CHALMERS

Energy optimisation of an industrial robot using iterative dynamic programming

Master's Thesis in Systems, Control and Mechatronics

Stefan Edmonds

Rasmus Lindberg

Department of Signals and Systems
Division of Automatic Control, Automation and Mechatronics
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden

EX033/2014

Energy optimisation of an industrial robot using iterative dynamic programming

Master's Thesis in Systems, Control and Mechatronics

Stefan Edmonds
Rasmus Lindberg

Department of Signals and Systems
Division of Automatic Control, Automation and Mechatronics
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2014

Energy Optimisation of an Industrial Robot using Iterative Dynamic Programming
Stefan Edmonds, Rasmus Lindberg

Technical report no: EX033/2014
Department of Signals and Systems
Division of Automatic Control, Automation and Mechatronics
Chalmers University of Technology
SE-412 96 Gothenburg
Sweden
Telephone + 46 (0)31-772 1000

Department of Signals and Systems
Gothenburg, Sweden 2014

ABSTRACT

Automated industrial robots have become a common addition to production systems which include monotonous tasks. To maximise product output and profit, it is not uncommon for such systems to run around the clock. Therefore, a small decrease of the energy consumption per robot cycle can make a big difference to the total energy consumed. This project investigates the possibility of saving energy by adjusting the rate of acceleration and speeds of the robot along a given trajectory by utilising a model-free optimisation method. This is achieved by the use of Iterative Dynamic Programming (IDP) and measuring energy data from simulations and has been implemented for two cases.

The first case modifies a trajectory for an ABB industrial robot. The IDP scheme has been implemented as an add-in for ABB RobotStudio, hence the optimisation is performed off-line via simulations. Various methods for the IDP have been tested, including iterative learning control and Grid Constriction (GC). Also, different warm-up procedures to obtain the initial cost data have been included in the tests. The obtained results were compared to a standard path in RobotStudio, and varied from consuming 85 % less energy to consuming 20 % more energy.

Two trajectories with an improved energy consumption were tested on a real ABB robot, the tests revealed that energy could be saved. They also validated that the optimisation can be performed off-line using simulations.

For the second case, a model of a KUKA industrial robot has been utilised to simulate the movement along a trajectory and measure the energy consumption. The entire scheme, including the model provided by KUKA, has been implemented in MATLAB. Two trajectories were optimised, resulting in an energy saving of 6.9 % and 26.2 %.

KEYWORDS: ABB RobotStudio, energy consumption, industrial robots, Iterative Dynamic Programming, KUKA, model-free optimisation.

ACKNOWLEDGEMENTS

First we wish to thank Oskar Wigström, who contributed with many great ideas and excellent supervision throughout the project.

From ABB we would like to show gratitude to Henrik Berlin, Niklas Skoglund and John Wiberg who helped us solve programming related problems during the development of the RobotStudio add-in.

We also wish to acknowledge Per Nyquist from the Product and Production Development department at Chalmers, who showed us how to operate ABB industrial robots and also aided the tests.

Finally, we would also like to acknowledge our examiner Bengt Lennartsson for making the project possible.

Gothenburg, June 2014

Stefan Edmonds & Rasmus Lindberg

CONTENTS

NOTATION

Abbreviations

DP	Dynamic Programming or Dynamic Program
IDP	Iterative Dynamic Programming or Iterative Dynamic Program
GC	Grid Constriction
TCP	Tool Center Point of a robot

Small Letters

h	Sampling time
p	Position
t	Time
u	Control signal
v	Velocity
x	State

Greek Letters

Δ	Step-size in space
γ	Reduction factor

Subscripts

$_0$	Start
$_f$	Final
$_k$	Iteration

Superscripts

*	Optimal

1 INTRODUCTION

Advanced production systems which employ automated industrial robots generally consume a great amount of energy. As such production systems commonly run around the clock, a small decrease of the energy consumption per cycle can make a big difference to the total energy consumed. When following a trajectory, industrial robots typically accelerate as fast as possible to a desired speed and then decelerate rapidly to a standstill when the target has been reached. This project investigates the possibility of reducing the energy consumption by adjusting the rate of acceleration and trajectory speeds. To accomplish this, model-free optimisation methods have been employed which use a cost function obtained from simulations in two cases, ABB RobotStudio and a mathematical model from KUKA.

1.1 Purpose

The purpose of this master's thesis is to minimise the energy consumption an industrial robot consumes while following a fixed given path.

1.2 Objective

The objectives of this project include developing an algorithm which iteratively minimises the consumed energy for a robot following a given fixed path. The algorithm itself should employ model-free optimisation techniques. The term model-free means that the algorithm will not have a model of the robot itself, but instead use external models to measure the energy consumption. This approach is attractive since modelling a 6 degree of freedom robot is a complex task.

The algorithm will be implemented for two applications, or in other words use two external models. The first is RobotStudio, where the algorithm will be included as an add-in coupled with MATLAB functions. The second is a KUKA mathematical model, the algorithm in this case is only developed in MATLAB.

Since cycle times can be critical for production systems, it is imperative that the new trajectory completes the path within the same amount of time. Therefore, it is required that the new trajectory must complete the given path within 5 % of the original target time.

Different methods to obtain the most energy efficient trajectory should be explored and later compared. Also, if possible, tests on a real robot should be conducted to validate the off-line optimisation.

1.3 Scope

The algorithm for RobotStudio will only consider linear movements between two targets. This is to reduce the complexity when dividing the path into segments which will be explained further in Section 3.3.2. Also, during these movements the robot will not carry any loads or perform any other tasks simultaneously such as welding or painting.

1.4 Approach

An optimisation method must be employed to find the most energy efficient trajectory for a given path. Dynamic Programming (DP) has been chosen as a suitable candidate for this application. It is a powerful method which divides a problem into stages, where a series of decisions can be made to minimise a given cost function. In this case, the cost function describes the energy consumption for the stages, and will have to be measured for both RobotStudio and the KUKA model. Since the cost function is initially unknown, it is preferable to run the DP iteratively, which is known as Iterative Dynamic Programming (IDP). A more thorough description of DP and IDP will follow in Section 2.2 and 2.3.

There are several previous projects which have also employed dynamic programming to solve trajectory planning optimisation problems, (Shin and McKay 1986) and (Field and Stepanenko 1996) are two examples.

1.5 Thesis Outline

In Chapter 2, the theory for the methods incorporated into the algorithm is presented. This includes a description of the mathematical model used for trajectory planning and the theory behind DP and IDP.

Chapter 3 presents how various aspects of the project have been implemented. The methods describe how DP and IDP have been utilised to minimise the energy consumption, and how the optimisation techniques have been incorporated into RobotStudio and the KUKA model. Different strategies to obtain measurement data for the cost function are also conveyed. A section describing how a real ABB robot has been utilised in the project has also been included in this chapter.

In Chapter 4 the results for several different cases are presented. These are later discussed in Chapter 5. Chapter 6 concludes the work of the project. Finally, Chapter 7 considers future work of the project.

2 THEORY

This chapter presents the theory behind the techniques utilised in the project. In Section 2.1 the mathematical model used to plan the trajectory is presented. Section 2.2 introduces DP and finally in Section 2.3, methods to iteratively improve the results from the DP are brought up.

2.1 Trajectory Planning

DP requires a simple model to follow a certain trajectory's speed, the distance covered and the time. One way of looking at the trajectory planning problem is to consider the motion along the path as a double integrating process. In that case, one can consider the system in discrete time with the position p_k and velocity v_k as states and the acceleration as the control signal. Using h_k as the sampling time, this can be formulated as

$$\begin{bmatrix} p_{k+1} \\ v_{k+1} \end{bmatrix} = \begin{bmatrix} 1 & h_k \\ 0 & 1 \end{bmatrix} \begin{bmatrix} p_k \\ v_k \end{bmatrix} + \begin{bmatrix} h_k^2/2 \\ h_k \end{bmatrix} u_k. \tag{2.1}$$

This model will ensure an equally spaced time-scale and the position will change according to the applied acceleration, where steps are taken along the time-scale. However, this model can be reformulated to instead taking steps in space. As a result, the model will be controlled by the sampling time. This can be achieved by introducing a varying sampling time h_k, the time state

$$t_{k+1} = t_k + h_k \tag{2.2}$$

and defining the distance step-size

$$\Delta_k = h_k v_k + h_k^2 u_k/2, \tag{2.3}$$

from which u_k, the acceleration, can be solved

$$u_k = \frac{2(\Delta_k - h_k v_k)}{h_k^2} \tag{2.4}$$

and substituted into Equation 2.1. Thus, a new state-space model can be defined as

$$\begin{bmatrix} t_{k+1} \\ v_{k+1} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} t_k \\ v_k \end{bmatrix} + \begin{bmatrix} h_k \\ 2\Delta_k/h_k \end{bmatrix}. \tag{2.5}$$

This model will have the time t_k as its first state, the velocity v_k as the second state, the sampling time h_k as the control signal and Δ_k as the distance in space between the steps. In this case steps are no longer taken along the time, instead steps are taken in space using predefined positions. The latter model is preferred since it is guaranteed that the destination and all steps in between will be reached at the exact position. This is important for the robot application. Note that both models assume constant acceleration. (Wigström *et al.* 2013)

2.2 Dynamic Programming

Dynamic Programming is an optimisation method that is used to solve complex problems by repeatedly solving many subproblems. The problem may be defined by a simple state-space model which is controlled by one or several control signals. To solve the problem, it is generally divided into N segments with N+1 stages where the control signal can be changed to push the state somewhere else. Figure 2.1 shows how a simple problem has been divided into three segments and four stages. Depending on the state in each stage, actions are taken dynamically to minimise or maximise a cost function.

The DP solves the problem by running through the segments once at a time, either backwards from the last segment to the start, or forwards from the first segment to the end, and for every state in each stage calculates the cost of being in that state and proceeding to the next one. Commonly, the DP starts by calculating backwards where a final cost has been defined by a penalty on all the states in the last stage. The final cost typically depends on the desired goal. By calculating backwards, the DP steers the solution to that goal.

A general formulation of an optimisation problem that could be solved using dynamic programming is as follows

$$\begin{aligned} \min_{u(t)} \quad & J(u(t)) \\ s.t. \quad & \dot{x} = f(x(t), u(t)) \\ & x(0) = x_0 \\ & x(t_f) = x_f \\ & x(t) \in \mathcal{X}(t) \\ & u(t) \in \mathcal{U}(t) \end{aligned} \tag{2.6}$$

where

$$J(u(t)) = P(x(t_f)) + \int_0^{t_f} H(x(t), u(t))dt. \tag{2.7}$$

J is the cost function to be minimised and P is a function that describes the final cost. H can be seen as a function that describes the cost to be in a certain state coupled with the cost to go to the next one. The set $\mathcal{X}$ spans the region for state combinations, and the set $\mathcal{U}$ spans the region for the control signals. The final time is t_f, the desired final state is x_f and the starting state is x_0. (Sundstrom and Guzzella 2009)

If an optimal state in the grid is denoted $J^*_{k+1}(x^*(k+1))$, then the Bellman equations state the functional equation for the dynamic program, which in this case takes the DP one step in the backwards direction

$$J^*_k(x(k)) = \quad min_{u(k)} \left[V[x(k), u(k)] + J^*_{k+1}(x^*(k+1))\right] \tag{2.8}$$

where $J_k = J_{tot}$, $V = J_{cost-to-go}$ and $J_{k+1} = J_{cost-at-target}$. This can be performed from any point in the state space.

The procedure for the DP can be explained by a simple example presented in Figure 2.1. The figure depicts a problem with three segments and four stages along the time axis, and a single state x. The objective is to solve

$$\begin{aligned} \min \quad & (x(t_f) - x_f)^2 + \sum u(t)^2 \\ s.t. \quad & x(t+1) = x(t) + u(t) \end{aligned} \tag{2.9}$$

Here it is assumed that the final cost P has been defined as $P(x(t_f)) = (x(t_f) - x_f)^2$ and that $x_f = 0$. So the final cost increases the further the final state is from zero. Also assume that the control signal $u(t)$ can move the state to any of the states in the next segment, and that the cost to go is equal to $(x(t+1) - x(t))^2$.

