FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Optimal motion strategies with logic-based constraints for ocean vehicles

Miguel Campos Pinto Coelho de Aguiar

U. PORTO

FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

Mestrado Integrado em Engenharia Eletrotécnica e de Computadores

Supervisor: João Borges de Sousa

Co-supervisor: Jorge Estrela da Silva

July 17, 2019

Abstract

Optimal trajectory generation for ocean vehicles has attracted considerable attention from both the research community and the industry. In the industry, the motivation is the reduction of travel times and fuel costs, and the focus is on long distance routes. Research on ship routing algorithms has shown that the fuel savings attained by such algorithms can be strongly affected by the ocean currents. In the research community, the focus is on trajectory generation for unmanned and autonomous marine craft in military and/or scientific applications. Here the mission times and distances are typically much shorter, but ocean current vary on smaller time scales and their magnitude can be as high as twice the vehicle's maximum speed. As such there is a pressing need for mission planning algorithms that are able to incorporate data from high temporal-spatial resolution ocean models. As mission requirements become more complex, planning methods must also be able to take into account spatial and temporal constraints which arise in scenarios such as multi-stage operations in areas with tidal driven currents.

We propose a method for trajectory generation for unmanned marine vehicles based on dynamic programming. The application of dynamic programming techniques converts an optimal control problem to the problem of solving a Hamilton-Jacobi-Bellman equation, which is a nonlinear partial differential equation. Data about ocean flows, produced by High Frequency radar or ocean models, is easily integrated in this framework. Our parallel implementation of a numerical method for solving Hamilton-Jacobi-Bellman equations allows us to obtain the solution in a few minutes for real-life sized problems. Once the equation is solved, optimal trajectories can be calculated from any deployment point in the operational area. Since dynamic programming can be applied to dynamical systems with both discrete and continuous states, the method is extensible to problems involving logic-based constraints. We present an efficient dynamic programming solution for trajectory generation in multi-stage missions. The problem is reduced to solving a sequence of partial differential equations, each of which can be solved by our numerical solver.

The method is validated through real-life mission scenarios using data from ocean models of the Tejo and Sado estuaries in Portugal.

Acknowledgments

I must begin by thanking my advisor, Professor João Borges de Sousa. His no-nonsense guidance and untiring encouragement were crucial for the success of this work, as were the many hours of interesting discussions.

The contributions of Jorge Estrela da Silva and Renato Mendes were also very important. I thank them sincerely for their constant availability, patience and interest in this work.

I also wish to gratefully acknowledge the contribution of our collaborators at NMEC-CESAM, Américo Ribeiro and João Miguel Dias, who graciously provided the data from their ocean models of the Sado and Tejo estuaries used in this work.

I was lucky to be able to work on this thesis at LSTS, where I found a great work team and environment, which helped make the daily grind a lot more tolerable. Special thanks to José Pinto, Maria Costa and Paulo Dias for their help with using the LSTS toolchain. LSTS also supported my participation in a number of eye-opening conferences and meetings, where I was able to get feedback on this work. My participation at the extremely interesting interdisciplinary Symposium on Oceanographic Data Analytics was also made possible thanks to the help of João Fortuna.

I made a lot of great friends these past five years, and I'll treasure our days at FEUP, as well as the nights in downtown Porto. I've learned a lot from all of them, and I know they'll continue to make me proud to know them in years to come.

Finally, I'd like to thank my family, and in particular my parents, who taught me the joys of learning.

Miguel Aguiar

The whole purpose of mountain-climbing to me isn’t just to show off you can get to the top, it’s getting out to this wild country.

Jack Kerouac (in *The Dharma Bums*)

Contents

List of Figures

List of Tables

Symbols and Abbreviations

UUV	Unmanned Underwater Vehicle
AUV	Autonomous Underwater Vehicle
LSTS	Laboratório de Sistemas e Tecnologia Subaquática
FTLE	Finite Time Lyapunov Exponent
LCS	Lagrangian Coherent Structure
PDE	Partial Differential Equation
HJB	Hamilton-Jacobi-Bellman
HJBE	Hamilton-Jacobi-Bellman Equation
FSM	Fast Sweeping Method
HPC	High Performance Computing
GCC	GNU Compiler Collection
WGS	World Geodetic System
UTM	Universal Transverse Mercator

Notation

$\mathbf{R}$	the set of real numbers
$\mathbf{R}_{\geq 0}$	the set of nonnegative real numbers
$\mathbf{R}^n$	the vector space of n-tuples of real numbers $\boldsymbol{p} = (p_1, \dots, p_n)$
$\lvert\boldsymbol{p}\rvert$	Euclidean norm of a vector $\boldsymbol{p} \in \mathbf{R}^n$
τ	deployment time of the vehicle
$\boldsymbol{x}$	horizontal position of the vehicle, an element of $\mathbf{R}^2$
$\dot{\boldsymbol{x}}$	velocity of the vehicle, an element of $\mathbf{R}^2$
r	maximum propulsion speed of the vehicle
$\boldsymbol{u}$	control function, taking values in $\mathbf{R}^2$
$\boldsymbol{v}$	ocean flow velocity, an element of $\mathbf{R}^2$
Ω	target set, a subset of $\mathbf{R}^2$
q, g	components of the cost function
J	cost function, mapping a trajectory to a positive real number
V	value function

Chapter 1

Introduction

1.1 Motivation and context

Over the past two decades, advances in navigation, control and communications have brought unmanned and autonomous marine vehicles (U/AUVs) to the forefront of ocean exploration [7]. From a surveillance and defense point of view, several kinds of missions, such as mine countermeasures, near-land and harbor monitoring, monitoring of undersea infrastructure such as communication cables, or anti-submarine warfare, greatly benefit from the use of unmanned ocean vehicles. Autonomous vehicles facilitate operations in areas which are unreachable by surface vessels, and can also reduce or eliminate the human risk factor in the mission [10]. From a scientific and industrial point of view, autonomous underwater vehicles have been successfully used in surveys such as seafloor mapping and monitoring and geochemical water column measurements, and have enabled data collection at resolutions that are not achievable with traditional ship-based surveys and in previously inaccessible areas, e.g. beneath ice sheets [70].

At the Underwater Systems and Technology Laboratory (LSTS), researchers in computer science, electrical engineering, mechanical engineering and oceanography have been working on the development and deployment of networked vehicle systems for marine applications, with emphasis on AUV operations [18, 19]. In this context, persistent operations are an important goal, and trajectory generation can play an important role in reducing energy consumption, for instance.

In the context of ship routing algorithms, simulations have shown that ocean currents can have a major influence in the fuel savings achieved by route optimization algorithms [35]. These studies also concluded that the magnitude of the fuel savings depends strongly on specific ocean current patterns, implying that rigorous model-based analyses can have a significant edge over heuristic methods [40].

Typical unmanned vehicle missions involve much smaller distances and time frames than those involved in ship routing problems, and as such it is expected that the impact of the ocean flow can be even more significant, as small scale variations in the current velocity become relevant. Besides their effect on energy consumption, ocean currents can have substantial impact on mission feasibility: in some operating environments such as fjords or estuaries, ocean vehicles can face currents whose magnitudes exceed their maximum speed, and even in areas where the ocean current has relatively low amplitude, it can still be relevant for missions with weak propulsion vehicles such as gliders.

1.2 Problem description

In this work we focus on two particular problems of trajectory generation for unmanned underwater vehicles. The first, simpler problem, which we call the ‘base problem’, consists in generating a trajectory from a given deployment position to a target position or target region, while minimizing some cost function. As an example of a practical implementation, consider a vehicle which is deployed from a ship and must travel to some predefined region where it will perform a survey. It only makes sense to use forecasts of the ocean current to plan the trajectory from the ship to the target survey region so as to take maximum advantage of the ocean current velocity.

The second problem we consider is that of generating trajectories for a multi-stage mission with logic constraints. A wide range of single-vehicle missions of practical interest can be expressed as a sequence of tasks to be completed by the vehicle in a predetermined sequence, where a trajectory must be designed for each step. For instance, one could consider a seafloor mapping mission where a vehicle is deployed from a harbor, has to reach a given survey area, execute the survey and return to the harbor.

1.3 Approach and contributions

The problem of generating robot trajectories is well studied for robots in passive environments, with a number of well-established methods which can be categorized roughly into discrete methods, sampling-based methods and combinatorial methods [33].

In their most basic incarnations, sampling-based and combinatorial methods assume that the kinematics are unconstrained, and thus begin by planning a geometric path through the configuration space of the robot, typically with obstacle avoidance in mind. This high-level path is then adjusted to satisfy any differential constraints and used to derive control references for the dynamics controllers. This places the emphasis on planning the unrestricted configuration space path, which may be appropriate when the differential constraints are not too harsh (e.g., bounded velocity and curvature), but not for underwater vehicles navigating in strong dynamic current fields. Additionally, these methods focus on generating feasible paths, not including in their formulation any notion of optimality. In our case, it seems appropriate to use some form of optimization to select the trajectories which can take the most advantage of the ocean currents.

Discrete methods, on the other hand, are by their nature amenable to optimization-based formulations. However, there is the issue of discretizing the robot motion, which becomes more difficult due to the presence of the ocean currents. It is also well known that there are inherent problems in approximating continuous problems in a discrete state space: in some cases, the discrete solution does not approach the optimal solution even as the discretization step goes to zero [51].

Control-theoretic approaches seem to be the most appropriate for addressing continuous-state problems where the differential constraints play a significant role [33]. The use of control-theoretic models such as differential equations and inclusions has a distinct advantage: there are well established formulations for dynamic optimization problems involving such models, known as optimal control problems. Besides being optimization-based formulations, optimal control formulations have several benefits. First, the problem is expressed using the 'natural' continuous state space and dynamics of the robot, so matters of discretization or approximation are decoupled from the formulation. Second, these formulations allow for the inclusion of multiple types of constraints, both on the geometric path of the robot and on the values of the control inputs. Third, the solution is given as the sequence of values taken by the control inputs, so this eliminates having to solve additional problems to map the state trajectory to control inputs. Finally, there is a well-established body of literature on both the theoretical aspects of the formulations, such as existence and characterization of solutions, and on the numerical computation of solutions.

Historically, there have been two main approaches to the solution of optimal control problems: variational methods and dynamic programming [56]. Variational methods are a generalization of the calculus of variations and the method of Lagrange multipliers, and their application typically gives necessary conditions for local optimality. In dynamic programming, the problem is embedded in a family of optimal control problems parameterized by one of the problem parameters, e.g. an initial or final condition, and the result is a sufficient condition for global optimality in the form of a partial differential equation known as the Hamilton-Jacobi-Bellman equation (HJBE). An important characteristic of dynamic programming methods is that an optimal feedback law can be recovered from the solution of the partial differential equation, since the solutions to all the problems in the family in which the original problem was embedded are obtained at once.

Direct use of optimal control techniques is often neglected in robotics due to computational considerations, and they are often used only to support heuristic methods [33]. In fact, both approaches are prohibitively expensive in general for complex robot models: numerical methods relying on the variational approach typically involve the solution of nonlinear equations, two-point boundary value problems or nonconvex optimization problems, while dynamic programming methods require the solution of a nonlinear partial differential equation, which is expensive even in low-dimensional state spaces ($\mathbf{R}^n$ with $n > 4$). However, the most relevant effects of the ocean currents on the motion of ocean vehicles are captured by kinematic models, and it is expectable that a kinematic trajectory which takes into account the differential constraints should give rise to feasible velocity references for the lower-level control loops. Hence, low dimensional kinematic models should be enough, and in that case the direct application of optimal control techniques becomes feasible.

Even though vehicles may be equipped with sensors able to obtain local measurements of the ocean current velocity, its effect on the vehicle's motion is a global issue, and decisions cannot be made at a local level: what can seem like a good decision based on local information may lead the vehicle to areas with unfavorable currents. Additionally, unless the mission time frame is very small, the time-varying nature of the ocean currents cannot be ignored. This aggravates the issue, as even if one is accounting for the variation of the ocean currents over the whole operational area, it all depends on the time at which the vehicle goes through any particular region. Thus, a complete picture of the ocean flow on the operational area over the mission time frame is necessary, and this can only be fulfilled by using forecasts from ocean models. This means that the downside of dynamic programming, which is the computationally expensive offline step of solving the HJBE, is irrelevant.

Our approach uses dynamic programming to solve the two problems outlined above. Dynamic programming is easily applied to the first class of problems. Besides its use in trajectory generation, the solution of the HJBE is useful for mission planning, as it can be used to compare deployment times and positions. The second type of problem that we consider can be naturally expressed in the framework of hybrid systems, which are systems whose dynamics contain both discrete and continuous components. Since dynamic programming also applies to this class of dynamical systems, the dynamic programming approach is extensible to multi-stage problems with logic-based constraints. In this framework the problem is reduced to solving a sequence of partial differential equations, one for each stage.

In what concerns the base problem, compared to existing approaches in the literature our method simultaneously obtains globally optimal trajectories, allows for comparison of deployment times and positions, and allows for general cost functions depending on the vehicle's position. Our approach to planning with logic-based constraints extends previous work in robotic path planning [4].

Over the past two decades, a wide variety of numerical methods for solving Hamilton-Jacobi-Bellman equations arising from a variety of optimal control problems have been developed. Some attention has also been given to the development of parallelizable versions of these methods. This enables the solution of large and high-dimensional problems in high performance computing platforms, which typically have compute nodes with multiple many-core processors, and on modern multi-core desktop and laptop computers. We implemented a multithreaded version of such a numerical method in C++, which we adapted specifically to our class of problems, in order to efficiently solve the partial differential equations arising from the two considered classes of problems. To our knowledge this is the only publicly available parallel implementation of a numerical solver for general Hamilton-Jacobi equations.

1.4 Publications

The following conference articles were published as a result of the work developed in this thesis:

- Miguel Aguiar et al. "Trajectory Optimization for Underwater Vehicles in Time-Varying Ocean Flows". In: *2018 IEEE/OES Autonomous Underwater Vehicle Workshop (AUV)*. IEEE, Nov. 2018. DOI: 10.1109/auv.2018.8729777

- M. Aguiar, J. Estrela da Silva, and J. Borges de Sousa. “Trajectory optimization for marine vehicles: models and numerical methods”. In: *SYMCOMP2019 - 4th International Conference on Numerical and Symbolic Computation*. Porto, 2019

Additionally, the work was presented at the following conferences:

- Symposium on Oceanographic Data Analytics – Poster presentation. NTNU, Trondheim, Norway, November 2018.
- Portuguese Meeting on Oceanography 2019 – Oral presentation. Peniche, Portugal, May 2019.
- Portuguese Meeting on Optimal Control 2019 – Oral presentation. FEUP, Porto, Portugal, June 2019.

1.5 Document Structure

The rest of the document is structured as follows. Chapter 2 contains background material on ocean modeling, hybrid systems and dynamic programming, as well as an overview of the LSTS software toolchain, which is used for software-in-the-loop simulations. Chapter 3 describes the two classes of problems considered in the thesis, formulating them as optimization problems. A review of recent work in trajectory generation for ocean vehicles is done in Chapter 4. In Chapter 5, the theoretical underpinnings of the approach are described, as well as the implementation of the multithreaded numerical solver used to solve the HJBE. Numerical examples using data from real ocean models are presented in Chapter 6. Finally, Chapter 7 presents the conclusions and future research directions.

Chapter 2

Background

2.1 Modeling ocean currents

2.1.1 Hydrodynamic Models

Although hydrodynamic modeling in itself is not the purpose of the dissertation, the proposed approach requires the availability of an ocean current forecast over the operational area and time window. Consequently, some familiarity with the output of these models is necessary.

Hydrodynamic models integrate the Navier-Stokes equations on a rectangular or curvilinear grid over the region of interest. We are interested in the quantities

$$u(x,y,z,t),\ v(x,y,z,t),\ w(x,y,z,t),$$

the velocities in the x, y and z directions, respectively. The horizontal position (x,y) is typically given in spherical coordinates (latitude and longitude), and velocity values are specified at the center of each grid cell. The z coordinate can be specified either as a Cartesian coordinate or in the σ-coordinate system, which consists of several layers bounded by a plane following the free surface and another plane following the bottom topography. For two dimensional models, the horizontal velocities are typically depth-averaged [49].

These models can integrate tidal forcing, sea surface elevation boundary conditions, surface boundary conditions imposed from weather forecasts, heat transport and freshwater inputs. For examples and details on model calibration and validation, see e.g. [2, 49, 50, 60].

In what concerns velocity modeling errors, [49] reports root mean square errors mostly between 0.1 and 0.3 m/s in each velocity component, although in some areas the error is between 0.4 and 0.5 m/s (for typical velocity amplitudes of 1 m/s).

2.1.2 Lagrangian Coherent Structures

If in some region $D \subset \mathbf{R}^2$ the ocean flow velocity is given by the vector field $\left(v^1, v^2\right)$, one can interpret the phase plane trajectories of

$$\begin{aligned} \dot{x}^1(t) &= v^1(x^1(t), x^2(t), t) \\ \dot{x}^2(t) &= v^2(x^1(t), x^2(t), t) \end{aligned} \tag{2.1}$$

for $\boldsymbol{x} = (x^1, x^2) \in D$ as the physical trajectories of a particle (known in this context as a *passive tracer*) being advected by the flow.

Traditional (asymptotic) dynamical system concepts such as fixed points, limit cycles or stable and unstable manifolds are not of great use in the practical situation where the flow velocity is time-variant and known only over a finite interval of time $[t_0, t_1]$ [24].

In this situation, the notion of a *Lagrangian Coherent Structure* is the correct tool for extracting information about qualitative behaviors of geophysical flows from data of the form (2.1). For a planar flow, Lagrangian Coherent Structures (LCSs) can be thought of as moving lines which separate the phase space into regions with dynamically distinct behaviors.

Trajectories of (2.1) typically exhibit high sensitivity to initial conditions, and modeling errors can accumulate when integrating (2.1). Hence, advection of individual particles may not be appropriate for model comparison, for instance. Lagrangian Coherent Structures, on the other hand, are robust to modelling errors [23] and delineate features of the 'global' fluid motion and not just of single trajectories [24].

Multiple mathematical definitions of LCSs have been proposed. Here we outline the definition based on Finite Time Lyapunov Exponents (FTLE) [55].

The *flow map* $f_\tau^t : D \to D$ is defined by

$$f_\tau^t \boldsymbol{x}_0 = \boldsymbol{x}(t)$$

where $\boldsymbol{x}(t)$ is the trajectory of (2.1) with initial condition $\boldsymbol{x}(\tau) = \boldsymbol{x}_0$. Take a trajectory $\boldsymbol{x}(t)$ with initial condition $\boldsymbol{x}(t_0) = \boldsymbol{x}_0$. If the initial condition is perturbed, i.e., we consider a trajectory $\tilde{\boldsymbol{x}}(t)$ with initial condition $\tilde{\boldsymbol{x}}(t_0) = \boldsymbol{x}_0 + \boldsymbol{y}_0$ and expand $\tilde{\boldsymbol{x}}$ as

$$\tilde{\boldsymbol{x}}(t) = \boldsymbol{x}(t) + \boldsymbol{y}(t) + o(|\boldsymbol{y}_0|)$$

then it is a standard result [5] that

$$\boldsymbol{y}(t) = \nabla_{\boldsymbol{x}} f_{t_0}^t \boldsymbol{y}_0$$

(where the gradient is calculated along the trajectory $\boldsymbol{x}(t)$). Hence the magnitude squared of the deviation from the original trajectory is

$$|\boldsymbol{y}(t)|^2 = \left\langle \nabla_{\boldsymbol{x}} f_{t_0}^t \boldsymbol{y}_0, \nabla_{\boldsymbol{x}} f_{t_0}^t \boldsymbol{y}_0 \right\rangle = \boldsymbol{y}_0^{\mathsf{T}} \left(\nabla_{\boldsymbol{x}} f_{t_0}^t\right)^{\mathsf{T}} \nabla_{\boldsymbol{x}} f_{t_0}^t \boldsymbol{y}_0 = y_{01}^2 \lambda_1 + y_{02}^2 \lambda_2$$

where $0 < \lambda_1 \leq \lambda_2$ are the eigenvalues of $\boldsymbol{C}(\boldsymbol{x}_0) = \left(\nabla_{\boldsymbol{x}} f_{t_0}^t\right)^\mathsf{T} \nabla_{\boldsymbol{x}} f_{t_0}^t$ and y_{0i} are the coefficients of the expansion of $\boldsymbol{y}_0$ as a linear combination of the eigenvectors of $\boldsymbol{C}(\boldsymbol{x}_0)$.

Therefore, the deviation is maximized when $\boldsymbol{y}_0$ is aligned with the eigenvector corresponding to λ_2. The FTLE is defined as

$$\Lambda_{t_0}^t(\boldsymbol{x}_0) = \frac{1}{t - t_0} \ln \sqrt{\lambda_2} = \frac{1}{t - t_0} \ln \sqrt{\lambda_{\max}(\boldsymbol{C}(\boldsymbol{x}_0))}$$

so that

$$|\boldsymbol{y}(t)| = |\boldsymbol{y}_0| \exp\left((t - t_0)\Lambda_{t_0}^t(\boldsymbol{x}_0)\right).$$

We can see $\Lambda_{t_0}^t$ as a scalar field giving at each point the maximum logarithmic stretching in the flow over the interval $[t_0, t]$. To measure the contraction, the same method can be used, only integrating backwards in time.

The position of a LCS at time t_0 is then defined as a *ridge* in the graph of $\Lambda_{t_0}^t$, that is, a curve in D along which the change in $\Lambda_{t_0}^t$ is smaller than the change along the direction transverse to the curve. This admits multiple mathematical definitions [55], but the intuition is essentially the same for all of them.

In Figure 2.1 an example of the determination of LCS from the FTLE field is shown. The ocean flow data is obtained from Very High Frequency Radar measurements. The arrows indicate the magnitude and direction of the horizontal ocean current velocity. Land points are represented in green. Regions with low values of the FTLE are indicated in blue, and high values are shown in red. The LCS is highlighted in green over the region with high FTLE values.

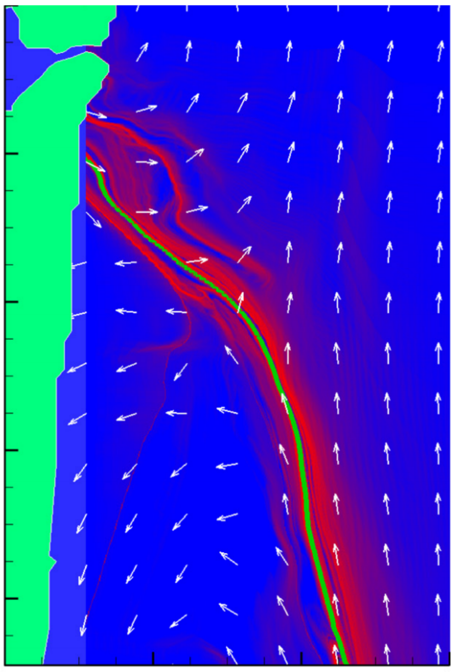


Figure 2.1: LCS off the coast of Florida (Adapted from Shadden et al. [55])

For our purposes, LCSs are interesting because these structures have been found to be linked to optimal trajectories of marine vehicles [26, 71]. Specifically, time- and energy-optimal trajectories between points on a LCS have been shown experimentally to stay on the LCS. Space mission design has benefited extensively from ideas from dynamical systems theory [30], so it makes sense that the same will happen to ocean mission design for autonomous vehicles, in particular in highly dynamical regions of the ocean. In recent experiments, Lagrangian Coherent Structures have been shown to be a valuable tool for planning long distance glider missions [47].

2.2 Hybrid system models

Hybrid systems are dynamical systems which combine aspects of continuous time and state dynamical systems governed by differential equations with discrete-state event-based systems, which are governed by logic-based transition rules.

A variety of models may be considered for hybrid systems, taking into account controlled and autonomous jumps and switching [9]. We follow the hybrid automaton model presented in [25].

The state space is $\mathscr{Z} \times \mathbf{R}^n$, where $\mathscr{Z} \cong \{1, 2, \ldots, n_\zeta\}$ is a finite set. The finite states ζ are sometimes referred to as 'modes'. The control variables consist of a continuous control variable $\boldsymbol{u}$ taking values in a set $\mathscr{U} \subset \mathbf{R}^m$ and a discrete control variable σ taking values in a finite set $\mathscr{S} \cong \{1, 2, \ldots, n_\sigma\}$. The dynamics are then given by

$$\begin{aligned} \dot{\boldsymbol{x}}(t) &= \boldsymbol{f}(\zeta(t), \boldsymbol{x}(t), \boldsymbol{u}(t)) \\ (\zeta(t), \boldsymbol{x}(t)) &= \Phi\big(\zeta(t^-), \boldsymbol{x}(t^-), \sigma(t^-)\big) \end{aligned} \tag{2.2}$$

where $\boldsymbol{f} : \mathscr{Z} \times \mathbf{R}^n \times \mathbf{R}^m \to \mathbf{R}^n$ is the usual dynamics vector field and $\Phi : \mathscr{Z} \times \mathbf{R}^n \times \mathscr{S} \to \mathscr{Z} \times \mathbf{R}^n$ is the discrete transition map. Here $\boldsymbol{x}(t^-) := \lim_{\tau \uparrow t} \boldsymbol{x}(t)$ and similarly for ζ and σ.

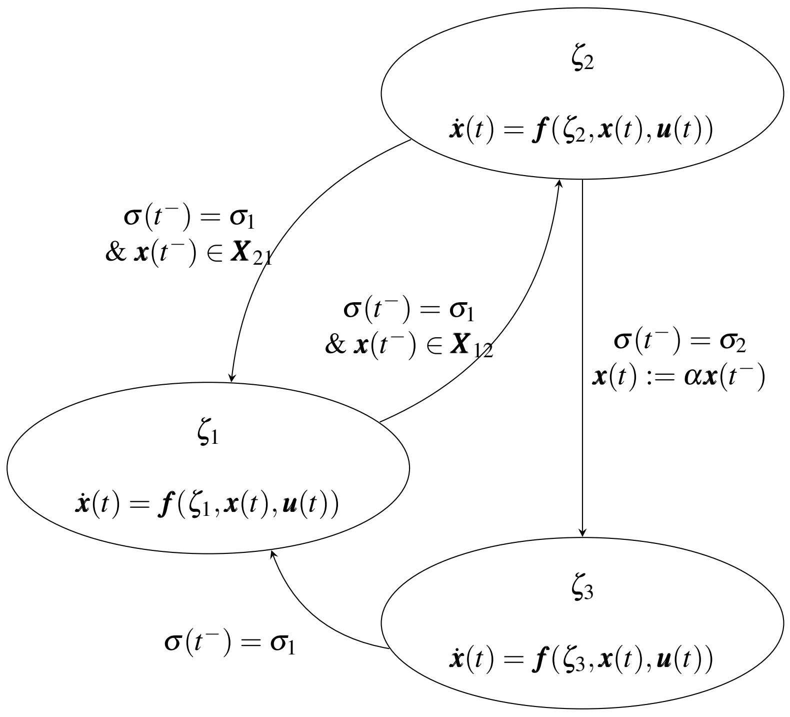
This can be seen as a finite state automaton with state ζ which 'runs' a different control system in each state [34], where the guard conditions on the transitions can depend on the continuous state as well as on the discrete input to the automaton. This is illustrated in Figure 2.2 for $n_\zeta = 3$ and $n_\sigma = 2$. The transition map Φ is specified by the arrows in the state diagram. The model allows for instantaneous changes in the state when a transition occurs, as in the transition between modes ζ_2 and ζ_3 in Figure 2.2.

A solution of (2.2) is defined as a pair of functions

$$\boldsymbol{\xi} : [0, \infty) \to \mathbf{R}^n,\ z : [0, \infty) \to \mathscr{Z},$$

both right continuous such that on any interval (t_1, t_2) on which ζ is constant and $\boldsymbol{x}$ is absolutely continuous the continuous dynamics are satisfied in the usual sense (as detailed in Section 2.3.2.2):

$$\boldsymbol{\xi}(t) = \boldsymbol{\xi}(t_1) + \int_{t_1}^{t} \boldsymbol{f}(z(t_1), \boldsymbol{\xi}(\tau), \boldsymbol{u}(\tau)) \mathrm{d}\tau$$



ζ_2
$\dot{\boldsymbol{x}}(t) = \boldsymbol{f}(\zeta_2, \boldsymbol{x}(t), \boldsymbol{u}(t))$
$\sigma(t^-) = \sigma_1$ & $\boldsymbol{x}(t^-) \in \boldsymbol{X}_{21}$
$\sigma(t^-) = \sigma_1$ & $\boldsymbol{x}(t^-) \in \boldsymbol{X}_{12}$
$\sigma(t^-) = \sigma_2$ $\boldsymbol{x}(t) := \alpha \boldsymbol{x}(t^-)$
ζ_1
$\dot{\boldsymbol{x}}(t) = \boldsymbol{f}(\zeta_1, \boldsymbol{x}(t), \boldsymbol{u}(t))$
ζ_3
$\sigma(t^-) = \sigma_1$
$\dot{\boldsymbol{x}}(t) = \boldsymbol{f}(\zeta_3, \boldsymbol{x}(t), \boldsymbol{u}(t))$

Figure 2.2: A hybrid automaton with three discrete states

and the discrete dynamics are satisfied everywhere:

$$(z(t), \boldsymbol{\xi}(t)) = \Phi\big(z(t^-), \boldsymbol{\xi}(t^-), \sigma(t^-)\big)$$

for all $t \geq 0$.

2.3 Dynamic Programming

2.3.1 Introduction

Dynamic programming is a general approach for the solution of optimization problems involving multi-stage decision processes. These are problems where one is interested in controlling the state of a process, represented by a variable $\boldsymbol{x}$. The temporal evolution of the state is influenced by both the previous values of the state and a decision variable $\boldsymbol{u}$ which we may choose at instants of time $t \in S \subset \mathbf{R}$ (this set may be a collection of discrete instants, or the decisions may be taken in a continuous manner, or any combination of the two). It is also possible to consider non-deterministic or stochastic components affecting the evolution of $\boldsymbol{x}$, but here the focus is on the deterministic case. The problem is then to choose the values of $\boldsymbol{u}$ over the interval of time of interest while minimizing some given performance index which is a real-valued function of the temporal evolution of $\boldsymbol{x}$ and $\boldsymbol{u}$.

A naive approach to solving this problem would be to see the optimization variable as the whole sequence of values of the decision variable over the interval of time of interest, i.e., reducing the problem to a mathematical programming problem. Even for moderately sized problems, this leads to high-dimensional search spaces, and the solution cannot be obtained in an efficient manner. In the case of problems over infinite or unknown time horizons, or even just finite time problems where the state and decision variable range over 'continuous' sets, the resulting mathematical programming problem is infinite-dimensional.

Dynamic programming avoids this complexity by propagating the information between decision stages. Roughly speaking, this enables the reduction of the complexity of the problem to that of a single stage, the price being the ability to store the information that must be shared between stages. In this way it is even possible to reduce some infinite dimensional problems to tractable finite dimensional problems.

The basic tool for finding dynamic programming solutions is the *principle of optimality* typically attributed to Bellman [8]. Suppose we have an optimal sequence of decisions $\{\boldsymbol{u}_t\}_{t\in S'}$, where $S' = \{t \in S \mid t_0 \leq t \leq t_1\}$. At each $t \in [t_0, t_1]$, the variable $\boldsymbol{x}$ takes some value $\boldsymbol{x}_t$ which is influenced by the sequence of decisions. The principle of optimality then states that, for any $\tau \in [t_0, t_1]$, the sequence of decisions $\{\boldsymbol{u}_t\}_{t\in S''}$, where $S'' = \{t \in S \mid \tau \leq t \leq t_1\}$, is optimal for the problem starting from $\boldsymbol{x}_\tau$. This was summarized by Bellman [8] in the following way:

> An optimal policy has the property that whatever the initial state and initial decisions are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision.

An illustrative example is in order. A problem where dynamic programming has found wide application in practice is that of finding optimal paths through a network. Consider the network in Figure 2.3.

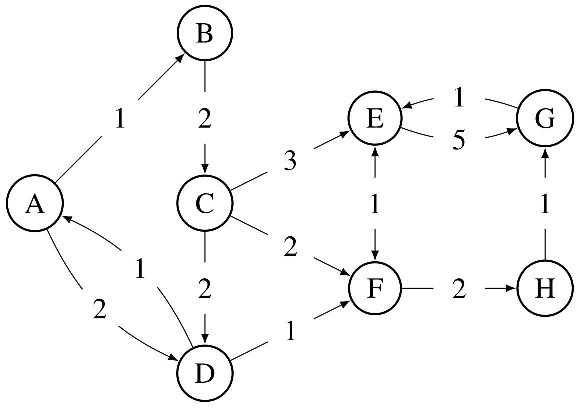


Figure 2.3: A network

The nodes represent locations (i.e., a city or an Internet router), and the links represent ways to move between locations (i.e., a highway or a network link). The labels on the links represent the cost of taking each route. A path is a sequence of nodes, and to each path we can associate a cost given by the sum of the costs of each link. For instance, the path ABCD will have an associated cost of

$$\text{cost(AB)} + \text{cost(BC)} + \text{cost(CD)} = 1 + 2 + 2 = 5.$$

Suppose one wishes to find the cheapest path from node A to node G. Such a path is of the form $AN_1N_2\cdots N_lG$, where each N_i is a node in the network, and the links AN_1, N_iN_{i+1} and N_lG exist. By the principle of optimality, $N_iN_{i+1}\cdots N_lG$ must be an optimal path for each i. Thus, if we know an optimal path to G from every node which has a link from A, we can find an optimal path from A to G by finding the neighbor of A which minimizes the sum of the cost of moving from A to that neighbor and the cost of the optimal path to G from said neighbor.

Hence we can start by finding the cost of an optimal path from each node to G. We call this the *cost-to-go* from each node. Obviously, G has a cost-to-go of 0. We represent this as in Figure 2.4. A node colored in blue indicates that its cost-to-go has been determined, and the cost-to-go is given by the number on the node.

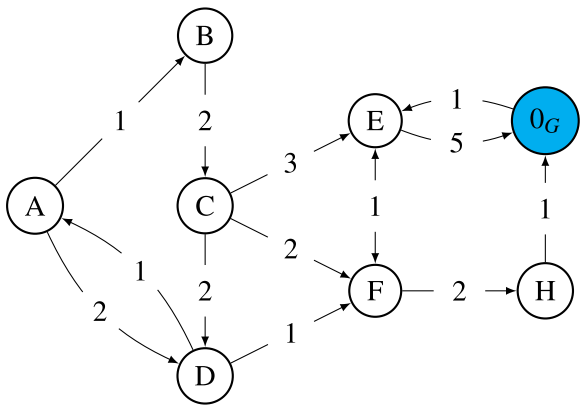


Figure 2.4: Initial condition for the cost-to-go calculation.

The next step would be to obtain the cost-to-go of each node which has a link to G. A first estimate would be the link costs to G, as in Figure 2.5. However, note that the path EFHG has a cost of 4, so E's cost-to-go cannot be 5. For this reason we have colored the nodes in gray to indicate that these are upper bounds and not the final values.

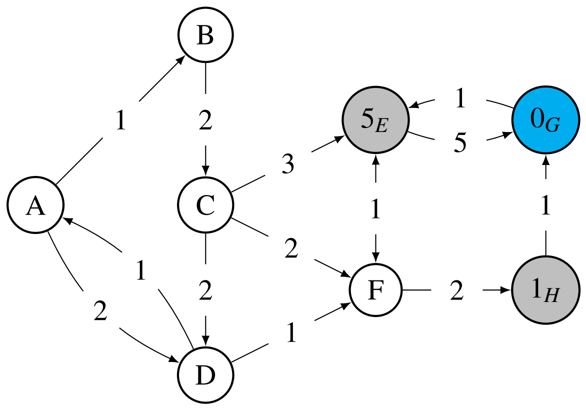


Figure 2.5: Upper-bounding the cost-to-go of nodes E and H

On the other hand, we can see that H's cost-to-go must be 1, for if not, there is another path from H to G with lower cost. But such a path must pass through either E or H to get to G. In the first case, it will contain the sequence EG, which means that its cost is greater than or equal to 5. If it passes through H, it will contain the sequence HG, so its cost will be at least 1. Thus, we can label H in blue.

Since we now know H's cost-to-go, we can estimate F's cost-to-go as the sum of the link cost to H and the cost-to-go from H (Figure 2.6).

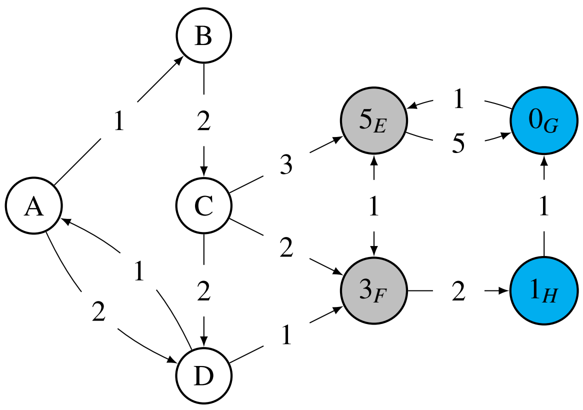


Figure 2.6: Using H's cost-to-go, F's cost-to-go can be upper-bounded

By the same argument we used for H, we can see that F's cost-to-go must be equal to this estimate, since the only other path is through E and such a path would imply a higher cost.

We continue by estimating the cost-to-go of D and C. We can also re-estimate E's cost-to-go. This new estimate must be correct, since all paths from E to G contain either EF or EG.

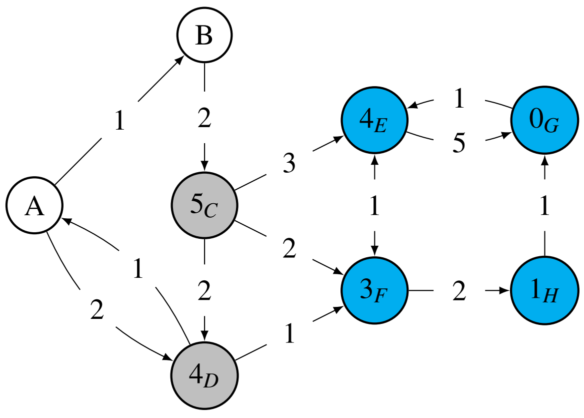


Figure 2.7: The procedure continues, expanding outward from the target node

Note that any path starting from of the nodes colored in white or grey in Figure 2.7 must go through either E or F. In particular, this implies that A cannot have a cost-to-go which is less than 3, and for this reason node D's estimate must be correct. In addition, once we have D's cost-to-go, we see that C's cost-to-go estimate must also be correct, and then we can update nodes A and B.

Node A can go through B or D. A path through B must go through C, and this would result in a cost of at least 5. Hence A's cost-to-go estimate is correct. Finally, B's cost-to-go estimate is correct, since it can only go through C. The result is represented in Figure 2.9.

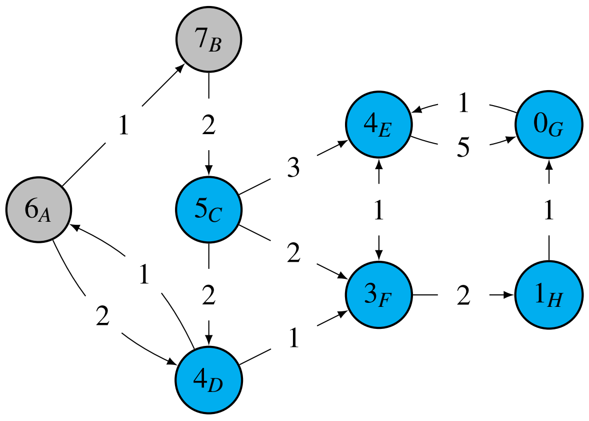


Figure 2.8: The procedure continues, expanding outward from the target node

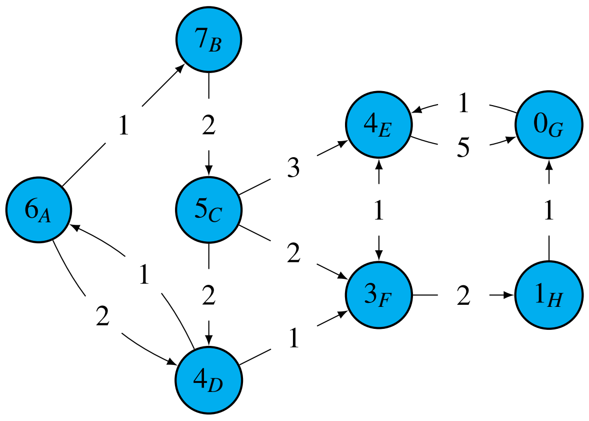


Figure 2.9: The cost-to-go from each node

It is now quite easy to calculate the optimal path between A and G, by choosing at each instant a neighbor with least cost-to-go. The result is ADFHG, which has a cost of 6, equal to A's cost-to-go.

