TECHNISCHE UNIVERSITÄT DORTMUND

Faculty of Computer Science

DESIGN AUTOMATION FOR EMBEDDED SYSTEMS GROUP

Master Thesis

System Optimization of Hybrid Marine Systems

Supervisors:	Prof. Dr. Jian-Jia Chen
	Dr.-Ing. Philip Nagel
Submitted by:	Osama Maqbool
Student ID Number:	181037
Handover of the Topic:	18.10.2016
Submitted on:	18.04.2017

Eidesstattliche Versicherung

Maqbool, Osama Matr.-Nr. 181037

Ich versichere hiermit an Eides statt, dass ich die vorliegende Masterarbeit mit dem Titel '**System Optimization of Hybird Marine Systems**' selbstständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate kenntlich gemacht. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Ort, Datum Unterschrift

Belehrung:

Wer vorsätzlich gegen eine die Täuschung über Prüfungsleistungen betreffende Regelung einer Hochschulprüfungsordnung verstößt, handelt ordnungswidrig. Die Ordnungswidrigkeit kann mit einer Geldbuße von bis zu 50.000,00 € geahndet werden. Zuständige Verwaltungsbehörde für die Verfolgung und Ahndung von Ordnungswidrigkeiten ist der Kanzler/die Kanzlerin der Technischen Universität Dortmund. Im Falle eines mehrfachen oder sonstigen schwerwiegenden Täuschungsversuches kann der Prüfling zudem exmatrikuliert werden. (§ 63 Abs. 5 Hochschulgesetz - HG -)

Die Abgabe einer falschen Versicherung an Eides statt wird mit Freiheitsstrafe bis zu 3 Jahren oder mit Geldstrafe bestraft.

Die Technische Universität Dortmund wird gfls. elektronische Vergleichswerkzeuge (wie z.B. die Software „turnitin") zur Überprüfung von Ordnungswidrigkeiten in Prüfungsverfahren nutzen.

Die oben stehende Belehrung habe ich zur Kenntnis genommen:

Ort, Datum Unterschrift

Acknowledgements

Firstly, I would like to express my gratitude to my supervisor Prof. Jian Jia Chen for the support and guidance during the Master thesis. Secondly, I would sincerely like to thank Dr. Philip Nagel for his supervision. His guidance helped me in all the time of research and writing of this thesis. I could not have imagined having a better advisor and mentor for my Master thesis.

Besides my supervisors, I would especially like to thank Mr. Nguyen Quang Huy for his support and suggestions. The research carried out would not have been possible without his valuable insights.

Lastly, I would like to express my gratitude to the CR/AEI department at Robert Bosch GmbH for providing me the opportunity to work at their project, and granting me access to the research facilities.

Abstract

Dynamic Programming is a highly effective technique for the offline optimization of power trains, as it guarantees a globally optimal solution. This study investigates the two major problems associated with the application of dynamic programming for the optimization of hybrid marine power trains.

The first issue is the high number of computations, which increases exponentially with the size of the problem. In order to reduce the number of computations, a variant of dynamic programming is investigated, called the iterative dynamic programming. The implementation problems with iterative dynamic programming have also been investigated and solutions are proposed to tackle the problems.

The second issue is the loss of optimality due to the discretization of the continuous power train model. The introduction of different errors and their propagation through the optimization process is investigated. Multiple solutions are proposed in this area that aim to reduce the discretization errors without increasing the number of computations.

In order to test and experimentally verify the investigations, two test cases of hybrid power trains are defined. The regular dynamic programming algorithm, as well as the proposed methodologies for improving the performance of the algorithm are implemented and tested for the two test cases. The performance of the test cases for different algorithms is presented in a comparative fashion.

Table of Contents

List of Figures

List of Tables

List of Symbols and Abbreviations

Symbol	**Description**	**Unit**
x_{SoC}	State of charge of battery	%
x_{gen}	Generator state whether it is on or off	-
u_{bat}	Control input to battery	Amperes
u_{gen}	Control input to generator	Amperes
u_{brk}	Control Input to brake resistance	Amperes
z	Load cycle/ Load requirement	Watts
I_{bat}	DC current from battery to DC grid	Amperes
I_{gen}	DC current from generator to DC grid	Amperes
I_{brk}	DC current from brake resistance to DC grid	Amperes
I_{load}	DC current equivalent of the load requirement	Amperes
π	Control policy	-
g	Fuel consumption (greedy cost)	Grams
J	Cost-to-go	Grams
SoC	State of Charge	-
dpm	Implemented dynamic programming function	-
IDP	Iterative Dynamic Programming	-

1 Introduction

Fuel efficiency of marine systems is one of the top concerns of the marine community, due to the recent shifts in fuel costs and income rates [1]. In addition to the fluctuating fuel prices, the increasing restrictions related to pollution and emissions has further increased the demand for development of vessels with flexibility in terms of optimizing fuel costs.

1.1 Motivation

The hybrid power train is a promising alternative to classic diesel engines with regards to reducing fuel costs as well as emissions, as it allows the usage of electric batteries in addition to existing power train components [2]. Due to the addition of degrees of freedom, hybrid power trains open up the possibilities of searching for optimal power management strategies. There already exists a vast literature on the optimal power management strategies for hybrid electric vehicles, for example in [2; 3; 4], and the concepts investigated within can be extended to hybrid marine systems.

1.2 Prior Art

The upgrade from a diesel engine to a hybrid engine can provide the designer with various possible system topologies. The Power Management Tool, developed by Bosch GmbH, aims to aid the designer by computing and comparing the optimal power management stratregies for different topologies of marine power trains. As the real time control is not a target of the Power Management Tool, the load cycle for the entire optimizaion interval can be known apriori.

The Power Management Tool uses dynamic programing for optmization of power management stratregy, which is the ideal candidate for systems with perfectly known load cycles and disturbances. Dynamic programming offers the advantage of guaranteeing the globally optimal solution for the given problem.

1.3 Goals of the Project

There are two major issues that are associated with the implementation of dynamic programming to continuous state space systems. Both the issues arise from the discretization of the continuous state space system, which is necessary to formulate it as part of the dynamic programming optimization problem. The first issue is the high number of computations involved in optimization, which increases exponentially as the problem size increases. The second issue is the mismatch between the discretized and continuous state space systems, which leads to sub-optimal solutions from dynamic programming.

The goal of this work is the study of two issues, and investigation of methodologies that attempt to solve them.

1.4 Structure of Thesis

The thesis is structured as follows. The modeling of marine power trains for the Power Management Tool is introduced in Chapter 2, and two test vessels are defined for demonstration and experimentation. The dynamic programming algorithm, its theory and implementation is discussed in Chapter 3. In Chapter 4, an alternate of dynamic programming is introduced, iterative dynamic programming, which aims to reduce the computational complexity of the optimization procedure. The various errors involved with the application of dynamic programming to discretized drive trains are studied in Chapter 5. Several techniques to minimize the errors are also investigated. The simulation and experimentation results for the two test vessels are compiled in Chapter 6. Finally, the conclusions and further ideas are summarized in Chapter 7.

2 Power Management of Marine Drive Trains

The Power Management Tool is designed to aid in assessment of various layouts of marine vessel drive trains with respect to the fuel efficiency. In addition to the fuel consumption, it also allows the identification of relevant power bonds across various components of the drive trains. The results can further be used for the sensitivity analysis of the system with respect to the size and parameters of various components.

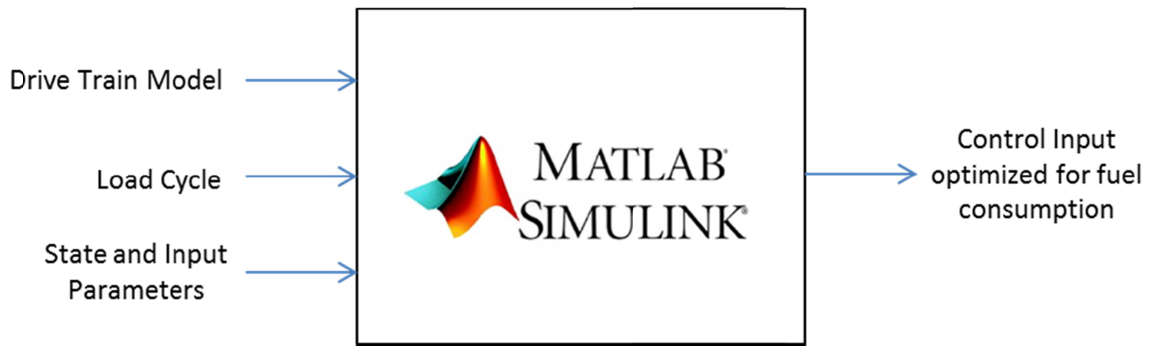
2.1 Power Management Tool

The Power Management Tool calculates the trajectory of control inputs for a drive train model that minimizex the fuel consumption for a given load cycle. The tool provides a Simulink library that includes models for various components involved in marine drive trains. The table below summarizes the types of components included in the tool. Each category is further divided into multiple types of models, and the parameters for each model can be set by the user.

Table 2.1 Drive train components included in the Power Management Tool

Power Sources/ Sinks	**Actuators**
Diesel Engine	Electric Motor
Electric Storage	Hydraulic Machine
Brake resistance	

The complete model of the drive train must follow the conventions discussed in the next section. The load cycle must be specified for every point in the complete optimization interval. In addition to the model and the load cycle, the bounds and discretization of the state and input vectors also need to be specified by the user.


Drive Train Model
Load Cycle
State and Input Parameters
MATLAB SIMULINK
Control Input optimized for fuel consumption

The result from the tool is an optimal control input trajectory, calculated for the minimum fuel consumption for the complete load cycle. For the optimal input trajectory, the resultant state trajectory, power flow across different components and resultant loses, and the fuel consumption trajectory are also presented. The performance of various system layouts can be compared, and can be used as a broad guide in the design of drive trains, a feasibility study for system upgrades, or isolation of drive trains with higher losses.

2.2 Modeling of Drive Trains

The Power Management Tool uses *backward modeling* of the drive train. The simulation in backward modeling uses the torque and speed requirement on the actuators to calculate the required energy which must be provided by the power sources. The torque and speed requirement are specified by the user as the load requirement. An example of the backwards modeling of a drive train is shown in Figure 2.2. The load requirement at the actuators is converted into a DC current demand which is fed to the DC grid. The convention set in the power management tool is that positive value of the DC current to indicate the demand. The control inputs of the system are the currents supplied by the power sources to fulfill the load requirement from the actuators. Each of the power sources has a nominal voltage specified as a parameter.

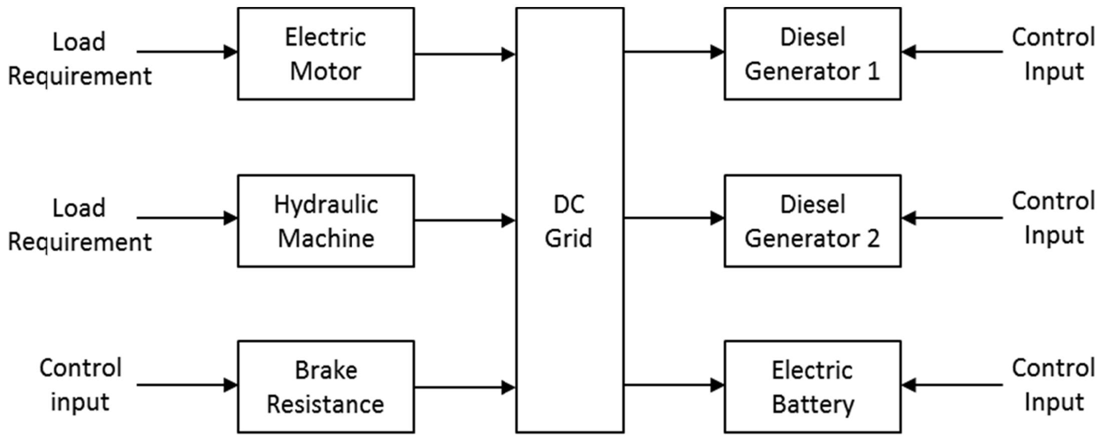


Figure 2.2 Backwards modelling of Drive Trains

In addition to the control inputs, the states of the system are also associated with the power sources. The general states and inputs of the system are:

$$\underline{x} = \begin{pmatrix} x_{SoC} \\ x_{gen,1} \\ \cdot \\ \cdot \\ \cdot \\ x_{gen,n} \end{pmatrix}, \qquad \underline{u} = \begin{pmatrix} u_{bat} \\ u_{brk} \\ u_{gen,1} \\ \cdot \\ \cdot \\ \cdot \\ u_{gen,n} \end{pmatrix} \tag{2.1}$$

x_{SoC} refers to the State of Charge (SoC) of the electric battery. The application of a positive control input u_{bat} controls the current supplied to the battery (charging) and a negative u_{bat} controls the current supplied by the battery (discharging). The brake resistance acts as a sink for extra current in the system, and the current delivered to the brake resistance is controlled by u_{brk}. u_{brk} can only have a positive value, as the brake resistance cannot supply any current. For n diesel generators, $u_{gen,1}$ to $u_{gen,n}$ determine the current supplied by each corresponding generator. The control input of the generator is always negative, as the generator cannot input any power. $x_{gen,1}$ to $x_{gen,n}$ are binary states, and represent whether the corresponding generator is on or off. The user is free to choose the number of generators, and whether to include a battery or not in the drive train model. The control inputs are not directly fed to the DC grid. The general relation between the control inputs, load requirement and the corresponding DC currents at the DC grid can be characterized by the following equations:

$$\begin{pmatrix} I_{g1} \\ I_{g2} \\ \cdot \\ I_{gn} \\ I_{bat} \\ I_{brk} \end{pmatrix} = \underline{d}(\underline{u}, \underline{x}) \tag{2.2a}$$

$$I_{load} = h(\underline{z}) \tag{2.2b}$$

where z represents the load requirement. The functions $\underline{d}$ and h depend on the dynamics of the components modeled and are associated with various parameters that can be set by the user respectively. These are not relevant for the studies in this thesis, and are therefore not discussed further.

The net DC current at the DC grid must be zero to ensure that the load requirement is satisfied.

$$I_{g0} + \sum_{i=1}^{n} I_{gi} + I_{bat} + I_{brk} + I_{load} = 0 \tag{2.3}$$

For n generators included in a model, an extra generator called *balance generator* must also be included, which is incorporated separately in Equation (2.3) as I_{g0}. The balance generator has no state or control input associated with it. As the control inputs can only take values from a discrete range, the corresponding DC currents are also from a discrete set of values. It is possible that no combination of the possible discrete values of the DC current exactly satisfies the user defined load requirement. The DC current from the balance generator I_{g0} can take a value from continuous range to ensure that the net DC current is zero. The balance generator does have a maximum limit and can only assume negative values. Thus the only feasible control inputs to the system for a given load requirement are ones that allow Equation (2.3) to be satisfied.

Specification of Load Cycle

The load cycle is the load requirement on the actuators of the model for the complete length of the optimization interval. It is specified by a speed and torque vector for every actuator. The time interval for optimization must be discretized, and a load requirement must be defined for every point in time.

Simulation of Model

The Simulink model can be called in a Matlab script as a function. At any time point k, the modeled drive train can be described by the following equations:

$$\underline{x}' = F(\underline{x}_k, \underline{u}_k, z_k) \tag{2.4a}$$

$$g_k = g(\underline{x}_k, \underline{u}_k, z_k), \tag{2.4b}$$

$$I_k = I(\underline{x}_k, \underline{u}_k, z_k) \tag{2.4c}$$

$\underline{x}_k \in X_k$ represents the states, $\underline{u}_k \in U_k$ represents the control inputs and z_k the user defined load cycle. F is the transition function for a control input, g determines the fuel consumption for an input, and I decides whether a control input is feasible. The ranges of the states and inputs must be defined, within which they can only assume a finite set of values. X and U represent the discrete spaces for states and inputs respectively. Since F is a continuous function, $\underline{x}' \in \chi$, where χ represents the continuous state space.

2.3 Test Cases

For the study of the optimization procedures used in the Power Management Tool, two test cases are defined. The two cases will be used to test the developed algorithms throughout the thesis.

Vessel 1

The Simulink model for the test case named ‘Vessel 1’ is shown in Figure 2.4. Vessel 1 has no generator state, and only uses the balance generator. The only state is the battery State of Charge (SoC), and the control inputs are the battery current and the brake resistance current:

$$x = (x_{SoC}), \qquad \underline{u} = \begin{pmatrix} u_{bat} \\ u_{brk} \end{pmatrix} \tag{2.5}$$

Vessel 2

The Simulink model for the test case named ‘Vessel 1’ is shown in Figure 2.5. The states and the control inputs are:

$$\underline{x} = \begin{pmatrix} x_{SoC} \\ x_{gen} \end{pmatrix}, \qquad \underline{u} = \begin{pmatrix} u_{bat} \\ u_{brk} \\ u_{gen} \end{pmatrix} \tag{2.6}$$

The objective of including the state of the generator is to include a penalty for turning it on. The efficiency of the generators used in both the vessels is a non-linear function shown in Figure 2.3(a). This efficiency curve results in the fuel consumption functions of the vessels shown in Figure 2.3(b), based on the sizes of the generators set by the user.

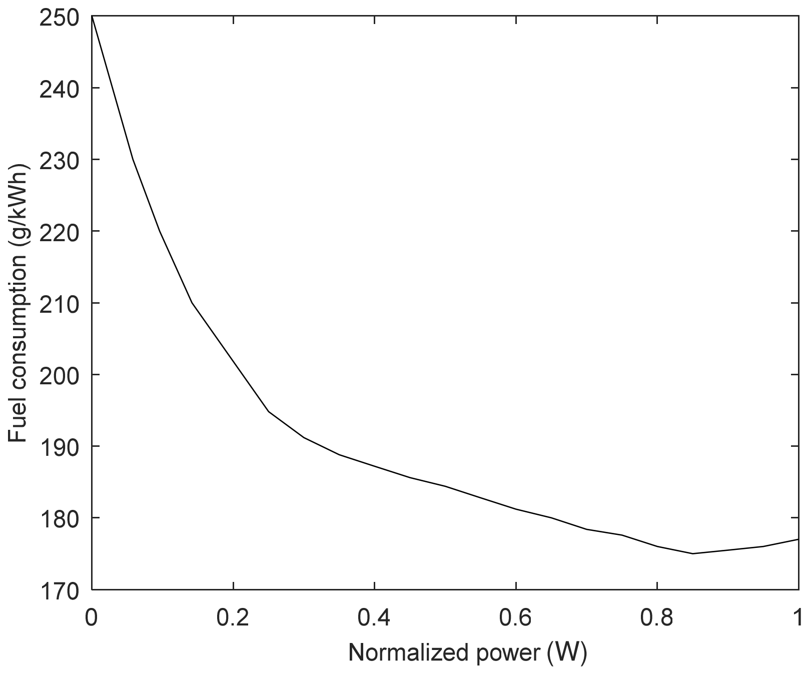


Figure 2.3(a) Efficiency of the generators (Power normalized to maximum capacity)

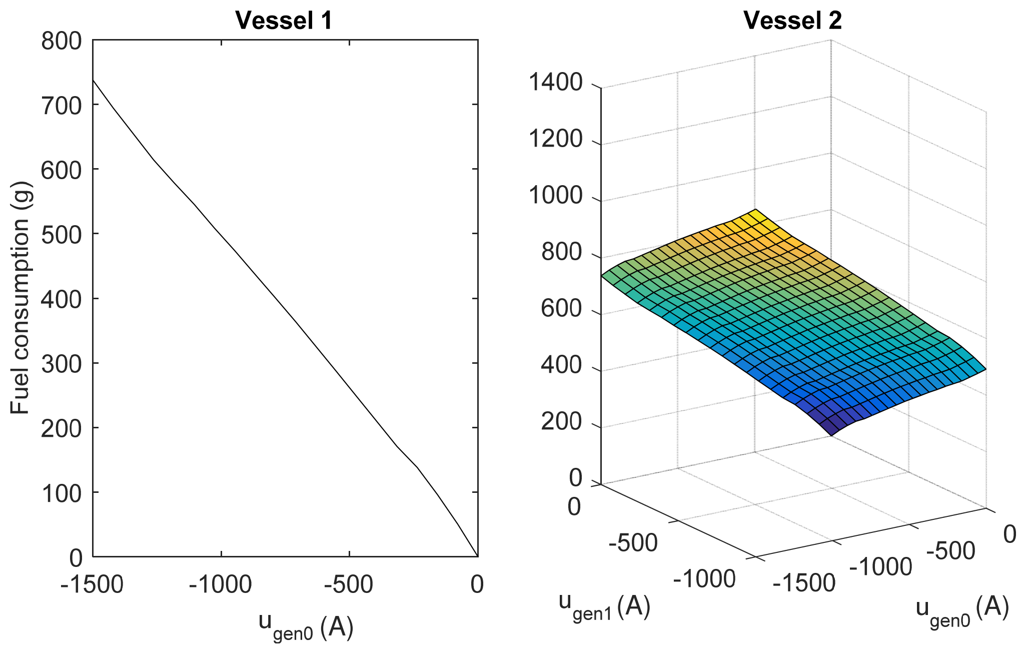


Figure 2.3(b) Fuel consumption curves of vessel 1 and 2

Figure 2.4 Simulink model of vessel 1

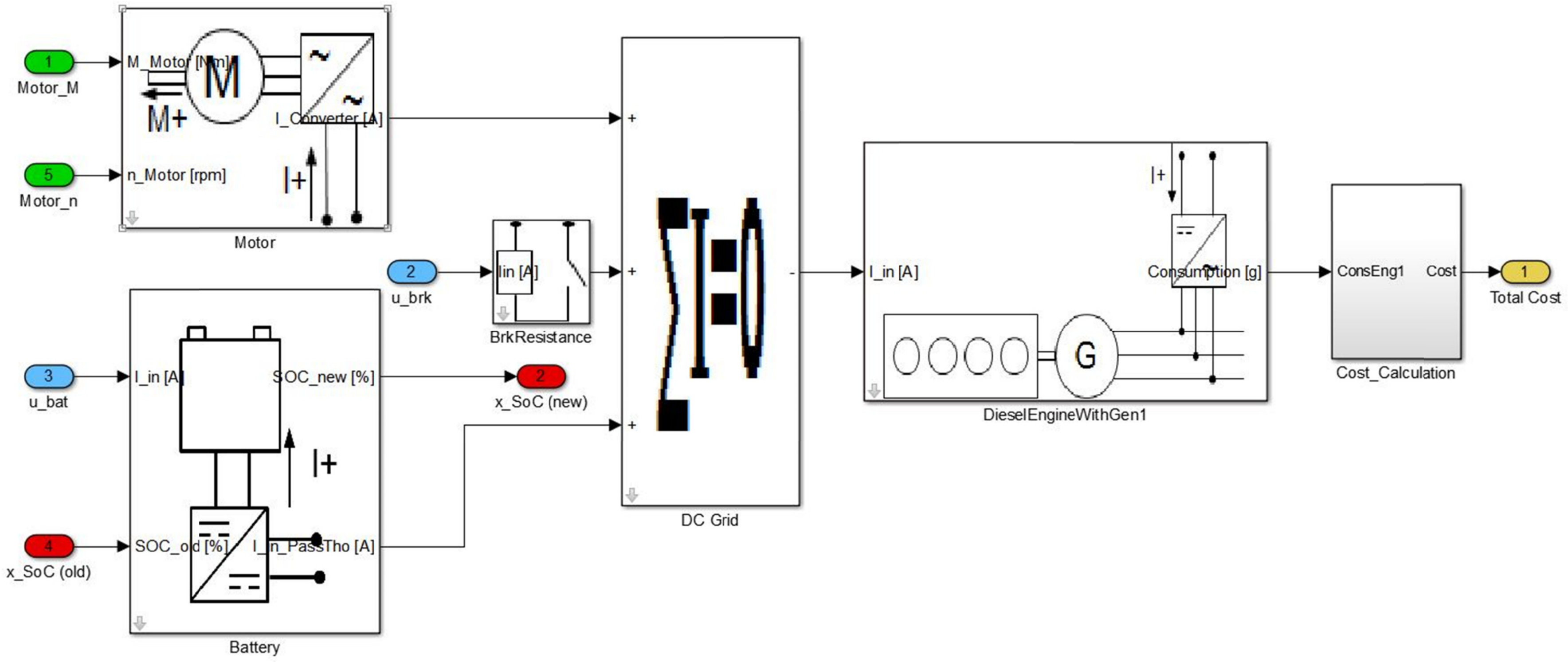
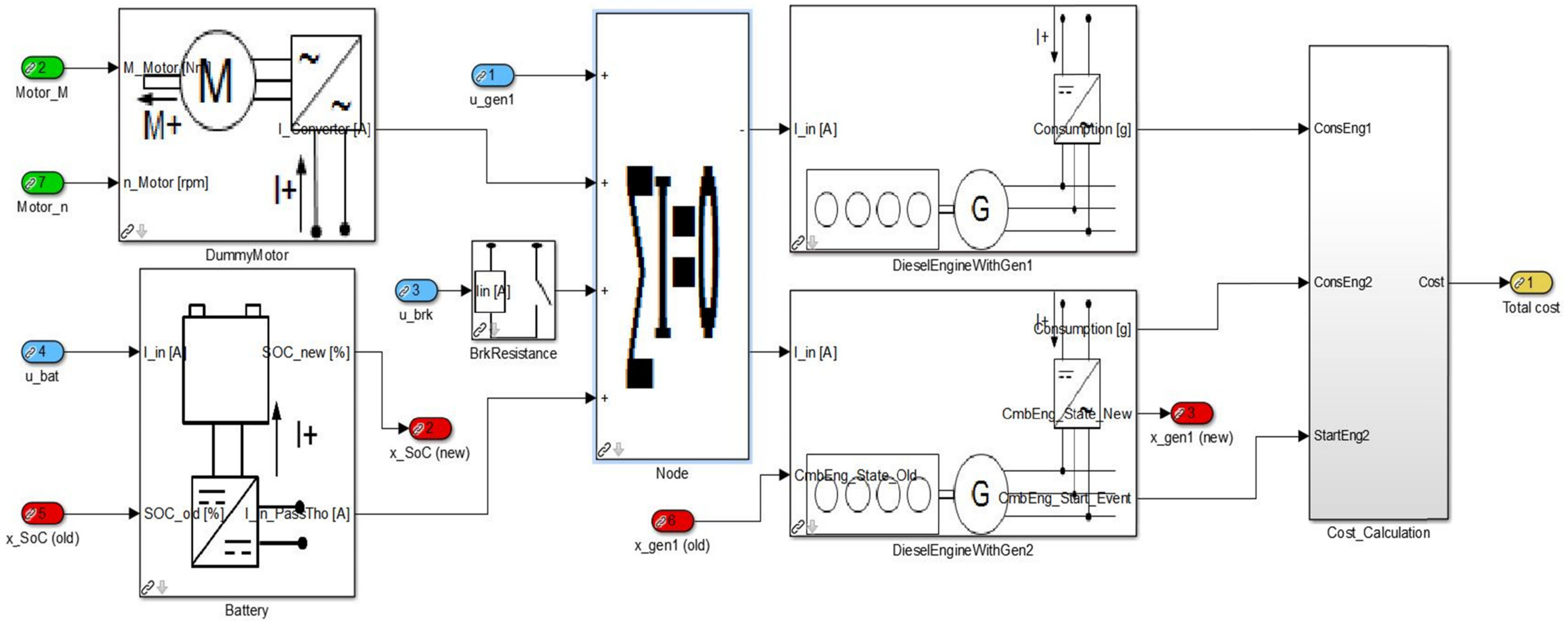


Figure 2.5 Simulink model of vessel 2



3 Optimization with Dynamic Programming

The technique for optimization used in the Power Management tool is Dynamic Programming. Dynamic programming is an optimization technique which divides a complex problem into a sequence of simpler sub-problems. The sub-problems are optimized recursively, using the previous solutions, to calculate the optimum for the original problem. Dynamic programming is a powerful optimization technique used in many fields of study. The biggest advantage of the algorithm is that it guarantees a globally optimum solution for the given conditions of a problem.