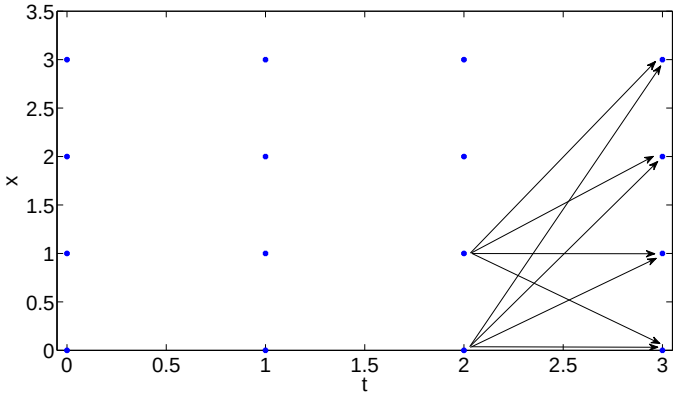

Figure 2.1. Dynamic programming example.

A DP running backwards in this case starts with the states at $t = 2$, and for every state within the grid considers every possible $u(t)$ and the cost of performing that action, and the cost of being at the future state. This procedure is depicted for states $x(2) = 0$ and $x(2) = 1$.

Lets say that the DP starts at the state $x(2) = 0$. Now consider every possible control signal and see which states are reachable. In this example every state in the

next segment, $x(t_f) = 0, 1, 2, 3$ can be reached with the control signals $u = 0, 1, 2$ and 3 respectively. Now consider the cost to go to the next state and the cost of being in the next state. For $u = 0$, the cost to go is 0. The next target $x(tf) = 0$ has, according to the final cost, also a cost of 0. So, the total cost is $J(0) = 0$ for the state $x(2) = 0$.

The next control signal to check for the state $x(2) = 0$ is $u = 1$. The next state after applying this control signal is $x(tf) = 1$. The cost to go is now 1, and the cost to be at the next target according to the final cost is also one. So, the total cost is $J(1) = 2$. In an identical fashion, the costs $J(2) = 8$ and $J(3) = 18$ are obtained. Now it is clear that the optimal control signal for the state $x(2) = 0$ is $u = 0$;

The calculation is repeated for the next state, $x(2) = 1$, where the possible control signals $u = -1, 0, 1$ and 2 are possible. $J(u)$ is built up in this manner until every state has been evaluated. From there, the problem can be solved as an optimisation problem.

To implement a DP, one first has to define a grid for each state and control signal that spans all of the values to consider while running the program. In Figure 2.2, an example where a problem with two states is shown.

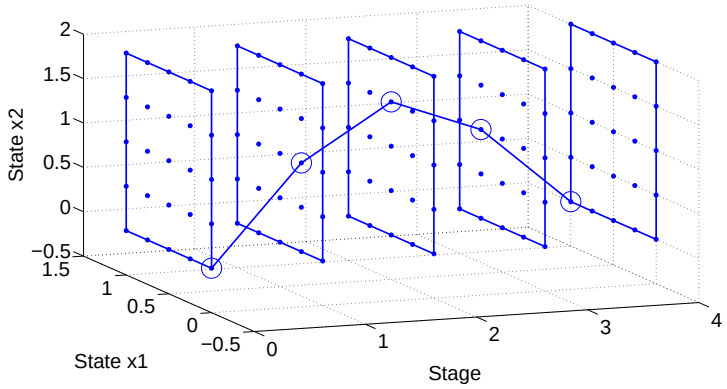

Figure 2.2. The state grid of the DP, which in this example has two states, four segments and five stages. Each stage has its own grid, which is represented by the blue rectangles. The grid includes 25 possible combinations of the states. Each rectangle therefore contains the region where all possible combinations of the two states are included. The line that starts in the first stage and runs to the last represents a possible optimal path.

It is imperative that the grid is large enough to ensure that the optimal solution lies within the grid region. It is also important that the grid has a sufficient resolution, otherwise minima's can be missed and therefore never examined. However, increasing the grid region while retaining the same resolution increases the number of grid points. Too many points will have a dire effect on the computation time, which increases exponentially with the number of points. Therefore, to obtain a satisfying result the grid must be carefully considered. A method presented in Section 2.3 offers a solution to this problem. (Naidu 2003)

2.3 Iterative Dynamic Programming

This section presents how the DP can be run iteratively to obtain improved results. Two methods have been incorporated in this project, iterative learning control and Grid Constriction (GC).

2.3.1 Iterative Learning Control

The key to successfully minimising the energy consumption for a robot trajectory, which is described by the trajectory model 2.5, is to know how much certain robot movements cost in energy. The energy data is obtained by running simulations from an external model. With sufficient data, the DP can select the most energy efficient trajectory. However, the costs are initially unknown; there is no data. Therefore, a scheme similar to iterative learning control can be employed so that with each iteration of the DP, data is added to the cost function which describes the energy costs for movements along that specific trajectory. With each iteration, more is learnt about the robot and the next DP iteration will depend then on the results from all previous iterations. This can be seen in Equation 2.10. (Arimoto *et al.* 1984)

$$u(k+1) = f(x(1), x(2), ..., x(k), u(1), u(2), ...u(k), J(1), J(2), ...J(k)) \qquad (2.10)$$

Equation 2.10 basically expresses that the next control signal is a non-linear function of the previous states, control signal and cost function. To employ such a scheme, the DP must be run iteratively and the results after each iteration must be incorporated into the cost function.

2.3.2 Grid Constriction

As previously explained, the values that the states and control signal can adopt are defined by several grids, one for every state and control signal. The resolution of the grid depends naturally on the span of the minimum and maximum value and the number of points there are in between. Typically, for improved results it is desired to have a high resolution and therefore have many grid points. However, too many grid points can have a devastating effect on the computation time of the IDP. A GC method to reduce the grid region's size after each iteration offers a solution to this problem.

For IDP without GC, the grids for a state or control signal are identical in each segment. This grid configuration is recommended for at least the first few iterations of the IDP with GC, since the optimal trajectory is unknown. However, the resolution of the following iterations can be improved by employing a systematic GC method.

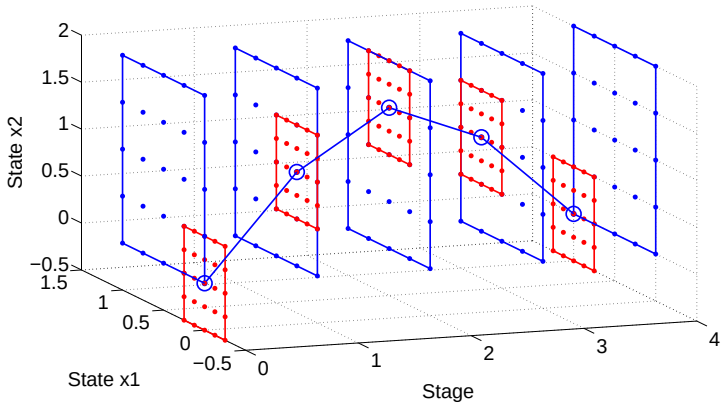

Figure 2.3. The state grids for the IDP after one iteration. The blue rectangles represent the grids/regions for the first iteration. The red rectangles represent the new grids for the second iteration, which are centred around the last found state for that stage.

This method involves for each stage taking the state or control signal from the last optimal trajectory, and creating grids with narrower regions with the optimal state or control signal as the centre point. So each stage has different state and control signal grids.(Luus 2000)

The grid sizes vary then for each iteration after the initial iterations without GC. The first iteration with GC may have a start region, one that is typically smaller than the region for the general first iterations. The grid regions are then iteratively reduced by a reduction factor γ, where $0 < \gamma \leq 1$. Thereby, the resolution is increased without increasing the number of grid points and the computation time is left unaffected.

Figure 2.3 shows a simple case where there are two states. The large blue boxes represent the state grids for the first iteration. Since it is the first iteration they are identical for each segment. The blue line represents an optimal solution found from the first iteration. The red boxes show how the grids for the second iteration may be like, the resolution is clearly improved around the optimal state for each segment. If for example $\gamma = 0.98$, the following iterations will decrease the size of the red boxes by 2 % per iteration. However, the centre point of the following grids may vary, since they depend on the last found trajectory. Also, the number of iterations where the grid is reduced is a design parameter.

This method does introduce a few complications. It is important not to restrict the IDP by selecting a too small starting region size or γ value. In that case, the IDP might get stuck in a local solution and never be able to examine the region where the global optimal trajectory exists.

3 METHOD

In this chapter, methods used to implement the algorithm are presented. The first two sections concern the DP and IDP. The three remaining sections describe the applications where the IDP has been incorporated and/or tested. These include RobotStudio, an ABB robot of type IRB1600 and a KUKA model.

3.1 Dynamic Programming

To find an optimal trajectory for an industrial robot, a cost function is minimised using a DP. This section describes how the DP has been realised.

Following the case presented in Section 2.1, two states form the state grid where the first state is time and the second is the TCP speed. The control signal is the sampling time, precisely as shown in Equation 2.5. Using Equations 2.5 and a discrete version of 2.6, a formulation of the minimisation problem can be defined as shown in Equation 3.1.

$$
\begin{aligned}
\min_{u(t)} \quad & P(g(t_f)) + J_{tot}(h_k) \\
s.t. \quad & \begin{bmatrix} t_{k+1} \\ v_{k+1} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} t_k \\ v_k \end{bmatrix} + \begin{bmatrix} h_k \\ 2\Delta_k / h_k \end{bmatrix} \\
& g(0) = \begin{bmatrix} t_0 \\ v_0 \end{bmatrix} \\
& g(t_f) = \begin{bmatrix} t_f \\ v_f \end{bmatrix} \\
& x(k) \in \mathcal{X}(k) \\
& h(k) \in \mathcal{H}(k)
\end{aligned}
\tag{3.1}
$$

The DP is implemented as a function in MATLAB. By doing so, important inputs to the function can be changed for each iteration. The cost function is one of those inputs that must be updated. The cost function and how it is created is discussed further in Section 3.1.1.

The DP function is designed to receive data for a predefined path that has already been divided into segments. The function returns a solution as an array containing the optimal state and control values to be applied and eventually simulated. The DP works in this fashion for both the RobotStudio and KUKA applications.

3.1.1 Cost Function

The cost function $P(g(t_f)) + J_{tot}(h_k)$ can be divided into three parts:

- $P(g(t_f))$
- $J_{cost-at-target}$
- $J_{cost-to-go}$

$P(g(t_f))$ is the final cost which is used to initiate the DP by setting the cost on the final state. The final cost is formulated as

$$P(g(t_f)) = \alpha(t - t_f)^2 + \beta(v - v_f)^2 \tag{3.2}$$

where α and β are the tunable penalties on the deviation of the desired final time and velocity.

$J_{cost-at-target}$ is the cost to be in a certain state. It depends on both the final cost $P(g(t_f))$ and $J_{cost-to-go}$, and is built up as the DP goes backwards through the segments.

Finally, $J_{cost-to-go}$ is the cost to go from one state to another. In this case, it consists of energy data, and stores for each segment the amount of energy consumed by accelerating with a constant acceleration from a certain start speed to an end speed. This cost can only be obtained by doing simulations to obtain energy data. These simulations hereinafter are referred to as the warm-up simulations. Since it is impossible to have data for every possible situation, this data is used to obtain surface functions or surface fits for every segment. This is achieved by linear interpolation and curvefitting functions generated by MATLAB's function *cftool*. Naturally, this step must be completed before running the DP. Figures 3.5 and 3.10 show examples of curve fits from energy data for RobotStudio and KUKA respectively. The energy data and curve fitting process will be described in more detail for RobotStudio and KUKA in the following sections 3.3.2 and 3.5.

The decision taken in the DP is then based on:

$$J_{tot} = J_{cost-at-target} + J_{cost-to-go}. \tag{3.3}$$

so that the chosen path has the lowest combined cost.

3.2 Iterative Dynamic Programming

IDP techniques have been employed to improve the results of the algorithm. This section presents how IDP has been incorporated into the applications.

This is achieved by showing the work flow of the algorithm.

1. Run warm-up simulations and measure the energy consumption. Save data.
2. Iterate.
   (a) Create/update cost function, i.e. with energy data, create a new surface fit for each segment as described in Section 3.1.1.
   (b) Run the DP to obtain an optimal trajectory.
   (c) Simulate the optimal trajectory and measure the energy consumption.
   (d) Add energy consumption to data from which the cost function is created.