From this single calculation we are also able to obtain optimal paths from any node to G. This procedure can be generalized to arbitrary directed graphs with positive edge weights, resulting in Dijkstra's algorithm [14].

If the edge weights were to change at each step, the procedure would no longer be applicable, since the cost-to-go from each node would depend on the time of arrival at the node. In that case one solution is to consider a bigger graph where each node represents a possible arrival time at each of the nodes of the graph represented in Figure 2.3.

2.3.2 Application of the principle of optimality

We now proceed to give examples of typical applications of dynamic programming to control problems. Only a brief overview of the simplest examples is considered, and the focus is on intuitive arguments. Mathematical details can be found in the literature [6, 8, 21, 56].

2.3.2.1 Discrete time systems

Consider the discrete time dynamical system given by the transition equation

$$\boldsymbol{x}_{t+1} = \boldsymbol{\varphi}(\boldsymbol{x}_t, \boldsymbol{u}_t) \tag{2.3}$$

where t is an integer. The state variable $\boldsymbol{x}_t$ takes values in some set $\mathscr{X}$. The decision or control variable $\boldsymbol{u}_t$ is restricted to some set $\mathscr{U}$.

A standard problem in control is the infinite horizon regulator problem, where the objective is to find a sequence of control values $\boldsymbol{u}_\tau$, $\tau \geq 0$ which minimize the cost function

$$J(\boldsymbol{\xi}, \boldsymbol{u}) = \sum_{t=0}^{\infty} g(\boldsymbol{x}_t(\boldsymbol{\xi}, \boldsymbol{u}), \boldsymbol{u}_t)$$

where $g : \mathscr{X} \times \mathscr{U} \to \mathbf{R}_{\geq 0}$. Note that in the above expression $\boldsymbol{u}$ represents the whole control sequence $\{\boldsymbol{u}_\tau\}_{\tau\geq 0}$ and $\boldsymbol{x}_t(\boldsymbol{\xi}, \boldsymbol{u})$ is the value of the system state at time t when the initial condition is $\boldsymbol{x}_0(\boldsymbol{\xi}, \boldsymbol{u}) = \boldsymbol{\xi}$ and the control sequence is $\boldsymbol{u}$.

Following the example of the previous section, we begin by defining the cost-to-go from each state:

$$\vartheta(\boldsymbol{x}) = \inf_{\boldsymbol{u}\in U} J(\boldsymbol{x}, \boldsymbol{u}),$$

where U is the set of sequences $\boldsymbol{u}$ taking values in $\mathscr{U}$. The function $\vartheta : \mathscr{X} \to [0, +\infty]$ is also known in this context as the *value function*.

Suppose $\boldsymbol{u}^\star$ is an optimal control sequence from the state $\boldsymbol{x}$, i.e.

$$J(\boldsymbol{x}, \boldsymbol{u}^\star) = \vartheta(\boldsymbol{x}).$$

By the principle of optimality, the portion of the control starting at $t = \tau \geq 0$, is an optimal control from the initial condition $\boldsymbol{x}_\tau(\boldsymbol{x}, \boldsymbol{u}^\star)$, i.e.

$$J\big(\boldsymbol{x}_\tau(\boldsymbol{x}, \boldsymbol{u}^\star), \{\boldsymbol{u}_t^\star\}_{t\geq\tau}\big) = \vartheta(\boldsymbol{x}_\tau(\boldsymbol{x}, \boldsymbol{u}^\star)).$$

In particular this is true for $\tau = 1$:

$$J\big(\boldsymbol{\varphi}(\boldsymbol{x}, \boldsymbol{u}_0^\star), \{\boldsymbol{u}_t^\star\}_{t\geq 1}\big) = \vartheta(\boldsymbol{\varphi}(\boldsymbol{x}, \boldsymbol{u}_0^\star))$$

so that

$$\vartheta(\boldsymbol{x}) = g(\boldsymbol{x}, \boldsymbol{u}_0^\star) + \vartheta(\boldsymbol{\varphi}(\boldsymbol{x}, \boldsymbol{u}_0^\star)). \tag{2.4}$$

Consider any $\boldsymbol{w}_0 \in \mathscr{U}$. Let $\boldsymbol{w}^\star$ be an optimal sequence from $\boldsymbol{\varphi}(\boldsymbol{x}, \boldsymbol{w}_0)$ and let $\boldsymbol{u}$ be defined by $\boldsymbol{u}_0 = \boldsymbol{w}_0$ and $\boldsymbol{u}_\tau = \boldsymbol{w}_{\tau-1}^\star$ for $\tau \geq 1$. Then

$$\begin{aligned}
\vartheta(\boldsymbol{x}) &\leq J(\boldsymbol{x}, \boldsymbol{u}) \\
&= g(\boldsymbol{x}, \boldsymbol{w}_0) + J\big(\boldsymbol{\varphi}(\boldsymbol{x}, \boldsymbol{w}_0), \{\boldsymbol{w}_\tau^\star\}_{\tau\geq 1}\big) \\
&= g(\boldsymbol{x}, \boldsymbol{w}_0) + \vartheta(\boldsymbol{\varphi}(\boldsymbol{x}, \boldsymbol{w}_0))
\end{aligned}$$

Together with (2.4), this implies that

$$\vartheta(\boldsymbol{x}) = \min_{\boldsymbol{\eta}\in\mathscr{U}} \{g(\boldsymbol{x}, \boldsymbol{\eta}) + \vartheta(\boldsymbol{\varphi}(\boldsymbol{x}, \boldsymbol{\eta}))\}. \tag{2.5}$$

This formula is known as the Bellman equation, and it provides a way of calculating the cost-to-go from each state. Note that $\boldsymbol{u}_0^\star$ is a minimizer in (2.5), so that once ϑ has been found one can also recover the optimal control sequence.

The classical discrete time Linear Quadratic Regulator problem can be solved using this method [27].

2.3.2.2 Continuous time systems

Consider the dynamical system given by the system of ordinary differential equations

$$\dot{\boldsymbol{x}}(t) = \boldsymbol{f}(\boldsymbol{x}(t), \boldsymbol{u}(t)) \tag{2.6}$$

where $t \in \mathbf{R}$, $\boldsymbol{x} \in \mathbf{R}^n$ is the state variable and $\boldsymbol{u} \in \mathscr{U} \subset \mathbf{R}^m$ is the control variable.

Assume that $\boldsymbol{f}$ satisfies a standard set of assumptions [6] which guarantee that for each measurable function $\boldsymbol{u} : \mathbf{R}_{\geq 0} \to \mathbf{R}^m$ and initial state $\boldsymbol{x}_0 \in \mathbf{R}^n$ there exists a unique absolutely continuous solution of (2.6) defined for all $t \in \mathbf{R}_{\geq 0}$, which we denote by $\boldsymbol{\xi}(\cdot; \boldsymbol{x}_0, \boldsymbol{u}) : \mathbf{R}_{\geq 0} \to \mathbf{R}^n$, that is, we have

$$\begin{aligned}
\boldsymbol{\xi}(0; \boldsymbol{x}_0, \boldsymbol{u}) &= \boldsymbol{x}_0 \\
\left.\frac{\mathrm{d}}{\mathrm{d}t}\right|_{t=\tau} \boldsymbol{\xi}(t; \boldsymbol{x}_0, \boldsymbol{u}) &= \boldsymbol{f}(\boldsymbol{\xi}(\tau; \boldsymbol{x}_0, \boldsymbol{u}), \boldsymbol{u}(\tau))
\end{aligned}$$

Consider the problem of controlling the state $\boldsymbol{x}$ to some closed set $\Omega \subset \mathbf{R}^n$. We begin by defining

$$T(\boldsymbol{x},\boldsymbol{u}) = \inf\{t \geq 0 \mid \boldsymbol{\xi}(t;\boldsymbol{x},\boldsymbol{u}) \in \Omega\},$$

i.e., T is the time of first arrival at Ω. Then we can define a cost function on the trajectories:

$$J(\boldsymbol{x},\boldsymbol{u}) = q(\boldsymbol{\xi}(T;\boldsymbol{x},\boldsymbol{u})) + \int_0^T g(\boldsymbol{\xi}(t;\boldsymbol{x},\boldsymbol{u}),\boldsymbol{u}(t))\mathrm{d}t$$

where $q : \partial\Omega \to \mathbf{R}_{\geq 0}$ and $g : (\mathbf{R}^n \setminus \Omega) \times \mathscr{U} \to \mathbf{R}_{>0}$ are the *arrival cost* and *running cost*, respectively. A particularly interesting case is $q \equiv 0$ and $g \equiv 1$, in which case the cost function is the time taken to reach Ω, since $J(\boldsymbol{x},\boldsymbol{u}) = T(\boldsymbol{x},\boldsymbol{u})$.

We can now formulate the problem as that of minimizing J. As before, the cost-to-go from $\boldsymbol{x}$ is defined as

$$\vartheta(\boldsymbol{x}) = \inf_{\boldsymbol{u}\in U} J(\boldsymbol{x},\boldsymbol{u})$$

where U is the set of measurable functions $\boldsymbol{u} : \mathbf{R}_{\geq 0} \to \mathscr{U}$.

Consider a state $\boldsymbol{x}$ and a corresponding optimal control $\boldsymbol{u}$, that is

$$\vartheta(\boldsymbol{x}) = J(\boldsymbol{x},\boldsymbol{u}).$$

For any $0 \leq s \leq T$, let $\boldsymbol{u}^s$ denote a shift of $\boldsymbol{u}$ by s units of time, i.e.

$$\boldsymbol{u}^s(t) = \boldsymbol{u}(t+s). \tag{2.7}$$

Then

$$\begin{aligned}
\vartheta(\boldsymbol{x}) &= \int_0^s g(\boldsymbol{\xi}(t;\boldsymbol{x},\boldsymbol{u}),\boldsymbol{u}(t))\mathrm{d}t + q(\boldsymbol{\xi}(T;\boldsymbol{x},\boldsymbol{u})) + \int_s^T g(\boldsymbol{\xi}(t;\boldsymbol{x},\boldsymbol{u}),\boldsymbol{u}(t))\mathrm{d}t \\
&= \int_0^s g(\boldsymbol{\xi}(t;\boldsymbol{x},\boldsymbol{u}),\boldsymbol{u}(t))\mathrm{d}t + q(\boldsymbol{\xi}(T-s;\boldsymbol{\xi}(s;\boldsymbol{x},\boldsymbol{u}),\boldsymbol{u}^s)) \\
&\qquad + \int_0^{T-s} g(\boldsymbol{\xi}(t+s;\boldsymbol{x},\boldsymbol{u}),\boldsymbol{u}(t+s))\mathrm{d}t \\
&= \int_0^s g(\boldsymbol{\xi}(t;\boldsymbol{x},\boldsymbol{u}),\boldsymbol{u}(t))\mathrm{d}t + q(\boldsymbol{\xi}(T-s;\boldsymbol{\xi}(s;\boldsymbol{x},\boldsymbol{u}),\boldsymbol{u}^s)) \\
&\qquad + \int_0^{T-s} g(\boldsymbol{\xi}(t;\boldsymbol{\xi}(s;\boldsymbol{x},\boldsymbol{u}),\boldsymbol{u}^s),\boldsymbol{u}^s(t))\mathrm{d}t \\
&= \int_0^s g(\boldsymbol{\xi}(t;\boldsymbol{x},\boldsymbol{u}),\boldsymbol{u}(t))\mathrm{d}t + J(\boldsymbol{\xi}(s;\boldsymbol{x},\boldsymbol{u}),\boldsymbol{u}^s) \\
&= \int_0^s g(\boldsymbol{\xi}(t;\boldsymbol{x},\boldsymbol{u}),\boldsymbol{u}(t))\mathrm{d}t + \vartheta(\boldsymbol{\xi}(s;\boldsymbol{x},\boldsymbol{u}))
\end{aligned}$$

The last equality follows from the principle of optimality, as $\boldsymbol{u}^s$ must be optimal for $\boldsymbol{\xi}(s;\boldsymbol{x},\boldsymbol{u})$.

Now let $\boldsymbol{u}_1 : [0,s] \to \mathscr{U}$ be any measurable function and let $\boldsymbol{u}_2 \in U$ be an optimal control for

$\boldsymbol{\xi}(s;\boldsymbol{x},\boldsymbol{u}_1)$. Then, letting $\boldsymbol{u}$ be the concatenation of $\boldsymbol{u}_1$ and $\boldsymbol{u}_2$, by a similar computation we find

$$\begin{aligned}\vartheta(\boldsymbol{x}) &\leq J(\boldsymbol{x},\boldsymbol{u}) \\ &= \int_0^s g(\boldsymbol{\xi}(t;\boldsymbol{x},\boldsymbol{u}_1),\boldsymbol{u}_1(t))\mathrm{d}t + J(\boldsymbol{\xi}(s;\boldsymbol{x};\boldsymbol{u}_1),\boldsymbol{u}_2) \\ &= \int_0^s g(\boldsymbol{\xi}(t;\boldsymbol{x},\boldsymbol{u}_1),\boldsymbol{u}_1(t))\mathrm{d}t + \vartheta(\boldsymbol{\xi}(s;\boldsymbol{x};\boldsymbol{u}_1))\end{aligned}$$

So that

$$\vartheta(\boldsymbol{x}) = \inf_{\boldsymbol{u}_1}\left\{\int_0^s g(\boldsymbol{\xi}(t;\boldsymbol{x},\boldsymbol{u}_1),\boldsymbol{u}_1(t))\mathrm{d}t + \vartheta(\boldsymbol{\xi}(s;\boldsymbol{x};\boldsymbol{u}_1))\right\}, \tag{2.8}$$

a counterpart to the discrete-time Bellman equation we found before.

Assuming ϑ is differentiable at $\boldsymbol{x}$, we can obtain a local version of this equation. First, rewrite this as

$$0 = \inf_{\boldsymbol{u}_1}\{g(\boldsymbol{x},\boldsymbol{u}_1(0))s + o(s) + \vartheta(\boldsymbol{\xi}(s;\boldsymbol{x};\boldsymbol{u}_1)) - \vartheta(\boldsymbol{x})\}$$

dividing by s and taking a limit $s \to 0$, this becomes

$$0 = \inf_{\boldsymbol{w}}\{g(\boldsymbol{x},\boldsymbol{w}) + \nabla\vartheta(\boldsymbol{x})\cdot\boldsymbol{f}(\boldsymbol{x},\boldsymbol{w})\} \tag{2.9}$$

This gives a partial differential equation (PDE) for the value function (known as the *Hamilton-Jacobi-Bellman PDE*, with the boundary condition

$$\vartheta(\boldsymbol{x}) = q(\boldsymbol{x}),\ \boldsymbol{x} \in \partial\Omega$$

The issue of differentiability of ϑ is a delicate one, and so are the questions of existence of uniqueness of a solution to the boundary value problem. The theory of viscosity solutions was developed to tackle these problems, and the results show that for a wide variety of problems and under quite general conditions the value function is the unique viscosity solution of the boundary value problem [6]. For the special case of a minimum-time problem, under some controllability assumptions it can also be shown that ϑ is locally Lipschitz, which implies it is differentiable almost everywhere [6].

If we can compute a map $\kappa : \mathbf{R}^n \to \mathscr{U}$ which satisfies

$$\kappa(\boldsymbol{x}) \in \arg\min_{\boldsymbol{w}}\{g(\boldsymbol{x},\boldsymbol{w}) + \nabla\vartheta(\boldsymbol{x})\cdot\boldsymbol{f}(\boldsymbol{x},\boldsymbol{w})\}$$

then κ is a feedback law which drives the system to Ω from any state.

Other types of problems can be considered, such as infinite horizon regulator-type problems or finite-horizon problems (in which case the value function has a time dependence).

For problems with time-dependent dynamics, i.e., problems involving dynamical systems of the form

$$\dot{\boldsymbol{x}}(t) = \boldsymbol{f}(t,\boldsymbol{x}(t),\boldsymbol{u}(t)),$$

the dynamic programming principle cannot be applied directly. This is due to the fact that the cost-to-go at $\boldsymbol{x}$ depends on the time at which the trajectory passes through $\boldsymbol{x}$. As long as $\boldsymbol{f}$ is sufficiently regular in t, we can view t as a state variable. Setting $\boldsymbol{z} = (\boldsymbol{x}, \tau)$ and $\boldsymbol{g}(\boldsymbol{z}, \boldsymbol{u}) = (\boldsymbol{f}(\tau, \boldsymbol{x}, \boldsymbol{u}), 1)$ we then have a dynamical system

$$\dot{\boldsymbol{z}}(t) = \boldsymbol{g}(\boldsymbol{z}(t), \boldsymbol{u}(t)),$$

which is of the same form as (2.6), so dynamic programming may be applied to this model.

A different kind of model may be adopted for continuous-time dynamical systems, using the concept of a differential inclusion. If we consider the set-valued function F defined by

$$F(\boldsymbol{x}) = f(\boldsymbol{x}, \mathscr{U}),$$

then (2.6) can be written as

$$\dot{\boldsymbol{x}}(t) \in F(\boldsymbol{x}(t)),$$

i.e., the selection of a control value is interpreted as the selection of a velocity at each point of a trajectory. If the running cost g does not depend on $\boldsymbol{u}$, the PDE (2.9) can then be written as

$$-g(\boldsymbol{x}) = \inf_{\boldsymbol{v} \in F(\boldsymbol{x})} \nabla \vartheta(\boldsymbol{x}) \cdot \boldsymbol{v}$$

2.3.2.3 Hybrid systems

Let us now consider how dynamic programming can be applied to models of the form described in Section 2.2. The control problem under consideration is again the minimum-cost-to-target problem, with the addition of a switching cost. Here we can consider a target Ω defined in $\mathscr{Z} \times \mathbf{R}^n$. Each projection of the target, i.e., the sets

$$\Omega_\zeta := \{\boldsymbol{x} \in \mathbf{R}^n \mid (\zeta, \boldsymbol{x}) \in \Omega\}$$

should be closed.

Then, defining as before the time of arrival

$$T = \inf\{t \geq 0 \mid (z(t), \boldsymbol{\xi}(t)) \in \Omega\}$$

the cost associated to a trajectory with $\boldsymbol{\xi}(0) = \boldsymbol{x}$ and $z(0) = \zeta$ is defined as

$$\begin{aligned} J(\boldsymbol{x}, \zeta, \boldsymbol{u}, \sigma) = q(z(T), \boldsymbol{\xi}(T)) &+ \int_0^T g(z(t), \boldsymbol{\xi}(t), \boldsymbol{u}(t)) \mathrm{d}t \\ &+ \sum_{k=1}^N c(\boldsymbol{\xi}(t_k), z(t_k), \boldsymbol{\xi}(t_k^-), z(t_k^-), \sigma(t_k^-)) \end{aligned}$$

where the switching cost c is assumed to satisfy $c(\cdot, \zeta_i, \cdot, \zeta_j, \cdot) > 0$ if $i \neq j$, and $\{t_1, \dots, t_N\}$ are the times at which the discrete state switches. The positive switching cost ensures only a finite number of switching times exist.

For simplicity, consider the case where $n_\zeta = n_\sigma = 2$ and the discrete dynamics are simply

$$\zeta(t) = \sigma(t^-)$$

so that no jumps in the continuous state can occur and

$$c(\boldsymbol{\xi}(t), z(t), \boldsymbol{\xi}(t^-), z(t^-), \sigma(t^-)) = c(\boldsymbol{\xi}(t), z(t), z(t^-)).$$

The value function must also include the discrete state. Hence, we define $\vartheta_i,\ i = 1,2$ as the cost-to-go from $\boldsymbol{x}$ if the discrete state is initially ζ_i. The application of the principle of optimality from the state $(\boldsymbol{x}, \zeta_1)$ then leads to

$$\vartheta_1(\boldsymbol{x}) \le \min \left\{ \inf_{\boldsymbol{u}} \left\{ \int_0^s g(\zeta_1, \boldsymbol{\xi}(t;\boldsymbol{x},\boldsymbol{u}), \boldsymbol{u}(t))\mathrm{d}t + \vartheta_1(\boldsymbol{\xi}(s;\boldsymbol{x},\boldsymbol{u})) \right\}, \right.$$
$$\left. c(\boldsymbol{\xi}(s;\boldsymbol{x},\boldsymbol{u}), \zeta_2, \zeta_1) + \vartheta_2(\boldsymbol{\xi}(s;\boldsymbol{x},\boldsymbol{u})) \right\}$$

since the possibilities are remaining in the first discrete state over $[0,s]$, switching at time s or switching before time s, in which case the cost must be smaller than the cost of the first or second possibilities.

Taking a limit as $s \to 0$ yields

$$0 = \min \left\{ \inf_{\boldsymbol{v}} \{ g_1(\boldsymbol{x},\boldsymbol{v}) + \nabla\vartheta_1(\boldsymbol{x}) \cdot \boldsymbol{f}_1(\boldsymbol{x},\boldsymbol{v}) \},\ c(\boldsymbol{x}, \zeta_2, \zeta_1) + \vartheta_2(\boldsymbol{x}) - \vartheta_1(\boldsymbol{x}) \right\} \tag{2.10}$$

Similar equations hold for ϑ_2.

A problem of this type is considered in [53], where a person must reach some point in minimum time, either by walking or skating. Walking or skating correspond in that case to the two discrete states, and there is a time penalty for putting on or taking off the skates, which corresponds to the switching cost.

2.3.3 Issues with the application of the principle of optimality

In some problems it can be nontrivial to apply the principle of optimality directly. For instance, in a problem with fuel constraints, i.e., a constraint of the form

$$\int_0^\infty |\boldsymbol{u}(t)|\mathrm{d}t \le \alpha,$$

Equation (2.8) is ‘incomplete’ since it might be the case that the concatenation of the two controls does not satisfy the fuel constraint even if the individual controls do.

Thus, for the application of the principle of optimality the set of admissible controls should satisfy some conditions, namely [6]:

1. If $\boldsymbol{u}$ is an admissible control, so is $\boldsymbol{u}^s$ for all $s > 0$, defined by

$$\boldsymbol{u}^s(t) = \boldsymbol{u}(t+s)$$

2. If $\boldsymbol{u}_1$ and $\boldsymbol{u}_2$ are admissible controls, so is $\boldsymbol{u}$ defined as the concatenation of $\boldsymbol{u}_1$ up to time $\tau > 0$ with $\boldsymbol{u}_2$, i.e.

$$\boldsymbol{u}(s) = \begin{cases} \boldsymbol{u}_1(t) & 0 \le t \le \tau \\ \boldsymbol{u}_2(t-\tau) & \tau < t \end{cases}$$

Additionally, there are conditions on the cost functional γ, namely

$$\gamma(\boldsymbol{x}(t_i[.]t_f)) = \beta(\boldsymbol{x}(t_i[.]t^*), \alpha)$$

when $\alpha = \gamma(\boldsymbol{x}(t^*[.]t_f))$, and β is continuous and non-decreasing in α. Under these conditions γ is called a *positional functional* [31]. The principle optimality is applicable to cost functionals which are positional.

2.3.4 Numerical methods for solving Hamilton-Jacobi equations

A static Hamilton-Jacobi equation is a first-order nonlinear PDE of the type

$$\begin{aligned} H(\boldsymbol{x}, \nabla\vartheta(\boldsymbol{x})) = 0,\ \boldsymbol{x} \in \Gamma \\ \vartheta(\boldsymbol{x}) = q(\boldsymbol{x}),\ \boldsymbol{x} \in \partial\Gamma \end{aligned} \tag{2.11}$$

where $\Gamma \subset \mathbf{R}^n$ is open, $H : \Gamma \times \mathbf{R}^n \to \mathbf{R}$ and $\vartheta : \Gamma \cup \partial\Gamma \to \mathbf{R}$ is the unknown function. These equations arise from several applications including geometric optics, computer vision, computational geometry, geophysics and optimal control. Note that (2.9) is itself a static Hamilton-Jacobi PDE, if we set

$$H(\boldsymbol{x}, \nabla\vartheta(\boldsymbol{x})) := \inf_{\boldsymbol{v} \in \mathscr{U}} \{g(\boldsymbol{x}, \boldsymbol{v}) + \nabla\vartheta(\boldsymbol{x}) \cdot \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{v})\}$$

It is well known that even if H and q are smooth, a smooth solution of (2.11) need not exist, and one must adopt the notion of a viscosity solution [6]. For a given problem there can be an infinite number of nonsmooth ‘weak’ solutions, but only one matches the physical interpretation of the problem, and the viscosity solution is defined so that it matches this ‘physically correct’ solution [51].

In most numerical methods, the first step to the numerical solution of this type of equation is to discretize the domain $\Gamma \cup \partial\Gamma$ into a finite set of points. The viscosity solution of (2.11) is then approximated over that set of points. These numerical methods can be classified as Eulerian, Lagrangian and Semi-Lagrangian. In Eulerian methods, the set of points is a fixed grid and the equation is discretized using finite differences. In Lagrangian methods, on the other hand, the set of points moves according to the solution (in the case of a Hamilton-Jacobi PDE originating from optimal control problem, each point would follow an optimal trajectory). Semi-Lagrangian methods are a hybrid, in that there is also a fixed grid, but the solution is calculated by moving each gridpoint as in a Lagrangian method for a small timestep. Here we discuss Eulerian and Semi-Lagrangian methods which are suitable for solving Hamilton-Jacobi equations arising from the application of dynamic programming to optimal control problems.

The grid used in Eulerian and Lagrangian methods can be regular or unstructured. A regular (also known as Cartesian or rectangular) grid is a set of regularly spaced points $\boldsymbol{x}^{i_1,i_2,\ldots,i_n}$ discretizing a rectangular domain D which contains $\Gamma \cup \partial\Gamma$. An unstructured grid can be any set of points, with the advantage that it can be adapted to the geometry of $\Gamma \cup \partial\Gamma$. In both cases, the grid has a graph-like structure, so that each gridpoint has a set of neighbors. The set of points which do not have a neighbor in some direction is the *computational boundary*. Although most of the methods described below have extensions to unstructured grids, we focus on regular grids. In that case, every point not in the computational boundary has two neighbors for each dimension, and the computational boundary consists of points on the boundary of the rectangular domain D.

The simplest case of (2.11) can be seen as the Hamilton-Jacobi-Bellman PDE of a particularly simple instance of the minimum-cost-to-target problem we considered in Section 2.3.2.2. If the dynamics are given by the equation

$$\dot{\boldsymbol{x}}(t) = \boldsymbol{u}(t),\ \boldsymbol{u}(t) \in \mathbf{B}_\gamma,$$

where $\mathbf{B}_\gamma$ is a closed ball with radius γ centered at the origin, then the PDE for the value function is

$$\gamma |\nabla \vartheta(\boldsymbol{x})| = g(\boldsymbol{x}) \tag{2.12}$$

which is the well-known Eikonal equation from geometric optics (here $|\boldsymbol{p}|$ denotes the standard Euclidean norm of $\boldsymbol{p} \in \mathbf{R}^n$). If an approximation of ϑ is known, the optimal control value at each point can be computed from the formula

$$\boldsymbol{u}(t) = -\frac{\nabla \vartheta(\boldsymbol{x}(t))}{|\nabla \vartheta(\boldsymbol{x}(t))|}$$

Tsitsiklis [64] described an efficient Semi-Lagrangian method for the approximation of ϑ based on direct discretization of this optimal control problem. At each grid point, the optimal path passing through that grid point is approximated by a straight line until it exits the simplex defined by the point under consideration and its neighbors. The cost-to-go at the point is then approximated by the cost of the straight line path plus the cost-to-go at the intersection of the path and the simplex, which is linearly interpolated from the values of the cost-to-go at neighboring nodes. Hence, if the problem is to be solved on a grid of side h, letting Δ be the unit simplex in $\mathbf{R}^n$, $\mathscr{A} = \{-1,1\}^n$ and $\boldsymbol{e}_i$ be the i-th coordinate vector in $\mathbf{R}^n$, ϑ is approximated using the expression

$$\vartheta(\boldsymbol{x}) \approx \min_{\alpha \in \mathscr{A},\ \theta \in \Delta} \left\{ g(\boldsymbol{x}) \frac{h}{\gamma} |\theta| + \sum_{i=1}^{n} \theta_i \vartheta(\boldsymbol{x} + h\alpha_i \boldsymbol{e}_i) \right\} \tag{2.13}$$

which can be seen as a first order approximation of (2.8). It can be shown that at the optimal α, θ in (2.13) it holds that if $\theta_i > 0$ then $\vartheta(\boldsymbol{x}+\alpha_i \boldsymbol{e}_i) < \vartheta(\boldsymbol{x})$, that is, the solution at a point depends only on the solution at neighboring nodes with a smaller value of $\vartheta(\boldsymbol{x})$. This is known as a *causality* property. Using this fact, an algorithm similar to Dijkstra's method can be devised with $\mathscr{O}(N \log N)$ complexity, N being the total number of grid points. For each possible choice of α_i, the optimal θ_i is determined from the solution of a convex optimization problem. This problem is solved for each α_i, and the solution resulting in the lowest value of ϑ is chosen.

This idea can be extended to obtain Semi-Lagrangian methods for solving a wider class of optimal control problems, such as the one considered in Section 2.3.2.2. The dynamic programming principle can be discretized by selecting a timestep h:

$$\vartheta(\boldsymbol{x}) = \inf_{\boldsymbol{u}} \left\{ \int_0^h g(\boldsymbol{\xi}(t;\boldsymbol{x},\boldsymbol{u}),\boldsymbol{u}(t))\mathrm{d}t + \vartheta(\boldsymbol{\xi}(h;\boldsymbol{x},\boldsymbol{u})). \right\}$$

Choosing appropriate discretization schemes for the integral and the trajectory, this can be approximated by a finite-dimensional minimization. For instance, with first order scheme for both the cost integral and the solution of $\boldsymbol{\xi}(h;\boldsymbol{x},\boldsymbol{u})$:

$$\vartheta(\boldsymbol{x}) \approx \inf_{\boldsymbol{u}\in\mathscr{U}} \{h \cdot g(\boldsymbol{x},\boldsymbol{u}) + \vartheta(\boldsymbol{x}+h\boldsymbol{f}(\boldsymbol{x},\boldsymbol{u}))\} \tag{2.14}$$

By restricting the computation to a discrete grid of points, $\vartheta(\boldsymbol{x}+h\boldsymbol{f}(\boldsymbol{x},\boldsymbol{u}))$ can be approximated by an interpolation of the values of ϑ at the gridpoints near $\boldsymbol{x}+h\boldsymbol{f}(\boldsymbol{x},\boldsymbol{u})$. In general this equation does not obey a causality principle, so the formula must be applied iteratively in all gridpoints until ϑ is stationary [6]. The method requires the solution of a possibly nonconvex optimization problem at each gridpoint. This can be avoided by discretizing the control set $\mathscr{U}$ into a finite set, which obviously induces a loss of accuracy.

Returning to the Eikonal equation, the complementary approach to Tsitsiklis's method is the approximation of the gradient in (2.12) by an appropriate finite-difference formula to obtain an Eulerian method. The result is a system of coupled nonlinear equations, one for the value of ϑ at each grid point. The problem then consists in finding an efficient way of solving such a system. By exploiting the causality property to decouple the system, the fast marching method [51, 52] achieves the same complexity as Tsitsiklis's method. The simplest finite-difference scheme which can be used is

$$\sum_{i=1}^{n} (\vartheta(\boldsymbol{x}) - \vartheta(\boldsymbol{x}_i))^2 = \left(\frac{h}{\gamma} g(\boldsymbol{x})\right)^2.$$

Where each $\boldsymbol{x}_i$ is a neighbor of $\boldsymbol{x}$ in the i-th coordinate direction. In fact, if such a first order approximation is used, it can be shown that fast marching is equivalent to Tsitsiklis's algorithm [3]. These algorithms and Dijkstra's algorithm are particular cases of a generic algorithm which can approximate the viscosity solution of

$$\gamma |\nabla \vartheta(\boldsymbol{x})|_p = g(\boldsymbol{x})$$

where $|\cdot|_p$ is the p-norm on $\mathbf{R}^n$. This is discussed in [3] where applications to robotic path planning are discussed.

Fast marching methods can be extended to optimal control problems where the dynamics are of the type

$$\dot{\boldsymbol{x}}(t) = \gamma(\boldsymbol{x}(t), \boldsymbol{u}(t))\boldsymbol{u}(t), \ |\boldsymbol{u}(t)| = 1$$

where γ (the speed of motion) is a strictly positive real-valued function [54]. The associated Hamilton-Jacobi equation is

$$\inf_{|\boldsymbol{v}|=1} \{\gamma(\boldsymbol{x}, \boldsymbol{v})\nabla\vartheta(\boldsymbol{x}) \cdot \boldsymbol{v}\} = -g(\boldsymbol{x}).$$

The complexity in this case is $\mathscr{O}(\Upsilon N \log N)$, where Υ quantifies how much γ depends on the direction of $\boldsymbol{u}$ (the degree of anisotropy). This class of methods, known as ordered upwind methods, has been shown to apply to certain classes of problems with time-varying dynamics [65] and to hybrid control problems [53].

For the particular case where H is affine in one of the components of the gradient, i.e., (2.11) can be written as

$$\frac{\partial}{\partial t}\vartheta(t, \boldsymbol{y}) + H_0(t, \boldsymbol{y}, \nabla_{\boldsymbol{y}}\vartheta(t, \boldsymbol{y})) = 0, \tag{2.15}$$

with Γ equal to the complement of $\{(t, \boldsymbol{y}) \mid t = t_0\}$ for some t_0 , so that the boundary condition is

$$\vartheta(t_0, \boldsymbol{y}) = q(\boldsymbol{y}), \text{ all } \boldsymbol{y},$$

a class of algorithms known as *level set methods* may be applied. These are adaptations of finite difference schemes originally developed for hyperbolic conservation laws, which were observed to have a connection with equations of this type [42]. General equations of the form (2.11) can be embedded in a problem involving a related equation of the form (2.15) in $n+1$ dimensions [41].

For the hybrid control problem of simultaneously planning the optimal rendezvous locations of a set of robots and the trajectories of each robot between the locations given a set of meeting events with the structure of a tree, an efficient method was derived by combining the application of dynamic programming over the tree structure and the fast marching method or Dijkstra's algorithm depending on whether the robot state space is considered to be continuous or discrete [4].

2.3.4.1 Fast sweeping methods

Fast sweeping methods (FSM) are a class of iterative Eulerian methods which were originally formulated for the Eikonal equation and later generalized to arbitrary Hamilton-Jacobi equations.

A fast sweeping method is formulated for Eikonal equations in [72]. The left-hand-side is discretized using a first-order finite-difference formula similar to the one employed for the fast marching method. This results in an update formula for the value of the solution at a grid point as a function of the values at the neighbors, e.g., in two dimensions:

$$\vartheta_{\text{new}}(\boldsymbol{x}^{i,j}) = \texttt{update}(\vartheta(\boldsymbol{x}^{i-1,j}), \vartheta(\boldsymbol{x}^{i+1,j}), \vartheta(\boldsymbol{x}^{i,j-1}), \vartheta(\boldsymbol{x}^{i,j+1})) \tag{2.16}$$

Naturally, the update formula must be adapted to appropriately accommodate points in the computational boundary.

The algorithm works by applying this update formula to all points of the grid in different orders, or *sweeping directions*. In two dimensions, we can sweep the gridpoints $\boldsymbol{x}^{ij}$ in four different orders:

1. Ascending i and j;
2. Ascending i and descending j;
3. Descending i and ascending j;
4. Descending i and j.

The sweeps are done sequentially over the same data (Gauss-Seidel iteration). In n dimensions there are 2^n different possible sweeping orders.

For general Hamilton-Jacobi equations an update formula like (2.16) may not be easy to derive, or its computation can be nontrivial, requiring for instance the solution of a nonlinear optimization problem [29]. In order to extend the fast sweeping method to a wider class of Hamilton-Jacobi equations, a first order Lax-Friedrichs discretization of the Hamiltonian is adopted [28]. Hence, for equations of the form

$$H(\boldsymbol{x}, \nabla\vartheta(\boldsymbol{x})) = g(\boldsymbol{x}),\ \boldsymbol{x} \in \Gamma \tag{2.17}$$

$$\vartheta(\boldsymbol{x}) = q(\boldsymbol{x}),\ \boldsymbol{x} \notin \Gamma, \tag{2.18}$$

the Hamiltonian H is discretized as follows (considering $\boldsymbol{x} \in \mathbf{R}^2$ for simplicity):

$$H_{\text{LF}}(\boldsymbol{x}^{i,j}) = H\left(\boldsymbol{x}^{i,j}, \frac{p_1^+ + p_1^-}{2}, \frac{p_2^+ + p_2^-}{2}\right) + \sigma_1(\boldsymbol{x}^{i,j})\frac{p_1^+ - p_1^-}{2} + \sigma_2(\boldsymbol{x}^{i,j})\frac{p_2^+ - p_2^-}{2}$$

where h_i is the grid spacing in dimension i, $p_{1,2}^{+,-}$ are the forward and backward first order finite differences in each dimension, i.e.

$$p_1^+ = \frac{\vartheta(\boldsymbol{x}^{i+1,j}) - \vartheta(\boldsymbol{x}^{i,j})}{h_1},\ p_1^- = \frac{\vartheta(\boldsymbol{x}^{i,j}) - \vartheta(\boldsymbol{x}^{i-1,j})}{h_1}$$

$$p_2^+ = \frac{\vartheta(\boldsymbol{x}^{i,j+1}) - \vartheta(\boldsymbol{x}^{i,j})}{h_2},\ p_2^- = \frac{\vartheta(\boldsymbol{x}^{i,j}) - \vartheta(\boldsymbol{x}^{i,j-1})}{h_2}$$

and the *artificial viscosity coefficients* σ_i satisfy

$$\sigma_i(\boldsymbol{x}) \geq \max_{\boldsymbol{p}} \left| \frac{\partial H}{\partial p_i}(\boldsymbol{x}, \boldsymbol{p}) \right|.$$

The viscosity terms promote convergence but affect the accuracy of the numerical solution, so the σ_i should be as small as possible.

Solving $H_{\mathrm{LF}}(\boldsymbol{x}^{i,j}) = g(\boldsymbol{x}^{i,j})$ for $\vartheta(\boldsymbol{x}^{i,j})$, the updated value is given by

$$\begin{aligned}\vartheta(\boldsymbol{x}^{i,j}) = \frac{1}{\frac{\sigma_1}{h_1}+\frac{\sigma_2}{h_2}}\Bigg(g(\boldsymbol{x}^{i,j}) - H\bigg(\boldsymbol{x}^{i,j}, \frac{\vartheta(\boldsymbol{x}^{i+1,j})-\vartheta(\boldsymbol{x}^{i-1,j})}{2h_1}, \frac{\vartheta(\boldsymbol{x}^{i,j+1})-\vartheta(\boldsymbol{x}^{i,j-1})}{2h_2}\bigg) \\ + \sigma_1\frac{\vartheta(\boldsymbol{x}^{i+1,j})+\vartheta(\boldsymbol{x}^{i-1,j})}{2h_1} + \sigma_2\frac{\vartheta(\boldsymbol{x}^{i,j+1})+\vartheta(\boldsymbol{x}^{i,j-1})}{2h_2}\Bigg)\end{aligned} \tag{2.19}$$

The generalization to an arbitrary number of dimensions is straightforward.

The authors of [28] also suggest a simple way of handling the points in the computational boundary. Recall that the value function gives the cost of the optimal trajectory from each point. This cost decreases along the optimal trajectories, so assuming that the optimal trajectories are contained in the computational region, the value function should increase toward the computational boundary. Consider for instance the point $\boldsymbol{x}^{0,j}$, which lies on the computational boundary corresponding to the first dimension. If $\frac{\partial\vartheta}{\partial x_1} \approx \frac{\vartheta(\boldsymbol{x}^{2,j})-\vartheta(\boldsymbol{x}^{1,j})}{h_1} < 0$, the solution is increasing toward the boundary, so we can extrapolate linearly:

$$\vartheta(\boldsymbol{x}^{0,j}) = \vartheta(\boldsymbol{x}^{1,j}) + \vartheta(\boldsymbol{x}^{1,j}) - \vartheta(\boldsymbol{x}^{2,j}) = 2\vartheta(\boldsymbol{x}^{1,j}) - \vartheta(\boldsymbol{x}^{2,j})$$

If on the other hand $\frac{\partial\vartheta}{\partial x_1} \approx \frac{\vartheta(\boldsymbol{x}^{2,j})-\vartheta(\boldsymbol{x}^{1,j})}{h_1} \geq 0$, we force $\frac{\partial\vartheta}{\partial x_1} = 0$ by setting the centered derivative to zero:

$$\frac{\vartheta(\boldsymbol{x}^{2,j}) - \vartheta(\boldsymbol{x}^{0,j})}{h_1} = 0 \iff \vartheta(\boldsymbol{x}^{0,j}) = \vartheta(\boldsymbol{x}_{2,j})$$

In summary, we have

$$\vartheta(\boldsymbol{x}^{0,j}) = \begin{cases} 2\vartheta(\boldsymbol{x}^{1,j}) - \vartheta(\boldsymbol{x}^{2,j}), & \vartheta(\boldsymbol{x}^{1,j}) > \vartheta(\boldsymbol{x}^{2,j}) \\ \vartheta(\boldsymbol{x}^{2,j}), & \vartheta(\boldsymbol{x}^{1,j}) \leq \vartheta(\boldsymbol{x}^{2,j}) \end{cases}$$

Since $\vartheta(\boldsymbol{x}^{1,j}) > \vartheta(\boldsymbol{x}^{2,j}) \iff 2\vartheta(\boldsymbol{x}^{1,j}) - \vartheta(\boldsymbol{x}^{2,j}) > \vartheta(\boldsymbol{x}^{2,j})$, this can be written succinctly as

$$\vartheta(\boldsymbol{x}^{0,j}) = \max\{\vartheta(\boldsymbol{x}^{2,j}), 2\vartheta(\boldsymbol{x}^{1,j}) - \vartheta(\boldsymbol{x}^{2,j})\} \tag{2.20}$$

This is easily generalized to the remaining points of the computational boundary.