Dynamic programming is a popular approach used in the Optimization of Drive Trains that involve electric components in addition to diesel generators (Hybrid Vehicles) [3] [4]. A variety of goals for optimization can be set related to the safety, comfort and driving dynamics. For the Marine Drive Systems relevant in this thesis, the goal of optimization is the fuel consumption. The tool used for Dynamic Programming, called the dpm function, has been developed in [5] for MATLAB.

3.1 Optimization of Marine Drive Trains

For the discretized state-space model of the Drive Trains in Equations (2.4), the control policy π dictates the actions for every stage in the horizon T.

$$\pi = \{\underline{u}_0, \underline{u}_1, \dots, \underline{u}_{T-1}\} \text{ for } \pi \in \Pi \tag{3.1}$$

The set Π spans all the possible control policies for the system. The problems in this thesis are assumed with a perfect knowledge of the parameters and disturbances $\underline{w}_k$ for the complete horizon N of the optimization interval. The total cost for the time horizon of N, for a control policy π can be formulated as:

$$J_\pi = \sum_{k=0}^{N-1} g_{k,\pi} \tag{3.2}$$

The total cost is the total fuel consumption that results from a control policy for the complete optimization interval horizon. The goal of optimization is to find the optimal policy π^* that minimizes the total fuel consumption for the complete duration of the optimization interval:

$$J_\pi^* = \min_\pi J_\pi \tag{3.3}$$

The formulation of the total cost function J_π is quite useful, as it includes the costs g_k for the complete horizon of the optimization interval. The alternative is to only minimize the current cost for a stage g_k with no regard to the future stages. This technique is commonly known as the *greedy approach*. For hybrid Drive Trains, the greedy approach would discharge the battery at every step, as the battery does not consume fuel. Due to the inclusion of the future parameters $z_{k+1}\colon z_N$ in addition to the current z_k, the optimal approach would not always be the full discharge of the battery, but rather to *save* the battery for high load requirements in the future. Thus it represents the optimal Power Management strategy for hybrid Drive Trains for a discretized interval of N steps, given that the parameters and disturbances $\underline{w}$ are known for every stage of the interval.

3.2 Principles of Dynamic Programming

The Dynamic programming algorithm defines a *multi-stage decision process* for the optimization problem in Equation (3.3). A discretized state vector $\underline{x}_k$ of the system is defined at every stage. $\underline{u}_k$ is a permissible action at the stage k . A feasible set of actions must exist for the state $\underline{x}_k$ at any given stage k. Each action $\underline{u}_k$ causes a transition of the state $\underline{x}_k$ into one of the system states $\underline{x}_{k+1}$ defined for the next stage $k+1$. Each action also has a cost associated with it. To ensure the applicability of dynamic programming, the properties of the system modeled in the problem must fulfill the following conditions as described by Bellman [6] [4]:

- The decisions in a stage should not have any influence from the already considered stages or their stages.
- Except the state of the system, the history of the decisions on the system has no influence on the selection of future actions.

The systems considered in this study are deterministic and every action to a state determines a unique transition. The deterministic multi-stage decision process for N stages is demonstrated in Figure 3.1.

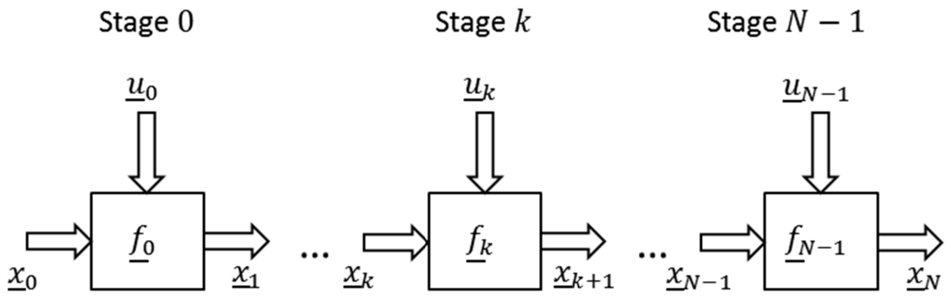


Figure 3.1 Multi-stage decision process [4]

The Dynamic programming approach for the optimization of the multi-stage decision process is based on the Bellman's principle of optimality.

Principle of Optimality

Bellman describes the property of the optimal policy defined for the systems with characteristics defined in the above section. Coined as Bellman's principle of optimality, it states that:

„*An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision.*" - Principle Of Optimality

The principle of optimality for three stages is demonstrated in Figure 3.2. g_{1a} represents the cheapest action that causes x_1 to transition to x_2. The cheapest policy that transitions from x_0 to x_2 can only be formulated by including g_{1a}.

Generally speaking, if the optimal policy from stage 0 to $N-1$ $\pi_0^* = \{\underline{u}_0, \underline{u}_1, \dots, \underline{u}_{N-1}\}$ is known, it can be used to find the optimal policy from any stage $k = i$, $\pi_i^* = \{\underline{u}_i, \dots, \underline{u}_{N-1}\}$ [3]. This allows the formulation of a *cost-to-go* for every stage.

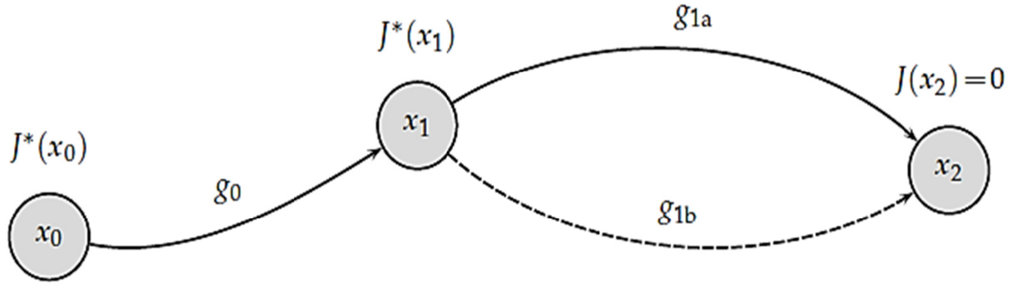


Figure 3.2 Example of Bellman's principle of optimality [3]

$$J^*(\underline{x}_i) = J^*_{\pi_0^*}(\underline{x}_i) = \min_{\pi \epsilon \Pi}\{\sum_{k=i}^{N-1} g_k(\underline{x}_k, \underline{u}_k, z_k) + g_N(\underline{x}_N)\} \tag{3.4}$$

g_N describes the costs associated with the last stage, while g_k describes the cost by applying an action $\underline{u}_k$ to the state $\underline{x}_k$.

Using the principle of optimality, dynamic programming solves the problem by backward recursion from stage N, a principle also termed as *backward induction.*

$$J_N(\underline{x}_N) = g_N(\underline{x}_N) \tag{3.5a}$$

$$J_k(\underline{x}_k) = \min_{\underline{u}_k}\{g(\underline{x}_k, \underline{u}_k, z_k) + J_{k+1}(\underline{x}')\} \tag{3.5b}$$

Starting from the last stage N, backward induction calculates the optimal cost from each stage to the end stage. The usage of cost-to-go requires the optimization of the transitions from only the current stage to the next. Cost-to-go is the optimal cost achievable from a stage k, and is defined as a function of the state $\underline{x}_k$.

The backward induction is not the only method in Dynamic programming. A forward induction, that starts from $g_0(\underline{x}_0)$, can be used in a similar way. Backward induction is more commonly used as it tackles the problem defined for a state space in a more intuitive manner.

3.3 Application to Drive Trains

Equation 3.6a and 3.6b very aptly depict the application of Dynamic programming as a backward recursive algorithm for the power management optimization of Drive Trains. For each stage of the multi-stage decision process in the Drive Train, the parameter $\underline{w}_k$ is defined as a scalar load requirement z_k. The set of feasible actions at any stage must be able to fulfill the load requirement for the stage without violating any state bounds. The implementation of Dynamic programming can be intuitively explained by its application to the test vessels defined in Section 2.3.

3.3.1 Optimization of Vessel 1

Vessel 1 defines a Drive Train with a single-dimensional state space.

a. Discretization of State and Input Space

The discretization of the SoC state is a decision taken by the user of the Power Management Tool. Thus for each state, the discretization creates multiple *points* of a state within a stage. The examples in this section have the SoC discretized into three points.

For the demonstration of the application of Dynamic programming, the input u_{brk} is ignored for the sake of simplification. The battery input is discretized into three points.

$$U = \{u_1, u_2, u_3\} \tag{3.6}$$

u_1 increases the SoC (charging), u_2 keeps SoC the same, and u_3 decreases the SoC (discharging)

Despite the discretization of the state and input space, the transition function in $\underline{x}' = F(\underline{x}_k, \underline{u}_k, z_k)$ still has a continuous range.

b. Specification of Load Cycle

The load cycle for the optimization interval is defined as:

$$z = \{high, high, low, low\} \tag{3.7}$$

A $high$ load requirement is higher than the maximum capacity of the balance generator, and the battery must be discharged to fulfill the requirement. A low load requirement is lower than both the maximum capacities of the balance generator and the battery, and can be satisfied by only discharging the battery.

c. Formulation of Sub-problem

The backward-induction algorithm described by Equation (3.5) can be applied to the discretized states and inputs of Vessel 1. Due to the discretized state points, the sub-problem at each stage can be further divided into smaller problems. Furthermore, no costs for the end stage N are modelled in the systems studied in this thesis. The optimization algorithm can therefore be divided into a set of the following sub-problems, generalized for any n-dimensional state space:

$$J_N(\underline{x}_N) = 0 \tag{3.8a}$$

$$J_k(\underline{x}_k^i) = \min_{\underline{u}_k}\{g(\underline{x}_k^i, \underline{u}_k(\underline{x}_k^i), z_k) + J_{k+1}(\underline{x}')\}, \ \underline{x}_k^i \in X_k, \underline{u}_k \in U_k, \underline{x}' \in \chi_{k+1}$$

such that

$$\underline{x}' = F(\underline{x}_k, \underline{u}_k, z_k)$$

$$I(\underline{x}_k, \underline{u}_k, z_k) = 0 \tag{3.8b}$$

x_k^i represents the state point i for the discretization done at stage k.The set of feasible control inputs $\underline{u}_k$ for stage k result in a set of transitions defined by Equation (3.8).

d. Backwards Induction

The backwards induction method used for the optimization of the fuel consumption for Vessel 1 is demonstrated in Figure 3.3. The figure is a schematic to demonstrate the process of Backwards Induction, and any values shown are only an approximation of the actual system.

1. For stage $N-1$, there is no cost-to-go of the future stages. The cost of all the feasible inputs are computed for every state point x_{N-1}^{i}, and the minimum cost is saved for the state point, forming the cost-to-go function for stage $N-1$, $J_{N-1}(x_{N-1})$. The minimum costs, or equivalently the saved cost-to-go's are displayed within the circles in Figure 3.3. For the lowest state point at any stage, input u_3 is infeasible as the SoC cannot go below 0%.

$$\text{for } x_n^5\,, \qquad U_n = \{u_1, u_2\}$$

2. For each state point x_{N-2}^{i}, the optimal input minimizes not just the cost, but the cost and the cost-to-go. The cost-to-go J_{N-1} is only saved for the discretized state space $x_{N-1} \in X_{N-1}$. The transitions from u_1 and u_3 however, do not belong to the discretized state space X_{N-1}. There can be several methods to calculate the appropriate cost-to-go for the continuous state space $J_{N-1}(F(x_{N-2}, u, z_{N-2}))$, such as using the nearest neighbors or using linear interpolation. The dpm function uses linear interpolation between the neighboring points x_{k+1}.
3. At stage $N-3$, the cost and the cost-to-go is minimized in the same way as the last stage, using linear interpolation where necessary. The load requirement z_{N-3} is higher than the total capacity of the generator, which means that the battery must discharge to fulfill the requirement. Thus for every state point x_{N-3}^{i}, the only feasible input is u_3 which discharges the battery.

$$U_{N-3} = \{u_3\}$$

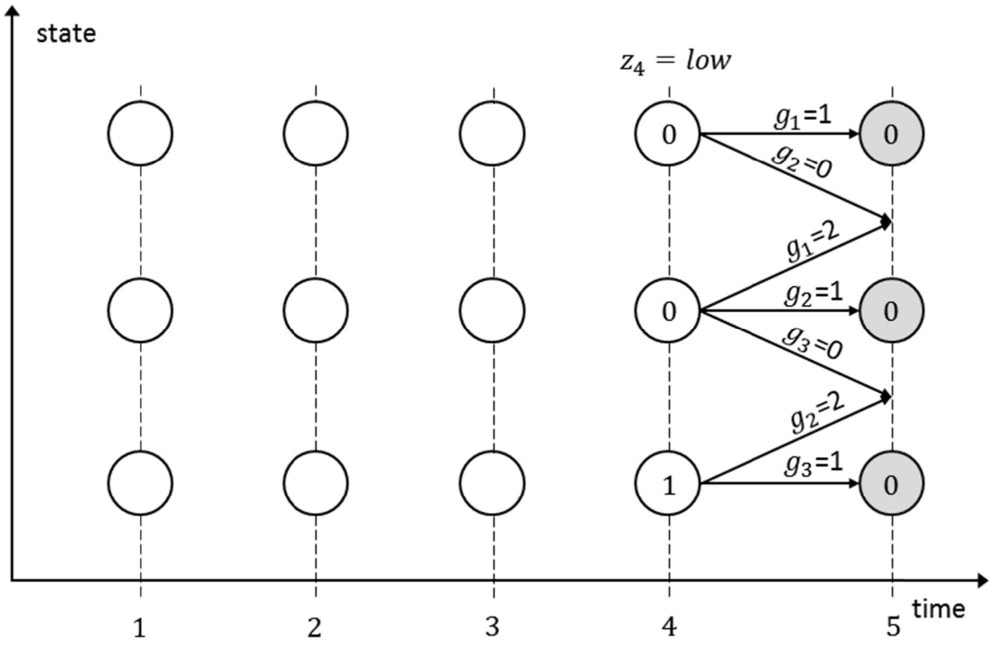


Figure 3.3(a) Backwards induction: step 1

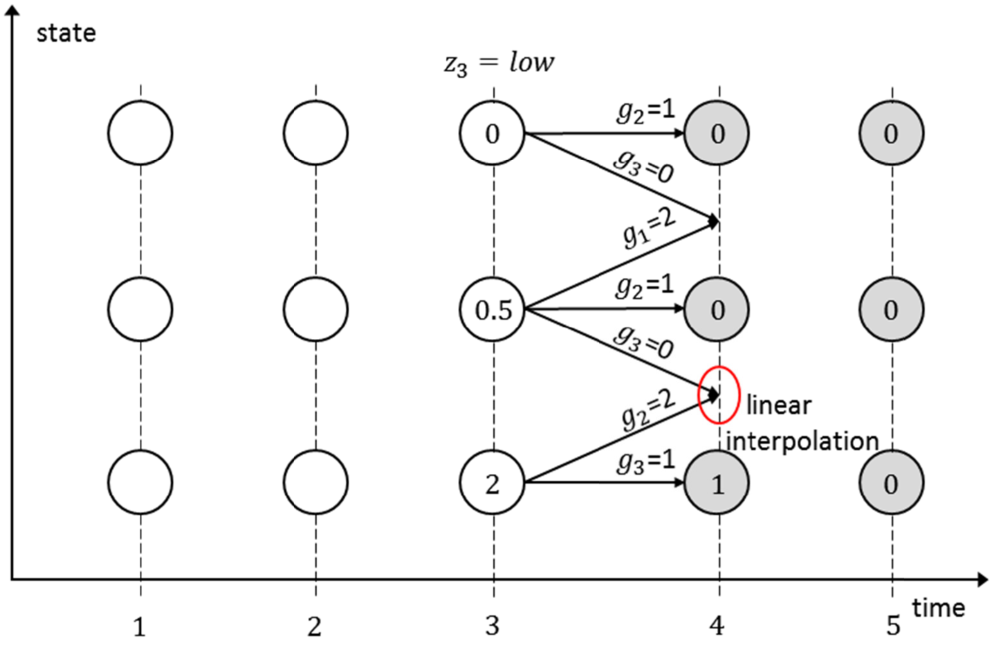


Figure 3.3(b) Backwards induction: step 2

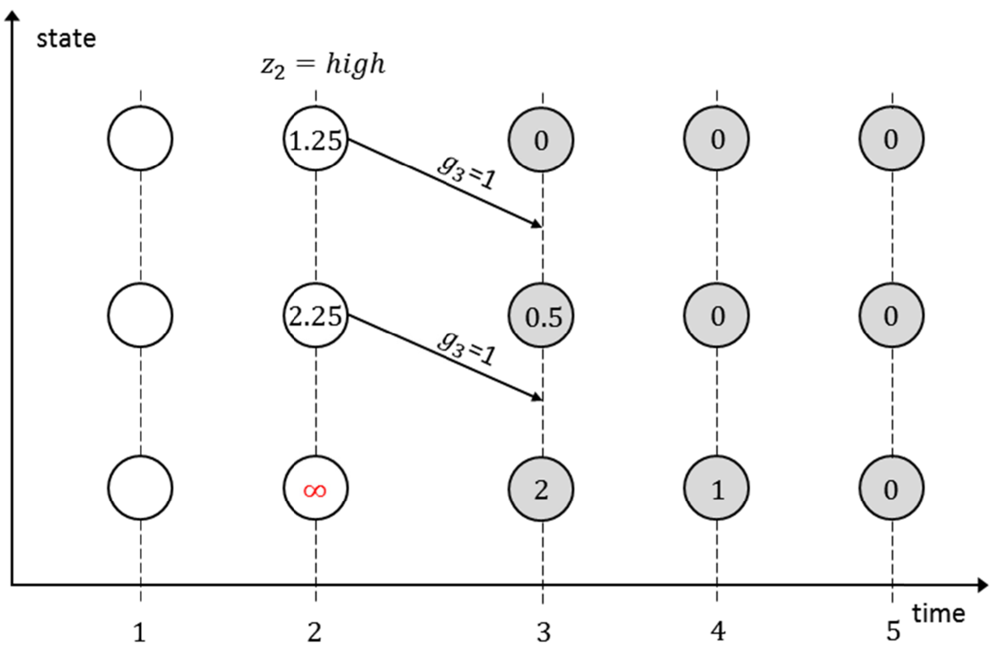


Figure 3.3(c) Backwards induction: step 3

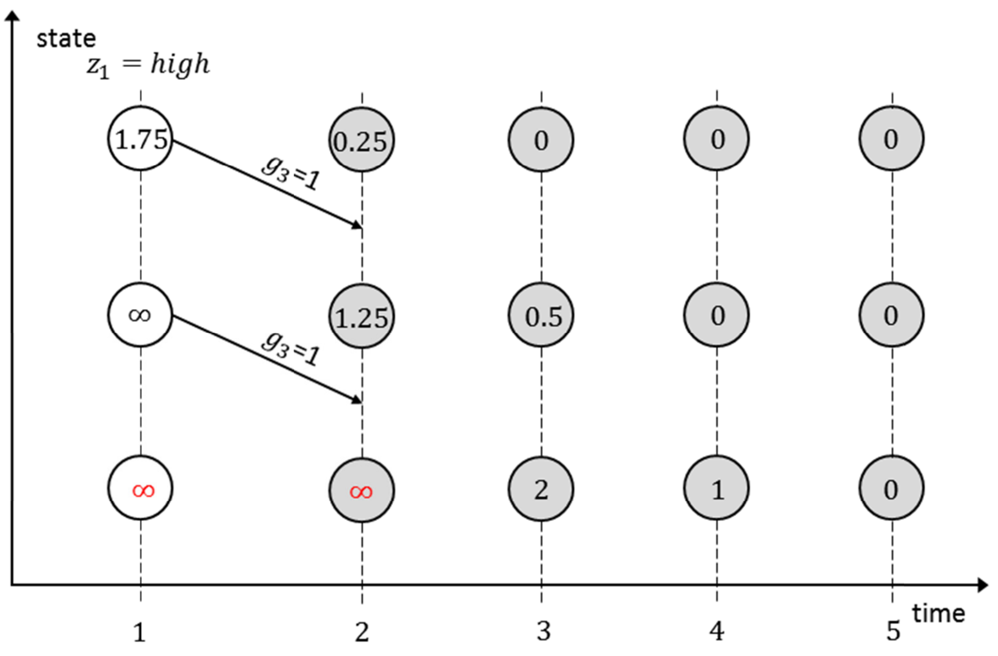


Figure 3.3(d) Backwards induction: step 4

For x_{N-3}^{5}, input u_3 is also infeasible as the SoC cannot discharge below 0%. This state point has no feasible inputs, and it violates the initial condition mentioned for formulation of a multi-stage problem, i.e., every state must have a feasible set of inputs. Thus this state point is infeasible, and the cost-to-go $J_{N-3}(x_{N-3}^{5}) = \infty$. The infinite cost is implemented by a very large number.

4. Stage $N-4$ is the final step of backwards induction, and the first stage for the multi-stage decision process. The load cycle is w_{N-4} is also high, requiring the battery to discharge. The notable phenomenon for this stage occurs for the state point x_{N-4}^{4}. On application of input u_3 to x_{N-4}^{4}, the transition lies in between x_{N-3}^{4} and x_{N-3}^{5}. Thus the cost-to-go $J_{N-3}(x_{N-3}^{4})$ must be linearly interpolated with an infeasible cost (representing infinity) at $J_{N-3}(x_{N-3}^{5})$, to calculate $J_{N-1}(F(x_{N-4}^{4}, u_3, \underline{z}_{N-4}))$, the resultant of which will also be a very high cost. The conditions for the state point x_{N-4}^{5} will be the same as the last step and the point will be treated as infeasible.

The computation at the four stages calculates a map of optimal policies for the complete discretized state space. The optimal state trajectory through the four stages can now be computed for any initial state.

Evolution of Cost-to-Go

The cost-to-go saved at stage $N-1$ for the actual costs in Vessel 1 as well as the optimal control inputs are shown in Figure 3.4a and b. The cheapest action is to discharge the battery as much as the feasible input range allows, and fulfill the remaining load requirement using the generator. The same minimum battery input is feasible for most of the SoC values, which is why a constant cost-to-go is observed for most SoC values. As the SoC approaches 0%, the feasible input range decreases. The closer the SoC moves to 0%, the less it can discharge and the more the generator power is required. This creates a gradient in the cost-to-go at a point close to 0%. Figure 3.5 shows the propagation of the cost-to-go as the optimizer moves backwards through the stages. After stage $N-1$ discharging is not always the best strategy.