One can see that every time a new solution is obtained from the DP, the energy data for that trajectory is added to the cost function. Hence the cost-function is improved at the region of the solution for each iteration. This was described as iterative learning control in section 2.3. GC can also be implemented, however this is accomplished internally in step 2(b) in the DP function.

3.3 RobotStudio

This section starts by offering a brief introduction to RobotStudio. A description of how RobotStudio has been incorporated into the project then follows, including the different phases of the add-in and some limitations. Finally, the trajectories used to test the algorithm are presented.

3.3.1 Introduction to RobotStudio

RobotStudio is ABB's simulation and off-line programming software for industrial robots.[1] It enables users to define production sequences/tasks and then accurately simulate them. The simulations are realistic since RobotStudio is built on the ABB VirtualController, a copy of the real software that runs the robots. The simulation is controlled by RAPID code, the same programming language used to control the industrial robots.

Before running a simulation, a procedure must first be defined. A simple procedure may involve a path which the robot is to follow. The path itself is determined by instructions to move the robot to specific targets, which are in turn specified by a position, orientation and a robot configuration.

The simplest path requires naturally two targets and one move instruction. A move instruction has several instruction arguments, but for this application it is sufficient

[1] http://new.abb.com/products/robotics/robotstudio

to specify the motion type, speed and zone. In a procedure, a move instruction simply states: move to this target with this speed, motion type and come within at least this specific zone around the target before continuing to the next move instruction.

There also exists an instruction to limit the acceleration and/or deceleration. This instruction is simply included after a specific move instruction, and the acceleration/deceleration limit is included as an input. However, there is no support in RobotStudio to implement a constant acceleration/deceleration. This can only be achieved by having the correct input to the instruction.

Hereinafter, a standard path in RobotStudio is defined as a path with two targets and one move instruction with no acceleration limitations.

Users can define paths either directly in RAPID code, or create one in the virtual workspace. In the virtual workspace the user creates a station, in which a specific robot and the surrounding objects and work items may be included. This way it is easier for the user to understand what the robot is actually doing, since the whole process is visualised.

RAPID code can either be written manually, imported from an external source or automatically generated by RobotStudio. RobotStudio automatically generates code when the user synchronises a station or path from the virtual workspace to the VirtualController. Once the RAPID code has been written or automatically generated and the simulation parameters have been defined, RobotStudio is ready to start running simulations. During a simulation it is possible to sample data from the robot and export it to a text file, or view it in the signal analyser in RobotStudio.

RobotStudio can also be used for on-line controllers, if the controller is connected to the computer via an Ethernet cable. In that case, it is possible to synchronise a path or station to the real controller. RAPID code can also be written directly to the controller. Similarly, during an actual run, data can be sampled and either presented in the on-line signal analyser or exported to a text file.

RobotStudio also offers users the tools to develop an add-in which can access data and functions within RobotStudio. To aid the implementation of such an add-in, ABB provides reference a source[2] which includes the classes, properties and methods of RobotStudio. The add-in is implemented in C#.

3.3.2 The Add-in

The IDP requires cost data and a base from which to iteratively run and simulate solutions. An add-in for RobotStudio has been developed for this purpose. The add-in's tasks include to prepare the path for the IDP, run the warm-up simulations, sample and process the measured data and call the IDP. To accomplish these tasks,

[2]http://developercenter.robotstudio.com/DevCenter.aspx?DevCenter=RobotStudio

the add-in is divided into three stages: a preparation stage, a warm-up stage and an IDP stage.

Preparation stage

Initially the add-in prepares a given path, typically a simple path defined by two targets, by dividing it into segments to, or close to, a specified length. This is achieved by simply iteratively checking the current length between two targets and adding a target in between if the specified length has not been reached. For the path to remain smooth and to ensure that the robot moves identically to the original path, it is imperative that the added target is located and oriented at the exact half-way point between the original targets. The targets are then the stages which split the path into segments, the speed in between is controlled by move instructions and the acceleration/deceleration limitations in the RAPID program.

The segment length can have a significant effect on the results from the IDP. Shorter segments are preferred, since this increases the resolution of the IDP and yields a better result. But shorter segments can be troublesome to implement in RobotStudio, since there is a higher risk that the robot will fail to accelerate from the start-speed to the desired end-speed within the segment distance. If this occurs, the cost function might become corrupt and/or inconsistent. In that case the IDP will not fully function. Also, if the IDP does find a solution, it is unlikely the simulation will be successful since the robot will struggle to complete the acceleration demands within the segments. Shorter segments will furthermore increase the computation time of the IDP. Longer segments on the other hand are easier for RobotStudio to handle, but not suitable for the IDP since the resolution decreases and the path characteristics can change too much in one segment.

Tests revealed that an appropriate segment length lies in the regions of 0.1 m. This gave a sufficient resolution and the simulations were reasonably reliable.

To record the necessary data from the simulations, the add-in must additionally set-up data sinks which sample the speed, energy and next target data. The sinks are activated to record as the simulation commences, and export the data to a text file for each of the data signals. The preparation stage is complete when the modified path has been synchronised to the virtual controller, and the simulation parameters have been set.

Warm-up stage

The warm-up stage consists of performing simulations to generate data for the cost function $J_{cost-to-go}$. To achieve this, several warm-up paths are simulated where the target speeds follow a certain predefined pattern.

The patterns are designed to obtain as much information as possible, from as few

simulations as possible. Several different methods were tested, including a zig-zag pattern, a triangular pattern and a pattern in which a constant speed is chosen. In addition to these, a method that employs a minimisation problem to generate random paths that fulfil certain constraints linked to the given path was also incorporated.

The zig-zag pattern is presented in Figures 3.1 and 3.2. It is divided into different speed step sizes, 400, 200 and 100 mm/s. A pattern with constant speeds is also incorporated into the warm-up after the zig-zag pattern. From Figure 3.2, one can see that it is a thorough method to obtain energy data for different speeds since many speed changes are tested. Also, the number of iterations required for the warm-up depends only on the number of speed jumps to be investigated, not the number of segments. The number of iterations required for the presented example is 63.

There are however a number of drawbacks to this approach. Firstly, the robot could have difficulties following the zig-zag pattern, especially for the higher speeds and with constant acceleration enforced. The consequence of this was inconsistent cost data which resulted in poor results. Additionally, because of these restrictions, the maximum jumps are limited to a modest 400 mm/s. It is not unusual for the robot to accelerate from 0 to 900 mm/s, so this is a major restriction which again hindered the IDP. For these reasons, this approach was abandoned.

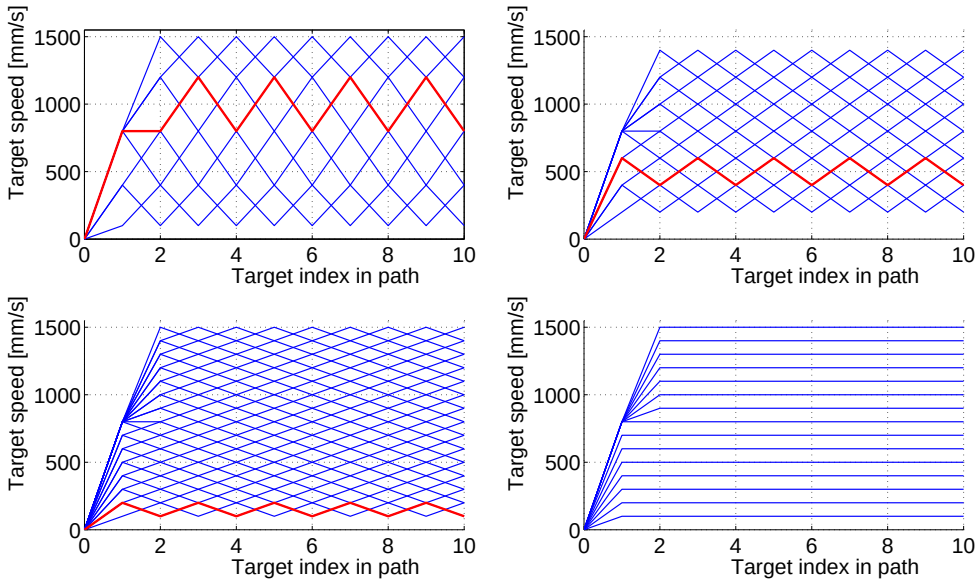

Figure 3.1. The zig-zag pattern broken down into parts. Upper left: Zig-zag with speed changes of 400 mm/s. Upper right: Speed changes of 200 mm/s. Lower left: Speed changes of 100 mm/s. Lower right: Constant speed. There are 11 targets in the path, so a there is a total of 10 segments in this case. A red path is added to show how one specific path is completed with the zig-zag warm-up.

One of the methods employed instead of the zig-zag approach, was to generate trian-

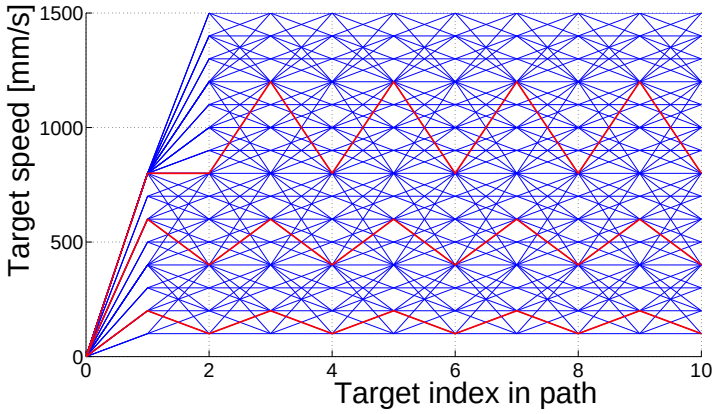

Figure 3.2. The complete zig-zag pattern. The same three red paths from Figure 3.1 are also included.

gular shaped paths with a varying velocity peak at each target. Figure 3.3 presents the two parts of the warm-up. Similarly to the zig-zag pattern, the triangular pattern also incorporated a constant speed pattern in the warm-up. This warm-up is thorough and investigates relevant paths, since it is expected that the optimal path will have either a triangular/dome shape, or follow a constant speed. Also, there are rarely any failures in RobotStudio where the robot cannot follow a path. As the consequence, a good cost function is produced from running this warm-up.

One drawback with the triangular warm-up approach is that the number of warm-up iterations now depends on the number of segments. The example presented in Figure 3.3 has the same number of segments as the zig-zag example in Figure 3.2, but requires 75 warm-up iterations instead of 63. A path with 11 segments requires 83 iterations. But, on the other hand, a path with 8 segments will only require 59 iterations.

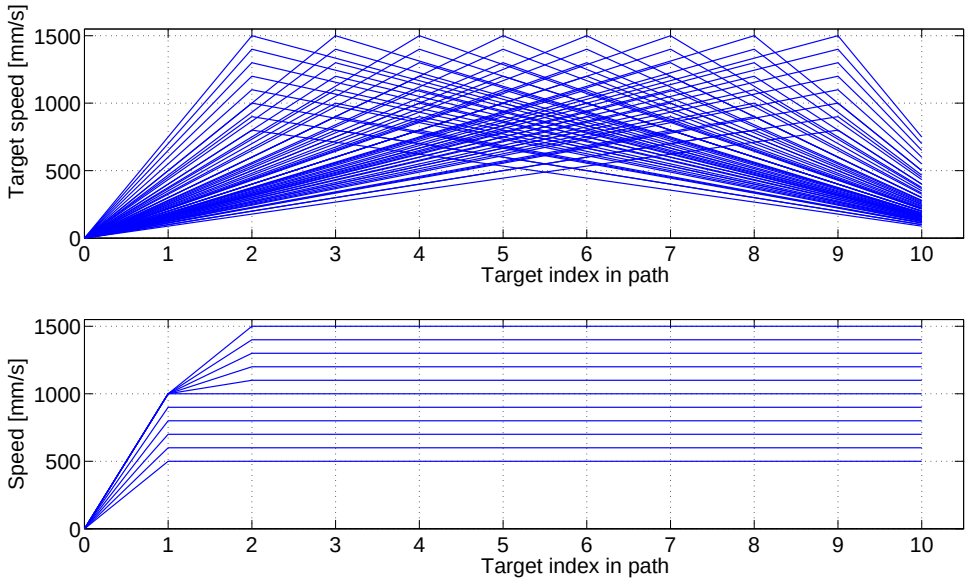

Figure 3.3. Triangular warm-up paths. Upper: Triangular warm-up paths with varying peak speeds at each target. Lower: A constant speed pattern.