To initialize the algorithm, we set all grid points in Γ to some value M larger than the maximum value of the true solution; points not in Γ are set to their true value. At each iteration, we only update the value of a point if the new value is smaller than the old one.

In summary, the algorithm can be described as the following sequence of steps:

1. Initialization: for each gridpoint $\boldsymbol{x}$
   - If $\boldsymbol{x} \in \Gamma$, $\vartheta(\boldsymbol{x}) \leftarrow M$
   - If $\boldsymbol{x} \notin \Gamma$, $\vartheta(\boldsymbol{x}) \leftarrow q(\boldsymbol{x})$.

2. Iteration: While $\left|\vartheta^{i+1} - \vartheta^{i}\right|_{\infty} \geq \varepsilon$
    (a) For each sweeping order $1, \ldots, 2^n$
        i. For each gridpoint $\boldsymbol{x} \in \Gamma$ not in the computational boundary
            A. Calculate an updated value ϑ_{new} using formula (2.19)
            B. $\vartheta(\boldsymbol{x}) \leftarrow \min\{\vartheta_{\text{new}}, \vartheta(\boldsymbol{x})\}$
        ii. For each gridpoint $\boldsymbol{x} \in \Gamma$ in the computational boundary
            A. Calculate an updated value ϑ_{new} using formula (2.20)
            B. $\vartheta(\boldsymbol{x}) \leftarrow \min\{\vartheta_{\text{new}}, \vartheta(\boldsymbol{x})\}$

The algorithm terminates when it reaches a stationary point, i.e.

$$\left|\vartheta^{i+1} - \vartheta^{i}\right|_{\infty} = \max_{\boldsymbol{x}}\left\{\vartheta^{i}(\boldsymbol{x}) - \vartheta^{i+1}(\boldsymbol{x})\right\} < \varepsilon$$

where ϑ^i is the value function at the i-th iteration and $\varepsilon > 0$ is a user-supplied parameter indicating the desired precision.

No proof of convergence is available, but numerical evidence for the convergence and accuracy of the algorithm is given in [28]. The numerical experiments suggest first order convergence and $\mathscr{O}(N \log N)$ complexity on average.

As the algorithm has quite a simple structure, it is amenable to parallel implementations. An immediately obvious and simple way to parallelize the algorithm is to perform the different sweeps in parallel [73]. To do this, one keeps $2^n + 1$ versions of the value function, ϑ_i, $i = 0, \ldots, 2^n$. The for loop in step 2a of the algorithm is done in parallel, with sweep i and the following computational boundary update performed on ϑ_i, $i = 1, \ldots, 2^n$. Each iteration is followed by the following synchronization step:

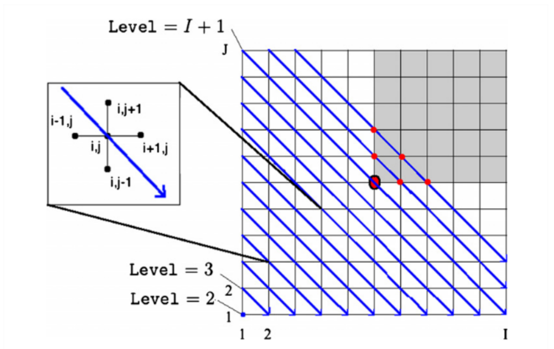
1. For each gridpoint $\boldsymbol{x} \in \Gamma$
    (a) $\vartheta_0(\boldsymbol{x}) \leftarrow \min_{0 \leq i \leq 2^n} \vartheta_i(\boldsymbol{x})$
    (b) For $i = 1, \ldots, 2^n$, $\vartheta_i(\boldsymbol{x}) \leftarrow \vartheta_0(\boldsymbol{x})$

Since the sweeps are no longer done sequentially, more iterations may be necessary for convergence. In addition, 2^n hardware threads are required in order to do all the sweeps in parallel, and the amount of memory required to store the $2^n + 1$ copies of ϑ can be prohibitively high for large or high-dimensional problems.

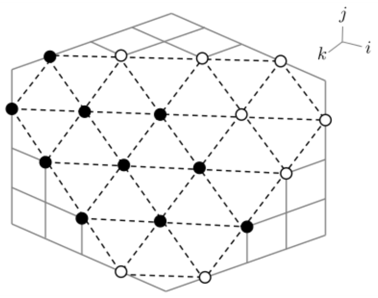
A subtler but more efficient and scalable parallelization of this algorithm is the hyperplane stepping method [13]. Let L_m be the set of gridpoints whose grid indices $(i_1, i_2, \ldots, i_n)$ satisfy

$$\sum_{k=1}^{n} i_k = m.$$

Then, if $\boldsymbol{x}^1, \boldsymbol{x}^2 \in L_m$, it is easy to see that the update formula (2.19) for $\boldsymbol{x}^1$ does not depend on $\boldsymbol{x}^2$ and vice-versa. This is illustrated in Figure 2.10a for $n = 2$, in which case each L_m is a diagonal



(a) Illustration of the independence between the points in L_m in a two-dimensional problem



(b) Graphical representation of $L_m = \{(i, j, k) \mid i + j + k = m\}$ in a three-dimensional problem

Figure 2.10: Graphical representation of the hyperplane stepping method (adapted from Detrixhe et al. [13])

line. In general, the sets L_m are shaped like hyperplanes, as represented in Figure 2.10b for $n = 3$. Consequentially, the points in L_m can be updated simultaneously, so that a speedup close to the optimal speedup of p can be achieved, where p is the number of processors. This method can be used with any number of processors, and the memory requirements and number of iterations will be exactly the same as in the original (serial) version.

Both parallel algorithms can be combined with a domain decomposition approach [12, 73].

2.4 The LSTS software toolchain

The LSTS software toolchain is a set of open-source modular tools for operations with networked vehicle systems. Together, these tools form a cohesive framework for mixed-initiative (humans-in-the-loop) control of multiple unmanned air, ground and sea vehicles [18]. The three main components of the LSTS software toolchain are the DUNE Unified Navigation Environment, the Inter-Module Communication protocol (IMC) and Neptus.

Neptus is a command and control infrastructure which allows a human operator to interact with a networked vehicle system. It provides a configurable and extensible graphical interface which can adapt to different types of vehicles. In addition, Neptus has built-in support for all phases of a mission's life cycle, namely mission planning and simulation, execution and control, and review and analysis (e.g. data visualization) [45, 46]

IMC is a message-oriented protocol for networked vehicle and sensor operations, abstracting communications hardware and media by providing a shared set of messages without assuming any specific software architecture for the client applications [39]. IMC messages are used within the toolchain for inter-process, inter-vehicle and operator-vehicle communications, as well as logging and web dissemination. The protocol is defined in a single XML file from which bindings for different programming languages can be generated [46].

DUNE is a platform- and architecture-independent runtime environment written in C++, providing onboard software components for sensing, control, navigation, communication and vehicle supervision. The runtime environment comprises a set of independent building blocks which are called 'DUNE tasks'. Each task has a distinct purpose and runs in a separate thread of execution. Examples of DUNE tasks are sensor drivers, controllers or loggers. Tasks communicate by publishing and subscribing to IMC message types which are forwarded by a global message bus [16, 46]. The set of tasks which is active in any particular instance of DUNE is defined in a runtime configuration file. For instance, a configuration file can be defined for a particular vehicle, enabling tasks which are relevant for the vehicle's hardware and payload. This configuration file can also be used to define configuration parameters, so that tasks can be written in a vehicle-independent fashion, and adapted to each vehicle at runtime. Sets of tasks can also be enabled and disabled according to different operating *profiles*. Available profiles are for instance `Never` (the task is never enabled), `Always` (the task is always enabled), `Simulation` (the task is enabled when there is no hardware in the loop), `HIL` (for hardware-in-the-loop simulation), and `Hardware` (for running DUNE with the vehicle's onboard computer connected to the actual vehicle hardware). [45, 46].

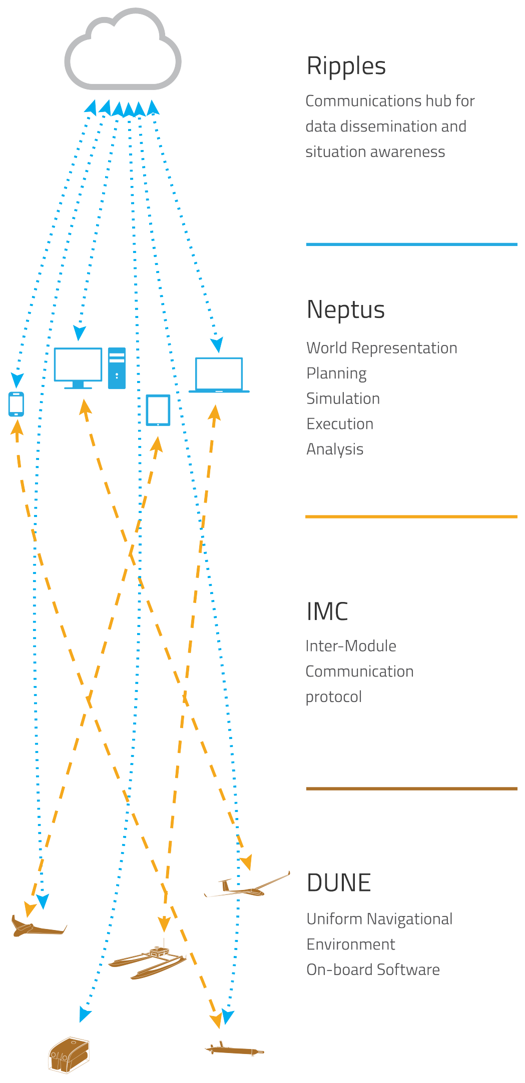
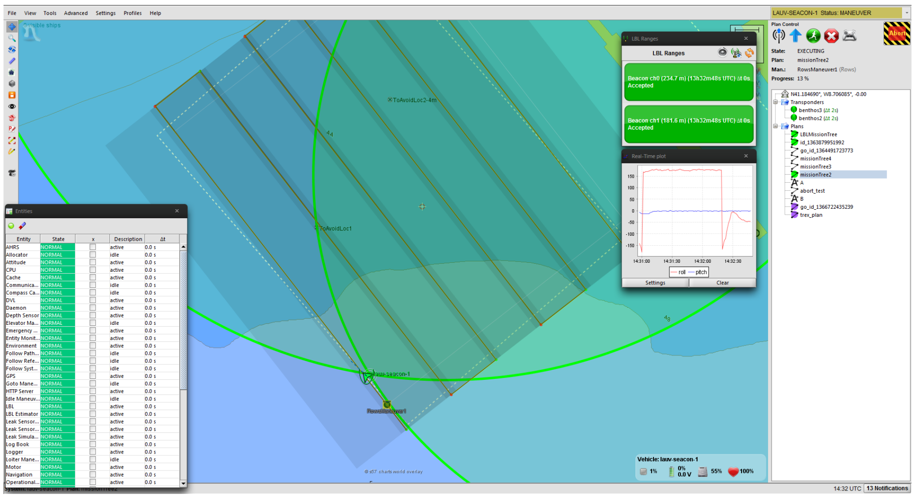

Ripples
Communications hub for data dissemination and situation awareness
Neptus
World Representation
Planning
Simulation
Execution
Analysis
IMC
Inter-Module Communication protocol
DUNE
Uniform Navigational Environment On-board Software

Figure 2.11: LSTS toolchain components

These different components support a layered control architecture where a *mission* is defined as a set of *plans*, each of which is a sequence of *maneuvers*. A maneuver is a high-level control objective such as a waypoint the vehicle should reach, a loiter command or a trajectory to be tracked. In DUNE, maneuver controllers then translate these objectives into navigation and guidance commands, which are used as actuation references by low-level controllers [45, 46]. The different layers interact via IMC messages, so that plans and maneuvers can be generated either onboard by the vehicle or through an operator console such as the one provided by Neptus, with no change in the lower layers [45].

As described in Estrela da Silva et al. [16], DUNE contains an underwater vehicle simulation engine which effectively replaces a vehicle's sensors and actuators without affecting the rest of the toolchain ecosystem, since the simulator communicates with the remaining components via IMC messages. Thus, the toolchain can be used for realistic software-in-the-loop simulation and testing.



LAUV-SEACON-1 Status: MANEUVER

Plan Control

State: EXECUTING
Plan: missionTree2
Man.: RowsManeuver1 (Rows)
Progress: 13 %

N41.184690°, W8.706085°, -0.00
Transponders
benthos3 (Δt 2s)
benthos2 (Δt 2s)
Plans
LBLMissionTree
id_1363879951992
go_id_1364491723773
missionTree4
missionTree3
missionTree2
A
abort_test
B
go_id_1366722435239
trex_plan

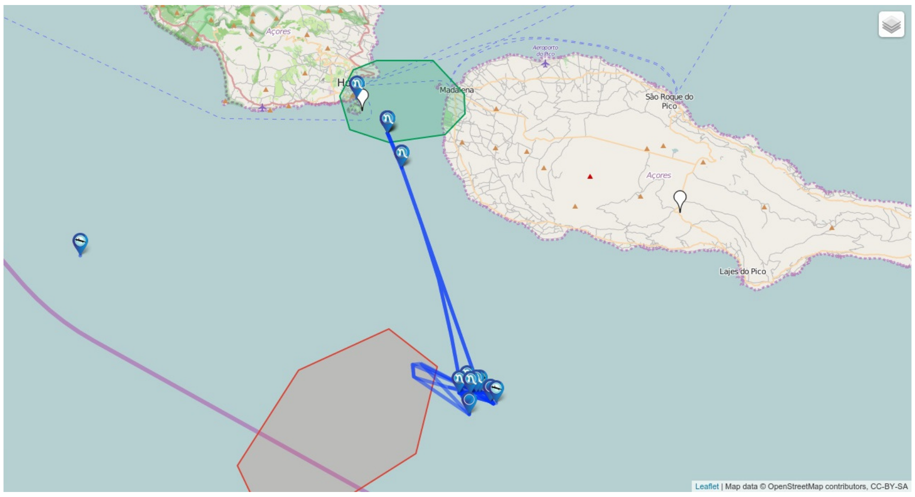
Beacon ch0 (234.7 m) (13h32m48s UTC) Δt 0s Accepted

Beacon ch1 (181.6 m) (13h32m48s UTC) Δt 0s Accepted

Entities

Entity	State	x	Description	Δt
AHRS	NORMAL	☐	active	0.0 s
Allocator	NORMAL	☐	idle	0.0 s
Attitude	NORMAL	☐	active	0.0 s
CPU	NORMAL	☐	active	0.0 s
Cache	NORMAL	☐	active	0.0 s
Communica...	NORMAL	☐	idle	0.0 s
Compass Ca...	NORMAL	☐	idle	0.0 s
DVL	NORMAL	☐	active	0.0 s
Daemon	NORMAL	☐	active	0.0 s
Depth Sensor	NORMAL	☐	active	0.0 s
Elevator Ma...	NORMAL	☐	idle	0.0 s
Emergency ...	NORMAL	☐	active	0.0 s
Entity Monit...	NORMAL	☐	active	0.0 s
Environment	NORMAL	☐	active	0.0 s
Follow Path...	NORMAL	☐	idle	0.0 s
Follow Refe...	NORMAL	☐	idle	0.0 s
Follow Syst...	NORMAL	☐	idle	0.0 s
GPS	NORMAL	☐	active	0.0 s
Goto Mane...	NORMAL	☐	idle	0.0 s
HTTP Server	NORMAL	☐	active	0.0 s
Idle Maneuv...	NORMAL	☐	idle	0.0 s
LBL	NORMAL	☐	active	0.0 s
LBL Estimator	NORMAL	☐	active	0.0 s
Leak Sensor...	NORMAL	☐	active	0.0 s
Leak Sensor...	NORMAL	☐	active	0.0 s
Leak Simula...	NORMAL	☐	active	0.0 s
Log Book	NORMAL	☐	active	0.0 s
Logger	NORMAL	☐	active	0.0 s
Loiter Mane...	NORMAL	☐	idle	0.0 s
Motor	NORMAL	☐	active	0.0 s
Navigation	NORMAL	☐	active	0.0 s
Operational...	NORMAL	☐	active	0.0 s

(a) A Neptus console



(b) Ripples

Also included in the LSTS toolchain are Ripples, a cloud-based centralized hub for data disseminations and situation awareness, ACCU, a command and control Android application, and GLUED, a minimal Linux distribution for embedded systems. The toolchain is currently used in over 15 countries, both by academic research groups and in the industry.

Chapter 3

Problem Description

3.1 The base problem

The simplest version of the type of problems considered in this work can be stated as follows:

Given the initial position of a vehicle, the deployment time and a forecast of the ocean currents in the operational area, what is the best trajectory which takes the vehicle to some target position?

Of course, one has to specify the meaning of ‘best’ trajectory, and this is done using a cost function, which allows us to compare or rank trajectories. The most common objectives would be to minimize the traveling time or energy. However, one may also be interested in generating risk-averse trajectories, for instance.

There is an immediate generalization of this problem which is important in practice. Namely, the deployment position and time should be allowed to vary. Consider for instance the case of deployment of an unmanned vehicle from other marine craft, such as a ship or a manned submarine, where practical circumstances may force changes in the deployment position. Even if one can accommodate these changes, it is only logical to want to choose the deployment position in order to obtain the best possible performance. The situation is even more dramatic when it comes to the deployment time: in practice one may have only a window of time for the deployment, and in some scenarios of practical interest (such as deployments in zones with tidal-forced ocean currents) the deployment time can make the difference between feasibility and infeasibility of the mission objectives.

Thus, an updated version of the problem statement reads as follows:

Given a possible range of initial positions of a vehicle, a range of deployment times, and a forecast of the ocean currents in the operational area, what is the best trajectory which takes the vehicle to some target position, for each possible initial position and time?

At this point we can introduce some notation to express this rigorously. In what follows we consider planar motions only, although it is directly applicable to trajectories in three-dimensions. We denote by $\boldsymbol{x}(t) \in \mathbf{R}^2$ the horizontal position of the vehicle at some time $t \in \mathbf{R}$. Consider first that the deployment position and time are fixed and equal to $\boldsymbol{\xi}_0$ and t_i, respectively, and let the target position be $\boldsymbol{\xi}_T$. A *trajectory* is then a function $\boldsymbol{x} : [t_i, t_f] \to \mathbf{R}^2$ mapping time instants to vehicle

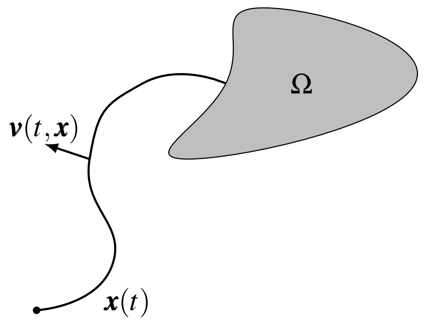


Figure 3.1: A trajectory reaching the target set Ω under the influence of the ocean current $\boldsymbol{v}$.

positions, which naturally should be at least continuous. We should consider only those trajectories satisfying $\boldsymbol{x}(t_i) = \boldsymbol{\xi}_0$ and $\boldsymbol{x}(t_f) = \boldsymbol{\xi}_T$. Trajectories are compared using an integral cost function:

$$J(\boldsymbol{x}) = \int_{t_i}^{t_f} g(\boldsymbol{x}(t))\mathrm{d}t \tag{3.1}$$

where g is a piecewise continuous positive real-valued function which gives the cost per unit time of travelling through a given point. The simplest example would be to take g identically equal to one, in which case

$$J(\boldsymbol{x}) = \int_{t_i}^{t_f} \mathrm{d}t = t_f - t_i,$$

which equals the time taken by the vehicle to reach $\boldsymbol{\xi}_T$. If g is for instance a risk map which assigns a risk level to each position, the optimal trajectories will be the trajectories which minimize the cumulative risk of travelling from the deployment position to the deployment time.

The ‘best’ trajectory $\boldsymbol{x}_\star$ can then be defined as a minimizer of this cost function over all trajectories $\boldsymbol{x}$:

$$\boldsymbol{x}_\star \in \arg\min\left\{J(\boldsymbol{x}) \;\middle|\; \boldsymbol{x} : [t_i, t_f] \to \mathbf{R}^2,\ \boldsymbol{x}(t_i) = \boldsymbol{\xi}_0,\ \boldsymbol{x}(t_f) = \boldsymbol{\xi}_T\right\}. \tag{3.2}$$

Note that t_f is unspecified in this formulation.

If we consider that the initial time and position can range over some set, the formulation is the same, but the optimal trajectory then depends on the initial time and position:

$$\boldsymbol{x}_\star^{t;\boldsymbol{\xi}} \in \arg\min\left\{J(\boldsymbol{x}) \;\middle|\; \boldsymbol{x} : [t, t_f] \to \mathbf{R}^2,\ \boldsymbol{x}(t) = \boldsymbol{\xi},\ \boldsymbol{x}(t_f) = \boldsymbol{\xi}_T\right\}.$$

A basic extension of this problem is to consider a target set $\Omega \subset \mathbf{R}^2$ rather than a single target position, as illustrated in Figure 3.1 ($\boldsymbol{v}$ is the forecasted ocean current). In that case it makes sense to consider a cost function of the form

$$J(\boldsymbol{x}) = q(\boldsymbol{x}(t_f)) + \int_{t_i}^{t_f} g(\boldsymbol{x}(t))\mathrm{d}t$$

where q is a nonnegative piecewise continuous real-valued function assigning to each point of Ω the cost of ending the mission at that point.

In this case, the optimal trajectory $\boldsymbol{x}_\star^{t,\boldsymbol{\xi}}$ is defined as

$$\boldsymbol{x}_\star^{t,\boldsymbol{\xi}} \in \arg\min\big\{J(\boldsymbol{x}) \mid \boldsymbol{x} : [t,t_f] \to \mathbf{R}^2,\ \boldsymbol{x}(t) = \boldsymbol{\xi},\ \boldsymbol{x}(t_f) \in \Omega\big\}.$$

As an example, consider $\Omega = \left\{\boldsymbol{\xi}_T^1, \boldsymbol{\xi}_T^2\right\}$, i.e., we have a two possible points where the vehicle can end the mission. These can represent two different harbors where the vehicle can be recovered, for instance. If we would rather have the vehicle stop at $\boldsymbol{\xi}_T^1$ than at $\boldsymbol{\xi}_T^2$, we can set $q(\boldsymbol{\xi}_T^1) < q(\boldsymbol{\xi}_T^2)$ to promote trajectories ending at $\boldsymbol{\xi}_T^1$, but still allow the vehicle to stop at $\boldsymbol{\xi}_T^2$ if the integral component of the cost dominates.

3.2 Planning with logic-based constraints

Going beyond single-stage problems, we want to be able to plan a whole mission given a high-level specification of the objectives and constraints. As an example, consider the following mission specification for an ocean vehicle in an estuarine environment:

1. The vehicle is deployed at sea from a predefined location $\boldsymbol{\xi}^{\text{start}}$; the deployment time is unspecified.

2. After it is deployed the vehicle should go towards a prespecified region Ω inside the river.

3. After arriving at Ω, the vehicle should remain inside it for at least 60 minutes.

4. After the 60 minutes have passed, the vehicle should begin the exfiltration at some unspecified time and go toward some safe region Γ outside the river.

Here too the trajectories will be optimized according to a cost function:

$$J(\boldsymbol{x}^{\text{in}},\boldsymbol{x}^{\text{out}}) = \gamma_{\text{start}}(\tau_{\text{start}}) + \int_{\tau_{\text{start}}}^{\tau_{\text{arrival}}} g(\boldsymbol{x}^{\text{in}}(t))\mathrm{d}t + \gamma_{\text{exit}}(\tau_{\text{exit}}) + \int_{\tau_{\text{exit}}}^{\tau_{\text{end}}} g(\boldsymbol{x}^{\text{out}}(t))\mathrm{d}t + \gamma_{\text{end}}(\tau_{\text{end}},\boldsymbol{x}^{\text{out}}(\tau_{\text{end}}))$$

where $\boldsymbol{x}^{\text{in}} : [\tau_{\text{start}}, \tau_{\text{arrival}}] \to \mathbf{R}^2$ denotes the vehicle trajectory from its starting position to the target region inside the river and $\boldsymbol{x}^{\text{out}} : [\tau_{\text{exit}}, \tau_{\text{end}}] \to \mathbf{R}^2$ denotes the vehicle trajectory during the exfiltration. The cost functions γ_{start} and γ_{exit} can be used to specify preferences and constraints on the start and exit times, respectively, while γ_{end} allows us to do this on the end time and the arrival position, similarly to the function q in the base problem.

The optimal solution to the problem is then defined as

$$(\boldsymbol{x}_\star^{\text{in}},\boldsymbol{x}_\star^{\text{out}}) \in \arg\min\Big\{J(\boldsymbol{x}^{\text{in}},\boldsymbol{x}^{\text{out}}) \mid \boldsymbol{x}^{\text{in}} : [\tau_{\text{start}}, \tau_{\text{arrival}}] \to \mathbf{R}^2,\ \boldsymbol{x}^{\text{out}} : [\tau_{\text{exit}}, \tau_{\text{end}}] \to \mathbf{R}^2, \tau_{\text{exit}} - \tau_{\text{arrival}} \geq 60,$$
$$\boldsymbol{x}^{\text{in}}(\tau_{\text{start}}) = \boldsymbol{\xi}^{\text{start}},\ \boldsymbol{x}^{\text{in}}(\tau_{\text{arrival}}) \in \Omega,\ \boldsymbol{x}^{\text{out}}(\tau_{\text{exit}}) \in \Omega,\ \boldsymbol{x}^{\text{out}}(\tau_{\text{end}}) \in \Gamma\Big\}$$

(assuming that the times are given in minutes). Note that this has a similar form to the minimization in the first problem, only involving two trajectories. The times $\tau_{\text{start}}, \tau_{\text{arrival}}, \tau_{\text{exit}}, \tau_{\text{end}}$ are not fixed, and are chosen as part of the minimization.

The problem of optimizing an entire mission can be arbitrarily complex, given a sufficiently complex mission specification. In this work we focus on a class of problems which generalize the given example, which are n-stage single-vehicle missions. At the start of stage i, $i = 0, \dots, n-1$ we want the vehicle to be in the target set Ω_i, and between stages i and $i+1$ the vehicle follows a trajectory $\boldsymbol{x}^i : [\tau_i, \tau_{i+1}] \to \mathbf{R}^2$. The last trajectory $\boldsymbol{x}^{n-1}$ consists of the single point $\boldsymbol{x}^{n-1}(\tau_{n-1})$. Define a cost function for each stage recursively as

$$\begin{aligned} J_0(\boldsymbol{x}^0) &= \gamma_0(\tau_0, \boldsymbol{x}^0(\tau_0)) \\ J_{i+1}(\boldsymbol{x}^{i+1}, \boldsymbol{x}^i, \dots, \boldsymbol{x}^0) &= J_i(\boldsymbol{x}^i, \dots, \boldsymbol{x}^0) + \int_{\tau_i}^{\tau_{i+1}} g_i(\boldsymbol{x}^i(t))\mathrm{d}t + \gamma_{i+1}(\tau_{i+1}, \boldsymbol{x}^{i+1}(\tau_{i+1})). \end{aligned}$$

The optimal mission plan is then a selection of trajectories $(\boldsymbol{x}_\star^0, \dots, \boldsymbol{x}_\star^{n-1})$ which minimizes the cost function of the last stage:

$$\begin{aligned} (\boldsymbol{x}_\star^0, \dots, \boldsymbol{x}_\star^{n-1}) \in \arg\min \Big\{ & J_{n-1}(\boldsymbol{x}^{n-1}, \dots, \boldsymbol{x}^0) \;\Big|\; \boldsymbol{x}^0(\tau_0) \in \Omega_0, \\ & \boldsymbol{x}^0(\tau_1) = \boldsymbol{x}^1(\tau_1) \in \Omega_1, \dots, \boldsymbol{x}^{n-2}(\tau_{n-1}) = \boldsymbol{x}^{n-1}(\tau_{n-1}) \in \Omega_{n-1} \Big\}. \end{aligned}$$

Note that the trajectories should be consistent, i.e. $\boldsymbol{x}^i(\tau_{i+1}) = \boldsymbol{x}^{i+1}(\tau_{i+1})$. As in the example problem, the τ_i are not fixed.

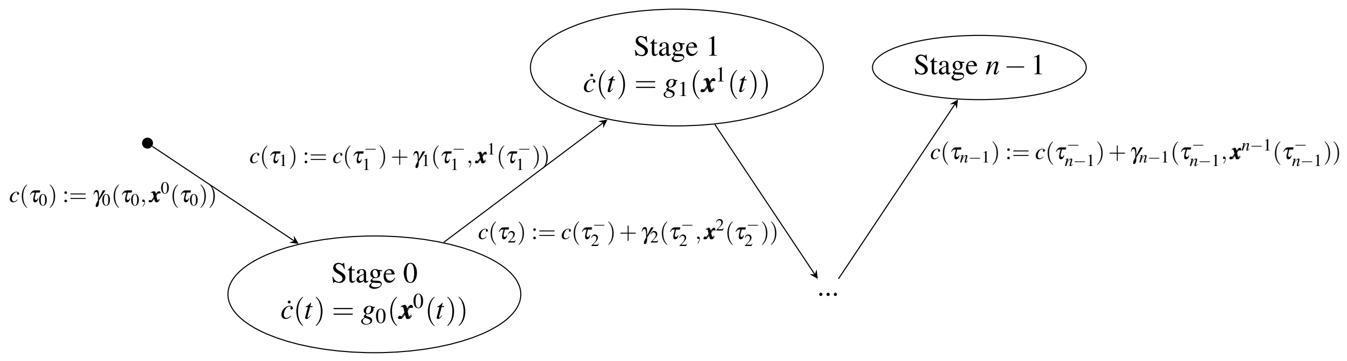

Stage 1
$\dot{c}(t) = g_1(\boldsymbol{x}^1(t))$
Stage $n-1$
$c(\tau_1) := c(\tau_1^-) + \gamma_1(\tau_1^-, \boldsymbol{x}^1(\tau_1^-))$
$c(\tau_{n-1}) := c(\tau_{n-1}^-) + \gamma_{n-1}(\tau_{n-1}^-, \boldsymbol{x}^{n-1}(\tau_{n-1}^-))$
$c(\tau_0) := \gamma_0(\tau_0, \boldsymbol{x}^0(\tau_0))$
$c(\tau_2) := c(\tau_2^-) + \gamma_2(\tau_2^-, \boldsymbol{x}^2(\tau_2^-))$
Stage 0
$\dot{c}(t) = g_0(\boldsymbol{x}^0(t))$
...

Figure 3.2: Representation of the general n-step problem as a hybrid automaton

The problem is represented as a hybrid automaton in Figure 3.2, where $c(t)$ is the temporal evolution of the cost.

Our initial example fits in this general formulation by setting

$$\begin{aligned}
&\Omega_0 = \{\boldsymbol{\xi}^{\text{start}}\},\ \gamma_0(\boldsymbol{\xi}^{\text{start}}, \tau_0) = \gamma_{\text{start}}(\tau_0),\ \boldsymbol{x}^0 = \boldsymbol{x}^{\text{in}},\ g_0 = g \\
&\Omega_1 = \Omega,\ \gamma_1 = 0,\ g_1 = 0 \\
&\Omega_2 = \Omega,\ \gamma_2(\tau_{\text{exit}}, \boldsymbol{x}(\tau_{\text{exit}})) = \gamma_{\text{exit}}(\tau_{\text{exit}}),\ \boldsymbol{x}^2 = \boldsymbol{x}^{\text{out}},\ g_2 = g \\
&\Omega_3 = \Gamma,\ \gamma_3 = \gamma_{\text{end}}.
\end{aligned}$$

Note that there was no cost imposed on the trajectory of the vehicle during the second stage, only that the vehicle stay inside Ω. It is assumed that the vehicle can hold its position while it is inside Ω, so that $\boldsymbol{x}^{\text{out}}(\tau_{\text{exit}}) = \boldsymbol{x}^{\text{in}}(\tau_{\text{arrival}})$.

Chapter 4

Related Work

In this chapter we provide an overview of the literature on trajectory optimization for ocean vehicles which is most relevant for this work. In particular, we focus on methods for route optimization using ocean current data. For this reason, we do not discuss methods which also have an optimization component but which have radically different objectives, such as path planning for adaptive sampling.

Inanc et al. [26] describe a method for trajectory optimization based on an optimal control formulation. A holonomic motion model is adopted, and the cost function is a linear combination of the travelling time and the expended energy. An optimal control solver which parametrizes the optimal trajectories using B-spline functions is used to convert the optimal control problem to a nonlinear optimization problem. This method is susceptible to local minima as the resulting optimization problem is nonconvex. Simulations using ocean current data from high frequency radar stations show that there is a link between the optimal trajectories and Lagrangian Coherent Structures in the flow. Namely, an optimal trajectory departing from a point in the LCS to a point in the LCS stays in the LCS. The authors suggest that this may be used as a heuristic for trajectory generation. The time-varying nature of the ocean currents is accounted for using a receding horizon approach, which can lead to suboptimal solutions. Hence, this work is extended by Zhang et al. [71], by considering a fully time-varying model of the ocean currents. Zhang et al. also consider a dynamic model of an AUV with turning rate constraints, but do not compare the results with the original holonomic model. The link between LCS and the optimal trajectories is shown to hold in this case.

Petres et al. [44] discuss an extension of fast marching method for underwater path planning which incorporates elements of the A* algorithm. The original motivation is obstacle avoidance using underwater imagery, from which an obstacle map is derived. The ocean currents, which are assumed to be static, as well as the obstacles are included in the cost function. An ordered upwind method to solve the resulting equation is derived. The authors also show that the radius of curvature of the optimal paths is lower bounded. Soulignac et al. [57, 58] point out that the resulting paths can be unfeasible in the presence of strong currents (i.e., currents with speed exceeding the vehicle's maximum speed), and propose a different wavefront expansion algorithm based on a

Voronoi partition of the operational area and a continuous motion model. This algorithm is however still limited to static current fields.

A trajectory generation method for AUVs in estuarine environments is presented in Kruger et al. [32]. The method represents a trajectory as a sequence of waypoints consisting of three-dimensional position and time, and optimizes these positions and times of using a gradient-based optimization method; the initial and final waypoints are fixed and determine the initial and target position. The number of waypoints between the final and target positions is determined adaptively. No kinematic constraints apart from the forward speed limit are considered, and this constraint is enforced via a penalization term in the cost function. Variations in the forward speed are allowed during the trajectory. The cost function can include both energy and time components, and obstacles are handled using a penalization term. Since the resulting optimization problem is nonconvex, the algorithm may converge to a local minimum, and the solution depends on the path used to initialize the algorithm. To address this, Jonas et al. [69] build on this work, using global optimization techniques to avoid local minima. The impact of the global optimization algorithm on the computation time is not discussed, and there is still no guarantee that a globally optimal path can be found.

The approach most similar to this work is that of Rhoads et al. [48], who propose a solution of the trajectory optimization problem based on solving the Hamilton-Jacobi-Bellman equation associated to the minimum-time-to-target problem. A Lagrangian method for the numerical solution of the HJBE is proposed. Applying the Pontryagin minimum principle, a differential equation for the optimal trajectories is found, parametrized by the vehicle's heading angle at the final position, which coincides with the final value of the costate variable in the minimum principle equations. The operational area is discretized into a grid, and by adaptively sampling the arrival time and heading, an optimal trajectory passing through every gridpoint is eventually found. The authors suggest that the method can be extended to arbitrary target sets (as opposed to the simple case of a point), requiring however that a separate value function be calculated for each connected component of the target set (so that two separate computations of the value function are required if the target set is the union of two points, for instance).

Lolla et al. [36–38] derive a partial differential equation for a time-varying scalar field whose zero level set is the reachability front (the boundary of the reachable set) when the vehicle starts from a fixed deployment position and time, considering a holonomic motion model. Time-optimal trajectories from the fixed starting position to any other position can be recovered from the solution of the partial differential equation. As the PDE is a time-dependent Hamilton-Jacobi equation, it is solved using a level set method. The authors discuss extensions to multiple vehicles in a fixed formation geometry. As usual when employing level set methods, the problem is embedded in a space with an extra dimension, which is not the most efficient approach. The authors suggest that a narrow band level set method can be used to obtain the solution in a more efficient manner. Subramani et al. [62] report on real deployments of AUVs using this method where the performance of the time-optimal trajectories is compared to that of straight-line paths, confirming the advantages of using the time-optimal trajectories. An extension to multi-waypoint missions is presented in

Ferris et al. [20] where both the optimal ordering of the waypoints and the optimal trajectory between each pair of waypoints is calculated. Since the optimal path from a single point to every other point can be calculated using a single computation of the reachability front, some computation is spared in comparison to a brute-force approach. The authors also discuss the use of high-performance computing platforms to reduce the total computation time.

Chapter 5

Approach

The discussion in Chapter 3 motivates the application of dynamic programming to the problem of finding optimal trajectories for marine vehicles with forcing from ocean currents. This will require the specification of a motion model, which also determines the set of trajectories over which the minimization in (3.1) is performed.

5.1 Motion model

As our focus is on trajectory generation, we restrict our focus to kinematic models, and the dynamics are assumed to controlled by suitably fast inner control loops.

Since we are concerned exclusively with planar planning, only three variables are at play, namely the horizontal position, the yaw angle and the corresponding velocities in the body frame. A typical motion model taking these variables into account is the following [15]:

$$\begin{aligned}
\dot{x}_1 &= \mu \cos\psi - \eta \sin\psi + v_1(t, x_1(t), x_2(t)) \\
\dot{x}_2 &= \mu \sin\psi + \eta \cos\psi + v_2(t, x_1(t), x_2(t)) \\
\dot{\psi} &= \omega
\end{aligned}$$

Here (x_1, x_2) is the horizontal position in an inertial frame, μ and η are the body frame velocities (surge and sway, respectively), v_i are the components of the ocean current, ψ is the yaw or heading angle and ω is the vehicle angular velocity (for a discussion on the reduction of a six-degree-of-freedom AUV model with dynamics to this model, see Borges de Sousa et al. [59]). For our purposes, we can simplify the model further by disregarding the sway velocity η, as it is small in practice when compared to the surge velocity [15]. Then we have a model with three state variables and two input variables:

$$\begin{aligned}
\dot{x}_1 &= \mu \cos\psi + v_1(t, x_1(t), x_2(t)) \\
\dot{x}_2 &= \mu \sin\psi + v_2(t, x_1(t), x_2(t)) \\
\dot{\psi} &= \omega
\end{aligned}$$

This is a Dubins car model with the additional term for the ocean current velocity.

Note that there is not much to be gained by considering the angular velocity ω: for each position (x_1, x_2) we can consider the heading angle $\psi_\star$ which minimizes the cost from the initial state (x_1, x_2, ψ). If the vehicle has an initial yaw angle different from $\psi_\star$, its heading controller can regulate it to $\psi_\star$ in a matter of seconds, so that the vehicle can follow the optimal path from $(x_1, x_2, \psi_\star)$ with minimal effect on the total cost for most practical applications. For this reason we eliminate ω and consider ψ to be a control input, arriving at the reduced model

$$\begin{aligned}\dot{x}_1 &= \mu \cos\psi + v_1(t, x_1(t), x_2(t)) \\ \dot{x}_2 &= \mu \sin\psi + v_2(t, x_1(t), x_2(t))\end{aligned} \tag{5.1}$$

This model allows for trajectories which are not feasible for most ocean vehicles, since these typically can only track trajectories whose radius of curvature is above a given minimum value at every point. The consequences of this simplification will be discussed and studied in Chapter 6, but for now we remark that if the heading angle trajectory is absolutely continuous and satisfies $|\dot{\psi}| < \omega_{\max}$ almost everywhere, where $\omega_{\max}$ is the bound on the vehicle's angular velocity, then the trajectory is feasible for the Dubins model.