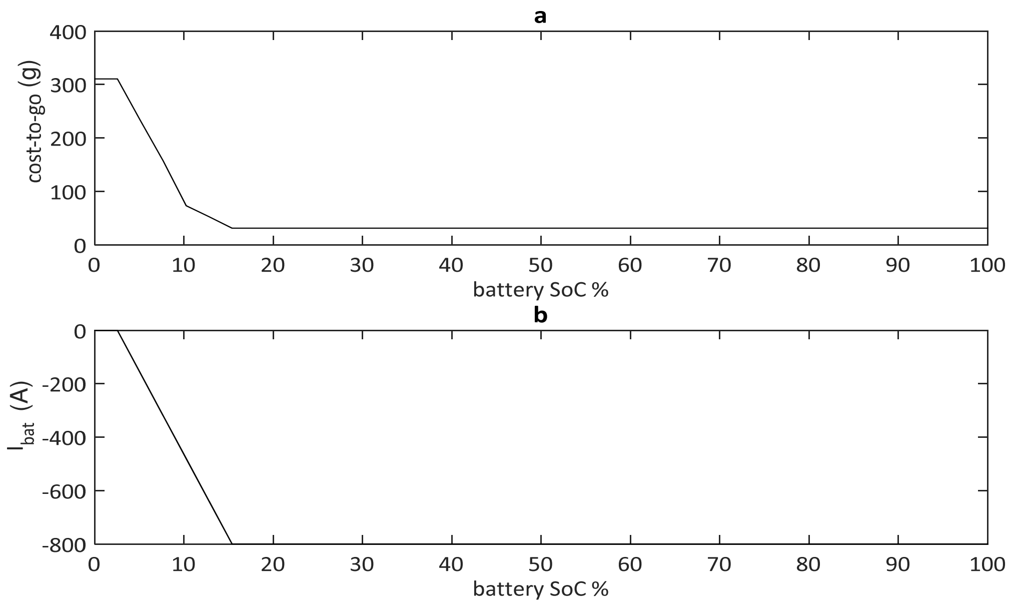


Figure 3.4 Cost-to-go and optimal inputs saved for stage N-1

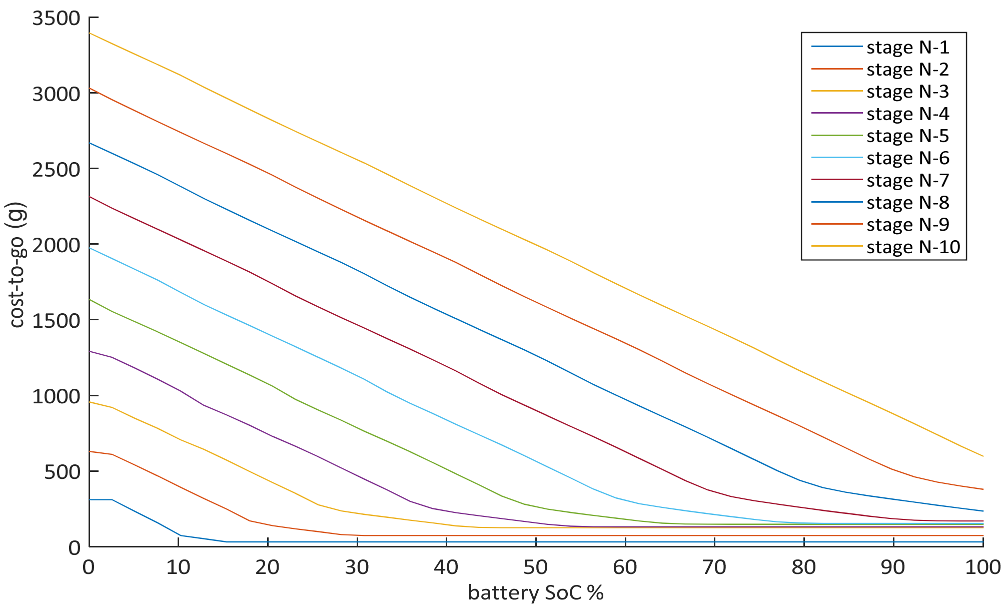


Figure 3.5 Cost-to-go evolution through stages

Discharging will save the fuel consumption costs, but will transition towards a low SoC where the cost-to-go is higher. A high slope, indicating a high future cost propagates towards higher SoC's as previous stages are evaluated.

A difference in the slopes between different regions of the cost-to-go acts as the point of *restriction* to transition into a region.

e. Forwards Run

The forwards run process calculates the optimal control input and the corresponding state trajectory for an initial state by using the cost-to-go map saved in the backwards induction.

1. An initial state point is chosen for stage 1. Let it be $x_{init} = x(1) = 25\%$.
2. The costs for the complete set of feasible control inputs applied to the initial state point is calculated. The optimal input is minimized for both the cost and cost-to-go. The cost-to-go map has already been computed for each state point during the backwards induction. The required linear interpolation also follows the same procedure as before. The optimal control input is saved as the first point of the optimized control input trajectory.
3. The next state point is chosen at stage 2 by applying the optimal control input calculated in the previous step to the initial state.

   $$x_2 = F\big(x_2, u_{opt}(1), z_2\big), x_2 \in \chi_2 \tag{3.9a}$$

   Note that the x_2 may or may not be one of the points of the discretized state grid, as it belongs to the continuous state space. The next optimization is done by applying all the feasible control inputs to x_2. Once again, the optimization is done for both the cost and the cost-to-go.
4. The optimal state and input trajectory is computed for all the stages as described in step 1 and 3.

The recursive algorithm for forwards run can generally be described as:

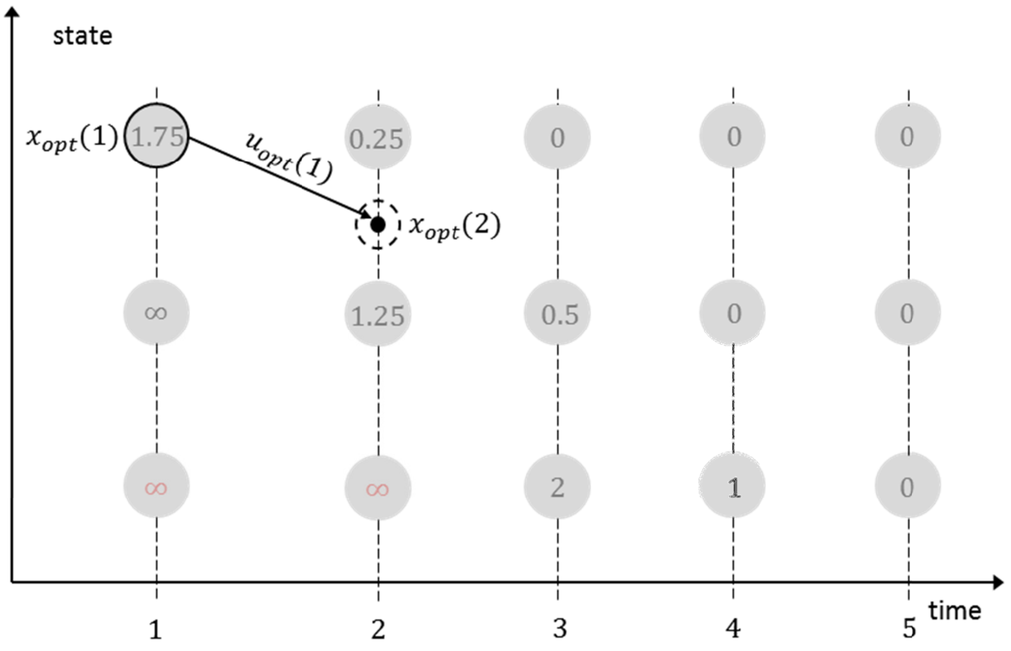


Figure 3.6(a) Forwards induction: Calculation of optimal trajectory using an initial state

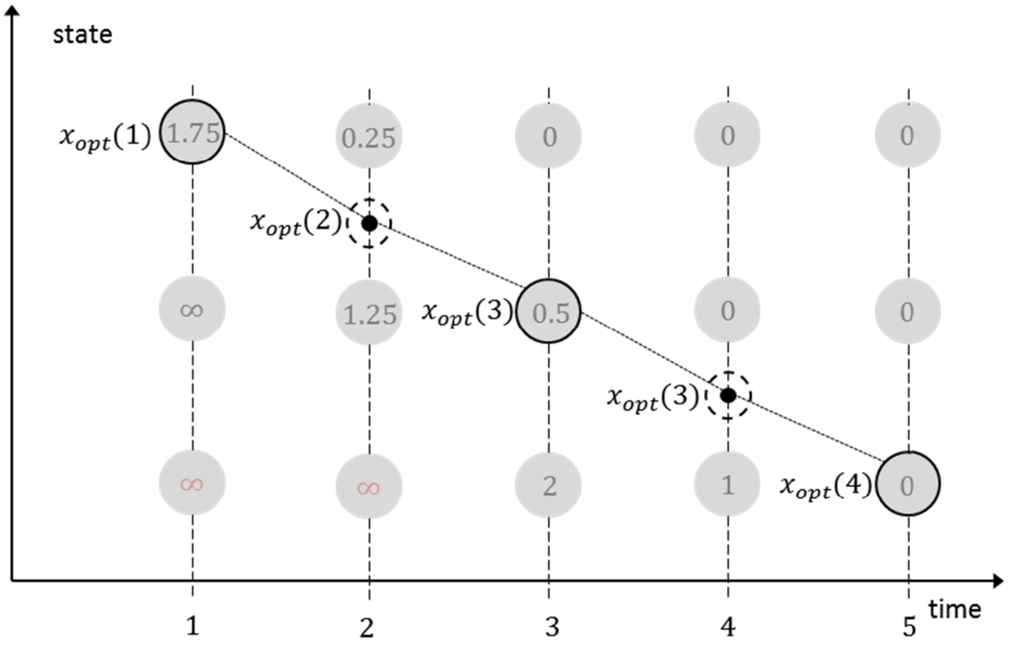


Figure 3.6(b) Forwards inuction: The optimal sate trajectory points lie also outside the defined state grid

$$\underline{x}_{opt}(1) = \underline{x}_{init} \tag{3.10a}$$

$$C(k) = \min_{\underline{u}_k} \left\{ g\left(\underline{x}_{opt}(k), \underline{u}_k, \underline{w}_k\right) + J_{k+1}\left(F\left(\underline{x}_{opt}(k), \underline{u}_k, z_k\right)\right)\right\}, \quad \underline{x}_{opt}(k) \in \chi_k, \underline{u}_k \in U_k \tag{3.10b}$$

$$\underline{x}_{opt}(k+1) = F\left(\underline{x}_{opt}(k), \underline{u}_{opt}(k), \underline{w}_k\right) \tag{3.10c}$$

$\underline{x}_{opt}$ describes the optimal state, C the optimal cost, and $\underline{u}_{opt}$ the optimal input trajectories.

A solution using just the discretized state space points does not need the forward run and can simply be computed using the cost-to-go map for any initial state. The forward run reduces the dependency of the solution on the state discretization and offers a more robust solution.

3.3.2 Computational Complexity

Dynamic programming guarantees a globally optimal solution for a given discretization of the problem, but poses a high computational complexity. BELLMAN referred to the computational complexity of implementing the algorithm as the 'curse of dimensionality' [7]. Given an application of dynamic programming that optimizes a problem by an exhaustive search on all possible solutions, the number of computations for the backwards induction process in a discrete grid is given by:

$$N_{c,back} = (N.L^I.P^K). \tag{3.11}$$

where

N: number of stages

L: Equally spaced intervals of the state space

I: Number of dimensions for the state space

P: Equally spaced intervals of the input space

K: Number of dimensions for the input space

Equation 3.11 describes the number of computations for a problem with equally spaced state intervals and input intervals across the dimensions of state and input respectively and for all the stages of the problem. A computation refers to evaluation of the application of control input to the system. Generally, the states and inputs require a different discretization interval in different dimensions based on the properties and their influence to the system. Adaptive discretization schemes can also be used that vary the discretization intervals of the state and input across stages. The general number of computations in backwards induction can be given by:

$$N_{c,back} = \sum_{i=1}^{N} \left\{ \prod_{j=1}^{I} L_j^i \cdot \prod_{k=1}^{K} P_k^i \right\} \tag{3.12}$$

where

L_j^i: Equally spaced intervals for state dimension j at stage i

P_k^i: Equally spaced intervals for input dimension k at stage i

The backwards induction process occupies most of the percentage of the computations of the Dynamic programming algorithm. Compared to this, the forward run has an exponentially fewer number of computations:

$$N_{c,for} = \sum_{i=1}^{N} \left\{ \prod_{k=1}^{K} P_k^i \right\} \tag{3.13}$$

Thus for the complete algorithm, the total number of computations is:

$$N_c = \sum_{i=1}^{N} \left\{ \prod_{j=1}^{I} L_j^i \cdot \prod_{k=1}^{K} P_k^i \right\} + \sum_{i=1}^{N} \left\{ \prod_{k=1}^{K} P_k^i \right\} \tag{3.14}$$

The cost J is implemented in the form of a matrix of size $L^I \times N$. The matrix must be saved for the computation of an optimal trajectory from an initial state. The implementation of the matrix also introduces a memory requirement for computer, again putting a limit on the size of the system.

3.3.3 Optimization of Vessel 2

The optimization of Vessel 2 is a good demonstration of the application of Dynamic programming to multi-dimensional systems. The multi-dimensional state space implemented for vessels in the Power Management Tool is a simpler class of multi-dimensional systems, which helps avoid a lot of complexities that are associated with typical multi-dimensional state spaces. The states of Vessel 2 are composed of the battery State of Charge (SoC) and the generator *on* or *off* state, and the control inputs are the battery current, brake resistance current and the generator current.

$$\underline{x} = \begin{pmatrix} x_{SoC} \\ x_{gen} \end{pmatrix}, \qquad \underline{u} = \begin{pmatrix} u_{bat} \\ u_{brk} \\ u_{gen} \end{pmatrix} \tag{3.15}$$

The multi-stage decision process for the two-dimensional state space of Vessel 2 is shown in Figure 3.6.

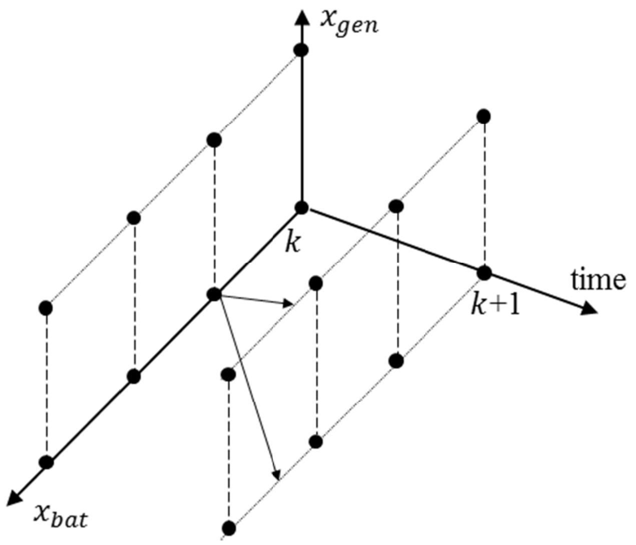


Figure 3.7 Projection of a state point on a two-layered state space

At every stage, the battery SoC state exists on two layers, one with the generator off and one with on. As long as the load requirement is satisfied, any state point can project on any of the two layers. The backwards induction and the forwards run will be executed in the same method as for Vessel 1, but with a two-dimensional cost-to-go matrix.

3.4 Issues with Dynamic Programming

The biggest and the most notorious issue with dynamic programming is the high number of computations involved. Even for simple problems, the computations are so high that it is rarely used as the algorithm for real-time optimal control.

The second issue arises due to the discretization of the state and input space of the system, which is a requirement of dynamic programming. While Dynamic programming itself guarantees an optimal solution, the discretization puts a limit on the quality of the solution. Increasing the discretization, or refining the state and input grid improves the chances of approaching the true optimal solution of the continuous system, but the computations increase in a combinatorial fashion, hence the curse of dimensionality.

The next two chapters study the two issues in depth, and investigate methodologies that aim to reduce the problems associated with discretization and dimensionality.

4 Iterative Dynamic Programming

Iterative Dynamic Programming is a variant of the Dynamic programming algorithm that attempts to optimize a problem through a series of iterations. Based from an initial guess of the optimal policy, each iteration builds its parameters on the solution of the previous iteration. The process is repeated until the solution converges.

4.1 Motivation

The motivation behind the iterative approach to Dynamic programming is the impractically high number of computations. Section 1.3.1 showed the limits that the computational requirement places on the size and the complexity of the problem. The three commonly used classical techniques for reducing the dimensionality of problems have been discussed in [8]. Among the three techniques, this chapter discusses the method of iteratively computing *nominal trajectories.* In addition, parallel computing for Dynamic programming implementation is also considered.

a. Parallel Processing

For the fast and multi-core processors available today, parallel processing is the most obvious techniques to speed up the process of Dynamic programming. Parallel Programming can never be fully applied to the problems formulated in this thesis. This is because the sub-problem for each stage has a dependence on the previously solved sub-problem. For example, during the backwards induction process in Section 1.3.1, no two stages can be optimized in parallel, as the optimization problems at each stage have dependence on the solutions of the future stages.

The computations for one stage can be parallelized. Some programming algorithms for *serial monadic problems* (the sub-problems at each stage depend only on the solutions from the immediately preceding stage) has been suggested in [9]. A generic framework for implementation of parallel dynamic programming has been offered in [10]. The implementation of most published techniques, however, requires significant modification (if not complete rewriting) of the dpm function implemented in MATLAB.

The MATLAB Parallel Computing Toolbox also offers the feature to assign the sub-problems of a task to various workers that compute in parallel. The toolbox functions are also quite easy to integrate with the dpm function. Unfortunately, the generation of workers and assignment of tasks requires a significant computation time on its own. It has been observed even for medium sized problems (Vessel 2) that although the parallel loops reduce the time taken for the computations at each stage, the overhead at the start of the loop causes the total algorithm time to be about the same or even worse than using single-thread loops.

b. Iterative Dynamic Programming

The techniques in Iterative Dynamic Programming (IDP) aim to tackle the problem of high complexity by reducing the search space. Instead of searching in the complete search space, the region with a high likelihood of containing the optimal solution is isolated and the search is constrained within this region. The process is carried out iteratively, and the search region is modified in each iteration based on the solution of the previous iteration.

Wahl has suggested two methods of Iterative Dynamic Programming to optimize the speed strategy for a Hybrid Electric Vehicle [4]. The method investigated and implemented in this thesis is called the iterative reduction of search space, or simply IDP2.

4.2 Iterative Reduction of State Space (IDP2)

As the name suggests, this method follows a similar iterative pattern as policy iteration, but also shrinks the search space progressively. This results in a more refined grid in every iteration. Bellman first proposed the idea in *Applied Dynamic programming* [7]. A vast literature can be found on its application in different fields for the reduction of dimensionality.

Luus first applied the technique successfully to a multidimensional optimal problem in chemical engineering [11]. A similar application for optimization of DAE systems can be found in [12]. The application to Hybrid Electric Vehicle, as well as the implemented methodology in this thesis, can be found in [4] [13]. The algorithm implemented in this thesis carries out an iterative reduction of both the state and input space. The stages of the Iterative Reduction of Search Space algorithm for an N stage process can be formally described with the following steps [4]:

1. An initial state space is defined.

$$\chi_l := \{\underline{x}^l, |x^l_{min}(k) \leq x^l_k \leq x^l_{max}(k)\, ; k = 0, \ldots, N\} \tag{4.4}$$

For the initial stage, x^l_{min} and x^l_{max} represent the physical, or global boundaries of the system. χ_l is a discrete grid within the physical bounds. The number of points L in the grid is set according the desired accuracy and the computation time. The grid at the first step, therefore, is very coarse representation of the complete state space.

2. Dynamic programming is applied to the calculated grid to calculate an optimal policy.

$$J_k(\underline{x}_k) := \min_{\underline{u}_k}\{g(\underline{x}_k, \underline{u}_k, z_k) + J_{k+1}(\underline{x}')\}\Big|_{\underline{x}_k \in \chi^l_k,\, \underline{u}_k \in U^l_k,\, \underline{x}' \in \chi^l_{k+1}} \tag{4.5}$$

3. The search region is reduced.

$$r_{l+1} := \gamma . r_l \tag{4.6}$$

The *search region* r_l corresponds to the size of the state space in iteration l. For one-dimensional space, it is simply the difference between the upper and lower boundaries in the state space. The *reduction factor* γ controls the reduction of state and input space.

4. New boundaries around the optimal state trajectory calculated in step 2 are calculated.

$$\underline{x}_{min}^{l+1} := \underline{x}^{*} - \frac{r_{l+1}}{2} \tag{4.7a}$$

$$\underline{x}_{max}^{l+1} := \underline{x}^{*} + \frac{r_{l+1}}{2} \tag{4.7b}$$

5. The state space for the next iteration is calculated using the boundaries.

$$\chi_{l+1} := \{\underline{x}^{l+1}, |x_{min}^{l+1}(k) \leq x_k^{l+1} \leq x_{max}^{l+1}(k)\,; k = 0, \dots, N\} \tag{4.8}$$

The new state space will be defined around the optimal trajectory calculated earlier. As this state space is calculated for a grid with a smaller area, discretizing it with the same number of points L results in a finer grid than before. The new state space is used for the repetition of step 3. The process is repeated iteratively until the optimal policy converges.

Figure 4.1 shows the advantage of iterative reduction of search space: refinement of grid at each iteration without an increase in the number of computations. The function displayed is the performance function of a system, which displays the optimal solution for all the set of possible control policies. The optimal performance, termed as the performance index is the total cost of the optimal cost trajectory. Due to the iterative refinement of the grid, IDP2 is able to find the global optimum despite the sharp variations in the performance function.

The displayed function, however, has a generally convex nature despite the variations. The sub-optimal performance of IDP2 for a non-convex function is exhibited in Figure 4.2. The convergence highly depends on the initial solution. The likelihood of convergence can also be controlled by the reduction factor. A very high reduction factor leads to a fast convergence, but has a high chance of converging to sub-optimal solutions. In contrast, a very low reduction factor has a higher chance of escaping sub-optimal solutions, but will exhibit a slow rate of convergence.

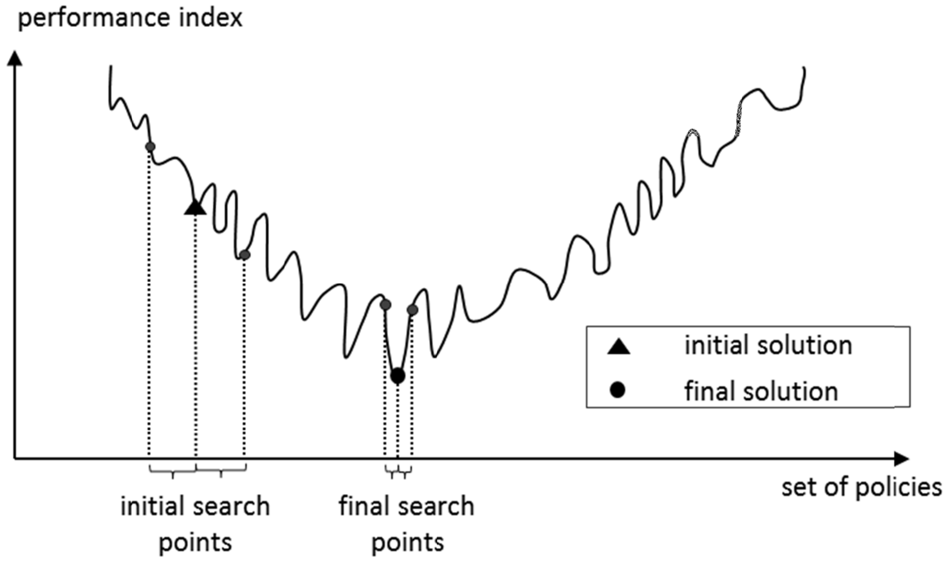


Figure 4.1 Iterative Reduction of search space on a convex performance function (with small but highly frequent variations)

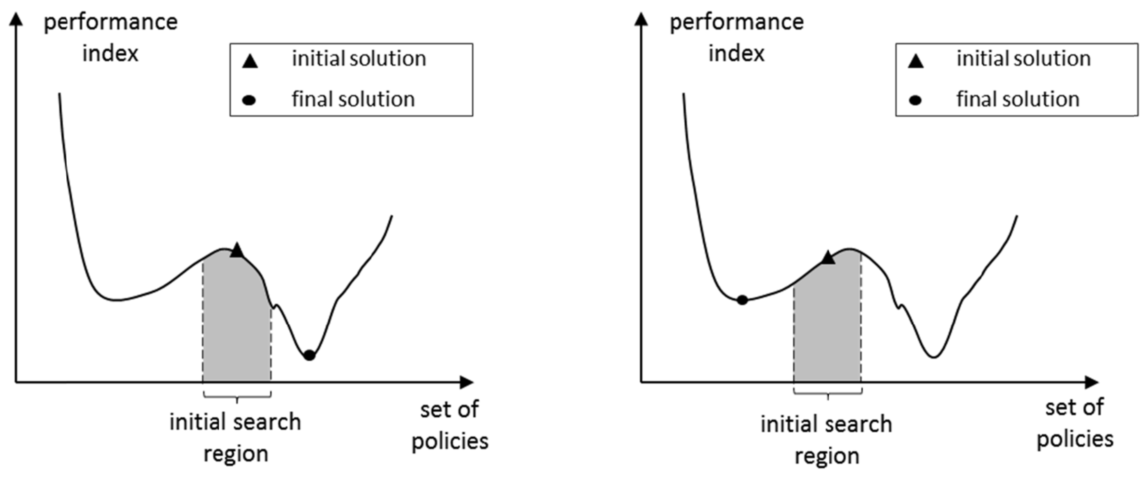


Figure 4.2 (a) Convergence of IDP2 with a good inital policy (b) Convergence of IDP2 with a bad initial policy

Extension to Multi-dimensional Systems

For the Vessels defined in this thesis, the generator states are binary. The state space for these states cannot be reduced or modified, and the only remaining continuous battery SoC state makes the IDP2 implementation equivalent to one-dimensional state systems.

