the module is loaded and

a `cleanup_module()` function that is invoked when the module is removed from the kernel. The `init_module()` function initializes the real-time task and notifies the scheduler in the real-time plug-in that it wishes to be scheduled as a real-time task. Since real-time tasks runs in the Linux kernel address space it is not possible to call standard C-library functions (such as `printf()` or `malloc()`), however it is possible to use Linux system-calls (such as `printk()`) but it is *not* safe and it may incur unpredictability in the system!

Scheduling

Two different modes of scheduling are provided by RT-Linux. Periodic mode is used for tasks that need to be run at a specific interval and sporadic mode is used for tasks that are activated by an external interrupt. Every real-time task is provided with a priority and Linux is assigned the lowest priority. This makes Linux act like an idle-task, running only when there are no real real-time tasks ready.

The reason for running real-time tasks in one address space (i.e. the kernel address space) is mainly due to performance issues. When switching between tasks in different address spaces penalties such as TLB-invalidations (**T**ranslation **L**ook-aside **B**uffer) and protection level changes occur, this is discussed further in [24]. RT-Linux provides two schedulers, one priority based scheduler that always runs the task with highest priority, automatically preempting all lower priority tasks. The other scheduler uses the *earliest deadline first* [21] (EDF) algorithm. RT-Linux also provides easy means of extending the real-time plug-in with custom schedulers. Custom schedulers can be implemented as Linux kernel modules. This is discussed further in [25].

Interprocess Communication

RT-Linux provides two basic mechanisms for IPC. The first is a shared memory mechanism that can be used to communicate efficiently between different real-time tasks (as well as the Linux kernel). The second is a FIFO mechanism providing IPC between real-time tasks and user-level applications (standard Linux software). The RT-Linux API provides real-time tasks with functions for creating and destroying FIFOs as well as non-blocking read and write functions. To Linux applications these FIFOs appear to be ordinary character devices located in the `/dev` portion of the file system.

8.2 RTAI

RTAI (**R**eal**t**ime **A**pplication **I**nterface) is the product of an open-source effort begun at the DIAPM (**D**ipartimento di **I**ngegneria **A**erospaziale del **P**olitecnico di **M**ilano). Very similar to RT-Linux (section 8.1) RTAI also provides real-time support by patching the Linux kernel, introducing a new intermediate layer. This layer, known as a hardware abstraction layer (HAL), is depicted in figure 8.3. RTAI has a

cleaner design than RT-Linux, consisting of two parts. First there is the HAL acting basically as an interrupt dispatcher (similar to that described in section 8.1), then there is the RTAI real-time kernel which is responsible for running real-time tasks as well as the Linux kernel as a low priority task. In order to avoid confusion, RTAI is the name of *both* the project (consisting of the HAL and a real-time kernel) and the real-time kernel provided with the project.

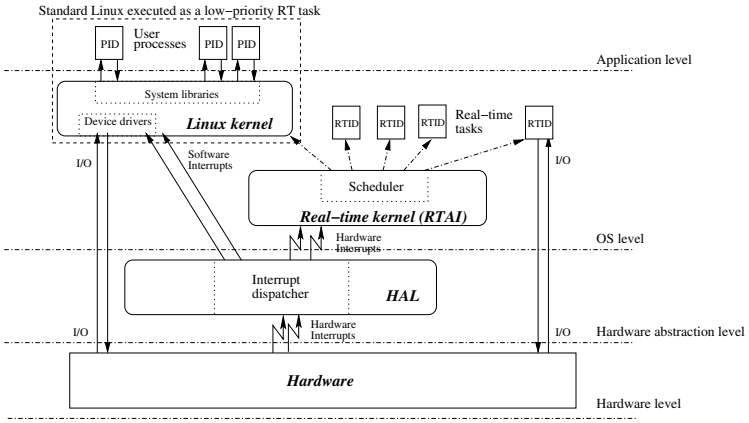


Figure 8.3. Architecture of the RTAI/HAL Linux kernel.

Interrupt Handling

The HAL (along with a few modifications to the Linux kernel itself) is responsible for uncertainty related to hardware, i.e. interrupts. The HAL basically supplies three functions: [26]

- Gather all the pointers to the required internal data and functions into a single structure, rthal, to allow easy trapping of all the kernel functionalities that are important for real time applications, mainly related to the hardware, so that they can be dynamically switched to appropriate software emulation functions by RTAI when hard real-time is needed.
- Makes available the substitutes of the above grabbed functions and sets rthal pointers to point to them.
- Substitutes the original function calls with calls to the rthal pointers in all the kernel functions using them.

To deal with the fact that Linux often disables interrupts RTAI patches the Linux kernel to call function pointers instead of in-lined code. Wherever Linux issues either one of the instructions used to enable or disable interrupts (the `cli` and `sti` instructions on the x86 architecture) RTAI patches the code and inserts a call to

the functions pointed to by the `linux_cli` or `linux_sti` pointers in the HAL data-structure. These function pointers are updated (changed) when switching between real-time and normal mode, thus when no real-time tasks are running interrupt disabling/enabling works as usually. For a more in depth explanation of the HAL see [26].

Real-Time Tasks

As in RT-Linux, RTAI runs real-time tasks in the same address space as the kernel. Real-time tasks are created as loadable Linux kernel modules and are responsible for setting up the real-time scheduling needs in the `init_module()` function. This usually comes down to starting a timer, creating and starting a task (thread), as well as setting up any resources required by the real-time thread.

Scheduling

Because of the highly modular design of RTAI it is quite simple to write custom scheduler modules available for insertion in the system "on the fly". RTAI comes with a priority based one-shot and periodic mode scheduler. The modular design of RTAI makes it possible to easily change, not only the scheduler, but the whole real-time kernel to a kernel different than the RTAI-kernel.

Interprocess Communication

Similar to RT-Linux, RTAI supplies shared-memory and FIFOs for interprocess communication. RTAI also has support for mailboxes that allows messages between processes to be automatically stored and retrieved as needed in a priority queue. The mailbox service is very flexible [27].

- It can be explicitly set up to accept messages of custom sizes.
- Multiple receivers and senders can be connected to the same mailbox where the order in which messages are taken depends on the priority of the receivers.
- When large messages need to be sent, the service provides functions to allow the process to send only the portion of the message that can be stored, returning the number of unsent bytes, or to continue to send the message until all of it has been accepted.

LXRT

One of the most interesting features of RTAI is the LXRT (**L**inu**x** **R**eal-**T**ime) module that allows symmetric access to RTAI services and schedulers in both user space and kernel-space. LXRT makes it possible to develop soft and hard real-time systems in user space through the "buddy system". The (user space) application mates (i.e. forms a connection) with a small real-time task, known as the buddy, who is

responsible for waking up and running the application. Thus the buddy is a small real-time task with capabilities to carry out RTAI services on behalf of its user space counterpart.

When developing hard real-time applications some restrictions are imposed. It is not possible to (safely) use Linux standard libraries or kernel services and the memory-pages of the process must be locked into physical RAM with a call to `mlockall()` to avoid virtual memory delays [28]. When using LXRT for soft real-time applications such restrictions do not apply. Often (and especially in academia) people who use or will use a real-time system often do not have much experience with real-time systems or kernel programming. For these people LXRT will be invaluable in getting started, quickly developing and testing the real-time software. For a good introduction to LXRT see [28].

8.3 KURT

KURT Linux, or Kansas University Real-Time Linux, was first developed in 1998 at the Information and telecommunication technology center at the University of Kansas. KURT is *not* a hard real-time OS but rather *firm* according to the people behind it. KURT was first developed to meet the need of real-time applications residing somewhere between the soft real-time applications that standard Linux provides and hard real-time applications provided by operating systems such as QNX [25].

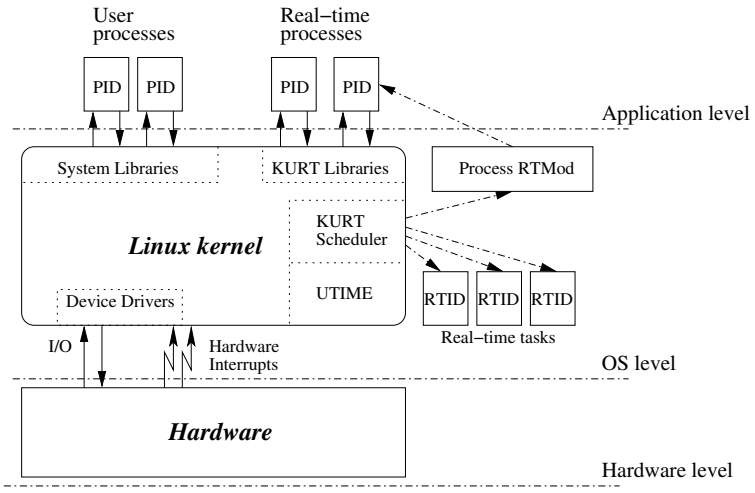


Figure 8.4. Architecture of the KURT Linux kernel.