Assuming that the surge velocity can vary in the interval $[-r, r]$ where $r > 0$, we can rewrite (5.1) as

$$\dot{\boldsymbol{x}} = \boldsymbol{u} + \boldsymbol{v}(t, \boldsymbol{x}), \; \boldsymbol{u} \in \mathbf{B}_r$$

where $\boldsymbol{x} = (x_1, x_2)$ and similarly for $\boldsymbol{v}$ (here $\mathbf{B}_r$ denotes the closed ball of radius r centered at the origin). This is the model we will use throughout the thesis. It can also be written as a differential inclusion:

$$\dot{\boldsymbol{x}} \in \boldsymbol{v}(t, \boldsymbol{x}) + \mathbf{B}_r \tag{5.2}$$

which has a simple geometric interpretation, depicted in Figure 5.1.

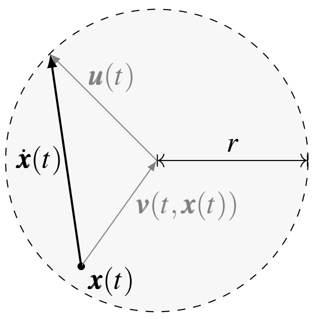

$\boldsymbol{u}(t)$
r
$\dot{\boldsymbol{x}}(t)$
$\boldsymbol{v}(t, \boldsymbol{x}(t))$
$\boldsymbol{x}(t)$

Figure 5.1: Geometric interpretation of the differential inclusion (5.2)

5.2 The base problem

We can now apply the techniques outlined in 2.3.2.2 to the problem described in Chapter 3.

As the ocean current is time-varying, we augment the state variable in order to obtain a time-invariant model:

$$\begin{aligned}\dot{\boldsymbol{x}}(t) &= \boldsymbol{u}(t) + \boldsymbol{v}(\tau(t), \boldsymbol{x}(t)) \\ \dot{\tau}(t) &= 1\end{aligned}$$

The initial conditions can be specified at any point in time, so we assume they are always given at $t = 0$, that is, $\boldsymbol{x}(0)$ and $\tau(0)$ are the deployment position and time, respectively. Hence $\tau(t)$ will indicate the actual mission time while t is time relative to the deployment time in the same units.

The target set Ω is a subset of $\mathbf{R}^2$, while the state space is $\mathbf{R}^3$. It is thus necessary to augment the set Ω to include the time variable. The simplest way to achieve this is to consider the extended target set Ω' as the product $\Omega \times \mathbf{R}$, which is the same as specifying that the target set can be reached at any time. Other possibilities include $\Omega \times [\tau_0, \infty]$ or $\Omega \times [\tau_0, \tau_1]$, or an arbitrary $\Omega' \subset \mathbf{R}^3$ which describes a more complex relationship between possible arrival times and positions (e.g., different arrival positions for different intervals of arrival times). Below we derive the HJBE for the case $\Omega' = \Omega \times \mathbf{R}$.

The cost function J in Equation 3.1 can be written as a function of the initial conditions and the control:

$$J(\boldsymbol{x}_0, \tau_0, \boldsymbol{u}) = q(\boldsymbol{x}(T; \boldsymbol{x}_0, \tau_0, \boldsymbol{u})) + \int_0^T g(\boldsymbol{x}(t; \boldsymbol{x}_0, \tau_0, \boldsymbol{u}))\mathrm{d}t$$

where $\boldsymbol{x}(t; \boldsymbol{x}_0, \tau_0, \boldsymbol{u})$ is the horizontal position of the vehicle t units of time after starting from $\boldsymbol{x}_0$ at time τ_0, when the control is $\boldsymbol{u}$, and T is the time taken to reach Ω:

$$T = T(\boldsymbol{x}_0, \tau_0, \boldsymbol{u}) = \inf\{t \geq 0 \mid \boldsymbol{x}(t; \boldsymbol{x}_0, \tau_0, \boldsymbol{u}) \in \Omega\}$$

Note that J is finite if and only if T is finite.

The value function is defined in the usual way:

$$V(\tau_0, \boldsymbol{x}_0) = \inf\{J(\boldsymbol{x}_0, \tau_0, \boldsymbol{u}) \mid \boldsymbol{u} \in \mathscr{U}(\boldsymbol{x}_0, \tau_0)\}$$

where $\mathscr{U}(\boldsymbol{x}_0, \tau_0)$ is the set of measurable $\boldsymbol{u} : \mathbf{R}_{\geq 0} \to \mathbf{B}_r$ such that $T(\boldsymbol{x}_0, \tau_0, \boldsymbol{u})$ is finite. We now apply (2.9), denoting partial derivatives by subscripts:

$$0 = \inf_{\boldsymbol{w} \in \mathbf{B}_r} \{g(\boldsymbol{x}) + V_{\boldsymbol{x}}(\tau, \boldsymbol{x}) \cdot (\boldsymbol{w} + \boldsymbol{v}(\tau, \boldsymbol{x})) + V_\tau(\tau, \boldsymbol{x})\} \tag{5.3}$$

$$\iff g(\boldsymbol{x}) = -\inf_{\boldsymbol{w} \in \mathbf{B}_r} \{V_{\boldsymbol{x}}(\tau, \boldsymbol{x}) \cdot \boldsymbol{w}\} - (V_{\boldsymbol{x}}(\tau, \boldsymbol{x}) \cdot \boldsymbol{v}(\tau, \boldsymbol{x}) + V_\tau(\tau, \boldsymbol{x})). \tag{5.4}$$

The minimizing $\boldsymbol{w}$ is

$$\boldsymbol{w}^\star = -r\frac{V_{\boldsymbol{x}}(\tau, \boldsymbol{x})}{|V_{\boldsymbol{x}}(\tau, \boldsymbol{x})|}$$

(recall that r is the maximum speed of the vehicle). Substituting in (5.3), we arrive at the Hamilton-Jacobi equation

$$r|V_{\boldsymbol{x}}(\tau, \boldsymbol{x})| - (V_{\boldsymbol{x}}(\tau, \boldsymbol{x}) \cdot \boldsymbol{v}(\tau, \boldsymbol{x}) + V_\tau(\tau, \boldsymbol{x})) = g(\boldsymbol{x}). \tag{5.5}$$

Since for each $\boldsymbol{x}_\Omega \in \Omega$ one has

$$J(\boldsymbol{x}_\Omega, \tau, \boldsymbol{u}) = q(\boldsymbol{x}_\Omega),$$

V must satisfy the boundary conditions

$$V(\tau, \boldsymbol{x}) = q(\boldsymbol{x}),\ \boldsymbol{x} \in \Omega.$$

If the target set is specified directly as a subset Ω' of $\mathbf{R}^3$, the partial differential equation is unaffected, but the boundary condition is then

$$V(\tau, \boldsymbol{x}) = q(\boldsymbol{x}),\ (\boldsymbol{x}, \tau) \in \Omega'.$$

Assuming we have computed V, we can use it to compute an optimal feedback law:

$$\boldsymbol{u}(\tau, \boldsymbol{x}) = -r \frac{V_{\boldsymbol{x}}(\tau, \boldsymbol{x})}{|V_{\boldsymbol{x}}(\tau, \boldsymbol{x})|} \tag{5.6}$$

Then we have two ways of using this feedback law. The first is to simply plug it in the motion model, so that we get an autonomous nonlinear differential equation:

$$\dot{\boldsymbol{x}}(\tau) = \boldsymbol{v}(\tau, \boldsymbol{x}(\tau)) - r \frac{V_{\boldsymbol{x}}(\tau, \boldsymbol{x}(\tau))}{|V_{\boldsymbol{x}}(\tau, \boldsymbol{x}(\tau))|} \tag{5.7}$$

We can solve this differential equation with any initial condition $\boldsymbol{x}(\tau_0) = \boldsymbol{x}_0$ to obtain an optimal trajectory with that initial condition. Such a trajectory can then be used as a reference for the lower level level control loops in the vehicle's onboard navigation and control software. Alternatively, the feedback law (5.6) can be used directly as a heading controller. Since we are concerned with trajectory generation, our focus is on the first option.

Note that the components of the cost q and g can depend on τ with no changes to the resulting equations; for simplicity we considered them to be time-invariant.

5.2.1 Incorporating obstacles and constraints

When operating in areas such as estuarine regions, the vehicle will have to be able to navigate around the geography and bathymetry of the region to prevent it from running aground. This must be included directly in the formulation because the optimal unobstructed path can be quite different from the optimal path which takes into account the geography and bathymetry constraints. In addition, it may be desirable to set other constraints such as no-go zones and to-avoid zones. These constraints can be added in one of two ways, as hard constraints or soft constraints.

Hard constraints are adequate for the geography constraints or no-go zones. If those points are represented by a set $\mathscr{K}_h$, we impose an additional boundary condition:

$$V(\tau, \boldsymbol{x}) = M,\ \boldsymbol{x} \in \mathscr{K}_h,$$

where M is large in the sense that any trajectory having a cost larger than M is considered infeasible in practice. This is equivalent to adding the points in $\mathscr{K}_h$ to the target set with a large value of q. For instance, if the cost function is the time taken to reach the target, M can be any number larger than the length of the mission time window. This way, any trajectory which contains a point in $\mathscr{K}_h$ will have a cost of at least M, and the target is considered unreachable from the corresponding initial condition.

Soft constraints are adequate for to-avoid zones. If we want to add a soft constraint for the set $\mathscr{K}_s$, we can simply add a multiple of the indicator function of $\mathscr{K}_s$

$$\mathbf{1}_{\mathscr{K}_s}(\boldsymbol{x}) = \begin{cases} 1, & \boldsymbol{x} \in \mathscr{K}_s \\ 0, & \boldsymbol{x} \notin \mathscr{K}_s \end{cases}$$

to the cost component g. Trajectories crossing $\mathscr{K}_s$ will then pay an additional cost proportional to the amount of time spent inside $\mathscr{K}_s$.

We observe that this problem is not amenable to the application of exact penalization methods such as those described in Clarke et al. [11].

5.3 Planning with logic-based constraints

In this section we extend the previous result to the generic n-step problem described in Section 3.2.

We begin by relaxing the constraints $\boldsymbol{x}^i(\tau^i) \in \Omega_i$. These can be included in the functions γ_i, by setting γ_i to a high value outside of the set Ω_i.

Given two points $\boldsymbol{x}_i, \boldsymbol{x}_{i+1} \in \mathbf{R}^2$ and two time points t_i, t_{i+1}, either there is no trajectory $\boldsymbol{x}(\cdot)$ of the vehicle satisfying $\boldsymbol{x}(t_i) = \boldsymbol{x}_i$, $\boldsymbol{x}(t_{i+1}) = \boldsymbol{x}_{i+1}$ or there is a single such trajectory which minimizes the integral cost

$$\int_{t_i}^{t_{i+1}} g_i(\boldsymbol{x}(t))\mathrm{d}t.$$

Consider an optimal solution to the problem, $(\boldsymbol{x}^0, \ldots, \boldsymbol{x}^{n-1})$. Then $\boldsymbol{x}^i$ must be the trajectory connecting $\boldsymbol{x}^i(\tau^i)$ and $\boldsymbol{x}^{i+1}(\tau^{i+1})$ which minimizes the integral cost; for if not then the cost J_{n-1} can be reduced by replacing $\boldsymbol{x}^i$ with the optimal trajectory connecting those two points, and this would imply that the solution is not optimal after all.

Hence we can reduce the problem to that of planning the initial positions of the vehicle at each stage, $\boldsymbol{\xi}_i = \boldsymbol{x}^i(\tau_i)$, and the associated times τ_i. Once the $\boldsymbol{\xi}_i$ and τ_i have been found, the solution to the original problem can be recovered by solving $n-1$ optimal control problems for the trajectories connecting those positions.

This is similar to the class of problems considered in Alton and Mitchell [4], and here we adapt their approach. We begin by defining

$$\delta_i(\tau', \boldsymbol{\xi}', \tau, \boldsymbol{\xi}) = \inf\left\{ \int_{\tau}^{\tau'} g_i(\boldsymbol{x}(t))\mathrm{d}t \;\middle|\; \boldsymbol{x}(\tau) = \boldsymbol{\xi},\, \boldsymbol{x}(\tau') = \boldsymbol{\xi}' \right\}.$$

Then the cost functions for each stage can be expressed in terms of the τ_i and $\boldsymbol{\xi}_i$:

$$\begin{aligned} J_0(\tau_0, \boldsymbol{\xi}_0) &= \gamma_0(\tau_0, \boldsymbol{\xi}_0) \\ J_{i+1}(\tau_{i+1}, \boldsymbol{\xi}_{i+1}, \dots, \tau_0, \boldsymbol{\xi}_0) &= J_i(\tau_i, \boldsymbol{\xi}_i, \dots, \tau_0, \boldsymbol{\xi}_0) + \delta_i(\tau_{i+1}, \boldsymbol{\xi}_{i+1}, \tau_i, \boldsymbol{\xi}_i) + \gamma_{i+1}(\tau_{i+1}, \boldsymbol{\xi}_{i+1}) \end{aligned}$$

We can then define the functions V_i, W_i as

$$\begin{aligned} V_i(\tau, \boldsymbol{\xi}) &= \min_{(\tau_{i-1}, \boldsymbol{\xi}_{i-1}, \dots, \tau_0, \boldsymbol{\xi}_0)} J_i(\tau, \boldsymbol{\xi}, \dots, \tau_0, \boldsymbol{\xi}_0),\ i = 0, \dots, n-1 \\ W_i(\tau, \boldsymbol{\xi}) &= \min_{(\tau_i, \boldsymbol{\xi}_i, \dots, \tau_0, \boldsymbol{\xi}_0)} \{J_i(\tau_i, \boldsymbol{\xi}_i, \dots, \tau_0, \boldsymbol{\xi}_0) + \delta_i(\tau, \boldsymbol{\xi}, \tau_i, \boldsymbol{\xi}_i)\},\ i = 0, \dots, n-2. \end{aligned}$$

The value $V_i(\tau, \boldsymbol{\xi})$ is the optimal cost up to stage i if $\tau_i = \tau$ and $\boldsymbol{x}^i(\tau_i) = \boldsymbol{\xi}$, i.e., if the vehicle starts stage i of the mission at time τ and at position $\boldsymbol{\xi}$. The value of $W_i(\tau, \boldsymbol{\xi})$ is the optimal cost of performing stages $0, \dots, i-1$ and then travelling to $\boldsymbol{\xi}$.

Obviously we have $V_0 = \gamma_0$, and so

$$W_0(\tau, \boldsymbol{\xi}) = \min_{\tau', \boldsymbol{\xi}'} \{\gamma_0(\tau', \boldsymbol{\xi}') + \delta_0(\tau, \boldsymbol{\xi}, \tau', \boldsymbol{\xi}')\} = W_0(\tau, \boldsymbol{\xi}) = \min_{\tau', \boldsymbol{\xi}'} \{V_0(\tau', \boldsymbol{\xi}') + \delta_0(\tau, \boldsymbol{\xi}, \tau', \boldsymbol{\xi}')\}.$$

This relation actually holds for all i:

$$\begin{aligned} W_i(\tau, \boldsymbol{\xi}) &= \min_{(\tau_i, \boldsymbol{\xi}_i, \dots, \tau_0, \boldsymbol{\xi}_0)} \{J_i(\tau_i, \boldsymbol{\xi}_i, \dots, \tau_0, \boldsymbol{\xi}_0) + \delta_i(\tau, \boldsymbol{\xi}, \tau_i, \boldsymbol{\xi}_i)\} \\ &= \min_{(\tau_i, \boldsymbol{\xi}_i)} \left\{ \min_{(\tau_{i-1}, \boldsymbol{\xi}_{i-1}, \dots)} J_i(\tau_i, \boldsymbol{\xi}_i, \dots) + \delta_i(\tau, \boldsymbol{\xi}, \tau_i, \boldsymbol{\xi}_i) \right\} \\ &= \min_{(\tau_i, \boldsymbol{\xi}_i)} \{V_i(\tau_i, \boldsymbol{\xi}_i) + \delta_i(\tau, \boldsymbol{\xi}, \tau_i, \boldsymbol{\xi}_i)\}. \end{aligned}$$

There is also a simple formula linking V_{i+1} and W_i:

$$\begin{aligned} V_{i+1}(\tau, \boldsymbol{\xi}) &= \min_{(\tau_i, \boldsymbol{\xi}_i, \dots)} J_{i+1}(\tau, \boldsymbol{\xi}, \dots) \\ &= \min_{(\tau_i, \boldsymbol{\xi}_i, \dots)} \{J_i(\tau_i, \boldsymbol{\xi}_i, \dots) + \delta_i(\tau, \boldsymbol{\xi}, \tau_i, \boldsymbol{\xi}_i)\} + \gamma_{i+1}(\tau, \boldsymbol{\xi}) \\ &= W_i(\tau, \boldsymbol{\xi}) + \gamma_{i+1}(\tau, \boldsymbol{\xi}). \end{aligned}$$

Assuming for the moment that we can compute δ_i, using these relations we can compute all the V_i and W_i (since V_0 is known). We now show that the optimal solution can be calculated using V_i and W_i. Let $(\tau_{n-1}, \boldsymbol{\xi}_{n-1}, \dots, \tau_0, \boldsymbol{\xi}_0)$ be an optimal solution. Suppose that $V_{n-1}(\tau_{n-1}, \boldsymbol{\xi}_{n-1}) > V_{n-1}(\tau, \boldsymbol{\xi})$ for some $(\tau, \boldsymbol{\xi})$. Then

$$\begin{aligned} V_{n-1}(\tau, \boldsymbol{\xi}) &\geq J_{n-1}(\tau, \boldsymbol{\xi}, \tau_{n-2}, \boldsymbol{\xi}_{n-2}, \dots, \tau_0, \boldsymbol{\xi}_0) \\ &\geq J_{n-1}(\tau_{n-1}, \boldsymbol{\xi}_{n-1}, \tau_{n-2}, \boldsymbol{\xi}_{n-2}, \dots, \tau_0, \boldsymbol{\xi}_0) \end{aligned}$$

implying that

$$V_{n-1}(\tau_{n-1},\boldsymbol{\xi}_{n-1}) > J_{n-1}(\tau_{n-1},\boldsymbol{\xi}_{n-1},\tau_{n-2},\boldsymbol{\xi}_{n-2},\dots,\tau_0,\boldsymbol{\xi}_0) \geq J_{n-1}(\tau_{n-1},\boldsymbol{\xi}_{n-1},\tau'_{n-2},\boldsymbol{\xi}'_{n-2},\dots,\tau'_0,\boldsymbol{\xi}'_0)$$

for any $(\tau'_{n-2},\boldsymbol{\xi}'_{n-2},\dots,\tau'_0,\boldsymbol{\xi}'_0)$, which contradicts the definition of V_{n-1}. Hence

$$(\tau_{n-1},\boldsymbol{\xi}_{n-1}) \in \arg\min_{(\tau,\boldsymbol{\xi})} V_{n-1}(\tau,\boldsymbol{\xi}). \tag{5.8}$$

Now suppose that

$$V_{n-2}(\tau_{n-2},\boldsymbol{\xi}_{n-2}) + \delta_{n-2}(\tau_{n-1},\boldsymbol{\xi}_{n-1},\tau_{n-2},\boldsymbol{\xi}_{n-2}) > V_{n-2}(\tau,\boldsymbol{\xi}) + \delta_{n-2}(\tau_{n-1},\boldsymbol{\xi}_{n-1},\tau,\boldsymbol{\xi}).$$

Then

$$\begin{aligned}
&V_{n-2}(\tau_{n-2},\boldsymbol{\xi}_{n-2}) + \delta_{n-2}(\tau_{n-1},\boldsymbol{\xi}_{n-1},\tau_{n-2},\boldsymbol{\xi}_{n-2}) + \gamma_{n-1}(\tau_{n-1},\boldsymbol{\xi}_{n-1}) > \\
&\qquad\qquad V_{n-2}(\tau,\boldsymbol{\xi}) + \delta_{n-2}(\tau_{n-1},\boldsymbol{\xi}_{n-1},\tau,\boldsymbol{\xi}) + \gamma_{n-1}(\tau_{n-1},\boldsymbol{\xi}_{n-1}) \\
&\min_{(\tau'_{n-3},\boldsymbol{\xi}'_{n-3},\dots,\tau'_0,\boldsymbol{\xi}'_0)} J_{n-1}(\tau_{n-1},\boldsymbol{\xi}_{n-1},\tau_{n-2},\boldsymbol{\xi}_{n-2},\tau'_{n-3},\boldsymbol{\xi}'_{n-3},\dots,\tau'_0,\boldsymbol{\xi}'_0) > \\
&\qquad\qquad \min_{(\tau'_{n-3},\boldsymbol{\xi}'_{n-3},\dots,\tau'_0,\boldsymbol{\xi}'_0)} J_{n-1}(\tau_{n-1},\boldsymbol{\xi}_{n-1},\tau,\boldsymbol{\xi},\tau'_{n-3},\boldsymbol{\xi}'_{n-3},\dots,\tau'_0,\boldsymbol{\xi}'_0) \\
&\qquad J_{n-1}(\tau_{n-1},\boldsymbol{\xi}_{n-1},\tau_{n-2},\boldsymbol{\xi}_{n-2},\tau_{n-3},\boldsymbol{\xi}_{n-3},\dots,\tau_0,\boldsymbol{\xi}_0) > \\
&\qquad\qquad \min_{(\tau'_{n-3},\boldsymbol{\xi}'_{n-3},\dots,\tau'_0,\boldsymbol{\xi}'_0)} J_{n-1}(\tau_{n-1},\boldsymbol{\xi}_{n-1},\tau,\boldsymbol{\xi},\tau'_{n-3},\boldsymbol{\xi}'_{n-3},\dots,\tau'_0,\boldsymbol{\xi}'_0).
\end{aligned}$$

The last inequality contradicts the fact that the $(\tau_i,\boldsymbol{\xi}_i)$ constitute an optimal solution. Hence

$$(\tau_{n-2},\boldsymbol{\xi}_{n-2}) \in \arg\min_{(\tau,\boldsymbol{\xi})} \left\{ V_{n-2}(\tau,\boldsymbol{\xi}) + \delta_{n-2}(\tau_{n-1},\boldsymbol{\xi}_{n-1},\tau,\boldsymbol{\xi}) \right\}.$$

By the same argument, we conclude that

$$(\tau_i,\boldsymbol{\xi}_i) \in \arg\min_{(\tau,\boldsymbol{\xi})} \left\{ V_i(\tau,\boldsymbol{\xi}) + \delta_i(\tau_{i+1},\boldsymbol{\xi}_{i+1},\tau,\boldsymbol{\xi}) \right\}. \tag{5.9}$$

for all i.

Thus, any optimal solution satisfies (5.8) and (5.9). To show that the converse also holds, consider an arbitrary tuple $(\tau_{n-1},\boldsymbol{\xi}_{n-1},\dots,\tau_0,\boldsymbol{\xi}_0)$. Then

$$\begin{aligned}
J_{n-1}(\tau_{n-1},\boldsymbol{\xi}_{n-1},\dots,\tau_0,\boldsymbol{\xi}_0) &= \gamma_{n-1}(\tau_{n-1},\boldsymbol{\xi}_{n-1})+\sum_{i=0}^{n-2}\gamma_i(\tau_i,\boldsymbol{\xi}_i)+\delta_i(\tau_{i+1},\boldsymbol{\xi}_{i+1},\tau_i,\boldsymbol{\xi}_i)\\
&\geq \min_{\tau_0',\boldsymbol{\xi}_0'}\left\{\gamma_{n-1}(\tau_{n-1},\boldsymbol{\xi}_{n-1})+\sum_{i=0}^{n-2}\gamma_i(\tau_i,\boldsymbol{\xi}_i)+\delta_i(\tau_{i+1},\boldsymbol{\xi}_{i+1},\tau_i,\boldsymbol{\xi}_i)\right\}\\
&= \gamma_{n-1}(\tau_{n-1},\boldsymbol{\xi}_{n-1})+W_0(\tau_1,\boldsymbol{\xi}_1)+\sum_{i=1}^{n-2}\gamma_i(\tau_i,\boldsymbol{\xi}_i)+\delta_i(\tau_{i+1},\boldsymbol{\xi}_{i+1},\tau_i,\boldsymbol{\xi}_i)\\
&= \gamma_{n-1}(\tau_{n-1},\boldsymbol{\xi}_{n-1})+V_1(\tau_1,\boldsymbol{\xi}_1)+\delta_1(\tau_2,\boldsymbol{\xi}_2,\tau_1,\boldsymbol{\xi}_1)+\sum_{i=2}^{n-2}\gamma_i(\tau_i,\boldsymbol{\xi}_i)+\delta_i(\tau_{i+1},\boldsymbol{\xi}_{i+1},\tau_i,\boldsymbol{\xi}_i)\\
&\geq \gamma_{n-1}(\tau_{n-1},\boldsymbol{\xi}_{n-1})+W_1(\tau_2,\boldsymbol{\xi}_2)+\sum_{i=2}^{n-2}\gamma_i(\tau_i,\boldsymbol{\xi}_i)+\delta_i(\tau_{i+1},\boldsymbol{\xi}_{i+1},\tau_i,\boldsymbol{\xi}_i)\\
&= \gamma_{n-1}(\tau_{n-1},\boldsymbol{\xi}_{n-1})+V_2(\tau_2,\boldsymbol{\xi}_2)+\delta_2(\tau_3,\boldsymbol{\xi}_3,\tau_2,\boldsymbol{\xi}_2)+\sum_{i=3}^{n-2}\gamma_i(\tau_i,\boldsymbol{\xi}_i)+\delta_i(\tau_{i+1},\boldsymbol{\xi}_{i+1},\tau_i,\boldsymbol{\xi}_i)\\
&\geq \dots\\
&\geq V_{n-1}(\tau_{n-1},\boldsymbol{\xi}_{n-1})\\
&\geq \min_{\tau,\boldsymbol{\xi}} V_{n-1}(\tau,\boldsymbol{\xi}).
\end{aligned}$$

If $(\tau_{n-1},\boldsymbol{\xi}_{n-1},\dots,\tau_0,\boldsymbol{\xi}_0)$ satisfies (5.9) and (5.8), then each of the inequalities is an equality, so J_{n-1} is minimized.

Hence, the optimal solutions can be recovered from W_i and V_i. Thus, all that is left is a way to efficiently compute δ_i. Using the motion model, we can derive a partial differential equation which allows us to effectively compute W_i, V_i and δ_i simultaneously. As in the previous problem, we consider that the vehicle's trajectories are solutions of the differential equation

$$\dot{\boldsymbol{x}}(t)=\boldsymbol{u}(t)+\boldsymbol{v}(t,\boldsymbol{x}(t))$$

with $\boldsymbol{u}\in\mathbf{B}_r$. Let $U[a,b]$ be the set of measurable control functions $\boldsymbol{u}:[a,b]\to\mathbf{B}_r$. For each $\boldsymbol{u}\in U[\tau,\tau']$ denote by $\boldsymbol{x}(t;\tau,\boldsymbol{\xi},\boldsymbol{u})$ the value at time t of the solution of the differential equation which satisfies $\boldsymbol{x}(\tau)=\boldsymbol{\xi}$, where $t\in[\tau,\tau']$. We can write δ_i as

$$\delta_i(\tau',\boldsymbol{\xi}',\tau,\boldsymbol{\xi})=\inf\left\{\int_\tau^\tau g_i(\boldsymbol{x}(t;\tau,\boldsymbol{\xi},\boldsymbol{u}))\mathrm{d}t\;\middle|\;\boldsymbol{x}(\tau';t,\boldsymbol{\xi},\boldsymbol{u})=\boldsymbol{\xi}',\ \boldsymbol{u}\in U[\tau,\tau']\right\},$$

so that W_i can be expressed as

$$W_i(\tau,\boldsymbol{\xi})=\min_{(\tau',\boldsymbol{\xi}')}\left\{V_i(\tau',\boldsymbol{\xi}')+\delta_i(\tau,\boldsymbol{\xi},\tau',\boldsymbol{\xi}')\right\}=\inf_{\tau',\boldsymbol{u}\in U[\tau',\tau]}\left\{V_i(\tau',\boldsymbol{x}(\tau';\tau,\boldsymbol{\xi},\boldsymbol{u}))+\int_\tau^{\tau'}g_i(\boldsymbol{x}(t;\tau,\boldsymbol{\xi},\boldsymbol{u}))\mathrm{d}t\right\}.$$

The function W_i can hence be seen as a value function for an optimal starting problem: defining the cost function

$$H_i(\tau,\boldsymbol{\xi},\tau',\boldsymbol{u}) = V_i(\tau',\boldsymbol{x}(\tau';\tau,\boldsymbol{\xi},\boldsymbol{u})) + \int_{\tau}^{\tau'} g_i(\boldsymbol{x}(t;\tau,\boldsymbol{\xi},\boldsymbol{u}))\mathrm{d}t,$$

we have

$$W_i(\tau,\boldsymbol{\xi}) = \inf_{\tau',\boldsymbol{u}\in U[\tau',\tau]} H_i(\tau,\boldsymbol{\xi},\tau',\boldsymbol{u})$$

This is a type of optimal control problem where, besides the usual optimization of the control, the initial time is also optimized. We adapt the proofs in Bardi and Capuzzo-Dolcetta [6], who developed a dynamic programming approach for a time-invariant optimal stopping problem.

We begin by noting that $W_i \leq V_i$, since

$$W_i(\tau,\boldsymbol{\xi}) \leq H_i(\tau,\boldsymbol{\xi},\tau,\mathbf{0}) = V_i(\tau,\boldsymbol{\xi})$$

Now fix some $\boldsymbol{\xi}$, τ, $s \leq \tau$ and $\boldsymbol{u} \in U[s,\tau]$. Let $\varepsilon > 0$. By definition there exist τ^ε and $\boldsymbol{u}^\varepsilon \in U[\tau^\varepsilon, s]$ such that

$$H_i(s,\boldsymbol{x}(s;\tau,\boldsymbol{\xi},\boldsymbol{u}),\tau^\varepsilon,\boldsymbol{u}^\varepsilon) \leq W_i(s,\boldsymbol{x}(s;\tau,\boldsymbol{\xi},\boldsymbol{u})) + \varepsilon.$$

Letting $\bar{\boldsymbol{u}}$ be the concatenation of $\boldsymbol{u}^\varepsilon$ and $\boldsymbol{u}$,

$$\begin{aligned}
W_i(t,\boldsymbol{\xi}) &\leq H_i(\tau,\boldsymbol{\xi},\tau^\varepsilon,\bar{\boldsymbol{u}}) \\
&= V_i(\tau^\varepsilon,\boldsymbol{x}(\tau^\varepsilon;s,\boldsymbol{x}(s;\tau,\boldsymbol{\xi},\boldsymbol{u}),\boldsymbol{u}^\varepsilon)) + \int_{\tau^\varepsilon}^{s} g_i(\boldsymbol{x}(t;s,\boldsymbol{x}(s;\tau,\boldsymbol{\xi},\boldsymbol{u}),\boldsymbol{u}^\varepsilon))\mathrm{d}t + \int_{s}^{\tau} g_i(\boldsymbol{x}(t;\tau,\boldsymbol{\xi},\boldsymbol{u}))\mathrm{d}t \\
&= H_i(s,\boldsymbol{x}(s;\tau,\boldsymbol{\xi},\boldsymbol{u}),\tau^\varepsilon,\boldsymbol{u}^\varepsilon) + \int_{s}^{\tau} g_i(\boldsymbol{x}(t;\tau,\boldsymbol{\xi},\boldsymbol{u}))\mathrm{d}t \\
&\leq \varepsilon + W_i(s,\boldsymbol{x}(s;\tau,\boldsymbol{\xi},\boldsymbol{u})) + \int_{s}^{\tau} g_i(\boldsymbol{x}(t;\tau,\boldsymbol{\xi},\boldsymbol{u}))\mathrm{d}t
\end{aligned}$$

Since ε is arbitrary, we conclude that

$$W_i(\tau,\boldsymbol{\xi}) \leq \int_{s}^{\tau} g_i(\boldsymbol{x}(t;\tau,\boldsymbol{\xi},\boldsymbol{u}))\mathrm{d}t + W_i(s,\boldsymbol{x}(s;\tau,\boldsymbol{\xi},\boldsymbol{u})) \tag{5.10}$$

for any $s \leq \tau$ and $\boldsymbol{u} \in U[s,\tau]$. Note that this is a weak form of the dynamic programming principle, and all we have to show now is that equality holds when $\boldsymbol{u}$ is optimal. If $W_i(\tau,\boldsymbol{\xi}) = V_i(\tau,\boldsymbol{\xi})$, then the optimal starting time is τ and the optimal control is the empty function. Hence, we consider τ and $\boldsymbol{\xi}$ such that $W_i(\tau,\boldsymbol{\xi}) < V_i(\tau,\boldsymbol{\xi})$.

Let $\{(\tau_n,\boldsymbol{u}_n)\}_{n=1}^{\infty}$ be a sequence such that $\tau_n \leq \tau$ and

$$\lim_{n\to\infty} H_i(\tau,\boldsymbol{\xi},\tau_n,\boldsymbol{u}_n) = W_i(\tau,\boldsymbol{\xi}).$$

Bardi and Capuzzo-Dolcetta [6] show that if V_i is bounded and uniformly continuous, then so is W_i. Hence we assume that V_0 (which we are free to choose) is bounded and uniformly continuous, so that all the V_i and W_i are too. Therefore, by uniform continuity we can find a function $\omega : \mathbf{R}_{\geq 0} \to \mathbf{R}_{\geq 0}$

such that

$$\omega(0) = \lim_{t\to 0}\omega(t) = 0$$
$$\left|V_i(\tau_0,\boldsymbol{\xi}_0) - V_i(\tau',\boldsymbol{\xi}')\right| \le \omega(\left|\tau_0-\tau'\right| + \left|\boldsymbol{\xi}_0 - \boldsymbol{\xi}'\right|).$$

Then, setting $\mu_n = H_i(\tau,\boldsymbol{\xi},\tau_n,\boldsymbol{u}_n) - W_i(\tau,\boldsymbol{\xi})$, one has

$$\begin{aligned} W_i(\tau,\boldsymbol{\xi}) + \mu_n &= V_i(\tau_n,\boldsymbol{x}(\tau_n;\tau,\boldsymbol{\xi},\boldsymbol{u}_n)) + \int_{\tau_n}^{\tau} g_i(\boldsymbol{x}(t;\tau,\boldsymbol{\xi},\boldsymbol{u}_n))\mathrm{d}t \\ &\ge V_i(\tau_n,\boldsymbol{x}(\tau_n;\tau,\boldsymbol{\xi},\boldsymbol{u}_n)) \\ &\ge V_i(t,\boldsymbol{\xi}) - \omega(|\tau-\tau_n| + |\boldsymbol{\xi} - \boldsymbol{x}(\tau_n;\tau,\boldsymbol{\xi},\boldsymbol{u}_n)|). \end{aligned}$$

If some subsequence of τ_n tends to τ, then taking a limit along that subsequence in each side of the above inequality yields $W_i(\tau,\boldsymbol{\xi}) \ge V_i(\tau,\boldsymbol{\xi})$, contradicting our initial assumption. Hence we can find some $\tau_0 < \tau$ such that $\tau_n \le \tau_0$ for all n sufficiently large. For $s \in [\tau_0,\tau]$,

$$\begin{aligned} H_i(\tau,\boldsymbol{\xi},\tau_n,\boldsymbol{u}_n) &= H_i(s,\boldsymbol{x}(s;\tau,\boldsymbol{\xi},\boldsymbol{u}_n),\tau_n,\boldsymbol{u}_n) + \int_s^{\tau} g_i(\boldsymbol{x}(t;\tau,\boldsymbol{\xi},\boldsymbol{u}_n))\mathrm{d}t \\ &\ge W_i(s,\boldsymbol{x}(s;\tau,\boldsymbol{\xi},\boldsymbol{u}_n)) + \int_s^{\tau} g_i(\boldsymbol{x}(t;\tau,\boldsymbol{\xi},\boldsymbol{u}_n))\mathrm{d}t \\ &\ge \inf_{\boldsymbol{u}\in U[s,\tau]}\left\{W_i(s,\boldsymbol{x}(s;\tau,\boldsymbol{\xi},\boldsymbol{u})) + \int_s^{\tau} g_i(\boldsymbol{x}(t;\tau,\boldsymbol{\xi},\boldsymbol{u}))\mathrm{d}t\right\} \end{aligned}$$

taking a limit as $n\to\infty$ and using (5.10), we conclude

$$W_i(\tau,\boldsymbol{\xi}) = \inf_{\boldsymbol{u}\in U[s,\tau]}\left\{W_i(s,\boldsymbol{x}(s;\tau,\boldsymbol{\xi},\boldsymbol{u})) + \int_s^{\tau} g_i(\boldsymbol{x}(t;\tau,\boldsymbol{\xi},\boldsymbol{u}))\mathrm{d}t\right\}.$$

We can obtain a PDE form of this dynamic programming principle via the usual procedure. We have for $h \ge 0$

$$0 \le \inf_{\boldsymbol{u}\in U[\tau-h,\tau]}\left\{W_i(\tau-h,\boldsymbol{x}(\tau-h;\tau,\boldsymbol{\xi},\boldsymbol{u})) - W_i(\tau,\boldsymbol{\xi}) + \int_{\tau-h}^{\tau} g_i(\boldsymbol{x}(t;\tau,\boldsymbol{\xi},\boldsymbol{u}))\mathrm{d}t\right\}$$

with equality if $W_i < V_i$. Dividing by h and taking a limit as $h\to 0$,

$$0 \le \min_{\boldsymbol{u}\in\mathbf{B}_r}\left\{g_i(\boldsymbol{\xi}) - \nabla_{\boldsymbol{\xi}} W_i(\tau,\boldsymbol{\xi})\cdot(\boldsymbol{u} + \boldsymbol{v}(\tau,\boldsymbol{\xi})) - \frac{\partial W_i}{\partial\tau}(\tau,\boldsymbol{\xi})\right\}.$$

Therefore we have

$$\begin{aligned} &0 \ge W_i(\tau,\boldsymbol{\xi}) - V_i(\tau,\boldsymbol{\xi}) \\ &0 \ge \max_{\boldsymbol{u}\in\mathbf{B}_r}\left\{\nabla_{\boldsymbol{\xi}} W_i(\tau,\boldsymbol{\xi})\cdot(\boldsymbol{u} + \boldsymbol{v}(\tau,\boldsymbol{\xi})) + \frac{\partial W_i}{\partial\tau}(\tau,\boldsymbol{\xi}) - g_i(\boldsymbol{\xi})\right\} \end{aligned}$$

and at least one of these is equal to zero. This can be summarized as

$$0 = \max\left\{ W_i(\tau,\boldsymbol{\xi}) - V_i(\tau,\boldsymbol{\xi}), \max_{\boldsymbol{u}\in\mathbf{B}_r}\left\{ \nabla_{\boldsymbol{\xi}} W_i(\tau,\boldsymbol{\xi})\cdot(\boldsymbol{u}+\boldsymbol{v}(\tau,\boldsymbol{\xi})) + \frac{\partial W_i}{\partial \tau}(\tau,\boldsymbol{\xi}) - g_i(\boldsymbol{\xi}) \right\}\right\}.$$

The maximizer is $\boldsymbol{u} = r\frac{\nabla_{\boldsymbol{\xi}} W_i(\tau,\boldsymbol{\xi})}{|\nabla_{\boldsymbol{\xi}} W_i(\tau,\boldsymbol{\xi})|}$, so this becomes

$$0 = \max\left\{ W_i(\tau,\boldsymbol{\xi}) - V_i(\tau,\boldsymbol{\xi}), \nabla_{\boldsymbol{\xi}} W_i(\tau,\boldsymbol{\xi})\cdot\boldsymbol{v}(\tau,\boldsymbol{\xi}) + r\left|\nabla_{\boldsymbol{\xi}} W_i(\tau,\boldsymbol{\xi})\right| + \frac{\partial W_i}{\partial \tau}(\tau,\boldsymbol{\xi}) - g_i(\boldsymbol{\xi}) \right\}. \tag{5.11}$$

This type of equation is known as a variational inequality. The PDE term, however, is almost exactly what we encountered previously. Note also the similarity to (2.10), which was to be expected due to the interpretation of the n-stage problem as a standard optimal control problem for a hybrid system.

Using these results, we have the following algorithm for computing the optimal trajectories:

1. Calculate $W_0, V_1, W_1, \ldots, V_{n-1}$ in that order by solving (5.11) and using the relation $V_{i+1} = W_i + \gamma_{i+1}$.

