Chair of Mechanics

Master's Thesis

Investigating the possibility to solve the Hamilton-Jacobi-Bellman Equation by the Finite Element Method to enable feedback control of nonlinear dynamical systems

Wolfgang Flachberger, BSc

August 2021

MONTANUNIVERSITÄT LEOBEN
www.unileoben.ac.at

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich diese Arbeit selbständig verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt, und mich auch sonst keiner unerlaubten Hilfsmittel bedient habe.

Ich erkläre, dass ich die Richtlinien des Senats der Montanuniversität Leoben zu "Gute wissenschaftliche Praxis" gelesen, verstanden und befolgt habe.

Weiters erkläre ich, dass die elektronische und gedruckte Version der eingereichten wissenschaftlichen Abschlussarbeit formal und inhaltlich identisch sind.

Datum 17.08.2021

Unterschrift Verfasser/in
Wolfgang Flachberger

Machbarkeitsstudie zur Anwendbarkeit der Methode der Finiten Elemente auf die Hamilton-Jacobi-Bellman Gleichung für Probleme der Regelungstechnik

Kurzfassung

Obwohl die Theorie der Optimalen Steuerung bereits in den 50er Jahren entwickelt wurde, beschränkt sich ihre Anwendung in der Regelungstechnik auf lineare dynamische Systeme. Nach wie vor sind PID-Regler und LQ-Regler als Industriestandard im Einsatz. Das ist vor allem auf deren einfache Implementierung und hohe Zuverlässigkeit zurückzuführen. Es existiert aber eine Vielzahl an Problemstellungen der Robotik, Automatisierungstechnik und Mechatronik, bei welchen diese Methoden aufgrund von nichtlinearem Systemverhalten nicht funktionieren. Um die Theorie der Optimalen Steuerung für diesen allgemeinen Einsatz tauglich zu machen, muss ihre Anwendung über eine Software generalisiert und vereinfacht werden. Es existiert die Möglichkeit, Probleme der Optimalen Steuerung als mehrdimensionales Randwertproblem zu formulieren. Das Ziel dieser Arbeit war es, im praktischen Versuch herauszufinden, ob sich dieses Randwertproblem mittels FEM lösen lässt und ob die Berechnung für beliebige Problemstellungen verallgemeinert und automatisiert werden kann. Ausgangspunkt für die Berechnung war die Hamilton-Jacobi-Bellman Gleichung. In den ersten Kapiteln wurden die mathematischen Konzepte der Theorie der Optimalen Steuerung hergeleitet und erläutert. Danach wurden verschiedene Ansätze zur Lösung der Hamilton-Jacobi-Bellman Gleichung mittels FEM vorgestellt. Eine der größten Herausforderungen im Zuge der Arbeit war es, eine Finite Elemente Software zu entwickeln, welche partielle Differentialgleichungen in beliebig dimensionalen Räumen lösen kann. Indem das Problem in ein konvexes, eindeutig lösbares Variationsproblem überführt wurde, konnte ein zuverlässiges numerisches Lösungsverfahren entwickelt werden. Die Resultate belegen, dass es tatsächlich möglich ist, mithilfe der Finite Elemente Methode Feedback-Regelungen nichtlinearer dynamischer Systeme zu ermöglichen. Der Autor ist überzeugt, dass solche Methoden zukünftig bedeutende Akzente für die Weiterentwicklung von Forschungsgebieten wie Steuer- und Regelungstechnik, Robotik, Automatisierungstechnik und Künstlicher Intelligenz setzen werden.

Investigating the possibility to solve the Hamilton-Jacobi-Bellman Equation by the Finite Element Method to enable feedback control of nonlinear dynamical systems

Abstract

Even though Optimal Control Theory was developed in the 1950s its usage in feedback control systems is restricted to linear dynamical systems. In General PID and LQR controllers are still the state of the art which is due to their simple implementation and reliability. There is, however, a wide range of problems in robotics and mechatronics that exceed the ability of these practices due to nonlinear dynamical behaviour. To make Optimal Control Theory feasible in these fields, its application has to be unified and simplified within a software that can reliably solve these problems. It is possible to formulate the Optimal Control Problem in such a way that a solution to the resulting boundary value problem does not only reveal one optimal trajectory but a feedback control law or *strategy*, as it is called in Game Theory. The PDE which could provide the strategy is called Hamilton-Jacobi-Bellman Equation and it was the aim of this thesis to find out whether or not it is possible to solve it via the Finite Element Method. The first chapters were used to outline the applications of Optimal Control Theory and to derive the needed mathematical concepts. Next, various ways to prepare a problem formulation for the Finite Element Analysis were presented. One of the major challenges of this thesis was to develop a Finite Element software that can solve PDEs in a space with arbitrary dimension. Finally, a problem formulation was developed that features a convex variational form with a unique solution which promises reliable solvability by numerical methods. The results showed that it is fact possible to generate feedback control laws for nonlinear dynamical systems by the Finite Element Method. The author is convinced that the developed methods will shape the future of fields such as control engineering, robotics, automation and AI.

Contents

Chapter 1

Introduction

1.1 The Historic Development of Optimal Control

The theory of Optimal Control is a subject of applied mathematics that was developed in the 1950s and 60s to satisfy the high requirements of space travel. Before the formalisms of Optimal Control were introduced, only special cases of dynamic optimization problems could be solved by extensive mathematical effort. Optimal Control enables the generalization of variational problems by making a distinction between state and control variables. Thereby, a variety of different control problems can be solved in a standardized manner. The formulation of the correct mathematical problem for a specific task, however, often remains a challenge. The author of DIDO refers to this as *the problem of problems* [Ros09]. In the following, an example on how the process of finding the correct problem formulation may look like, is presented:

1.2 Problem Formulation

1.2.1 Example: Efficient Landing of a Rocket

A Rocket accelerates by thrusting out exhaust gasses in a certain direction. As a result the rocket's mass slowly decreases. The velocity of the exhaust gasses can be assumed to be constant for chemical propulsions and shall be named w in the following calculations. According to that the rocket equation for a constant gravitational field reads:

$$m\ddot{h} = \dot{m}w - mg \tag{1.1}$$

Here, h and m are the current height and mass of the rocket, respectively. The rocket can be controlled by acting on the thrust force $-\dot{m}w$. It might be the objective of an engineer to steer it from an initial height $h(0) = h_0$ and initial velocity $\dot{h}(0) = v_0$ to the ground and land it safely and as fuel efficient as possible. *Safely* means in this context that the velocity of the rocket has to be zero when it touches the ground. It follows that the endpoint constraints $h(t_f) = 0$ and $\dot{h}(t_f) = 0$ have to be satisfied. Note that t_f denotes the finial time of the maneuver.

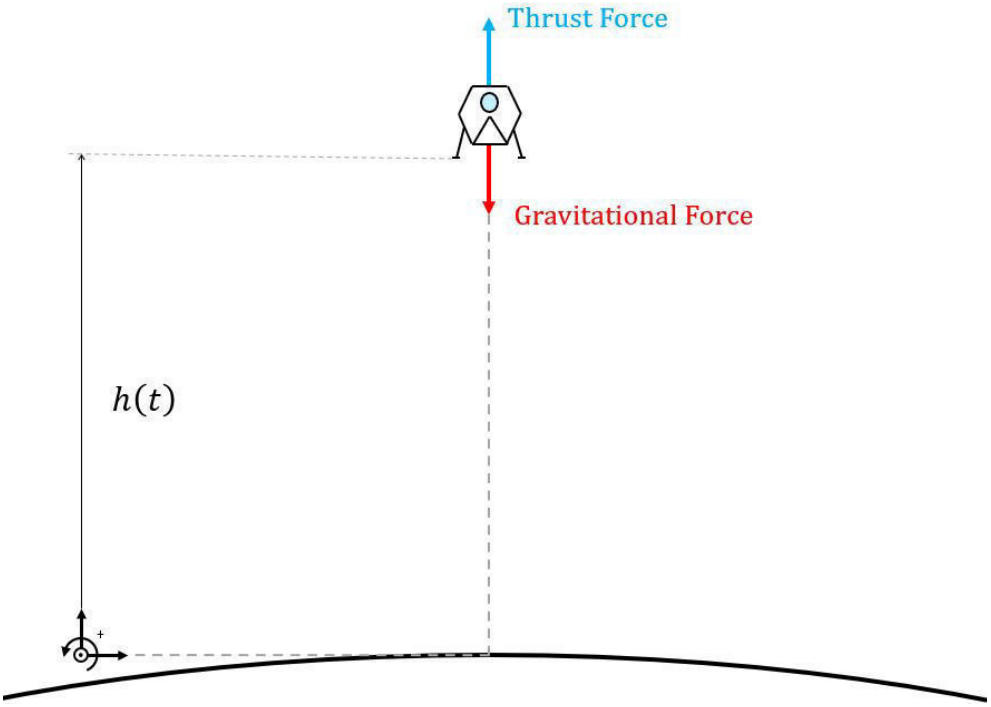

Figure 1.1: The rocket during a landing maneuver.

If the landing shall be fuel efficient we also require the rocket to lose as little mass as possible. The change in mass during the maneuver can be calculated as follows:

$$\Delta m = \int_0^{t_f} -\dot{m}(t)\ dt \tag{1.2}$$

To sum up, the following optimization problem can be formulated:

$$\min_h \Delta m \tag{1.3}$$

$$m\ddot{h} = \dot{m}w - mg \tag{1.4}$$

$$m(0) = m_0 \tag{1.5}$$

$$h(0) = h_0 \tag{1.6}$$

$$\dot{h}(0) = v_0 \tag{1.7}$$

$$h(t_f) = 0 \tag{1.8}$$

$$\dot{h}(t_f) = 0 \tag{1.9}$$

Or, in words: Find a function $h(t)$ for which Δm is minimized and for which the ordinary differential equation (1.4) as well as the boundary conditions (1.5)-(1.9) are satisfied. Note that the functions $h(t)$ and $m(t)$ are only linked by equation (1.4). This problem can be declared as variational problem with ODE-constraints. To call it an Optimal Control problem, state- and control-variables have to be introduced.

1.2.2 State and Control Space

The state vector $\underline{x}$ links the configuration of a dynamical system to a location in a Euclidean state space. State variables, the components of the state vector, change

steadily with time. Typical examples are canonical location and velocity variables. Note that the state vector of a dynamical system can only occupy a certain feasible domain of the state space - there are usually invalid regions in the state space. The control vector $\underline{u}$ assigns externally controlled quantities to a dynamical system. Control variables are not required to change continuously with time and can switch unsteadily. In mechanical systems there is often the possibility to control a force or the acceleration of certain degrees of freedom. The control vector lives in a Euclidean space that is constrained by the maximum and minimum magnitudes the control variables can obtain (e.g. in dependence of a motor that is used to control a system). Constraints of the form $a \leq u(t) \leq b$ are referred to as *box-constraints* and are common in engineering.

1.2.3 Reformulation as Optimal Control Problem

As mentioned before, the engineer can control the rocket by acting on its thrust force $-\dot{m}w$. It is therefore reasonable to arrange the problem in such a way that the thrust becomes a control variable:

$$u = -\dot{m}w \tag{1.10}$$

The control vector has in this case just one entry u. The state vector for the rocket problem becomes:

$$\underline{x} = \begin{bmatrix} h \\ v \\ m \end{bmatrix} \tag{1.11}$$

The equations of motion can now be expressed in *standard form*:

$$\dot{\underline{x}} = \underline{f}(\underline{x}, \underline{u}) \tag{1.12}$$

$$\begin{bmatrix} \dot{h} \\ \dot{v} \\ \dot{m} \end{bmatrix} = \begin{bmatrix} v \\ \frac{u}{m} - g \\ -\frac{u}{w} \end{bmatrix} \tag{1.13}$$

This formulation is common in engineering as it also makes it easy to solve the ODE numerically (as it is now a first order ODE). Another feature of the standard form is that the change of the state $\dot{\underline{x}}$ is now expressed explicitly as function of the state space (and control variables). The dynamics can now be interpreted as a vector field in the valid region of the state space, changing only with the control effort. This feature can be utilized by means of variational calculus to derive necessary conditions for optimality known as *Pontryagin's Principle*. In addition to the dynamics, the endpoint constraint function is introduced:

$$\underline{e} := \begin{bmatrix} h(t_f) \\ v(t_f) \end{bmatrix} \tag{1.14}$$

$$\underline{e} = \underline{0} \tag{1.15}$$

With this constraint, only two of the three state variables are fixed at the end of the maneuver. As the state $m(t_f)$ is free but sought to be maximal, the endpoint

constraint defines in this case not an actual point in the state space but a straight line. Every possible trajectory that is a solution to the problem ends somewhere on this line. There are of course many examples where every solution will actually end at a single point (when all state variables are specified at the end), but in general it can only be stated that the trajectories end on a *hypersurface* [Bry75]. The problem of the rocket can now be expressed in terms of the following framework for Optimal Control Problems (as suggested by I. Michael Ross, the Author of [Ros09]):

$$\min_{\underline{u}(t)\in U} J[\underline{x}(\cdot),\underline{u}(\cdot),t_f] := \int_{t_0}^{t_f} F(\underline{x}(t),\underline{u}(t))\,dt + E(\underline{x}(t_f)) \tag{1.16}$$

$$\dot{\underline{x}}(t) = \underline{f}(\underline{x}(t),\underline{u}(t)) \tag{1.17}$$

$$\underline{0} = \underline{e}(\underline{x}(t_f)) \tag{1.18}$$

$$\underline{x}(t_0) = \underline{x}_0 \tag{1.19}$$

Here F denotes the running or *Lagrangian* cost function and E is the endpoint or *Mayer* cost function [Ros09]. Note also that U denotes the control space. For the problem of the efficient landing of the rocket a cost functional of Lagrangian type is used (on the basis of equation (1.3)). Therefore, the Optimal Control problem reads:

$$\min_{u(t)} J[u(\cdot)] = \int_0^{t_f} \frac{u(t)}{w}\,dt \tag{1.20}$$

$$\begin{bmatrix}\dot{h}\\ \dot{v}\\ \dot{m}\end{bmatrix} = \begin{bmatrix} v\\ \frac{u}{m}-g\\ -\frac{u}{w}\end{bmatrix} \tag{1.21}$$

$$(0,0) = (h(t_f), v(t_f)) \tag{1.22}$$

$$(h(0), v(0), m(0)) = (h_0, v_0, m_0) \tag{1.23}$$

$$0 \le u(t) \le u_{max} \tag{1.24}$$

Chapter 2

Optimal Control and Theoretical Mechanics

2.1 Review of the principles of Mechanics

In this section the variational principles of mechanics are discussed. As shall be shown, the mathematical concepts of variational calculus used to retrieve more fundamental laws from simple Newtonian Mechanics are the very same that can be utilized to optimize dynamical processes.