The third warm-up employs AMPL with a MATLAB interface, and generates random paths by solving the following minimisation problem,

$$\begin{aligned} \min \quad & u^T Q u \\ s.t. \quad & \begin{bmatrix} t_{k+1} \\ v_{k+1} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} t_k \\ v_k \end{bmatrix} + \begin{bmatrix} h_k \\ 2\Delta_k / h_k \end{bmatrix} \quad \forall k \in [0, ..., N-2] \\ & u_k = \frac{2(\Delta_k - h_k v_k)}{h_k^2} \\ & v_0 = v_N = t_0 = 0 \\ & t_N = t_f \end{aligned}$$

where

$$Q = diag(r_0, ..., r_{N-2}), \quad r_k \in U(a, b).$$

Here the weights a and b have been chosen to be 3 and 200 respectively, so that r_k is a random number between 3 and 200. This has proven to yield a good distribution of the random paths. This method has, for this reason, been entitled as the random path generator.

An advantage with this method is that the time required to complete the path can be specified. Therefore, all of the warm-up paths will be relevant to the IDP

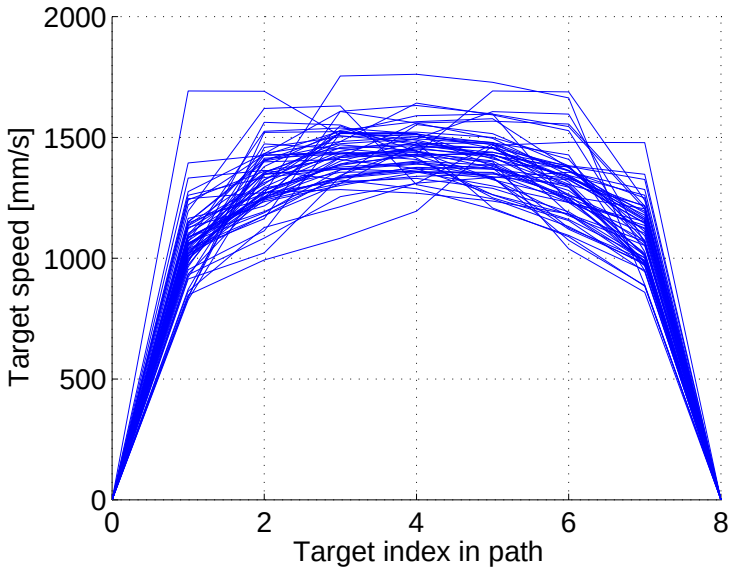

Figure 3.4. The random warm-up. 59 paths which are randomly generated to complete a given path according to several constraints. In this case the path is defined by 9 targets and has been divided into eight segments.

problem. Also, the number of iterations is completely independent from the number of segments and can be chosen freely. Figure 3.4 presents an example with 59 iterations, for a path with 8 segments. One disadvantage for this warm-up is that a new warm-up must be performed each time the target time changes, since the generated paths depend on the target time. The triangular warm-up on the other hand is more versatile and the same warm-up for a path can be used for most target times.

Once the warm-ups have been completed, the energy cost function is produced as a surface by interpolating the warm-up data. An example of a surface is shown in Figure 3.5. The cost is divided into the segments according to the targets, and has for each start- and end-speed in that segment a cost in energy. The two warm-up methods are compared later in the results section.

IDP stage

The IDP can finally start after the cost function has been created. The stage follows the steps presented in Section 3.2. The IDP returns an array of target speeds. These are tested by inserting them into the move instructions and then running a simulation. The speed and energy data from each simulation are also stored in the cost function for future iterations, thus implementing the iterative learning technique discussed in Section 2.3. Two versions of the IDPs were implemented, one with GC and the other without. The two methods will be compared in the results.

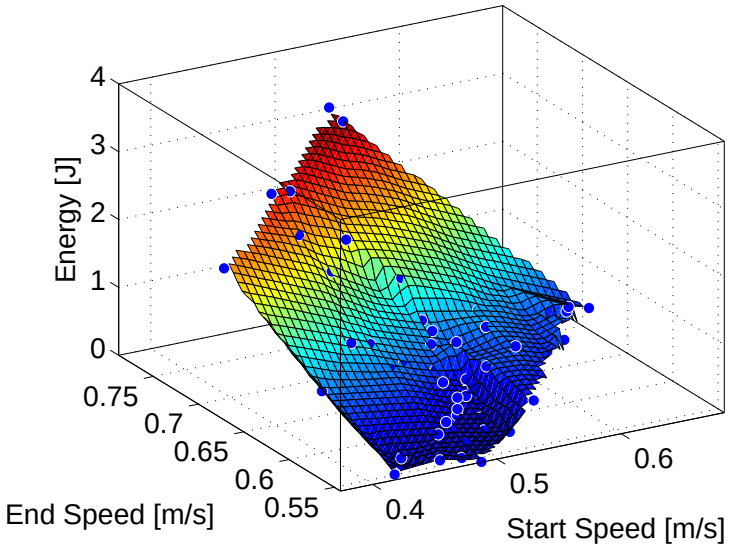

Figure 3.5. The interpolated surface for a segment of a path. The points represent the warm-up data points.

The GC has been included with a few simple modifications. To ensure that the IDP with GC does not centre around a non-optimal trajectory, a number of iterations are initially run without any GC. Thereafter, the GC for a number of iterations until the grid span becomes small enough. When this grid span is reached, the grid does not constrict any more but remains at the span defined by the final constriction.

3.3.3 RobotStudio Limitations

Some limitations of RobotStudio were encountered during the project; they include the energy measurements and the acceleration/deceleration limiter.

The energy measurements can vary heavily. Differences of 20 % have been found for two different simulations of the exact same trajectory. This naturally can have a negative effect on the cost data, since the measured energy might be much larger than what it could be for another simulation. This can of course be solved by running the same trajectory several times to obtain an average energy consumption. But this action would make the execution time of the algorithm several times longer.

RobotStudio as standard accelerates as quickly as possible to the desired speed. As previously mentioned, the IDP assumes constant acceleration, so this must be duplicated in RobotStudio to obtain accurate results. The employed acceleration/deceleration limiter is an instruction called PathAccLim. Constant acceleration can only be achieved by setting the correct parameters to the instruction. However, this has proved to be somewhat unreliable. The constant acceleration can easily be calculated by following the formula

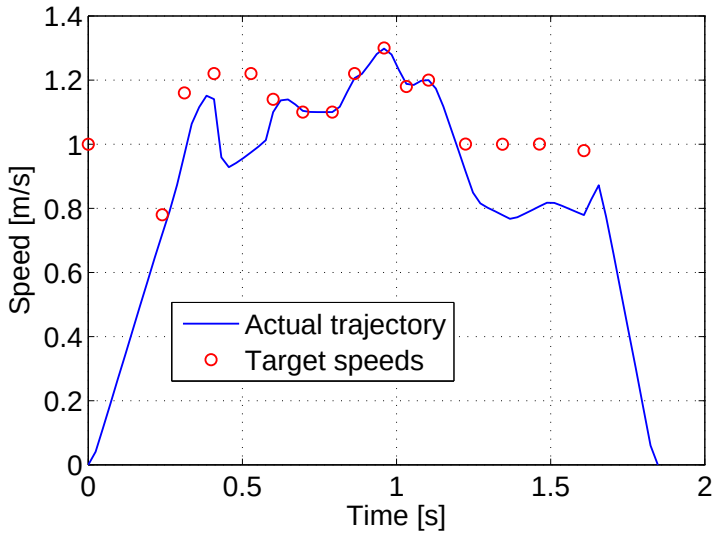

Figure 3.6. An example of a PathAccLim failure. The red circles represent the desired target speeds, and the blue line shows the performed trajectory. Clearly, the target speeds have not been reached due to too heavy acceleration limitations and the simulation can be concluded as a failure.

$$acc = \frac{v_2^2 - v_1^2}{2d}, \tag{3.4}$$

where v_2 and v_1 represent the end- and start-speed respectively, and d is the distance in between. However, inserting the answer from this formula directly into the instruction often results in a failure to follow the specified trajectory. Figure 3.6 shows an example of this. Instead, the formulas

$$acc = \frac{(1.1v_2)^2 - (0.9v_1)^2}{2d} \tag{3.5}$$

$$dec = \frac{(0.9v_2)^2 - (1.1v_1)^2}{2d} \tag{3.6}$$

are used. As a result, RobotStudio rarely fails to complete a trajectory, but constant acceleration/deceleration is rarely achieved as desired.

3.3.4 The Tested Trajectories

To test the algorithm, two different paths were tested. The paths include a horizontal path and a diagonal path where the robot goes in a downwards direction. The time it takes to complete the path, the target time, naturally has a great impact on the result. Therefore, two target times were chosen for each path, one which represents

a standard path time and another which takes longer time. Figures 3.7 and 3.8 present the horizontal and diagonal path respectively.

The horizontal path is 1.6 m long and the diagonal path is around 1 m. A standard TCP speed in RobotStudio is 1 m/s, so a target time of 1.65 s for the horizontal path and 1 s for the diagonal path were chosen. To test the algorithm for lower speeds, the target time for each path was doubled.

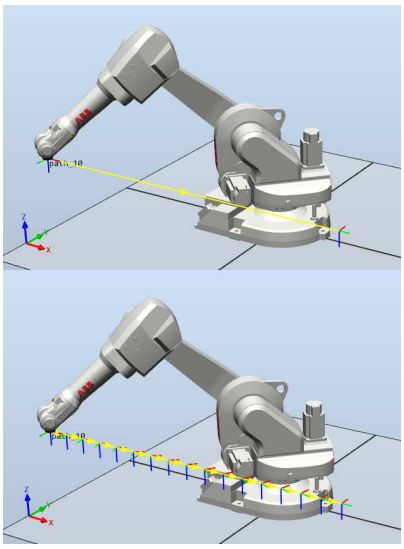

Figure 3.7. The horizontal path, the upper figure shows the path before it has been divided into segments, the lower figure after. The path is divided into 16 segments each with a length of 0.1 m.

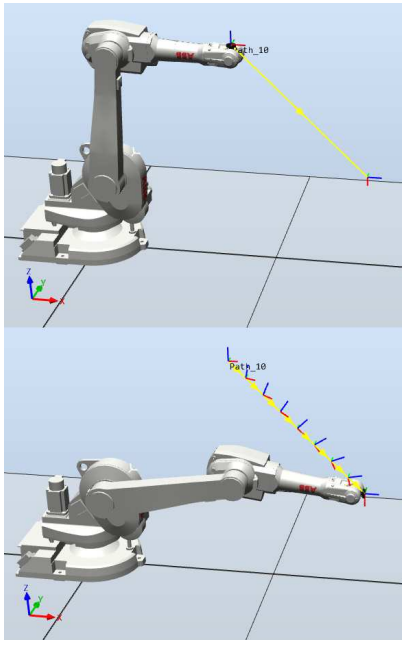

Figure 3.8. The diagonal path, the upper figure shows the path before it has been divided into segments, it also shows the starting position for when the robot. The lower figure shows the path after it has been divided into segments and the final robot position. The path is divided into 8 segments, each with a segment length of approximately 0.125 m.

3.4 ABB IRB1600 Robot

The opportunity arose to test some trajectories on an actual ABB robot. This section presents how the tests were conducted.

The robot, which is located in a test lab facility at Chalmers, is an ABB robot of type IRB1600. As explained in the introduction to RobotStudio in section 3.3, the robot is controlled by the same RAPID code that is used for simulations. Unfortunately, the process to start a programme or sample data is not identical to the procedure in an off-line environment. For this reason, the add-in has not been developed further to function with a real controller.

Instead, all the warm-up simulations and the optimisation process have been completed off-line with a model of the IRB1600 type robot. The most optimal trajectories were then tested directly on the actual robot to see whether or not an improvement of the energy consumption was made. The measurement of the energy

and TCP speed has been carried out using the on-line signal analyser in RobotStudio.