The space reduction is also applied to the input space, which is multi-dimensional for both Vessel 1 and Vessel 2. For numerical implementation of the input space, a discretized input vector, containing the possible values for the input, must be defined for each dimension. The search region r_l is calculated for each vector separately, as are the boundaries.

4.2.1 Implementation Issues of IDP2

There are two major issues that arise in the implementation of IDP2. This section discusses the issues and the proposed solutions.

a. Premature convergence to Local Optima

As the algorithm uses the highly expensive dynamic programming in multiple iterations, the size of the iterations must be reasonably small. This requires a small number of points in the state and input grid. Although the grid can be refined even with a small number of points by reducing its size, the first optimization is always carried out over the complete state and input space. The highly coarse grid in the first iteration causes high possibility of the optimal solution to be far away from the global optimum. In such a case, the next iterations will only search in the neighborhood of the first solution, iteratively reducing the search space. The global optimum will remain unchecked.

The solution implemented for this problem is that at the expense of extra computations, the number of grid points in the first iterations is kept relatively high. In fact, this becomes necessary for the state grid as it is advisable to always observe the *minimum state discretization* (introduced in Section 5.3.1). A higher number of grid points increases the possibility of the first solution in the neighborhood of the global optimum. For the iterations after the first, the number of grid points can be reset to a smaller number according to the desired complexity of each iteration.

b. Progressive deterioration of Optima

The solution of increasing the initial discretization of the problem introduces a new problem. The coarseness of the grid can be expressed as a function of the number of points and the grid size:

$$\Delta_l = \Delta(n, r_l), \tag{4.9}$$

where n is the number of points in the grid and r_l is the size of the initial grid.

Generally, the reduction of the size of the grid at every iteration ensures a less coarseness as the number of points is kept the same. A higher number of points in the initial grid, however, might cause the grid in the second iterations to be coarser despite the smaller size of the grid. It might in fact take several iterations until the grid size is small enough so that the grid coarseness is less than the initial grid. The iterations in between are wasted. To avoid this phenomenon, the initial number of points, the number of points for the remaining iterations and the reduction factor is kept such that:

$$\Delta(L + a_l, r_l) > \Delta(L, r_{l+1}), \qquad a_l = \begin{cases} a, & l = 0 \\ 0, & l \geq 1 \end{cases} \tag{4.10}$$

a corresponds to the extra number of points for the initial grid.

Unfortunately, the problem does not end here. Consider the minima on the performance functions in Figure 4.3, found by two different grids. The two grids have the same number of points, but grid 2 has a smaller size and consequently less coarseness. Despite the less coarseness of grid 2, it returns a poorer result than grid 1. This phenomenon, discussed in detail in Section 5.2.2, is highly undesirable, as any iteration that does not improve the previous solution is just wasted computations.

The IDP2 algorithm inherently offers the solution to this problem in Equations (4.7) by including the solution as a mid-point of the new grid. The calculated grid in is therefore always centered on the solution point (optimal trajectory point from previous iteration). This ensures that the next solution is better or at least the same as the previous one.

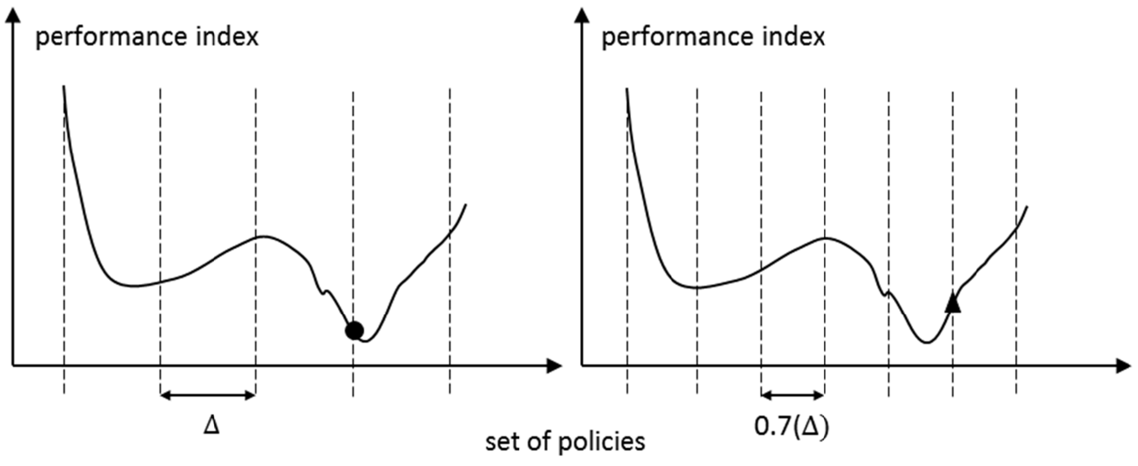


Figure 4.3 Evaluation of the optimal policy with two different discretization schemes. A finer grid does not guarantee a better solution

c. Loss of Optima near Boundaries

Figure 4.4 demonstrates the problem with the mid-point approach. If the points in the vicinity of the boundary are used as the mid-points for the calculation, the new boundaries might exceed the global boundaries of the system.

There are two solutions for this problem:

1. Use a stage dependent reduction factor γ. The stages, for which the optimal state lies near boundaries should have a large reduction factor. This would result in a grid so small that when centered on an optimal point near the global boundary, none of the grid boundaries exceed the global boundary. For an optimal point near the lower boundary, the reduction factor can be

$$\gamma = \frac{\left|x_{min}^{0} - x^{*}\right|}{r_l} \tag{3.11}$$

   x_{max}^{0} can similarly be used for points near the upper boundary. This method has the draw back that the search space is unnecessarily reduced at points near boundaries. Due to the prediction horizon involved in Dynamic programming, this also affects the optimal solutions at other stages.

2. Chip off any grid boundary that exceeds the global boundary. This would result in a grid that is no longer centered around the optimal trajectory near the boundaries. This brings about the problem again that the new solution has may be being worse than in the previous iteration, as the previous solution point is no longer a grid point near the boundaries. The dpm function only allows the definition of a uniform grid (equally spaced grid points). Thus the only way to include the previous solution point as one of the grid point is to increase the number of grid points until one of the points matches the previous solution point. The number of points required to match it with the discretized grid could go very high.

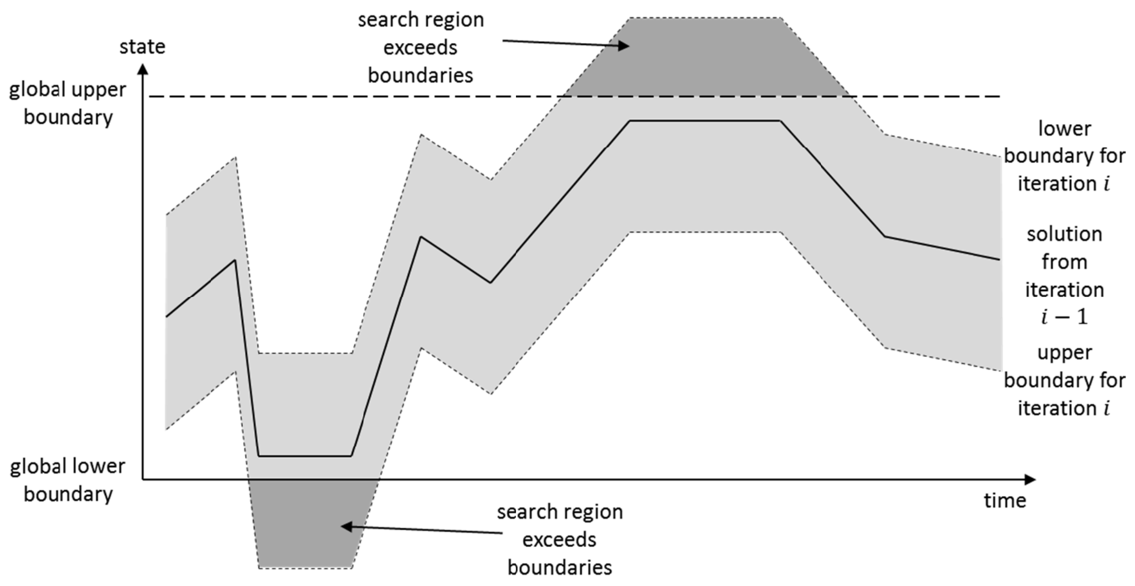

search region exceeds boundaries
state
global upper boundary
lower boundary for iteration i
solution from iteration $i-1$
upper boundary for iteration i
global lower boundary
time
search region exceeds boundaries

Figure 4.4 A search region defined around an optimal trajectory can exceed the physical bounds of the system

3. In addition to chipping off the grid boundary that exceeds the global boundary, adjust the grid until one of the grid points matches the optimal point. This method has been found to be the most useful and has implemented by the name of *grid adjustment algorithm.*

4.2.2 Grid Adjustment

The grid adjustment algorithm is applied to every point in the optimal state and input trajectories that lies close to the boundary. It is summarized in Algorithm 1. The threshold and the maximum number of grid points can be defined by the user. With every point, the information of whether it lies near the upper or the lower boundary is also needed. For the lower boundary, the algorithm will only shift the grid *upwards*, and vice versa. Therefore for an optimal point near the lower boundary, the difference considered is the minimum difference of the point from a grid point that lies below it, and the grid can only be shifted upwards. Similarly, the minimum difference considered for an optimal point near the upper boundary is only from the grid points above it. If the difference is less than the defined threshold, the grid is simply shifted by the difference. Otherwise, new points are added in the grid until the difference is less than the threshold.

Algorithm 1 Grid Adjustment

```
1.  procedure GridAdjust(xSol,grid,mode)
2.      thresh ← acceptable shifting of the grid
3.      n ← number of points in the grid
4.      nMax ← maximum number of points allowed
5.      if mode = up then
6.          diff ← positive difference of xSol from the closest grid point above it
7.      if mode = down then
8.          diff ← negative difference of xSol from the closest grid point below it
9.  loop:
10.     if diff ≤ thresh then
11.         grid ← grid + diff
12.         return grid
13.     if diff > thresh then
14.         n ← n + 2
15.         grid ← generate grid with n points
16.     if n > nMax then
17.         break;
18.     goto loop
```

The grid adjustment does not always guarantee the improvement of solutions. It is possible that despite increasing the number of points to the maximum, the minimum difference between the grid points and the previous solution point is greater than the defined threshold. Based on the range of the state and input space, the threshold and maximum grid points can be defined separately by the user for each state and input to maximize the chances of the convergence of the grid adjustment algorithm.

Optimality of IDP2

As the first iteration searches the complete search space, IDP2, similar to regular dynamic programming can guarantee the global optimum of the problem under the initial discretization. Further iterations have a high chance of improving the solution due to the grid adjustment algorithm. To be shown in Section 6.2, the final solution of IDP2 far exceeds the initial solution for the defined test cases.

5 Errors in Dynamic Programming

Section 3.3.1 discusses the curse of dimensionality associated with the application of dynamic programming due to discretization. The errors associated with discrete dynamic programming that cause the solution to diverge from the global optimum were raised by Bellman himself [7], and are generally considered in most applications that use dynamic programming. The intuitive deduction is that the optimal solution for the discretized space will approach the true solution, i.e., the solution with a continuous space, as the number of points in the discretization grid increases. This convergence of discretization procedures has also theoretically been proven in [14] and [15], under the assumptions of a smooth cost function. Even though no analytical models are included for the systems discussed in this thesis, the assumptions of smoothness can be extended to the cost functions and thus the convergence of discretization can be assumed. This chapter discusses the errors of both the input and state discretization in context of Marine Drive Systems and the implemented dpm function.

5.1 Sources of Error

The backwards induction method for a discretized state and input space was formulated as such in Section 3.3.1.

$$J_k(\underline{x}_k) = \min_{\underline{u}_k}\{g(\underline{x}_{N-1}^{i}, \underline{u}_k, z_k) + J_{k+1}(\underline{x}')\} \tag{5.1a}$$

$$J_k(\underline{x}_k) = \min_{\underline{u}_k}\{\theta(\underline{x}_{N-1}^{i}, \underline{u}_k, z_k, \underline{x}')\} \tag{5.1b}$$

A new term, the total cost θ is introduced in this chapter, which helps in the study of numerical errors. Error at stage k is the difference of the solution of Equation (5.1) from the true solution. There are two sources of error in Equation (5.1):

1. **Error in Greedy Cost**: The cost function g, also called the *greedy cost* in context of optimization problems should be well represented by the state-input space. The

discretization of the continuous state and control input variables $\underline{x}$ and $\underline{u}$ limits the search on the functions to a set of finite points. The higher the discretization, the higher is the probability of containing the globally optimum points as one of the search points.

2. **Error in Cost-to-Go**: The error of representation also exists for the cost-to-go J_{k+1}. In addition, this function exhibits an error of interpolation. As the cost-to-go J_{k+1} is saved only for the discrete state space $\underline{x}_{k+1} \in X_{k+1}$, the cost-to-go for the continous variable $\underline{x}'$ is calculated using linear interpolation. This introduces an error for a non-linear cost-to-go function.

All of the above sources of errors cause the total cost θ to be misrepresented in comparison to the true total cost, which results in the estimation of false minima.

5.2 Error in Greedy Cost

For an N stage problem, Equation (5.1) for the second last stage $N-1$ is reduced to:

$$J_{N-1}(\underline{x}_{N-1}^{i}) = \min_{\underline{u}_k}\{g(\underline{x}_{N-1}^{i}, \underline{u}_{N-1}, z_{N-1})\}\,|\underline{x}_{N-1}^{i} \in X_{N-1}, \underline{u}_{N-1} \in U_{N-1} \qquad (5.2)$$

There is no cost-to-go defined for stage N, therefore only the greedy cost needs to be optimized. The dpm function optimizes Equation (5.2) by running an exhaustive search on the cost of all the feasible control inputs $\underline{u}_{N-1}$ on the state point $\underline{x}_{N-1}^{i}$. The exhaustive search takes the form of a projection on the fuel consumption function. In Figure 5.1, the projected search region consists of the three marked points on the fuel consumption curve.

5.2.1 Greedy Cost Dependence on Experiment Variables

The fuel consumption is not a function of the control inputs, rather the current that the generator must supply to satisfy the load requirement. However the load requirement, states and inputs related to non-generator components influence the generator demand and consequently the greedy cost.

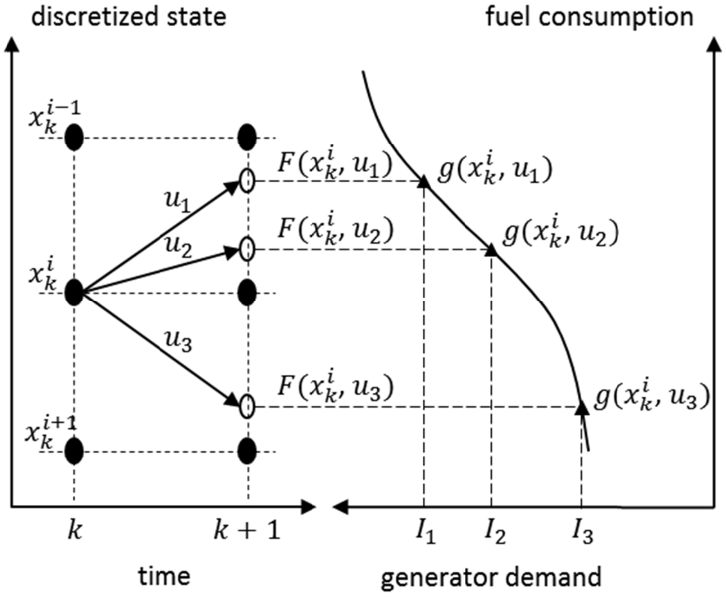


Figure 5.1 Greedy Cost representation by the input grid

The DC grid equation for a vessel containing one generator state is:

$$I_{g0} + I_{g1} + I_{bat} + I_{brk} + I_{load} = 0 \tag{5.3a}$$

The demand on the generators can be defined as:

$$I_{bat} + I_{brk} + I_{load} = I_{demand} \tag{5.3b}$$

$$\Rightarrow I_{demand} = -(I_{g0} + I_{g1}) \tag{5.3c}$$

Any parameters of the system that influence I_{demand} will have an influence on the fuel consumption. The dependence of fuel consumption on the state, inputs and load cycle is discussed in the following sections.

a. Dependence on Inputs

For a system with one generator state (Vessel 2), the inputs have the following influence on the greedy cost:

- The input u_{brk} for the brake resistance is generally the same as I_{brk}. Therefore $I_{demand} \propto u_{brk}$.
- The relation between u_{bat} and I_{bat} depends on the efficiency of the battery converter. The cases considered in this study assume the efficiency to be constant. Therefore u_{bat}=αI_{bat}, where α is the constant efficiency. $I_{demand} \propto u_{bat}$.
- The inputs for the generator u_{g1} also influences the grid current I_{g1} based on the converter efficiency of the generator. This is also assumed to be constant for all cases in this study. Again u_{g1}=βI_{g1}, where β is the constant efficiency. Therefore $I_{demand} \propto u_{g1}$.

The three discretized inputs form an input grid, which is the set of all the possible input combinations that are feasible for the state point. Each node in the grid is associated with a greedy cost, and the set of all greedy costs form the search region on the fuel consumption function. Input grid discretization determines the resolution of the search region on the fuel consumption function. A finer discretization of the input grid corresponds to a higher resolution of the search region, which increases the probability of finding the optimal point with an exhaustive search.

b. Dependence on States and Load Cycle

- In a real battery, the battery state, the state of charge (SoC) influences the efficiency of the battery. This changes the relationship of the DC current I_{bat} to the applied input u_{bat}. The battery state SoC will therefore influence I_{demand}, which limits the choice of control inputs that satisfy the load requirement.
- For the optimization experiments in this study, the battery model implemented is simplified and a constant efficiency is assumed. For most battery SoC points, the feasible control inputs remain the same. As the control inputs correspond to the greedy cost, each state point has the same search region on the fuel consumption function, as shown in Figure 5.2. At boundaries however, the available control inputs are limited, which shrinks the search region. For example, battery SoC at 0% cannot discharge any further, and any negative control inputs are infeasible.

- The generator states are binary states, indicating the switching action of a generator. Switching a generator on incurs a penalty in the fuel consumption.

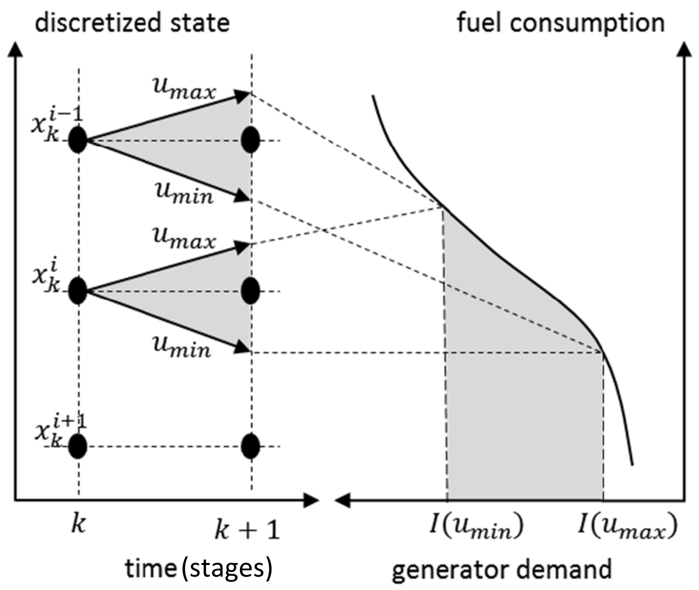


Figure 5.2 Projection on the greedy cost from different state points within a stage

- The load cycle contributes directly to I_{demand} , but only varies from one stage to the next. For every stage, the load cycle can be considered as on offset to the search region.

5.2.2 Representation error of Greedy Cost

The difference between the optimal point in the discretized search region on the greedy cost and the continuous search region on the greedy cost is the *representation error.* Generally, a finer input discretization leads to a search region with high resolution, which increases the probability of reducing the representation error. A better solution, however, is not guaranteed. Consider the cost consumption functions in Figure 5.3. The three figures show the function represented for three different discretizations. The first figure shows the minimum obtained by a reference discretization Δu (shown by a circle). The second has a discretization better than Δu, but is reduced by a non-integer factor. The original grid points are displaced, and the calculated minimum (shown by a triangle) is actually worse. The third figure reduces Δu by an integer factor of 2. The original points will remain in the new grid, which will result in a minima better or at least equal to the one obtained with the original grid (shown by square).

Improving the discretization by a non-integer factor therefore only increases the chances of obtaining a better solution. The fact is crucial in the study of discretization convergence.

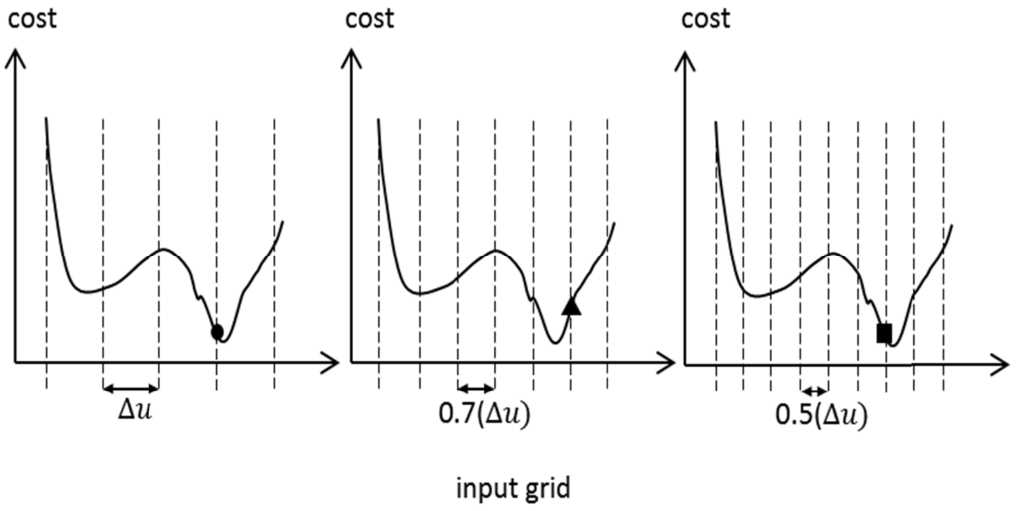


Figure 5.3 Representation errors from different discretization schemes

The greedy cost function can be formulated to include the represntation error as:

$$g(\underline{x}_{N-1}^{i}, \underline{u}_{N-1}, z_{N-1}) = g_{min} + e_{rep}^{g}(\Delta \underline{u}_{N-1}) \tag{5.4}$$

g_{min} refers to the global solution on the search space projected by $\underline{x}_{N-1}^{i}$. e_{rep}^{g} refers to the representation error, which only depends on the input discretization.

For the simplified battery model implemented throughout this study, every state point $\underline{x}_{k+1}^{i}$ will project on the same search region. Therefore, the state discretization is not relevant for improving the representation error.

Propagation of Error

For an N stage problem, Equation (5.1) for the stage $N-2$ is:

$$J_{N-2}\left(\underline{x}_{N-2}^{i}\right) = \min_{\underline{u}_k}\left\{g\left(\underline{x}_{N-2}^{i}, \underline{u}_{N-2}, z_{N-2}\right) + J_{N-1}\left(\underline{x}'\right)\right\} \tag{5.5a}$$

$$J_{N-2}\left(\underline{x}_{N-2}^{i}\right) = \min_{\underline{u}_k}\left\{\theta\left(\underline{x}_{N-2}, \underline{u}_{\underline{x}_{N-2}}, z_{\underline{x}_{N-2}}, \underline{x}'\right)\right\} \tag{5.5b}$$

The cost-to-go J_{N-1} will be computed with a representation error:

$$J_{N-1}\left(\underline{x}_{N-1}^{i}\right) = g_{min} + e_{rep}^{g}\left(\Delta\underline{u}_{N-1}\right) \tag{5.6}$$

Thus the representation error propagates the next stage. Unlike stage $N-1$, the objective at stage $N-2$ is not to find the optimum on the greedy cost g, but the total cost θ. In addition to the representation of the greedy cost, the cost-to-go should also be well represented.

5.3 Error in Cost-to-go

5.3.1 Representation error of Cost-to-go

As shown in Figure 5.4, each state point $\underline{x}_{N-2}^{i}$ also projects a search region on the cost-to-go function. In this case, the resultant state from a control input, $\underline{x}' = F(\underline{x}_k, \underline{u}_k)$, rather than the control input itself, corresponds to a point on the cost-to-go function. Therefore unlike the greedy cost, each state point $\underline{x}_k^{i}$ has a different projection on the cost-to-go function for the same set of control inputs.

The principle for the control input discretization works the same way as the greedy cost. A fine input grid results in a projection of a high resolution search region on the cost-to-go function, which increases the probability of finding the true optimum.