2.1.1 The Principle of Virtual Work

Although the first virtual Work principles date back for as long as antiquity [Red84] modern mathematical formulations needed a proper definition of force and work and had to await the discoveries of Newton to be made. The Principle of Virtual Work states that a body is in equilibrium if the virtual work of all forces acting on the body is zero upon a virtual displacement.

$$\delta \mathscr{W} = 0 \tag{2.1}$$

2.1.2 d'Alembert's Principle

D'Alembert's Principle follows from the idea that the inertial force, although non Newtonian as it appears without a reactional force, can also perform work (and therefore also virtual work).

$$(\underline{F} - m\ddot{\underline{r}}) \cdot \delta \underline{r} = 0 \tag{2.2}$$

Here, $\underline{r}$ denotes an arbitrary position vector. Another discovery made by d'Alembert was the fact that a virtual displacement $\delta \underline{r}$ is not entirely arbitrary as it has to be consistent with geometric constraints:

$$\delta \underline{r} = \frac{\partial \underline{r}}{\partial \underline{q}} \cdot \delta \underline{q} \tag{2.3}$$

Here, $\underline{q}$ denotes the vector of the degrees of freedom of the system. Obviously, the partial derivative takes the value zero if the corresponding force performs no work. Therefore, it is enough to concentrate on active forces $\underline{F}^{(e)}$ and neglect the others:

$$(\underline{F}^{(e)} - m\underline{\ddot{r}}) \cdot \delta\underline{r} = 0 \tag{2.4}$$

2.1.3 Lagrangian Mechanics and Hamilton's Principle

Of course, because d'Alembert's Principle is satisfied at any time, the following integral has to vanish:

$$\int_{t_0}^{t_f} (m\underline{\ddot{r}} - \underline{F}^{(e)}) \cdot \delta\underline{r}\, dt = 0 \tag{2.5}$$

By performing integration by parts and substituting the condition for conservative forces $\underline{F}^{(e)} = -\nabla V$, the equation can be expressed in terms of kinetic energy T and potential energy V (see [Red84] for a detailed derivation).

$$\delta \int_{t_0}^{t_f} (T - V)\, dt = 0 \tag{2.6}$$

One might think of this as the weak formulation of Newtons Law for conservative dynamical systems. Obviously, the equation requires the variation of the integral to vanish. This condition is called *stationary* and means that the functional has to take either the value of a local minimum, maximum or inflection. This functional is called *action* and it is usually denoted by S. The integrand of the functional is termed the *Lagrangian*.

$$S[\underline{q}(\cdot)] = \int_{t_0}^{t_f} (T - V)\, dt = \int_{t_0}^{t_f} \mathscr{L}(\underline{q}(t), \underline{\dot{q}}(t))\, dt \tag{2.7}$$

The equation the Lagrangian has to satisfy in order to make the functional stationary is called *Euler-Lagrange Equation*. It can be derived by computing the first variation or by taking the total derivative of the Lagrangian, which is more common in applied mechanics:

$$\delta S = \int_{t_0}^{t_f} \left(\frac{\partial \mathscr{L}}{\partial \underline{q}} \cdot \delta\underline{q} + \frac{\partial \mathscr{L}}{\partial \underline{\dot{q}}} \cdot \delta\underline{\dot{q}} \right) dt \tag{2.8}$$

Once again, we can use integration by parts to simplify the functional:

$$\delta S = \int_{t_0}^{t_f} \left(\frac{\partial \mathscr{L}}{\partial \underline{q}} - \frac{d}{dt}\left(\frac{\partial \mathscr{L}}{\partial \underline{\dot{q}}} \right) \right) \cdot \delta\underline{q}\, dt \tag{2.9}$$

As $\delta\underline{q}$ is completely arbitrary the only way for the functional to satisfy the stationary condition is that the term in brackets is equal to zero (*Fundamental Lemma of Variational Calculus*):

$$\delta S = 0 \quad \Longleftrightarrow \quad \frac{\partial \mathscr{L}}{\partial \underline{q}} - \frac{d}{dt}\left(\frac{\partial \mathscr{L}}{\partial \underline{\dot{q}}} \right) = 0 \tag{2.10}$$

2.1.4 The Canonical Form

Just like in finite-dimensional optimization methods, constraints can be taken into account by the Lagrange Multiplier Method [HC24]:

$$S = \int_{t_0}^{t_f} \mathscr{L}(\underline{q}, \underline{\dot{q}}) + \underline{p} \cdot \left(\frac{d\underline{q}}{dt} - \underline{\dot{q}} \right) \, dt \tag{2.11}$$

Here the connection between $\underline{q}$ and $\underline{\dot{q}}$, the definition of the first derivative, is incorporated into the functional. As $d\underline{q}/dt = \underline{\dot{q}}$, the value of the action remains unchanged for every choice of the Lagrange Multiplier $\underline{p}$. The constraining equation, nevertheless, gets formally eliminated and the additional information in the functional can be utilized by means of variational calculus under the requirement that another function $\underline{p}(t)$ appears in the problem [HC24]. Integration by parts yields:

$$S = \int_{t_0}^{t_f} \mathscr{L}(\underline{q}, \underline{\dot{q}}) - \underline{\dot{p}} \cdot \underline{q} - \underline{p} \cdot \underline{\dot{q}} \, \, dt \tag{2.12}$$

Applying the Euler-Lagrange Equation with respect to $\underline{q}$, $\underline{\dot{q}}$ and $\underline{p}$ leads to:

$$\mathscr{L}_{\underline{q}} - \underline{\dot{p}} = 0 \tag{2.13}$$

$$\mathscr{L}_{\underline{\dot{q}}} - \underline{p} = 0 \tag{2.14}$$

$$\frac{d\underline{q}}{dt} - \underline{\dot{q}} = 0 \tag{2.15}$$

Note that subscripts denote partial derivatives with respect to the subscripted functions. In classical Mechanics, where $\mathscr{L} = T - V$, the Lagrange Multiplier p becomes the conservative quantity of *Momentum*. The generalized speeds can now be expressed in terms of generalized coordinates and momentum:

$$\text{with} \quad \mathscr{L}_{\underline{\dot{q}}} = \mathscr{L}_{\underline{\dot{q}}}(\underline{q}, \underline{\dot{q}}) \quad \text{and} \quad \underline{p} = \mathscr{L}_{\underline{\dot{q}}} \quad \Rightarrow \quad \underline{\dot{q}} = \underline{\dot{q}}(\underline{q}, \underline{p}) \tag{2.16}$$

By substitution of $\underline{\dot{q}} = \underline{\dot{q}}(\underline{q}, \underline{p})$ the original variational problem is transformed and projected into a *dual space* [Ros09].

$$S[\underline{q}(\cdot), \underline{p}(\cdot)] = \int_{t_0}^{t_f} \mathscr{L}(\underline{q}, \underline{\dot{q}}(\underline{q}, \underline{p})) - \underline{\dot{p}} \cdot \underline{q} - \underline{p} \cdot \underline{\dot{q}}(\underline{q}, \underline{p}) \, \, dt \tag{2.17}$$

By selecting a Hamiltonian function as follows:

$$\mathscr{H}(\underline{q}, \underline{p}) = \underline{p} \cdot \underline{\dot{q}}(\underline{q}, \underline{p}) - \mathscr{L}(\underline{q}, \underline{\dot{q}}(\underline{q}, \underline{p})) \tag{2.18}$$

The Functional can be expressed as:

$$S[\underline{q}(\cdot), \underline{p}(\cdot)] = \int_{t_0}^{t_f} -\mathscr{H}(\underline{q}, \underline{p}) - \underline{\dot{p}} \cdot \underline{q} \, \, dt \tag{2.19}$$

This is known as the canonical form of the variational problem. It is equivalent to the action functional but as $\underline{p}$ is already chosen in order to make the integral statement stationary it bears certain features that are crucial to Hamilton-Jacobi Theory (see section 2.1.6). Also the seemingly arbitrary choice of the Hamiltonian function will

be clarified in this section. Applying the Euler-Lagrange Equation with respect to all mutually independent variables, however, leads to the canonical equations:

$$\frac{\partial}{\partial \underline{p}}\left(-\mathscr{H}(\underline{q},\underline{p}) - \underline{\dot{p}}\cdot\underline{q}\right) - \frac{d}{dt}\left(\frac{\partial}{\partial \underline{\dot{p}}}\left(-\mathscr{H}(\underline{q},\underline{p}) - \underline{\dot{p}}\cdot\underline{q}\right)\right) = 0 \tag{2.20}$$

$$\frac{\partial}{\partial \underline{q}}\left(-\mathscr{H}(\underline{q},\underline{p}) - \underline{\dot{p}}\cdot\underline{q}\right) = 0 \tag{2.21}$$

Which simplifies to:

$$\mathscr{H}_{\underline{p}} - \underline{\dot{q}} = \underline{0} \tag{2.22}$$

$$\mathscr{H}_{\underline{q}} + \underline{\dot{p}} = \underline{0} \tag{2.23}$$

In this *dual-* or *phase space* (q,p) the dynamics (vector field) possesses zero divergence which leads to conservation of energy.[1]

2.1.5 Hamilton-Jacobi Theory

Hamilton-Jacobi Theory is concerned with expressing the above results in a more compact and generalized PDE-Formulation. We continue the derivation of the Hamilton-Jacobi Equation from the canonical form of the variational problem:

$$S[\underline{q}(\cdot),\underline{p}(\cdot)] = \int_{t_0}^{t_f} \underline{p}\cdot\underline{\dot{q}} - \mathscr{H}(\underline{q},\underline{p})\ \ dt \tag{2.24}$$

This expression is clearly a functional but integrals over functions can under certain conditions as well be interpreted as functions [HC37]. To achieve this we view the action as explicit function of time and canonical coordinates. By integrating from initial time to an arbitrary time t it is reasonable to think of S as a function.

$$S(\underline{q},t) = \int_{t_0}^{t} \underline{p}\cdot\underline{\dot{q}} - \mathscr{H}(\underline{q},\underline{p})\ \ dt \tag{2.25}$$

To get rid of the integral we differentiate the equation with respect to time:

$$\dot{S} = S_t + S_{\underline{q}}\cdot\underline{\dot{q}} = \underline{p}\cdot\underline{\dot{q}} - \mathscr{H}(\underline{q},\underline{p}) \tag{2.26}$$

Note that this step is only reasonable when it is computed for the canonical form of the variational problem as it already features the stationary condition in the integral. From equation (2.13) we conclude:

$$S_{\underline{q}} = \underline{p} \tag{2.27}$$

Therefore, the Hamilton-Jacobi Equation (also known as *Eikonal Equation* [HC24]) becomes:

$$S_t + \mathscr{H}(\underline{q}, S_{\underline{q}}) = 0 \tag{2.28}$$

This PDE also explains the at first seemingly arbitrary choice of the Hamiltonian function and why it is a constant for every Lagrangian that is not explicitly time dependent (which is the case in mechanics). Moreover it can be seen, that the existence of the immeasurable quantities energy and momentum such as their conservation laws naturally arises by introducing the powerful tools of variational calculus and the Lagrange multiplier method. For more informations on Hamilton-Jacobi Theory [HC24], [HC37] and [Lev14] are a good starting point.

[1] Note that in a space with axis $q[m]$ and $p[kg\cdot m/s]$ the area has the unit $[J\cdot s]$. See also *Liouville's Theorem* [Lev14].

2.1.6 The Legendre Transformation

In section 2.1.5 the Legendre Transformation was introduced without ever being mentioned as natural consequence of the Lagrange Multiplier method. By viewing it as independent mathematical concept it can be formulated as follows:

$$\mathscr{H}(\underline{q},\underline{p}) = \sup_{\dot{\underline{q}}(\underline{q},\underline{p})} \left(\underline{p}\cdot\dot{\underline{q}} - \mathscr{L}(\underline{q},\dot{\underline{q}})\right) \tag{2.29}$$

Or, in words: Choose $\dot{\underline{q}}$ in dependence of $\underline{q}$ and $\underline{p}$ such, that the term in parenthesis takes a maximum value. Of course, the Legendre Transformation is therefore an optimization problem of its own that can be treated by the regular methods of analysis (e.g.: setting the first derivative zero etc.). In section 2.1.5 this was achieved by application of the Euler-Lagrange Equation. Another Legendre transform leads back to the Lagrangian:

$$\mathscr{L}(\underline{q},\dot{\underline{q}}) = \sup_{\underline{p}(\underline{q},\dot{\underline{q}})} \left(\underline{p}\cdot\dot{\underline{q}} - \mathscr{H}(\underline{q},\underline{p})\right) \tag{2.30}$$

Note that this definition holds only for problems that require a functional to be *stationary* in terms of variational calculus (the action has to take the value of either minimum, maximum or inflection). It will, however, not always work in Optimal Control Theory as the cost functional is required explicitly to take a minimum value. Moreover, the above definition of the Legendre Transform only holds for convex functions. This is, due to the quadratic dependency of kinetic energy on velocity, always satisfied in mechanics. For general control problems this is not necessarily true. In fact, most problems that occur in engineering are linear with respect to the control variable and therefore, do not feature convexity. The next chapter is concerned with finding the Legendre Transform's equivalent for Optimal Control Problems.

2.2 The Hamilton-Jacobi-Bellman Equation

In the following we consider a *stationary* Optimal Control Problem with ODE constraints. Hereby, *stationary* refers to a control task where the cost functional as well as the dynamics are not explicitly time dependent [Bry75], which is common for mechanical systems.

$$\min_{\underline{u}(t)\in U} J[\underline{x}(\cdot),\underline{u}(\cdot),t_f] := \int_{t_0}^{t_f} F(\underline{x}(t),\underline{u}(t))\,dt + E(\underline{x}(t_f)) \tag{2.31}$$

$$\dot{\underline{x}}(t) = \underline{f}(\underline{x}(t),\underline{u}(t)) \tag{2.32}$$

$$\underline{0} = \underline{e}(\underline{x}(t_f)) \tag{2.33}$$

$$\underline{x}(t_0) = \underline{x}_0 \tag{2.34}$$

As before we can take the constraints into account by introducing the Lagrange Multiplier $\underline{p}$. As the endpoint constraint is constant it is only relevant to the boundary conditions of the problem but it can be fit into the functional by the method as well:

$$\min_{\underline{u}(t)\in U} S[\underline{x}(\cdot),\underline{u}(\cdot),\underline{p}(\cdot),\underline{\lambda},t_f] :=$$
$$\int_{t_0}^{t_f} F(\underline{x}(t),\underline{u}(t)) + \underline{p}(t)\cdot(\underline{f}(\underline{x}(t),\underline{u}(t)) - \dot{\underline{x}}(t))\,dt + E(\underline{x}(t_f)) + \underline{\lambda}\cdot\underline{e}(\underline{x}(t_f)) \tag{2.35}$$

Before proceeding we introduce the *Control Hamiltonian*:

$$H(\underline{x}, \underline{u}, \underline{p}) := F(\underline{x}, \underline{u}) + \underline{p} \cdot \underline{f}(\underline{x}, \underline{u}) \tag{2.36}$$

This step is helpful for simplifying the further derivation but it is important to note that the Control Hamiltonian is not the Hamiltonian's equivalent for control problems but just an intermediate step towards it. We now assume the control problem to be convex with respect to all dependent functions which means that we can use the Euler-Lagrange Equation to derive the necessary optimality conditions:

$$\frac{\partial H}{\partial \underline{u}} = \underline{0} \tag{2.37}$$

$$\frac{\partial H}{\partial \underline{p}} = \dot{\underline{x}} \tag{2.38}$$

$$\frac{\partial H}{\partial \underline{x}} = -\dot{\underline{p}} \tag{2.39}$$

As can be seen the obtained system is already closely related to the canonical equations of Mechanics. While equation (2.38) simply results in the given dynamical constraint ($\dot{\underline{x}} = \underline{f}(\underline{x}, \underline{u})$), equation (2.39) is called the *adjoint system* with the multiplier $\underline{p}$ as the *adjoint state*. Equation (2.37) however, is called *Hamiltonian Minimization Condition* and of course it can be used to compute a $\underline{u}$ for which H becomes stationary (in the sense of regular calculus) and, hopefully, a minimum:

$$\frac{\partial H}{\partial \underline{u}} = \underline{0} \;\; \Rightarrow \;\; \underline{u}^* := \underline{u}(\underline{x}, \underline{p}) \tag{2.40}$$

The star indicates that we are already dealing with the optimized control. As mentioned before these practices will only work if H is convex with respect to $\underline{u}$. For the general case, i.e. an arbitrary dependency between H and $\underline{u}$, the following *Hamiltonian Minimization Condition* has to be satisfied:

$$\underline{u}^*(\underline{x}, \underline{p}) = \arg\min_{\underline{u} \in U} H(\underline{x}, \underline{u}, \underline{p}) \tag{2.41}$$