The tested paths are the same paths as presented earlier in the previous section. However, the horizontal path has been modified to avoid collisions with surrounding objects around the robot in the test lab. Instead, the path is in front of the robot, and has only a length of 1 m. Only a target time of 2 s was tested for both paths.

3.5 KUKA

This section introduces the second application where the IDP has been implemented. It is a mathematical model written in MATLAB by the industrial robot manufacturer KUKA. It simulates a KUKA KR2210 industrial robot. In this model, a path is defined by vectors with time points and the corresponding robot joint angles. The consumed energy for the path is measured by running the simulation, during which the robot moves with constant acceleration. This section begins by describing the warm-up stage, where energy data is obtained for the cost function. An explanation of the implementation of the IDP follows and the section concludes by presenting limitations of the KUKA model and the tested trajectories.

Warm-up stage

The cost function that is used to solve the IDP for the KUKA model has the same format as for the RobotStudio application, where it is created as a surface with the axes start-speed, end-speed and energy. Warm-up paths have only been generated by the random method described in Section 3.3.2 to create the cost function. An example of the cost function for the KUKA model is illustrated in Figure 3.10, and the collected warm-up data used for the cost function is presented in Figure 3.9. In this particular case the cost function is created for a path that rotates the robot from 0° to 90° around its base with 20 segments, making each segment 4.5° long.

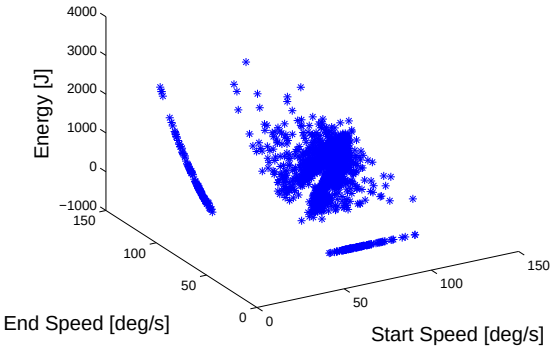

Figure 3.9. The warm-up data, which contains the start-speed, end-speed, and energy for that motion

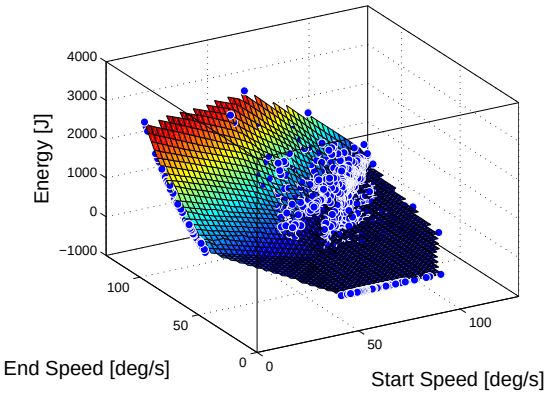

Figure 3.10. The surface created from the data points in Figure 3.9 using linear interpolation.

However, there is a complication with the KUKA model. Since it is not possible to enter a start-speed and end-speed for a segment in the KUKA model, the time scale for the positions has to be calculated. This can be done by defining the velocities a path should have in each position i.e. each start and end of a segment. Having both the position p_k and the speed v_k, the sampling time needed for that motion can be calculated by solving h_k from Equation 2.5 as

$$h_k = \frac{2p_{k+1} - 2p_k}{v_k + v_{k+1}}. \tag{3.7}$$

Accumulating these sample times will form a non uniform time scale

$$t_{h,k} = t_{h,k} + h_k \tag{3.8}$$

where each time corresponds to one position of the robot. These positions can be calculated as stated in Equation 2.5 as

$$p_{k+1} = p_k + v_k h_k + \frac{u_k h_k^2}{2}. \tag{3.9}$$

For Equation 3.9, the acceleration u_k is also required which can be solved from Equation 2.1 as

$$u(k) = -\frac{(v_k + v_{k+1})(v_k - v_{k+1})}{2\Delta} \tag{3.10}$$

However, another complication exists. The KUKA model requires equidistant time steps in order to calculate the energy accurately. Therefore the non-uniform time

scale has to be up-sampled into a finer time scale with equidistant time steps.

This can be achieved by first creating a new partially uniform time scale and corresponding positions. This new time scale is created by adding N equally distant samples between each t_h thus creating the new time scale t_e. To get the positions for this new partially uniform time scale, Equation 3.9 is applied again, but now with h_e instead of h where

$$h_{e,k} = t_{e,k+1} - t_{e,k} \tag{3.11}$$

Thus, a finer grid for both the time and position values is obtained. Values for a final equidistant time scale with the desired sampling time is calculated by interpolating the fine grid. This new uniform time vector and its corresponding position vector can now be simulated to get the energy.

As a final point regarding the KUKA model, it has been discovered that the accuracy of the simulations is improved by adding samples before and after the actual motion. Therefore, 10 extra samples have been added before and after the trajectory values where the robot remains still.

IDP stage

Similarly to RobotStudio, the IDP can finally start after the cost function has been created. This step utilises the same function as RobotStudio and is therefore carried out in the exact same manner. However, only an IDP without GC has been implemented for the KUKA application.

3.5.1 KUKA Limitations

The KUKA model has restricted the user to only be able to define a path by entering the corresponding time and joint angles in an Excel-spreadsheet. From there, the simulation is carried by running an executable file using the parameters from the Excel-spreadsheet. This method lacks visual aids that show how the robot is moving which makes it difficult to get an overview of the movements when several joints are manipulated simultaneously.

3.5.2 The Tested Trajectories

For the KUKA case, a standard path is a trajectory that has been externally provided from a real robot simulated in DELMIA, a software used to simulate KUKA robots. This ensures that the motion is realistic. Three different reference trajectories were provided from KUKA. The first trajectory rotates the robot from -180° to

-90° around its base. The second trajectory rotates in the same way but is longer, it rotates 360° from -180° to 180°. The third and final trajectory moves both around its base and up at the same time, hence activating multiple joints.

For paths that activate multiple joints, as for a linear motion or the third trajectory, it can be quite complex to calculate the multiple joint angles. For this reason, only motions where one joint angle changes are considered. The largest joint, the one that rotates the robot around its base, has been chosen for this purpose since the energy consumption for this joint will be larger than the others. Hence, it will be easier to compare the consumption for a standard path to the optimised one. Another reason to choose the base joint is that when dividing the trajectory into segments, all of the segments will have the same properties from an energy perspective. On the basis of this, all data collected to create the cost for the IDP can be used to form only one surface that will fit for all segments.

4 RESULTS

4.1 RobotStudio

This section presents the results from RobotStudio. A total of 16 tests were conducted, eight for each path described in Section 3.3.4. The tests include running the IDP with or without GC, and each of those cases is tested with either the triangular or the random warm-up method. The triangular warm-up required 123 iterations for the horizontal path, and 59 iterations for the diagonal path. For a fair comparison, the same number of iterations was chosen for the random warm-up in each case.

Each test runs the warm-up procedure and then runs 50 iterations of IDP. After each test, the most promising trajectory is chosen and compared against a standard path in RobotStudio. Since the energy consumption varies, data for each trajectory and standard path is obtained 10 times to get an average. The results from these tests are presented in Table 4.1.

For the GC case, 10 IDP iterations are initially run without GC, then 25 iterations where the grid contracts. The remaining 15 iterations are run with the grid size obtained from the final constriction. The grids for the first 10 iterations naturally span every possible state and control value, exactly as the IDP without GC does. At the eleventh iteration the grid span is halved. For the next 25 iterations the grid spans are constricted by 2 % ($\gamma = 0.98$). All of the grids have 81 points. For the IDP without GC, a grid size of 161 points is used.

Table 4.1. Results from the IDP for RobotStudio. Presents the percentage of energy saved compared to a standard path with the same target time. A negative percentage means that the IDP trajectory consumes on average more energy compared to the standard trajectory, and vice versa.

Trajectory	Time [s]	R [%]	Tri [%]	GC, R [%]	GC, Tri [%]
Diagonal Path	2	75.36	**84.9**	57.8	80.3
Diagonal Path	1	**-0.8**	-17.4	-8.0	-21.5
Horizontal Path	3.3	**-1.00**	-1.4	0.2*	-1.2
Horizontal Path	1.65	-1.2	-1.3	**-0.7**	-5.6

R - Random warm-up
Tri - Triangular warm-up
GC - IDP with grid constriction
Bold - Best result for that path
* - Not within 5 % of the target time

Figures 4.1 and 4.2 present an example of how the IDP with or without GC converge to a certain trajectory. The example is taken from the horizontal path with a target time of 1.65 s and a random warm-up.

The computation time has also been measured for the IDP with or without GC. For

the IDP without GC and 16 segments, one iteration took approximately 1 minute to complete; the IDP with GC only took around 12 s. The comparison was performed on a PC laptop with a 2.27GHz i5 processor and 3 GB of RAM.

The following two subsections display the trajectories and details of the best result for each path. These trajectories are marked with a bold font in Table 4.1.

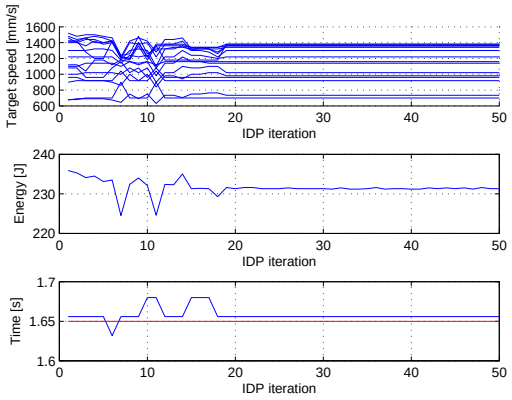

Figure 4.1. The targets speeds, energy consumption and path time for all iterations of the IDP without GC for the horizontal path. The actual target speed values are not important, note instead that after 20 iterations the IDP has converged to a trajectory that minimises the total cost J. Note also that the DP finds another trajectory that actually consumes less energy, but does not fulfil the target time, which is represented by the red line, as accurately as the final trajectory.

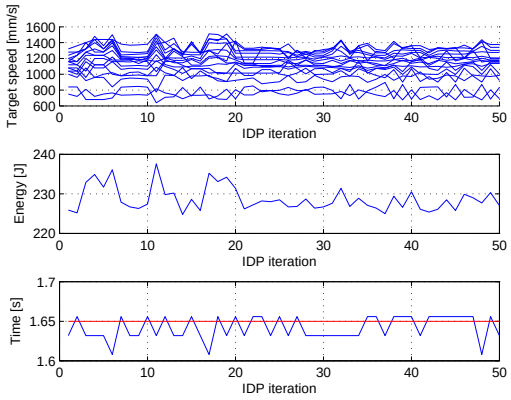

Figure 4.2. The targets speeds, energy consumption and path time for all iterations of the IDP with GC for the horizontal path. In this case the IDP does not converge to a certain trajectory, but a region around the optimal trajectory. Note that the target speeds and path time never stabilise.

4.1.1 Diagonal Path

The best trajectory for the diagonal path with a target time of 2 and 1 s are presented in Figure 4.3 and 4.5 respectively, where they are compared to a standard path with the same target time. The details of the trajectories can be found in Table 4.2 and 4.3.

Table 4.2. Detailed results for the diagonal path with a target time of 2 s. The average is obtained from running the trajectory and the reference path 10 times in RobotStudio.

Average standard time	2.03	[s]
Average IDP time	1.96	[s]
Average time difference	0.07	[s]
Average standard energy consumption	5.0	[J]
Average IDP energy consumption	0.7	[J]
Average saved/lost percentage of energy consumption	84.9	[%]

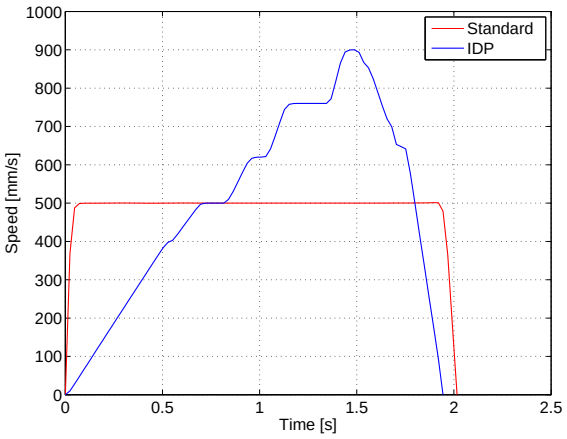

Figure 4.3. TCP speed of the diagonal IDP trajectory compared to the standard movement of 500 mm/s.