The objective function for minimization in Equation 5.5 is not just the greedy cost or the cost-to-go, but the sum of both. Changing the state discretization will result in the same set of greedy cost search regions, but different cost-to-go search regions. Therefore, the state discretization also has an effect in the representation error on the total cost function. The representation error can therefore be formulated for stage $N-2$ as:

$$\theta(\underline{x}^i_{N-2}, \underline{u}_{N-2}, \underline{z}_{N-2}, \underline{x}') = \theta_{min} + e^{\theta}_{rep}(\Delta\underline{x}_{N-2}, \Delta\underline{u}_{N-2}) + e^{g}_{rep}(\Delta\underline{u}_{N-1}) \tag{5.8}$$

θ_{min} refers to the true optimum on the total cost function, without any errors of discretization. e^{θ}_{rep} is the representation error on θ , the difference of the minima of the discrete search space on total cost from the global minima of the continuous search space. e^{θ}_{rep} depends on the misrepresentation of both the greedy cost and the cost-to-go function, and therefore depends on both the input and state discretization. The representation error from the previous step e^{g}_{rep} is also propagated to the solution.

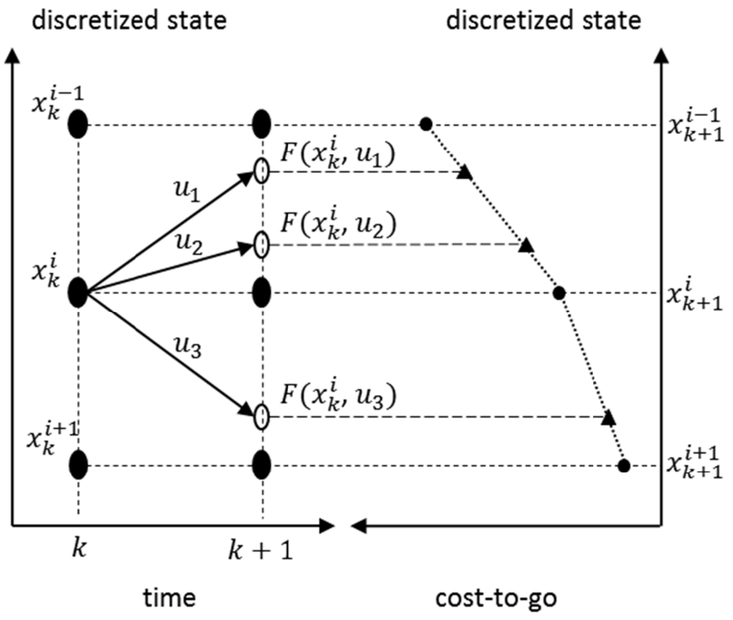


Figure 5.4 Cost-to-go representation by the input grid

The discussion of state discretization also brings about a special case of discretization errors discussed in the next section, which can be avoided by setting a *minimum state discretization.*

Minimum State Discretization

The representation of the cost-to-go for the exhaustive searches of two consecutive state points is shown in Figure 5.5. With the state discretization in the figure, a region of the cost-to-go is left un-searched, even with infinite input discretization.

This can cause major errors in optimization, as the minimum on the total cost θ might lie in the unsearched area. This phenomenon should be avoided to avoid loss of information, and a *minimum state discretization* should be observed at every stage.

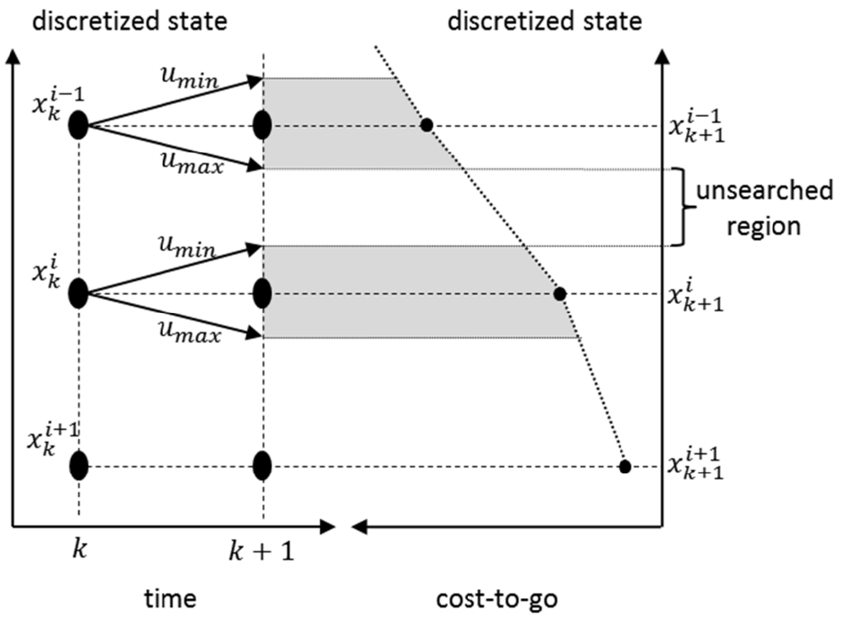


Figure 5.5 Representation of cost-to-go with a poorly discretized state grid

Thus for any stage k, the state should be discretized such that

$$F(x_k^i, u_{max}, w) \geq x_k^{i+1} \tag{5.7a}$$

$$F(x_k^i, u_{min}, w) \leq x_k^{i-1} \tag{5.7b}$$

The equation above is only relevant for the battery SoC as it is the only discretizable state. The state and input parameters must be set such that a discretized SoC point can be charged or discharged to the adjacent points. This allows the search of the complete cost-to-go function. The resolution of the search region can then be improved by increasing the control input discretization.

Satisfying the minimum state discretization does not eliminate the dependency of representation error on state discretization.

5.3.2 Interpolation error in Cost-to-go

In addition to representation errors, the cost-to-go J_{N-1} will also introduce errors of interpolation in Equation 5.5. This error is demonstrated in Figure 5.6.

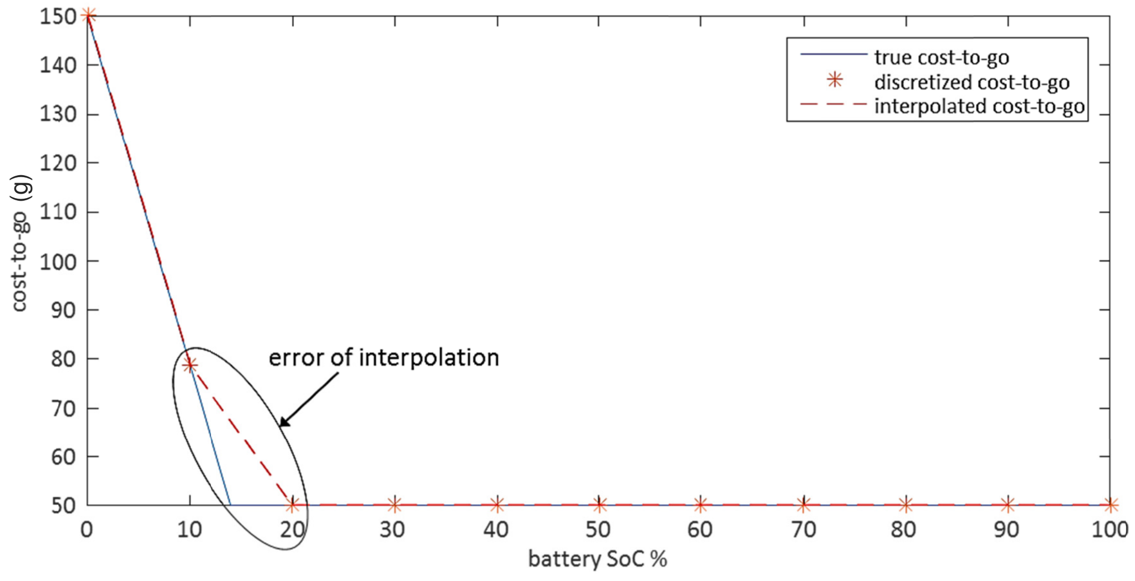


Figure 5.6 Interpolation error in cost-to-go

The figure shows the calculated cost-to-go at stage $N - 1$ for two discretization schemes, using a generator with linear fuel consumption characteristics. The first discretization scheme is assumed to be infinite, and the cost-to-go calculated represents the true function (continuous line).The second discretization scheme sets the state interval at 10%, but the cost-to-go calculated at the state points is without any representation error (asterisks). The cost-to-go function calculated for the remaining state space, using linear interpolation from discretized points is also shown (dotted line). Despite the absence of representation errors, the error of interpolation can be observed at the area of non-linearity.

The true cost-to-go calculatedshows a non-linearity at SoC $= 15\%$ because as SoC goes below 15%, the available discharge of the battery is limited, which increases the demand on the generator. Even though the cost-to-go is computed without error for the discretized state space at stage $N-1$, the discretization will introduce errors of interpolation during the minimization at the next stage, i.e., stage $N-2$.

The error for stage N-2 in equation 5.8 can be updated to include the expected error of interpolation as well.

$$\theta = \theta_{min} + e_{rep}^{\theta}\left(\Delta\underline{x}_{N-2}, \Delta\underline{u}_{N-2}\right) + e_{interp}^{J}\left(\Delta\underline{x}_{N-1}\right) + e_{rep}^{g}(\Delta\underline{u}_{N-1}) \qquad (5.9)$$

5.3.3 Additional Errors

a. Generator Switching Penalty

With the inclusion of the generator state, the cost-to-go becomes a two layered function similar to the state grid. The generator penalty can introduce a new error of interpolation, shown by Figure 5.7. The cost-to-go in Figure 5.7 shows the cost-to-go for the generator 'off' state at stage $N-1$, calculated for a relatively large battery and a high load cycle. At 35% SoC, the battery can no longer fully discharge, which increases the demand on one generator, resulting in the gradient at 35%. The generator used will be the balance generator, as turning generator 1 on will incur a penalty in addition to the generator fuel consumption function. At 15% however, the battery discharge is limited to less than half its complete range, and balance generator is no longer enough to fulfill the load requirement. At this point, generator 1 must be turned on, the penalty of which causes the plateau and the higher slope at 15%. The figure highlights the additional errors of interpolation around the resultant points of non-linearity. Thus the inclusion of the generator penalty can also result in higher errors of interpolation.

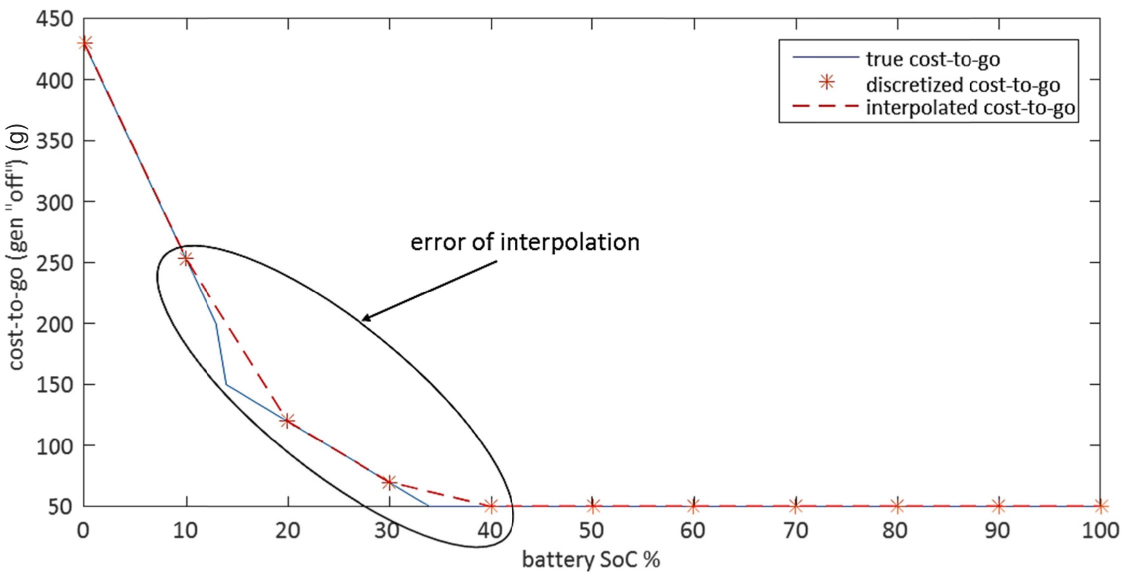


Figure 5.7 Interpolation error in cost-to-go with generator penalty

b. Dependence of Greedy Cost on SoC

In real systems, the battery efficiency is not constant, but is a variable dependent on the SoC. Thus, the relation of the battery control input to the DC current supplied by the battery changes with the SoC. Therefore the assumption of a greedy cost independent of SoC, shown in Figure 5.2, no longer holds. Changing state discretization will result in a different set of both the greedy cost and the cost-to-go functions, and the non-linearity of the total cost will increase even more, further increasing the error of interpolation.

5.3.4 Summarizing the Errors

The cases in Figures 5.6 - 5.7 are shown for generators with linear fuel consumption characteristics. As was seen in Section 2.3, both the test cases studied in the thesis use generators with non-linear characteristics. The addition of the non-linear cost-to-go function results in a highly non-linear total cost, due to which the error of interpolation does not occur at one point, rather throughout the cost-to-go function.

Thus the error in for the sub-problem at any stage, formulated by Equation (5.1) has the following components:

$$J_k(\underline{x}_k^i) = J_{k,true}(\underline{x}_k^i) + e_{rep}^{\theta}(\Delta\underline{x}_k, \Delta\underline{u}_k) + e_{interp}^{J}(\Delta\underline{x}_{k+1}) + e_{prop} \tag{5.10}$$

$J_{k,true}(\underline{x}_k^i)$ represents the true optimal cost-to-go for the sub-problem i at stage k. The true optimal cost-to-go is the cost-to-go without any discretization errors. The cost-to-go calculated for the discretized state and input grids has a difference from the true optimum, which is due to the representation error of the total cost function, dependent on both the input and state discretization for stage k, the interpolation error, dependent on the state discretization at stage $k + 1$, and an error propagated from the previously solved stages.

5.4 Offline estimation of Representation error

This section proposes methodologies to predict the representation errors at different stages before running the optimization procedure, or the offline estimation of errors.

5.4.1 Error Prediction based on Empirical Data

The greedy cost term is predictable as it is always a subset of the fuel consumption function. For all the set of greedy costs calculated during DP optimization,

$$\bigcup_{i=1}^{N-1} g_i \subset C \tag{5.14}$$

g_i is the greedy cost for a stage, while C represents the set of fuel consumption data. It was discussed in Section 5.2.1 that due to the simplifications of the battery in this thesis, the greedy cost only changes through stages as the load requirement changes. The fuel consumption data is included as part of the model before running the optimization. This section proposes a method to use empirical observations from a black box Drive Train model to estimate the quality of the fuel consumption function representation. The quality is represented by an error factor.

a. Calculation of Error Factor

The ideal discretization of a grid that represents a function would have no difference between the discrete optimum and the global optimum. For the ideal discretization, the function is linear between any two adjacent points, and the linear interpolation between any adjacent points will represent the true function. The error factor is the measured deviation of the linearly interpolated value from the true function value, and thus corresponds to the quality of the discretization. The error factor calculation is demonstrated graphically in Figure 5.9 for a one-dimensional grid. Between every two points of the original grid, new points are added on which the function is evaluated. The evaluation is compared with the linear interpolation between the original points to calculate a measure of the representation error, or the error factor. For a two dimensional grid, intermediate points will be inserted between a square of original points, and for a three dimensional grid, a cube. As the n- dimensional grid is generated from n vectors, the addition of intermediate points between every two elements of the vector allows the generation of extra points for any n- dimensional grid.

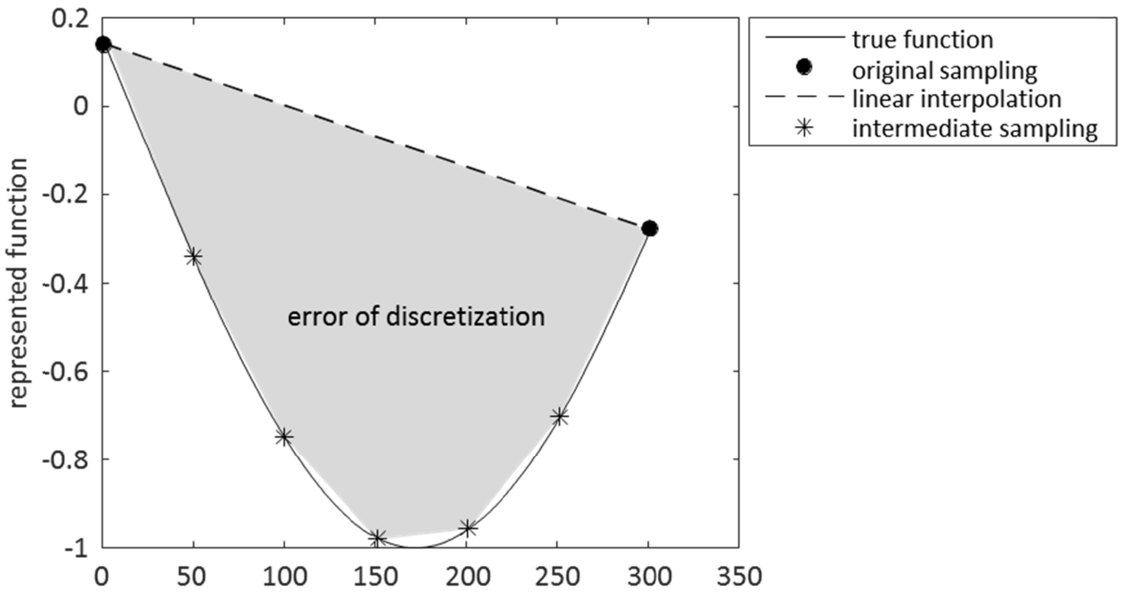


Figure 5.8 Error Factor calculation for one-dimensional grid

Using the methodology shown in the figure above, the error factor can be calculated to include the representation errors for all the adjacent points in a grid as follows:

$$E = \frac{\sum_{i=1}^{n} \left| w_i . \left(L\left(\underline{u}'_i\right) - f\left(\underline{u}'_i\right)\right) \right|}{\sum_{i=1}^{n} w_i} \tag{5.15}$$

u' defines a grid of size n that contains intermediate points between the points in the original grid u. L denotes linear interpolation, and $L\left(\underline{u}'_i\right)$ represents the linearly interpolated values calculated using the original grid evaluations $f(\underline{u}_i)$. Any deviation between two values that is higher than the user defined threshold is penalized extra using the weights. In addition, the existence of both negative and positive errors indicates a point of inflexion between adjacent points. This is penalized extra by setting the weights high. The points in the vicinity of infeasible points are ignored.

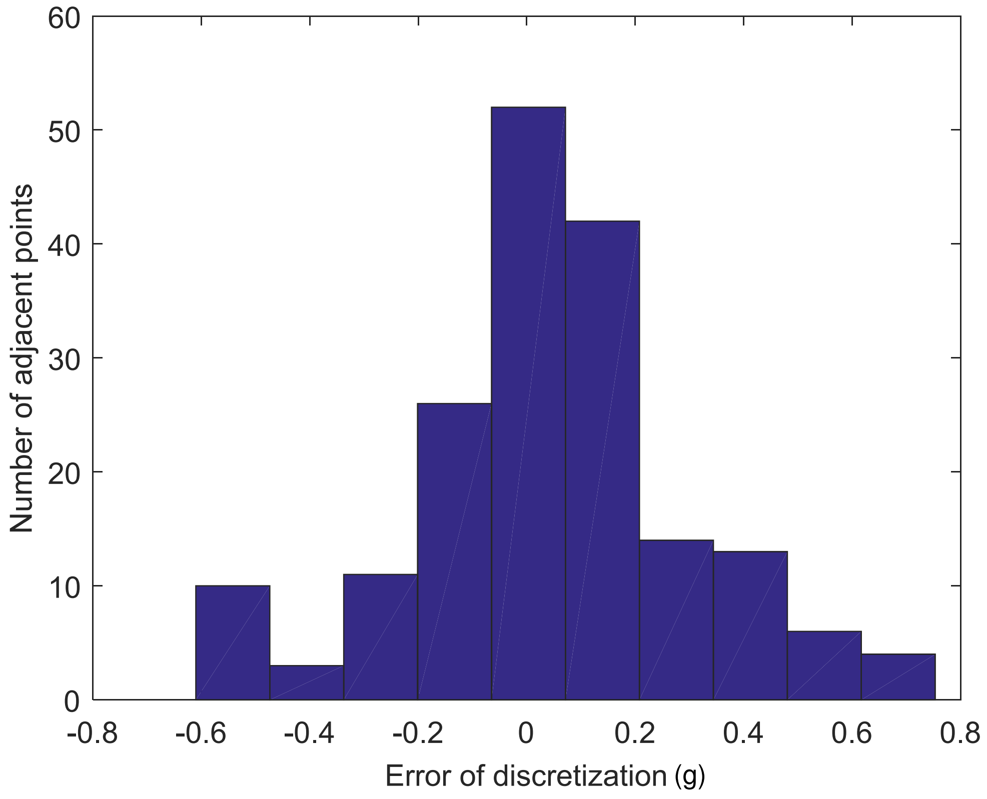


Figure 5.9 Offline computed errors of discretization for adjacent points in the input grid

The represented function corresponds to the greedy cost search region on the fuel consumption function. As the control input grid has been found to represent the greedy cost, the error factor is only calculated for the input grid. The error factor, therefore, is a measure of the representation error of the greedy cost.

Figure 5.10 shows the error distribution between all adjacent input points for a given sub-problem i at stage k, calculated from real optimization interval data. The symmetric distribution of the errors around zero reflects the symmetry of the cost consumption curve, indicating an equal distribution of local minima and maxima. As input discretization improves, the errors cluster closer to zero. An infinite input discretization will return an error distribution that is a straight line on zero, i.e., the error of discretization for all the adjacent points will be zero.

The second factor influencing the search region on the cost consumption curve is the load requirement. As the load requirement changes along the stages of the optimization process, the error factor is calculated for various stages and averaged.

b. Adaptive Input Grid

Based on the offline error factor calculation, an adaptive input grid can also be calculated. The adaptive input grid calculation is summarized in Algorithm 2.

The motivation behind the adaptive input is that the load cycle changes with the stage. As the different load requirements in the load cycle may have search region on the fuel consumption function, different load requirements may search on regions with different degrees of linearity.

The algorithm calculates a discretization for multiple load requirements in the load cycle, such that the error factor corresponding to each load requirement does not surpass a defined threshold. The error factor threshold is defined by the user. Calculating the error factor for the complete load cycle can be computationally infeasible. Therefore, error factor is calculated for just a few samples of load requirement, spaced evenly between the minimum and the maximum load requirement throughout the multi-stage process. For each sample of the load requirement, sufficient input discretization is calculated that brings the error factor within the defined threshold.

Algorithm 2 Calculation of Adaptive Input

```
1.  procedure CALCULATE ADAPTIVE(grid)
2.      thresh ← desired error factor
3.      loadReq ← samples from the load cycle
4.      n ← 1
5.  loopU:
6.  loopL:
7.      e ← errorFactor (loadReq(n) , grid(n))
8.      if $e >$ thresh then
9.          grid(n) ← increase grid discretization
10.         goto loopL
11.     $n \leftarrow n + 1$
12.     if $n <$ length(loadReq) then
13.     goto loopU
14.     if $n =$ length(loadReq) then
15.         grid ← discretization for remaining stages using linear interpolation
```

For the remaining load cycle, the required input discretization is linearly interpolated, under the assumption of a smooth fuel efficiency curve. This results in an input grid that is adapted to the load cycle set for optimization.

For stages with load requirement that search on a region of high non-linearity, the adaptive grid will have a finely discretized grid and for stages with load requirement that search on a region of relatively low non-linearity, the adaptive grid will have a coarsely discretized grid. An example of the adaptive grid calculated for a load cycle is shown in Figure 5.10, which shows the total number of points in the input grid at each stage, calculated based on the load cycle at the stage and the region it corresponds to on the fuel consumption curve.

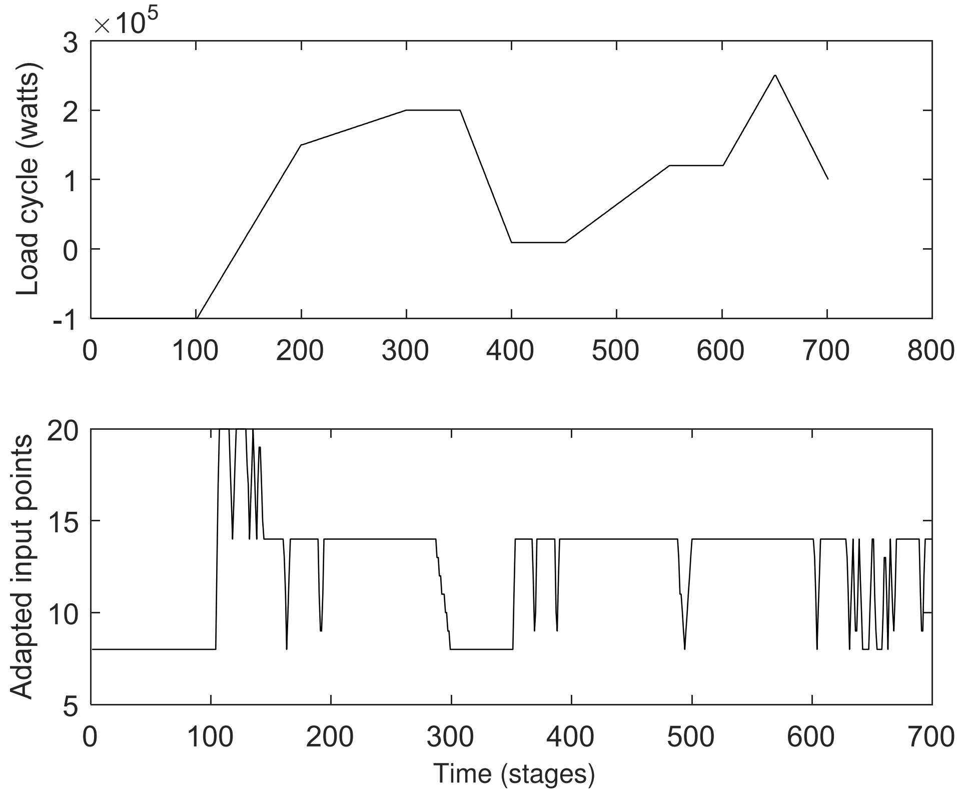


Figure 5.10 Number of grid points adapted for the load cycle

5.4.2 Nyquist based approach

This approach aims to calculate maximum bounds on the discretization errors before starting the optimization procedure. It requires the analytical modeling of the cost consumption of the generators. As this requires additional effort and it must be done every time a new model of the generators is implemented, the approach proposed is not implemented and only has been mentioned to suggest further study into the idea.