2. Find the optimal ending time and position from

$$(\tau_{n-1}, \boldsymbol{\xi}_{n-1}) \in \arg\min_{(\tau,\boldsymbol{\xi})} V_{n-1}(\tau,\boldsymbol{\xi})$$

3. For $i = n-2, \ldots, 0$, integrate the differential equation

$$\dot{\boldsymbol{x}}_i(\tau) = r\frac{\nabla_{\boldsymbol{\xi}} W_i(\tau,\boldsymbol{x}_i(\tau))}{\left|\nabla_{\boldsymbol{\xi}} W_i(\tau,\boldsymbol{x}_i(\tau))\right|} + \boldsymbol{v}(\tau,\boldsymbol{x}_i(\tau)) \tag{5.12}$$

   backwards in time with terminal condition $\boldsymbol{x}_i(\tau_{i+1}) = \boldsymbol{\xi}_{i+1}$ until $W_i(\tau,\boldsymbol{x}_i(\tau)) = V_i(\tau,\boldsymbol{x}_i(\tau))$, at which point set $\tau_i = \tau$, $\boldsymbol{\xi}_i = \boldsymbol{x}_i(\tau)$.

5.4 Selection of the numerical algorithm

As we typically can not hope to solve (5.5) analytically, both because of the complexity of the equation itself and the fact that $\boldsymbol{v}$ is given in practice by numerical data and not by an analytical expression, we must turn to the numerical methods described in Section 2.3.4.

Equation (5.5) is a static Hamilton-Jacobi equation, and its Hamiltonian is

$$H(\boldsymbol{z},\boldsymbol{p}) = r\sqrt{p_2^2 + p_3^2} - (p_1 + p_2 v_1(\boldsymbol{z}) + p_3 v_2(\boldsymbol{z}))$$

where $\boldsymbol{z} = (\tau, \boldsymbol{x})$ and $\boldsymbol{p} = (p_1, p_2, p_3) = (V_\tau, V_{\boldsymbol{x}})$. Given the type of equation, we are limited to the following options:

- Ordered upwind methods [54]
- Level set methods [41]

- Semi-Lagrangian methods [17]
- Fast sweeping methods [28]

Ordered upwind methods require that H be homogeneous in $\boldsymbol{p}$, so that there is a function F such that

$$H(\boldsymbol{z},\boldsymbol{p}) = |\boldsymbol{p}|F\left(\boldsymbol{z},\frac{\boldsymbol{p}}{|\boldsymbol{p}|}\right)$$

and this is satisfied in our case with $F = H$. However, there is an additional requirement on F, namely that that F be bounded below and above by positive numbers. In our case, if $\boldsymbol{p} = (1,0,0)$ then $F = -1$, so this is clearly not satisfied. This is related to the fact that, in terms of the optimal control problem associated to a given Hamilton-Jacobi equation, ordered upwind methods require local controllability in all directions [54], and we clearly don't have controllability in the time direction.

The use of level set methods is based on converting between (5.5) and the related equation

$$\phi_s + \frac{r|\phi_{\boldsymbol{x}}| - (\phi_\tau + \phi_{\boldsymbol{x}}\cdot\boldsymbol{v})}{g(\boldsymbol{x})} = 0$$

via a change of variables, where $\phi = \phi(s,\tau,\boldsymbol{x})$. Level set methods allow us to solve this equation for ϕ and then find V from ϕ. Thus we have to solve an equation on a space with an extra dimension. The extra dimension weighs considerably on both memory requirements and the computational cost, and we would like to use a more efficient and scalable method.

Semi-Lagrangian methods are also not the most efficient choice, since these methods explicitly find the optimal control at each point by solving the minimization in (2.14). Since we have an explicit form of the Hamilton-Jacobi-Bellman equation in this case (i.e., the minimization in (2.9) is explicitly solved), this minimization step is unnecessary, and we can solve the HJBE directly.

Hence, the logical choice seems to be the class of fast sweeping methods. Since we have an explicit form of the Hamiltonian, we can use the Lax-Friedrichs discretization of the Hamiltonian described in [28].

5.5 Implementation of a numerical solver based on the fast sweeping method

To the best of our knowledge, there is at the time of writing no publicly available implementation of fast sweeping methods for general Hamilton-Jacobi equations. Hence it was necessary to implement a fast sweeping method solver from scratch. Our goal is to have an implementation which is able to quickly solve standard-sized problems and which will scale to problems in more than two dimensions or large two-dimensional problems. Increasing numbers of desktop computers and laptops now ship with multi-core processors [68], and high-performance computing (HPC) platform nodes typically have at least 16 processing units. It makes sense to leverage this computing power to achieve the stated goal through a parallel implementation of the fast sweeping method.

5.5.1 Preliminary design decisions

The development environment and target platform is x86_64 Linux. Nonetheless, portability was kept in mind while making the design decisions that follow.

The logical choice for the programming language of the implementation was C++, due to a multitude of reasons [22, 61]:

- Zero-cost abstractions;
- C-like efficiency with a stronger type system which makes writing correct programs easier;
- Support for generic, object-oriented and concurrent programming paradigms;
- Interfaces for most or all of the widely used C and FORTRAN scientific computing libraries are available, so adding new functionalities or integrating new libraries will not force a change of programming language.

It is also important to carefully choose the version of the C++ standard to be used in development. On the one hand, a recent standard will provide more language and library features, but on the other it requires the availability of a recent compiler release. We want to be able to build and run the solver on FEUP's HPC platform Grid FEUP, which has the GNU Compiler Collection (GCC) version 7.3.1 installed, so we opted for the ISO C++ 2014 standard, which is fully supported by GCC since version 5 [1] .

Since we want to solve at least three-dimensional problems, data input/output via text files is not practical, and a proper file format for data storage must be used. We chose the Hierarchical Data Format version 5 (HDF5) file format for the following reasons:

- The HDF5 C library is either preinstalled or easily downloadable in most (if not all) Linux distributions;
- There are multiple C++ interfaces available;
- Most data processing software supports reading and writing HDF5 format files;
- Filesystem-like interface which allows storing multiple related datasets in a single binary file, with metadata stored alongside the data;
- Native support for multidimensional datasets.

For reading and writing HDF5 files in C++, the h5cpp library was chosen due to its simple modern C++ interface. Besides the HDF5 C library, the h5cpp library depends only on the Boost C++ libraries, which typically come preinstalled in any Linux distribution, so the integration cost is minimal.

The CMake build system generator is used to simplify the build process and promote portability.

The source code repository is managed using the git version control system and hosted on GitHub.

[1] See `https://gcc.gnu.org/projects/cxx-status.html`

5.5.2 Interface and usage

For the sake of flexibility and scalability, we opted for an implementation which is independent of the number of dimensions. The number of dimensions is, however, set at compile time via a `constexpr` variable. The `constexpr` specifier indicates that the value of the variable is available at compile time [61], and this enables compiler optimizations that would be impossible if the number of dimensions is specified at run time. For instance, in the following code snippet

```
constexpr int dim = 3;

// ...

for(int i = 0; i < dim; ++i)
    do_something(i);
```

the for loop can be unrolled since the compiler can see that i will take on the values $0,1,2$ successively. Such loops where the number of iterations is related to the number of dimensions are quite common in numerical code, so this will enable a significant amount of optimization. Of course, this would be achievable using a preprocessor macro, but `constexpr` variables have the added benefits of type safety and greater code clarity. In addition, the fact that the value is determined at compile time allows the compiler to perform computations involving the variable. In our case, we can use this to compute the number of sweeping directions at compile time:

```
constexpr int dim = 3;
constexpr int n_sweeps = 1 << dim;      // Equals 2^dim
```

This will enable the same optimizations on the number of sweeps. Another benefit is that we can use the number of dimensions as the size of a fixed-size array:

```
constexpr int dim = 3;
int point[dim];                         // This is allowed
```

Naturally, some of the most common objects in the code will be arrays with size equal to the number of dimensions. This allows us to avoid a significant number of heap allocations which would be necessary if the number of dimensions was determined at run time. We do not use ‘raw arrays’ directly but instead the `std::array` class template available in the C++ standard library, thin wrapper around a raw array providing type safety at little or no added cost [61].

For flexibility, we specify the precision of the floating point variables at compile time, via a type alias:

```
using scalar_t = double;
```

All floating point variables are subsequently declared with type `scalar_t`.

One could argue that the dimension and floating point precision should be template parameters of some function or class. However, the reason we don’t adopt that approach is that this would require recompiling all of the solver’s source code every time the user program is changed. This way, the solver only needs to be recompiled when the user changes the dimension or precision.

The solver was implemented in an application-independent fashion, so that it can be used to solve any Hamilton-Jacobi equation of the form (2.17), which includes the problem described above. The setup for a generic d-dimensional problem (i.e., $\boldsymbol{x} \in \mathbf{R}^d$), is similar to that described for our particular problem: we solve the problem on a grid with N_i points in the i-th dimension, $i = 1, \dots, d$, over the computational domain

$$D = \prod_{i=1}^{d} [x_i^{\min},\, x_i^{\max}].$$

A gridpoint $\boldsymbol{x}^{i_1,\dots,i_d} = (x_1^{i_1,\dots,i_d}, \dots, x_d^{i_1,\dots,i_d})$ is defined by its grid indices $(i_1, \dots, i_d)$, where $0 \le i_k < N_k$:

$$x_k^{i_1,\dots,i_d} = x_k^{\min} + i_k \frac{x_i^{\max} - x_i^{\min}}{N_k - 1}.$$

We also define

$$\begin{aligned} \vartheta^{i_1,\dots,i_d} &= \vartheta(\boldsymbol{x}^{i_1,\dots,i_d}) \\ g^{i_1,\dots,i_d} &= g(\boldsymbol{x}^{i_1,\dots,i_d}),\ \boldsymbol{x}^{i_1,\dots,i_d} \in \Gamma \\ q^{i_1,\dots,i_d} &= q(\boldsymbol{x}^{i_1,\dots,i_d}),\ \boldsymbol{x}^{i_1,\dots,i_d} \notin \Gamma \end{aligned}$$

The following user inputs are considered:

- A string specifying the path to the HDF5 file containing the problem data (Γ and the values of g and q at the gridpoints);
- A function which returns the value of the Hamiltonian $H(\boldsymbol{x}^{i_1,\dots,i_d}, \boldsymbol{p})$ given $(i_1, \dots, i_d)$ and $\boldsymbol{p} = (p_1, \dots, p_d)$;
- A function which returns the value of the artificial viscosity coefficients $\sigma_i(\boldsymbol{x}^{i_1,\dots,i_d})$ given $(i_1, \dots, i_d)$;
- The intervals which define the computational domain D;
- The tolerance ε;
- The initial value M.

We can store $g^{i_1,\dots,i_d}$, $q^{i_1,\dots,i_d}$ and Γ together in the same dataset $C^{i_1,\dots,i_d}$ as follows:

$$C^{i_1,\dots,i_d} = \begin{cases} g^{i_1,\dots,i_d}, & \boldsymbol{x}^{i_1,\dots,i_d} \in \Gamma \\ -1 - q^{i_1,\dots,i_d}, & \boldsymbol{x}^{i_1,\dots,i_d} \notin \Gamma \end{cases}$$

Recall that $g > 0$, $q \ge 0$, so we know that negative values of C indicate points not in Γ. The shift by -1 guarantees some degree of separation when the values of g are close to zero.

There are a few different options for passing the functions giving the Hamiltonian and the viscosity coefficients to the solver. Our approach is to use the `std::function` class template available in the C++ standard library, which can hold any object which is callable with a given signature, e.g. regular free functions, function objects or lambdas. A more traditional ‘C-like‘ approach would be to pass pointers to functions with given signatures. This is certainly the lowest overhead approach, but it doesn’t offer a lot of flexibility.

Although in this context there is not much use for traditional object-oriented concepts such as runtime polymorphism, C++ classes are still useful for grouping together related functions and data. The resulting code is clearer, and there is also a performance advantage due to the data locality. The solver uses three main class types:

`solver_t` is used to hold the problem data and define the user interface;

`grid_t` holds a description of the computational grid and implements the operations for accessing multidimensional gridded data described in Section 5.5.3;

`data_t` is used to store and access multidimensional gridded data such as the cost function and the value function.

Of these three, the user only has access to `solver_t`.

The following code listing exemplifies the solver interface usage for the case of an eikonal equation

$$\begin{aligned} |\nabla\vartheta(\boldsymbol{x})| &= g(\boldsymbol{x}),\ \boldsymbol{x}\in\Gamma \\ \vartheta(\boldsymbol{x}) &= q(\boldsymbol{x}),\ \boldsymbol{x}\notin\Gamma \end{aligned}$$

in two dimensions, over the square $[-1,1]\times[-1,1]$.

```
#include <array>
#include <numeric>

#include "fsm/solver.hpp"

using point_t = std::array<size_t, fsm::dim>;
using vector_t = std::array<fsm::scalar_t, fsm::dim>;

int main() {
    auto hamiltonian = [](point_t const& x, vector_t const& p) {
        // Norm of p
        return std::sqrt(
            std::inner_product(std::begin(p), std::end(p), std::begin(p), 0.0));
    };

    auto viscosity = [](point_t const& x) { return vector_t{ 1.0, 1.0 }; };

    constexpr std::array<std::pair<fsm::scalar_t, fsm::scalar_t>, fsm::dim>
        vertices = { { { -1.0, 1.0 }, { -1.0, 1.0 } } };

    fsm::solver::params_t params;
    params.tolerance = 1.0e-4;
    params.maxval = 2.0;

    fsm::solver::solver_t s("data.h5",
                            hamiltonian,
                            vertices,
                            viscosity,
                            params);

    s.solve();
}
```

5.5.3 Loading and accessing gridded data

C++ has no native data structure for holding d-dimensional data [2] . One could rely on a third party library for this capability, but it is simpler to store the data in a one-dimensional array in some predefined order. Thus, we opted to store the data in row-major order, since it is the order used in the HDF5 format for storing multidimensional data. A gridpoint $\boldsymbol{x}^{i_1,\dots,i_d}$ is mapped to the index

$$n^{i_1,\dots,i_d} = i_d + N_d(i_{d-1} + N_{d-1}(i_{d-2} + \dots (i_2 + N_2 i_1) \dots))$$

The inverse mapping, from the index to the indices $(i_1,\dots,i_d)$ of the corresponding gridpoint $\boldsymbol{x}^{i_1,\dots,i_d}$, is calculated using the following algorithm:

```
1. $n \leftarrow n^{i_1,\dots,i_d}$
2. For $k = d, d-1, \dots, 1$:
    (a) $i_k := n \bmod N_k$
    (b) $n \leftarrow \frac{n - i_k}{N_k}$
```

When the data containing the values of the components of the cost is loaded, we can infer from it the dimensions of the grid (i.e. the number of points in each dimension), and check that the number of dimensions coincides with the number of dimensions the solver was compiled for.

As the size of the loaded data can be large if the problem is large or high-dimensional, we ensured that only a single copy of the data is loaded into memory, i.e., no extra copies are made when passing the data between two objects or returning it from a function. This is done simply by passing a pointer to the data instead of the data itself. The `std::unique_ptr` class template available in the C++ standard library makes it easy to do this safely, avoiding memory leaks [61].

[2] Where d is defined by a variable. Naturally, if the number of dimensions was fixed, we could simply use multidimensional C arrays.

5.5.4 Single-threaded implementation

5.5.4.1 The main loop

In a single-threaded (sequential) implementation, the main loop is simply a direct implementation of the second step of the algorithm described in Section 2.3.4.1, so we focus on the most relevant implementation details.

For the sweeping part, we encode each sweep directions as an integer s such that $0 \leq s < 2^d$. This is a d-bit number, and we can map it to a sweeping direction as follows: if the k-th bit is set (equal to one), we sweep from high to low in the k-th dimension, otherwise we sweep from low to high in that dimension. The sweep in the direction given by s starts at the gridpoint $\boldsymbol{x}^{i_1,\ldots,i_d}$ such that $i_k = 1$ if $s \,\&\, 2^{k-1} \neq 0$ and $i_k = N_k - 2$ otherwise, where & denotes a bitwise AND operation. During the sweep, if we have updated $\boldsymbol{x}^{i_1,\ldots,i_k}$, the indices of the next gridpoint can be calculated as follows:

1. If $s \,\&\, 1 \neq 0$ then $i_1 \leftarrow i_1 - 1$, else $i_1 \leftarrow i_1 + 1$
2. For $k = 1, \ldots, d-1$:
    (a) If $i_k = 0$ or $i_k = N_k - 1$:
        i. If $s \,\&\, 2^{k-1} \neq 0$ then $i_k \leftarrow N_k - 2$, else $i_k \leftarrow 1$
        ii. If $s \,\&\, 2^k \neq 0$ then $i_{k+1} \leftarrow i_{k+1} - 1$, else $i_{k+1} \leftarrow i_{k+1} + 1$

The basic idea is to increment the first dimension, and if we are at the computational boundary then we 'reset' the first dimension and increment the second, if we are at the boundary of the second dimension we 'reset' the second and increment the third, and so on.

After each sweep we must update the points in the computational boundary. The computational boundary is the union of $2d$ segments of hyperplanes:

$$\bigcup_{k=1}^{d} \left\{\boldsymbol{x} \in D \;\middle|\; x_k = x_k^{\min}\right\} \cup \{\boldsymbol{x} \in D \mid x_k = x_k^{\max}\}.$$

Denoting these hyperplane segments as

$$S_k = \left\{\boldsymbol{x} \in D \;\middle|\; x_k = x_k^{\min}\right\},\ S_{d+k} = \{\boldsymbol{x} \in D \mid x_k = x_k^{\max}\},$$

To iterate along the points in S_j, we start at the gridpoint $\boldsymbol{x}^{i_1,\ldots,i_d}$, where $i_k = 0$ if $d + k \neq j$ and $i_k = N_k - 1$ otherwise. If $\boldsymbol{x}^{i_1,\ldots,i_d}$ has been updated, the next gridpoint can be calculated using the following algorithm:

1. If $j \leq d$ then $k_b := j$, else $k_b := j - d$
2. $i_{r(k_b,1)} \leftarrow i_{r(k_b,1)} + 1$
3. For $k = 1, \ldots, d-2$:

(a) If $i_k = N_k - 1$:

i. $i_{r(k_b,k)} \leftarrow 0$

ii. $i_{r(k_b,k+1)} \leftarrow i_{r(k_b,k+1)} + 1$

where

$$r(k_b,k) = \begin{cases} k_b + k, & k_b + k \leq d \\ k_b + k - d & k_b + k > d \end{cases}$$

The idea is to 'rotate' the indices so that k_b (the 'boundary' index) is the first dimension and we increment all other dimensions sequentially. In other words, we map to a system of coordinates on the $d-1$ dimensional manifold S_k and iterate over the points using that system of coordinates.

After each sweep and boundary update we must check if the ℓ_∞ norm of the difference between the value function before the sweep and the updated value function is less than the tolerance ε. The naive way of doing this would be to keep two copies of the value function and compare them after each sweep and boundary update, but we can save memory and time by calculating the ℓ_∞ norm as we update the points. We first set $\delta = 0$, and when updating gridpoint $\boldsymbol{x}^{i_1,\dots,i_d}$, if the new value is ϑ' we simply set

$$\delta = \max\left\{\delta, \vartheta' - \vartheta^{i_1,\dots,i_d}\right\}$$

Since each gridpoint is updated once per sweep and boundary update, in the end δ will be equal to the maximum difference between the old and new values, i.e. the ℓ_∞ norm of this difference.

5.5.4.2 Validation and benchmarks

Gridpoints per dimension	Wall clock time (s)	ℓ_∞ Absolute error
51	6.41	0.28797977078281200
101	64.58	0.15862420696771995
121	121.18	0.13533268851117297
151	254.89	0.11132716009398536
171	612.67	0.09973797954314878
201	1405.85	0.08643654921008004
221	1883.34	0.07946110092067471
251	2924.09	0.07096529729804590

Table 5.1: Computation time and absolute error for various grid sizes – single-threaded implementation

The implementation is validated using the three-dimensional eikonal equation

$$|\nabla\vartheta(\boldsymbol{x})| = 1$$
$$\vartheta(\boldsymbol{0}) = 0.$$

this equation has an analytical solution $\vartheta(\boldsymbol{x}) = |\boldsymbol{x}|$, so we can measure the error in the numerical solution.

The solution is computed over the square $[-1,1]^3$. The parameters for the fast sweeping solver are $M = 2.0$, $\varepsilon = 1 \times 10^{-16}$, $\sigma_i(\boldsymbol{x}) = 1$.

The results for various grid sizes are summarized in Table 5.1. The computations were performed on a laptop running Linux with an Intel$^{(R)}$ Core^{TM} i7-8550U CPU @ 1.80GHz.

The absolute error decreases only slightly worse than linearly as the resolution $h = \frac{1}{N-1}$ decreases, where N is the number of gridpoints per dimension, consistent with the results reported in Kao et al. [28].

5.5.5 First parallel implementation

The first attempt at a parallel implementation is based on the first method described in [73]. As described in Section 2.3.4.1, the method consists simply of performing the different sweeps simultaneously. Thus, we have 2^d threads of execution, and must keep $2^d + 1$ different copies of the solution in memory: 2^d copies to be able to perform each sweep in different data, and an extra copy to store the old value of the solution, so that we can keep track of the convergence progress.

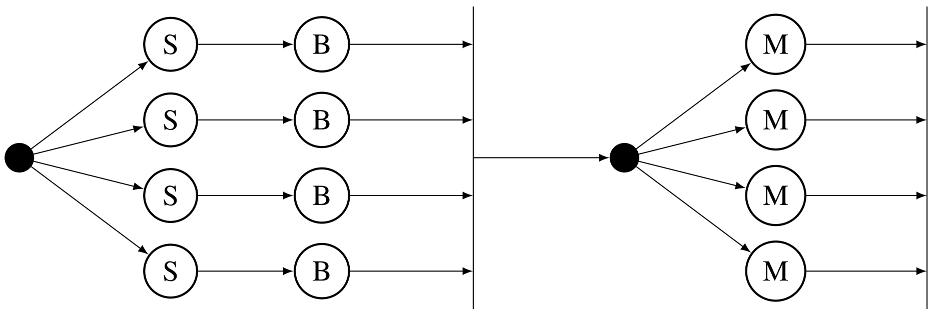


Figure 5.2: Task structure in the main loop for $d = 2$

The task structure is represented in Figure 5.2 for $d = 2$ (For $d \neq 2$ the only difference is the number of tasks in each step). The black nodes represent steps where tasks are spawned, white nodes represent tasks, arrows represent the task flow and the 'walls' represent steps where all preceding tasks must finish before proceeding. The nodes labelled 'S' denote a sweep, those labelled 'B' denote the boundary update and those labelled 'M' denote the 'merge' or synchronization step where the different solutions are combined and the ℓ_∞ norm of the difference between the values before and after the iteration is calculated.

In order to parallelize the merge step, we divide the gridpoints among the worker threads and each of them calculates the solution value on its part of the grid as well as the ℓ_∞ difference between iterations on that part of the grid. After all threads have finished the merge step, the maximum difference among all parts of the grid is found, which is the ℓ_∞ difference between iterations on the whole grid.

During the sweep and boundary updates, each thread is working only on its own array of solution values, so no locking or synchronization is necessary. In the merge step, all threads have access to all the solution arrays, but each thread only reads from/writes to a segment of the array which no other thread is accessing, so there is no risk of data races here either.

Hence, the only possible parallel overhead is due to thread creation. In order to avoid creating a thread each time a task is spawned, we use a thread pool. This is a data structure which starts a given number of threads at program startup. Tasks can then be pushed to a task queue and when a thread is idle it will pop a task from the queue and execute it [68]. We use an implementation based on the open source thread pool library available at `https://github.com/progschj/ThreadPool`. This implementation uses the concurrency facilities available in the C++ standard library, so that it is platform-independent and introduces no additional dependencies.

Since at program startup we already have one thread of execution (the main thread), we only need to start $2^d - 1$ additional threads for the thread pool. In the main loop, the main thread enqueues the first $2^d - 1$ sweep tasks in the thread pool queue and executes the last sweeping direction itself. After it is done executing its sweep, the main thread waits for all other threads to finish the sweep and boundary update. The same method is used for the merge step, where the main thread splits the grid into 2^d sections, queues the first $2^d - 1$ sections on the thread pool and executes the merge task on the last section before waiting on the remaining threads.

5.5.5.1 Validation and benchmarks

The implementation is validated using the same example as before (three-dimensional eikonal equation), using the same platform and settings. The laptop used has four cores with 2 processing units each, so that 8 threads threads can be executed in parallel, which is just what is needed for a three-dimensional problem.

The results are shown in Table 5.2. The speedup is calculated as the wall clock time for the single-threaded implementation divided by the wall clock time for the parallel implementation. The parallelization does not seem to bring any advantage. This is because a larger number of iterations are necessary for convergence when the sweeps are done in parallel, which cancels out the reduction in the time taken to compute a single iteration.

Gridpoints per dimension	Wall clock time (s)	Speedup	ℓ_∞ Absolute error
51	10.07	0.64	0.28797977078281090
101	105.85	0.61	0.15862420696771684
121	197.90	0.61	0.13533268851117053
151	452.57	0.56	0.11132716009398247
171	748.37	0.82	0.09973797954314612
201	1313.37	1.07	0.08643654921007737
221	1860.80	1.01	0.07946110092067005
251	2762.36	1.06	0.07096529729804213

Table 5.2: Computation time and absolute error for various grid sizes – first parallel implementation

5.5.6 Second parallel implementation

Due to the shortcomings of the method used for the first parallel implementation, which were confirmed in the benchmarks, we opted to implement the hyperplane stepping method [13], which promises near-optimal parallel speedup.

Recall that the gist of the method is to partition the grid into sets of points ('levels') L_m such that the points in L_m can be updated simultaneously. As we are targeting a multi-CPU architecture where the number of threads will be small compared to the number of points in L_m, we split the points in L_m into as many groups as there are threads, and each thread updates its group of points concurrently.

The number of threads is not determined by any of the problem parameters and can be set freely. In our implementation, it can be set at compile time by the user via a `constexpr` variable. Alternatively, if the user does not provide the desired number of threads, we use the `std::thread::hardware_concurrency` function from the C++ standard library which lets us determine the number of parallel threads supported by the hardware in a platform independent way. If `std::thread::hardware_concurrency` is is unable to determine the number of parallel threads it will return zero, in which case we use two threads.

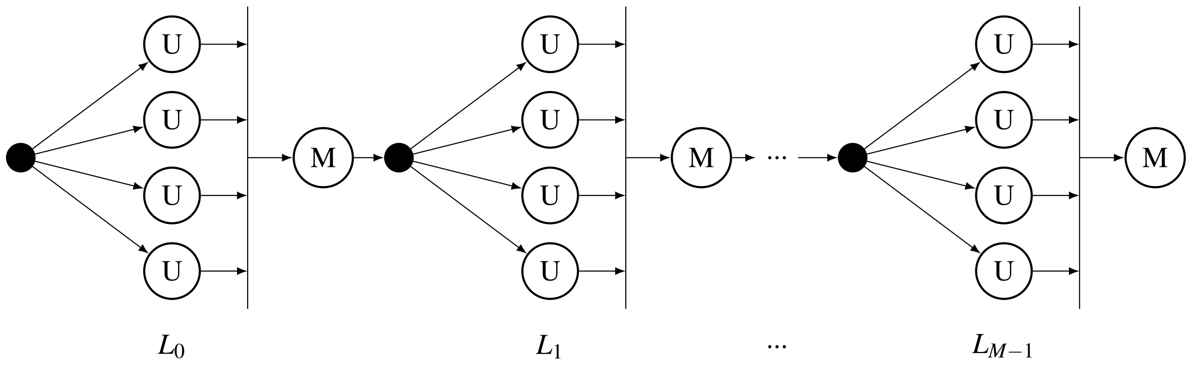


Figure 5.3: Task structure during sweeping with four threads

Figure 5.3 represents the task structure during sweeping with four threads. The 'U' tasks consist of updating a group of points in a level. As in the single-threaded implementation, the ℓ_∞ norm difference of the difference before and after the sweep is calculated as the points are being updated. After a 'U' task has finished, we have the norm of the difference for the corresponding group of points. The purpose of the 'M' tasks is to keep track of the maximum norm of the difference among the groups and the previous levels that have been calculated, so that after the last 'M' task has finished we have the norm of the difference on the whole grid. The total number of levels is $M = 1+\sum_{k=1}^{d}(N_k-1)$.

Similarly to the previous implementation, the 'U' tasks are assigned to threads using a thread pool, and the main thread performs a 'U' task itself after it has scheduled all the other tasks.

If $d = 2$, generating the sets L_m is simple. Starting from the point $\boldsymbol{x}^{i,j}$ with

$$i = \min\{m, N_1 - 1\}, \; j = m - i$$

we generate the next point in L_m by moving along the diagonal:

$$i \leftarrow i-1, \; j \leftarrow j+1$$

until either $i = 0$ or $j = N_2 - 1$. For $d > 2$ we can do it recursively on the number of dimensions. Let L'_m be the levels for the grid with dimensions $(N_2, \dots, N_d)$. Then we can iterate over the points $\boldsymbol{x}^{i_1,\dots,i_d} \in L_m$ using the following algorithm:

```
1. For $m' = m, m-1, \dots, 0$:
    (a) $i_1 \leftarrow \min\{m', N_1 - 1\}$, $n \leftarrow m' - i_1$, $S \leftarrow L'_n$
    (b) While $S \neq \emptyset$:
        i. Remove a point $\boldsymbol{x}^{j_2,\dots,j_d}$ from $S$
        ii. For $k = 2, \dots, d$, $i_k \leftarrow j_k$
```

As in [13], instead of generating the levels in different orders for the different sweeping directions, we can generate them for a single order and rotate the axes for the different sweeping orders, i.e., if we are sweeping in the order given by s, we apply the following transformation to each point in L_m:

```
1. For $k = 1, \dots, d$:
    (a) If $s \,\&\, 2^{k-1} \neq 0$, $i_k \leftarrow N_k - 1 - i_k$
```

In addition, instead of generating the sets L_m in every iteration, we can generate and store them after the solver has been set up. This implies storing $\prod_{k=1}^{d} N_k$ points, so it might not be an option for large or high-dimensional problems, depending on the available memory.

5.5.6.1 Validation and benchmarks

We validate the implementation using the same example, settings and platform as in the previous two implementations. Eight parallel threads were used in the computations.

The results are shown in Table 5.3. The implementation is clearly more efficient and near-optimal speedup is achieved for the larger problems. Note that the absolute errors are exactly the same as in the single-threaded implementation tests, which is evidence that the points are updated in the same sequence (so that the number of iterations is the same).

Gridpoints per dimension	Wall clock time (s)	Speedup	ℓ_∞ Absolute error
51	3.19	2.01	0.28797977078281200
101	19.02	3.40	0.15862420696771995
121	32.84	3.69	0.13533268851117297
151	65.25	3.91	0.11132716009398536
171	101.90	6.01	0.09973797954314878
201	182.85	7.69	0.08643654921008004
221	245.44	7.67	0.07946110092067471
251	386.06	7.57	0.07096529729804590

Table 5.3: Computation time and absolute error for various grid sizes – second parallel implementation

5.6 Configuration of the solver for optimal planning problems

5.6.1 Base problem

In order to solve the equation numerically, the first step is to choose an hyperrectangle on which the solution is calculated, i.e., a product of intervals

$$D = [\tau^{\min}, \tau^{\max}] \times [x_1^{\min}, x_1^{\max}] \times [x_2^{\min}, x_2^{\max}].$$

This computational domain D should contain the three-dimensional target set Ω'.

Note that we will only be able to compute the value of $V(\tau,\boldsymbol{x})$ for $(\tau,\boldsymbol{x})$ such that $\tau + T(\boldsymbol{x},\tau,\boldsymbol{u}^\star) < \tau^{\max}$, where $\boldsymbol{u}^\star$ is the optimal control, since if this inequality does not hold at a gridpoint, the optimal trajectory from that point is not contained in the computational boundary.

We then discretize this domain with a given resolution in each dimension. It makes sense to discretize x_1 and x_2 with the same resolution, because there is no preferred direction of motion. Hence we choose resolutions δ_τ and $\delta_{\boldsymbol{x}}$ for time and position, respectively. The number of gridpoints in each dimension is chosen so that the resolution is equal to or finer than the specified resolutions:

$$\begin{aligned} N_\tau &= 1 + \operatorname{ceil}\left(\frac{\tau^{\max} - \tau^{\min}}{\delta_\tau}\right) \\ N_{x_1} &= 1 + \operatorname{ceil}\left(\frac{x_1^{\max} - x_1^{\min}}{\delta_{\boldsymbol{x}}}\right) \\ N_{x_2} &= 1 + \operatorname{ceil}\left(\frac{x_2^{\max} - x_2^{\min}}{\delta_{\boldsymbol{x}}}\right) \end{aligned}$$

($\operatorname{ceil}(x)$ denotes the ceiling of x, i.e., the smallest integer larger than x). The effective resolution in each dimension will then be

$$\begin{aligned} h_\tau &= \frac{\tau^{\max} - \tau^{\min}}{N_\tau - 1} \leq \delta_\tau \\ h_{x_i} &= \frac{x_i^{\max} - x_i^{\min}}{N_{x_i} - 1} \leq \delta_{\boldsymbol{x}},\ i = 1,2. \end{aligned}$$

and the gridpoints are defined as

$$\begin{aligned} \tau^i &= \tau^{\min} + i \cdot h_\tau,\ i = 0,\ldots,N_\tau - 1 \\ x_1^j &= x_1^{\min} + j \cdot h_{x_1},\ j = 0,\ldots,N_{x_1} - 1 \\ x_2^k &= x_2^{\min} + k \cdot h_{x_2},\ k = 0,\ldots,N_{x_2} - 1. \end{aligned}$$

The output of a numerical solver will be the approximate values of the solution at the gridpoints:

$$V^{i,j,k} = V(\tau^i, x_1^j, x_2^k).$$

To minimize roundoff errors and ensure that all operations are done with as much precision as possible, it is wise to normalize the problem data so that D is as close to a unit cube as possible and the grid cell widths h_τ, h_{x_1}, h_{x_2} are within an order of magnitude of each other.

The target set is specified simply as a binary mask on the gridpoints, which determines which gridpoints are considered to belong to Ω'. The terminal cost q specifies the values of the solution on those gridpoints:

$$V^{i,j,k} = q(x_1^j, x_2^k)$$

where x_1^j, x_2^k are such that the gridpoint (τ^i, x_1^j, x_2^k) is in the discretized target set. The running cost function g and the values of the ocean flow velocity $\boldsymbol{v} = (v_1, v_2)$ also only need to be specified at the gridpoints.

The fast sweeping method needs two additional user inputs, namely the initial value M of the numerical solution $V^{i,j,k}$ at points not in the target set and the tolerance ε. The initial value M should be an upper bound of the value function over the computational region. For our problem, this is easy to estimate:

$$V^{i,j,k} \leq (\tau^{\max} - \tau^{\min}) g_{\max} + q_{\max}$$

where $g_{\max}$ is an upper bound on g over the computational region, and $q_{\max}$ is an upper bound on q on the target set. As for the tolerance ε, we could simply take it to be equal to the machine epsilon (approximately 2.22×10^{-16} for 64-bit double precision). However, the error due to the discretization will almost always be orders of magnitude larger than this value, as shown in the solver benchmarks. Hence it is probably wiser to set ε according to the problem data, that is, choosing it such that a difference of ε in the value of the value function at a gridpoint does not change the meaning of that value. For instance, if we are solving a minimum time problem, then a difference of one second in the optimal cost from some point is immaterial if the optimal cost from that point is in the order of several hours.

The values of the solution along the time axis follow a causality property. Namely, the value of the solution at earlier times depends on the value of the solution at later times:

$$V(\tau_0, \boldsymbol{x}_0) = \int_{\tau_0}^{\tau} g(\boldsymbol{x}(t))\,\mathrm{d}t + V(\tau, \boldsymbol{x}(\tau))$$

where $\boldsymbol{x}(t)$ is an optimal trajectory, and the values of the solution at the terminal times are determined by q. This means that the values at the computational boundary $\tau = \tau^{\max}$ should not be updated using formula (2.20), since it uses the values at points with $\tau < \tau^{\max}$ to determine the value at the points with $\tau = \tau^{\max}$.

In fact, the correct approach is to not updated the points with $\tau = \tau^{\max}$ at all, since one of two things will happen:

1. The point is in the target set, and then its correct value is already determined by q;

2. The point is not in the target set, in which case the optimal trajectory starting at the point would necessarily exit the computational region, so the optimal cost from that point can not be determined. In this case, the target should be considered unreachable from that point, and

the value of the value function at that point is equal to $+\infty$. For our purposes, the upper bound M is the same as infinity, and that is the initial value of the value function at the gridpoints not in the target set; therefore the correct approach is to not update these points.

5.6.2 Planning with logic-based constraints

In terms of the selection of the computational domain and its discretization, as well as the selection of the tolerance parameter ε, the procedure is exactly the same as in the base problem. The main difference lies in the boundary conditions. The variational equation (5.11) does not specify any boundary conditions, but since we have that $W_i \leq V_i$, the numerical approximation of W_i should be initialized with the values of V_i in every gridpoint. We can discretize (5.11) as

$$0 = \max\left\{W_i^{i_1,i_2,i_3} - V_i^{i_1,i_2,i_3}, W_i^{i_1,i_2,i_3} - \texttt{update}(W_i)\right\}$$

where $\texttt{update}(W_i)$ is the Lax-Friedrichs update formula (2.19) adapted to the PDE which appears in the second argument of the max in (5.11). This is the same as

$$W_i^{i_1,i_2,i_3} = \min\left\{V_i^{i_1,i_2,i_3}, \texttt{update}(W_i)\right\}.$$

Since W_i is initialized as V_i and the fast sweeping method never allows the solution to become larger than the initial value, the first part of the min is unnecessary. Therefore, the update function for this variational inequality is exactly the same as the one for Hamilton-Jacobi PDEs, and the solver does not need to be adapted in any way to solve this equation.

There is again a causality property which should be observed, however, in this case values of the value function at later times depend on the values at earlier times, i.e., the direction of the causality in this case is forward in time. This is because there is an initial cost given by V_i, instead of a terminal cost as in the base problem. Hence, the $\tau = \tau_{\min}$ boundary should not be updated in this case.

5.7 Calculating optimal trajectories from the value function

After an approximation of the value function has been computed by numerically solving the HJBE, we can calculate an optimal trajectory with initial condition $\boldsymbol{x}(\tau_0) = \boldsymbol{x}_0$ for any $(\tau_0, \boldsymbol{x}_0) \in D$ by integrating the ordinary differential equation (5.7) with this initial condition. This can be done using any standard ODE integration method such as Euler's method or fourth order Runge-Kutta. The spatial gradient $V_{\boldsymbol{x}}$ can be calculated in each gridpoint using a finite-difference formula, but we must be able to calculate the right-hand side of the equation at any point of D in order to use such methods. This is done by interpolating the gradient $V_{\boldsymbol{x}}$ to the point of interest. If $\boldsymbol{v}$ is also known only at the gridpoints then it must also be interpolated when solving (5.7). The same method is used for numerically integrating (5.12).

It is important to note that despite the fact that we only have knowledge of the value function at a discrete set of points, we can generate trajectories from any deployment position and time inside the computational region. Additionally, the trajectories are not a discrete sequence of gridpoints but continuous curves which can be sampled with arbitrarily small timesteps (within the limits of floating point computations, obviously).

Chapter 6

Numerical examples

6.1 Test cases

The method will be validated using three test cases which will illustrate its effectiveness and usability in real-life operations scenarios. These will be performed using two datasets from high resolution ocean models of the Sado and Tejo estuaries in Portugal. The ocean current data is courtesy of Américo S. Ribeiro, João Miguel Dias and Renato Mendes (NMEC-CESAM, Physics Department, University of Aveiro, Portugal).

Following is a description of each of the tests.

Entering and exiting the Sado estuary – We consider two mission scenarios in the Sado estuary:

1. The vehicle is deployed outside the estuary; the objective is to enter the estuary and reach a given location in minimum time;
2. The vehicle starts from a point located inside the estuary and must reach a target area outside the estuary in minimum time.

Each of the two scenarios will be tested using 24 different tidal periods extracted from the data. This allows us to get an estimate of the computational time of the method for a real life problem, as well as to see how different patterns in the ocean current affect the solution.

The Sado estuary is an area with tidal-driven currents with large velocities, reaching and sometimes exceeding 2 meters per second. The geography of the region also makes it an interesting test site, as the vehicle has to go through a narrow channel to enter or exit the inner part of the estuary, and there are a lot of low-depth 'islands' which the vehicle must avoid.

Currents with high spatial variability – The second test will involve an ocean flow in the Sado and Tejo estuaries which displays rapid variations in its direction with respect to the spatial variable.

This will allow us to test the method in extreme conditions in order to determine whether the generated trajectories violate any kinematic constraints not taken into account in the motion model.

Software-in-the-loop simulations – Using the simulation capabilities of the LSTS toolchain, three missions will be simulated, where in each mission the vehicle tracks a trajectory generated using the results of the first test case. A simple sampling strategy is used to translate the trajectories into mission plans.