This generalization is the key element of *Pontryagin's Principle*. The notation indicates that we are looking for a control $\underline{u}^*$ that globally minimizes H for a given (and time dependent) pair of variables $(\underline{x}, \underline{p})$. Having found this control, we can introduce the *Lower Control Hamiltonian*, which is the equivalent of the Hamiltonian in classical Mechanics:

$$\mathscr{H}(\underline{x}, \underline{p}) = H(\underline{x}, \underline{u}, \underline{p}) \Big|_{\underline{u} = \underline{u}^*(\underline{x}, \underline{p})} \tag{2.42}$$

According to that, if we were to define the Legendre Transform for Optimal Control Problems, it can be stated as follows:

$$\mathscr{H}(\underline{x}, \underline{p}) = \min_{\underline{u} \in U} \big(F(\underline{x}, \underline{u}) + \underline{p} \cdot \underline{f}(\underline{x}, \underline{u}) \big) \tag{2.43}$$

We can now view the Optimal Control Problem as Variational Problem in its canonical form:

$$S(\underline{x}, t) = \int_t^{t_f} \big(\mathscr{H}(\underline{x}, \underline{p}) - \underline{p} \cdot \dot{\underline{x}} \big) \; dt + E(\underline{x}_f) + \underline{\lambda} \cdot \underline{e}(\underline{x}_f) \tag{2.44}$$

Again the representation as a function has been chosen. Note that this time we integrate from an arbitrary time to a final time t_f. Many authors are also referring to this as "solving problems backwards in time"[Bry75][Isa65][Ros09]. Unlike in mechanics, where only the initial state is known for feedback control problems, the final state of the system is prescribed. Just like in the derivation of the Hamilton-Jacobi Equation differentiation with respect to time yields:

$$S_t + S_{\underline{x}} \cdot \dot{\underline{x}} = -\mathscr{H}(\underline{x}, \underline{p}) + \underline{p} \cdot \dot{\underline{x}} \tag{2.45}$$

As $\underline{p}$ is just a multiplier and the outcome of the variational problem is unaffected by its value we set $\underline{p} = S_{\underline{x}}$. In the following, "HJ" will be short for Hamilton-Jacobi Equation and "HJB" will refer to Hamilton-Jacobi-Bellman Equation. It is now obvious that the Control Hamiltonian and its Legendre Transform were also defined differently to the Hamiltonian of Mechanics to ensure that the HJ and the HJB look similar.

$$S_t + \mathscr{H}(\underline{x}, S_{\underline{x}}) = 0 \tag{2.46}$$

By taking the constant terms of the cost function (2.47) into account we get the following boundary conditions:

$$S(\underline{x}) = E(\underline{x}) \quad \{\underline{x} \in \Omega : \underline{e}(\underline{x}) = \underline{0}\} \tag{2.47}$$

For stationary Optimal Control Problems the partial time derivative of the cost function vanishes ($S_t = 0$). Many authors, especially in Game Theory, prefer for this kind of problem a more compact form of the HJB that does not mention Hamiltonians and the relation to mechanics:

$$\min_{\underline{u}} \left[F(\underline{x}, \underline{u}) + \nabla S(\underline{x}) \cdot \underline{f}(\underline{x}, \underline{u}) \right] = 0 \tag{2.48}$$

In [Isa65] this is referred to as the *Main Equation*

2.3 Everything is stationary

Even though stationary Optimal Control problems constitute a special case one can rearrange every problem, without effectively changing the outcome, in such a way that it becomes mathematically stationary. The focus on stationary problems is therefore not a restriction but a step towards finding a unified and simple approach for Optimal Control problems (especially due to $S_t = 0$ which holds only for stationary problems). To prove this point the following general Optimal Control problem is considered:

$$\min_{\underline{u}(t) \in U} J[\underline{x}(\cdot), \underline{u}(\cdot), t_f] := \int_{t_0}^{t_f} F(\underline{x}(t), \underline{u}(t), t)\, dt + E(\underline{x}(t_f), t_f) \tag{2.49}$$

$$\dot{\underline{x}}(t) = \underline{f}(\underline{x}(t), \underline{u}(t), t) \tag{2.50}$$

$$\underline{0} = \underline{e}(\underline{x}(t_f), t_f) \tag{2.51}$$

$$\underline{x}(t_0) = \underline{x}_0 \tag{2.52}$$

As can be seen, the dynamics, cost and constraints are explicitly time dependent. To eliminate this dependency a new state variable x_{n+1} is introduced to replace time in the set of ODEs:

$$\min_{\underline{u}(t)\in U} J[\underline{x}(\cdot),\underline{u}(\cdot),t_f] := \int_{t_0}^{t_f} F(\underline{x}(t),\underline{u}(t),x_{n+1})\,dt + E(\underline{x}(t_f),x_{n+1}(t_f)) \tag{2.53}$$

$$\begin{bmatrix}\dot{\underline{x}}\\ \dot{x}_{n+1}\end{bmatrix} = \begin{bmatrix}\underline{f}(\underline{x},\underline{u},x_{n+1})\\ 1\end{bmatrix} \tag{2.54}$$

$$\underline{0} = \underline{e}(\underline{x}(t_f),x_{n+1}(t_f)) \tag{2.55}$$

$$\begin{bmatrix}\underline{x}(t_0)\\ x_{n+1}(t_0)\end{bmatrix} = \begin{bmatrix}\underline{x}_0\\ t_0\end{bmatrix} \tag{2.56}$$

$$\tag{2.57}$$

As this new state variable has the simple dynamics $\dot{x}_{n+1} = 1$, it will behave just like the actual time and rise linearly. Therfore, the engineer obtains a formal return to the stationary system with the consequence that the dimension of the state space is increased by one.

Chapter 3

Motivation

3.1 Solving Optimal Control problems

DIDO[1] approaches Optimal Control Problems by using the canonical equations to reveal ODE optimality conditions. The resulting system of equations is then numerically solved via the Pseudospectral Method[2] [Ros09]. The solution to the resulting boundary value problem returns only one optimal trajectory for a certain initial condition $\underline{x}_0$. Feedback solutions using this technique require extensive computing power and became feasible during the last two decades. The realization of such a system however is still a complex task and the question arises if there could be other solutions more accessible to users of other fields than aerospace. By solving the HJB PDE one obtains not just one optimal trajectory and the associated time dependent control effort $\underline{u}(t)$ but a whole strategy. This means that the control effort can be expressed in terms of the actual state $\underline{u}(\underline{x})$. This could enable Optimal Feedback Control without ongoing calculations during the control process.

3.2 Optimal Control and Artificial Intelligence

One approach by Russell and Norvig defines artificial intelligence (AI) as *designing systems that behave optimal* [RN16]. The validity of this statement is justified as experiments show, that the behaviour of AI equipped robots can be very well reproduced by Optimal Feedback Control [Ros09]. It even seems that Optimal Control, and especially real-time Optimal Control, offer a more sophisticated way of generating AI than most *machine learning* algorithms that critically rely on data. A popular example where Optimal Control produces similar outcomes as machine learning is called *Hurni's Sliding Door Experiment* [Ros09]. Here, a simple craft has the task to move to another location as quickly as possible (the craft will take the shortest path; a straight line). Whilst the craft carries out the task, additional informations and

[1] DIDO is a software-tool which is capable of solving Optimal Control problems It is developed and distributed by *Elissar Global*. Visit *https://www.elissarglobal.com/industry/get-dido/* for more information.

[2] Pseudospectral Methods constitute a subclass of Spectral Methods. These methods are related to the Finite Element Method with the main difference that interpolation functions can obtain non-zero-values over the whole domain, which makes the use of elements and meshing practices obsolete.

constraints enter the problem: A sliding door moves in the way of the craft and it has to find its way around the obstacle. The author of [Ros09] developed a sophisticated mathematical formulation to get hold of time dependent outer factors that change whilst the control task is carried out and he calls this framework *Problem P*[3]. *Problem P* is mathematically challenging but can still be solved by DIDO. The major drawback of this solution technique is that the information of the moving obstacle has to be considered beforehand, which is a significant disadvantage compared with the performance of trained AI. Assuming that it is possible to solve the HJB Equation via the linear Finite Element Method on the other hand, reveals possibilities that exceed the capabilities of hard to train neural networks by far. By solving the HJB Equation one obtains a control law for every feasible location in the state space. If a region in the state space is not feasible any more because, for example, a door moves through its way, the strategy can be updated by excluding corresponding equations from the linear system (setting the value of the cost in the affected nodes to zero). A schedule like this obviously requires the craft to be equipped with a sufficient sensory apparatus, which is however, the same for all AI equipped devices. Another profound consequence of the ability to solve the HJB Equation regards robotics. Knowing the dynamics of robots with a high number of degrees of freedom, one could use the HJB Equation to compute the optimal control schedule for the robot to walk. This could enable engineers to no longer have to teach a robot how to walk by time expensive gradient descent methods.

3.3 Optimal Control and the Linear Quadratic Regulator

Many authors describe the Linear Quadratic Regulator as an optimal feedback controller. This is of course true but there are also restrictions for the problem formulation. The most important one is of course that the system dynamics have to be linear which is a huge drawback for general cases. But there are more restrictions regarding the running cost functions. To reveal all differences to Optimal Control the following linear system of equations is introduced:

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} x_2 \\ -x_2 \end{bmatrix} \tag{3.1}$$

These equations can be interpreted as the dynamics of a craft which is allowed to move in one spatial dimension x_1 and is, if moving, slowed down by viscous drag (velocity proportional drag). Without any control effort acting on the craft the dynamics can be visualized by the following representation in state space, where x_2 represents the velocity of the craft:

[3] In contrast to the simpler Problem B that simply results from the canonical equations.

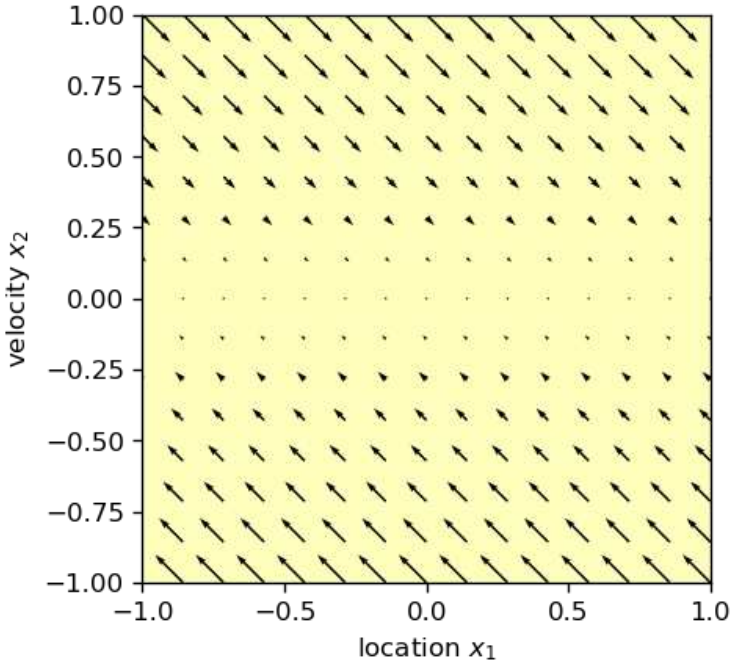

Figure 3.1: Dynamical system without control effort

Of course, the craft is slowed down and will halt on some location depending on the initial value (the initial position in state space). It could now be the objective of some sort of regulator to control the craft in such a way that it halts exactly at the location zero. Lets assume that the craft is controlled by acting directly on its acceleration. Therefore, we introduce the control variable u and modify the dynamics:

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} x_2 \\ -x_2 + u \end{bmatrix} \tag{3.2}$$

The properties of a linear system allow the dynamics to be expressed in terms of matrix multiplication. Therefore, the constant coefficient matrices $\underline{\underline{A}}$ and $\underline{\underline{B}}$ are introduced:

$$\dot{\underline{x}} = \underline{\underline{A}}\, \underline{x} + \underline{\underline{B}}\, \underline{u} \tag{3.3}$$

For the current dynamics one would obtain:

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} u \tag{3.4}$$

Without going deeper into the details of the theory of Linear Quadratic Regulators it can be said that an infinite time-horizon regulator will solve the following Optimal Control problem (for $\underline{\underline{R}}$ and $\underline{\underline{Q}}$ being square matrices of appropriate size):

$$\min_{\underline{u}} J[\underline{x}, \underline{u}] := \int_0^\infty \frac{1}{2}\, \underline{u}^\top \underline{\underline{R}}\, \underline{u} + \frac{1}{2}\, (\underline{x} - \underline{x}_f)^\top \underline{\underline{Q}}\, (\underline{x} - \underline{x}_f)\ \ dt \tag{3.5}$$

$$\dot{\underline{x}} = \underline{\underline{A}}\, \underline{x} + \underline{\underline{B}}\, \underline{u} \tag{3.6}$$

$$\underline{0} = \underline{x}(t \to \infty) - \underline{x}_f \tag{3.7}$$

Note that the matrix $\underline{\underline{R}}$ is symmetric, positive definite and that the matrix $\underline{\underline{Q}}$ has to be symmetric, positive semidefinite [Nai02]. This is to ensure convexity of the resulting variational problem. For the sake of simplicity we choose $\underline{\underline{R}}$ and $\underline{\underline{Q}}$ to be identity matrices of appropriate size, which, in this example, leads to $[1\times 1]$ and $[2\times 2]$ respectively.

$$\min_u J[\underline{x}, u] := \int_{t_0}^{\infty} u^2 + x_1^2 + x_2^2 \, dt \tag{3.8}$$

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} x_2 \\ -x_2 + u \end{bmatrix} \tag{3.9}$$

$$\underline{0} = \underline{x}(t_f) \tag{3.10}$$

Here, it is the objective to steer the state $\underline{x}$ from an arbitrary location to the location $\underline{0}$. Note that it is usual in control engineering to treat state variables equal and independently of their actual unit. By closer examination of a problem, the cost function is usually constructed with multipliers of certain units to ensure the cost functional to have a legitimate unit. The solution to the above problem can be revealed by solving the *Algebraic Riccati Equation* [Nai02]:

$$u(\underline{x}) = -x_1 - x_2 \tag{3.11}$$

With this control law, the dynamics receive an attractor at the desired location, for arbitrary initial conditions:

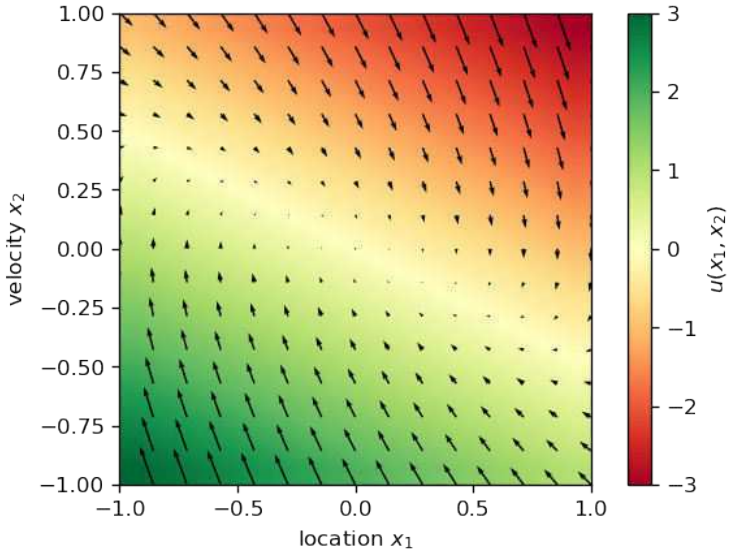

Figure 3.2: Dynamical system with LQR

The cost functional (3.2) shall now be examined more closely. The quadratic dependency of u, for example, is artificially introduced to ensure that the Euler-Lagrange Equation can be applied (a convex function is needed). It is not optimal with respect to the actual control task, however, as there is no technical or economic reason to minimize the integral of the square of the control effort[4]. Moreover the terms $x_1^2 + x_2^2$