Figure 5.11 shows a hypothetical total cost function θ, calculated for a state point and represented by a given input grid. For any input grid, it can be inferred that

$$\underline{u}_{min,grid} - \underline{u}_{min,true} \leq \Delta\underline{u} \qquad (5.11)$$

$\underline{u}_{min,grid}$ represents the grid point that returns the minimum cost, while $\underline{u}_{min,true}$ is a hypothetical point that would return the true minimum cost. This means that the true minimum on the function must lie between any of the two grid points.

The Nyquist–Shannon theorem for sampling signals states that the sampling frequency must be at least twice the maximum frequency of the continuous signal to preserve complete information about the continuous signal [16]:

$$f_s \geq 2f_{max} \tag{5.12}$$

The input discretization $\Delta\underline{u}$ can be thought of as the sampling frequency of the signal. Figure 5.11 shows the signal sampled at $2f_{max}$, and the worst result that can be achieved with this discretization.

Sampling at this discretization allows a more feasible upper bound on the difference between the grid point samples the minimum cost and the point of the true minimum cost.

$$\underline{u}_{min,grid} - \underline{u}_{min,true} \leq \Delta\underline{u}/2 \tag{5.13}$$

This means that the true minimum lies in an area around the grid point that is 50% of the distance between subsequent grid points. Sampling at frequencies higher than two will similarly allow an upper bound that is even closer to the grid points. Although an analytical model can be developed for the greedy cost function, the cost-to-go is an implicit DP function and cannot be predicted according to the best knowledge of the author. However, the state discretization can be used to calculate an upper bound on the maximum frequency possible for the cost-to-go functions throughout the multi-stage optimization. Based on the maximum frequencies calculated, this approach can be used to provide probabilistic error bounds for representation errors in addition to the optimal solution.

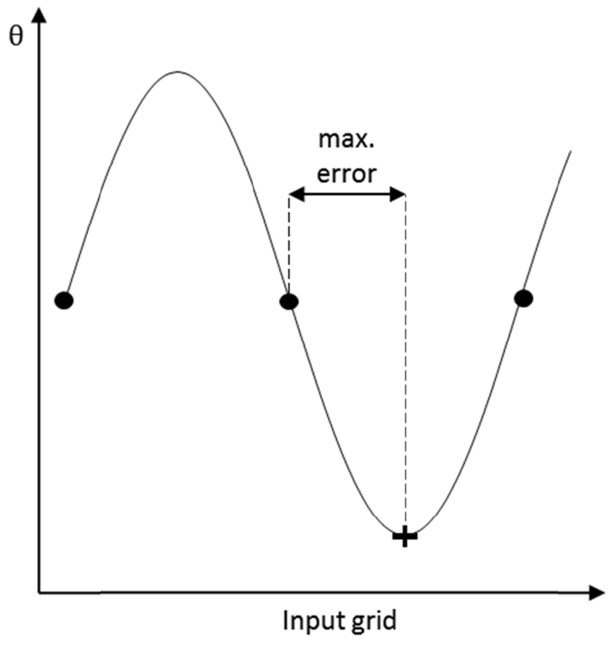


Figure 5.11 Hypothetical total cost function represented with grid interval twice the maximum frequency

5.5 Gradients near boundary points

5.5.1 Level set Method

The dpm function provides the option of using the level set algorithm to ensure that the optimal trajectories of states end up within the user-defined end constraints. The level t-set algorithm defines a *backward-reachable* space for each stage [17]. The backward-reachable space at a stage spans the state points in a state space, on which control inputs can be applied such that the states end up within the end constraints at the last stage. For a multi-dimensional state space, the backward reachable space is demonstrated for end-constraints is Figure 5.12. For a given state point at stage k, a control input causes a transition into the next state by the function:

$$x' = f_k(x_k, u_k)$$

Level set algorithm considers only those control inputs feasible that cause x' to be in the backward-reachable space.

The algorithm is illustrated in Figure 5.13. For a point x^p at stage k, point-set algorithm identifies the valid control input candidates based on the values of the point-set function at the resultant points $\zeta(f_k(x^p, u))$. Only the positive point–set values, indicating a transition into the backward reachable state is considered (black dots vs. empty diamonds). There can also be points, such as x^q, for which no control input transitions into the backward reachable space. As the figure shows, the cost-to-go at these points is based on an alternative control scheme $\tilde{u}_k(x^q)$. The optimal control input in this case drives the state as close as possible to the backward reachable space (black triangle). This approach gives the advantage that the cost at points outside the backwards reachable space does not need to be set as infinite, which avoids high gradients.

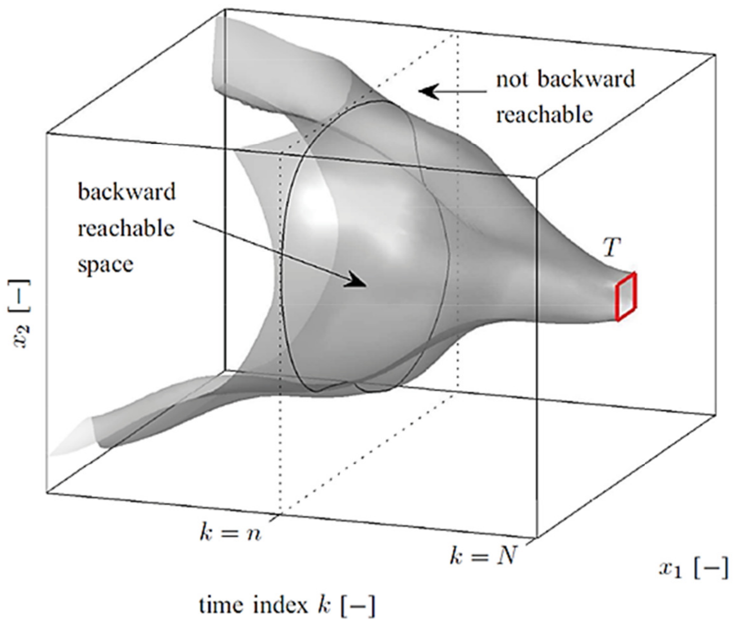


Figure 5.12 Backwards reachable space in a two-dimensional state space due to end-constraints [17]

a. Effect of low State Discretization

A state discretization less than the minimum state discretization is not compatible with the level set algorithm. The minimum state discretization (Section 5.3.1) for the battery SoC is the state discretization such that any point in the discretized state space can be charged as well as discharged to at least its closest points in the discretized state space, by the application of control inputs that exist within the user-defined range. If the level set method is used for state discretization less than the minimum state discretization, the backwards reachable space of the end constraints will only include the state points that are already within the end-constraints, and the feasible state space will not diverge as shown in Figure 5.12. Using the level set method without following the minimum state discretization can therefore lead to highly sub-optimal solutions, as the feasible state space is limited to the end-constraints.

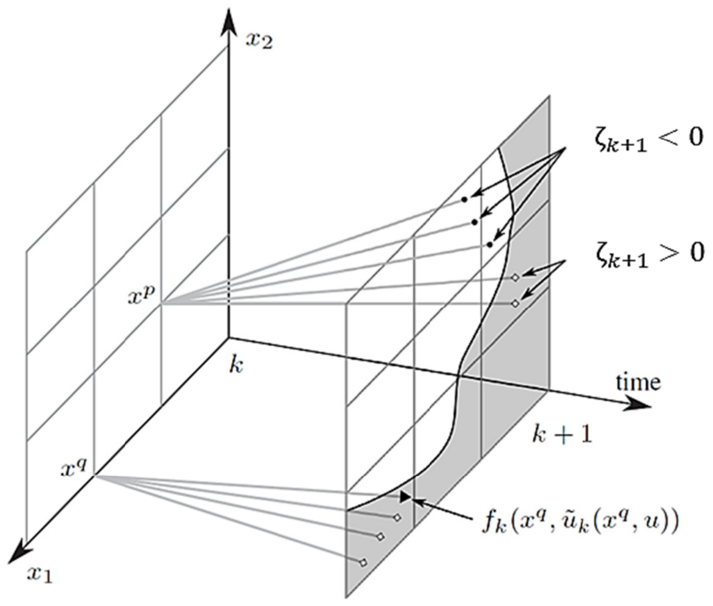


Figure 5.13 Calculation of the backward reachable space [17]

b. Infeasibilities for Load Requirements

The load requirement can also introduce infeasible state points within backwards reachable space. The introduction of infeasible state points due to high load requirements was introduced in Section 1.3.3. It is demonstrated on a two-layered state space for a system with one generator state.in Figure 5.14.

The load cycle is high at stage $k+1$, such that it cannot be satisfied if the battery SoC is at 0% (the battery cannot be discharged). The infeasible cost-to-go is saved at 0% SoC regardless of the generator state, as turning on the generator and running it on full capacity is not enough to satisfy the load requirement if the battery is empty. The load requirement at stage k is similarly high, and an infeasible cost-to-go is saved for 0% SoC. The next SoC point must project on the generator 'on' layer at stage $k+1$, as the generator must be turned on to satisfy the load requirement. However, the battery must also discharge. The cost-to-go for this transition is obtained by the linear interpolation between a feasible and an infeasible cost-to-go, which results in a very high cost. A similar infeasible cost-to-go is also possible if the load requirement is too negative, i.e., the actuators provide so much current to the DC grid that the battery must discharge to keep the net zero.

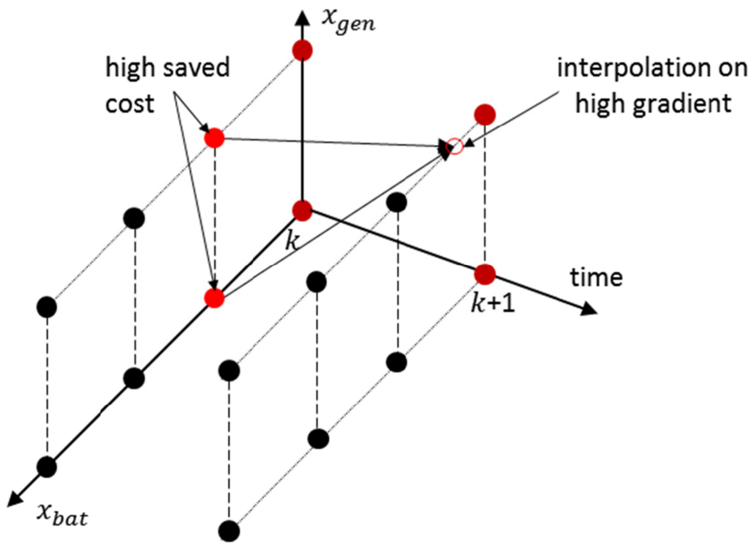


Figure 5.14 Propagation of infeasible points due to high load requirements

The high cost-to-go can propagate to the complete state space, if the load requirement is persistently high. The lesser the state discretization, the faster it propagates to other states, which can result in high erroneous optimal solution.

5.5.2 Boundary Line Method

The dpm function also offers a boundary line method to cater for end-constraints [18]. The boundary line method calculates the boundary before running Dynamic programming, and comes up with a much more precise state space boundary between the feasible and the infeasible regions. It is only implemented for one-dimensional states.

The Boundary Line method defines a function such that

$$f_k(x_k, u_k) = F_k(x_k, u_k) - x_k \tag{5.14}$$

As the method is only implemented for one dimension, there only exists the upper and the lower boundary. The lower boundary is initialized by the straight constraint $x_{k,low} = x_{f,min}$. At any stage k during the calculation of the lower boundary, the following problem is solved to find the boundary point:

$$\begin{gathered} \max_{u_k} f_k(x_{k,low}, u_k), \\ \text{such that} \\ f_k(x_{k,low}, u_k) + x_{k,low} = x_{k+1,low} \end{gathered} \tag{5.15}$$

The function f_k is maximized iteratively until a specific tolerance ε is achieved:

$$\left(x_{k+1,low} - \max_{u_k} f_k(x_{k,low}, u_k)\right) < \varepsilon \tag{5.16}$$

For the case of a battery SoC as the state, the function f_k refers to the *charging* of the battery. Given a boundary point $x_{k+1,low}$, the boundary line method will calculate the minimum boundary point $x_{k,low}$ that can charge to $x_{k+1,low}$. Only boundary points $x_{k,low}$, that can charge to the point $x_{k+1,low}$ and also fulfill the load requirement z_k are considered. Thus the infeasible points in Figure 5.13 will be excluded during the calculation of the boundary. With no infeasible points, high gradients can completely be avoided using this method. The same technique is used for calculating the upper boundary, but Equation (5.15) is solved as a minimization problem.

5.5.3 Implementation of Boundary Line Method

a. Extension to multi-dimensional Systems

The boundary line method cannot be directly implemented to drive trains, as the system generally has a multi-dimensional state space. Due to the binary nature of the generator states however, the boundary only needs to be calculated for the battery SoC, and the concept of the *upper* and the *lower* boundary can still be applied to the multi-dimensional state grid.

The maximization in Equation (5.15) is subject to the load requirement at stage k. Any feasible input combination must satisfy the DC grid Equation:

$$I_{g0} + \sum_{i=1}^{N} I_{gi} + I_{bat} + I_{brk} + I_{load} = 0 \tag{5.17}$$

Charging of the battery is achieved by a positive I_{bat}. The generator currents can only assume negative values, and the brake resistance can only assume positive values. Therefore, the maximum available charge to the battery is:

$$I_{g0} + \sum_{i=1}^{N} I_{gi} - I_{brk} - I_{load} = I_{bat} \tag{5.18}$$

I_{brk} can be disregarded as it can be driven to zero independently. Assuming the sum of generators large enough to fully charge the battery, a negative or zero load cycle allows the battery to charge as much as possible, consequently allowing $x_{k,low}$ to assume the lowest possible value from $x_{k+1,low}$. As the load cycle increases in the positive direction, the maximum possible value of I_{bat} also decreases, consequently decreasing the charge available to the battery. The higher the load cycle, the closer will $x_{k,low}$ be to $x_{k+1,low}$. In case of very high load cycles that exceed the maximum current from the generators, the battery must also be discharged and $x_{k,low}$ will be higher than $x_{k+1,low}$, as I_{bat} can only assume negative values.

The maximization in Equation (5.15) does not take any costs into account, therefore, the generator running or switching costs are irrelevant. Thus the lower boundary can be calculated for the battery SoC regardless of the number of generator states. Similar methodology can also be applied for the calculation of the upper boundary.

b. Integration of Boundary line Method with dpm function

Under the assumption that the balance generator is large enough to fully charge the battery, the Boundary line method offered by the dpm function can also be implemented to system with generator states. The integration requires a definition of a reduced model. The reduced model consists of only the battery, brake resistance and the balance generator. Vessel 1 describes a reduced model. The size and parameters of the components are set the same as of their counterparts in the original model. The definition of a single state model allows the use of the Boundary line method implemented in the dpm function, as it is only programmed for single-dimensional systems. An equivalent load cycle must be calculated for the reduced model. The equivalent load cycle for a reduced model is the load cycle that returns the same boundaries for the reduced model as the original load cycle would for the original model. The equivalent load cycle is calculated separately for the lower and the upper boundary calculation.

Lower Boundary Calculation

For each stage of the multi-stage process, the equivalent load cycle for the lower boundary is:

$$I_{leq}(k) = \begin{cases} 0, & I_l(k) \leq \sum_{i=1}^{N} I_{gi,max} \\ I_l(k) - \sum_{i=1}^{N} I_{gi,max}, & I_l(k) > \sum_{i=1}^{N} I_{gi,max} \end{cases} \tag{5.19}$$

For the original model, load requirement achievable by the generators with states leaves balance generator free to fully charge the battery. For the reduced model lower boundary calculation, this is equivalent to 0 load requirement, as 0 load requirement will also leave balance generator to fully charge the battery.

For load requirement that also requires the balance generator, the maximum charge available to the battery is what remains of the balance generator current after the load requirement has been satisfied. The equivalent load requirement is the remaining demand on the balance generator, if all the other generators are running on their maximum capacity.

Upper Boundary Calculation

The upper boundary is calculated by the maximum amount the battery can discharge. The amount of discharge possible is only limited if the load requirement is close to zero or negative. For the lower or negative load requirements, the battery and the brake resistance are the only relevant components and they are identical in the reduced as well as the original model. To keep the load cycle feasible, an upper cap on the load requirement is set.

$$I_{leq}(k) = I_{g0,max}, \qquad I_l(k) > I_{g0,max} \tag{5.20}$$

Any load requirement that exceeds the maximum capacity of the balance generator is set to an achievable value. The equivalence does not matter. As the battery is smaller than the balance generator, it can be fully discharged for both load requirements, equal to balance generators maximum capacity or above.

The calculated boundaries by the boundary line-method on the equivalent basic system can be used as the global boundaries for the original system. The method can be applied to a system regardless of the number of generators with states. A comparison of the boundaries calculated for Vessel 2 using the boundary line and the level set method is shown in Figure 5.15. Both the boundaries are calculated for user defined end-constraints. At high load requirements, the minimum achievable points from the boundary line method are higher.

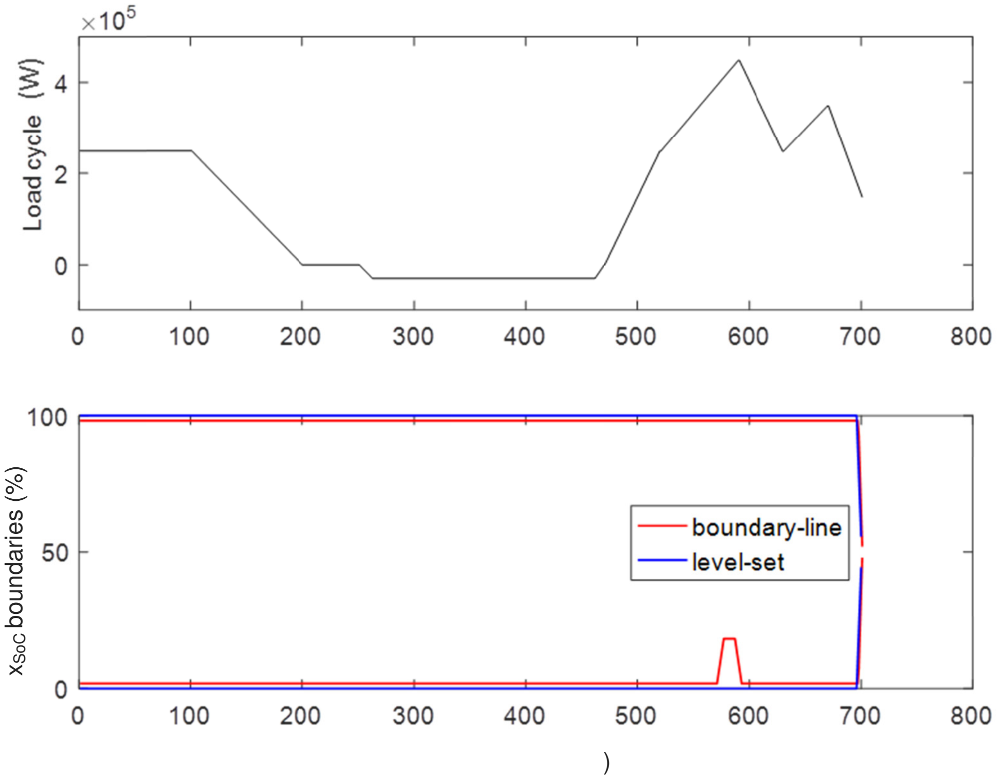


Figure 5.15 Calculation of boundaries using the boundary line method

These points are included in the boundaries calculated from the level set method, and would include infeasible costs within the state space. The boundary line method also introduces an offset from 0% and 100% throughout, which is absent in the level set method. This is due to the tolerance set in Equation (5.16), and is actually a drawback of using this method.

6 Experimental Results

This chapter presents the results of the different aspects of Dynamic Programming investigated in this thesis, applied to the test cases defined in Section 2.3. The studies have been carried out using the dpm function developed in [5].

6.1 Dynamic Programming

This section presents the results of the application of regular dynamic programming to both the test cases. The discretization scheme used for both the test cases is labeled as Scheme 1, to distinguish it from other schemes that will be used in this chapter. The time interval and step size have been kept the same throughout the chapter as summarized in Table 6.1.

Table 6.1 Interval and the step size of optimization

Time steps	Step size
700	1

The dpm function parameters specified for the application of regular dynamic programming are specified in Table 6.2.

Table 6.2 dpm function parameters for regular dynamic programming

Boundary Method	Infeasible cost
Level set	1×10^6

The level set boundary method has been discussed in Section 5.5.1.**Error! Reference source not found.** The infeasible cost is a very high number set to approximate the infinity cost of infeasible points. The infeasible cost is set high enough so that it is higher than any cost or the cost-to-go accumulated during the optimization. An unnecessarily high infeasible cost should also be avoided, as it increases the gradients near boundaries if the Level set method is used.

6.1.1 Application to Vessel 1

The calculation parameters for the application of regular dynamic programming to vessel 1 specified are summarized in Tables 6.3.

Table 6.3(a) State parameters in Scheme 1 for vessel 1

Parameters	**Battery SoC**
Steps	25
Initial State	50
Final State Minimum	48
Final State Maximum	52
State Grid Minimum	0
State Grid Maximum	100

The end constraints for the battery are specified around 50% so that the battery is not completely depleted at the end of the opimization interval. This represents a realistic scenario, as the availabl battery SoC at the start of the interval is also 50%.