Interrupt Handling

The architecture of KURT, which is shown in figure 8.4, is quite different from RT-Linux and RTAI. It does not introduce an intermediate layer underneath the

Linux kernel, but rather modifies the kernel itself to provide microsecond resolution scheduling, accurate time keeping, and the ability to specify a real-time schedule [29]. As such KURT does not deal with the fact that Linux disables interrupts and thus can not be considered a hard real-time system, however there exists a special mode to which KURT may be switched called *focused real-time* mode where this problem may be somewhat circumvented. The focused real-time mode will be described later.

UTIME

UTIME is a package that patch the Linux kernel to provide high resolution timers (timers with microsecond resolution, hence the name). A naive approach to increase the resolution of the timers in Linux would be to decrease the time-slice alloted to different processes, however this would result in unacceptable overhead. Instead UTIME uses the fact that there is a difference between the temporal resolution and the frequency of events [30]. This difference leads to the implementation of *dynamic* time-slices. To implement dynamic time-slices UTIME reprograms the timer-chip to interrupt the CPU when the next event is scheduled, instead of after a fixed time. This method is similar to those used by RT-Linux and RTAI to schedule periodic processes. This is accomplished by patching the Linux kernel itself, making modifications to the scheduler and the Linux timer data structure. UTIME uses the high resolution TSC (**t**ime **s**tamp **c**ounter, see section 9.1) to keep track of the absolute time that has elapsed. UTIME even provides a mechanism for synchronizing the TSC using the Network Time Protocol (NTP), effectively removing the clock drift over time. The overhead of UTIME to Linux is reported to be about 0.15% [31], however the overhead is probably less on systems with on-dye timers. The implementation details of UTIME can be found in [31, 30].

Real-Time Tasks

Real-time tasks in KURT are called RTMods (real-time modules) and run as threads in the kernel space. Real-time tasks are created as loadable Linux kernel modules. There exists one built-in real-time module, called the *"Process RTMod"*, this module is responsible for running a user space process when invoked. The Process RTMod allows for user space real-time processes and is similar to the "buddy" used in LXRT (section 8.2).

Scheduling

The KURT scheduler is an explicit plan scheduler, and as such the real-time task must specify explicitly the times at which it needs to be invoked. A real-time task specifies the schedule via a *schedule file*. KURT also supports periodic mode scheduling for *one* real-time task. To minimize jitter KURT programs the timer to interrupt the processor 50 μs *before* a deadline is to be met. The interrupt handler then busy-waits until the actual time expires and then invokes the real-time task.

Since KURT resides *inside* the kernel instead of underneath as RTAI and RT-Linux, it suffers from unpredictability due to the fact that the Linux kernel disables interrupts as a mean of synchronization. To overcome this drawback KURT allows the user to switch the system between three different modes. These modes are: *normal mode, mixed mode* and *focused mode*. In normal mode a KURT Linux system behaves as a normal Linux system except that microsecond resolution timers are available through the UTIME package. In mixed mode, which are the most commonly used for "desktop" computers, both standard Linux processes and real time processes are allowed to run. Thus in mixed mode it is possible for standard Linux processes to incur timing distortions to the real-time system by using system services that disables interrupts, such as disk access[2]. In focused mode *only* processes that are marked as real-time are allowed to run. When running in focused mode Linux is basically disabled and only the set of real-time processes, which often are known to the developer, are allowed to run. Thus the developer can take care that none of the real-time tasks use any system services that disables interrupts. Since focused mode does not allow Linux processes to run it is most commonly used on embedded or dedicated real-time systems.

8.4 Summary

From chapter 8 it is clear that the designs of RT-Linux and RTAI are very similar, the RT plugin in figure 8.2 roughly corresponds to the HAL and RTAI kernel in figure 8.3. However KURT uses a completely different approach and try to provide real-time capabilities from *inside* the Linux kernel rather than *underneath* as RT-Linux and RTAI do. KURT also uses an explicit schedule, constructed from a schedule file instead of the priority based scheduler with support for one-shot and periodic tasks. Below is given a brief summary of some key features in the different operating systems.

- **Implementation**
  - RT-Linux - A small predictable (real-time) OS and interrupt buffer is introduced underneath the normal Linux kernel, which is run as a low priority task.
  - RTAI - A HAL is introduced to handle uncertainty related to hardware. A small predictable (real-time) OS runs on top of the HAL. Linux runs as a low priority task. The real-time OS module is separated from the HAL and can easily be changed.
  - KURT - The Linux kernel is patched to provide microsecond timer resolution and dynamic time slices. An explicit real-time scheduler is introduced in the Linux kernel.

[2]The file system code and the device driver code for the disk drive may block interrupts for up to 250 μs [31].

- **Available architectures**
  - RT-Linux - x86, PowerPC, MIPS, AMD Elan NetSC520 and Alpha.
  - RTAI - x86, PowerPC, MIPS, m68k and ARM.
  - KURT - All platforms supported by Linux that have the same TSC feature as the Pentium processor.
- **IPC**
  - RT-Linux - Shared memory for communication between RT tasks and user space applications. FIFOs for communication between RT tasks and user space applications via a character device in the `/dev` portion of the file system.
  - RTAI - Shared memory for communication between RT tasks and user space applications. FIFOs for communication between RT tasks and user space applications via a character device in the `/dev` portion of the file system. Mailboxes (advanced FIFOs) provides advanced messaging capabilities with custom sized messages, priorities and multiple listeners. When using LXRT in soft real-time mode normal Linux communication primitives can be used.
  - KURT - In soft real-time mode normal Linux communication primitives can be used.
- **Uncertainty related to hardware (interrupts)**
  - RT-Linux - Instructions for disabling/enabling interrupts in Linux is replaced by calls to the RT plugin which maintains a variable that keeps track of the interruptible state of Linux. When interrupts are disabled by Linux they are queued by the RT plugin and delivered when Linux re-enables interrupts.
  - RTAI - Instructions for disabling/enabling interrupts in Linux is replaced by calls to function using function pointers maintained in the HAL. When no RT tasks are running interrupts work as usual. When one or more RT tasks are running the HAL dispatches interrupts to the appropriate place, queuing them if necessary.
  - KURT - Not handled, when run in focused mode it is up to the programmer to make sure no such uncertainty occurs.
- **Real-time processes**
  - RT-Linux - RT tasks are run in the kernel address space.
  - RTAI - RT tasks are run in the kernel address space or user space via the LXRT package.

  - KURT - RT tasks can run in the kernel address space, but more common is that they run in user space via the Process RTMod.

- **Schedulers**

  - RT-Linux - Priority based and EDF schedulers are provided. It is possible to write custom scheduler modules. Support for periodic and sporadic scheduling.
  - RTAI - A priority based scheduler is provided. The scheduler or the complete RT kernel can be changed to a custom one. Support for periodic and sporadic scheduling.
  - KURT - Explicit scheduling via a schedule file.

Chapter 9

Implementation

To select an OS on which to implement the control system of the PUMA 560 manipulator the real-time performance of the three RT Linux derivatives was measured and their usability was evaluated.

9.1 Measuring Scheduling Jitter

Implementing algorithm 2 (page 41) on RT-Linux and RTAI proved to be quite easy since they both support periodic mode. KURT also claims to have a periodic mode, but as will be described later, it has some serious issues.

One crucial part of algorithm 2 is the `GetTime()` function. The accuracy of the results will be limited by the resolution and correctness of the `GetTime()` function, and it is therefore important to implement this with as high accuracy as possible. The platform on which these tests were carried out is described in appendix B. Note that the processor is an Intel Pentium II. Most modern processors, including the Pentium II, have something called a *time stamp counter*, or TSC for short. On the Pentium II the TSC is implemented as an 8-byte integer that is incremented by one on each clock cycle [22]. The system on which the tests were performed runs at 400 MHz, thus the TSC was incremented by one every 2.5 ns. A temporal resolution of 2.5 ns is more than enough to accurately measure the scheduling jitter of a periodic process with a period of 500 μs. For a better understanding of jitter and the effects of deviations in the CPU frequency see [22].

The periodic mode of KURT is broken and will possibly be removed from future releases of KURT [32]. There is also an issue with the UTIME package that makes it unreliable [33]. Finally I was unable to reliably create and run an explicit repeatable schedule. Most of the time the scheduling stopped after repeating itself a few times. It is my opinion that, until the problems with KURT mentioned above have been resolved, it is not possible to reliably measure the scheduling jitter of KURT.