[4]There exist certain tasks in electrical engineering where the system equations are in fact linear and the squared control effort is actually sought to be minimized. This has to do with Ohm's Law and arises for minimum energy problems: $P = UI = RI^2$ (The current I is the control variable)

are actually endpoint constraints as they have to be zero at the end (desired location in state space). It is not optimal to minimize the time integral of the squared error which is the reason Optimal Control handles endpoint constraints as a boundary condition and not as a running cost. The LQR is very elegant as it allows to express a strategy via simple matrix multiplications but it is not optimal with respect to actual engineering tasks, even when the system equations are linear (which usually is not the case). Optimal Control, on the other hand, obtains even for simple linear problems like the one above boundary value problems that require elaborate solution techniques. To implement an optimal controller the engineer also has to define the task more accurately, for example, controlling the craft in such a way that it halts exactly at the location zero and gets there in as little time as possible or with as little energy effort as possible. The minimum time problem for example can be formulated as follows:

$$\min_u J[\underline{x}, u, t_f] = \int_0^{t_f} dt \tag{3.12}$$

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} x_2 \\ -x_2 + u \end{bmatrix} \tag{3.13}$$

$$\underline{0} = \underline{x}(t_f) \tag{3.14}$$

$$-1 \leq u(t) \leq 1 \tag{3.15}$$

The solution to the minimum time problem is now presented with an image borrowed from Hale and Lasalle[5][HL63]:

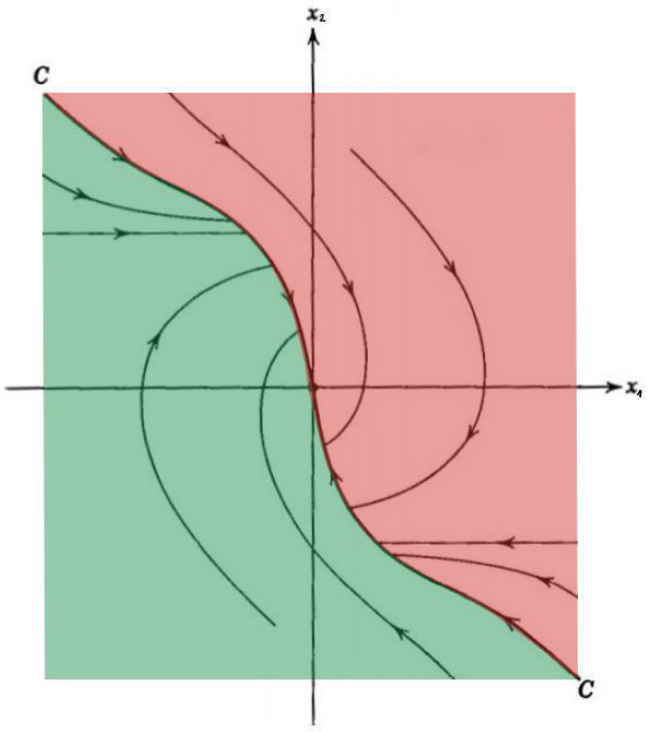

Figure 3.3: Time-optimal control with the semiuniversal curve C

[5]The graphic is actually taken from Isaacs [Isa65] who also borrowed it from Hale and Lasalle.

For the green region of the state space the optimal control effort is the maximum which is $u = 1$ whereas for the red region it is the minimum with $u = -1$. In Control Engineering a control law of this type is called a *Bang-Bang-Control*. Another remarkable feature of this *Optimal Strategy*, as it is called in game theory, is that every regular path through the state space the system can take ends on the *semiuniverssal curve* C which is also the location where the control switches its sign. The Curve C is actually the only path that reaches the endpoint as every other trajectory ends up on C.

3.4 Advantages of the Finite Element Method in Optimal Control

The Finite Element Method can not only be used to solve boundary value problems but also to build simple linear models of the original problems. These models are of course only linear and easy to solve if the original PDE exhibits a linear operator which explains the linearization efforts applied to the HJB Equation in the next chapter. If the linearization of the HJB Equation and the application of the Finite Element Method succeeds then it is also possible to express the whole stiffness matrix of a certain control task linearly dependent on a set of parameters occurring in the dynamical system. This dependence could be realized by simple matrix multiplication:

$$\underline{\underline{K}} = \underline{\underline{C}}\ \underline{\underline{K^*}} \tag{3.16}$$

This possibility would enable the even more general and powerful concept of *Adaptive Control*. During the control process an algorithm could compare the calculated system behaviour with the actual data from the sensory apparatus and update certain parameters when they change (for example friction coefficients or the mass if a robot picks something up etc.). If the FE-analysis of the HJB Equation succeeds the new information could be incorporated easily even for nonlinear dynamical systems. Another interesting feature of the FEM in control theory is that by having a payoff of endpoint or Mayer type, the actual control purpose is not defined until the boundary conditions are applied. Hence it would be possible to solve many different optimal control maneuvers for one system in little time which is of special interest for robotics.

Chapter 4

Problem Modifications for the Finite Element Analysis

4.1 Linearity of the Control Hamiltonian

4.1.1 Bang-Bang-Control

In Engineering and especially in Mechanical Engineering, the Control Hamiltonian is usually linear with respect to the Control Vector $\underline{u}$. This is mainly due to the fact that one takes control over a mechanical system by acting on a force or directly on the acceleration of certain Degrees of Freedom. These are quantities that occur as linear terms in Newtons Law of Motion and, therefore, they also occur linearly in the Control Hamiltonian. Even considering mechatronic systems, where electric current or voltage is controlled, we can assume the dynamics to be linear in $\underline{u}$ because the equations that describe the behaviour of electric motors are usually also linear in $\underline{u}$. The second reason for the linearity of the Control Hamiltonian is related to the running cost function, or, the *Lagrangian Cost Function* $F(\underline{x}, \underline{u})$. Most engineering problems are concerned with minimizing the time for a certain maneuver and/or making it as energy efficient as possible. For the minimal time problem the running cost function is simply $F := 1$ and therefore not dependent on $\underline{u}$. The efficiency problem is mainly concerned with expressing F as the momentary power input to the system (the time integral of F should give the energy input). In mechanical as well as in electrical engineering the power can be expressed as linear combination of measurable or controlled quantities. Therefore, we can assume that a large class of engineering problems possesses a Control Hamiltonian that is linear with respect to $\underline{u}$. Another feature of engineering problems is that usually every control variable has an upper and lower limit. In Control Engineering this is called *Box-Constraints*. Together with the linearity of the Control Hamiltonian the Box-Constraints result in what is called a *Bang-Bang-Control*. This is a Control Law that acts in its constrained extremum only. Every control variable of the Control Vector takes either its maximum value, minimum value or the value zero. Bang-Bang-Controls are known to act aggressively and perform better than *Linear-Quadratic-Regulators*. This is due to the fact that Bang-Bang-Controls are the result of solving the 'real' optimization problem and not a simplified one where the convexity of the Control Hamiltonian is artificially imposed by introducing a quadratic cost functional, such as for the LQR.

The linearity of the Control Hamiltonian is the reason why one can't simply use the Euler-Lagrange-Equations to formulate the necessary optimality conditions and need Pontryagin's Principle to get a grip on the problem. But as shall be shown later, the Engineering Problem will reveal one crucial advantage over quadratic problems in the numerical treatment of the HJB Equation.

4.1.2 The Rocket example and another Lagrangian

We consider the introduction example 'Efficient Landing of a Rocket' and its formulation as Optimal Control Problem from section 1.2.3 with a slight manipulation of the cost functions:

$$\min_u J[\underline{x}(\cdot), u(\cdot)] = -m(t_f) \tag{4.1}$$

$$\begin{bmatrix} \dot{h} \\ \dot{v} \\ \dot{m} \end{bmatrix} = \begin{bmatrix} v \\ \frac{u}{m} - g \\ -\frac{u}{w} \end{bmatrix} \tag{4.2}$$

$$(0,0) = (h(t_f), v(t_f)) \tag{4.3}$$

$$(h(0), v(0), m(0)) = (h_0, v_0, m_0) \tag{4.4}$$

$$0 \leq u(t) \leq u_{max} \tag{4.5}$$

As can be seen the cost functional has changed from *Lagrange* to *Mayer* type. The solution to the problem will nevertheless be equivalent as:

$$\int_0^{t_f} \frac{u(t)}{w}\, dt = \int_0^{t_f} -\dot{m}(t)\, dt = -m(t_f) + m_0 \tag{4.6}$$

As predicted, the Control Hamiltonian of the problem is linear in u:

$$H := p_h \cdot v + p_v \cdot \left(\frac{u}{m} - g\right) + p_m \cdot \left(-\frac{u}{w}\right) \tag{4.7}$$

$$H := p_h v - p_v g + \left(\frac{p_v}{m} - \frac{p_m}{w}\right) \cdot u \tag{4.8}$$

To minimize H with respect to u for $0 \leq u \leq u_{max}$ we introduce the *Lagrangian of the Hamiltonian*[1] $\overline{H}$:

$$\overline{H} := p_h v - p_v g + \left(\frac{p_v}{m} - \frac{p_m}{w}\right) \cdot u + \mu u \tag{4.9}$$

Here, μ denotes a *Lagrange Multiplier*. For linear optimization problems the value of u that minimizes H can either be 0 or u_{max} depending on the present values of parameters and state variables. To express the minimizing value u as a function of parameters and state variables we differentiate the Lagrangian of the Hamiltonian and set its value to zero:

$$\frac{\partial \overline{H}}{\partial u} = 0 \Rightarrow \mu = \frac{p_m}{w} - \frac{p_v}{m} \tag{4.10}$$

[1] In this context *Lagrangian* simply refers to a function that has to be minimized in terms of mathematical optimization methods. It is not related to the Lagrangian of Theoretical Mechanics.

The minimizing value of u is now dependent on the sign of μ. To handle this mathematically we define the Heavisside Function as:

$$\eta(x) := \begin{cases} 1 & \text{for } x > 0 \\ 0 & \text{for } x \leq 0 \end{cases} \tag{4.11}$$

The u minimizing H can now be formulated as:

$$u^* := -u_{max}\ \eta\left(\frac{p_m}{w} - \frac{p_v}{m}\right) \tag{4.12}$$

This leads to the *Lower Control Hamiltonian*:

$$\mathscr{H} := p_h v - p_v g - u_{max}\ \left(\frac{p_v}{m} - \frac{p_m}{w}\right) \cdot \eta\left(\frac{p_m}{w} - \frac{p_v}{m}\right) \tag{4.13}$$

Before formulating the Hamilton Jacobi Bellman Equation we change the name of the state variables to emphasize that we are dealing with a PDE:

$$\underline{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} h \\ v \\ m \end{bmatrix} \qquad S_{\underline{x}} = \begin{bmatrix} \partial S/\partial x_1 \\ \partial S/\partial x_2 \\ \partial S/\partial x_3 \end{bmatrix} = \begin{bmatrix} p_h \\ p_v \\ p_m \end{bmatrix} \tag{4.14}$$

With these changes the Hamilton Jacobi Bellman Equation becomes:

$$0 = \mathscr{H} \tag{4.15}$$

$$0 = \frac{\partial S}{\partial x_1} x_2 - \frac{\partial S}{\partial x_2} g - u_{max}\ \left(\frac{\partial S}{\partial x_2}\frac{1}{x_3} - \frac{\partial S}{\partial x_3}\frac{1}{w}\right) \cdot \eta\left(\frac{\partial S}{\partial x_3}\frac{1}{w} - \frac{\partial S}{\partial x_2}\frac{1}{x_3}\right) \tag{4.16}$$

It has to be solved with the following boundary conditions:

$$S(\underline{x}) = -x_3 \quad \{\underline{x} \in \Omega : x_1 = 0,\ x_2 = 0\} \tag{4.17}$$

With the solution $S(\underline{x})$ the optimal strategy can be formulated:

$$u^*(\underline{x}) := -u_{max}\ \eta\left(\frac{\partial S(\underline{x})}{\partial x_3}\frac{1}{w} - \frac{\partial S(\underline{x})}{\partial x_2}\frac{1}{x_3}\right) \tag{4.18}$$

Note that, because of the special features of the function η, the operator of the PDE (4.16) becomes *piecewise linear*. The reasoning behind this assumption follows in section 4.3.

4.1.3 A unified Solution for Engineering Problems

In the previous sections the term *Engineering Problem* was loosely defined as Optimal Control problem with a Control Hamiltonian linear with respect to $\underline{u}$. We now introduce a mathematical clarification. The Author suggests to name the following type of Optimal Control problem the *Engineering Problem*:

$$\min_{\underline{u}(t)} J[\underline{x}(\cdot)] := \int_{t_0}^{t_f} F(\underline{x}(t))\, dt + E(\underline{x}(t_f)) \tag{4.19}$$

$$\dot{\underline{x}} = \underline{f}(\underline{x}, \underline{u}) \tag{4.20}$$

$$\underline{0} = \underline{e}\,(\underline{x}(t_f)) \tag{4.21}$$

$$\underline{0} \leq \underline{u}(t) \leq \underline{a} \tag{4.22}$$

where the gradient of the dynamics $\underline{f}$ with respect to $\underline{u}$ results in a matrix $\underline{\underline{M}}(\underline{x})$ that is not dependent on $\underline{u}$:

$$\underline{f} := \underline{f}(\underline{x}, \underline{u}) \tag{4.23}$$

$$\nabla_{\underline{u}}\, \underline{f}^{\top} := \underline{\underline{M}}(\underline{x}) \tag{4.24}$$

Note that also the running cost function F has no explicit dependence on $\underline{u}$. This is necessary to simplify the further numerical treatment of the problem. An explanation for this follows in section 4.3.1. Ways to fit the standard cost functions into the *Engineering Problem* are discussed in section 4.2.1 . Another important feature of the *Engineering Problem* is that the control vector $\underline{u}$ is box constrained with each control variable u_i from zero to its maximum value a_i. Therefore, as the problem results in a *Bang-Bang-Control*, the control results in an on/off-switching scheme. Having defined the problem we proceed by forming the Control Hamiltonian and minimizing it with respect to the control vector:

$$H := F(\underline{x}) + \nabla_{\underline{x}} S(\underline{x}) \cdot \underline{f}(\underline{x}, \underline{u}) \tag{4.25}$$

$$\overline{H} := F + \underline{f}^{\top} \nabla_{\underline{x}} S + \underline{\mu}^{\top} \underline{u} \tag{4.26}$$

$$\nabla_{\underline{u}} \overline{H} := (\nabla_{\underline{u}}\, \underline{f}^{\top}) \nabla_{\underline{x}} S + \underline{\mu} = 0 \tag{4.27}$$

$$\Rightarrow \underline{u}^{*} := -\underline{a}\, \eta \left((\nabla_{\underline{u}}\, \underline{f}^{\top}) \nabla_{\underline{x}} S \right) \tag{4.28}$$

The unified Hamilton-Jacobi-Bellman Equation for the *Engineering Problem* can now be expressed compactly as:

$$F(\underline{x}) + \nabla_{\underline{x}} S(\underline{x}) \cdot \underline{f}(\underline{x}, \underline{u}^{*}) = 0 \tag{4.29}$$

$$\underline{u}^{*} := -\underline{a}\, \eta \left((\nabla_{\underline{u}}\, \underline{f}^{\top}) \nabla_{\underline{x}} S \right) \tag{4.30}$$

with the boundary condition:

$$S(\underline{x}) = E(\underline{x}) \quad \{\underline{x} \in \Omega : \underline{e}(\underline{x}) = \underline{0}\} \tag{4.31}$$

In section 2.1.1 it was shown how to transform a general Optimal Control problem into the stationary Optimal Control problem. In the next section it will be explained how and why we can fit every technically relevant Control Problem into the formalism of the *Engineering Problem*.