Table 6.3(b) Input Parameters in Scheme 1 for vessel 1

Parameters	**Battery Current (A)**	**Brake resistance Current (A)**
Steps	25	5
Minimum	-800	0
Maximum	800	500

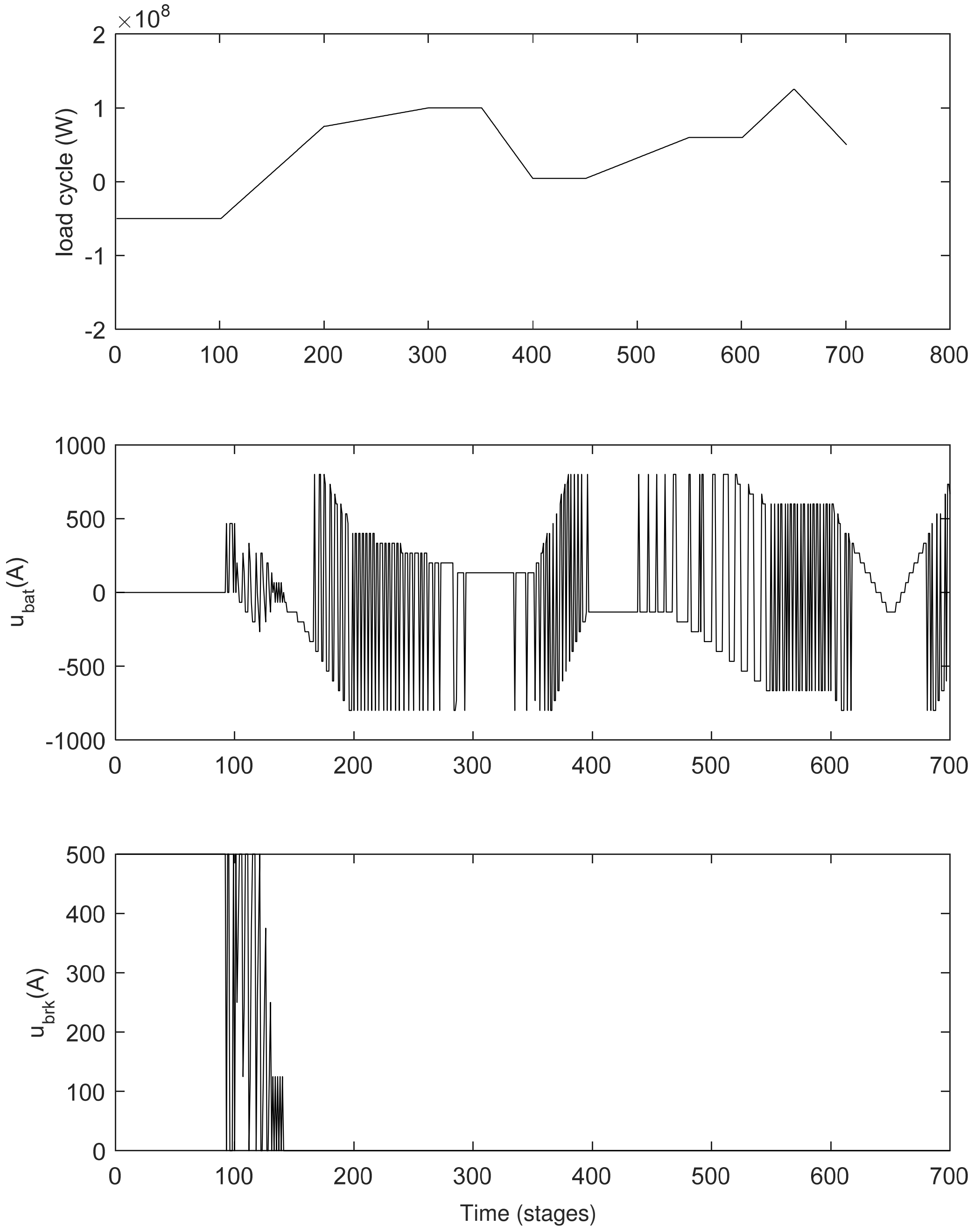


Figure 6.1(a) Optimal input trajectories for vessel 1 for given load cycle

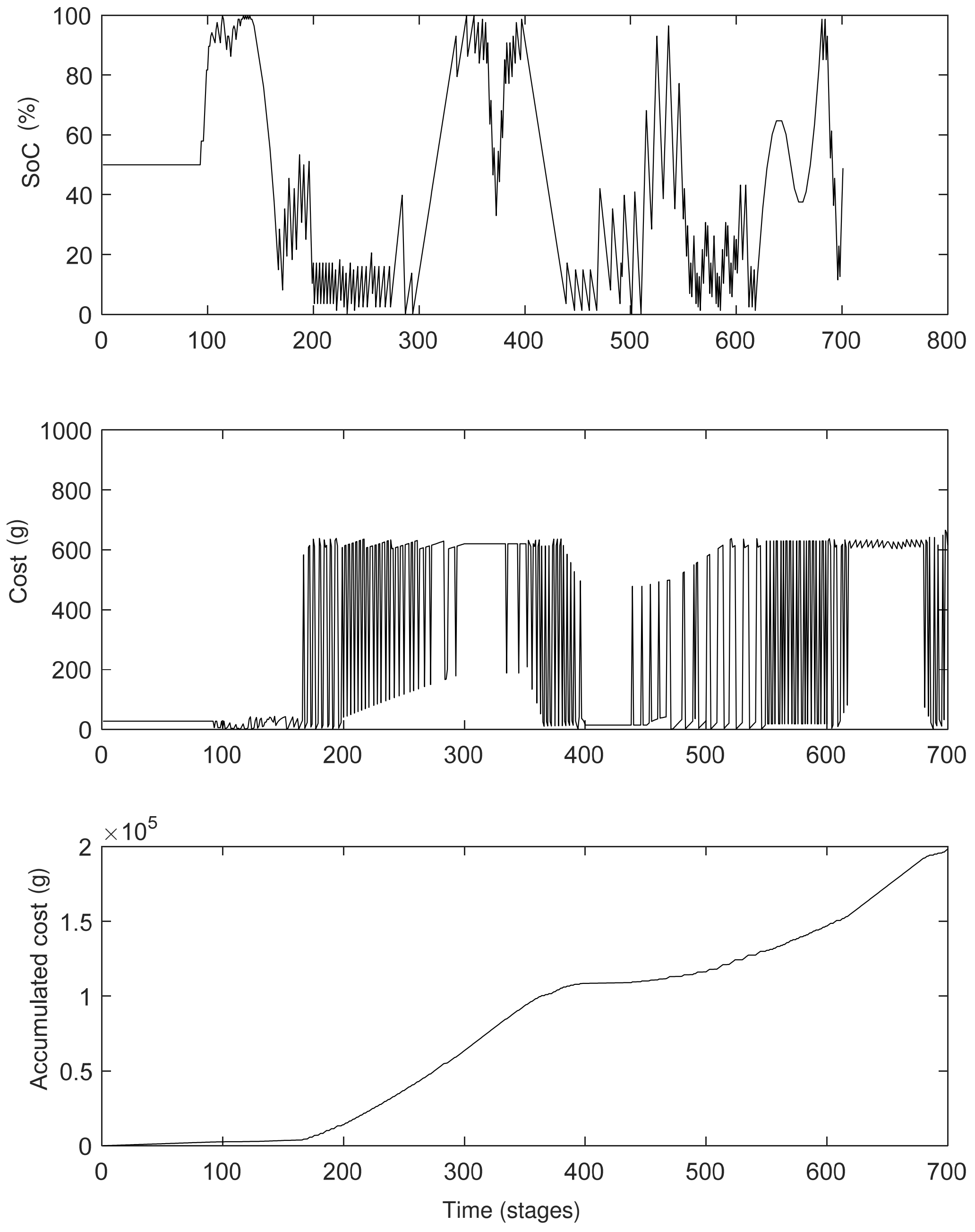


Figure 6.1(b) Optimal state and cost trajectories for vessel 1

6.1.2 Application to Vessel 2

The calculation parameters for the application of regular dynamic programming to vessel 2 specified are summarized in Tables 6.4.

Table 6.4(a) State parameters in Scheme 1 for vessel 2

Parameters	Battery SoC	Generator state
Steps	25	2
Initial State	50	0
Final State Minimum	48	0
Final State Maximum	52	1
State Grid Minimum	0	0
State Grid Maximum	100	1

The generator states are only binary, and represent wether the generator has been switched on or off. The state is implemented in order to introduce a penalty for switching on a generator.

Table 6.4(b) Input Parameters in Scheme 1 for vessel 2

Parameters	Battery Current (A)	Brk. Res. Current (A)	Generator Current (A)
Steps	25	5	20
Minimum	-800	0	-1000
Maximum	800	500	0

Trajectory Oscillations

The oscillatory behavior of the state as well as the optimal control inputs can be observed for both the test cases. The behavior arises due to the non-linear nature of the generator fuel consumption function.

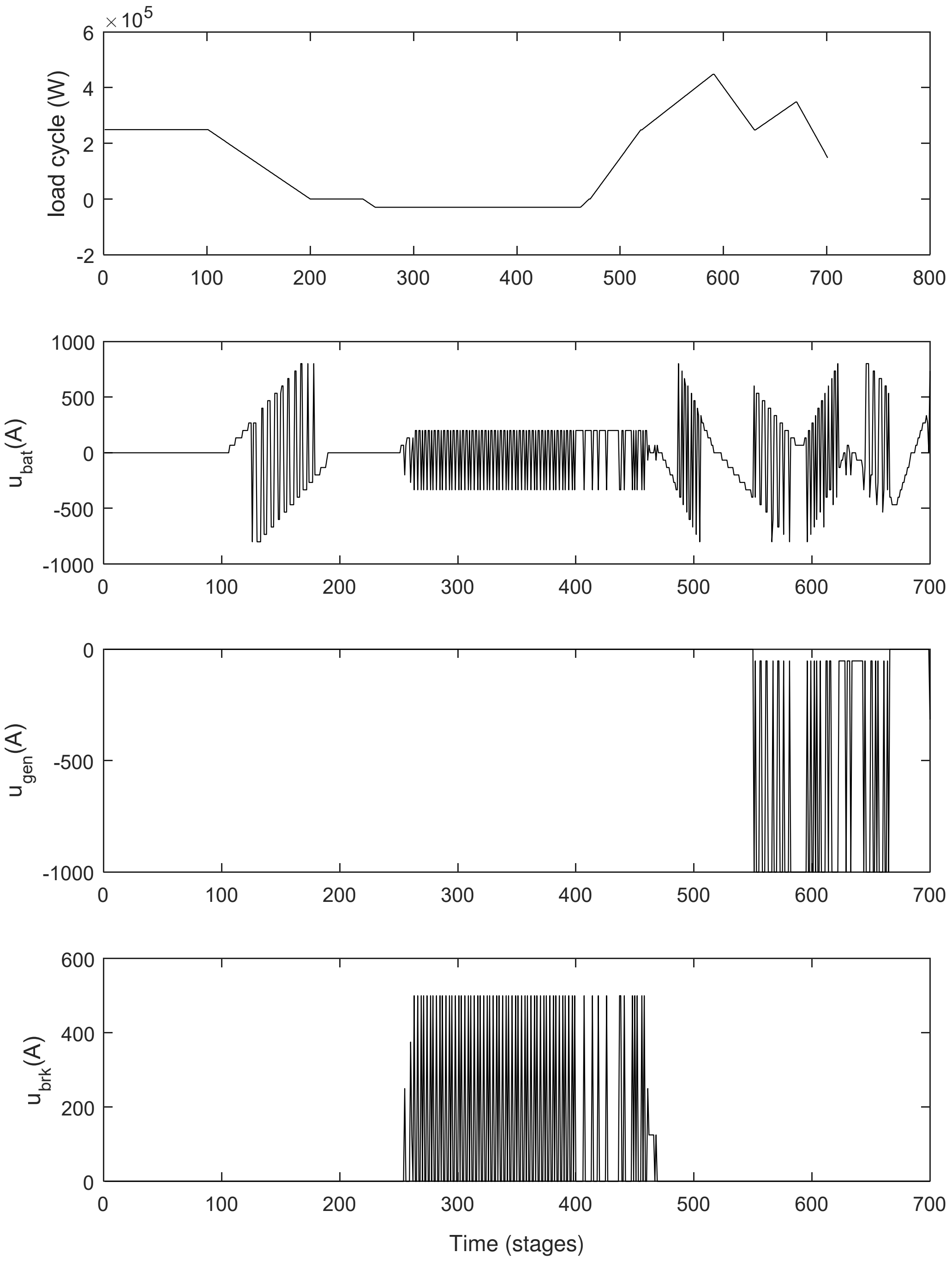


Figure 6.2(a) Optimal input trajectories for vessel 2 for given load cycle

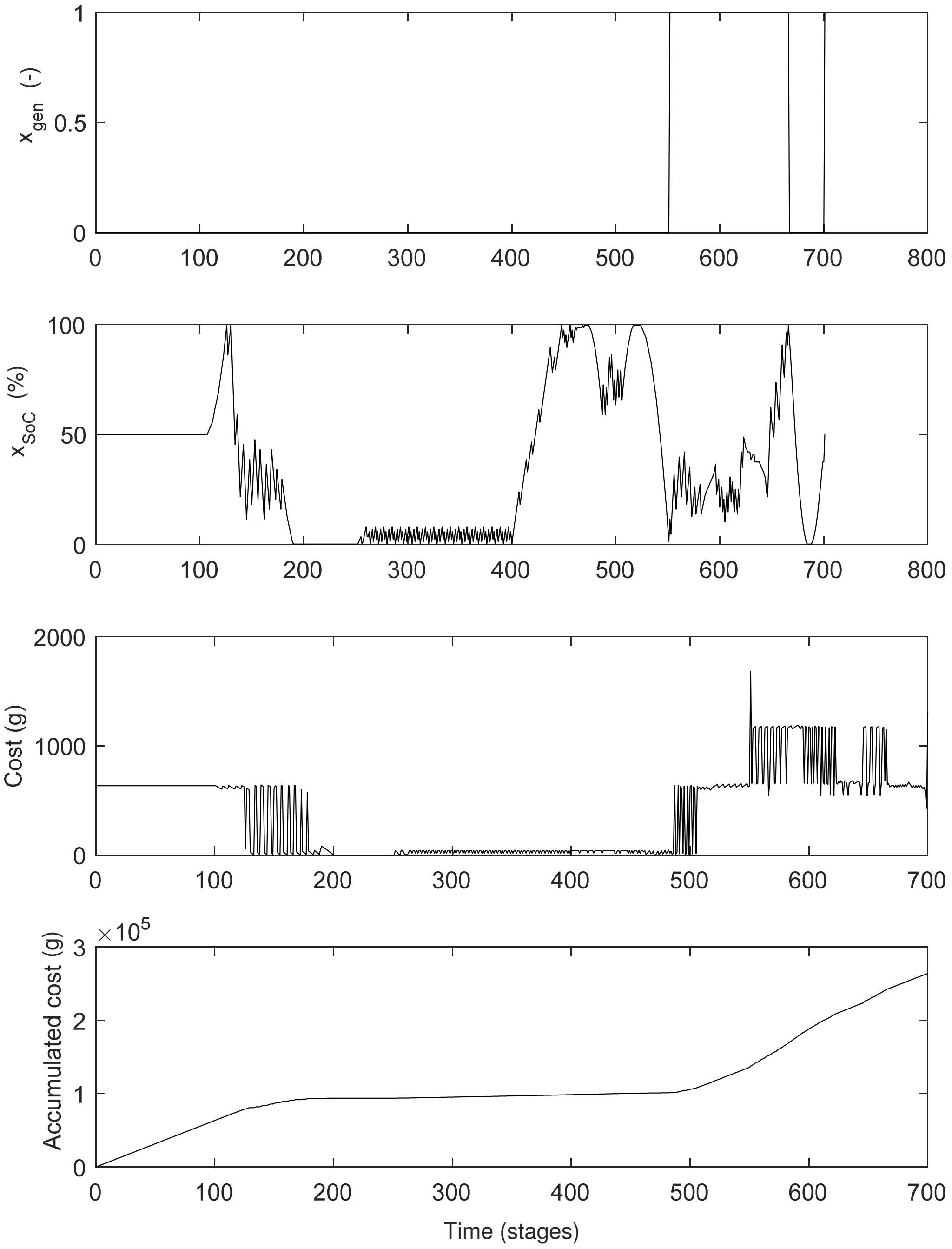


Figure 6.2(b) Optimal state and cost trajectories for vessel 2

For the load cycle interval that is below the optimal capacity of the generators, it is cheaper to alternatively use the genera tors at full and minimum capacity, as compared to running them at mid-capacity. This can explicitly be seen in Figure 6.2 (a) in the optimal control input trajectory for the generator. The control input oscillates only between minimal and maximum operation.

6.2 Iterative Dynamic Programming

The results of the iterative dynamic programming are compared with regular dynamic programming applied to different discretization schemes, in addition to Scheme 1 introduced in Table 6.3 and 6.4. The range of the states and control inputs is kept the same, but the number of steps is varied. The schemes are summarized in the Tables 6.5 and 6.6.

Table 6.5 Discretization Scheme 2

State / Control Input	**Number of Steps**
Battery SoC	35
Battery Current	35
Brk. Res. Current	10
Generator Current	30

The discretization scheme 3 is only defined for Vessel 1, as the computational complexity for its application to Vessel 2 is extremely high.

Table 6.6 Discretization Scheme 3

State / Control Input	**Number of Steps**
Battery SoC	50
Battery Current	55
Brk. Res. Current	15

The discretization for IDP2, as well as the reduction factor is the same for both the test cases, and is summarized in table 6.7.

Table 6.7 Calculation paramters for IDP2

IDP2 parameters	
Reduction factor γ	0.8
Convergence Tolerance	200
SoC steps	10
Initial battery current steps	11
Initial brk. Res. current steps	5
Initial generator current steps	10
SoC steps	5
Battery current steps	5
Brk. Res. current steps	5

The convergence tolerance is the criteria for the termination of the iterative procedure. If the performance index (total cost) of two consecutive iterations falls within this tolerance, the procedure is assumed to be converged and terminated. As discussed in Section 0, the initial discretization is kept high to improve the chances of a good initial policy.

6.2.1 Application to Vessel 1

Figure 6.3 shows the perorfmance indexes of consequetive IDP2 iterations, compared with regular dynamic programming with the three afore-mentioned schemes, applied to Vessel 1.

Due to the implemented grid adjustment algorithm, the performance of IDP2 always improves along iterations. Despite the apparently good performance, the algorithm has converged to a local optimum with respect to discretization scheme 3.

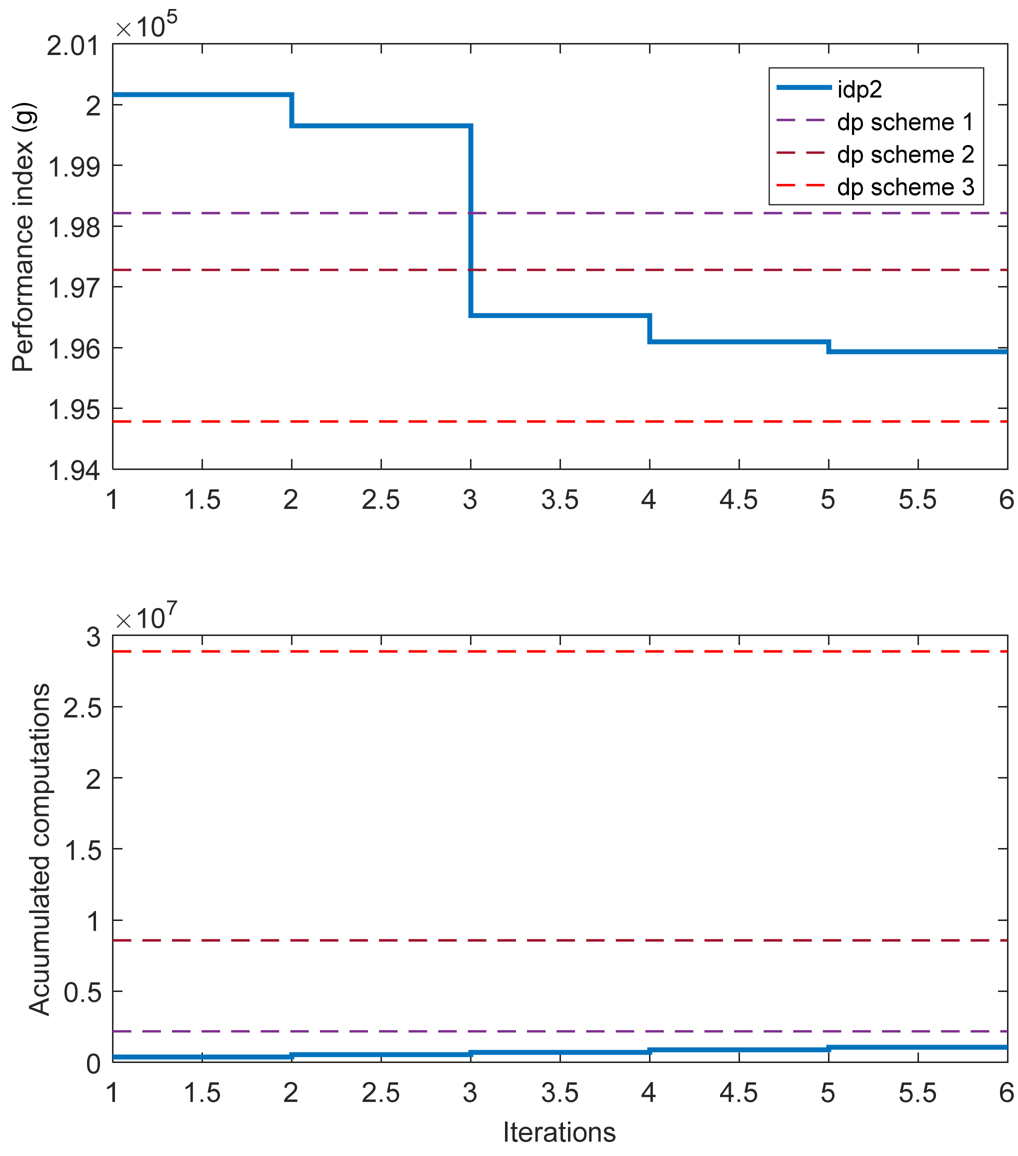


Figure 6.3 Performance of idp2 compared with regular dp on vessel 1

This is further highlighted in the comparison of the optimal trajectories between IDP2 and regular DP on scheme 3, shown in Figure 6.4. The solution is does not converge towards the better optimum, rather converges to a different one. Therefore, although Iterative Dynamic Programming shows superior performance in the cases tested, it always poses the risk of convergence to poor optima.

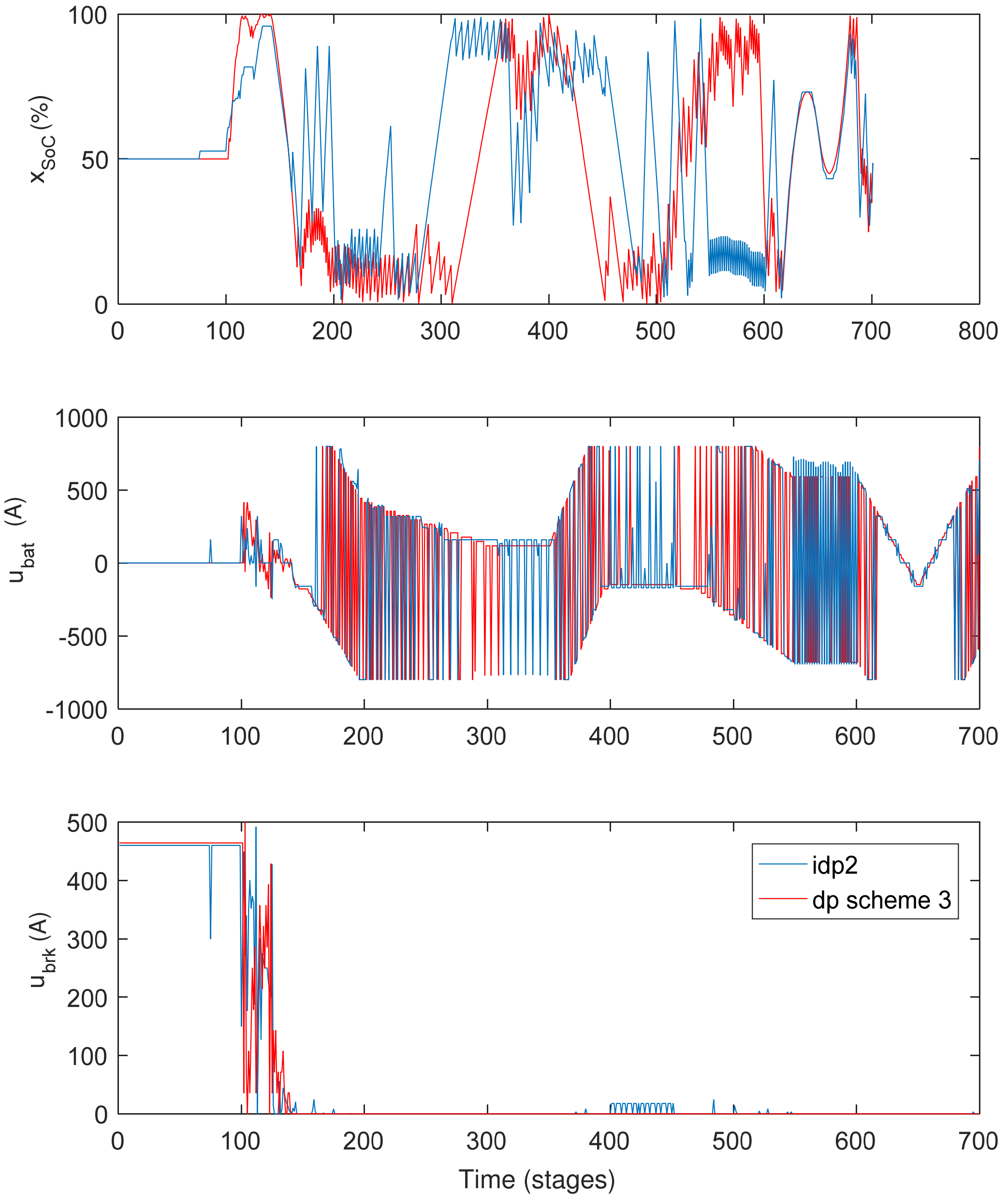


Figure 6.4 Comparison of optimal trajectories of idp2 and regular dp on vessel 1

6.2.2 Application to Vessel 2

Figure 6.5 and 6.6 show the application of IDP2 to vessel 2, compared with regular dynamic programming. This is one of the cases where the initial coarse discretization exceeds the performance of the finer discretization in discretization scheme 1.