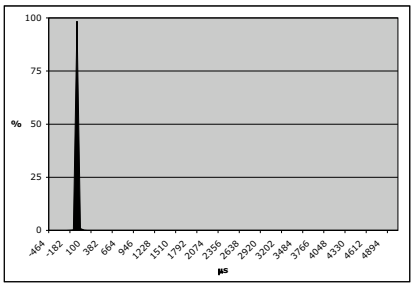
The code that implements algorithm 2 for RT-Linux and RTAI can be found in appendix A. Common for both these algorithms is that they store the results to a file rather than print them on the screen, this allows for later analysis of the

scheduling performance.

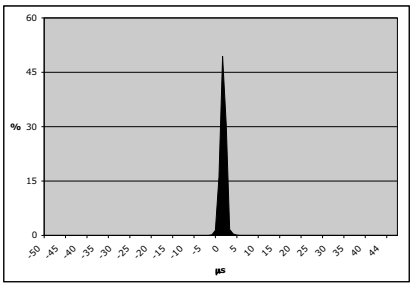
Figure 9.1 and 9.2 shows the scheduling jitter of RT-Linux and RTAI. All tests where run on the system described in appendix B. The real-time task that measured the jitter was run at a frequency of 2 kHz, for a total of 100 000 periods. The jitter was measured when the rest of the system was idle as well as very stressed (i.e. high CPU and I/O load). The following four commands where used to stress the CPU and I/O and gave near constant 100% CPU usage.

```
sudo ping -f 192.168.218.32

while "true"; do ls -aR; sync; done

while "true"; do cp /home/shared/kernel/linux-2.4.18.tar.bz2 \
/tmp/tmpkern; sync; rm /tmp/tmpkern; sync; done

top -d 0.05
```

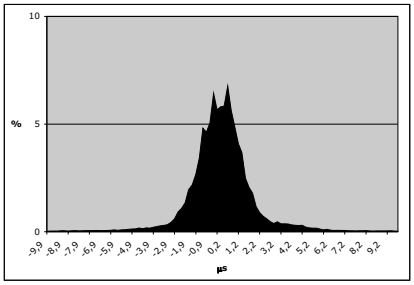
These commands do much of their work in the Linux kernel (file system I/O and network I/O) and should put a lot of stress on the system, in fact the system appeared very sluggish. Histograms showing the distribution of the scheduling jitter is shown in figure 9.1 and 9.2. Because RT-Linux has a few periods (3% and 0.05% respectively for stressed and idle system) with large scheduling jitter figure 9.1 (c) and 9.1 (d) show the histogram with the values above and below ± 10 μs cut off and ignored.



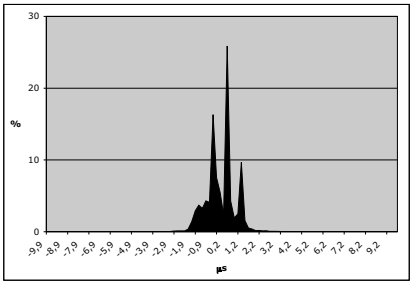
(a) Stressed CPU and I/O.



(b) No stress to CPU or I/O.



(c) Stressed CPU and I/O with the 1597 highest and the 1595 lowest values cut off.

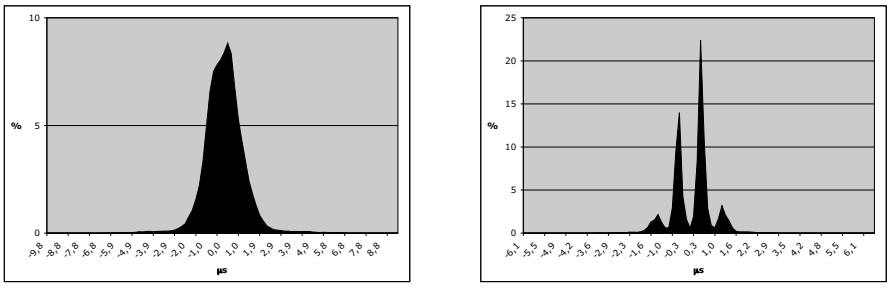


(d) No stress to CPU or I/O with the 25 highest and the 23 lowest values cut off.

Figure 9.1. Scheduling jitter histogram for RT-Linux.

	Max (μs)	Min (μs)	Mean (ns)	Median (ns)	Std. deviation (μs)
RTAI	6.686	-6.207	-1	225	0.7217
RTAI *	9.562	-9.859	-1	-11	1.073
RT-Linux	48.90	-50.24	0	32	0.9292
RT-Linux *	5148.0	-492.6	1024	-32	53.51

Table 9.1. Scheduling jitter properties for 100 000 periods running at 2 kHz. A * denotes stressed CPU and I/O. The table shows the maximum, minimum, mean and median deviation from the reference period (500 μs) as well as the standard deviation.



(a) Stressed CPU and I/O.

(b) No stress to CPU or I/O

Figure 9.2. Scheduling jitter histogram for RTAI.

The data presented in this section show some interesting results. The performance of RTAI is very good with a scheduling jitter of less than 10 μs even on a stressed system. Noticeable is that the performance is relatively unchanged weather the system is idle or stressed, this indicates that the preemption of Linux works well. On average RT-Linux performs well, however for some reason a fraction of of the deadlines are missed by a (relatively) large time. For instance on an idle system RT-Linux performs almost as well as RTAI 99% of the time but the worst case scheduling jitter is 10 times greater than that of RTAI. Most noticeable in table 9.1 is that the worst case scheduling jitter of RT-Linux on a stressed system is as large as 5 ms - a very long time when the period is only 0.5 ms!

Since the scheduling jitter is measured relative to the previous execution of the real-time code one would expect to find the distribution of the scheduling jitter in figure 9.1 and 9.2 to be approximately symmetric around 0 s, this is also the case.

To summarize, *RTAI schedules its events within 10 μs* from the deadline and *RT-Linux schedules its events within 10 μs* from the deadline *97% of the time* but as a worst case scenario *can miss up to 10 events* at 2 kHz.

9.2 Evaluating the Usability

This section provides a summary of my experience when using the three different operating systems. The system running the different kernels was Redhat Linux 7.0 which was installed on the hardware described in appendix B. Since Redhat Linux 7.0 is distributed with the 2.4.18 version of the Linux kernel it was decided that the same version (2.4.18) should be used for the real-time kernels as well. A table of *pro et contra* for each operating system is given in table 9.2, 9.3 and 9.4 at the end of each section.

9.2.1 RT-Linux

At the time the systems were evaluated[1] the most recent stable release of RT-Linux was 3.1. The 3.1 version did not officially support the 2.4.18 kernel, fortunately there was a contribution patch for the x86 architecture available on FSM Labs' ftp server[2]. This patch was successfully applied to the Linux 2.4.18 kernel available at ftp.kernel.org. The process of patching, configuring, compiling and installing the RT-Linux kernel and modules was well documented and was executed flawlessly except that the compilations produced a lot of warnings, however this did not seem to effect the system.

The RT-Linux distribution comes with some examples, most notable the *helloworld* example which is a very clear and simple example. The helloworld example shows, in less than thirty lines of code, how to create and schedule a periodic real-time task.

RT-Linux supports the POSIX 1003.13 "single process/minimal real-time system" interface [25]. The API is documented as Linux manual pages and on-line on FSM Labs' web-page.

Since RT-Linux runs at the same level as the Linux kernel it cannot use any standard Linux libraries, however it is possible, but dangerous, to call functions in the Linux kernel itself. RT-Linux supports both FIFOs and shared memory to allow communication between real-time tasks and Linux processes. Since all real-time processes are actually threads running in the same address space shared memory can also be used to communicate between different real-time tasks (and in fact also Linux kernel threads).

[1] September 2002

[2] RT-Linux 2.4.18 contribution patch ftp.fsmlabs.com:/pub/rtlinux/contrib/gearheart/rt-patch-2.4.18-rtl3.1.tgz

Pro

- Supports the POSIX 1003.13 interface.
- Small and simple API documented as manual pages.
- Support for custom schedulers.
- Installation process is simple and well documented.
- Very good example to get started.
- "Getting started" documentation that covers all basic and important features of RT-Linux.
- Supports the x86, PowerPC, MIPS, AMD Elan NetSC520 and Alpha platforms.

Contra

- No support for real-time in user-space.
- Impossible to call Linux libraries and dangerous to call Linux kernel functions.
- The amount of warnings generated during compilation is disturbing.

Table 9.2. *Pro et contra* for RT-Linux.