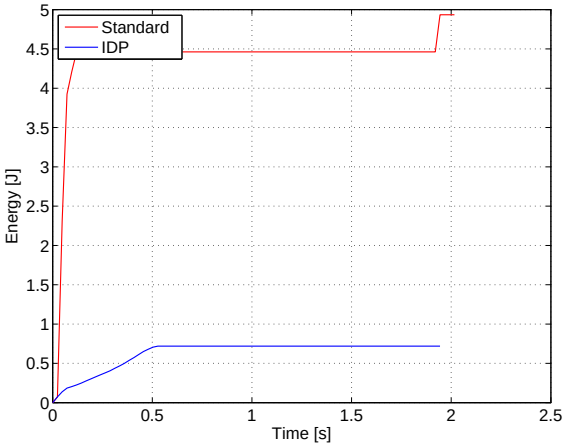

Figure 4.4. Energy consumption of the diagonal IDP trajectory compared to the standard movement of 500 mm/s.

Table 4.3. Detailed results for the diagonal path with a target time of 1 s.

Average standard time	1.08	[s]
Average IDP time	1.05	[s]
Average time difference	0.03	[s]
Average standard energy consumption	26.8	[J]
Average IDP energy consumption	27.0	[J]
Average saved/lost percentage of energy consumption	-0.8	[%]

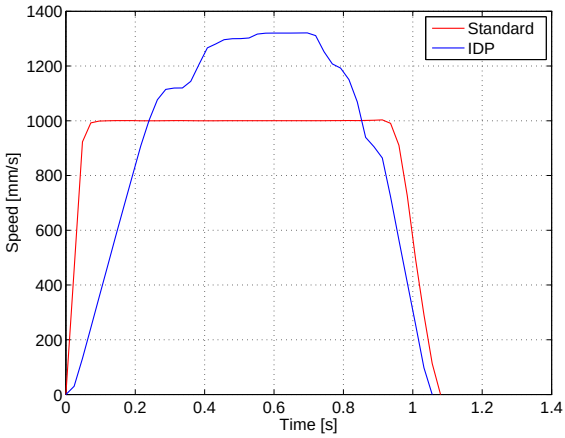

Figure 4.5. TCP speed of the diagonal IDP trajectory compared to the standard movement of 1000 mm/s.

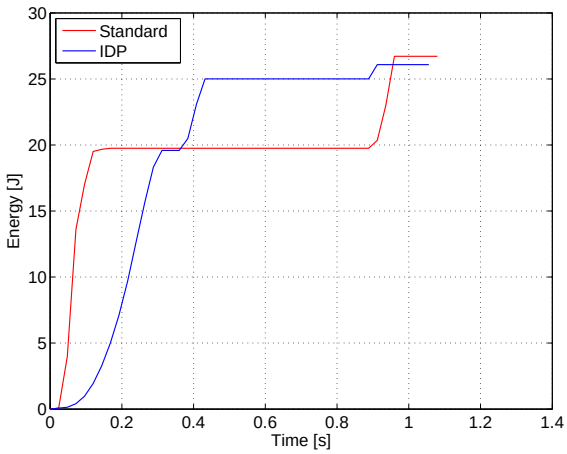

Figure 4.6. Energy consumption of the diagonal IDP trajectory compared to the standard movement of 1000 mm/s

4.1.2 Horizontal Path

Similarly to the diagonal path, the best trajectory for the horizontal path with a target time of 3.3 and 1.65 s are presented in Figure 4.7 and 4.9 respectively. They are also compared to a standard RobotStudio path with the same target time. The details of the trajectories can be found in Table 4.4 and 4.5.

Table 4.4. Detailed results for the horizontal path with a target time of 3.3 s.

Average standard time	3.23	[s]
Average IDP time	3.20	[s]
Average time difference	0.02	[s]
Average standard energy consumption	204.9	[J]
Average IDP energy consumption	206.9	[J]
Average saved/lost percentage of energy consumption	-1.0	[%]

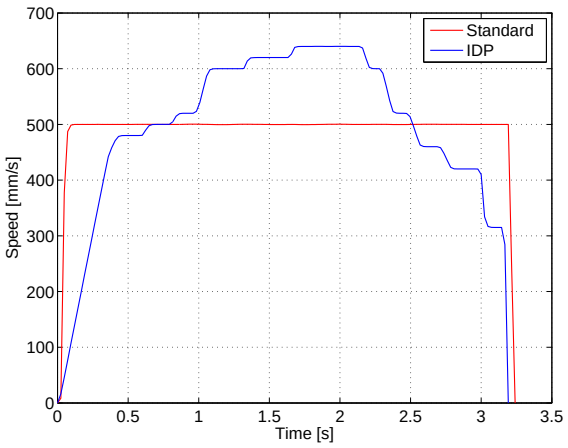

Figure 4.7. TCP speed of the horizontal IDP trajectory compared to the standard movement of 500 mm/s.

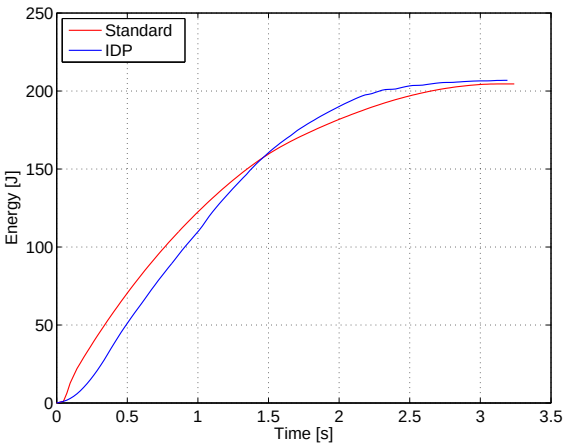

Figure 4.8. Energy consumption of the horizontal IDP trajectory compared to the standard movement of 500 mm/s

Table 4.5. Detailed results for the horizontal path with a target time of 1.65 s.

Average standard time	1.66	[s]
Average IDP time	1.61	[s]
Average time difference	0.05	[s]
Average standard energy consumption	223.6	[J]
Average IDP energy consumption	225.2	[J]
Average saved/lost percentage of energy consumption	-0.7	[%]

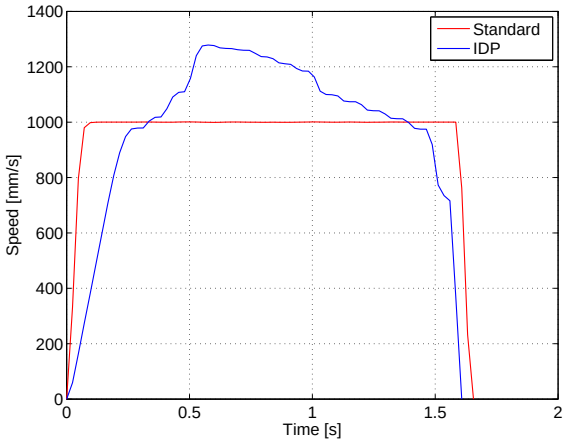

Figure 4.9. TCP speed of the horizontal IDP trajectory compared to the standard movement of 1000 mm/s.

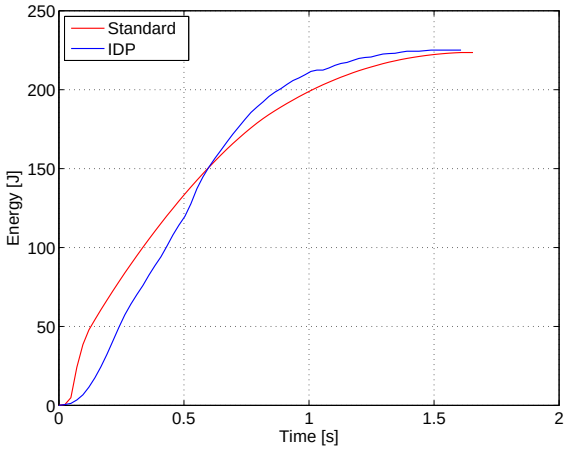

Figure 4.10. Energy consumption of the horizontal IDP trajectory compared to the standard movement of 1000 mm/s

4.2 ABB IRB1600 Robot

This section presents the results from the test lab, where an ABB IRB1600 industrial robot has been used to validate some results from RobotStudio. As described in Section 3.4, two paths have been tested. Similarly to the off-line tests with RobotStudio, deviations exist in the energy measurements. To counter this, the standard and IDP trajectory were run 10 times to obtain an average.

4.2.1 Diagonal Path

The IDP produced a trajectory that saved on average 85 % of the consumed energy when run off-line. The IDP used a triangular warm-up, and was without GC. GC and the random warm-up were not incorporated, since at the time of the testing they had not yet been developed.

When run on the real robot, an average saving of 89.9 % was made. The energy consumption for all 10 runs are included in Table 4.11. The trajectory itself is presented in Figure 4.12, along with the energy consumption in Figure 4.13.

Figure 4.11. Energy measurements for the real robot, diagonal path

Test	Standard trajectory	IDP trajectory	Savings [%]
1	14.9	0.7	95.3
2	17.5	2.0	88.5
3	17.9	1.8	89.9
4	18.2	2.6	85.7
5	17.0	2.2	87.3
6	15.9	2.0	87.4
7	16.6	1.5	90.9
8	18.2	1.1	93.8
9	20.0	1.7	91.7
10	15.4	1.8	88.4
Average	17.2	1.7	89.9

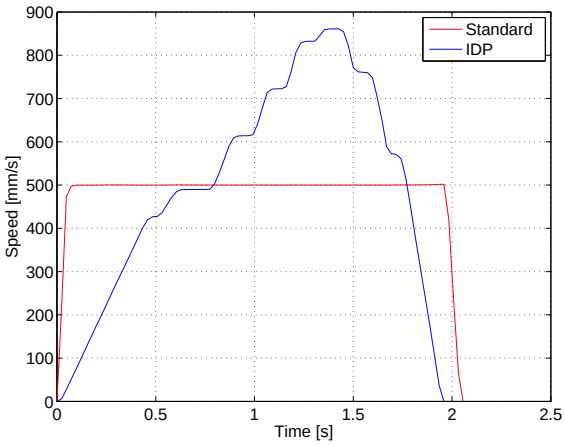

Figure 4.12. TCP speed, for the real robot, of the diagonal IDP trajectory compared to the standard movement of 500 mm/s.

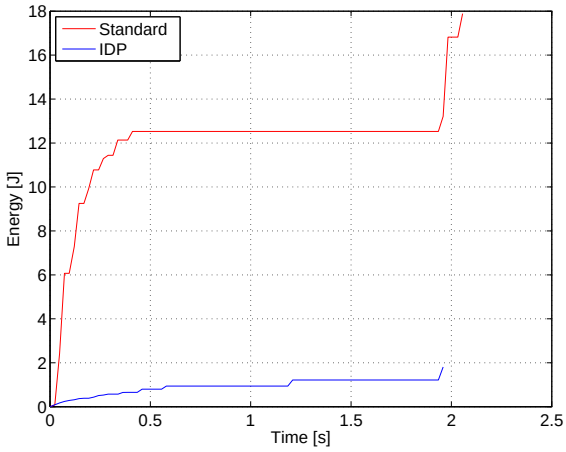

Figure 4.13. Energy consumption, for the real robot, of the diagonal IDP trajectory compared to the standard movement of 500 mm/s.

4.2.2 Horizontal Path

For the horizontal path, the IDP off-line actually achieved a 2.11 % improvement. Again, this was obtained with a triangular warm-up and no GC. The results from the ten measurements are presented in Table 4.14. The IDP and standard trajectories, and their energy consumption are presented in Figure 4.15 and 4.16 respectively. On average, 1.2 % of the consumed energy was saved.