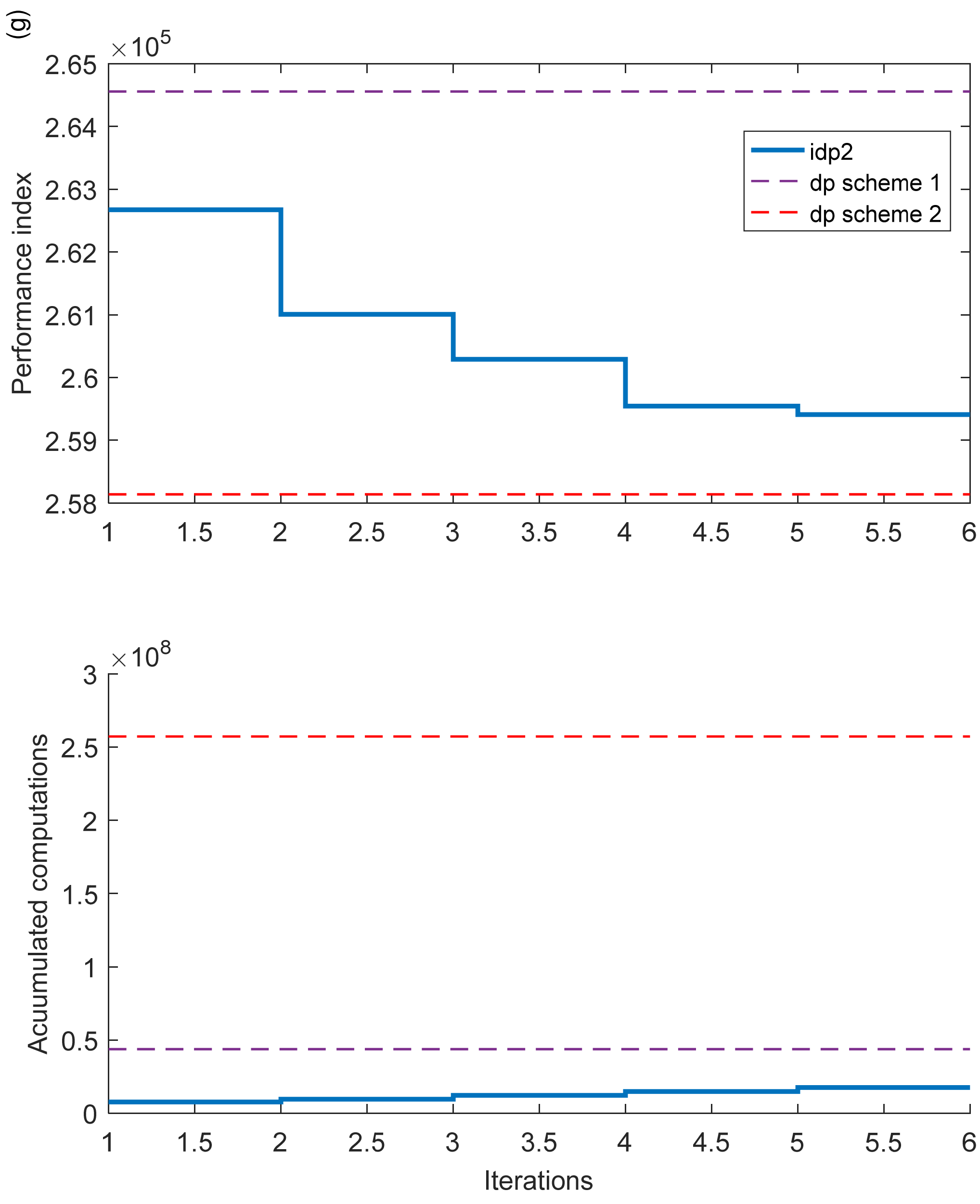


Figure 6.5 Performance of idp2 compared with regular dp on vessel 2

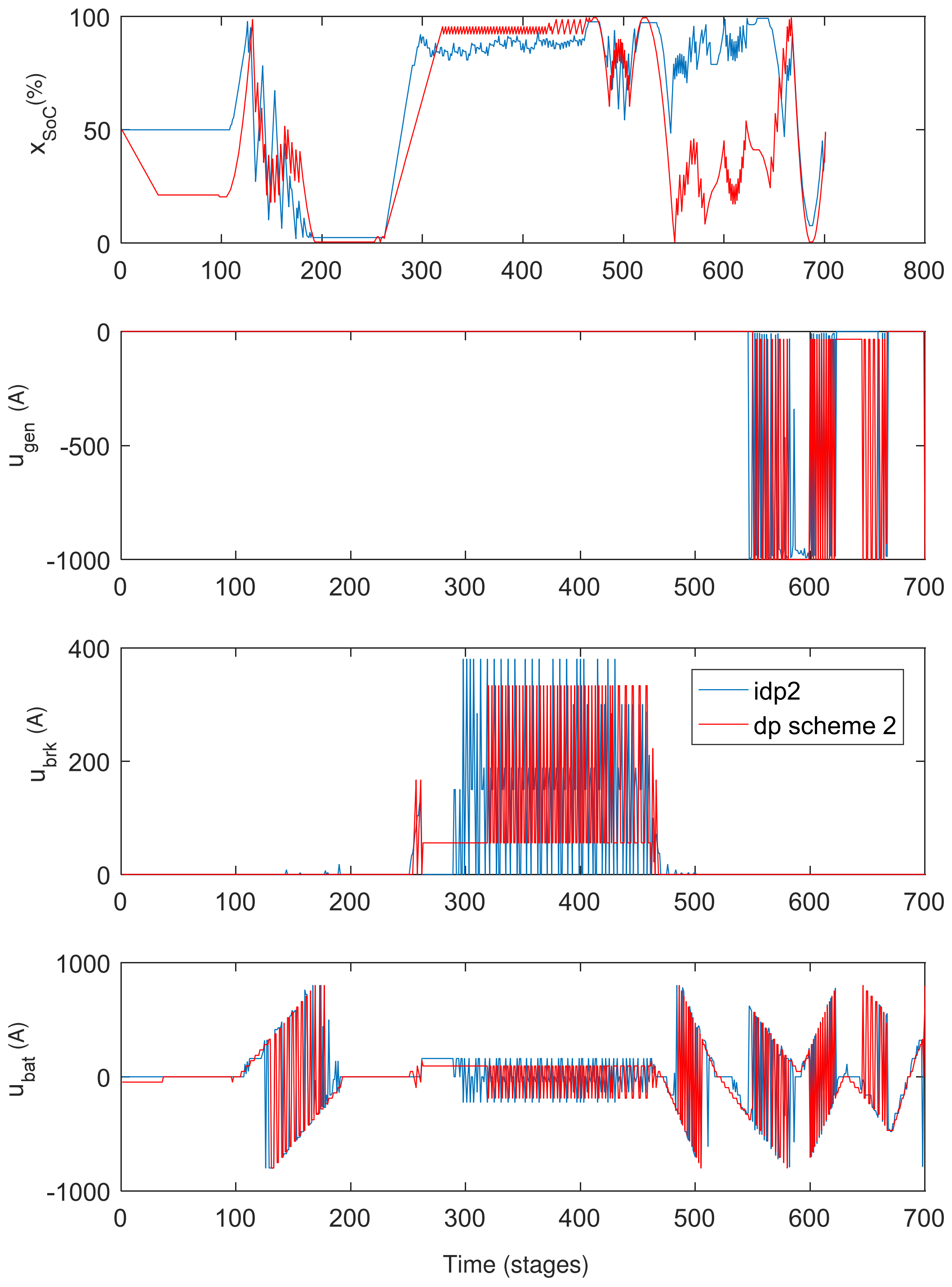


Figure 6.6 Comparison of optimal trajectories of idp2 and regular dp on vessel 2

6.3 Error in Dynamic Programming

This section presents the experimental results associated with the errors in dynamic programming as discussed in Chapter 5.

6.3.1 Sensitivity to Discretization

The sensitivity analysis is only carried out for vessel 1, as it requires repeated optimization experiments under different configurations. Such analysis for vessel 2 requires considerable time and computational effort. The performance index for varying state and input discretization schemes of vessel 1 is shown in Figure 6.7. The state and input discretization are increased geometrically.

$$\Delta u_i = \lceil 1.5(\Delta u_{i-1}) \rceil, i = 1,2,\dots,N \tag{6.1a}$$

$$\Delta x_i = \lceil 1.5(\Delta x_{i-1}) \rceil, i = 1,2,\dots,N \tag{6.1b}$$

The initial state and input discretization is summarized in Table 6.8. As the state and input discretization is varied independently, the corresponding constant state and input discretization is also listed.

Table 6.8 Parameters for sensitivity analysis to discretization

State / Control Input	State Variation	Input Variation
Battery SoC	3 (initial)	16 (constant)
Battery Current	5 (constant)	3 (initial)
Brk. Res. Curent	11 (constant)	5 (initial)

The experiments in Figure 6.7 are carried out with the same state and input ranges as listed in Table 6.3, but no end-constraints area added. The figure brings various observations about the sensitivity of the performance index to discretization to light.

Firstly, the performance index is more sensitive to input discretization as compared to the state discretization. An explanation of this is that the input discretization effects the error of representation on both the greedy cost and the cost-to-go, while the state discretization only effects the representation error on the cost-to-go.

Secondly, a sharp decline in the performance index can be observed between the state discretization 2 and 3, the point where the minimum state discretization is satisfied. As there are no end-constraints included, the error cannot be attributed to the restricted state space of the level set method (Section 5.5).

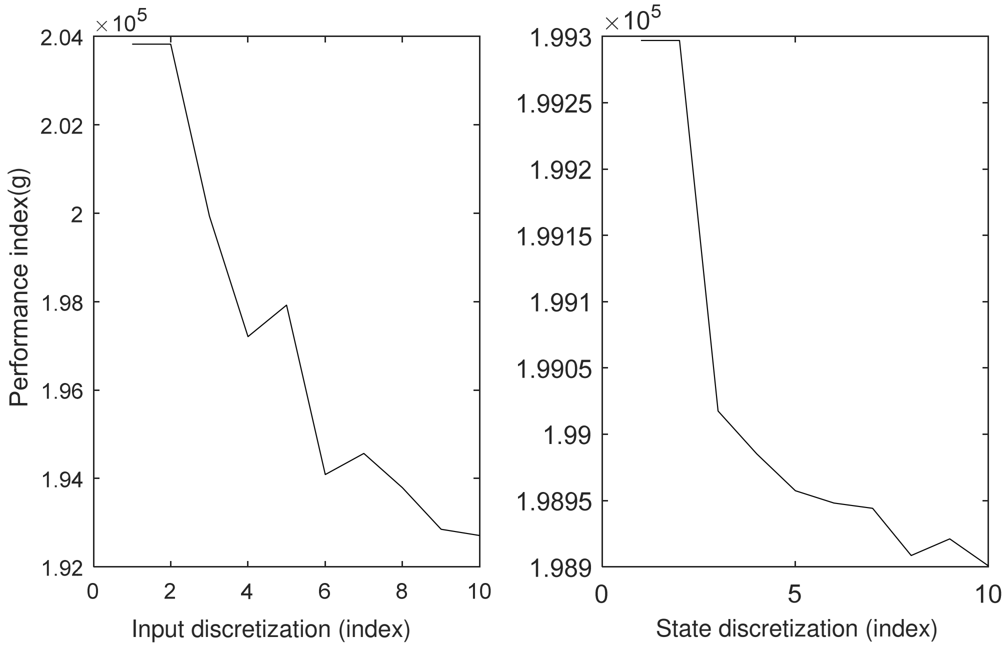


Figure 6.7 Sensitivity of performance index to discretization in Vessel 1

The other explanation is the additional representation error due to the voilation of the minimum state discretization (Section 5.3.1).

Thirdly, performance index does not always improve with the increase in the state and input discretization. This confirms that the increase in discretization only increases the probability of improving the solution (Section 5.2.2).

The explanations offered for the observations in Figure 6.5 are only conjecture on part of the author. As the experiments have not been repeated for more test cases, the phenomena (higher input discretization dependence and sharp decline for state discretization) might also be completely random events.

6.3.2 Error Factor Calculation

The offline error factor (Section 5.4.1) is calculated for vessel 1 for the different input discretizations used in Figure 6.8. The user parameters for error factor calculation are listed in Table 6.9.

Table 6.9 Parameters for error factor calculation

Parameter	**Set value**
Error threshold	0.5
Number of intermediate points	5

Any error factor lower than the error threshold is penalized. The number of intermediate points is the number of points added between all the adjacent points of the input grid to check the liearity of the cost function.

For each input discretization, the error factor is calculated for fifty evenly spaced samples of the load cycle in Figure 6.1 (a). The displayed error factor in Figure 6.8 is the average of the error factors for each load cycle sample.

The error factor corresponds to the probability of improving the opimal solution, and it does match the evolution of actual performance indexes for different input discretizations, particularly in the initial points of high slope. Despite the oscillations in the performance index at the later points, it also generally converges in the same region where the error factor converges. Thus the error factor can be used to find a reasonable input discretization before the optimization procedure.

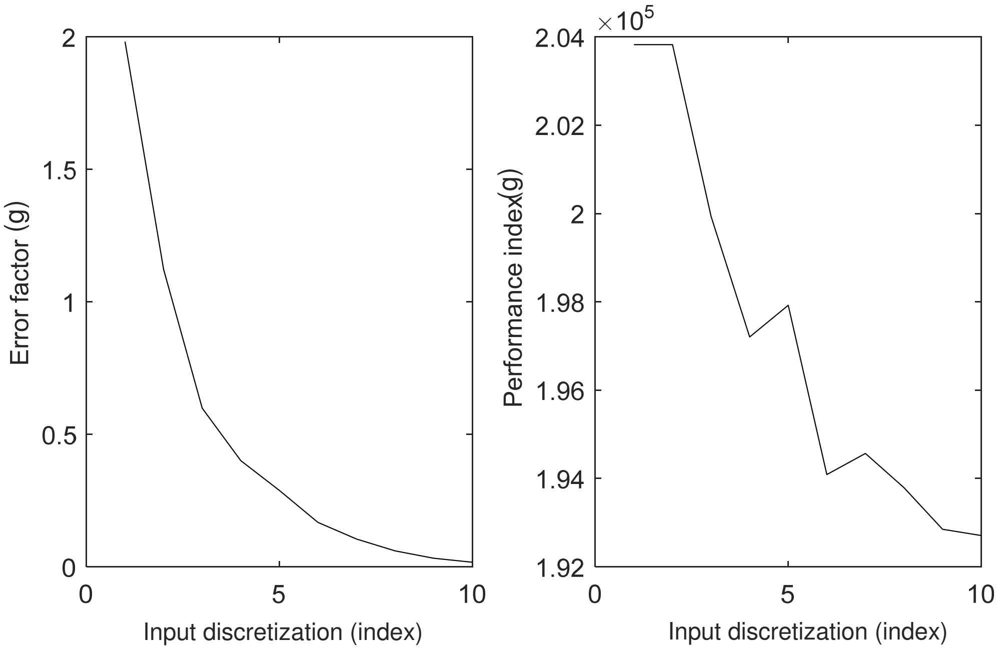


Figure 6.8 Offline error factor computed for different discretizatons vs actual performance

6.3.3 Adaptive Input Grid

A method to compute an adaptive input grid based on the error factor is introduced in Section 5.4.1. Based on five different thresholds of the error factor, adaptive input discretization schemes have been computed and the performance of dynamic programming on these schemes is shown in Figure 6.9. The performance is compared with uniform input discretization schemes.

The adaptive grid clearly has a superior performnce to the uniform grid, as it computes a better optimum in roughly ten times lesser number of computations.

6.3.4 Boundary line Method

The boundary line method has been implemented to vessel 2 using the simplification techniques introduced in Section 5.5.2. The cost-to-go functions for the two methods, level set and boundary line are compared in Figure 6.10 for stages with high load requirements. It can be seen that the high gradients in the cost-to-go from the level set method, that propagate through the stages, are absent in the cost-to-go from the boundary line method.

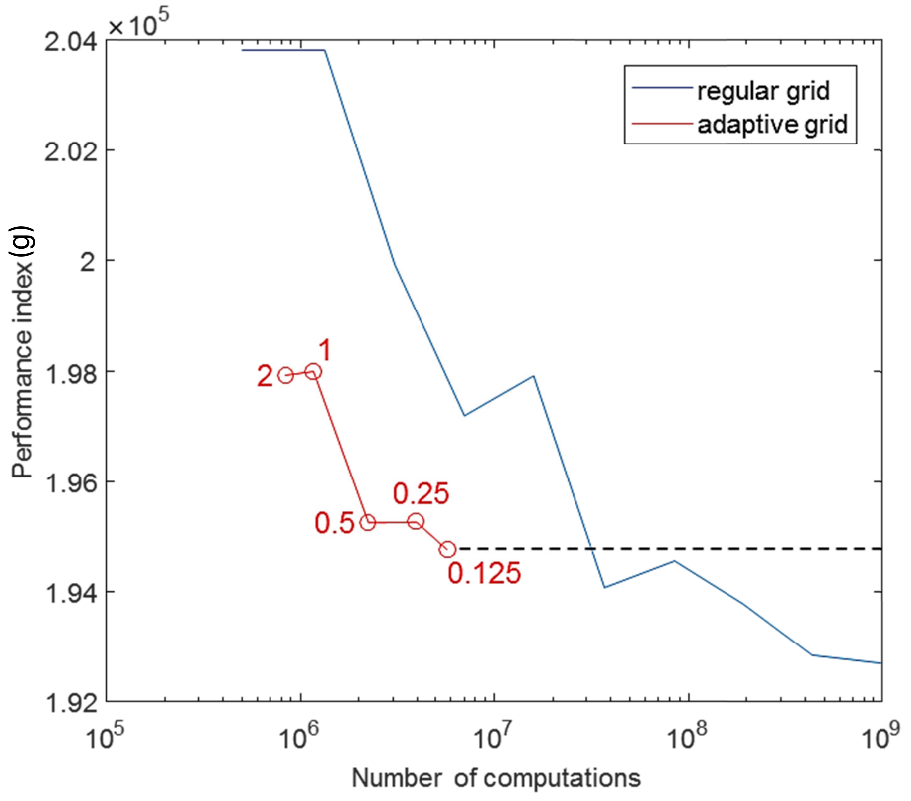


Figure 6.9 Comparison of adaptive input grid with uniform input grid

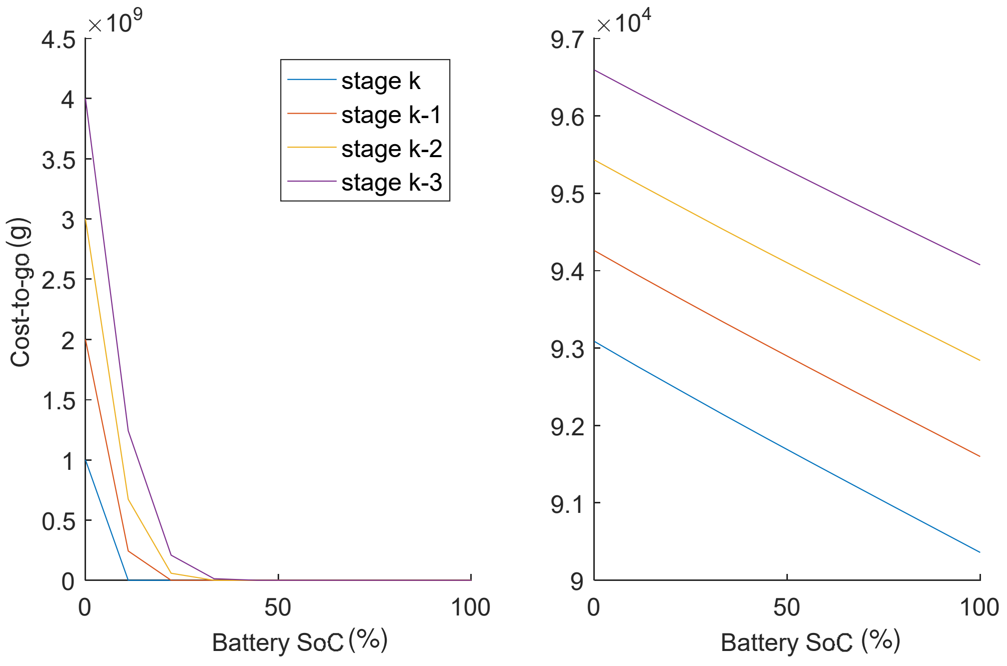


Figure 6.10 Comparison of the cost-to-go gradients from the level set and boundary line method

7 Conclusion and Future Work

The advantage of dynamic programming is that it guarantees the optimality for a given problem. For the application of dynamic programming to continuous systems, the continuous system must be discretized, which can lead to a difference of the optimal results for a discretized system from the optimal results for a continuous system. The optimization on a very fine discretization has a high chance of returning the global optimum close to that of the continuous system, but the computational restrictions make very fine discretization schemes practically infeasible.

7.1 Errors of Discretization

For the case of marine drive trains included in the Power Management Tool, the studies carried out in this thesis give a concrete analysis of the source and nature of the optimization errors due to discretization. The results can be used by the user to estimate (qualitatively or quantitatively) the probable error introduced by discretization, and include it with the optimal cost to offer more realistic results.

In addition, the analysis of errors has also lead to suggestions that can be taken to reduce the risk of errors without increasing the computational complexity. Firstly, as the optimal solution has shown a higher sensitivity to input as compared to state discretization, the input discretization should be set higher. Secondly, the proposed algorithm of adapting the input grid discretization to the fuel consumption curve can be used to further reduce the probability of representation errors.

There exist numerous aspects of study which can be carried out in future to further facilitate the user in using the optimal results. Some of them have been suggested as follows:

a. Analytical Modeling of Generators

The analytical modeling of generators can be highly useful in computing the mathematical probability of errors associated with discretization. One of the suggested areas of study was introduced in Section 5.4.2, which proposes the calculation of the worst case errors based on the frequency analysis of the fuel consumption of the generators.

b. Non-Uniform Grid

The dpm function, although highly useful, also puts limitations on the flexibility in dynamic programming algorithm. One of the limitations is that it only allows the generation of grids with equally spaced intervals between the set minimum and maximum range. A non-uniform spacing of the grid, with more points in regions of non-linearity may prove beneficial to reduce the representation errors without increasing the computational effort.

c. Non-Linear Interpolation

Another limitation by the dpm function is that it only allows linear interpolation in the cost-to-go values. A study into the implementation of polynomial based interpolations can also be carried out to see the effect on the interpolation errors.

7.2 Reduction of Computational Complexity

In order to reduce the computational complexity of optimization procedures, iterative dynamic programming has been shown as a promising alternative to dynamic programming. Iterative dynamic programming offers the probability of an optimal solution with significantly lesser number of computations. However, it also exhibits the drawback of only searching the limited state and input space, which poses the risk of sub-optimal convergence. Despite the additional risk of sub-optimal convergence, a guarantee for the worst possible solution (with high probability of significantly better results) can be formulated.

a. Parallelization of IDP

The reduction factor and the initial policy can have a large effect on the convergence and the rate of convergence of the algorithm to the final solution. A suggested area of study is to parallelize multiple processes of IDP with different parameters. There can be two variants of this approach:

1. Start the parallel processes with the same initial policy, but different reduction factors. In the case that the initial guess lies in the vicinity of the optimal solution, the process with a high reduction factor can quickly converge to the solution. In the case of a bad initial guess, processes with a lower reduction factor propose additional chances of escaping the local optimum. The parallel processes can also communicate the results amongst to improve successive iterations.
2. Use different initial policies for different parallel processes. This results in the search of multiple regions of the state space in parallel.

Bibliography

1. Energy Efficiency: the Other Alternative Fuel. *http://www.abb.com.* [Online] http://www.abb.com/cawp/abbzh252/1e61c6abed230ba6c12571bf0058af8a.aspx.

2. Ambühl, Daniel. *Energy management strategies for hybrid electric vehicles*. Diss. ETH ZURICH, 2009.

3. Radke, Tobias. *Energieoptimale Längsführung von Kraftfahrzeugen durch Einsatz vorausschauender Fahrstrategien*. Vol. 19. KIT Scientific Publishing, 2013.

4. Wahl, Hans-Georg. *Optimale Regelung eines prädiktiven Energiemanagements von Hybridfahrzeugen*. Vol. 43. KIT Scientific Publishing, 2015.

5. Sundstrom, Olle, and Lino Guzzella. "A generic dynamic programming Matlab function." *Control Applications,(CCA) & Intelligent Control,(ISIC), 2009 IEEE*. IEEE, 2009.

6. Richard Ernest Bellman. *Dynamic Programming*. Dover Publications, Incorporated, 2003.

7. Bellman, Richard E., and Stuart E. Dreyfus. *Applied dynamic programming*. Princeton university press, 2015.

8. De Madrid, A. P., S. Dormido, and F. Morilla. "Reduction of the dimensionality of dynamic programming: A case study." *American Control Conference, 1999. Proceedings of the 1999*. Vol. 4. IEEE, 1999.

9. Canto, Sebastian Dormido, Angel P. de Madrid, and Sebastiįn Dormido Bencomo. "Parallel dynamic programming on clusters of workstations." *IEEE Transactions on Parallel and Distributed Systems* 16.9 (2005): 785-798.

10. Tang, Shanjiang, et al. "Easypdp: an efficient parallel dynamic programming runtime system for computational biology." *IEEE Transactions on Parallel and Distributed Systems* 23.5 (2012): 862-872.

11. Luus, Rein. "Optimal control by dynamic programming using systematic reduction in grid size." *International Journal of Control* 51.5 (1990): 995-1013.

12. Dadebo, S. A., and K. B. McAuley. "Dynamic optimization of constrained chemical engineering problems using dynamic programming." *Computers & chemical engineering* 19.5 (1995): 513-525.

13. Wahl, Hans-Georg, and Frank Gauterin. "An iterative dynamic programming approach for the global optimal control of hybrid electric vehicles under real-time constraints." *Intelligent Vehicles Symposium (IV), 2013 IEEE*. IEEE, 2013.

14. Bertsekas, D. "Convergence of discretization procedures in dynamic programming." *IEEE Transactions on Automatic Control* 20.3 (1975): 415-419.

15. Fox, Bennett L. "Discretizing dynamic programs." *Journal of Optimization Theory and Applications* 11.3 (1973): 228-234.

16. Shannon, Claude Elwood. "Communication in the presence of noise." *Proceedings of the IRE* 37.1 (1949): 10-21.

17. Elbert, Philipp, Soren Ebbesen, and Lino Guzzella. "Implementation of dynamic programming for $ n $-dimensional optimal control problems with final state constraints." *IEEE Transactions on Control Systems Technology* 21.3 (2013): 924-931.

18. Sundström, O., D. Ambühl, and L. Guzzella. "On implementation of dynamic programming for optimal control problems with final state constraints." *Oil & Gas Science and Technology–Revue de l'Institut Français du Pétrole* 65.1 (2010): 91-102.

Filename:	master thesis 2.docx
Directory:	C:\Users\Maqbool\Documents
Template:	C:\Users\Maqbool\AppData\Roaming\Microsoft\Templates\Normal.dotm
Title:	
Subject:	
Author:	Maqbool
Keywords:	
Comments:	
Creation Date:	4/3/2017 2:27:00 AM
Change Number:	13
Last Saved On:	4/17/2017 8:44:00 PM
Last Saved By:	Maqbool
Total Editing Time:	1,742 Minutes
Last Printed On:	4/17/2017 8:45:00 PM

As of Last Complete Printing

Number of Pages:	101
Number of Words:	19,553 (approx.)
Number of Characters:	111,457 (approx.)