9.2.2 RTAI

The most recent stable version of RTAI that was available at the time of evaluation was 24.1.9, however during this project a new version of RTAI was released (24.1.10). The 24.1.10 release include NEWLXRT which is an improved version of the LXRT system described in section 8.2. Other things of interest in the new release was support for writing interrupt-handlers in user space and that the process of making a bootable RTAI floppy is simplified.

Installing RTAI was a breeze, except for one major drawback. RTAI is unable to compile with `gcc` version 3.0 or newer, version 2.96 is not supported either. However this is documented in the accompanying text file `GCC-WARNINGS` which states that: "This is mainly due to bad RTAI inline asm code that needs to be cleaned up". After downgrading gcc to the recommended version 2.95.3 the installation proceeded perfectly.

RTAI comes with almost the same set of examples as RT-Linux, unfortunately the excellent helloworld example is missing.

The documentation for RTAI is what really makes it stand out, there is well written and useful documentation available both for those who are getting started and for those who really are interested in using the full capabilities of the system.

Real-time tasks running under RTAI are Linux kernel threads and hence do not have access to Linux libraries or, at least not safely, Linux system calls. However there is an interesting addition to RTAI called LXRT (described in section 8.2) that allows kernel modules written for RTAI to be recompiled into user space. Especially the NEWLXRT provided in version 24.1.10 have a symmetric API that may be used to run hard real-time tasks in both kernel space and user space. LXRT/NEWLXRT provides means to call standard Linux library functions in real-time tasks, however this makes the task soft real-time. The advantage of being able to move between user space and kernel space can help shorten development times by doing most of the development in user space and only moving to the kernel for maximum performance.

Pro

- Powerful and well documented API.
- POSIX 1003.1c support, limited POSIX 1003.1b support.
- Support for different real-time kernels.
- Support for custom schedulers.
- Dynamic memory allocation using memory pools.
- Installation process is easy and well documented.
- Supports the x86, PowerPC, MIPS, m68k and ARM platform.
- LXRT/NEWLXRT allows development of real-time processes in a uniform environment inside and outside of the kernel.
- Easy to debug when developing in LXRT since gdb is available.
- When soft real-time is required Linux libraries can be accessed when using LXRT.

Contra

- Unable to compile with more recent versions of `gcc`.

Table 9.3. *Pro et contra* for RTAI.

9.2.3 KURT

The KURT distribution is divided in two separate parts, first there is the patch used to add UTIME and KURT support to the Linux kernel and secondly there is the KURT user space library that provides the real-time programmer with the KURT API. Unfortunately I have been unable to locate any version information in the KURT API package and thus can not tell which version was used[3]. The latest kernel patch at the time was version 2.4.18-beta. Both packages comes with a README file that explains the installation process in a satisfactory manner and both packages were also installed successfully without any problems.

The KURT API package comes with a number of examples, these examples differ from those of RT-Linux and RTAI and are, in general, more complex. Unfortunately there is no documentation describing the examples included in the package. Some of the examples print out a description of their use when run and thus it is possible to determine what they are supposed to do. Several of the examples appear to be broken and even tough they run they produce strange or incorrect results. Some of the more striking and annoying bugs in the examples are the disability to run a periodic task using the KURT periodic mode (which was discovered to be broken, see section 9.1) as well as the bug in the `make_binary_sched` example that does not allow a schedule to be repeated, thus making it difficult to run an infinite periodic task.

The main documentation for KURT is in the form of a user-manual[4]. The manual in itself is not bad, on the contrary, it is an extensive document that covers many aspects of KURT. The design of KURT with its explicit schedule policy, numerous operation modes, awkward API and some weird special cases still makes it hard to get started with KURT. The manual suffers from some of the same problems as the examples i.e. many code listings contain errors. I can only guess that this is because the manual and examples are not in sync with the KURT release.

Overall my experience with KURT was not a pleasant one. I never got anything to work exactly the way I wanted or expected. In my opinion KURT is not finished and to me KURT seams like a painting made by hundreds of artists each using his own bush and not caring to much about the others. During the evaluation of KURT these issues where briefly discussed on the KURT mailing-list. In this discussion it was stated that "This is not finished work. It seems as if it gets worked on when some grad student needs to get his thesis finished and is left to rot in the in betweens." [22], this remark was confirmed by Dr. Niehaus (one of the lead developers behind KURT) and is pretty much aligned with my own conclusion about KURT.

[3]The KURT API package was downloaded from the KURT web-site http:///www.ittc.ku.edu/kurt/ on September 11 2002

[4]KURT user manual http://www.ittc.ku.edu/kurt/papers/user-manual-DRAFT.pdf

Pro

- A promising user-manual.
- Installation process is easy and well documented.
- Real-time processes run above the Linux kernel and thus have access to all the standard features of Linux, including libraries (but are also soft real-time).
- Easy to debug since the real-time process runs above the kernel and allows the use of standard debugging tools such as gdb.
- Support for dedicated, kernel level, real-time tasks using the focused mode.
- Supports all architectures that have a the same kind of TSC as the Pentium processor.

Contra

- Broken periodic mode.
- Errors in examples and documentation.
- Not POSIX compliant.
- Many different modes of operation makes it hard to get started.
- Explicit scheduling makes it harder to get started.
- When run as a firm RTOS in focused mode no Linux applications are allowed to run, practically disabling Linux.
- Not stable.

Table 9.4. *Pro et contra* for KURT.

9.3 Summary

Below the topics of table 7.1 are repeated and each topic is provided with a comment for each operating system tested.

- **Ease of installation**
  - RT-Linux - Compiles without modifications but with a lot of warnings. Installation process is well documented.

- RTAI - Compiles without modifications with the right compiler, unable to compile with gcc 3.x compilers. Installation process is well documented.
- KURT - Compiles without modifications and the installation process is well documented. No version information in the KURT API package.

- **Documentation**
  - RT-Linux - Good documentation of functions in HTML or man-page format. Good set of examples that makes it easy to get started.
  - RTAI - Well documented API.
  - KURT - Well documented but documentation contains errors. Many broken and/or unexplained examples.

- **API**
  - RT-Linux - The API is kept small for simplicity, but nothing is really missing. Conforms to the POSIX 1003.13 standard.
  - RTAI - Good consistent API contains more functions than RT-Linux. If this is good or bad depends on the use. POSIX.1c compliant, limited POSIX.1b compatibility.
  - KURT - Large non standard API provides a lot of functionality.

- **Interaction with Linux**
  - RT-Linux - RT tasks run in the Linux kernel address space and can not safely use kernel functions. Linux libraries are unreachable. Shared memory and FIFOs are available for IPC.
  - RTAI - RT tasks run in the Linux kernel address space and can not safely use kernel functions. If the LXRT package is used, user space real-time applications is possible. With LXRT Linux libraries are reachable if only soft real-time requirements are needed. Shared memory, FIFOs and mailboxes are available for IPC.
  - KURT - When only soft real-time is required all Linux libraries are reachable, when in focused mode the programmer can not safely use any Linux services.

- **Development and debugging tools**
  - RT-Linux - Standard Linux kernel development and debugging tools.
  - RTAI - Standard Linux kernel development and debugging tools, gdb is available when LXRT is used.
  - KURT - Standard Linux development and debugging tools such as gcc and gdb are available.

- **Hardware support**
  - RT-Linux - Supports the x86, PowerPC, MIPS, AMD Elan NetSC520 and Alpha platforms.
  - RTAI - Supports the x86, PowerPC, MIPS, m68k and ARM platform.
  - KURT - Supports all architectures that have a the same kind of TSC as the Pentium processor.

9.4 Choosing an OS and Implementing the Control System

The OS that was chosen to implement the PUMA 560 control system was RTAI. KURT was out of the question because it was not reliable enough. RT-Linux and RTAI have roughly the same performance when it comes to scheduling jitter. What made RTAI the winner was the fact that it supports real-time applications in user space with the LXRT package, it has a more modular and extensible design with support for custom kernels and dynamic memory allocation. The implementation of the control system is described in further detail in another (not yet finished) master thesis, however section 9.4.1 provides an overview of the control system.

9.4.1 Control System Overview

To control the PUMA 560 a ServoToGo[5] card was used. This card provides analog-to-digital (AD) and digital-to-analog (DA) converters that are connected to the PUMA 560 and its power amplifier (PA). Thus the lowest level of the control system is a Linux driver for the ServoToGo card (`stg.o` in figure 9.3). This driver provides basic functionality such as setting joint torque and reading encoders and potentiometers. On top of this is the actual real-time module (`pumaCtrl.o` in figure 9.3) that runs two threads. One thread is the PID (**P**roportional, **I**ntegral, **D**erivative) controller and the other thread is a communication thread used to communicate with the user level application throughout the interface defined by the `PumaClient` class (`PumaClient` in figure 9.3). The details of `pumaCtrl.o` can be seen in figure 9.4. This control system has been implemented on the system described in appendix B and works well. The PID controller runs at a frequency of 1 kHz.