4.2 Problem Modifications

4.2.1 Lagrange and Mayer Cost

As the Engineering Problem requires the running cost function F to be a function of the state $\underline{x}$ only, a method to transform a cost function of Lagrange (or running) type to one of Mayer (or endpoint) type, is presented. Considering the following Optimal Control problem with $F := F(\underline{x}, \underline{u})$:

$$\min_{\underline{u}} J[\underline{x}(\cdot), \underline{u}(\cdot), t_f] := \int_{t_0}^{t_f} F(\underline{x}(t), \underline{u}(t))\, dt \tag{4.32}$$

$$\dot{\underline{x}} = \underline{f}(\underline{x}, \underline{u}) \tag{4.33}$$

$$\underline{0} = \underline{e}(\underline{x}(t_f)) \tag{4.34}$$

$$\underline{x}(t_0) = \underline{x}_0 \tag{4.35}$$

We can attach the running cost function to the dynamics by introducing a new state variable x_{n+1}:

$$\min_{\underline{u}} J[\underline{x}(\cdot), \underline{u}(\cdot), t_f] := E(\underline{x}(t_f)) = x_{n+1}(t_f) \tag{4.36}$$

$$\begin{bmatrix} \dot{\underline{x}} \\ \dot{x}_{n+1} \end{bmatrix} = \begin{bmatrix} \underline{f}(\underline{x}, \underline{u}) \\ F(\underline{x}, \underline{u}) \end{bmatrix} \tag{4.37}$$

$$\underline{0} = \underline{e}(\underline{x}(t_f)) \tag{4.38}$$

$$\begin{bmatrix} \underline{x}(t_0) \\ x_{n+1}(t_0) \end{bmatrix} = \begin{bmatrix} \underline{x}_0 \\ 0 \end{bmatrix} \tag{4.39}$$

This works because:

$$x_{n+1}(t_f) = \int \dot{x}_{n+1}\, dt + c = \int F(\underline{x}, \underline{u})\, dt + c \tag{4.40}$$

This is a method presented in [Isa65]. In many fields such as game theory a cost function of Mayer type brings certain advantages in the analysis. It also brings interesting possibilities for the FEM treatment of Optimal Control problems. By having a terminal cost function only, the optimization purpose is not defined until the boundary conditions are applied. Hence, having the stiffness matrix defined, many different control problems and maneuvers can be calculated in little time.

4.2.2 Example of a nonlinear Hamiltonian: Thrust-Vector Control

As Optimal Control Theory is applied to many aerospace engineering problems we consider a simple spacecraft that is controlled by acting directly on its thrust (u_1) and the angle (u_2) under which the thrust force acts. Furthermore, gravitational force as well as Newton-drag act on the craft:

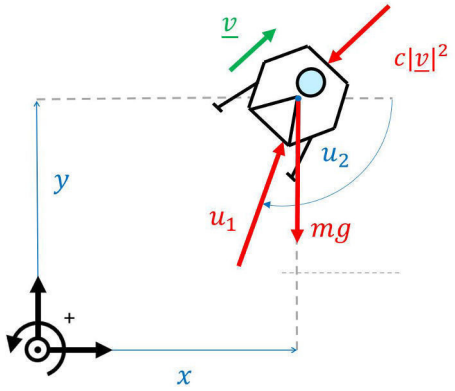

Figure 4.1: Thrust vector controlled aircraft 1

$$\begin{bmatrix} \dot{x} \\ \dot{v}_x \\ \dot{y} \\ \dot{v}_y \end{bmatrix} = \begin{bmatrix} v_x \\ -\frac{u_1}{m}\cos(u_2) - \frac{c}{m}v_x^2 \\ v_y \\ -\frac{u_1}{m}\sin(u_2) - \frac{c}{m}v_y^2 - g \end{bmatrix} \tag{4.41}$$

Note that the dynamics of the spacecraft is clearly nonlinear as the angle of the jet nozzle is a control variable. In Game Theory it's common to control such an angle directly although it is not realistic. Technically it would take some time to readjust the jet nozzle of the craft. One opportunity to tackle this problem is to control the angular velocity of the angle of attack directly. As a consequence, the angle of attack becomes a state variable:

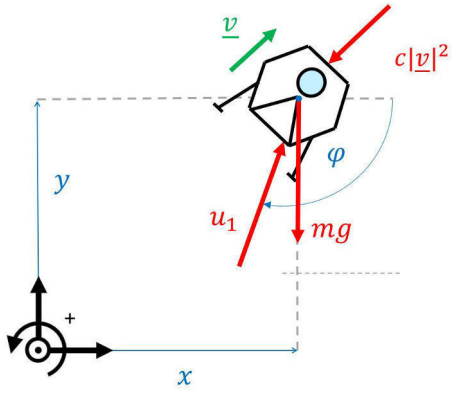

Figure 4.2: Thrust vector controlled aircraft 2

Modification of the equations of motion reveals a new system that is clearly linear with respect to $\underline{u}$:

$$\begin{bmatrix} \dot{x} \\ \dot{v}_x \\ \dot{y} \\ \dot{v}_y \\ \dot{\varphi} \end{bmatrix} = \begin{bmatrix} v_x \\ -\frac{u_1}{m}\cos(\varphi) - \frac{c}{m}v_x^2 \\ v_y \\ -\frac{u_1}{m}\sin(\varphi) - \frac{c}{m}v_y^2 - g \\ k(u_2 - u_3) \end{bmatrix} \tag{4.42}$$

The author believes that all technical control problems can be boiled down to simple on/off switching processes of various actuators. As a result the HJB Equation becomes a piecewise linear PDE and therefore manageable by linear Finite Element Methods. Hence it is reasonable to put effort in problem modification practices as described in this chapter.

4.3 Piecewise Linearity of the Hamilton-Jacobi-Bellman Equation

4.3.1 Weighted Residual Methods for the HJB

Consider the following Engineering Problem:

$$\min_{\underline{u}} J[\underline{x}(\cdot), \underline{u}(\cdot)] = \int_0^{t_f} dt \tag{4.43}$$

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} x_2 \\ u_1 - u_2 \end{bmatrix} \tag{4.44}$$

$$\underline{0} = \underline{x}(t_f) \tag{4.45}$$

$$0 \leq u_1(t) \leq 1 \tag{4.46}$$

$$0 \leq u_2(t) \leq 1 \tag{4.47}$$

This example can be interpreted as the problem of steering an electric car from an arbitrary location and velocity to standstill at location zero as quickly as possible. Applying the unified HJB for Engineering Problems reveals:

$$\underline{u}^* := -\eta\left((\nabla_{\underline{u}}\ \underline{f}^\top)\nabla_{\underline{x}} S\right) \tag{4.48}$$

$$\underline{u}^* := -\eta\left(\begin{bmatrix}0 & 1\\ 0 & -1\end{bmatrix}\begin{bmatrix}S_{x_1}\\ S_{x_2}\end{bmatrix}\right) \tag{4.49}$$

$$\underline{u}^* := \begin{bmatrix}-\eta(S_{x_2})\\ -\eta(-S_{x_2})\end{bmatrix} \tag{4.50}$$

By taking a closer look at the HJB Equation of the problem it can be seen that the Heaviside Functions ($\eta(\cdot)$) can be condensed to the modulus function ($|\cdot|$):

$$F + \nabla_{\underline{x}} S \cdot \underline{f}^* = 0 \tag{4.51}$$

$$1 + S_{x_1} x_2 + S_{x_2}(u_1^* - u_2^*) = 0 \tag{4.52}$$

$$1 + S_{x_1} x_2 + S_{x_2}\left(-\eta(S_{x_2}) + \eta(-S_{x_2})\right) = 0 \tag{4.53}$$

$$1 + S_{x_1} x_2 - |S_{x_2}| = 0 \tag{4.54}$$

$$|S_{x_2}| - S_{x_1} x_2 = 1 \tag{4.55}$$

The Function $|\cdot|$ is a piecewise linear and continuous function. To make a statement about its integrability the *sign function* is introduced:

$$\operatorname{sgn}(x) := \begin{cases} 1 & \text{for } x > 0\\ 0 & \text{for } x = 0\\ -1 & \text{for } x < 0\end{cases} \tag{4.56}$$

The application of the Finite Element Method requires integrability which is satisfied by the magnitude function as well as the Heaviside Function:

$$\int |x|\ dx = \frac{1}{2}\, x^2 \operatorname{sgn}(x) + c \quad \text{for } x \in \mathbb{R} \tag{4.57}$$

$$\int \eta(x)\ dx = \frac{1}{2}\,(|x| + x) + c \quad \text{for } x \in \mathbb{R} \tag{4.58}$$

Regarding differentiability and linearity of the functions, the following statements can be made:

$$\frac{d}{dx}|x| = const. \quad \text{for } x \in \mathbb{R} \setminus \{0\} \tag{4.59}$$

$$\Rightarrow \textit{linearity} \text{ of } |x| \text{ for } x \in \mathbb{R} \setminus \{0\} \tag{4.60}$$

$$\frac{d}{dx}\eta(x) = 0 \quad \text{for } x \in \mathbb{R} \setminus \{0\} \tag{4.61}$$

$$\Rightarrow \eta(x) = const. \text{ for } x \in \mathbb{R} \setminus \{0\} \tag{4.62}$$

Obviously, $|\cdot|$ is linear for every input but zero and $\eta(\cdot)$ is linear and constant for every input but zero. Due to this and the way these functions appear in the HJB equation for Engineering Problems, the author concludes that these PDEs are suitable for linear finite element methods and suggests to call their operators *piecewise linear*. Therefore,

the HJB Equation for Engineering Problems takes the form of a piecewise linear PDE of first order that allows the application of a linear weighted residuals FE-Method:

$$k_{ij}^{(e)} = \int_{\Omega^{(e)}} \left(\left| \frac{\partial h_i}{\partial \xi_1} \right| - \frac{\partial h_i}{\partial \xi_0} x_1 \right) \, w(h_j) \, d\Omega^{(e)} \tag{4.63}$$

$$f_j^{(e)} = \int_{\Omega^{(e)}} w(h_j) \, d\Omega^{(e)} \tag{4.64}$$

With $h_i := h_i(\underline{x})$ being interpolation functions, for example the following linear basis functions for a four-node quadrilateral element:

$$h_0(\underline{\xi}) = (1 + \xi_0)(1 + \xi_1)/4 \tag{4.65}$$

$$h_1(\underline{\xi}) = (1 + \xi_0)(1 - \xi_1)/4 \tag{4.66}$$

$$h_2(\underline{\xi}) = (1 - \xi_0)(1 + \xi_1)/4 \tag{4.67}$$

$$h_3(\underline{\xi}) = (1 - \xi_0)(1 - \xi_1)/4 \tag{4.68}$$

The assumption that the linear treatment of the nonlinear operator $|\cdot|$ will lead to valid results is firstly based on the fact that it is at least piecewise linear and secondly that a solution of the HJB is strictly positive ($0 \leq S(\underline{x}) \;\; \forall \underline{x} \in \Omega$). As $S(\underline{x})$ represents the cost it takes to move from an arbitrary state $\underline{x}$ to a desired state, it follows that it cannot take less than zero effort to reach the specified state. Therefore the following statement is always satisfied:

$$\left| s_i \frac{\partial h_i}{\partial \xi_k} \right| = s_i \left| \frac{\partial h_i}{\partial \xi_k} \right| \tag{4.69}$$

It seems as if the operator allows the application of the linear Finite Element Method but by examining the problem closer it can be seen that the algorithm would not be carried out free of errors as:

$$\left| \sum_{i=0}^{n} s_i \frac{\partial h_i}{\partial \xi_k} \right| \neq \sum_{i=0}^{n} s_i \left| \frac{\partial h_i}{\partial \xi_k} \right| \tag{4.70}$$

This is the main reason why the finite element treatment of these problems fails, as will be shown by various experiments in the next Chapter. Unfortunately, even the theoretically correct nonlinear FE-application fails as the iterative adaptation of the stiffness matrix does not converge for the piecewise linear PDE[2]. It seems like the magnitude operator is generally unsuitable to be treated by the Finite Element Method. Moreover does the whole family of Weighted Residual Methods even for simple test problems not show enough reliability. Every solution would have to be examined by an engineer, just like by using DIDO, which makes WR-Methods unattractive for the development of a straightforward Optimal Control software.

4.4 A Ritz Approach for the HJB Equation

Although weighted residual methods are convenient to use they often lack reliability when it comes to mesh convergence. Ritz's Method on the other hand obtains reliable

[2]The calculation was carried out with the open source software "FEniCS" [Aln+15] [LMW+12] [LM19] trying both, the Galerkin and the Least Squares Method (both unsuccessful)

solutions even when the element order is low and the elements are relatively big. The only drawback of the method is that the operator of the PDE is required to be symmetric and strictly positive. To achieve this the strictness on optimality of the previous sections is discarded and the following quadratic cost functional is introduced:

$$S[\underline{x}, \underline{u}, \underline{p}] := \int_{t_0}^{t_f} \frac{\alpha}{2}\underline{u}^2 + F(\underline{x}, \underline{u}) + \underline{p} \cdot (\underline{f}(\underline{x}, \underline{u}) - \dot{\underline{x}})\, dt \tag{4.71}$$

Here, F and $\underline{f}$ are both linear with respect to $\underline{u}$. Note that later in the analysis one could choose $\alpha \to 0$ to meet the aim of the original optimization purpose. The functional is closely related to the functional used to derive a feedback control law known as *Linear Quadratic Regulator* which is, together with PID-Regulators[3], still the state of the art for common industrial applications [Ros09]. It is also informative to mention that for this functional the (ODE) optimality conditions are simply derived by applying the Euler-Lagrange Equation with respect to $\underline{x}, \underline{u}$ and $\underline{p}$. The Control Hamiltonian of the problem is the following:

$$H(\underline{x}, \underline{u}, \underline{p}) := \frac{\alpha}{2}\underline{u}^2 + F(\underline{x}, \underline{u}) + \underline{p} \cdot \underline{f}(\underline{x}, \underline{u}) \tag{4.72}$$

Obviously, as the functional is a polynomial of degree two in $\underline{u}$ the *Hamiltonian Minimization Condition (HMC)*, is easy to satisfy:

$$\textit{HMC: } \nabla_{\underline{u}} H = \underline{0} \implies \underline{u}^*(\underline{x}, \underline{p}) \tag{4.73}$$

$$\underline{u}^*(\underline{x}, \underline{p}) = -\frac{1}{\alpha}\nabla_{\underline{u}}(F + \underline{f} \cdot \underline{p}) \tag{4.74}$$

This leads to the following nonlinear Hamilton-Jacobi-Bellman Equation:

$$\frac{\alpha}{2}\underline{u}^{*2} + F(\underline{x}, \underline{u}^*) + \underline{p} \cdot \underline{f}(\underline{x}, \underline{u}^*) = 0 \tag{4.75}$$

A nonlinear problem formulation is not desirable as it doesn't feature the advantages of linear algebraic systems mentioned throughout the last chapters. It is nevertheless reasonable to consider what happens if $\underline{p}$ is approached by a finite element discretization as follows:

$$\underline{p} = S_{\underline{x}} = \nabla S \approx \nabla \left(\sum_{i=0}^{n} s_i h_i(\underline{x}) \right) = \hat{\underline{p}} \tag{4.76}$$

Obviosly an approximation of this form will cause a residual and it is therfore natural to formulate another optimization problem. This time the target is to choose the design variables s_i of the approximation in such a way that the residual is minimized. Here the residual is denoted by J:

$$J(\underline{s}) := \int_{\Omega} \frac{\alpha}{2}\underline{u}^*(\underline{x}, \hat{\underline{p}})^2 + F(\underline{x}, \underline{u}^*(\underline{x}, \hat{\underline{p}})) + \hat{\underline{p}} \cdot \underline{f}(\underline{x}, \underline{u}^*(\underline{x}, \hat{\underline{p}}))\, d\Omega \tag{4.77}$$

This leads to a finite dimensional optimization problem, which, as the cost function is a polynomial of degree two, can be solved simply by taking partial derivatives.