The test will allow us to check whether a real AUV is able to track the trajectories generated by the method, as well as illustrate the integration of the method in existing software frameworks for operations with autonomous vehicles.

This is summarized in Table 6.1.

Test case		Number of tests	Dataset
Entering and exiting the Sado estuary	Mission 1	24	Sado
	Mission 2	24	Sado
Currents with high spatial variability		1	Sado and Tejo
Software-in-the-loop simulations		3	Sado

Table 6.1: Summary of the three test cases

6.2 Mission setup and data preprocessing

Section 5.4 describes how we define the parameters for the numerical solver. The computational domain is

$$D = \underbrace{[\tau^{\min}, \tau^{\max}]}_{D_\tau} \times \underbrace{[x_1^{\min}, x_1^{\max}] \times [x_2^{\min}, x_2^{\max}]}_{D_{\boldsymbol{x}}}.$$

The domains $D_{\boldsymbol{x}}$ and $D_{\boldsymbol{\tau}}$ are the operational area and the mission time frame, respectively. The spatial resolution $\delta_{\boldsymbol{x}}$ and the temporal resolution δ_τ will naturally be chosen according to the spatial and temporal resolutions of the data, respectively.

The spatial component of the computational domain, $D_{\boldsymbol{x}}$, describes a part of the Earth's surface, and the meaning of 'regular grid' changes depending on which coordinate system is used to identify points of the Earth's surface. In the motion model (5.1), the components $x_{1,2}$ of the position are in linear units, e.g., they are meters if the flow velocity $\boldsymbol{v}$ is in meters per second and the time t is in seconds. Accordingly, we must use a system of coordinates such that sampling uniformly along each coordinate will result in a uniform distance between the gridpoints. The Universal Transverse Mercator (UTM) coordinate system satisfies this requirement. This coordinate system parameterizes a region of the Earth's surface by mapping each point to its easting and northing, which are the distances to an origin point in meters along the North and East directions, respectively. As such a mapping can only be locally accurate, the UTM system divides the Earth into 60 *UTM zones*. In our case the zone of interest is Zone 29, which covers the whole Portuguese coast.

The data consists of values of the ocean current in the East, North and Down directions at points of a spatial mesh and sampled at uniformly spaced instants of time. Although the models output three-dimensional velocity values on multiple depth layers, we only use the horizontal (East and North) velocity values of the layer which is closest to the surface, and consider that the vehicle will track the trajectories at low and constant depth (0 to 2 meters).

Typically the grid on which the ocean current data is defined does not coincide with the grid on which we are solving the HJBE. This can be because the model data is defined on a curvillinear grid while the HJBE is solved on a regular grid, and/or because the data is defined on a superset of D. Thus it is necessary to first compute the values of the ocean flow velocity on the gridpoints.

Assume that the ocean flow data is defined on a mesh S in space and sampled at regular time intervals. Among the time samples of the data, select those sampling instants t^i, $i = 0,\dots,N-1$ which satisfy

$$\tau^{\min} - \Delta \leq t^i \leq \tau^{\max} + \Delta.$$

where Δ is some adjustable amount of time which guarantees that the whole interval $[\tau^{\min}, \tau^{\max}]$ is covered. Then we have datapoints

$$\boldsymbol{v}(t^i, x_1, x_2),\ (x_1, x_2) \in S,\ i = 0,\dots,N-1.$$

For each i, we interpolate these datapoints to find

$$\boldsymbol{v}(t^i, x_1^j, x_2^k),\ i = 0,\dots,N-1,\ j = 0,\dots,N_{x_1}-1,\ k = 0,\dots,N_{x_2}-1.$$

This is a two-dimensional interpolation of unstructured data. We opt for linear interpolation because, as we are interpolating to a fixed discrete set of points, smoothness of the interpolating function is not critical. We use the SciPy interpolation package written in Python to perform the interpolation.

The data is then interpolated in time to the regular grid, obtaining

$$\boldsymbol{v}(\tau^i, x_1^j, x_2^k),\ i = 0,\dots,N_\tau-1,\ j = 0,\dots,N_{x_1}-1,\ k = 0,\dots,N_{x_2}-1.$$

This is a one-dimensional interpolation for each (j,k). Linear interpolation would use only the two closest time slices to interpolate the data to some instant of time. However, the data varies on a large time scale compared to the sampling interval, so it makes sense to use a higher-order method in this case to capture large time scale trends. For this reason we opted for cubic spline interpolation, also performed using the SciPy interpolation package.

If the operational area includes land regions which the vehicle will have to avoid, to include these in the formulation as described in Section 5.2.1, we must obtain a binary water/land mask on the grid. We do this using the Generic Mapping Tools [66], which uses the Global Self-Consistent Hierarchical High Resolution Geography Database (GSHHG) shorelines dataset [67] to create such a mask.

In addition, areas of the operational area where the seafloor depth is less than the operating depth of the vehicle (plus a safety margin) must also be avoided in the generated trajectories.

This can be done using a nautical chart of the operational area. We approximate the low-depth areas using polygons, check which gridpoints are inside the polygons, and add those points to the water/land mask as land points.

6.3 Entering and exiting the Sado estuary

The output of the model are the values of the ocean current on a curvilinear irregular mesh with a mean resolution of 100 meters. Figure 6.1 shows the geographical region included in the model, which is roughly 32.25 km by 48.06 km in size. The values are given over three-day intervals with a temporal resolution of 10 minutes, over the period from September 10th to September 22nd 2018. Detailed descriptions of the model and its validation and calibration can be found in [2, 50].

6.3.1 Mission setup

We now illustrate the procedure described in Section 6.2 for the two missions in the Sado estuary; the setup is the same for both.

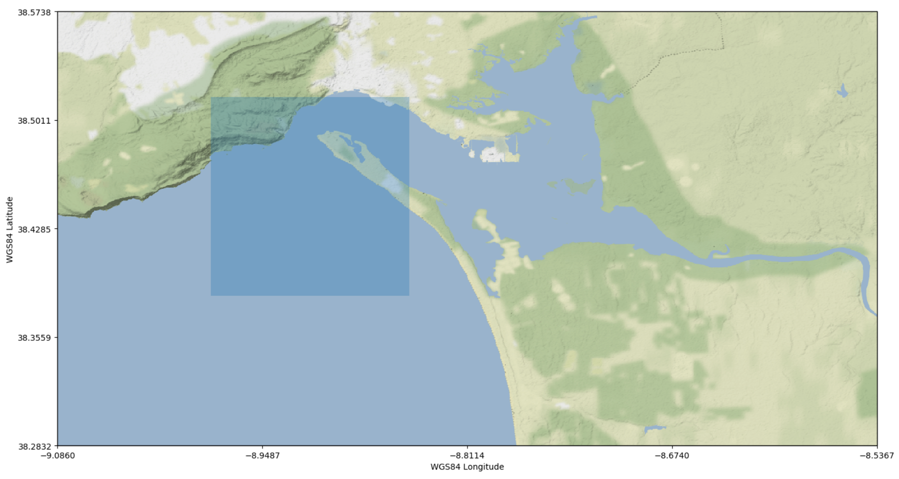
As the data spans quite a large region, we first choose a subset of the region for the operational area. The selected area is indicated by the blue rectangle in Figure 6.1, and this selection was motivated by the mission objectives. The WGS84 coordinates of the lower left and upper right corners of the rectangle are (38^o N 22’ 0.00”, 8^o W 59’ 0.00”) and (38^o N 31’ 0.00”, 8^o W 51’ 0.00”), respectively. The next step is to select the time interval on which the value function will be calculated. Figure 6.2 shows a few snapshots of the data. The currents are clearly tidal-driven, as expected for an estuarine region. For this reason, 12-hour intervals should be selected so as to capture a full tide period.

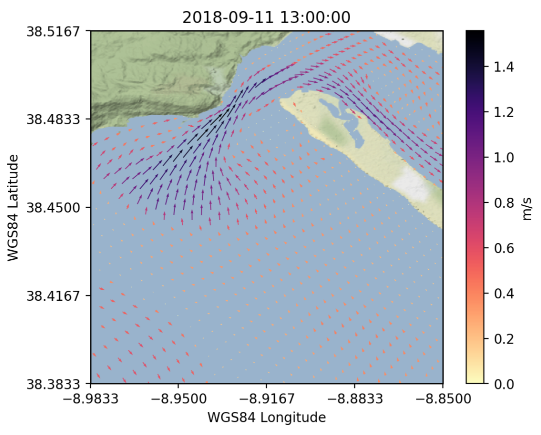
The current data has a mean spatial resolution of 100 meters. To avoid losing details in areas where the data has a spatial resolution finer than 100 meters, the value function will be calculated on a grid with a spatial resolution of 50 meters. The temporal resolution is chosen to be the same as that of the data, i.e. 10 minutes.

Figure 6.3 shows side-by-side comparisons of the actual data with the result of the interpolation algorithm for two different instants of time. One can see that the gridded vector field preserves the magnitude of the flow and the qualitative behavior of different parts of the flow observable in the data.

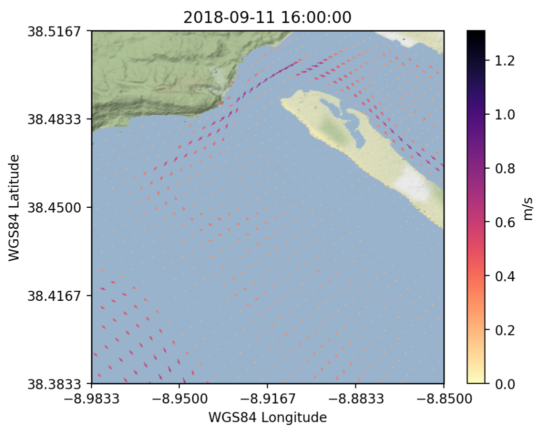
The land/water mask for this region (including low-depth regions) is shown in Figure 6.4.

Figure 6.1: Operational area for the mission.

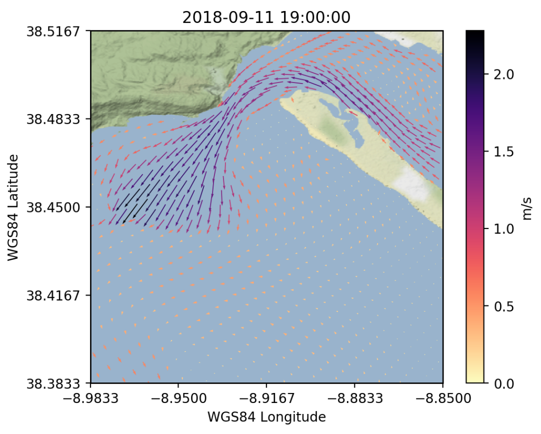




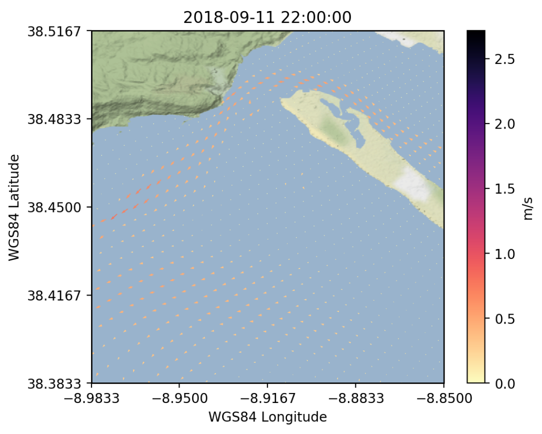
(a) Flood tide



(b) High tide

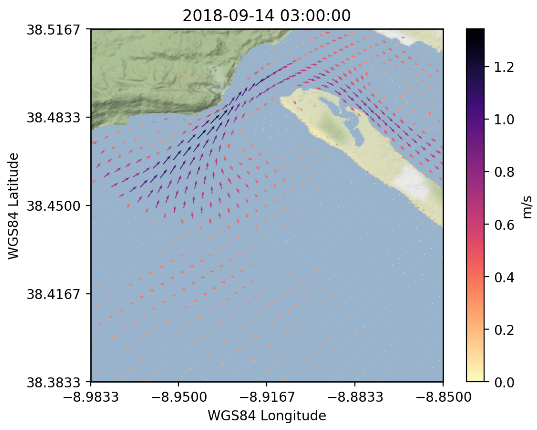


(c) Ebb tide

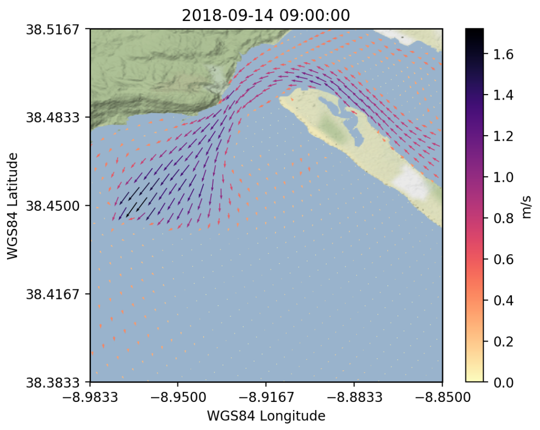


(d) Low tide

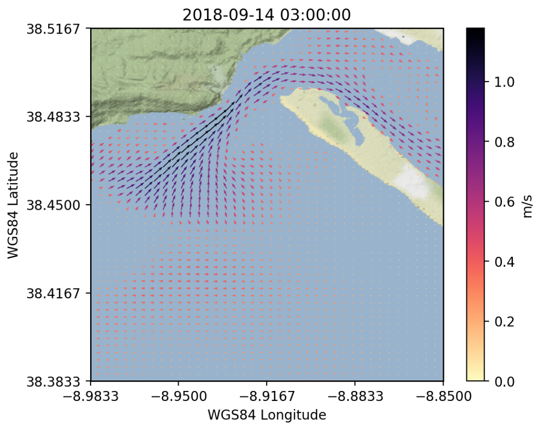
Figure 6.2: Snapshots of the model data over the operational area. For visualization purposes the data was downsampled by a factor of 3.



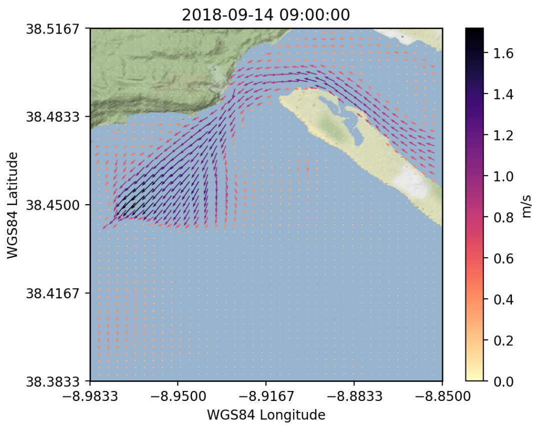
(a)



(b)



(c)



(d)

Figure 6.3: Comparison of snapshots of the model data ((a) and (b)) with the gridded data ((c) and (d)). For visualization purposes the model data was downsampled in space by a factor of 3; the gridded data was downsampled by a factor of 6.

Map tiles by Stamen Design, under CC BY 3.0. Data by OpenStreetMap, under ODbL.

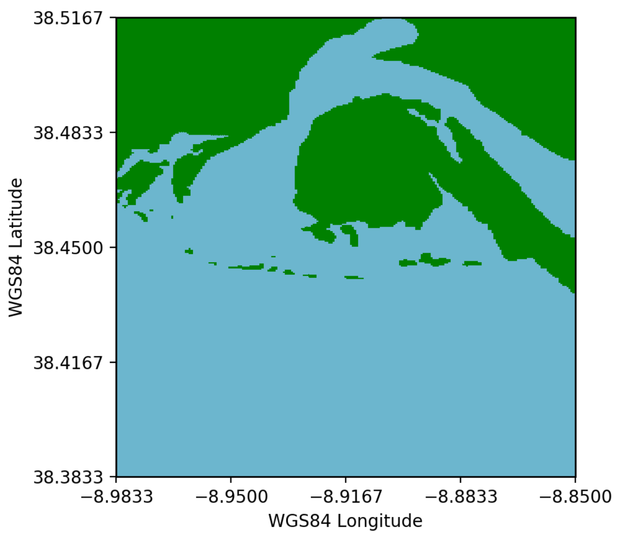


Figure 6.4: Land/water mask constructed from shoreline data and naval charts

6.3.2 First mission: Entering the estuary

The target set in this case is a sphere with a 100 meter radius centered at the point with coordinates (38º N 28’ 37.73”, 8º W 52’ 12.57”).

The integral cost component g is constant and equal to 1, i.e., the integral cost is equal to the time to target. The boundary cost q is zero on the target set and equal to 100 hours on the gridpoints marked as land in the land/water mask. The vehicle’s maximum velocity r was taken to be 1 meter per second.

#	Day	Time	Sea level (m)	Computation time (min)
1	10	09:33	0.5	2.22
2	10	22:03	0.3	2.33
3	11	10:15	0.4	2.27
4	11	22:43	0.4	2.42
5	12	10:55	0.5	2.10
6	12	23:23	0.6	3.32
7	13	11:36	0.7	3.27
8	14	00:02	0.8	3.53
9	14	12:17	0.9	3.48
10	15	00:42	1.0	3.30
11	15	13:03	1.1	3.28
12	16	01:28	1.3	3.13
13	16	13:59	1.4	3.07
14	17	02:27	1.5	2.90
15	17	15:18	1.5	2.70
16	18	03:50	1.6	2.75
17	18	16:54	1.6	2.73
18	19	05:18	1.6	2.87
19	19	18:10	1.5	2.90
20	20	06:22	1.5	3.30
21	20	19:02	1.3	3.33
22	21	07:10	1.3	3.95
23	21	19:42	1.1	3.80
24	22	07:48	1.1	4.02

Table 6.2: Tide periods used to test the first mission. The ‘Time’ column indicates the low tide times.

Source: Tide tables for Sesimbra harbor, Instituto Hidrográfico

We computed the value function for the 24 tide periods shown in Table 6.2, in order to compare the results for different ocean flow data, as well as evaluate the computation time for problems of this size. The low tide times and sea level data were taken from tide charts for the Sesimbra harbor for September 2018. The value function is computed over a 12-hour period starting at the instant 30 minutes before the start of low tide, as the most beneficial currents for the mission objective occur when the water level is rising. For each computation the value function was initialized at the upper bound of $M = 12$ hours (since the cost is the time to target) and the tolerance was set at $\varepsilon = 1 \times 10^{-4}$ hours, which is less than a second.

The computations were performed on a laptop running Linux with a four-core Intel$^{(R)}$ Core^{TM} i7-8550U CPU @ 1.80GHz with 8 parallel threads. The solver code was compiled using the GCC C++ compiler version 9.1.0 with level 3 optimizations enabled. The computation times reported in Table 6.2 measure only the time taken to solve the HJBE, i.e., the time taken to load and write data to disk is not included.

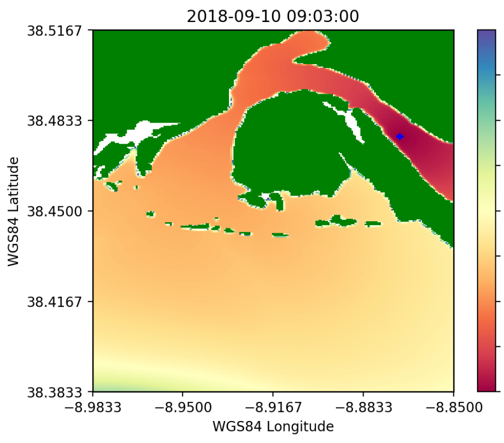
Figure 6.5 shows the values of the optimal cost as a function of the deployment position (i.e., the values of $V(\tau,\cdot)$) for six fixed values of τ, the deployment time, for the first test case. The target set is shown in blue.

Figure 6.5a shows the contours for $\tau = \tau^{\min}$. The highest cost values for this deployment time are about 6 hours, so at the latest the vehicle arrives at the target at high tide, before the currents have reversed direction. This means that the optimal trajectory from each deployment position will face favorable currents, which explains the homogeneity of the cost values across the operational area (modulo the distance to the target). In Figure 6.5b, we can see that at $\tau = \tau^{\min} + 2$ hours some points which are furthest away from the target are unable to reach it within the mission time frame. Notice that although the points on the bottom-left of the operational area are closer to the target than those on the bottom-right, the points on the right side have similar lower cost values, which is due to the currents having a more favorable direction on the right side on the operational area. Four hours after low tide, the current magnitude is starting to decrease, and the vehicle has only two hours to reach the target before facing opposing currents. This is reflected in Figure 6.5c, which shows that the target set is unreachable from most points when starting at that deployment time. Figure 6.5d shows that after 6 hours only the points closest to the target are able to reach it, with the cost increasing rapidly as the distance to the target increases on the left side. In Figures 6.5e and 6.5f we see that the target is only reachable within the mission time frame from points on its right side due to the strong outflow currents.

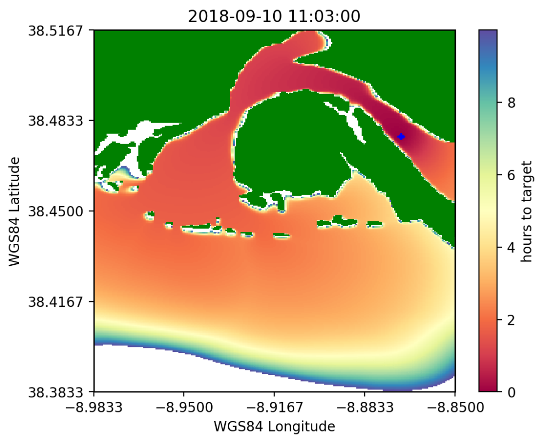
The optimal trajectories are calculated using the method described in 5.7. Here we integrate the ODE using Euler's method with a timestep of one second, and the values of the gradient and the ocean flow in points not on the grid are calculated by linear interpolation. The gradient of the value function is calculated using a second order finite difference formula. Four trajectories generated using the value function corresponding to the first tide period in Table 6.2 are shown in Figure 6.6. The deployment position and the target position are indicated by the blue circle and the red star, respectively. The titles in the figures indicate the deployment time.

The ocean current velocity is superimposed on the trajectory every 15 minutes. Note that once the vehicle has entered the channel, the ocean current is nearly tangent to its trajectory. This can be verified in Figure 6.7, where the flow magnitude along the trajectory and the projection of the flow velocity on the vehicle velocity (the cosine of the angle between the two velocity vectors) are shown.

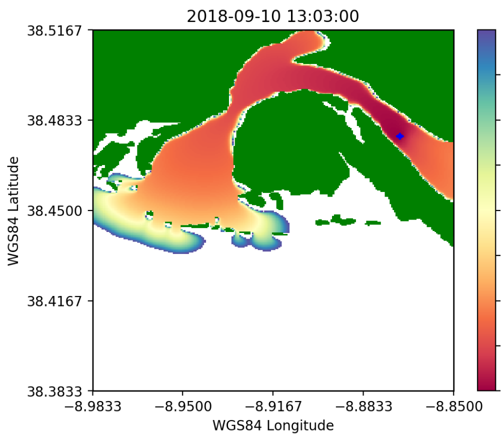
As mentioned in Section 5.1, the motion model does not take into account the vehicle's limited turning rate. We can verify whether the trajectories satisfy this constraint by calculating their



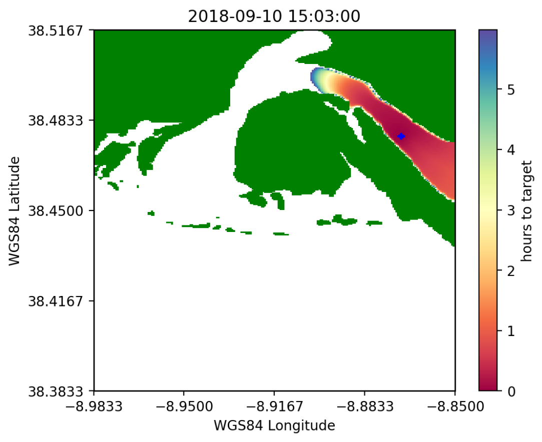
(a)



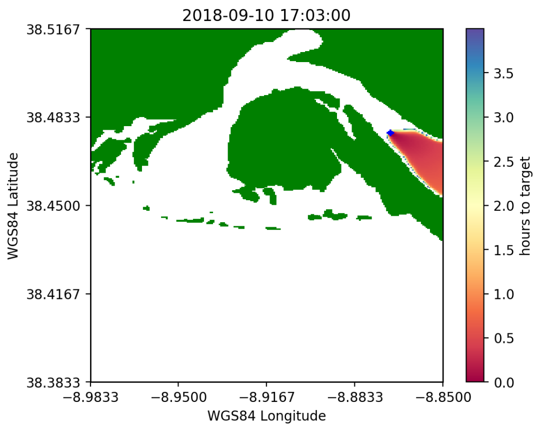
(b)



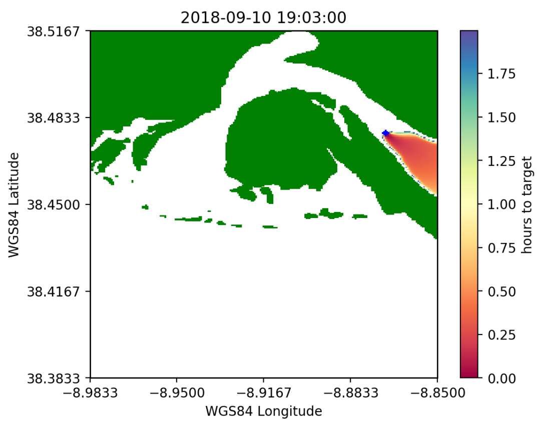
(c)



(d)

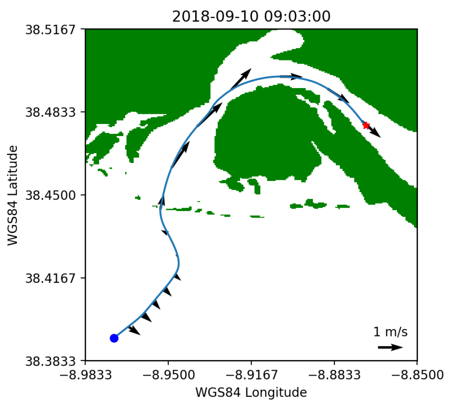


(e)

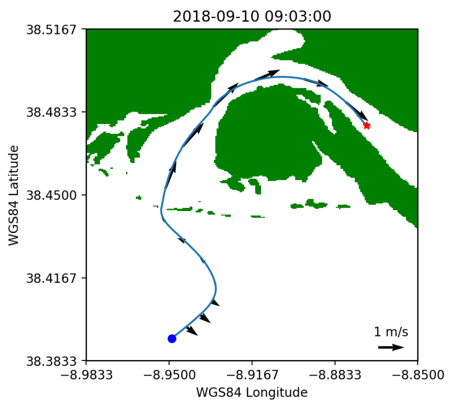


(f)

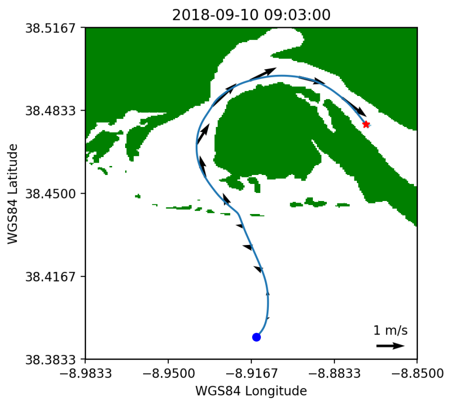
Figure 6.5: Values of $V(\tau, \cdot)$ for six different values of τ, for the first tide period in Table 6.2



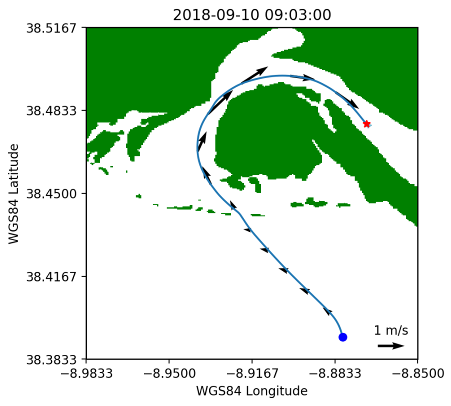
(a)



(b)

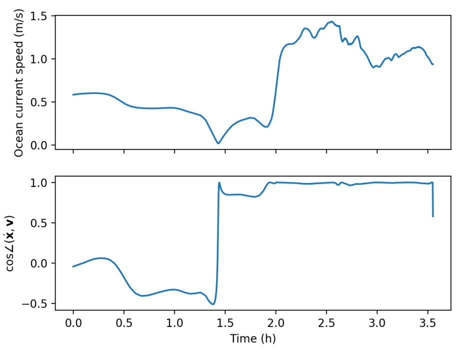


(c)

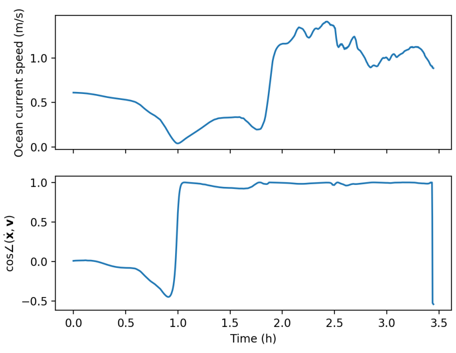


(d)

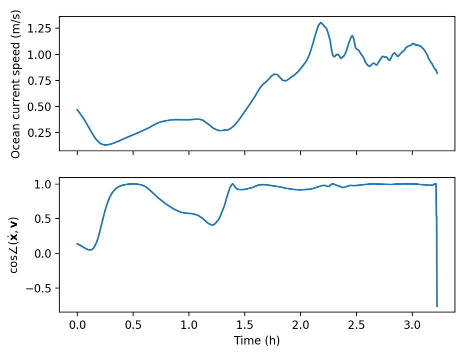
Figure 6.6: Four optimal trajectories calculated using the result of test #1



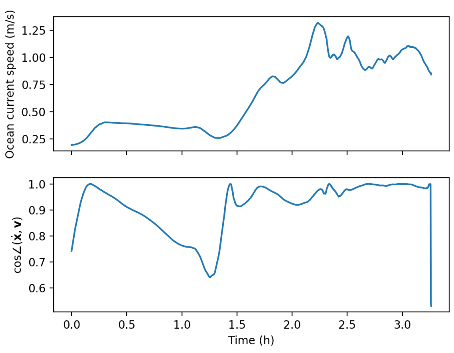
(a)



(b)



(c)



(d)

Figure 6.7: Ocean current magnitude and projection of the ocean current velocity on the vehicle's velocity for the trajectories in Figure 6.6

minimum radius of curvature using the standard formula

$$R_{\min} = \min_t \left| \frac{|\dot{\boldsymbol{x}}|^{3/2}}{\dot{x}_1\ddot{x}_2 - \dot{x}_2\ddot{x}_1} \right|$$

For the trajectories shown in Figure 6.6 the minimum radius of curvature is approximately 306 m, 285 m, 235 m and 197 m (clockwise from the top left), which is well above the minimum for most marine vehicles. For instance, the LAUV class AUVs can track trajectories with radius of curvature above 5 meters when the forward velocity is 1 meter per second [63].

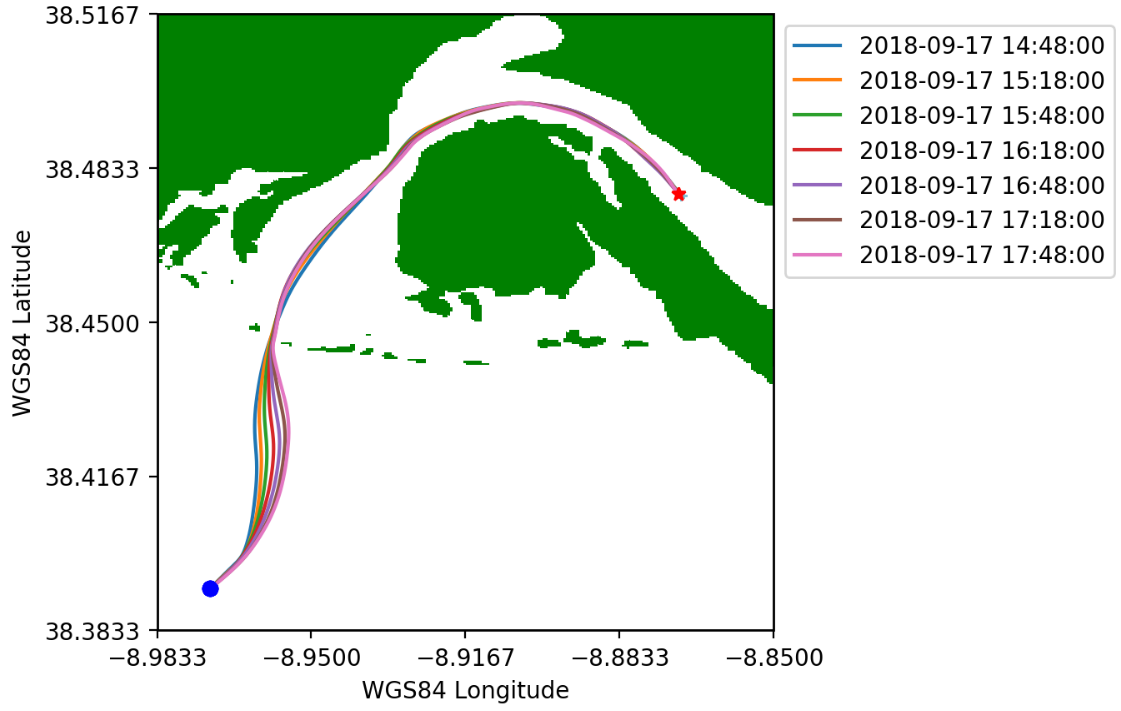
Although we can not measure the error in the value function, as the exact solution is not known, we can get an idea of its accuracy by comparing the value of the value function at the initial condition of the trajectories with the cost of the calculated trajectory, which is easily calculated numerically. In this case the cost is the time taken to reach the target, so it is enough to check the number of time steps in the integration. For the trajectories shown in Figure 6.6, the value of $V(\tau_0, \boldsymbol{x}(\tau_0))$ is 3.82, 3.69, 3.42 and 3.46 hours (clockwise from top left), while the actual time to target for each trajectory is 3.54, 3.42, 3.20 and 3.25 hours. Considering the second set of values to be the true values of the optimal cost, the relative errors are 7.91%, 7.89%, 6.88% and 6.46%. Note that the numerical approximation to the value function is an upper bound of the actual value of the cost, which is not guaranteed by the method, but was to be expected since the value function is initialized with a value which is an upper bound to the true cost. This error is due both to the finite discretization of the computational region and the finite number of iterations. However, note that an error in the value function is not qualitatively relevant for the optimal trajectories unless it changes the direction of the gradient.

As the currents are time-dependent, the optimal trajectories starting from a fixed deployment position at different initial times can be quite different. Figure 6.8 shows trajectories corresponding to deployment times 30 minutes apart over an interval of 3 hours for test cases #1 and #8 in Table 6.2. In Figure 6.8b we observe that the trajectories corresponding to the two last initial times are significantly different from the trajectories departing at earlier times, and in fact the two groups of trajectories approach the island at 38.448870° N, 8.961995° W from different sides. This indicates that the trajectories are indeed globally optimal. If an iterative trajectory generation methods based on improving an initial guess was used, the solution would inevitably approach the obstacle from the same side as the initial guess, which can result in a solution that is only a local minimum.

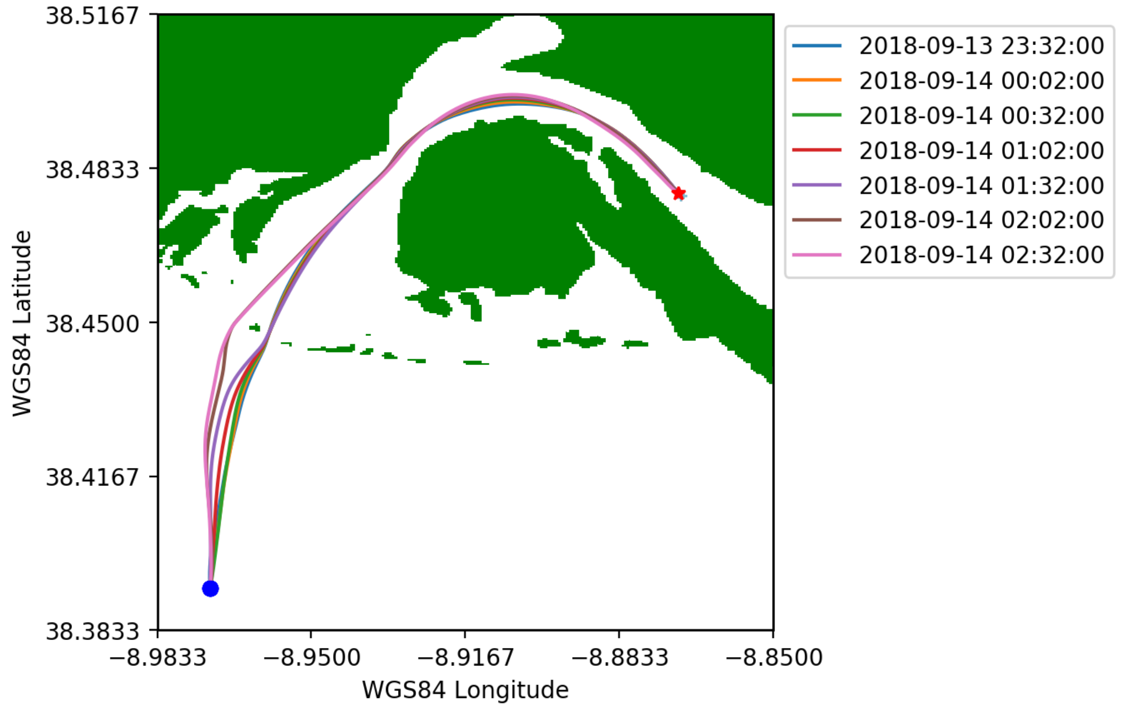
Since the trajectories are time-optimal, one way to measure the impact of the ocean currents on the trajectory is by comparing the distance travelled by the vehicle along the optimal trajectory with the distance the vehicle would travel in the same amount of time if there were no ocean currents, i.e.

$$g = \frac{s}{rT} - 1 \tag{6.1}$$

where s is the arc length of the trajectory, T is the time taken to reach the target and r is as always the speed of the vehicle. This is the total gain in velocity from the ocean currents. The total



(a) Test #1



(b) Test #8

Figure 6.8: Effect of varying the deployment time on the optimal trajectories

arclength s is easily calculated: if the optimal trajectory is $\boldsymbol{x}$ and the result of Euler's method is the set of points $\boldsymbol{x}_0, \ldots, \boldsymbol{x}_n$, then

$$s = \int_{\tau}^{T} |\dot{\boldsymbol{x}}(t)| \mathrm{d}t \approx \sum_{i=0}^{n-1} |\boldsymbol{x}_{i+1} - \boldsymbol{x}_i|. \tag{6.2}$$

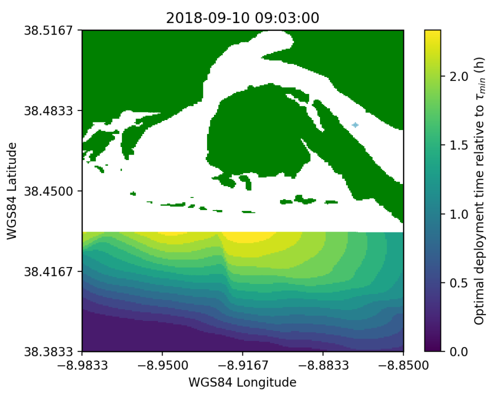
For the trajectories shown in Figure 6.6, the values of the velocity gain g are approximately 44.3%, 51.2%, 55.0% and 60.5% (clockwise from the top left).

Besides its role in trajectory generation, the value function contains information which can be useful for mission planning. For instance, we can calculate the optimal deployment time for each possible deployment position by finding the value of τ which minimizes $V(\tau, \boldsymbol{x})$ for fixed $\boldsymbol{x}$. This is shown in Figure 6.9 for an approximately 11.7 km by 5.7 km area in the lower latitudes of the operational area, for some of the test runs in Table 6.2. For each point in the area the color indicates the optimal deployment time in hours relative to τ_{min}, the first time instant for which the value function is computed, which is indicated in the title.