The `stg.o` module provides the following low-level interactions (along with some additional convenience functions) with the PUMA 560 and its PA:

- Toggle breaks.
- Toggle power.
- Set joint torque.

[5]Servo To Go, Inc http://www.servotogo.com

- Set encoder values.
- Read potentiometer and encoder values.

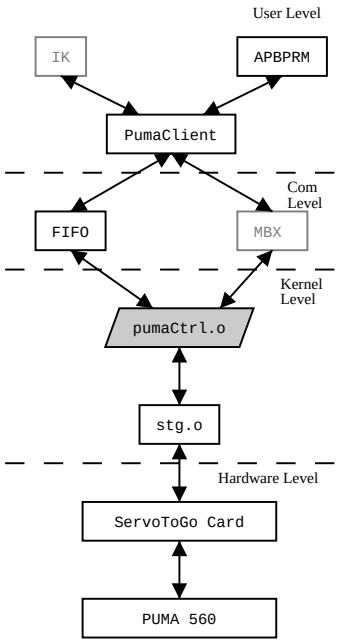


Figure 9.3. Control system block diagram.

The real-time module `pumaCtrl.o` consists of two parts, running in separate threads (figure 9.4). First there is the actual PID controller that uses the services provided by `stg.o` to retrieve information about the state of the PUMA and, after calculating a suitable torque according to the control law used, set the joint torques. The other thread is the communications thread that handles requests from user space applications. The communications thread works in the following way:

- If there are no messages available the thread may block for up to 10 ms.
- If there is one or more messages available in the message queue, they are received and handled in turn.
- When a message is received the communications thread locks access to the reference values by entering a mutex region. While in the mutex region, the communications thread may change the reference values of the PID controller or interact directly with the `stg.o` module (which is made thread safe internally) for operations such as turning power of or brakes on. Next, the communications thread leaves the mutex region.

- If the message requires a response, the response is put into a FIFO buffer which is connected to the user space application. Finally the communications thread returns to its original state, waiting for a new message.

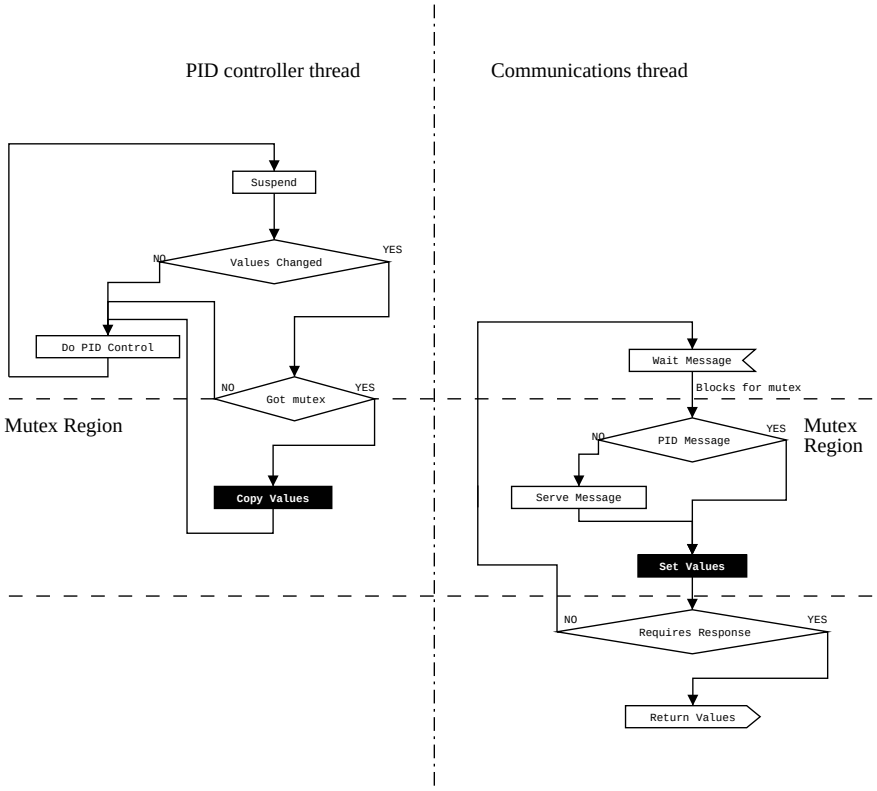


Figure 9.4. Internals of `pumaCtrl.o`.

The `PumaClient` module in figure 9.3 is the interface provided to user space applications. It is implemented as a C++ class and provides (among others) the following functions:

- Set and read the encoder values.
- Read the potentiometers.
- Set the reference value for each joint (i.e. move a joint to a specific angle).
- Read the current angles, angular velocities and angular acceleration of each joint.
- Toggle brakes and power.
- Calibrate the arm to provide absolute joint angles (instead of relative which occurs when the arm is not calibrated).

Chapter 10

Results and Conclusions - RTOS

Of the three real-time operating-systems evaluated in this thesis it is clear that RT-Linux and RTAI can perform the task of a manipulator controller well, however KURT is *not* finished work. The scheduling performances of RT-Linux and RTAI are similar, however RTAI appears to perform better on a stressed system with a maximum deviation of 10 μs (opposed to 5 ms on RT-Linux) from the reference period of 500 μs on the test system.

RTAI has the advantage of being a "true" open-source effort, begun at a university and currently developed by a relatively large number of people around the world. RT-Linux on the other hand is mainly developed by FSM Labs, that offers support, which can be good for corporate users. RTAI also offers the LXRT module which could potentially be a great advantage since it allows users to write both soft and hard real-time applications in both user space and kernel space using a symmetric API, eliminating the need for real-time "inside" of Linux (such as KURT). And of course having one OS and API for all your real-time needs is advantageous.

KURT was a bit of a disappointment, however the idea behind KURT is not actually that bad, and I could actually imagine the benefits of the UTIME package (microsecond timer resolution) in the "real" Linux kernel itself. What KURT would need is some serious clean up. Features that do not work should be stripped out and reimplemented. One way to accomplish this would be to try to gather a larger number of developers from outside KU, building a strong open-source community around KURT. This is what the RTAI project has done recently and it has taken on a more constant development pace, reducing the sporadic behavior that is still apparent with KURT.

References

[1] R. Bohlin, *Robot Path Planning*. PhD thesis, Chalmers university of technology, Göteborg university, 2002.

[2] J. Barraquand, L. Kavraki, J. Latombe, T. Li, R. Motwani, and P. Raghavan, *A random sampling scheme for path planning* in *Robotics Research* (G. Giralt and G. Hirzinger, eds.), pp. 249–264, Springer, 1996. URL http://citeseer.nj.nec.com/barraquand96random.html.

[3] D. Hsu, L. Kavraki, J. Latombe, R. Motwani, and S. Sorkin, *On finding narrow passages with probabilistic roadmap planners* in *Robotics: The algorithmic perspective* (L. E. K. P. K. Agrawal and M. Mason, eds.), (Natick, MA), pp. 141–153, A.K. Peters, 1998. URL http://citeseer.nj.nec.com/article/hsu98finding.html.

[4] J.-C. Latombe, *Robot Motion Planning*. Kluwer Academic Publishers, 1991.

[5] R. C. Gonzalez and R. E. Woods, *Digital Image Processing*. Prentice-Hall, Inc., 2001.

[6] L. Kavraki, P. Svestka, J.-C. Latombe, and M. Overmars, *Probabilistic Roadmaps for Path Planning in High-Dimensional Configuration Spaces* Tech. Rep. CS-TR-94-1519, 1994. URL http://citeseer.nj.nec.com/kavraki96probabilistic.html.

[7] L. Kavraki and J. Latombe, *Randomized preprocessing of configuraton space for fast path planning* in *IEEE Int. Conf. on Robotics and Automation*, 1994.

[8] Overmars and Svestka, *A Probabilistic Learning Approach to Motion Planning* in *Algorithmic Foundations of Robotics, The 1994 Workshop on the Algorithmic Foundations of Robotics, A. K. Peters* (Goldberg, Halperin, Latombe, and Wilson, eds.), 1995. URL http://citeseer.nj.nec.com/overmars94probabilistic.html.

[9] S. A. Wilmarth, N. M. Amato, and P. F. Stiller, *Motion Planning for a Rigid Body Using Random Networks on the Medial Axis of the Free Space* in *Symposium on Computational Geometry*, pp. 173–180, 1999. URL http://citeseer.nj.nec.com/wilmarth99motion.html.