[3]PID-Regulators are the simplest form of feedback control system. *PID* refers to the control law's dependency on the state using a *proportional, integral* and *differential* dependency of the former. See [Nai02] for more information.

$$\delta J(\underline{s}) := \frac{\partial J}{\partial \underline{s}} \cdot \delta \underline{s} = 0 \implies \frac{\partial J}{\partial s_i} = 0 \quad \text{for i = 0,1,2,..,n} \tag{4.78}$$

To be precise, it is actually a Ritz Method which minimizes not an energy functional but a residual statement (not a *weighted* residual statement, as the actual error and not the weighted error is minimized). This is, to the authors knowledge, a new approach. As the degree of the polynomial is two, the method will result in a simple linear system. The partial derivatives only have to be computed for one element to obtain a reproducible element stiffness matrix and element load vector. The scheme to compute the components of the ESM and ELV is presented for a 2 dimensional 4-node quadrilateral element:

$$J(\underline{s}) := J(s_0, s_1, s_2, s_3) \tag{4.79}$$

$$\tag{4.80}$$

$$f_0 := -\frac{\partial J}{\partial s_0}(0,0,0,0) \tag{4.81}$$

$$k_{00} := \frac{\partial J}{\partial s_0}(1,0,0,0) + f_0 \tag{4.82}$$

$$k_{01} := \frac{\partial J}{\partial s_0}(0,1,0,0) + f_0 \tag{4.83}$$

$$k_{02} := \frac{\partial J}{\partial s_0}(0,0,1,0) + f_0 \tag{4.84}$$

$$k_{03} := \frac{\partial J}{\partial s_0}(0,0,0,1) + f_0 \tag{4.85}$$

$$f_1 := -\frac{\partial J}{\partial s_1}(0,0,0,0) \tag{4.86}$$

$$k_{10} := \frac{\partial J}{\partial s_1}(1,0,0,0) + f_1 \tag{4.87}$$

$$k_{11} := \frac{\partial J}{\partial s_1}(0,1,0,0) + f_1 \tag{4.88}$$

$$k_{12} := \frac{\partial J}{\partial s_1}(0,0,1,0) + f_1 \tag{4.89}$$

$$k_{13} := \frac{\partial J}{\partial s_1}(0,0,0,1) + f_1 \tag{4.90}$$

Chapter 5

Coding a Finite Element Solver in n Dimensions

Before the FE-Approaches of the last chapters can be tested, a general framework for the Finite Element treatment of Control Problems has to be developed. The most profound difference to the usual application of FE-Methods, is that for dynamic optimization purposes the PDEs have to be solved in a state space and not in the usual three-dimensional space. For mechanical systems the state space usually has twice the number of dimensions as the number of degrees of freedom that the system shows. As for the most common type of Finite Element, the n-simplex, the meshing complexity rises drastically with the dimension there will have to be restrictions concerning element-type and degree of the used elements. An advantage regarding simplicity over the problems of continuum mechanics however, is that the mesh generation and application of boundary conditions will be easier as the state space is usually just an n-dimensional box.

5.1 The Poisson Equation

It seems to have become a tradition to introduce a new Finite Element Software by applying it to the Poisson Equation. The scalar function $u(\underline{x})$ is required to satisfy the following PDE for a given function $q(\underline{x})$:

$$\nabla^2 u(\underline{x}) + q(\underline{x}) = 0$$

The Poisson Equation can be derived by minimization of the following functional:

$$J[u] := \int_\Omega \frac{1}{2}|\nabla u|^2 - uq \; d\Omega$$

The necessary condition for a minimum of the functional requires the directional derivative of J with respect to u in the direction of the test function ν to vanish:

$$\delta J[u;\nu] := \frac{\partial}{\partial \epsilon} \int_\Omega \frac{1}{2}(\nabla u + \epsilon \nabla \nu)^2 - (u + \epsilon\nu)q \; d\Omega \Big|_{\epsilon=0} = 0$$

This leads to the *variational* or *weak* form of the Poisson Equation:

$$\int_\Omega \nabla u \cdot \nabla \nu \, dA = \int_\Omega q \, \nu \, dA$$

The Ritz Method makes use of the weak form by directly computing the element stiffness matrix and element load vector from it:

$$\sum_{i=1}^{e} u_i \int_{\Omega^{(e)}} \nabla h_i \cdot \nabla h_j \, dx = \int_{\Omega^{(e)}} h_j \, q \, dx$$

$$\underline{\underline{k}}^{(e)} \underline{u}^{(e)} = \underline{f}^{(e)}$$

$$\underline{\underline{k}}^{(e)} = \int_{\Omega^{(e)}} \nabla h_i \cdot \nabla h_j \, dx$$

$$\underline{f}^{(e)} = \int_{\Omega^{(e)}} h_j \, q \, dx$$

In the following section the Poisson equation will be solved in a two dimensional domain by a Finite Element Method. Note that this method would also work for an arbitrary number of dimensions with computing power being the only limitation.

5.2 Dissection of the Code

5.2.1 Treating multiple dimensions

Regarding the arbitrary number of dimensions the software has to cope with the most important feature to make use of is function `numpy.ndindex()`. It returns all indices of an array of arbitrary shape:

[1]:
```
import numpy

array_shape = (2,2,2) # shape of a 3-dimensional matrix of length 2 in each␣
↪dimension

for indices in numpy.ndindex(array_shape):
    print(indices)
```

```
(0, 0, 0)
(0, 0, 1)
(0, 1, 0)
(0, 1, 1)
(1, 0, 0)
(1, 0, 1)
(1, 1, 0)
(1, 1, 1)
```

It can be used to iterate quickly over every element in an n dimensional array.

5.2.2 Initialisation of the mesh

As mentioned in Chapter 4, n-dimensional *cube elements* shall be used. One might also call them n-dimensional *square elements* - the idea of extending the concept to n dimensions stays the same and is easily accomplished, which is also the reason the author chose this type of element. Trying to use the n dimensional equivalent of the *constant strain triangle*, which means filling an n-dimensional space with n-simplex elements, is mathematically challenging and requires a profound knowledge in mesh generation. The mesh object will be initialized by just the two inputs `domain` and `resolution`. The state space in optimal control is usually a simple n-dimensional box which makes the mesh generation really simple. The input `domain` is a list of n tuples with two entries each. The entries of the first tuple mark where the domain begins and ends relative to the first spacial axis, respectively.

[2]:
```
# example 1: unit cube

domain = [(0,1),(0,1),(0,1)]

# example 2: rectangle (it will be used for further calculations!)

domain = [(0,2),(0,1)]
```

The second input, `resolution`, refers to the side length of the n-cube elements the domain is filled with. To keep things simple all n-cubes are of the same size and the domain is - if the n-cubes of side length `resolution` do not fit in perfectly - just roughly approached by the elements. This is achieved by the following code:

[3]:
```
resolution = 0.2

dimension = len(domain) # the length of the domain input corresponds to the␣
 ↪number of axis of the domain
nr_el_dim = [] # initialize list with number of elements necessary per axis
nr_nd_dim = [] # initialize list with number of nodes necessary per axis
for index in range(dimension): # for every dimension
    length = domain[index][1] - domain[index][0] # take length of "box"
    nr_el_dim.append(int(length/resolution)) # approximate the domain
    nr_nd_dim.append(nr_el_dim[index] + 1)
    pass
nr_el_dim = tuple(nr_el_dim)
nr_nd_dim = tuple(nr_nd_dim)
number_elements = numpy.prod(nr_el_dim) # total amount of elements
DoF = numpy.prod(nr_nd_dim) # total amount of gloabal nodes (= DoFs)

print("Number of stacked elements per axis =",nr_el_dim)
print("Total number of elements used =",number_elements)
print("Total number of global nodes", DoF)
```

```
Number of stacked elements per axis = (10, 5)
Total number of elements used = 50
Total number of global nodes 66
```

We now know how many elements and nodes there are and how many elements we need to stack upon each other in each axis direction. The next step is to assign a number to every degree of freedom (node) and arrange the numbers in the same

way the corresponding nodes are arranged in the mesh. It is then easy to find the name (number) of a node as the elements of the mesh and the numbering array have corresponding indices.

[4]:
```
# initialize a list-array: 0, 1, 2,..
nodes = numpy.arange(DoF,dtype=int)

# as it is not important how we call each node we can simply reshape "nodes"
node_names_global = numpy.reshape(nodes,(nr_nd_dim),order='C')

print("FE-Mesh:")
print("--------")
print(node_names_global)
```

```
FE-Mesh:
--------
[[ 0  1  2  3  4  5]
 [ 6  7  8  9 10 11]
 [12 13 14 15 16 17]
 [18 19 20 21 22 23]
 [24 25 26 27 28 29]
 [30 31 32 33 34 35]
 [36 37 38 39 40 41]
 [42 43 44 45 46 47]
 [48 49 50 51 52 53]
 [54 55 56 57 58 59]
 [60 61 62 63 64 65]]
```

A general property of linear n-cubes is the relation between the dimension and the number of nodes:

[5]:
```
nodes_per_element = 2**dimension

print("Nodes per Element =",nodes_per_element)
```

```
Nodes per Element = 4
```

There is now enough information available to begin with the computation of the coincidence table.

5.2.3 The coincidence table

The task is to achieve a reliable coincidence table for box-shaped cube-meshes of arbitrary dimension.

[6]:
```
def coincidenceTable(number_elements, nodes_per_element, dimension, nr_el_dim,␣
↪node_names_global):
        """ coincidence_table[element,node] """

        # initialize coincidence table array
        coincidence_table = numpy.zeros((number_elements,nodes_per_element),␣
↪dtype=int, order='C')

        i = 0 # set row count to zero
        for element_indices in numpy.ndindex(nr_el_dim): # for every element
            j = 0 # set local node count to zero
            for node_indices in numpy.ndindex((2,)*dimension): # for every local␣
↪node in elmenet
                location = tuple([sum(x) for x in␣
↪zip(element_indices,node_indices)])# find global location of node
                coincidence_table[i,j] = node_names_global[location] # put node␣
↪name to table
                j += 1 # next element in row
                pass
            i += 1 # next row
            pass
        return coincidence_table

T = coincidenceTable(number_elements, nodes_per_element, dimension, nr_el_dim,␣
↪node_names_global)

print("Coincidence Table:")
print("------------------")
print(T)
```

```
Coincidence Table:
------------------
[[ 0  1  6  7]
 [ 1  2  7  8]
 [ 2  3  8  9]
 [ 3  4  9 10]
 [ 4  5 10 11]
 [ 6  7 12 13]
 [ 7  8 13 14]
 [ 8  9 14 15]
 [ 9 10 15 16]
 [10 11 16 17]
 [12 13 18 19]
 [13 14 19 20]
 [14 15 20 21]
 [15 16 21 22]
 [16 17 22 23]
 [18 19 24 25]
 [19 20 25 26]
 [20 21 26 27]
 [21 22 27 28]
 [22 23 28 29]
```

```
[24 25 30 31]
[25 26 31 32]
[26 27 32 33]
[27 28 33 34]
[28 29 34 35]
[30 31 36 37]
[31 32 37 38]
[32 33 38 39]
[33 34 39 40]
[34 35 40 41]
[36 37 42 43]
[37 38 43 44]
[38 39 44 45]
[39 40 45 46]
[40 41 46 47]
[42 43 48 49]
[43 44 49 50]
[44 45 50 51]
[45 46 51 52]
[46 47 52 53]
[48 49 54 55]
[49 50 55 56]
[50 51 56 57]
[51 52 57 58]
[52 53 58 59]
[54 55 60 61]
[55 56 61 62]
[56 57 62 63]
[57 58 63 64]
[58 59 64 65]]
```

We begin with the dissection of the function. The for-loops indicate that we are going through every element in the mesh and through every local node of each element. For every local node we define a location:

[7]:
```
# Example
element_indices = (0,1)
node_indices = (1,1)

location = tuple([sum(x) for x in zip(element_indices,node_indices)])

print("zip(element_indices,node_indices)␣
 ↪=",tuple(zip(element_indices,node_indices)))
print("location =",location)
```

```
zip(element_indices,node_indices) = ((0, 1), (1, 1))
location = (1, 2)
```

`location` is basically the actual location of the node in the meshgrid. The expression `sum(x) for x in zip(element_indices,node_indices)` can be understood as vector addition for tuple objects. By putting `location` into `node_names_global` the name of the node is returned. This vector becomes the row of the coincidence table that corresponds to the element it was evaluated for. Because of the special features of n-cubes the algorithm works for every given dimension and is limited just by software factors like computation-power, storage and recursion-depth.

5.2.4 The sympy Toolbox

In following sections the `sympy` toolbox is introduced to perform analytical calculations for problem formulation but also FE-related purposes. The advantage is clearly the clean, non-numeric output of sympy but is obvious that there will occur restrictions with rising number of DoFs and dimensions. For the purpose of this thesis, where the aim is to test whether or not the Finite Element Method is even suitable for this type of problems, the sympy-calculations will be sufficient. For extensive real-world use of the software the functions that use sympy might be replaced with numeric functions. Formulating the HJB without sympy in an automated manner could be realized by first converting the analytic system dynamics to a polynomial of certain degree and then splitting it up into a coefficient matrix and variable (Vandermonde) vector. It is then natural to also make use of the special features of the Finite Element Method to realize adaptive control concepts. But these practices exceed the scope of this thesis.

5.2.5 Interpolation-Functions

As we are using four node quadrilateral elements for the Poisson Problem the interpolation functions obtain the following form:

$$
\begin{aligned}
h_0(\underline{\xi}) &= (1+\xi_0)(1+\xi_1)/4 \\
h_1(\underline{\xi}) &= (1+\xi_0)(1-\xi_1)/4 \\
h_2(\underline{\xi}) &= (1-\xi_0)(1+\xi_1)/4 \\
h_3(\underline{\xi}) &= (1-\xi_0)(1-\xi_1)/4
\end{aligned}
$$

It has to be acknowledged that by building polynomials through multiplying terms like $(x+1)$ it is very easy to place the roots. This fact can be used to ensure that the function obtains zero values in nodes it doesn't belong to. For cube elements of dimension d and number of node per element N the computation of the interpolation can be expressed as:

$$
h_k = \frac{1}{N}\prod_{i=0}^{d-1}(1 \pm \xi_i)
$$

where the sign used is dependent on the specific interpolation function, but it is clear that every configuration has to be carried out once like in the two-dimensional

case. The following code computes interpolation functions for elements of arbitrary dimension by using the above considerations.