Figure 4.14. Energy measurements for the real robot, horizontal path

Test	Standard trajectory	IDP trajectory	Savings [%]
1	128.6	122.0	5.1
2	124.2	124.4	-0.1
3	129.5	120.7	6.8
4	128.0	125.0	2.4
5	121.0	122.9	-1.6
6	124.0	127.2	-2.6
7	123.4	121.6	1.4
8	127.1	123.7	2.7
9	121.5	125.1	-3
10	125.3	123.9	1.1
Average	125.3	123.6	1.2

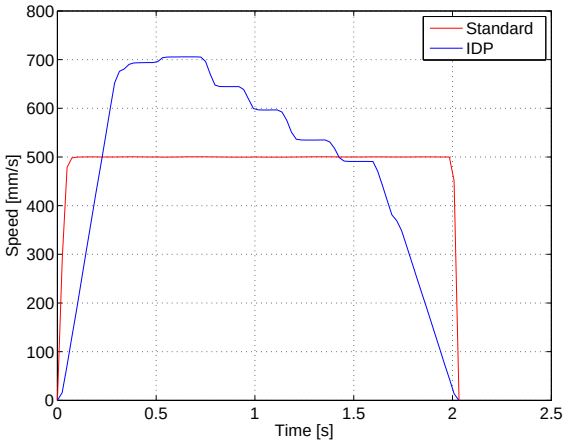

Figure 4.15. TCP speed, for the real robot, of the horizontal IDP trajectory compared to the standard movement of 500 mm/s.

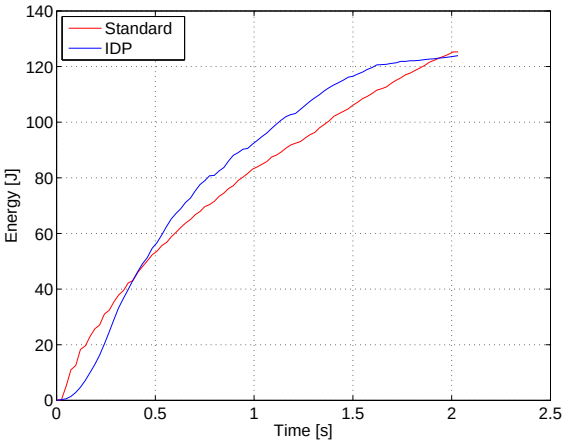

Figure 4.16. Energy consumption, for the real robot, of the horizontal IDP trajectory compared to the standard movement of 500 mm/s.

4.3 KUKA Model

The results of the KUKA implementation are presented in this section, where the two different paths described in Section 3.5.2 have been investigated. Similarly to the RobotStudio case, the IDP trajectories are compared to a standard one by running several simulations. But, in this case an average value is not needed, since the energy measurements have no variation. An investigation of how the results are affected by the number of warm-up paths and IDP-iterations is also carried out.

4.3.1 Path 1

Path 1 rotates the robot around its base from -180° to -90°, over 1.17 s. The standard trajectory consumes 5683.3 J. For the same path, the IDP has been tested also by varying the number of warm-up and IDP iterations. The results are gathered in Table A.1 in Appendix A.

Out of all of the cases included in Table A.1, the most frugal trajectory that consumes the least amount of energy is marked with bold font. It uses 25 warm-up iterations, and is found at IDP iteration 5 of 10. This solution consumes 4196.4 J, which corresponds to a saving of 26.2 %. The trajectory is depicted in Figure 4.17 and its corresponding energy consumption in Figure 4.18.

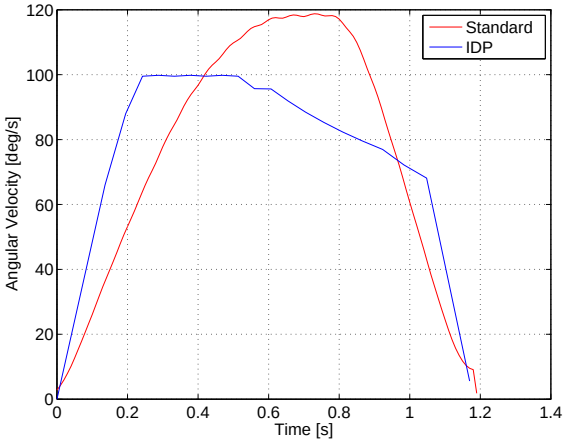

Figure 4.17. TCP speed of the most frugal trajectory compared to the standard movement, for Path 1.

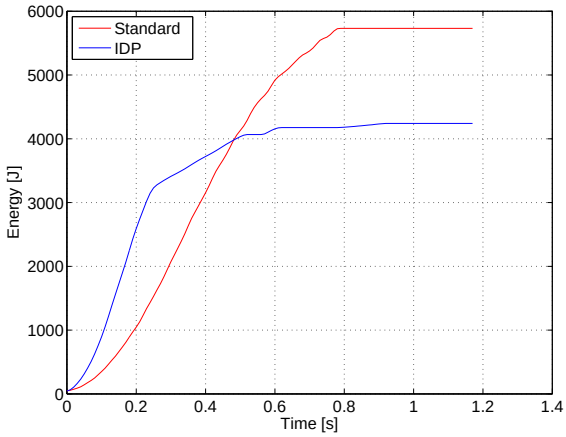

Figure 4.18. Energy consumption of the IDP trajectory compared to the standard movement, for Path 1.

Figure 4.19 and 4.20 shows the energy consumption for each IDP iteration for two different cases. Figure 4.19 presents a case where only a few initial warm-up iterations (poor initial data) are used, and the IDP converges to a solution rather fast. This solution however, is rather poor. Figure 4.20 shows a case where more warm-up data exists, and demonstrates how the IDP slowly converges to the optimum.

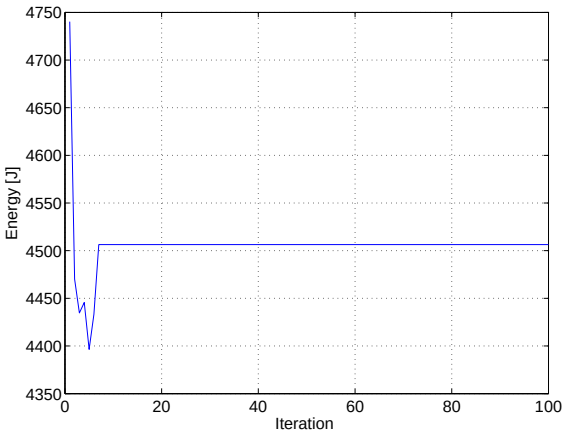

Figure 4.19. The energy consumption for 100 IDP iterations using only 5 warm-up paths. Note that the IDP converges quickly, but to a poor solution.

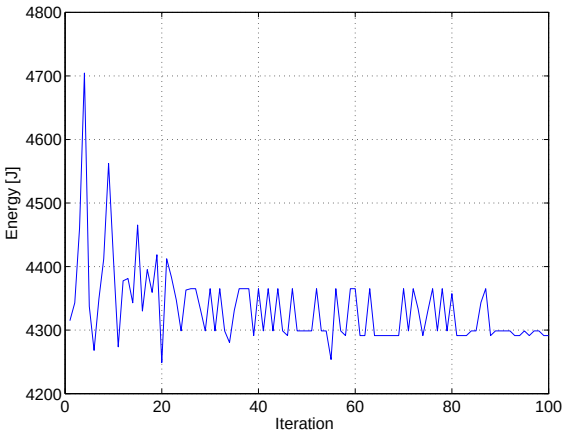

Figure 4.20. The energy consumption for 100 IDP iterations using 10 warm-up paths. Note that the IDP slowly converges to an optimum.

4.3.2 Path 2

Path 2 moves from -180° to 180°, over 3.47 s. The standard trajectory consumed 15024.8 J. Applying IDP in the same fashion as for Path 1 yields the results presented in Table A.2 in Appendix A. It was found that 100 warm-up paths and 58 IDP iterations were required. This trajectory consumed 13990.8 J, corresponding to an improvement of 6.9 %. The IDP trajectory and energy consumption is compared to the standard movement in Figure 4.21 and 4.22.

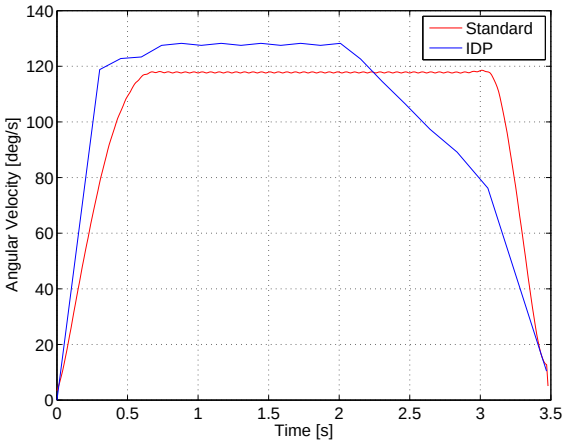

Figure 4.21. TCP speed of the IDP trajectory compared to the standard movement, for Path 2.

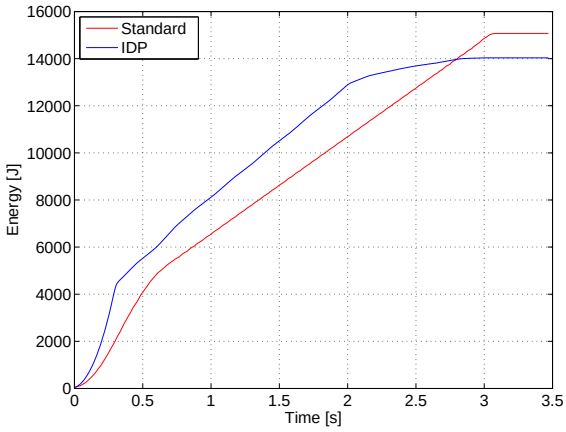

Figure 4.22. Energy consumption of the IDP trajectory compared to the standard movement, for Path 2.

As for Path 1, the energy consumption for each iteration of two different IDP cases are compared. Figure 4.23 presents a case with only three warm-up iterations. However, for this case the IDP managed to find a good trajectory. Figure 4.24 shows a case where 100 warm-up iterations were performed. This case converges slowly, but eventually finds a better trajectory than the previous case.

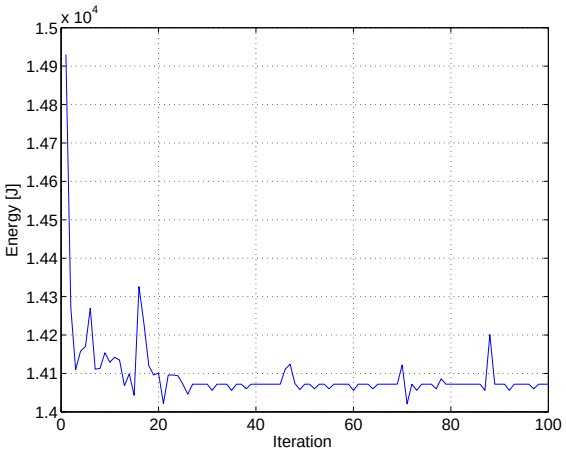

Figure 4.23. The energy consumption for 100 IDP iterations using only 3 warm-up paths. Note that the IDP converges quickly, and actually finds a good solution.

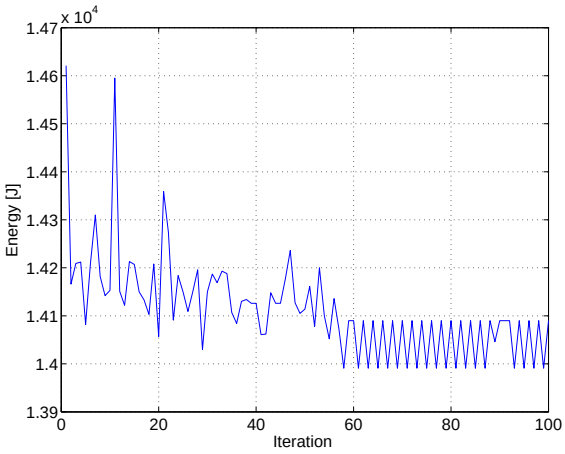

Figure 4.24. The energy consumption for 100 IDP iterations using 100 warm-up paths. Note that the IDP slowly, and eventually finds the best trajectory for Path 2.

5 DISCUSSION

5.1 RobotStudio

The results from RobotStudio are mixed. Only two cases where any improvement on the energy consumption was made. The first was for the diagonal path with a target time of 2 s, and that was an 85 % improvement. In this case, the IDP found a trajectory where the robot could accelerate slowly and utilise gravity to minimise the energy consumption. With the same path, but for a target of 1 s, the robot had to accelerate more rapidly which meant that gravity could not be used to the same extent. The most promising trajectory from 10 runs on average consumed 0.77 % more energy than the standard path.