[10] C. I. Connolly and R. A. Grupen, *Applications of Harmonic Functions to Robotics* Tech. Rep. UM-CS-1992-012, Computer and Information Science Department, 1992. URL http://citeseer.nj.nec.com/article/connolly93applications.html.

[11] E. Rimon and D. E. Koditschek, *Exact Robot Navigation Using Artificial Potential Functions IEEE on Robotics and Automation*, vol. 8, pp. 501–518, October 1992.

[12] Y. K. Hwang and N. Ahuja, *Gross Motion Planning - A Survey ACM Computing Survey*, vol. 24, pp. 219 – 291, September 1992.

[13] W. E. Boyce and R. C. DiPrima, *Elementary Differential Equations and Boundary Value Problems.* John Wiley & Sons, Inc, sixth ed., 1997.

[14] T. Erikson, H. Christiansson, E. Lindahl, J. Linde, L. Sandberg, and M. Wallin, *Fysikens Matematiska Metoder.* Teoretisk Fysik, KTH, third ed., 2001.

[15] P. Svestka, *On probabilistic completeness and expected complexity of probabilistic path planning* tech. rep., Utrecht University: Information and Computing Sciences., 1996. URL http://citeseer.nj.nec.com/article/svestka96probabilistic.html.

[16] A. M. Tenenbaum, Y. Langsam, and M. J. Augenstein, *Data Structures Using C.* Prentice-Hall, Inc, 1990.

[17] S. Russell and P. Norvig, *Artificial Inteligence: A Modern Approach.* Prentice-Hall, Inc, 1995.

[18] T. Glad and L. Ljung, *Reglerteknik - Grundläggande Teori.* Studentlitteratur, Lund, 1989.

[19] *Free On-Line Dictionary Of Computing* URL http://wombat.doc.ic.ac.uk/foldoc/.

[20] M. J. Bach, *The Design of the UNIX Operating System.* Prentice-Hall, Inc, 1986.

[21] A. S. Tanenbaum, *Modern Operating Systems.* Prentice-Hall, Inc, 2nd ed., 2001.

[22] F. M. Proctor, *Meassuring Performance in Real-Time Linux* tech. rep., National Institute of Standards and Technology, 2001. URL ftp://ftp.realtimelinuxfoundation.org/pub/events/rtlws-2001/proc/ko3-proctor.pdf.

[23] M. Barabanov, *A Linux-based Real-Time Operating System* Master's thesis, New Mexico Institute of Mining and Technology, 1997.

[24] FSM Labs, Inc., *Getting Started with RTLinux*, 04 2001.

[25] J. Nilsson and D. Rytterlund, *Modular Scheduling in Real-Time Linux* Master's thesis, MdH, Mälardalen University, 2000.

[26] P. Mantegazza, *Dissecting DIAPM RTHAL-RTAI* URL http://www.aero.polimi.it/~rtai/documentation/articles/paolo-dissecting.html.

[27] *RTAI Programming Guide 1.0* 2000. URL http://www.aero.polimi.it/~rtai/documentation/reference/rtai_prog_guide.pdf.

[28] *Some Experiences in fast hard-real time control in user space with RTAI-LXRT (!)*, 2000. URL http://www.aero.polimi.it/~rtai/documentation/conferences/App_Orlando00.pdf.

[29] D. Niehaus, W. Dinkel, and S. B. House, *Effective Real-Time System Implementation with KURT Linux* tech. rep., Information and Telecommunication Technology Center, Electrical Engineering and Computer Science Department, Univerisy of Kansas.

[30] R. Hill, B. Srinivasan, S. Pather, and D. Niehaus, *Temporal Resolution and Real-Time Extensions to Linux* tech. rep., Information and Telecommunication Technology Center, Electrical Engineering and Computer Science Department, June 1998.

[31] S. Balaji, *A Firm Real-Time System Implementation using Commercial Off-The-Shelf Hardware and Free Software* Master's thesis, University of Kansas, 1998.

[32] *Kurt mailing list archive Oct 2002* URL http://www.ittc.ku.edu/~majord/linux-kurt/200210/.

[33] *Kurt mailing list archive May 2002* URL http://www.ittc.ku.edu/~majord/linux-kurt/200205.

Part III

Appendix

Appendix A

Code listings

A.1 RT-Linux implementation of algorithm 2

```
/*************************************80************************************/
/* Use tab width = 5, indent level width = 5, and a fixed-font in a >= 80
 * character wide window to view this file.
 */

/* Tests the performance of RT-Linux in periodic mode.
 */

/*
 * rtl.c
 * RT_Eval
 *
 * Created by Daniel Aarno on Sep 23 2002.
 * Copyright (c) 2002 Daniel Aarno, Andreas Ragnerstam.
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
 */

//Usuall kernel/module stuff
#include <linux/kernel.h>
#include <linux/module.h>

//So we can use RT-Linux
#include <rtl.h>
#include <time.h>
#include <pthread.h>
#include <rtl_fifo.h>

//We're doing this for everyone
MODULE_LICENSE("GPL");

//No one should call us
EXPORT_NO_SYMBOLS;

//The number of fifos to allocate and deallocate at startup
#define kNrFifos 1

//The default fifo buffer size (500k)
//This will fit the entire test, so there is no risk of cat beeing too slow
//under heavy load
#define kDefaultFifoSize (1 << 19)

//The period at which to run (500 us)
```

```
#define kPeriod 500000

//The number of times to run (100000 * 500 us = 50s)
#define kLoops 100000

///The fifo type ID
enum {
     kDataOut = 0,
     kDataIn,
     kCtrl
};

///Pointer to the RT-task to schedule
pthread_t thread;

///The number of fifos actually allocated (0 means 1 fifo allocated)
static int nrFifos = -1;

/**
* Meassure the relative (to previous run) error of the RT-scheduler.
* This function tries to get scheduled kLoops times at the periodic
* rate specified by kPeriod.  It then meassures the difference between
* the current time and the time it was supposed to run acording to the
* last run.  It finally puts this value (that should be close to zero) in the
* FIFO coresponding to kDataOut.
* @param n Ignored
*/
static void Measure(int n)
{
     long long t, prev=0;
     long dt, i;

     rtl_printf("Starting RT-Linux meassurement...\n");
     for(i = 0; i < kLoops; i++) {
          pthread_wait_np(); //Wait to be scheduled
          t = gethrtime(); //Get the time when we are switched in
          if(!prev) { //If it's the first time, assume no fault
               prev=t-kPeriod;
          }

          //Calc.  the error with respect from LAST run and put it in the FIFO
          //Note this is different from the cumulative error from
          //t - start + i*kPeriod
          dt=t - prev - kPeriod;
          rtf_put(0,&dt,sizeof(dt));
          prev=t; //Store current switched-in time as the previous time
     }

     rtl_printf("RT-Linux meassurement completed");

     while(1) { //wait here here until someone kills us
          pthread_wait_np();
     }
}

void * start_routine(void *arg)
{
     struct sched_param p;

     //We're really interested in good performance
     p.sched_priority = sched_get_priority_max(SCHED_FIFO);

     //Set the scheduing for our RT-thread
     pthread_setschedparam (pthread_self(), SCHED_FIFO, &p);

     //Set the RT-thread to be awoken at a periodic rate
     pthread_make_periodic_np (pthread_self(), gethrtime() + 1000000000, kPeriod);

     //Call the Meassurement function (in this thread of execution)
     Measure(1);

     return 0;
}

int init_module(void)
{
     long i=0;
     long size;

     //Create fifos
     for(i = 0; i < kNrFifos; i++) {
          size = rtf_create(i, kDefaultFifoSize);
          if(size < 0) {
               rtl_printf("Error opening fifo\n");
               return size;
```

```
          }

          nrFifos = i;
     }

     //Create a new thread of execution (start_routine)
     return pthread_create (&thread, NULL, start_routine, 0);
}

void cleanup_module(void)
{
     //Return the FIFOs we allocated
     for(;nrFifos >= 0; nrFifos--)
     {
          rtf_destroy(nrFifos);
     }

     //Delete our thread in a safe way
     pthread_delete_np (thread);
}
```