[8]:
```
import sympy

XI = numpy.asarray(sympy.symbols('xi:' + str(dimension))) # initialize vector␣
↪with local coordinates

def interpolationFunctions(XI, nodes_per_element, dimension, resolution):
        """ interpolationfunction[localNode] """
        h = sympy.ones(nodes_per_element,1)
        i = 0
        for localNodeIndices in numpy.ndindex((2,)*dimension):
            for axis in range(dimension):
                if localNodeIndices[axis] == 0:
                    factor = -1
                else:
                    factor = 1
                h[i] = h[i] * ( 1 + factor * XI[axis] * 2/resolution )
            i += 1
        h = h/nodes_per_element
        return h

h = interpolationFunctions(XI, nodes_per_element, dimension, resolution)

from IPython.display import display
print("local coordinates:")
print("------------------")
for i in range(len(XI)):
    display(XI[i])
print("interpolation functions:")
print("------------------------")
display(h)
```

```
local coordinates:
------------------
```

ξ_0

ξ_1

```
interpolation functions:
------------------------
```

$$\begin{bmatrix} \frac{(1-10.0\xi_0)(1-10.0\xi_1)}{4} \\ \frac{(1-10.0\xi_0)(10.0\xi_1+1)}{4} \\ \frac{(1-10.0\xi_1)(10.0\xi_0+1)}{4} \\ \frac{(10.0\xi_0+1)(10.0\xi_1+1)}{4} \end{bmatrix}$$

For Finite Element Problems where the variational formulation is a first order integral PDE (just like the Poisson Equation or the Hamilton-Jacobi-Bellman Equation) the first derivatives of the interpolation functions have to be computed:

[9]:
```
def interpolationFunctionsDerivatives(XI, h, nodes_per_element, dimension):
        """ interpolationfunction[node, derivativeaxis] """
        dh_dX = sympy.ones(nodes_per_element, dimension)
        for index in range(nodes_per_element):
            for axis in range(dimension):
                dh_dX[index,axis] = sympy.diff(h[index], XI[axis])
        return dh_dX

dh_dx = interpolationFunctionsDerivatives(XI, h, nodes_per_element, dimension)

print("derivatives of interpolation functions:")
print("---------------------------------------")
display(dh_dx)
```

```
derivatives of interpolation functions:
---------------------------------------
```

$$\begin{bmatrix} 25.0\xi_1 - 2.5 & 25.0\xi_0 - 2.5 \\ -25.0\xi_1 - 2.5 & 2.5 - 25.0\xi_0 \\ 2.5 - 25.0\xi_1 & -25.0\xi_0 - 2.5 \\ 25.0\xi_1 + 2.5 & 25.0\xi_0 + 2.5 \end{bmatrix}$$

5.2.6 Problem Formulation

The following input script already makes use of the problem formulation scheme of optimal control problems. In order to make the analogy more readable we rename the functions occurring in the Poisson Equation in order to make it look like an optimal control problem:

$$\nabla^2 S(\underline{x}) + F(\underline{x}) = 0$$

$$S(\underline{x}) = E(\underline{x}) \quad \{\underline{x} \in \Omega : \underline{e}(\underline{x}) = \underline{0}\}$$

The domain Ω was already defined when the mesh array was initialized. The functions $F(\underline{x}), e(\underline{x})$ and $E(\underline{x})$ are defined here:

[10]:
```
""" input skript """

# running or "lagrangian" cost
def F(X):
    x0 = X[0]
    x1 = X[1]
    F = 1
    return F

# define boundary (endpoint constraint)
def e(X_f):
    xf0 = X_f[0]
    xf1 = X_f[1]
    e0 = xf0
    e1 = 0
    e = [e0, e1]
    return e
```

```
# define boundary value (endpoint or "mayer" cost)
def E(X_f):
    xf0 = X_f[0]
    xf1 = X_f[1]
    E = 0
    return E
```

The used syntax is also common for ODEs in Python. Note that the user can formulate every Optimal Control relevant constraint analogously to the analytical formulation. For a general purpose FEM software this kind of input is not sufficient as it is not possible to define complicated boundaries within this simple framework.

5.2.7 Variational Formulation

For the Poisson example the variational formulation of the operator equation can be expressed simply by taking the dot product of the two symbolic vectors $S_{\underline{x}}$ and $\nu_{\underline{x}}$.

[11]:
```
X = numpy.asarray(sympy.symbols('x:' + str(dimension)))
V = numpy.asarray(sympy.symbols('v_x:' + str(dimension)))
S_x = numpy.asarray(sympy.symbols('S_x:' + str(dimension)))

Au = numpy.dot( S_x, V ) # Operator Equation

print("Variational Formulation of the Operator Equation:")
print("------------------------------------------------")
display(Au)
```

```
Variational Formulation of the Operator Equation:
------------------------------------------------
```

$S_{x0}v_{x0} + S_{x1}v_{x1}$

5.2.8 ESM and ELV

As the specific Poisson example treated in this section is not explicitly dependent on the coordinate frame and all elements have the same shape and orientation in the domain, it follows, that every element has the same element stiffness matrix and element load vector. Note that the algorithm could also treat problems where the equation is explicitly dependent on the domain (on a global coordinate frame). The Software handles such problems, other than most FE-Software-tools, by formulating the ESM as a function of the domain. In the assembly every ESM is then evaluated according to the specific location of the element. ESM and ELV are computed by substituting $S_{\underline{x}}$ and $\nu_{\underline{x}}$ with specific derivates of the interpolation functions (according to the usual FE-scheme). Then every element of the arrays is integrated analytically over the domain of one element.

[12]:
```
def elementStiffnessMatrixAnalytic(Au, V, S_x, h, dh_dx, X, XI,␣
↪nodes_per_element, dimension, resolution):

    # evaluate componets of k_ij (not integrated jet)
    k_ij = sympy.zeros(nodes_per_element,nodes_per_element)
    for i in range(nodes_per_element):
        for j in range(nodes_per_element):
            sub = []
            for k in range(dimension):
                sub.append((S_x[k], dh_dx[i,k]))
                sub.append((V[k], dh_dx[j,k]))
            k_ij[i,j] = Au.subs(sub)

    # prepare substitution for coordinate transformation
    x_to_xi = []
    for j in range(dimension):
        x_to_xi.append( (X[j], XI[j] + X[j]) )

    # transform to local coordinate frame and integrate
    for indices in numpy.ndindex(k_ij.shape):
        k_ij[indices] = k_ij[indices].subs(x_to_xi)
        for i in range(dimension):
            k_ij[indices] = sympy.integrate(k_ij[indices],(XI[i],-resolution/
↪2,resolution/2))

    return k_ij

k_ij = elementStiffnessMatrixAnalytic(Au, V, S_x, h, dh_dx, X, XI,␣
↪nodes_per_element, dimension, resolution)

print("Elementstiffnessmatrix:")
print("-----------------------")
display(k_ij.evalf(3))
```

```
Elementstiffnessmatrix:
-----------------------
```

$$\begin{bmatrix} 0.667 & -0.167 & -0.167 & -0.333 \\ -0.167 & 0.667 & -0.333 & -0.167 \\ -0.167 & -0.333 & 0.667 & -0.167 \\ -0.333 & -0.167 & -0.167 & 0.667 \end{bmatrix}$$

[13]:
```
def elementLoadVectorAnalytic(h, F, X, XI, nodes_per_element, dimension,␣
↪resolution):

    f_j = h * F(X)

    # prepare substitution for transformation
    x_to_xi = []
    for j in range(dimension):
        x_to_xi.append( (X[j], XI[j] + X[j]) )

    # transform and integrate
    for j in range(nodes_per_element):
        f_j[j] = f_j[j].subs(x_to_xi)
        for dim in range(dimension):
            f_j[j] = sympy.integrate(f_j[j],(XI[dim], -resolution/2, resolution/
↪2))
    return f_j

f_j = elementLoadVectorAnalytic(h, F, X, XI, nodes_per_element, dimension,␣
↪resolution)

print("Elementloadvector:")
print("------------------")
display(f_j)
```

```
Elementloadvector:
------------------
```

$$\begin{bmatrix} 0.01 \\ 0.01 \\ 0.01 \\ 0.01 \end{bmatrix}$$

5.2.9 The Assembly

As mentioned before, the assembly-algorithm does not only build the stiffness matrix and load vector from given vectors and matrices but also evaluates the element matrix and element vector if they are passed to the *assembly* function as functions of the global coordinate frame. `assemble` does this by converting the indices of an element to the actual location of the element in the mesh. The variables are then evaluated for the specific location of the mesh. The approach is unusual and is also enabled and limited by the capabilities of sympy.

[14]:
```
def assemble(T, DoF, k_ij, f_j, X, nodes_per_element, resolution):

    # initialize transport arrays
    k = numpy.zeros((nodes_per_element, nodes_per_element))
    f = numpy.zeros((nodes_per_element, 1))

    # initialize stiffnessmatrix and loadvector of the whole system
    K_ = numpy.zeros((DoF,DoF))
    F_ = numpy.zeros((DoF,1))

    el = 0 # element counter
    for e_index in numpy.ndindex(nr_el_dim): # for every element

        # prepare substitution for element
        evaluate = []
        e_location = numpy.asarray(e_index) * resolution + numpy.
↪ones(dimension)*resolution
        for i in range(dimension):
            evaluate.append( (X[i], e_location[i] ))

        # evaluate k_ij and f_j for element
        for indices in numpy.ndindex((nodes_per_element, nodes_per_element)):
            k[indices] = k_ij[indices].subs(evaluate)

        for j in range(nodes_per_element):
            f[j,0] = f_j[j].subs(evaluate)

        # assemble K_ and F_
        for k_index in numpy.ndindex((nodes_per_element, nodes_per_element)):
            i = T[el,k_index[0]]
            j = T[el,k_index[1]]
            K_[i,j] += k[k_index] #round(k[k_index],1)

        for f_index in range(nodes_per_element):
            F_[T[el,f_index],0] += f[f_index,0] #round(f[f_index,0],1)

        el += 1 # next element

    return K_, F_

K_, F_ = assemble(T, DoF, k_ij, f_j, X, nodes_per_element, resolution)

numpy.set_printoptions(precision=3)
print("Stiffnessmatrix:")
print("----------------")
print(K_)
```

```
Stiffnessmatrix:
----------------
[[ 0.667 -0.167  0.     ...  0.     0.     0.   ]
 [-0.167  1.333 -0.167 ...  0.     0.     0.   ]
 [ 0.    -0.167  1.333 ...  0.     0.     0.   ]
 ...
 [ 0.     0.     0.    ...  1.333 -0.167  0.   ]
 [ 0.     0.     0.    ... -0.167  1.333 -0.167]
 [ 0.     0.     0.    ...  0.    -0.167  0.667]]
```

5.2.10 Applying the Boundary Condition

The following function searches nodes that lie on the defined boundary by evaluation of the boundary function $\underline{e}$ for every location of every node. If the boundary function returns the null-vector then the corresponding node lies on the boundary. the function returns then the names, indices and locations of the boundary-nodes.

[15]:
```
def boundaryNodes(e, domain, nr_nd_dim, dimension, resolution):
    node_locations_global = numpy.zeros(nr_nd_dim + (dimension,))
    initialPoint = []
    for i in range(dimension):
        initialPoint.append(domain[i][0])
    initialPoint = numpy.asarray(initialPoint)
    for indices in numpy.ndindex(nr_nd_dim):
        step = numpy.asarray(list(indices))
        node_locations_global[indices] = initialPoint + step * resolution
    boundary_nodes = []
    boundary_node_indices = []
    length = len(e(node_locations_global[indices]))
    for indices in numpy.ndindex(nr_nd_dim):
        if e(node_locations_global[indices]) == list((0,)*length):
            boundary_nodes.append(node_names_global[indices])
            boundary_node_indices.append(indices)
    if boundary_nodes == []:
        print("ERROR: no boundary nodes found.")
    return boundary_nodes, boundary_node_indices, node_locations_global

bNodes, bNodeInd, nodeLoc = boundaryNodes(e, domain, nr_nd_dim, dimension,␣
↪resolution)
print("Boundary Node (Names) =",bNodes)
print("Boundary Node Indices =",bNodeInd)
#print("Boundary Node Locations :")
#print("-------------------------")
#print(nodeLoc)
```

```
Boundary Node (Names) = [0, 1, 2, 3, 4, 5]
Boundary Node Indices = [(0, 0), (0, 1), (0, 2), (0, 3), (0, 4), (0,␣
↪5)]
```

Now the boundary condition is applied. The solution vector $\underline{U}$ is first initialized as array of zeros. Then the known values are put in to the solution vector. Therefore, the boundary value function E is evaluated for the locations of the boundary nodes. The resulting values are then placed in the correct position of $\underline{U}$. After this a matrix product $\underline{\underline{K}}\ \underline{U}$ is calculated. The result is subtracted from the load vector $\underline{F}$. This procedure is necessary for non-zero boundary values. Afterwards the columns and rows that correspond to known values of $\underline{U}$ are deleted from the system of equations in order to compute the remaining part of $\underline{U}$ by inverting $\underline{\underline{K}}$ and multiplying it with the load vector. afterwards, the known boundary values of the solution vector are substituted back and $\underline{U}$ is reshaped for the visualization.

[16]:
```
from numpy.linalg import inv

def applyBoundaryCondition(K_, F_, E, DoF, bNodes, bNodeInd, nodeLoc, nr_nd_dim):
    U_ = numpy.zeros((DoF,1))
    i = 0
    for node in bNodes:
        U_[node] = E(nodeLoc[bNodeInd[i]])
        i += 1
    delta_F = numpy.dot(K_,U_)
    F_ = F_ - delta_F

    # delete rows with fixed U values from the system of equations
    K_ = numpy.delete(K_, bNodes, 0)
    K_ = numpy.delete(K_, bNodes, 1)
    F_ = numpy.array([numpy.delete(F_, bNodes)]).T

    # solve system of equations
    U_ = numpy.dot( inv(K_), F_ )

    # insert boundary values
    bvals =[]
    for i in bNodeInd:
        bvals.append(E(nodeLoc[i]))

    for i in bNodes:
        U_ = numpy.insert(U_, i, bvals[i])

    U_ = numpy.array([U_]).T

    S = numpy.reshape(U_,(nr_nd_dim),order='C')

    return S

S = applyBoundaryCondition(K_, F_, E, DoF, bNodes, bNodeInd, nodeLoc, nr_nd_dim)

print("FE-Solution:")
print("------------")
print(S)
```

```
FE-Solution:
------------
[[0.   0.   0.   0.   0.   0.  ]
 [0.38 0.38 0.38 0.38 0.38 0.38]
 [0.72 0.72 0.72 0.72 0.72 0.72]
 [1.02 1.02 1.02 1.02 1.02 1.02]
 [1.28 1.28 1.28 1.28 1.28 1.28]
 [1.5  1.5  1.5  1.5  1.5  1.5 ]
 [1.68 1.68 1.68 1.68 1.68 1.68]
 [1.82 1.82 1.82 1.82 1.82 1.82]
 [1.92 1.92 1.92 1.92 1.92 1.92]
 [1.98 1.98 1.98 1.98 1.98 1.98]
 [2.   2.   2.   2.   2.   2.  ]]
```

[17]:
```
import matplotlib.pyplot as plt

S = numpy.flipud(S.T)

fig, ax = plt.subplots(dpi=120)
im = ax.imshow(S, origin='lower', extent=[0, 2, 0, 1],vmax=abs(S).max(),␣
↪vmin=abs(S).min())
cbar = plt.colorbar(im)
cbar.set_label('S(x,y)')
ax.set_ylabel('y')
ax.set_xlabel('x')
plt.show()
```

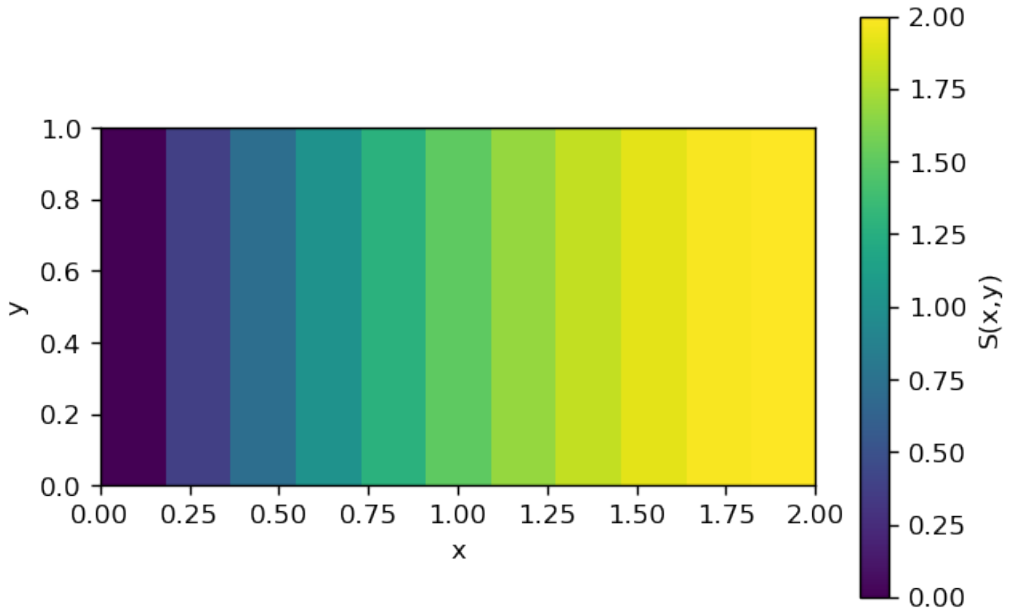

Figure 5.1: Output: FE-Solution of the Poisson Equation with low resolution mesh for Dirichlet boundary condition $u(x = 0, y) = 0$ and $u(x = 2, y) = 2$.

$$Error = \left(\int_{\Omega} (\nabla^2 S(\underline{x}) - F)^2 d\Omega \right)^{\frac{1}{2}}$$

[18]:
```
Laplace_S = numpy.zeros(S.shape)
F = numpy.ones(S.shape) * 2
for i in range(dimension):
    Laplace_S += numpy.gradient(numpy.gradient(S)[i]/resolution)[i]/resolution

Error = 0
for indices in numpy.ndindex(S.shape):
    Error += (Laplace_S[indices]+F[indices])**2 * resolution**dimension
Error = Error**0.5

print("Error:")
print("------")
print(Error)
```

```
Error:
------
1.8734993995195384
```

The above results can be verified by comparing them to the analytical solution of the same problem. Therefore, the developed software can be used to solve linear PDEs where the variational form is not dependent on boundary integrals and only Dirichlet boundary conditions have to be applied.