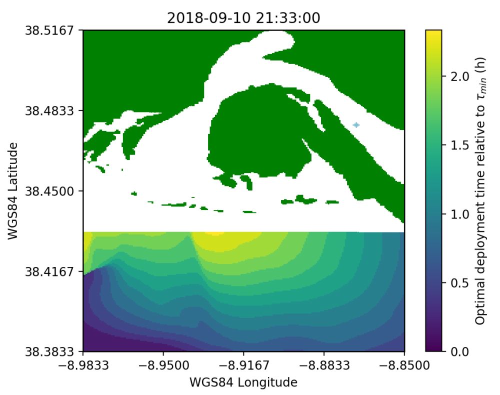
Note that the plots are qualitatively different, which shows that details in the ocean flow play a major role, and the optimal cost is not solely determined by the geometry of the operational area. In fact, note the discontinuity in Figures 6.9b and 6.9f. This is not a numerical artifact, but reflects the fact that the optimal trajectories departing from these points at the optimal deployment time are qualitatively quite different, as shown in Figure 6.10. The large areas with optimal deployment time equal to τ_{min} in Figures 6.9a, 6.9d and 6.9e (dark blue) suggest that deployment times earlier than τ_{min} could lead to smaller optimal cost. Hence this can also be used to adjust the computational boundary after an initial guess based on the tide tables.

We can plot the velocity gain g corresponding to the trajectory which departs from each point in the considered area at the optimal deployment time, as in Figure 6.11. Once again the maps corresponding to different tests are qualitatively different, and some of the patterns visible in Figure 6.9 are apparent. There are more discontinuities, since the velocity gain is strongly influenced by the geometry of the trajectories, and here too the discontinuities separate regions where the trajectories are qualitatively different. This can be verified in Figure 6.12, where pairs optimal trajectories starting from points which are close but on different sides of the discontinuities in Figure 6.11e are shown.

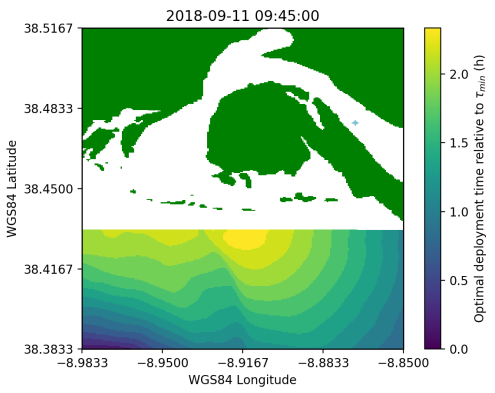
Note that in Figures 6.11b and 6.11f, in the left side of the considered area, there are some points where the gain value is missing. This is because the optimal trajectory from these points is not contained in the operational area, so it cannot be calculated. This unavoidable if the flow is strong and pointing towards the outside of the computational region, and should be taken into account. One way of roughly predicting this phenomena is to check points near the computational boundary where the ocean current speed is larger than the vehicle's speed, and in any case the areas considered for deployment should not be too close to the computational boundary.



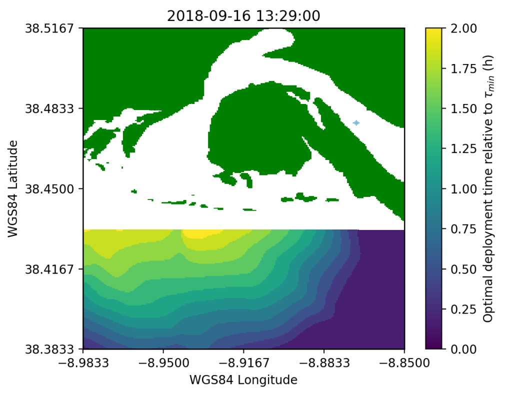
(a) Test #1



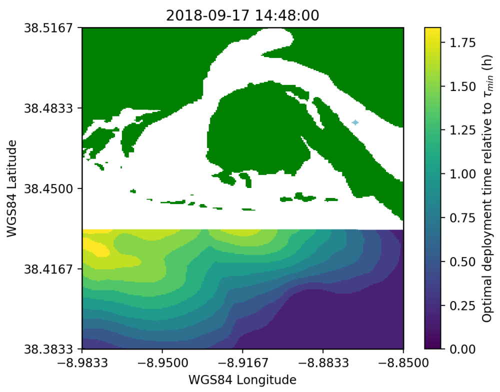
(b) Test #2



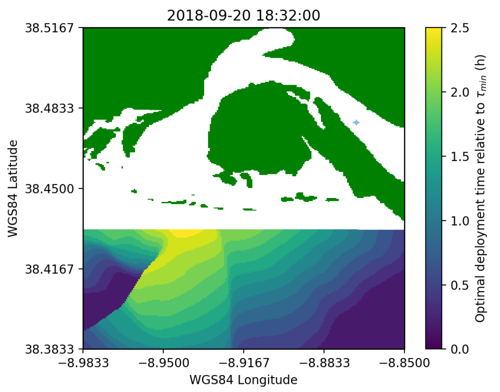
(c) Test #3



(d) Test #13



(e) Test #15



(f) Test #21

Figure 6.9: Maps of the optimal deployment time in a subset of the deployment area

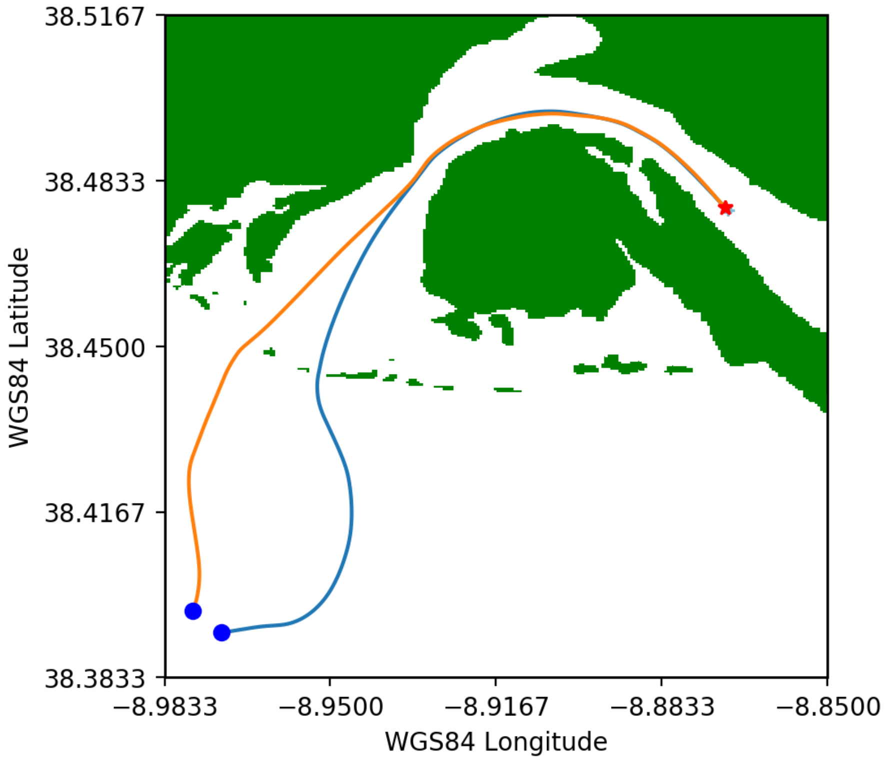
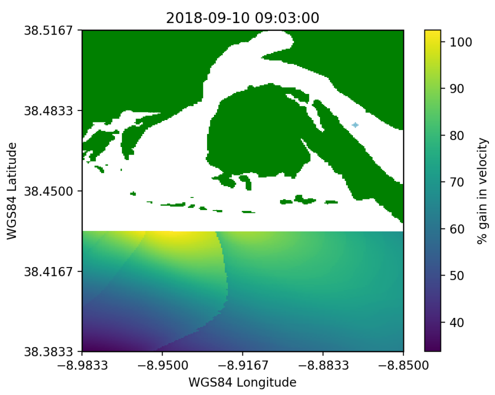
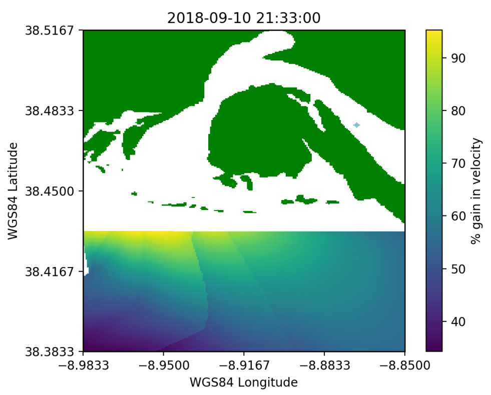


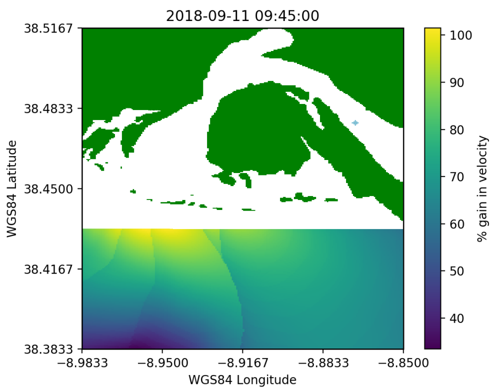
Figure 6.10: Two optimal trajectories corresponding to two different sides of the discontinuity in Figure 6.9f



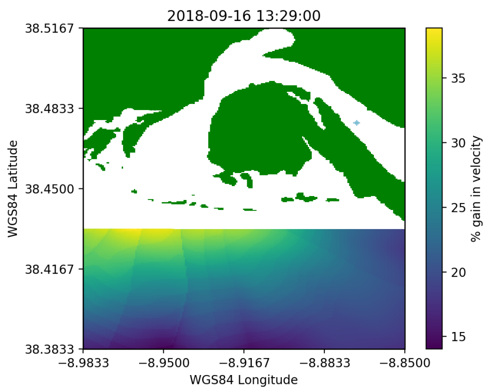
(a) Test #1



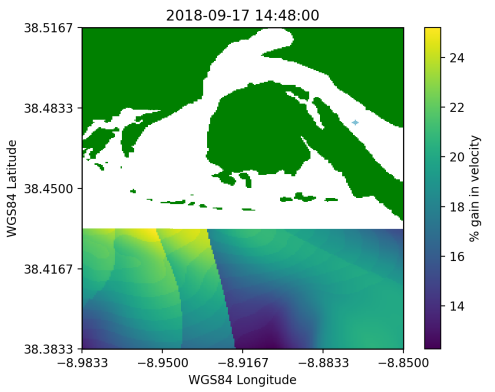
(b) Test #2



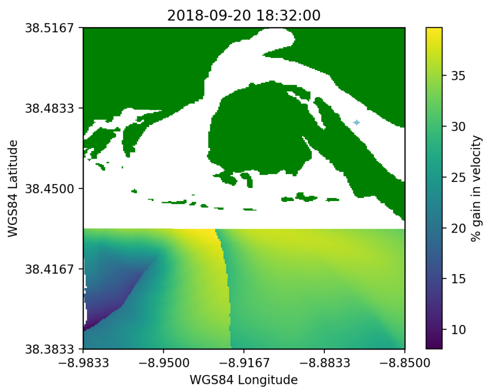
(c) Test #3



(d) Test #13



(e) Test #15



(f) Test #21

Figure 6.11: Maps of the velocity gain in a subset of the operational area

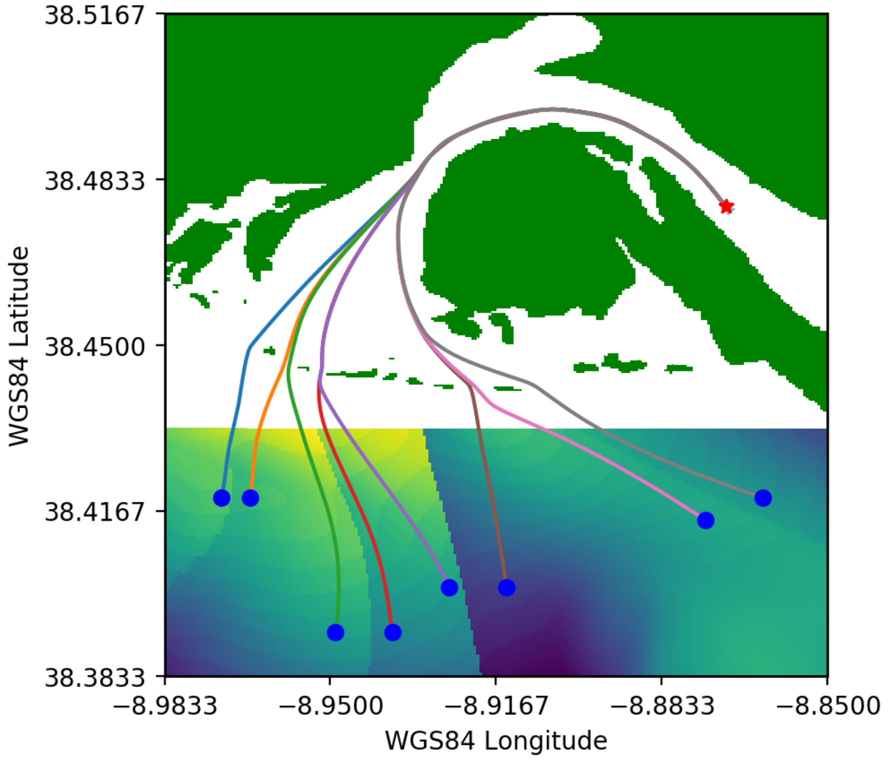


Figure 6.12: Trajectories starting on different sides of the discontinuities in Figure 6.11e are geometrically distinct

6.3.3 Second mission: Exiting the estuary

For this second mission, the target is a line segment at a fixed latitude, represented in black in Figure 6.13.

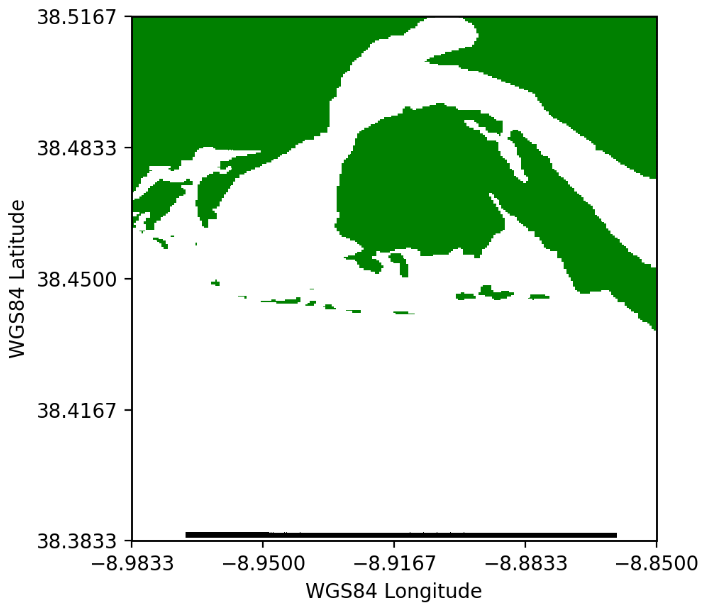


Figure 6.13: Target location for the second mission

The value function is computed for the 24 tide periods listed in Table 6.3 The same hardware, compiler and parameters as in Section 6.3.2 were used for the computation. As in the first mission, $\tau_{\min}$ is equal to the times listed in Table 6.3 minus 30 minutes.

Figure 6.14 shows the values of $V(\tau, \cdot)$ for siz fixed values of the deployment time τ, where V is the value function computed for the first tide period listed in Table 6.3

Some optimal trajectories starting from (38º N 28' 37.73", 8º W 52' 12.57") are shown in Figure 6.15. These are again computed using Euler's method with a timestep of 1 second. Note that the arrival position can vary significantly due to differences in the ocean current.

Table 6.4 shows the values of the parameters that were discussed in Section 6.3.2. The first row is the gain in the velocity, given by (6.1). The second row is the minimum radius of curvature, which, as before, is high enough for a typical ocean vehicle to be able to track the trajectories. The third row is the value of $V(\tau_0, \boldsymbol{x}(\tau_0))$ in hours, while the fourth row is the actual time taken by the trajectory to reach the target. As in Section 6.3.2, the value function appears to be an upper bound of the actual optimal time to target. The fourth row is the relative error in the numerical value of the value function $V(\tau_0, \boldsymbol{x}(\tau_0))$, considering that the true value of the cost is given by the values in the third row.

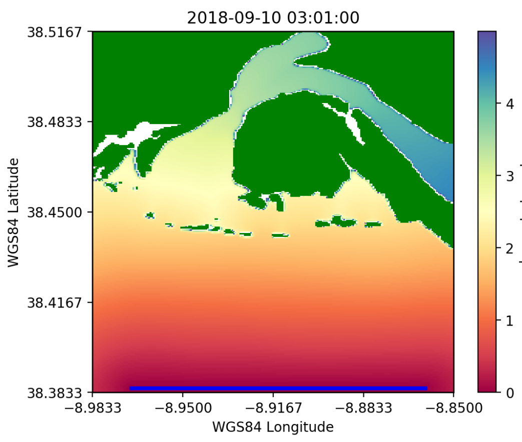
#	Day	Time	Sea level (m)	Computation time (min)
1	10	03:31	3.6	2.42
2	10	15:49	3.8	2.35
3	11	04:13	3.6	2.47
4	11	16:32	3.8	3.03
5	12	04:55	3.5	2.48
6	12	17:14	3.6	3.17
7	13	05:36	3.4	2.48
8	13	17:56	3.4	2.40
9	14	06:17	3.2	2.37
10	14	18:39	3.1	2.33
11	15	07:01	3.0	2.25
12	15	19:26	2.9	2.25
13	16	07:51	2.8	2.05
14	16	20:23	2.6	2.08
15	17	08:56	2.6	1.90
16	17	21:40	2.5	1.87
17	18	10:20	2.6	1.72
18	18	23:13	2.4	1.92
19	19	11:40	2.7	1.77
20	20	00:22	2.6	2.07
21	20	12:39	2.8	1.93
22	21	01:11	2.7	2.20
23	21	13:23	3.0	2.08
24	22	01:49	2.9	2.28

Table 6.3: Tide periods used to test the second mission. The 'Time' column indicates the high tide times.

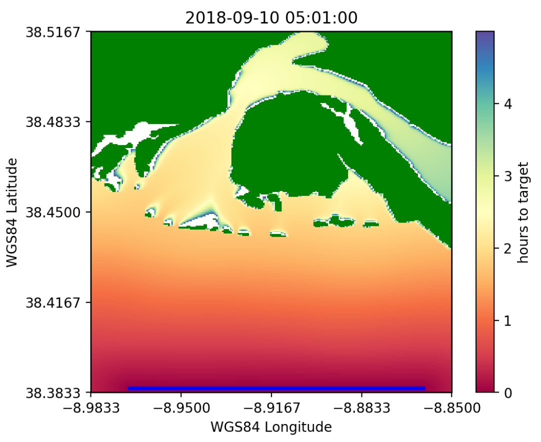
Source: Tide tables for Sesimbra harbor, Instituto Hidrográfico

Figure	6.15a	6.15b	6.15c	6.15d	6.15e	6.15f
Gain g	82.90%	71.23%	79.50%	50.84%	38.67%	40.79%
Min. radius of curvature (m)	497	658	521	298	636	279
Value of V (h)	3.22	3.40	3.19	3.62	4.00	3.81
Time to target (h)	3.00	3.19	2.98	3.42	3.83	3.65
Relative error	7.19%	6.69%	7.04%	5.94%	4.41%	4.47%

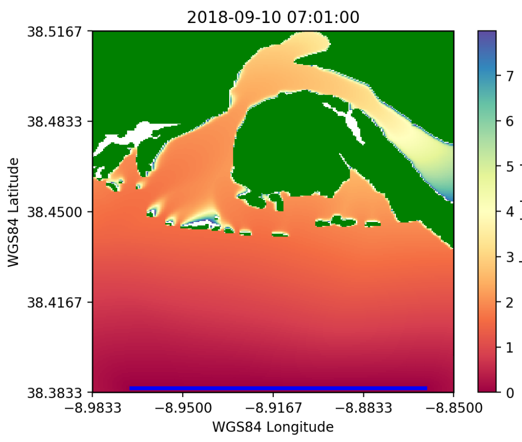
Table 6.4: Indicators for the trajectories in Figure 6.15.



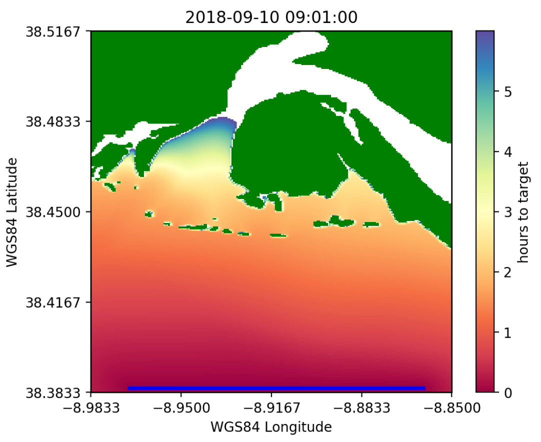
(a)



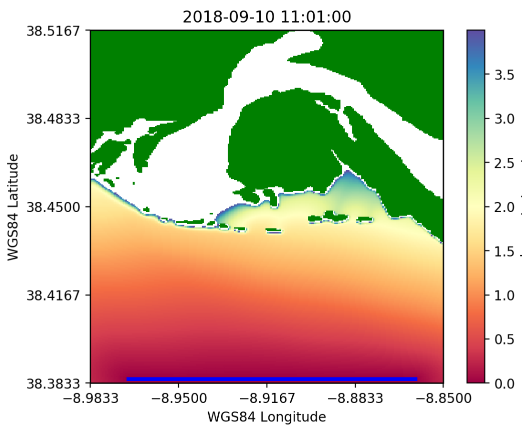
(b)



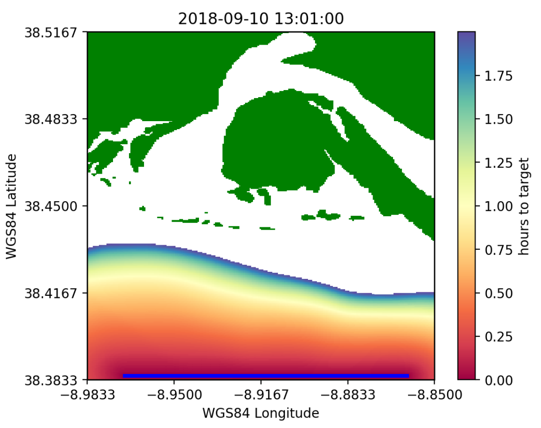
(c)



(d)

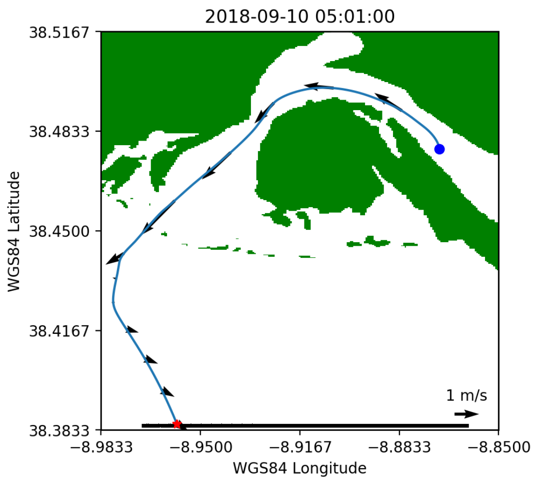


(e)

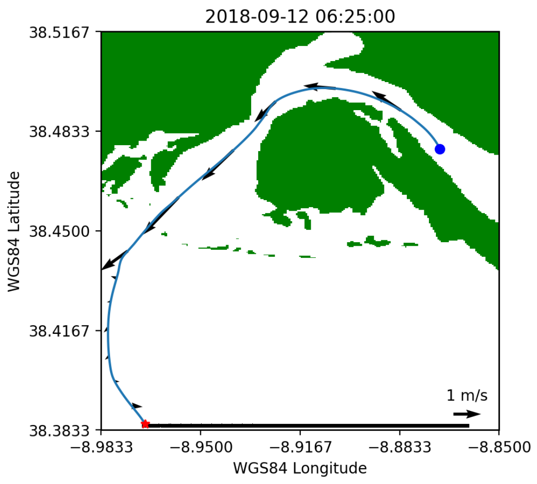


(f)

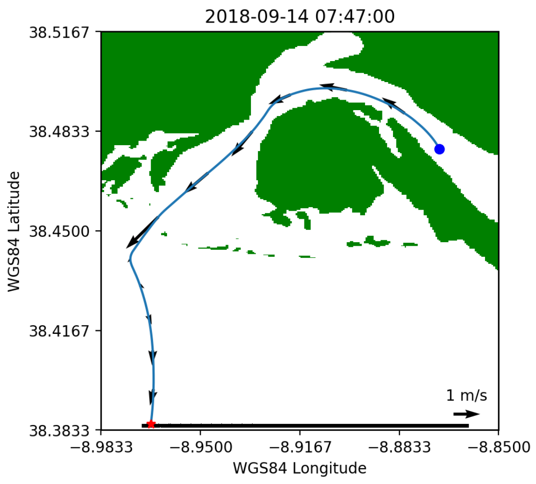
Figure 6.14: Values of $V(\tau, \cdot)$ for six different values of τ, for the value function corresponding to the first tide period listed in Table 6.3



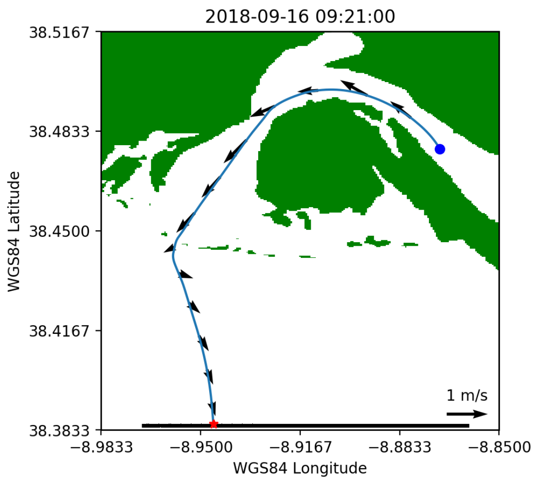
(a) Test #1



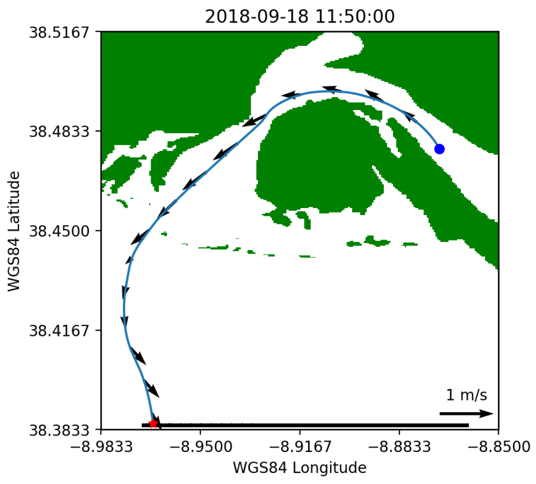
(b) Test #5



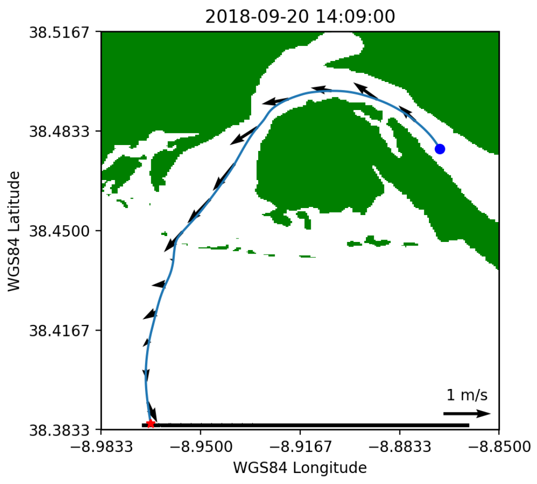
(c) Test #9



(d) Test #13



(e) Test #17



(f) Test #21

Figure 6.15: Six optimal trajectories departing from the same point

6.4 Currents with high spatial variability

It is not easy to obtain a theoretical bound on the radius of curvature of the optimal trajectories generated from the solution of the HJBEs we have studied. When there are no currents, the HJBE for the base problem reduces to an eikonal equation, and in this case, the curvature of a trajectory passing through a point $\boldsymbol{x}$ is upper bounded by the quantity

$$r\frac{|\nabla g(\boldsymbol{x})|}{g(\boldsymbol{x})},$$

so that the curvature of the trajectories is controlled by the cost function, as expected [72].

When there are currents, if the cost function does not lead to trajectories with high curvature, we expect that an optimal trajectory will contain a sharp turn only if there is an abrupt change of direction or magnitude in the ocean flow. This is often associated with a region of nonsmoothness (a 'shock') in V, which leads to nonsmooth optimal trajectories. In Aguiar et al. [1], two ideal flow fields are studied, where transitions between regions with considerably different ocean flow directions appear to be linked with shocks in the value function and nonsmooth trajectories. However, as remarked in Rhoads et al. [48], in realistic time-varying flows it is unlikely that shocks will form due to the lack of symmetry in the flow.

Hence, the hypothesis is that in situations where $\boldsymbol{v}$ models a physical ocean flow the optimal trajectories should be smooth and have large radius of curvature at all points, relative to the typical radius of curvature restrictions of unmanned underwater vehicles. This has been the case for the results shown in the previous section. To partially confirm the hypothesis, we use data from an ocean model of the Sado and Tejo estuaries in extreme conditions to evaluate the feasibility of the trajectories generated by the method.

The data spans 12 hours with a temporal resolution of 4 minutes and is defined over a curvilinear grid with a mean resolution of 300 meters. The region where the data is defined is shown in Figure 6.16. We selected an approximately 22.45 km by 23.09 km square region where the flow has abrupt changes of direction, indicated by the blue square in Figure 6.16.

Four snapshots of the data are shown in Figure 6.17. On the southwestern corner of the considered region, the flow generally point towards the southeast. From the southeastern corner towards the middle, there is a region of the flow which points towards the northeast. Note the high amplitudes in this portion of the flow. On the northeastern corner of the square, the flow changes direction again, pointing towards the southeast.

We compute the value function for a minimum time problem on a grid with a spatial resolution of 200 meters and a temporal resolution of 4 minutes. The solver settings and platform are the same as in the previous examples, and the computation took 0.75 minutes. The target set is a sphere with a 400 m radius centered at the point (38^o N 33' 45.72", 9^o W 19' 45.48) (in the northeastern corner of the operational area). In this case there is no need to account for obstacles since the operational area is in deep open sea waters, so the curvature of the optimal trajectories is influenced only by the ocean currents.

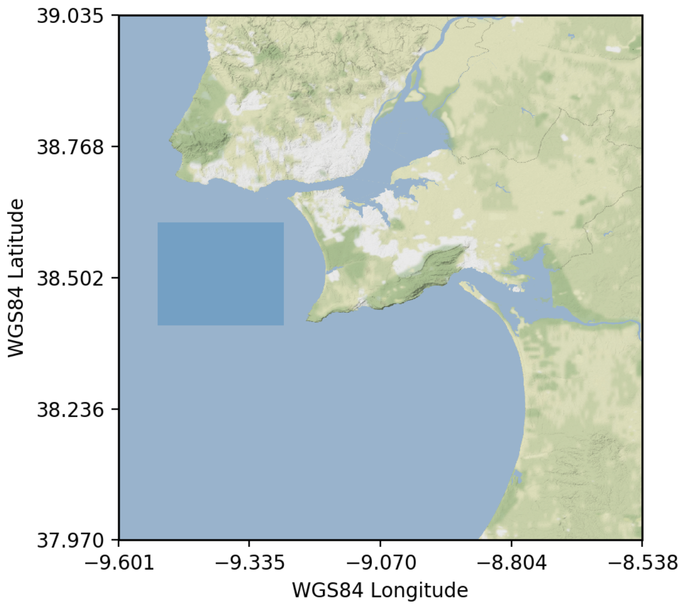
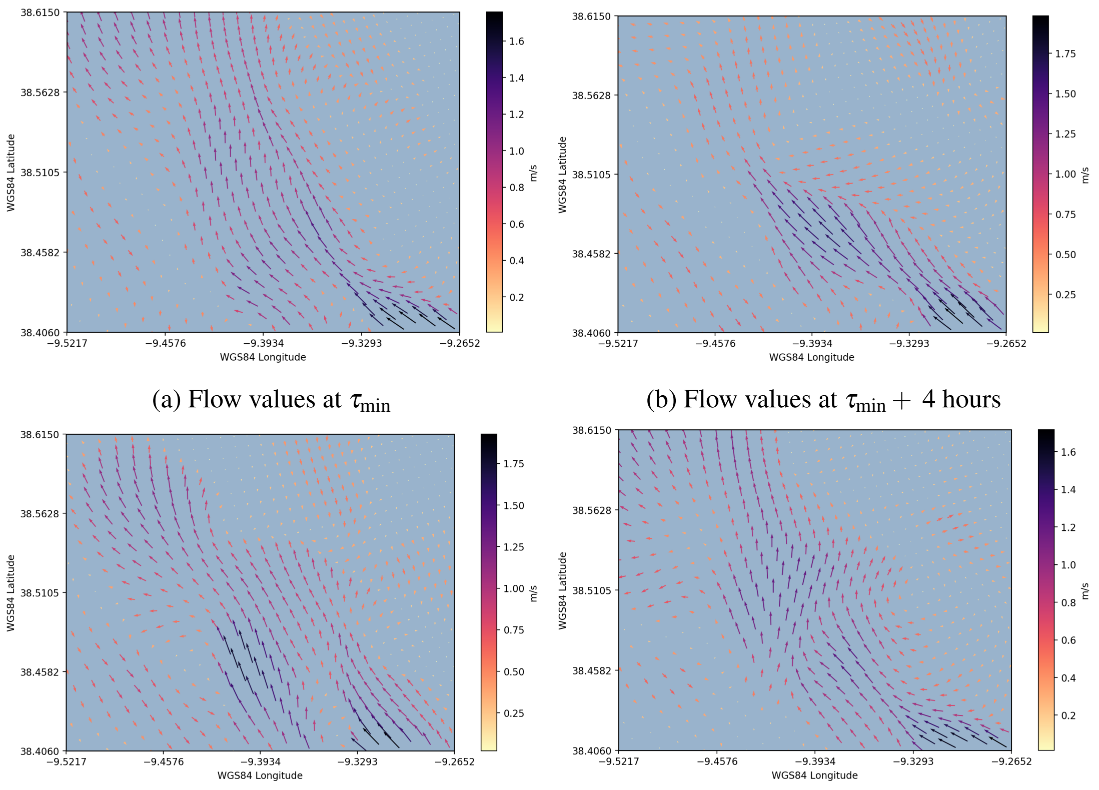


Figure 6.16: Map and operational area. Map tiles by Stamen Design, under CC BY 3.0. Data by OpenStreetMap, under ODbL.



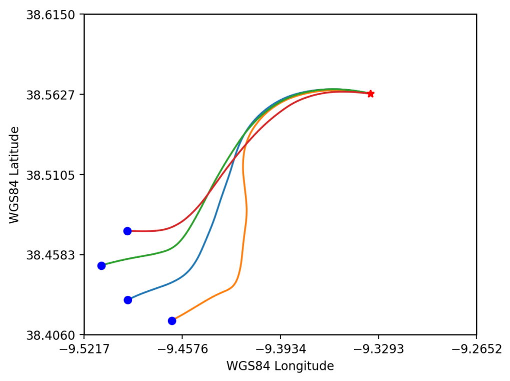
(a) Flow values at $\tau_{\min}$

(b) Flow values at $\tau_{\min} +$ 4 hours

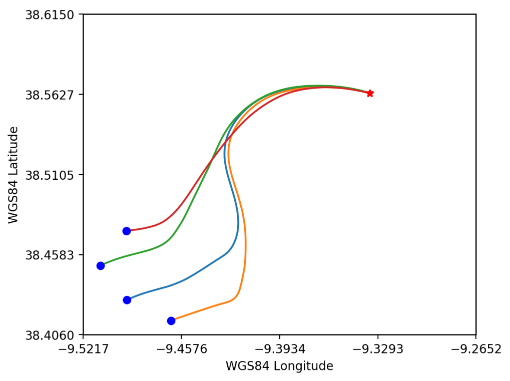
(c) Flow values at $\tau_{\min} +$ 8 hours

(d) Flow values at $\tau_{\min} +$ 12 hours

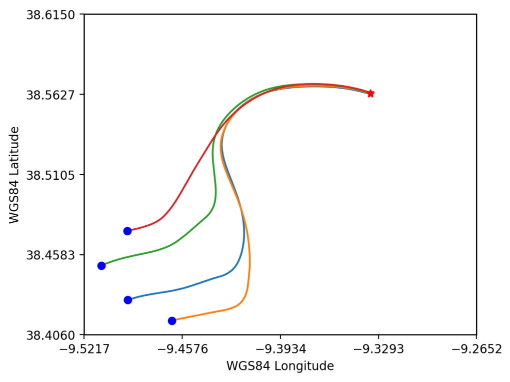
Figure 6.17: Four snapshots of the data.



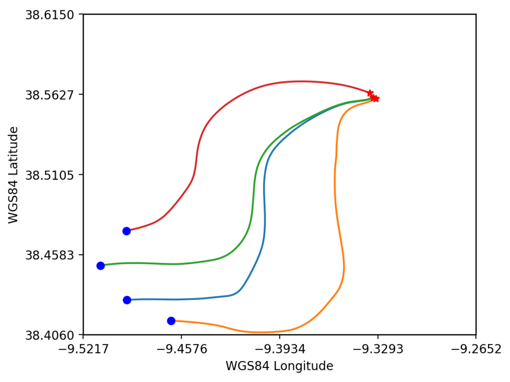
(a) Trajectories departing at $\tau_{\min}$



(b) Trajectories departing at $\tau_{\min} +$ 1 hour

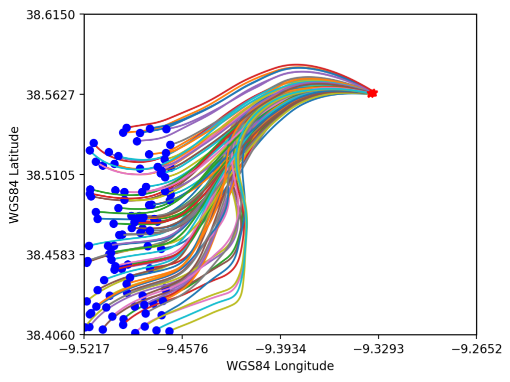


(c) Trajectories departing at $\tau_{\min} +$ 2 hours

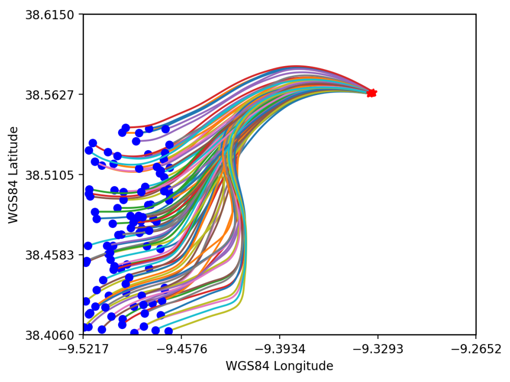


(d) Trajectories departing at $\tau_{\min} +$ 3 hours

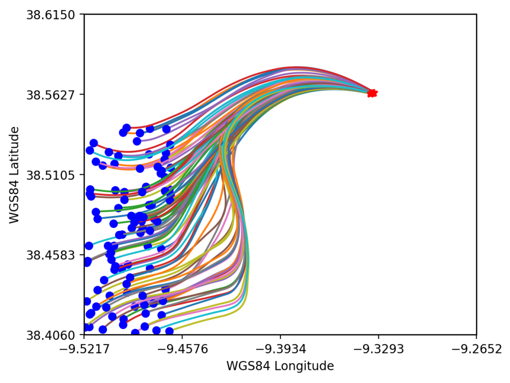
Figure 6.18: Optimal trajectories departing from four different points on the southwest of the operational area



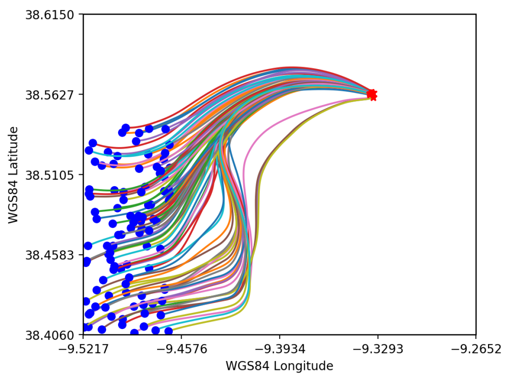
(a) Trajectories departing at $\tau_{\min}$



(b) Trajectories departing at $\tau_{\min} +$ 0.5 hours



(c) Trajectories departing at $\tau_{\min} +$ 1 hour



(d) Trajectories departing at $\tau_{\min} +$ 1.5 hours

Figure 6.19: Optimal trajectories departing from 100 uniformly distributed points on the southwest of the operational area.