A.2 RTAI implementation of algorithm 2

```
/************************************80************************************/
/* Use tab width = 5, indent level width = 5, and a fixed-font in a >= 80
 * character wide window to view this file.
 */

/* Tests the performance of RTAI real-time Linux in periodic mode.
 */

/*
 * rtai.c
 * RT_Eval
 *
 * Created by Daniel Aarno on Sep 18 2002.
 * Copyright (c) 2002 Daniel Aarno, Andreas Ragnerstam.
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
 */

//Usuall kernel/module stuff
#include <linux/kernel.h>
#include <linux/module.h>

//So we can use RTAI
#include <rtai.h>
#include <rtai_sched.h>

//We're doing this for everyone
MODULE_LICENSE("GPL");

//No one should call us
EXPORT_NO_SYMBOLS;

//The number of fifos to allocate and deallocate at startup
#define kNrFifos 1

//The default fifo buffer size (500k)
//This will fit the entire test, so there is no risk of cat beeing too slow
//under heavy load
#define kDefaultFifoSize (1 << 19)

//The period at which to run (500 us)
#define kPeriod 500000

//The number of times to run (100000 * 500 us = 50s)
#define kLoops 100000
```

```
///The fifo type ID
enum {
     kDataOut = 0,
     kDataIn,
     kCtrl
} FifoType;

///Pointer to the RT-task to schedule
static RT_TASK theTask;

///The number of fifos actually allocated (0 means 1 fifo allocated)
static int nrFifos = -1;

/**
* Meassure the relative (to previous run) error of the RT-scheduler.
* This function tries to get scheduled kLoops times at the periodic
* rate specified by kPeriod.  It then meassures the difference between
* the current time and the time it was supposed to run acording to the
* last run.  It finally puts this value (that should be close to zero) in the
* FIFO coresponding to kDataOut.
* @param n Ignored
*/
static void Meassure(int n)
{
     long long t, prev = 0;
     long dt, i;

     printk("Starting RTAI meassurement...\n");
     for(i = 0; i < kLoops; i++) {
          //Wait to be scheduled
          rt_task_wait_period();
          //Get the time when we are switched in
          t = count2nano(rdtsc());
          //If it's the first time, assume no fault
          if(!prev) {
               prev = t - kPeriod;
          }

          //Calc.  the error with respect from LAST run and put it in the FIFO
          //Note this is different from the cumulative error from
          //t - start + i*kPeriod
          dt = t - prev - kPeriod;
          rtf_put(kDataOut, &dt, sizeof(dt));

          //Store current switched-in time as the previous time
          prev = t;
     }

     printk("RTAI meassurement completed!\n");
     while(1) { //wait here here until someone kills us
          rt_task_wait_period();
     }
}

/**
* This function currently does nothing, it is called before the RT-Task
* resumes its execution after a call to rt_task_wait_period();
*/
void sig(void)
{
}

int init_module(void)
{
     int i, size;
     RTIME tick_period;

     //Create fifos
     for(i = 0; i < kNrFifos; i++) {
          size = rtf_create(i, kDefaultFifoSize);
          if(size < 0) {
               printk("Error opening fifo\n");
               return size;
          }
          nrFifos = i;
     }

     printk("%d fifos allocated", nrFifos + 1);
     //Set the timer mode
     rt_set_oneshot_mode();
     //Initialize the RT-task
     rt_task_init(&theTask, Meassure, 0, 1 << 12, 0, 0, sig);
     //Start the timer at our specified period
     tick_period = start_rt_timer(nano2count(kPeriod));
     //Make our task obey the periodic scheduling, and start it in one second
```

```
     rt_task_make_periodic(&theTask,
               rt_get_time() + nano2count(1000000000), tick_period);
}

void cleanup_module(void)
{
     //Return the FIFOs we allocated
     for(;nrFifos >= 0; nrFifos--) {
          rtf_destroy(nrFifos);
     }

     //Stop the timer
     stop_rt_timer();
     //Kill our RT-task
     rt_task_delete(&theTask);
}
```

Appendix B

System Specification

- CPU: Pentium II (Deschutes) @ 400.917 MHz
- Bogomips : 801.17
- Chipset: Intel Corp. 82443BX/ZX/DX Host bridge/controller[1] (rev 03).
- STGII-8: 8 Axis Model II ISA Card (ServoToGo)

[1]Datasheet available at http://www.intel.com/design/chipsets/datashts/29063301.pdf

Appendix C

Detailed Simulation Results

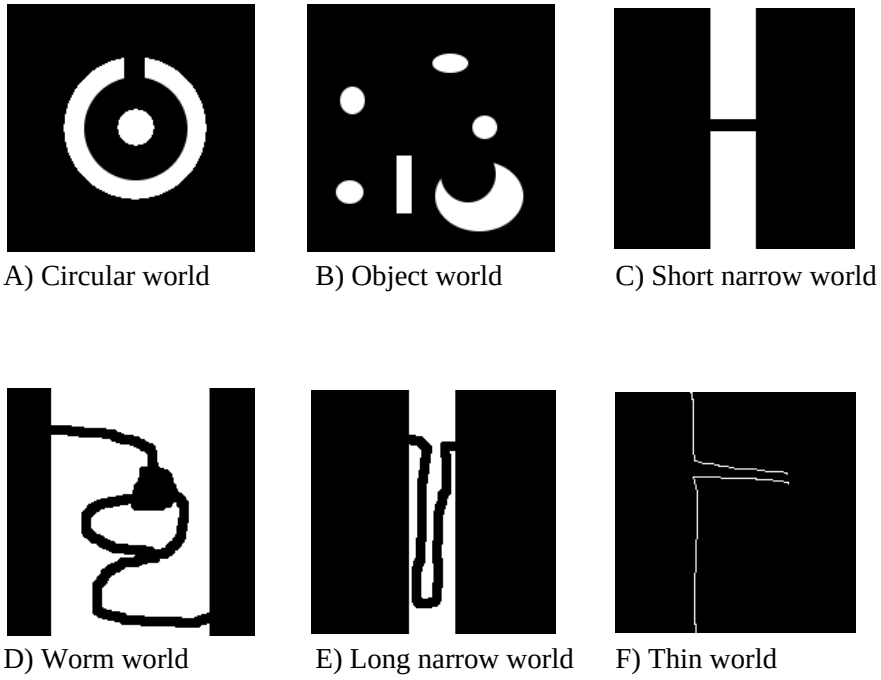
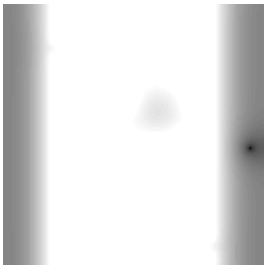


Figure C.1. The different worlds used to evaluate the performance of APBPRM.

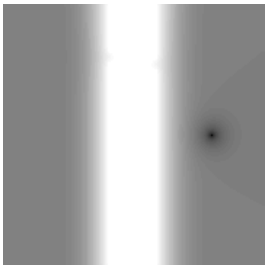
(a) Circular world

(b) Object world

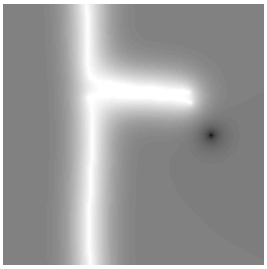
(c) Short narrow world



(d) Worm world

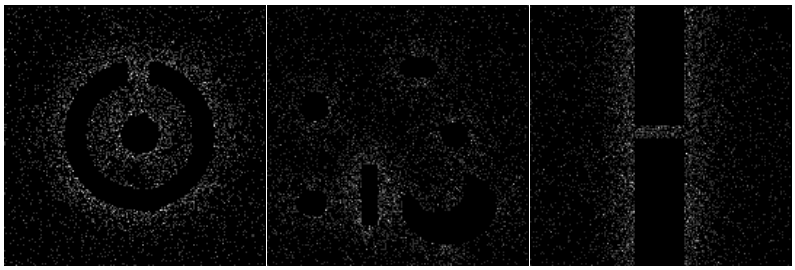


(e) Long narrow world

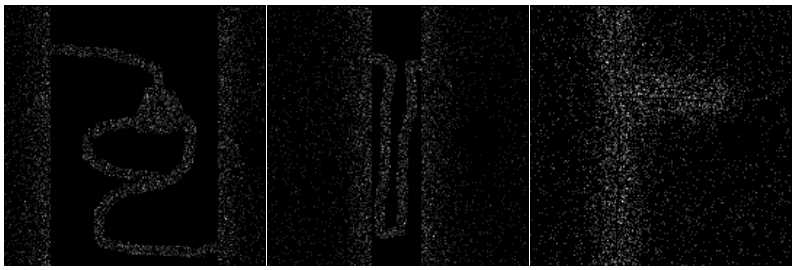


(f) Thin world

Figure C.2. Gray scale hight map of the partial solution to Laplace's equation (ϕ_{100}) for the worlds in figure C.1. Brighter color indicates higher potential.