The second case with a small improvement was 2 % for the shorter horizontal path. The best trajectories for the longer horizontal path consumed on average 0.72 % and 1 % more energy.

Generally, the random warm-up paths produced the most promising results. This is most likely due to the warm-up paths being more relevant. In other words, they had the all the same target time and the trajectories were similar to the IDP solution. Whereas only a few of the warm-up trajectories had the same target time for the triangular warm-up paths. However, a triangular warm-up produced the best IDP trajectory for the diagonal path with a 2 s target time. Again, this is most likely due to the IDP trajectory having a more triangular shape, instead of the parabolic shapes for the other cases.

A comparison between the IDP with or without GC can also be carried out. Overall, the IDP without GC obtained the best trajectories and in some cases, the diagonal path with a target time of 1 s for example, the IDP with GC performed much worse. One major difference between the two methods was that the IDP with GC was more likely to either under- or overshoot the target time. To solve this, the penalty α had to be heavily increased. Despite this though, the IDP with GC was unable to find a trajectory for the horizontal path that came within 5 % of the target time 3.3 s, and would complete the path much sooner instead. Another significant difference is the computation time of the two methods, which was considerably reduced with GC employed.

Figure 4.1 shows an example where the IDP without GC converges to a specific trajectory. Figure 4.2 shows that the IDP with GC also converges, but instead to a region around the optimal trajectory. This can be explained by considering the grids. For the IDP without GC, the grids are constant and therefore the possible trajectories are limited. It is also easier for the IDP to find the optimal solution, since the cost function is updated by iteratively adding data at the same points which always exist in the grid. With GC, the grids are constantly changing which

allows the IDP to explore new trajectories, but the target speeds and the energy consumption never stabilise. This also increases the possibility of the IDP with GC to lock itself to a certain region, since new speeds can always be tested in that region. It is possible that more IDP iterations would eventually yield better results for the IDP with GC. A cost function with more points and a larger region might also improve the results for both IDP cases.

The energy consumption for most cases was not decreased, there are a number of reasons that could explain why this is so.

By studying the trajectories in Figure 4.3, 4.5, 4.7 and 4.9, one can observe that they all have a shape one might expect for an optimal trajectory. However, constant acceleration is rarely achieved. Some of the speed changes are performed with little or no limitation on the acceleration/deceleration, the trajectories instead have a staircase form. This leads back to the difficulties with the instruction PathAccLim. It is unknown if constant acceleration would actually improve the results since this has never been achieved. However, since constant acceleration is assumed in the trajectory planning model, one might expect this to be the case.

While running the tests, it became evident that the variations of the energy measurements had a significant effect on the quality of the cost function and therefore the results. Running the same warm-up procedure twice would yield different cost functions and therefore the results would vary. This must also be the case then when a trajectory from the IDP is tested after each iteration. There is a chance that the energy measurement for a tested trajectory might be higher than another trajectory that in fact consumes more energy. If this occurs the optimal solution might never be found.

The method employed to divide the path into segments can also be considered as a limitation. As described in Section 3.3.2 under the preparation section, the path is divided by inserting targets in between two other targets until the desired segment length of 0.1 m is reached. However, by using this method, the number of segments can only be 2^n. For example, for the diagonal path with a length of 1 m, the segment length of 0.125 m is achieved with 8 segments. But if there were instead 10 segments, the exact desired segment length would be achieved.

5.2 ABB IRB1600

The tests from the real robots yielded positive results, but more tests are required to determine the full potential of the algorithm. They also verified the accuracy of RobotStudio and that the optimisation can be performed off-line.

5.3 KUKA

It can clearly be seen from the results that there exists a great opportunity of saving energy by altering the way a motion is carried out. This is first seen for Path 1 where a trajectory was found which saved as much as 26.2 %. For Path 2, the energy consumption is also decreased, but only by 6.9 %. This difference in percentage between the two trajectories probably originates from the fact that most of the energy is saved in the beginning and the end of a trajectory. Naturally for a longer path these regions where energy can be saved will constitute a smaller part of the trajectory.

The question of how much initial data and how many IDP-iterations that is required in order to obtain a satisfying result can be answered by reading the tables in Appendix A. The results foreclose that if less than three warm-up trajectories are used, then there is a high risk of not finding an IDP solution. It can also be seen from Path 1 and 2 that more IDP-iterations does not always guarantee that a better result will arise. This probably originates from the warm-up and that random weights are used when creating the warm-up paths for the KUKA-model. The random nature of the warm-up means that the quality of the cost function can vary, despite the number of iterations. This can be shown by considering what happened for Path 2, where a good solution was found using only 3 warm-up paths for initiating the cost function. Compare this to Path 1, where a case was presented with 5 warm-up iterations which produced poor results.

When few iterations are used there is a high probability for the IDP to converge to a value quickly as in Figure 4.19 and 4.23. As seen in Figure 4.19, an early convergence might stabilise around a trajectory that consumes more energy than the best solution found in another IDP run with more warm-up iterations. Therefore, to avoid this more warm-up iterations should be executed. However just adding many simulations will not always yield a good surface.

Observe in Figure 3.9 the energy costs, it can be seen that most of the data points are located in one region. Since the cost function already contains a lot of points, just adding more in the same regions will most likely not noticeably improve the cost function. By adding more points at other regions, the cost function may be extended to contain cheaper possibilities, from which better solutions may arise. One way of expanding the region would be to combine the triangular warm-up with paths from the random path generator.

6 CONCLUSION

The algorithm in several different versions have been successfully implemented with varying results, and the objectives defined in the introduction of this project have been accomplished.

For RobotStudio, only a few trajectories were found where the energy consumption was improved. The results varied from a 20 % increase and a 85 % decrease of the consumed energy. The GC method also yielded mixed results, but overall the IDP without GC which converged to a single trajectory yielded the best results. However, more IDP iterations may improve the GC method. One advantage of the IDP with GC was that the computation time is significantly lower than without GC. For this reason, it is easy to justify more iterations.

The tests on the robot in the test lab revealed positive results. In both cases an improvement on the energy consumption was made. The tests also validated RobotStudio's accuracy, and proved that the optimisation process may be performed off-line. However, further testing is required to fully appreciate the potential of the algorithm.

The implementation for the KUKA case was successful. Only two simple paths were tested, both gave positive results where the IDP trajectories saved 6.9 % and 26.2 % of the energy compared to a standard path. Again, further testing is required.

7 FUTURE WORK

7.1 RobotStudio

If the project were to continue, efforts to improve the cost function would be the first priority. This might include running more warm-up iterations and testing other warm-up methods to build up a cost surface with a larger region and more points. To reduce the variance of the energy measurements, every warm-up and IDP trajectory could be tested multiple times to obtain an average which would be incorporated into the cost function.

The second priority is to improve the method to calculate the parameters for PathAccLim to achieve constant acceleration. Or try other methods to reduce the rate of acceleration.

The method employed to divide the path can also be developed further, so that the segment length can come closer to the desired length. Also, techniques to iteratively adjust the segment lengths could be investigated. For example, a segment could be split up into smaller parts where large speed differences occur. Other segments where the speed is constant could be made longer or merged together.

The results of the IDP with GC could also be improved by a simple change. At the moment the IDP with GC is run initially for a few iterations without GC. It then continues with GC, using for the first iteration the final trajectory from these initial iterations to define the centre of the grids. However, it is not certain that the final trajectory is suitable for this task. Instead, the IDP should consider all previous iterations, and define the centre of the grids for the iteration with the best trajectory found so far. Thus increasing the possibility of centring itself around the most energy efficient trajectory.

7.2 KUKA

It would be interesting to test more trajectories, and not only ones where the robots rotates around its base. An investigation of this kind would lead to a better understanding of how energy could be saved and what motions that could gain the most from these optimisation methods. Further investigations would have been carried out if it was not for the limited time-frame of the project.

It may be possible to improve the results, as mentioned in the discussion, by expanding the cost function. This would be achieved by running more various warm-up paths, including perhaps a combination of the random and triangular paths.

Finally, tests on a real robot would be interesting to validate the accuracy of the KUKA model and the results found so far.

Bibliography

Arimoto, S., Kawamura, S. and Miyazaki, F. (1984). Bettering operation of robots by learning. *Journal of Robotic Systems*, **1**(2), 123–140.

Field, G. and Stepanenko, Y. (1996). Iterative dynamic programming: an approach to minimum energy trajectory planning for robotic manipulators. vol. 3. pp 2755–2760 vol.3.

Luus, Rein (2000). *Iterative dynamic programming*. vol. 110.; 110. Chapman & Hall/CRC. Boca Raton.

Naidu, Desineni S. (2003). *Optimal control systems*. CRC. Boca Raton, Fla.

Shin, Kang and McKay, N. (1986). A dynamic programming approach to trajectory planning of robotic manipulators. *IEEE Transactions on Automatic Control*, **31**(6), 491–500.

Sundstrom, O. and Guzzella, L. (2009). A generic dynamic programming matlab function. In: *Control Applications, (CCA) Intelligent Control, (ISIC), 2009 IEEE*. pp 1625–1630.

Wigström, Oskar, Lennartson, Bengt, Vergnano, Alberto and Breitholtz, Claes (2013). High-level scheduling of energy optimal trajectories. *IEEE Transactions on Automation Science and Engineering*, **10**(1), 57–64.

A KUKA

A.1 Path 1

Table A.1. Simulation results from IDP Path 1

Warm-Up Iterations	DP-Iteration	Lowest Energy Iteration	Lowest Energy Consumed	Savings [%]
2	1	1	4697.3	17.3
2	10	5	4563.0	19.7
2	20	13	4482.5	21.1
2	50	error @ 25	-	-
2	100	-	-	-
3	1	1	4779.2	15.9
3	10	8	4467.2	21.4
3	20	19	4272.2	24.8
3	50	48	4226.2	25.6
3	100	48	4226.2	25.6
5	1	1	4740.3	16.6
5	10	5	4396.4	22.6
5	20	5	4396.4	22.6
5	50	5	4396.4	22.6
5	100	5	4396.4	22.6
10	1	1	4314.8	24.1
10	10	6	4268.1	24.9
10	20	20	4248.9	25.2
10	50	20	4248.9	25.2
10	100	20	4248.9	25.2
25	1	1	4616.5	18.8
25	**10**	**5**	**4196.4**	**26.2**
25	20	5	4196.4	26.2
25	50	5	4196.4	26.2
25	100	5	4196.4	26.2
50	1	1	4276.4	24.8
50	10	7	4235.5	25.5
50	20	11	4234.8	25.5
50	50	32	4204.3	26.0
50	100	32	4204.3	26.0
100	1	1	4286.9	24.6
100	10	6	4237.5	25.5
100	20	6	4237.5	25.5
100	50	42	4223.5	25.7
100	100	42	4223.5	25.7

A.2 Path 2

Table A.2. Simulation results from IDP path 2

Warm-Up Iterations	DP-Iteration	Best Energy Iteration	Energy Consumed	Savings [%]
2	1	1	14453.1	3.8
2	10	error @ 9	-	-
2	20	-	-	-
2	50	-	-	-
2	100	-	-	-
3	1	1	14930.0	0.6
3	10	3	14110.0	6.1
3	20	15	14043.0	6.5
3	50	21	14022.0	6.7
3	100	71	14021.0	6.7
5	1	1	14316.0	4.7
5	10	2	14187.0	5.6
5	20	2	14187.0	5.6
5	50	2	14187.0	5.6
5	100	2	14187.0	5.6
10	1	1	14342.1	4.5
10	10	10	14233.1	5.2
10	20	19	14158.9	5.8
10	50	24	14118.3	6.0
10	100	24	14118.3	6.0
25	1	1	14258.6	5.1
25	10	8	14124.4	6.0
25	20	14	14050.8	6.5
25	50	25	14032.3	6.6
25	100	25	14032.3	6.6
50	1	1	14748.0	1.8
50	10	9	14104.5	6.1
50	20	14	14058.4	6.4
50	50	46	14018.2	6.7
50	100	46	14018.2	6.7
100	1	1	14621.4	2.7
100	10	5	14081.7	6.3
100	20	20	14056.8	6.4
100	50	29	14029.5	6.6
100	**100**	**58**	**13990.8**	**6.9**