Figure 6.18 shows that the trajectories indeed take relatively sharp turns when crossing the interfaces between regions in the flow with distinct behavior. However, the radius of curvature of each of the trajectories shown in Figure 6.18 is lower bounded by 600 meters.

Figure 6.19 shows trajectories departing from 100 randomly selected points from a uniform distribution (in UTM coordinates) over a rectangular 5 km by 15 km selection of the operational area. The minimum radius of curvature among all the trajectories is 360 meters, and only 7 of the 400 trajectories have a minimum radius of curvature smaller than 600 meters. Hence, the results confirm the hypothesis that for ‘physical’ models of the ocean currents the optimal trajectories are feasible, at least as long as the cost function does not induce sharp turns in the trajectories.

6.5 Software-in-the-loop simulations

In this section we report results of simulations of the generated trajectories with the LSTS toolchain. The purpose of these simulations is twofold. First, we want to confirm that an ocean vehicle can satisfactorily track the trajectories generated by the method. Second, we want to understand how the method can be integrated in existing frameworks for unmanned vehicle missions, such as the LSTS toolchain.

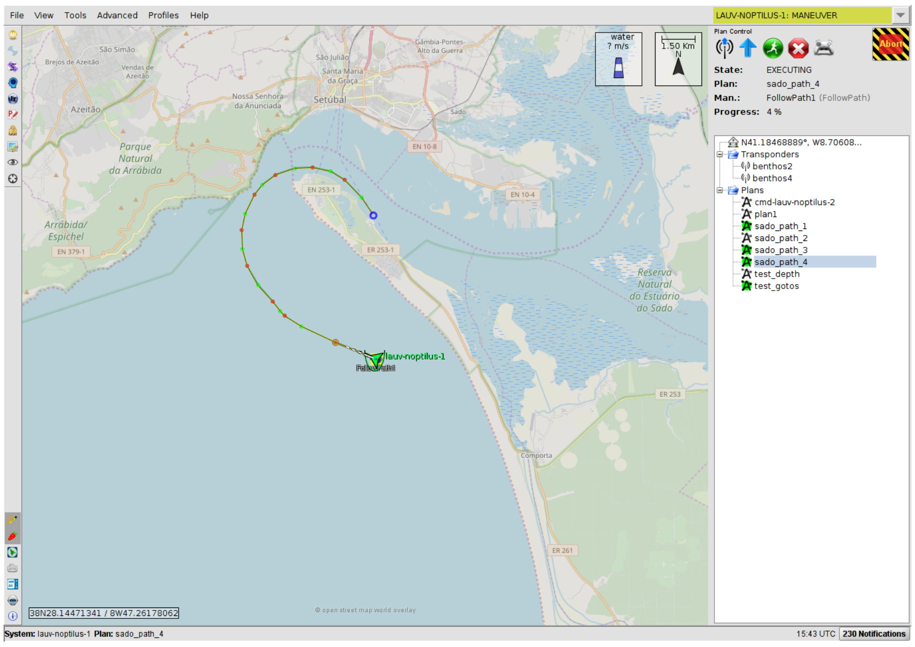
In Neptus, a curved trajectory can be defined using the `FollowPath` or the `FollowTrajectory` maneuvers. The `FollowPath` maneuver specifies the trajectories as a sequence of North, East and Down direction displacements given in meters, relative to an initial point specified in WGS84 coordinates. The `FollowTrajectory` maneuver is similar, except it allows the specification of an arrival time at each point relative to the time at which the maneuver starts being executed. While executing a `FollowTrajectory` maneuver, the vehicle varies its speed in order to reach each point as close to its associated time as possible. While our method generates trajectories and not just geometric paths, we want the vehicle to travel at constant speed, so we use the `FollowPath` maneuver.

The curved trajectories generated by the method must be sampled in order to represent them as `FollowPath` maneuvers. Sampling uniformly in time, however, would lead to a large number of points to represent a trajectory. Since each of the points in a `FollowPath` maneuver is treated as a waypoint, there is a heading transient every time a point is reached, so a large number of points leads to bad tracking performance. The solution is to have few sample points when the trajectory is nearly a straight line and more sampling points when the trajectory has high curvature. This is done using a method similar to that described in Pagani and Scott [43]. We define a new parameter q as

$$q(t) = \alpha \frac{s(t)}{s(T)} + (1-\alpha)\frac{\int_{t_0}^{\tau} k(\tau)\mathrm{d}\tau}{\int_{t_0}^{T} k(\tau)\mathrm{d}\tau} \tag{6.3}$$

where s is the arclength parameter, k is the positive curvature and α is an adjustable parameter. Then we sample the trajectory $\boldsymbol{x}(t)$ uniformly along this parameter, i.e. the sample points are $\boldsymbol{x}(t_0), \boldsymbol{x}(t_0 + q^{-1}(h)), \boldsymbol{x}(t_0 + q^{-1}(2h)), \ldots, \boldsymbol{x}(T)$, where $h = \frac{1}{N-1}$ for a given number N of sample points. The α parameter controls the weight between sampling at a fixed distance between sampling points or sampling only according to the variation in the curvature.

Since DUNE's simulation engine only allows setting a fixed value for the ocean current, we implemented an ocean current simulator and integrated in the simulation engine. A new simulation task was implemented which consumes the estimated position of the vehicle, reads current data from an HDF5 format file, interpolates it to the current position of the vehicle and dispatches it in an IMC `EstimatedStreamVelocity` message. This message is in turn consumed by the vehicle simulation engine, which uses the value of the ocean current velocity to update the vehicle's position. In keeping with the design principles described in Section 2.4, the vehicle simulator is independent of the source of the ocean current velocity values, so that in other scenarios other sources of data (e.g., remote sensing) can be used without changing the simulator source code.



LAUV-NOPTILUS-1: MANEUVER

Plan Control

State: EXECUTING
Plan: sado_path_4
Man.: FollowPath1 (FollowPath)
Progress: 4 %

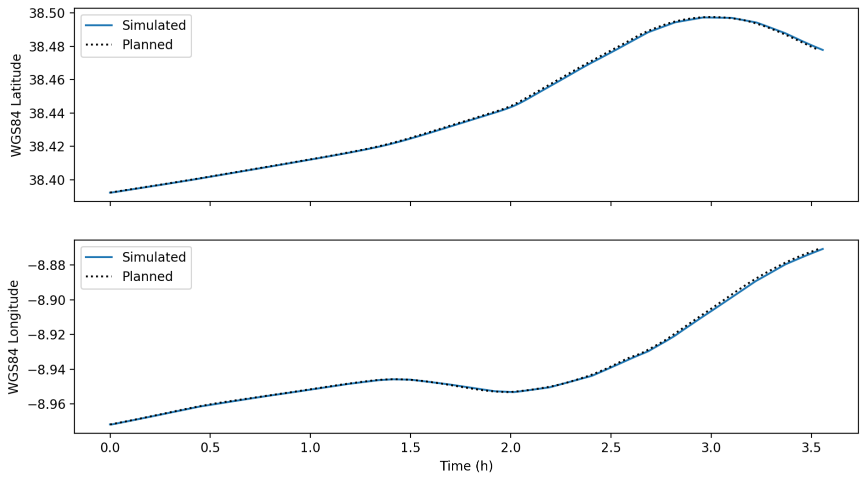
N41.18468889°, W8.70608...
Transponders
benthos2
benthos4
Plans
cmd-lauv-noptilus-2
plan1
sado_path_1
sado_path_2
sado_path_3
sado_path_4
test_depth
test_gotos

System: lauv-noptilus-1 Plan: sado_path_4 15:43 UTC 230 Notifications

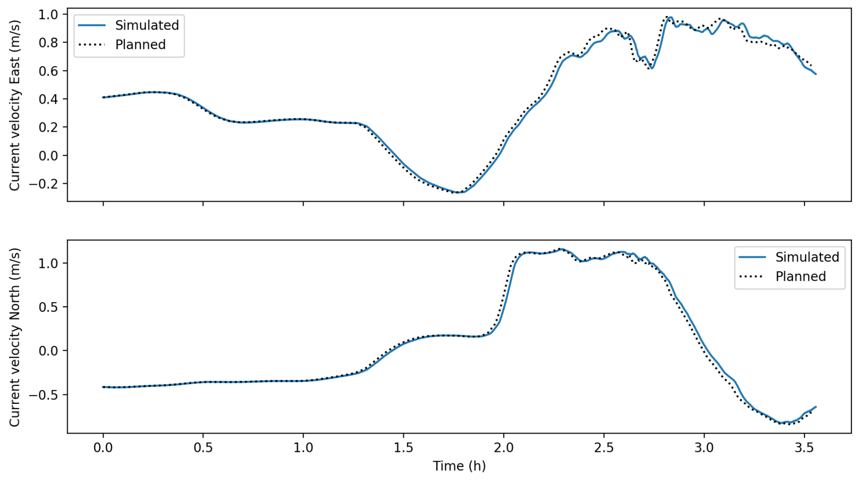
Figure 6.20: The third test trajectory in the Neptus operator console

We simulate three trajectories from the first test case described in Section 6.3.2 using the DUNE configuration file for LSTS's LAUV Noptilus 1 AUV. Figure 6.20 shows the vehicle tracking the third test trajectory in the Neptus operator console.

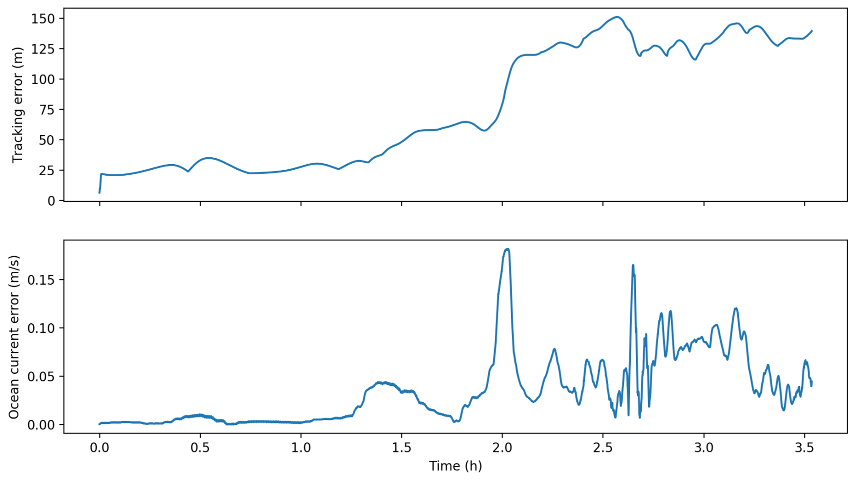
The results are shown in Figures 6.21 to 6.23. Note that the tracking errors are low relative to the scale of the trajectories and the spatial resolution of the computational grid (50 m). To further improve the results, one could take advantage of the feedback form of the solution, and the trajectory could be replanned online by the vehicle, either periodically in a receding-horizon type scheme, or when a certain tracking error threshold is exceeded.



(a) Simulated and planned position

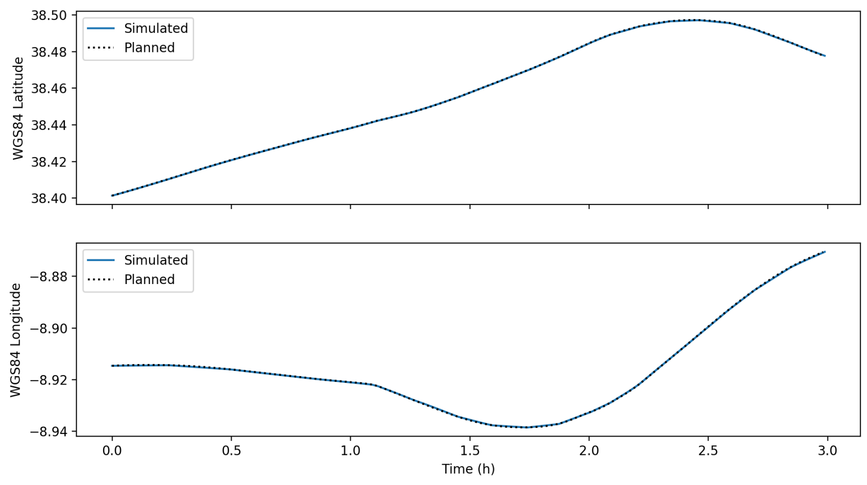


(b) Simulated and planned ocean current velocity components along the trajectory

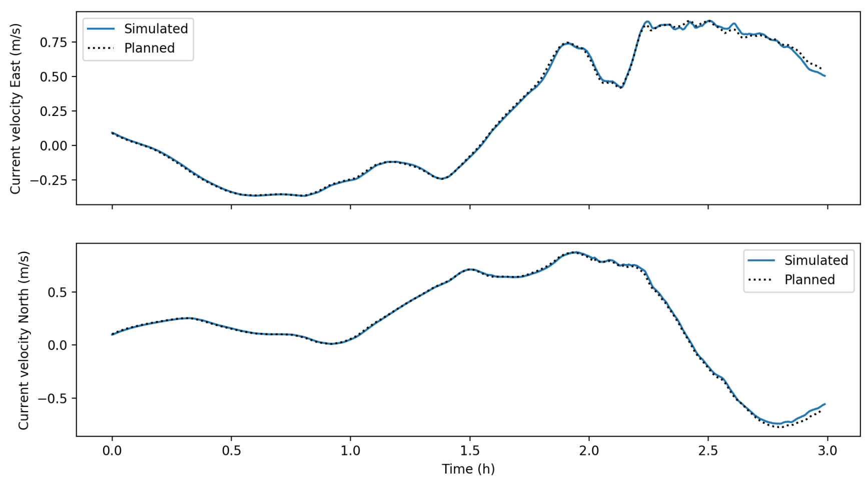


(c) Tracking error and Euclidean norm of the difference between the planned and simulated ocean current velocities.

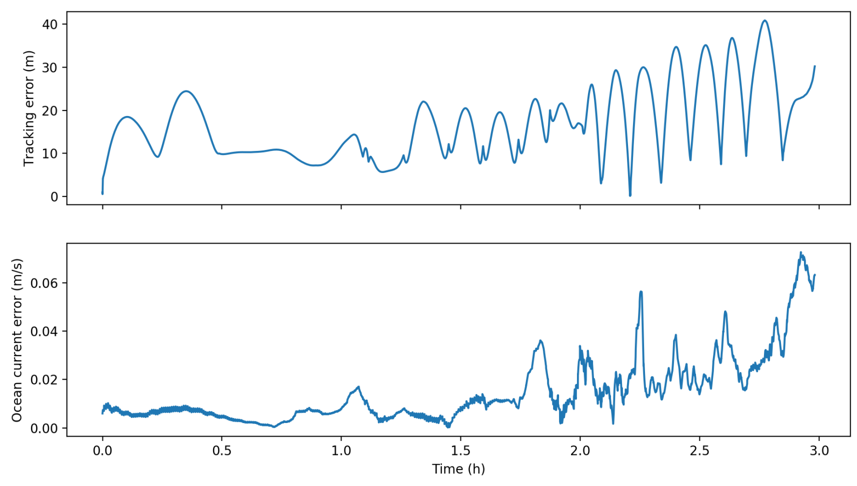
Figure 6.21: Results of the first software-in-the-loop simulation



(a) Simulated and planned position

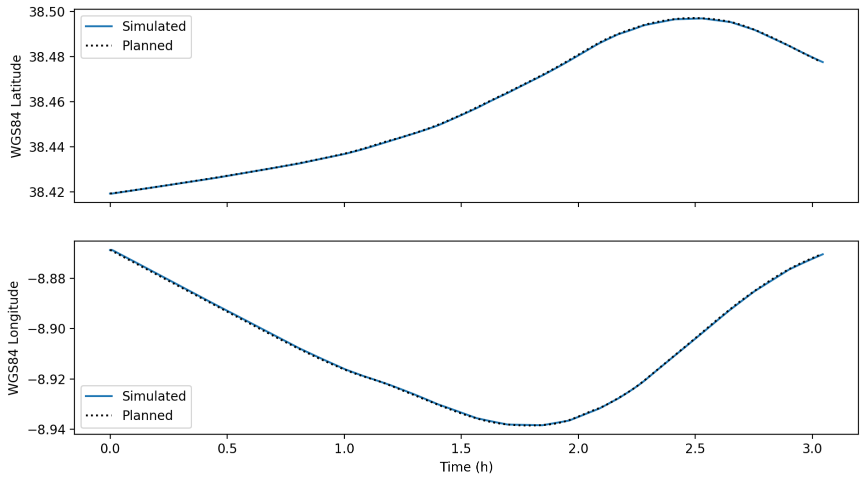


(b) Simulated and planned ocean current velocity components along the trajectory

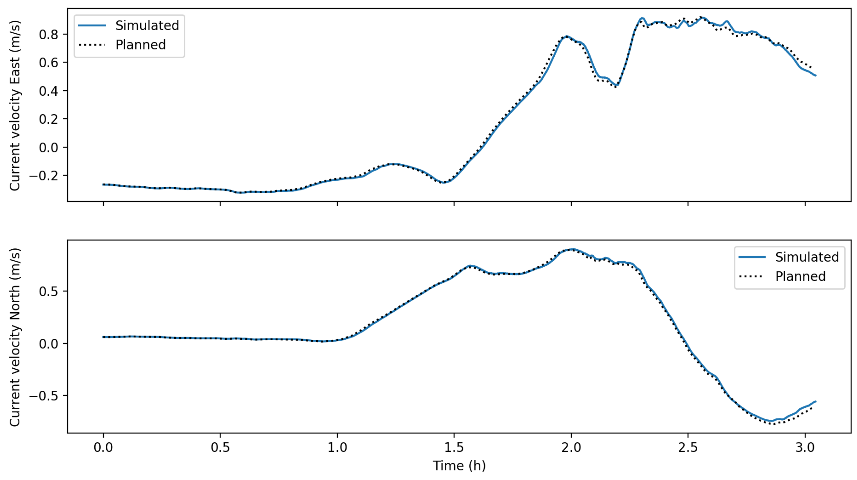


(c) Tracking error and Euclidean norm of the difference between the planned and simulated ocean current velocities.

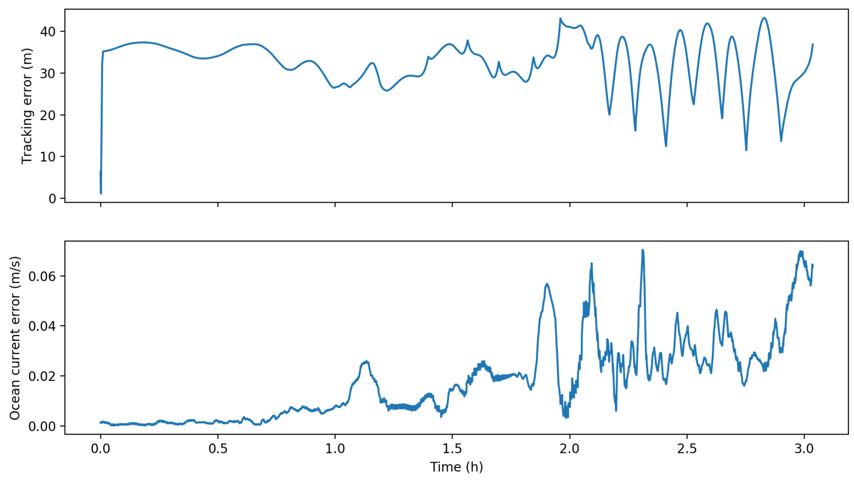
Figure 6.22: Results of the second software-in-the-loop simulation



(a) Simulated and planned position



(b) Simulated and planned ocean current velocity components along the trajectory



(c) Tracking error and Euclidean norm of the difference between the planned and simulated ocean current velocities.

Figure 6.23: Results of the third software-in-the-loop simulation

Chapter 7

Conclusions and Future Work

In that flash I realized, it's impossible to fall off mountains, you fool!

Jack Kerouac (in *The Dharma Bums*)

7.1 Summary

We presented a dynamic programming-based approach to solver single- and multi-stage optimal trajectory generation problems for unmanned ocean vehicles subject to forcing by currents. In the simplest case of generating a single trajectory from the deployment position to a target region, the approach reduces the problem to the solution of a nonlinear first-order partial differential, the Hamilton-Jacobi-Bellman equation, the solution of which is the value function associated to the optimal control problem.

Using our multithreaded C++ implementation of a fast sweeping method for Hamilton-Jacobi equations, we are able to solve real problems in a few minutes. After the value function has been calculated, globally optimal trajectories can be generated in real-time from any deployment position and time by integrating a first-order differential equation.

We extended the approach to multi-stage single-vehicle missions with logic-based constraints, reducing the problem to the solution of a sequence of partial differential equations. Each of these partial differential equations is similar to the one encountered in the single-stage problem, so we can obtain the solution to multi-stage problems using our numerical solver.

Using data from ocean flow models of the Sado estuary in Portugal, we confirmed the usability of the approach in real operational scenarios. The data is easily integrated in the framework, as are constraints arising from the geography and bathymetry of the operational region. These results show that the ocean currents can have a large impact on the optimal trajectories, and in some cases the average speed of the vehicle along the optimal trajectory is increased by more than 50 %.

We also demonstrated how the value function can itself be useful in the process of mission planning, as it gives an estimate of the optimal cost over the operational area and mission time frame As such, it can be used for checking mission feasibility from a given deployment position and time, planning the deployment position given a deployment time and vice-versa, or comparing possible deployment positions and times. Complex flow patterns and constraints are translated into quantitative visualizations such as those shown in Figures 6.5 and 6.14 and Figures 6.9 and 6.11, which are more easily interpreted by human operators.

Although the motion model used in this work does not take into account the turning rate constraints of most ocean vehicles, the numerical examples suggest that the generated trajectories are nonetheless feasible for those vehicles. We further confirmed this through a test using a simulation of extreme conditions of the ocean flow in the Tejo estuary, where the mission objectives required the vehicle's trajectory to cross the interface between two regions where the flow had radically different directions, and the results again showed that the generated trajectories are feasible for ocean vehicles. These results provide numerical evidence for the hypothesis that the cost function rather than the ocean currents is the most important factor affecting the curvature of the optimal trajectories.

Finally, we showed that the approach can be integrated easily in state-of-the-art software frameworks for autonomous vehicle operations such as the LSTS toolchain. Using a simple sampling method we obtained satisfactory tracking performance, showing that our kinematic motion model is sufficient for global trajectory optimization purposes.

7.2 Future research directions

The present work can be extended in several ways. Although here we limited ourselves to planar problems, the techniques can be directly extended to trajectory optimization in three-dimensional environments, the only difference being the additional computational burden. Since the numerical solver showed good performance in the two-dimensional case, we expect that it should be usable as-is for 3D environments. Given that the parallel implementation scales well with the number of processors, the use of grid computing facilities would make it simple to speed up computational times in that case, if needed.

Our numerical examples were limited to coastal environments. However, the approach is not tied to any particular kind of area, nor is it limited in terms of the size of the operational area or the time frame. Thus it is directly applicable to large-scale missions. The only bottleneck is the computational time. As we remarked above, the results indicate that the current solver implementation should scale to larger problems. In any case, there are several improvements which could be used to further reduce the computation time. As remarked by Detrixhe et al. [13], the hyperplane stepping method is amenable to GPU-based implementations. Recent work [12] also showed how the method can be used in conjunction with a domain-decomposition approach in a heterogeneous computing context. Another possible improvement would be the use of unstructured grids adapted to the ocean flow to concentrate the resolution in regions where the value function can be expected to be more heterogeneous. Given the literature linking optimal trajectories to Lagrangian Coherent Structures [26, 47, 71], such structures could be used to directly generate a grid from the ocean flow velocity.

In what concerns the multi-stage problem, one possible direction is the use of distributed computing to simultaneously calculate the value functions associated to each stage. Extensions to multiple vehicle operations such as coordinated rendezvous could also be considered in the framework of Alton and Mitchell [4].

The method can be integrated in the LSTS toolchain in several ways, building on the sampling approach shown here. Trajectories can be generated onboard by the vehicle, integrating with real-time obstacle avoidance algorithms. Integration with Ripples could also be considered, by deploying the solver on a server or directly on a web page (via WebAssembly compilation). Mission objectives could then be defined and updated during operations, and the corresponding value functions calculated and disseminated to the vehicle or vehicles.

Finally, we plan to evaluate and test the method in a real deployment to take place in the Sado river. This will be done in the context of the 10th edition of the Rapid Environmental Picture MUS exercise jointly organized by LSTS-Porto University, the Portuguese Navy and the Centre for Maritime Research and Experimentation. In this 10th edition the exercise will take place under the auspices of the Marine Unmanned Systems (MUS) initiative from NATO.

Bibliography

[1] M. Aguiar, J. Estrela da Silva, and J. Borges de Sousa. “Trajectory optimization for marine vehicles: models and numerical methods”. In: *SYMCOMP2019 - 4th International Conference on Numerical and Symbolic Computation*. Porto, 2019.

[2] Miguel Aguiar et al. “Trajectory Optimization for Underwater Vehicles in Time-Varying Ocean Flows”. In: *2018 IEEE/OES Autonomous Underwater Vehicle Workshop (AUV)*. IEEE, Nov. 2018. DOI: 10.1109/auv.2018.8729777.

[3] K. Alton and I. M. Mitchell. “Optimal path planning under different norms in continuous state spaces”. In: *Proceedings 2006 IEEE International Conference on Robotics and Automation, 2006. ICRA 2006.* IEEE, 2006. DOI: 10.1109/robot.2006.1641818.

[4] Ken Alton and Ian M. Mitchell. “Efficient dynamic programming for optimal multi-location robot rendezvous”. In: *2008 47th IEEE Conference on Decision and Control*. IEEE, 2008. DOI: 10.1109/cdc.2008.4738911.

[5] V. I. Arnold. *Ordinary Differential Equations*. MIT Press, 1973. 280 pp. ISBN: 0262510189.

[6] Martino Bardi and Italo Capuzzo-Dolcetta. *Optimal Control and Viscosity Solutions of Hamilton-Jacobi-Bellman Equations*. Birkhäuser Boston, 1997. DOI: 10.1007/978-0-8176-4755-1.

[7] J. G. Bellingham and K. Rajan. “Robotics in Remote and Hostile Environments”. In: *Science* 318.5853 (Nov. 2007), pp. 1098–1102. DOI: 10.1126/science.1146230.

[8] Richard Bellman. “The Theory of Dynamic Programming”. In: (1954).

[9] M. S. Branicky, V. S. Borkar, and S. K. Mitter. “A unified framework for hybrid control: model and optimal control theory”. In: *IEEE Transactions on Automatic Control* 43.1 (1998), pp. 31–45. DOI: 10.1109/9.654885.

[10] Robert W. Button, John Kamp, and Thomas B. Curtin. *A Survey of Missions for Unmanned Undersea Vehicles*. RAND PUBN. 189 pp. ISBN: 0833046888.

[11] Francis H. Clarke et al. *Nonsmooth Analysis and Control Theory*. Springer New York, 1998. DOI: 10.1007/b97650.

[12] Miles Detrixhe and Frédéric Gibou. “Hybrid massively parallel fast sweeping method for static Hamilton–Jacobi equations”. In: *Journal of Computational Physics* 322 (Oct. 2016), pp. 199–223. DOI: 10.1016/j.jcp.2016.06.023.

[13] Miles Detrixhe, Frédéric Gibou, and Chohong Min. “A parallel fast sweeping method for the Eikonal equation”. In: *Journal of Computational Physics* 237 (Mar. 2013), pp. 46–55. DOI: 10.1016/j.jcp.2012.11.042.

[14] E. W. Dijkstra. “A note on two problems in connexion with graphs”. In: *Numerische Mathematik* 1.1 (Dec. 1959), pp. 269–271. DOI: 10.1007/bf01386390.

[15] Jorge Estrela da Silva and João Borges de Sousa. “A dynamic programming approach for the motion control of autonomous vehicles”. In: *49th IEEE Conference on Decision and Control (CDC)*. IEEE, Dec. 2010. DOI: 10.1109/cdc.2010.5717937.

[16] Jorge Estrela da Silva et al. “Modeling and simulation of the LAUV autonomous underwater vehicle”. In: (Jan. 2007).

[17] M. Falcone. “A numerical approach to the infinite horizon problem of deterministic control theory”. In: *Applied Mathematics & Optimization* 15.1 (Jan. 1987), pp. 1–13. DOI: 10.1007/bf01442644.

[18] António Sérgio Ferreira et al. “The LSTS software toolchain for persistent maritime operations applied through vehicular ad-hoc networks”. In: *2017 International Conference on Unmanned Aircraft Systems (ICUAS)*. IEEE, June 2017. DOI: 10.1109/icuas.2017.7991471.

[19] António Sérgio Ferreira et al. “Advancing multi-vehicle deployments in oceanographic field experiments”. In: *Autonomous Robots* (Oct. 2018). DOI: 10.1007/s10514-018-9810-x.

[20] David L. Ferris et al. “Time-Optimal Multi-Waypoint Mission Planning in Dynamic Environments”. In: *OCEANS 2018 MTS/IEEE Charleston*. IEEE, Oct. 2018. DOI: 10.1109/oceans.2018.8604505.

[21] Wendell Fleming and Raymond Rishel. *Deterministic and Stochastic Optimal Control*. Springer New York, 1975. DOI: 10.1007/978-1-4612-6380-7.

[22] Peter Gottschling. *Discovering Modern C++*. Pearson Education (US), 2015. 480 pp. ISBN: 0134383583.

[23] G. Haller. “Lagrangian coherent structures from approximate velocity data”. In: *Physics of Fluids* 14.6 (June 2002), pp. 1851–1861. DOI: 10.1063/1.1477449.

[24] George Haller. “Lagrangian Coherent Structures”. In: *Annual Review of Fluid Mechanics* 47.1 (Jan. 2015), pp. 137–162. DOI: 10.1146/annurev-fluid-010313-141322.

[25] João P. Hespanha. “Lecture notes for Hybrid Control and Switched Systems”. In: (2005).

[26] Tamer Inanc, Shawn C. Shadden, and Jerrold E. Marsden. “Optimal trajectory generation in ocean flows”. In: *In Proc. American Control Conf. 674–679*. 2004. DOI: 10.1109/acc.2005.1470035.

[27] R. E. Kalman. “On the general theory of control systems”. In: *IFAC Proceedings Volumes* 1.1 (Aug. 1960), pp. 491–502. DOI: 10.1016/s1474-6670(17)70094-8.

[28] Chiu Yen Kao, Stanley Osher, and Jianliang Qian. “Lax–Friedrichs sweeping scheme for static Hamilton–Jacobi equations”. In: *Journal of Computational Physics* 196.1 (2004), pp. 367–391. DOI: 10.1016/j.jcp.2003.11.007.

[29] Chiu-Yen Kao, Stanley Osher, and Yen-Hsi Tsai. “Fast Sweeping Methods for Static Hamilton–Jacobi Equations”. In: *SIAM Journal on Numerical Analysis* 42.6 (Jan. 2005), pp. 2612–2632. DOI: 10.1137/s0036142902419600.

[30] Wang Sang Koon et al. “Dynamical Systems, the Three-Body Problem and Space Mission Design”. In: *Equadiff 99*. World Scientific Publishing Company, Sept. 2000, pp. 1167–1181. DOI: 10.1142/9789812792617_0222.

[31] N. N. Krasovskii and A. I. Subbotin. *Game-Theoretical Control Problems (Springer Series in Soviet Mathematics)*. Springer, 2011. ISBN: 978-1-4612-8318-8.

[32] Dov Kruger et al. “Optimal AUV path planning for extended missions in complex fast-flowing estuarine environments”. In: Apr. 2007, pp. 4265–4270. DOI: 10.1109/ROBOT.2007.364135.

[33] Steven M. LaValle. *Planning Algorithms*. CAMBRIDGE UNIVERSITY PRESS, May 11, 2006. ISBN: 0521862051.

[34] Edward Ashford Lee and Pravin Varaiya. *Structure and Interpretation of Signals and Systems*. Pearson Education, Inc, 2011. ISBN: 978-0-578-07719-2.

[35] Hong K. Lo and Mark R. McCord. “Routing through dynamic ocean currents: General heuristics and empirical results in the gulf stream region”. In: *Transportation Research Part B: Methodological* 29.2 (Apr. 1995), pp. 109–124. DOI: 10.1016/0191-2615(94)00029-y.

[36] T. Lolla et al. “Path planning in time dependent flow fields using level set methods”. In: *2012 IEEE International Conference on Robotics and Automation*. IEEE, 2012. DOI: 10.1109/icra.2012.6225364.

[37] Tapovan Lolla, Patrick J. Haley, and Pierre F. J. Lermusiaux. “Time-optimal path planning in dynamic flows using level set equations: realistic applications”. In: *Ocean Dynamics* 64.10 (Sept. 2014), pp. 1399–1417. DOI: 10.1007/s10236-014-0760-3.

[38] Tapovan Lolla et al. “Time-optimal path planning in dynamic flows using level set equations: theory and schemes”. In: *Ocean Dynamics* 64.10 (Sept. 2014), pp. 1373–1397. DOI: 10.1007/s10236-014-0757-y.

[39] Ricardo Martins et al. “IMC: A communication protocol for networked vehicles and sensors”. In: *OCEANS 2009-EUROPE*. IEEE, May 2009. DOI: 10.1109/oceanse.2009.5278245.

[40] Mark McCord and Young-Kyun Lee. “Beneficial Voyage Characteristics for Routing Through Dynamic Currents”. In: *Issues in marine, intermodal, and motor carrier transportation* (1995).

[41] Stanley Osher. "A Level Set Formulation for the Solution of the Dirichlet Problem for Hamilton–Jacobi Equations". In: *SIAM Journal on Mathematical Analysis* 24.5 (Sept. 1993), pp. 1145–1152. DOI: 10.1137/0524066.

[42] Stanley Osher and James A. Sethian. "Fronts propagating with curvature-dependent speed: Algorithms based on Hamilton-Jacobi formulations". In: *Journal of Computational Physics* 79.1 (Nov. 1988), pp. 12–49. DOI: 10.1016/0021-9991(88)90002-2.

[43] Luca Pagani and Paul J. Scott. "Curvature based sampling of curves and surfaces". In: *Computer Aided Geometric Design* 59 (Jan. 2018), pp. 32–48. DOI: 10.1016/j.cagd.2017.11.004.

[44] Clément Pêtrès et al. "Path Planning for Autonomous Underwater Vehicles". In: *IEEE Transactions on Robotics* 23.2 (2007), pp. 331–341. DOI: 10.1109/tro.2007.895057.

[45] José Pinto et al. "Implementation of a Control Architecture for Networked Vehicle Systems". In: *IFAC Proceedings Volumes* 45.5 (2012), pp. 100–105. DOI: 10.3182/20120410-3-pt-4028.00018.

[46] José Pinto et al. "The LSTS toolchain for networked vehicle systems". In: *2013 MTS/IEEE OCEANS - Bergen*. IEEE, June 2013. DOI: 10.1109/oceans-bergen.2013.6608148.

[47] A. G. Ramos et al. "Lagrangian coherent structure assisted path planning for transoceanic autonomous underwater vehicle missions". In: *Scientific Reports* 8.1 (Mar. 2018). DOI: 10.1038/s41598-018-23028-8.

[48] Blane Rhoads, Igor Mezic, and Andrew Poje. "Minimum time feedback control of autonomous underwater vehicles". In: *49th IEEE Conference on Decision and Control (CDC)*. IEEE, 2010. DOI: 10.1109/cdc.2010.5717533.

[49] Américo S. Ribeiro. "Coupled Modelling of the Tagus and Sado estuaries and their Associated Mesoscale Patterns". MA thesis. Departamento de Física, Universidade de Aveiro, 2015.

[50] Américo S. Ribeiro et al. "David and Goliath Revisited: Joint Modelling of the Tagus and Sado Estuaries". In: *Journal of Coastal Research* 75.sp1 (2016), pp. 123–127. DOI: 10.2112/si75-025.1.

[51] J. A. Sethian. "Fast Marching Methods". In: *SIAM Review* 41.2 (1999), pp. 199–235. DOI: 10.1137/s0036144598347059.

[52] James A. Sethian. "A fast marching level set method for monotonically advancing fronts". In: *Proceedings of the National Academy of Sciences* 93.4 (1996), pp. 1591–1595. ISSN: 0027-8424. DOI: 10.1073/pnas.93.4.1591.

[53] James A. Sethian and Alexander Vladimirsky. "Ordered Upwind Methods for Hybrid Control". In: *Hybrid Systems: Computation and Control*. Springer Berlin Heidelberg, 2002, pp. 393–406. DOI: 10.1007/3-540-45873-5_31.

[54] James A. Sethian and Alexander Vladimirsky. “Ordered Upwind Methods for Static Hamilton–Jacobi Equations: Theory and Algorithms”. In: *SIAM Journal on Numerical Analysis* 41.1 (2003), pp. 325–363. DOI: 10.1137/s0036142901392742.

[55] Shawn C. Shadden, Francois Lekien, and Jerrold E. Marsden. “Definition and properties of Lagrangian coherent structures from finite-time Lyapunov exponents in two-dimensional aperiodic flows”. In: *Physica D: Nonlinear Phenomena* 212.3-4 (2005), pp. 271–304. DOI: 10.1016/j.physd.2005.10.007.

[56] Eduardo D. Sontag. *Mathematical Control Theory: Deterministic Finite Dimensional Systems (Texts in Applied Mathematics) (v. 6)*. Springer, 1998. ISBN: 0-387-984895.

[57] M. Soulignac, P. Taillibert, and M. Rueher. “Time-minimal path planning in dynamic current fields”. In: *2008 IEEE International Conference on Robotics and Automation*. IEEE, 2008.

[58] Michaël Soulignac. “Feasible and Optimal Path Planning in Strong Current Fields”. In: *IEEE Transactions on Robotics* 27.1 (Feb. 2011), pp. 89–98. DOI: 10.1109/tro.2010.2085790.

[59] João Borges de Sousa et al. “A verified hierarchical control architecture for co-ordinated multi-vehicle operations”. In: *International Journal of Adaptive Control and Signal Processing* 21.2-3 (2007), pp. 159–188. DOI: 10.1002/acs.920.

[60] Magda C. Sousa et al. “Integrated High-resolution Numerical Model for the NW Iberian Peninsula Coast and Main Estuarine Systems”. In: *Journal of Coastal Research* 85 (May 2018), pp. 66–70. DOI: 10.2112/si85-014.1.

[61] Bjarne Stroustrup. *A Tour of C++*. Pearson Education (US), 2018. 256 pp. ISBN: 0134997832.

[62] Deepak N. Subramani et al. “Time-optimal path planning: Real-time sea exercises”. In: *OCEANS 2017 - Aberdeen*. IEEE, June 2017. DOI: 10.1109/oceanse.2017.8084776.

[63] A. Tinka et al. “Viability-based computation of spatially constrained minimum time trajectories for an autonomous underwater vehicle: Implementation and experiments”. In: *2009 American Control Conference*. IEEE, 2009. DOI: 10.1109/acc.2009.5160166.

[64] John N. Tsitsiklis. “Efficient algorithms for globally optimal trajectories”. In: *IEEE Transactions on Automatic Control* 40.9 (1995), pp. 1528–1538. DOI: 10.1109/9.412624.

[65] A. Vladimirsky. “Static PDEs for time-dependent control problems”. In: *Interfaces and Free Boundaries* (2006), pp. 281–300. DOI: 10.4171/ifb/144.

[66] P. Wessel and W. H. F. Smith. “Free software helps map and display data”. In: *Eos, Transactions American Geophysical Union* 72.41 (1991), pp. 441–441. DOI: 10.1029/90eo00319.

[67] Pål Wessel and Walter H. F. Smith. “A global, self-consistent, hierarchical, high-resolution shoreline database”. In: *Journal of Geophysical Research: Solid Earth* 101.B4 (1996), pp. 8741–8743. DOI: 10.1029/96jb00104.

[68] Anthony Williams. *C++ Concurrency in Action*. Manning, Feb. 1, 2019. 568 pp. ISBN: 1617294691.

[69] Jonas Witt and Matthew Dunbabin. "Go with the flow: Optimal AUV path planning in coastal environments". In: *Proceedings of the ACRA 2008* (Jan. 2008).

[70] Russell B. Wynn et al. "Autonomous Underwater Vehicles (AUVs): Their past, present and future contributions to the advancement of marine geoscience". In: *Marine Geology* 352 (June 2014), pp. 451–468. DOI: 10.1016/j.margeo.2014.03.012.

[71] Weizhong Zhang et al. "Optimal trajectory generation for a glider in time-varying 2D ocean flows B-spline model". In: *2008 IEEE International Conference on Robotics and Automation*. IEEE, 2008. DOI: 10.1109/robot.2008.4543348.

[72] Hongkai Zhao. "A fast sweeping method for Eikonal equations". In: *Mathematics of Computation* 74.250 (May 2004), pp. 603–628. DOI: 10.1090/s0025-5718-04-01678-3.

[73] Hongkai Zhao. "Parallel implementations of the fast sweeping method". In: *J. Comp. Math.* 25 (2007), pp. 421–429.