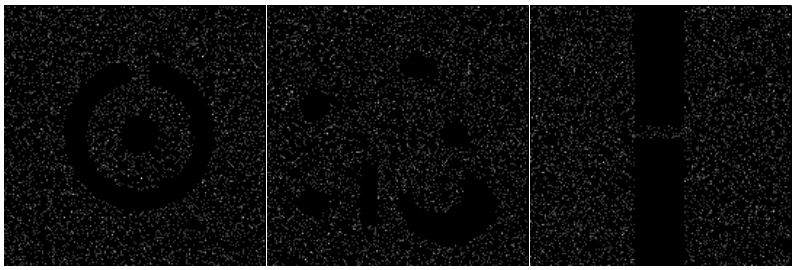


(a) Circular world (b) Object world (c) Narrow world



(d) Worm world (e) Long narrow world (f) Thin world

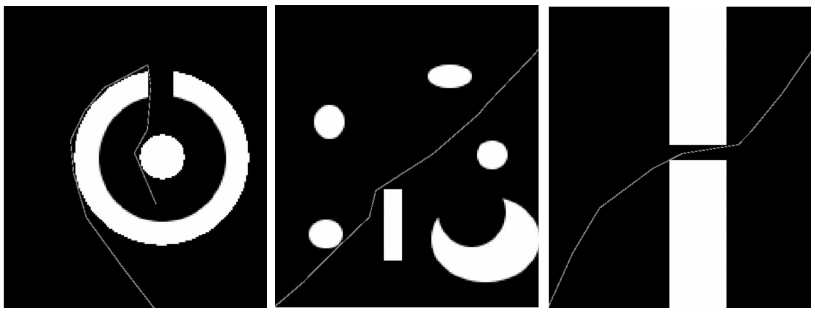
Figure C.3. Distribution density of 5000 nodes for APBPRM.



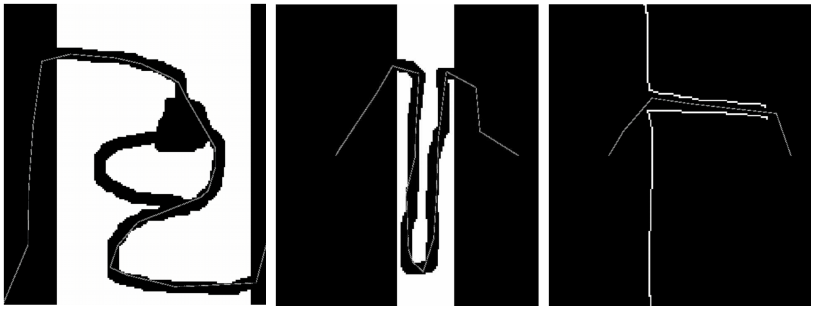
(a) Circular world (b) Object world (c) Narrow world

(d) Worm world (e) Long narrow world (f) Thin world

Figure C.4. Distribution density of 5000 nodes for uniformly sampled PRM.



(a) Circular world (b) Object world (c) Narrow world



(d) Worm world (e) Long narrow world (f) Thin world

Figure C.5. Example paths generated by the APBPRM planner.

# of nodes in roadmap	100	250	500	750
Graph build time (ms)	2	17	103	373
Graph search time (ms)	32	1319	22030	104879
Collision check time (ms)	4	14	64	183
Path Length (C-space)	1418	1380	1420	1504
# of edges in roadmap	1219	7702	30768	64370
# of collision checks	45.4	331	1268	2606
# of points checked	1450	3983	10416	19309
Probability of failure (%)	27	1	0	0

# of nodes in roadmap	100	250	500	750
Graph build time (ms)	3	25	165	431
Graph search time (ms)	141	5668	76776	239039
Collision check time (ms)	11	46	240	435
Path Length (C-space)	1394	1487	1664	1699
# of edges in roadmap	1735	10908	38513	64076
# of collision checks	188	1197	3945	5764
# of points checked	3054	11146	31582	48066
Probability of failure (%)	4	0	0	0

Table C.1. Detailed simulation results for world A. Top with uniform sampling, bottom with biased sampling.

# of nodes in roadmap	100	250	500	750
Graph build time (ms)	3	18	106	360
Graph search time (ms)	11	82	779	2648
Collision check time (ms)	4	4	5	10
Path Length (C-space)	1212	1154	1152	1190
# of edges in roadmap	1292	7789	31382	61620
# of collision checks	9	16	37	53
# of points checked	1190	1495	1941	2409
Probability of failure (%)	9	0	0	0

# of nodes in roadmap	100	250	500	750
Graph build time (ms)	3	24	144	393
Graph search time (ms)	21	295	2252	6178
Collision check time (ms)	4	8	12	18
Path Length (C-space)	1218	1180	1180	1176
# of edges in roadmap	1630	10229	35287	62142
# of collision checks	21	51	100	125
# of points checked	1195	2129	3329	3758
Probability of failure (%)	29	0	0	0

Table C.2. Detailed simulation results for world B. Top with uniform sampling, bottom with biased sampling.

# of nodes in roadmap	100	250	500	750
Graph build time (ms)	3	18	117	392
Graph search time (ms)	6	95	1296	5611
Collision check time (ms)	1	4	7	18
Path Length (C-space)	1241	1259	1241	1238
# of edges in roadmap	1349	8233	33261	63013
# of collision checks	5	18	59	115
# of points checked	597	1504	2589	4175
Probability of failure (%)	59	19	3	0

# of nodes in roadmap	100	250	500	750
Graph build time (ms)	2	21	147	392
Graph search time (ms)	21	788	10067	28741
Collision check time (ms)	5	10	49	61
Path Length (C-space)	1270	1243	1255	1295
# of edges in roadmap	1519	9567	36335	6159
# of collision checks	21	146	440	597
# of points checked	1333	3964	9011	11395
Probability of failure (%)	31	3	0	0

Table C.3. Detailed simulation results for world C. Top with uniform sampling, bottom with biased sampling.

# of nodes in roadmap	100	250	500	750
Graph build time (ms)	3	19	118	381
Graph search time (ms)	15	1267	26918	118428
Collision check time (ms)	1	22	121	300
Path Length (C-space)	N/A	2899	2927	3061
# of edges in roadmap	1351	8373	33425	61610
# of collision checks	20	287	1358	2647
# of points checked	119	6094	25539	48697
Probability of failure (%)	100	57	8	0

# of nodes in roadmap	100	250	500	750
Graph build time (ms)	2	19	115	389
Graph search time (ms)	67	3936	67022	267786
Collision check time (ms)	5	58	296	666
Path Length (C-space)	2916	2876	3090	3226
# of edges in roadmap	1360	8481	33035	63625
# of collision checks	86	865	3353	5887
# of points checked	1502	17902	58271	96387
Probability of failure (%)	90	16	1	0

Table C.4. Detailed simulation results for world D. Top with uniform sampling, bottom with biased sampling.

# of nodes in roadmap	100	250	500	750
Graph build time (ms)	2	19	120	382
Graph search time (ms)	27	1278	27020	131846
Collision check time (ms)	1	13	108	255
Path Length (C-space)	N/A	2194	2065	2109
# of edges in roadmap	1344	8337	32924	63290
# of collision checks	36	277	1306	2953
# of points checked	350	3322	20224	53629
Probability of failure (%)	100	98	78	39

# of nodes in roadmap	100	250	500	750
Graph build time (ms)	3	30	177	419
Graph search time (ms)	321	14372	153949	522347
Collision check time (ms)	13	133	584	967
Path Length (C-space)	2065	2105	2209	2275
# of edges in roadmap	1940	12046	37561	62180
# of collision checks	362	2643	7505	11684
# of points checked	3893	35441	109327	168093
Probability of failure (%)	99	32	0	0

Table C.5. Detailed simulation results for world E. Top with uniform sampling, bottom with biased sampling.

# of nodes in roadmap	100	250	500	750
Graph build time (ms)	2	18	112	391
Graph search time (ms)	73	1635	15797	48250
Collision check time (ms)	12	58	134	196
Path Length (C-space)	N/A	5422	5593	5597
# of edges in roadmap	1326	8107	32199	62993
# of collision checks	83	351	789	1069
# of points checked	7218	30722	56935	73172
Probability of failure (%)	100	55	10	2

# of nodes in roadmap	100	250	500	750
Graph build time (ms)	4	34	187	443
Graph search time (ms)	274	5287	39499	114100
Collision check time (ms)	39	105	245	380
Path Length (C-space)	5666	5446	5550	5618
# of edges in roadmap	2124	13175	38673	62731
# of collision checks	262	866	1739	2381
# of points checked	19798	48825	93429	136142
Probability of failure (%)	45	4	1	0

Table C.6. Detailed simulation results for world F. Top with uniform sampling, bottom with biased sampling.

(Note: these tests where performed with a 10 times higher resolution than other tests because otherwise the agent could sometimes penetrate the thin wall because values where truncated rather than rounded.)

Index