Chapter 6

Experiments and Results

6.1 The ARTOC Toolbox

To use the code of the former section in a more compact and practical way a simple library called `artoc`[1] is introduced. To test it the Poisson problem is revisited:

[1]:
```
from IPython.display import display
import matplotlib.pyplot as plt
from artoc import *
import numpy
import sympy
numpy.set_printoptions(precision=2)
numpy.set_printoptions(suppress=True)


""" input skript """

# define mesh
domain = [(0,2),(0,1)]
resolution = 0.02

# define boundary
def e(X_f):
    xf0 = X_f[0]
    xf1 = X_f[1]
    e0 = xf0
    e1 = 0
    e = [e0, e1]
    return e

# define boundary value
def E(X_f):
    xf0 = X_f[0]
    xf1 = X_f[1]
    E = 0
    return E
```

[1] A.R.T.O.C. stands for Adaptive Real-Time Optimal Control referring to the capacity of the method to also enable adaptive control

```
# define weak formulation
dimension = len(domain) # dimension of the state space
v = sympy.Symbol('v') # test function
V_x = numpy.asarray(sympy.symbols('v_x:' + str(dimension)))
S_x = numpy.asarray(sympy.symbols('S_x:' + str(dimension)))

Au_v = numpy.dot( S_x, V_x )
f_v = v

OperatorEq = sympy.Eq(Au_v - f_v, 0)

print("Variational Formulation of the Operator Equation:")
print("------------------------------------------------")
display(OperatorEq)
```

```
Variational Formulation of the Operator Equation:
------------------------------------------------
```

$$S_{x0}v_{x0} + S_{x1}v_{x1} - v = 0$$

As all key-classes and functions of `artoc` were already imported using the command `from artoc import *` the user can now initialize a mesh object simply by calling the function `mesh` which simply takes the input `domain` and `resolution`. Now the user can attach further requirements and commands to the mesh-object. In the code below the mesh-object is assigned to the variable `m`. To solve the Poisson equation in the defined mesh the user can simply call the function `m.solveWeakForm()` and input the symbolic functions according to the following syntax:

[2]:
```
m = mesh(domain, resolution) # initialize mesh object
S = m.solveWeakForm( Au_v, f_v, e, E) # solve weak form in defined mesh
print(S)
S = numpy.flipud(S.T)
fig, ax = plt.subplots(dpi=120)
im = ax.imshow(S,extent=[0, 2, 0, 1],vmax=S.max(), vmin=S.min())
cbar = plt.colorbar(im)
cbar.set_label('S(x,y)')
ax.set_ylabel('y')
ax.set_xlabel('x')
plt.show()
```

```
[[0.   0.   0.   ... 0.   0.   0.  ]
 [0.04 0.04 0.04 ... 0.04 0.04 0.04]
 [0.08 0.08 0.08 ... 0.08 0.08 0.08]
 ...
 [2.   2.   2.   ... 2.   2.   2.  ]
 [2.   2.   2.   ... 2.   2.   2.  ]
 [2.   2.   2.   ... 2.   2.   2.  ]]
```

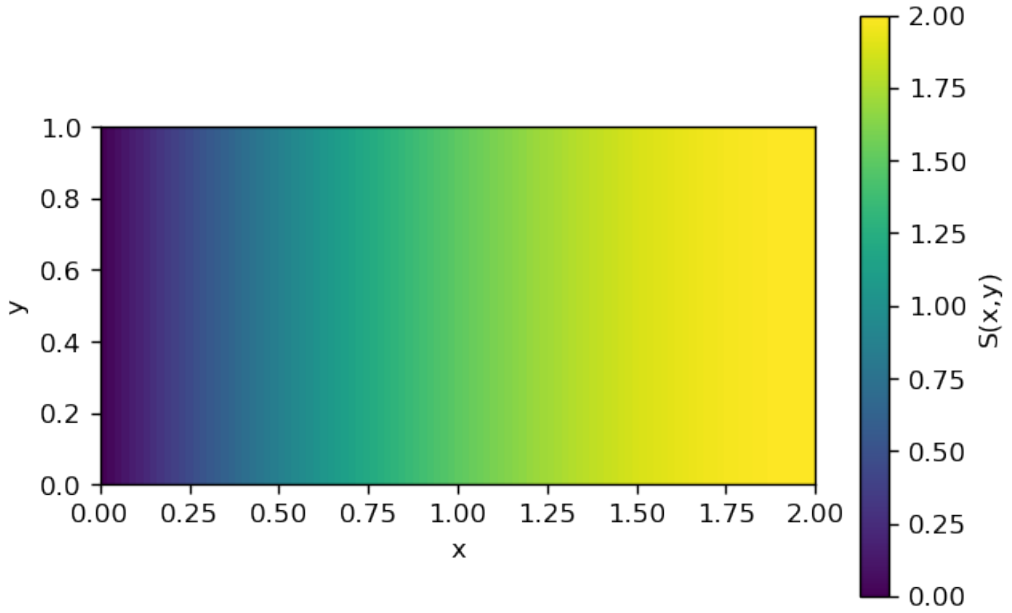

Figure 6.1: Output: FE-Solution to the Poisson Equation with high resolution mesh

[3]:
```
Laplace_S = numpy.zeros(S.shape)
F = numpy.ones(S.shape) * 2
for i in range(dimension):
    Laplace_S += numpy.gradient(numpy.gradient(S)[i]/resolution)[i]/resolution

Error = 0
for indices in numpy.ndindex(S.shape):
    Error += (Laplace_S[indices]+F[indices])**2 * resolution**dimension
Error = Error**0.5

print("Error:")
print("------")
print(Error)
```

```
Error:
------
1.4609414772709162
```

As expected the error in the numerical solution is reduced by increasing the number of finite elements used for the calculation.

6.2 The Poisson Equation in four dimensions

To test `artoc's` capability to solve higher dimensional PDEs another Poisson example is considered. This time the objective is to solve the Equation in a four dimensional unit cube. By calling additional features of `artoc` the drastic increase of nodes per element and interpolation functions can be viewed:

[4]:
```
domain = [(0,1),(0,1),(0,1),(0,1)]
resolution = 0.2
m = mesh(domain, resolution)

print("Nodes per Element:",m.nodes_per_element)
print("Global Nodes (DoF):", m.number_nodes)
```

```
Nodes per Element: 16
Global Nodes (DoF): 1296
```

[5]:
```
display(m.interpolationFunctions())
```

$$\begin{bmatrix} \frac{(1-10.0\xi_0)(1-10.0\xi_1)(1-10.0\xi_2)(1-10.0\xi_3)}{16} \\ \frac{(1-10.0\xi_0)(1-10.0\xi_1)(1-10.0\xi_2)(10.0\xi_3+1)}{16} \\ \frac{(1-10.0\xi_0)(1-10.0\xi_1)(1-10.0\xi_3)(10.0\xi_2+1)}{16} \\ \frac{(1-10.0\xi_0)(1-10.0\xi_1)(10.0\xi_2+1)(10.0\xi_3+1)}{16} \\ \frac{(1-10.0\xi_0)(1-10.0\xi_2)(1-10.0\xi_3)(10.0\xi_1+1)}{16} \\ \frac{(1-10.0\xi_0)(1-10.0\xi_2)(10.0\xi_1+1)(10.0\xi_3+1)}{16} \\ \frac{(1-10.0\xi_0)(1-10.0\xi_3)(10.0\xi_1+1)(10.0\xi_2+1)}{16} \\ \frac{(1-10.0\xi_0)(10.0\xi_1+1)(10.0\xi_2+1)(10.0\xi_3+1)}{16} \\ \frac{(1-10.0\xi_1)(1-10.0\xi_2)(1-10.0\xi_3)(10.0\xi_0+1)}{16} \\ \frac{(1-10.0\xi_1)(1-10.0\xi_2)(10.0\xi_0+1)(10.0\xi_3+1)}{16} \\ \frac{(1-10.0\xi_1)(1-10.0\xi_3)(10.0\xi_0+1)(10.0\xi_2+1)}{16} \\ \frac{(1-10.0\xi_1)(10.0\xi_0+1)(10.0\xi_2+1)(10.0\xi_3+1)}{16} \\ \frac{(1-10.0\xi_2)(1-10.0\xi_3)(10.0\xi_0+1)(10.0\xi_1+1)}{16} \\ \frac{(1-10.0\xi_2)(10.0\xi_0+1)(10.0\xi_1+1)(10.0\xi_3+1)}{16} \\ \frac{(1-10.0\xi_3)(10.0\xi_0+1)(10.0\xi_1+1)(10.0\xi_2+1)}{16} \\ \frac{(10.0\xi_0+1)(10.0\xi_1+1)(10.0\xi_2+1)(10.0\xi_3+1)}{16} \end{bmatrix}$$

The problem formulation looks the same as in any number of dimensions due to the syntax of `sympy`:

[7]:
```
# define weak formulation
dimension = len(domain) # dimension of the state space
v = sympy.Symbol('v') # test function
V_x = numpy.asarray(sympy.symbols('v_x:' + str(dimension)))
S_x = numpy.asarray(sympy.symbols('S_x:' + str(dimension)))

Au_v = numpy.dot( S_x, V_x )
f_v = v

OperatorEq = sympy.Eq(Au_v - f_v, 0)

print("Variational Formulation of the Operator Equation:")
print("------------------------------------------------")
display(OperatorEq)
```

```
Variational Formulation of the Operator Equation:
------------------------------------------------
```

$$S_{x0}v_{x0} + S_{x1}v_{x1} + S_{x2}v_{x2} + S_{x3}v_{x3} - v = 0$$

As four dimensional problems can't be visualized properly only the error-norm is computed for the solution.

[9]:
```
S = m.solveWeakForm( Au_v, f_v, e, E) # solve weak form in defined mesh

Laplace_S = numpy.zeros(S.shape)
F = numpy.ones(S.shape) * 2
for i in range(dimension):
    Laplace_S += numpy.gradient(numpy.gradient(S)[i]/resolution)[i]/resolution

Error = 0
for indices in numpy.ndindex(S.shape):
    Error += (Laplace_S[indices]+F[indices])**2 * resolution**dimension
Error = Error**0.5

print("Error:")
print("------")
print(Error)
```

```
Error:
------
1.823842098428484
```

As the computed error is close to the error of the two dimensional Poisson example with the same element size, which was obviously correct, the author concludes that the algorithm works properly.

6.3 Weighted Residual Methods for the piecewise linear HJB

6.3.1 Automated formulation of the HJB

In order to develop a user-friendly Optimal Control software the problem formulation was also automated as far as possible (again with the `sympy` toolbox). The following input script contains all the information about the Optimal Control problem from section 3.3.

[10]:
```
""" input skript """
# define mesh
domain = [(-1,1),(-1,1)]
resolution = 0.2
dimension = len(domain)

# dynamic constraint f := f(X,U)
def f(X,U):
    u0 = U[0]
    u1 = U[1]
    x0 = X[0]
    x1 = X[1]
    dx0_dt = x1
    dx1_dt = - x1 + u0 - u1
    X_dot = [dx0_dt, dx1_dt]
    return X_dot
```

```
u_max = [1,1] # max values of the control vector

# running cost F := F(X)
def F(X):
    x0 = X[0]
    x1 = X[1]
    F = 1
    return F

# endpoint constraint e := e(X_f) = 0
def e(X_f):
    xf0 = X_f[0]
    xf1 = X_f[1]
    e0 = xf0
    e1 = xf1
    e = [e0, e1]
    return e

# endpoint cost E := E(X_f)
def E(X_f):
    xf0 = X_f[0]
    xf1 = X_f[1]
    E = 0
    return E
```

As can be seen the operator of the HJB Equation can be formulated automatically by a function dependent on the dynamics. For more information on this function see Appendix B.

[11]:
```
from artoc import *
Au = HJB_piecewiseLinear(f, dimension, u_max)
display(Au)
```

$$S_{x0}x_1 - S_{x1}\left(x_1 + 1.0\,\mathrm{sign}\left(S_{x1}\right)\right)$$

The remaining Part of the equation is simply the running cost function $F(\underline{x})$.

6.3.2 Galerkin's Method

`artoc` is designed to resemble the analytical integral equations of the theory behind the finite element method. Therefore, having the operator defined, it is reasonable to define a symbolic test-function `v` and formulate the weighted integral statement according to Galerkin's Method.

[12]:
```
m = mesh(domain, resolution) # initialize mesh object
v = sympy.Symbol('v') # test function
X = numpy.asarray(sympy.symbols('x:' + str(dimension)))

S = m.solveWeakForm( Au*v,-F(X)*v, e, E) # solve weak form in defined mesh
print(S)

S = numpy.flipud(S.T)
```

```
fig, ax = plt.subplots(dpi=120)
im = ax.imshow(S,extent=[-1, 1, -1, 1],vmax=S.max(), vmin=S.min())
cbar = plt.colorbar(im)
cbar.set_label('S(x,y)')
ax.set_ylabel('y')
ax.set_xlabel('x')
plt.show()
```

```
[[ 0.12  0.07  0.04  0.06  0.    0.06 -0.01  0.05 -0.03  0.07 -0.06]
 [ 0.13  0.09  0.15  0.07  0.14  0.04  0.11  0.03  0.13 -0.    0.1 ]
 [ 0.09  0.13  0.08  0.15  0.08  0.17  0.09  0.16  0.03  0.18 -0.13]
 [ 0.14  0.08  0.15  0.06  0.15  0.03  0.12  0.02  0.2  -0.08  0.17]
 [ 0.07  0.16  0.06  0.18  0.07  0.2   0.11  0.24 -0.04  0.26 -0.21]
 [ 0.16  0.05  0.18  0.02  0.15  0.    0.1  -0.1   0.29 -0.17  0.26]
 [ 0.04  0.19  0.03  0.24  0.06  0.4   0.05  0.47 -0.16  0.37 -0.31]
 [ 0.21  0.01  0.22 -0.08  0.17 -0.28  0.1  -0.32  0.3  -0.21  0.3 ]
 [-0.02  0.25 -0.01  0.38  0.08  0.63  0.2   0.71 -0.1   0.44 -0.38]
 [ 0.29 -0.09  0.25 -0.27  0.08 -0.6  -0.11 -0.73  0.22 -0.36  0.42]
 [-0.14  0.41  0.    0.67  0.26  1.1   0.55  1.26  0.15  0.6  -0.48]]
```

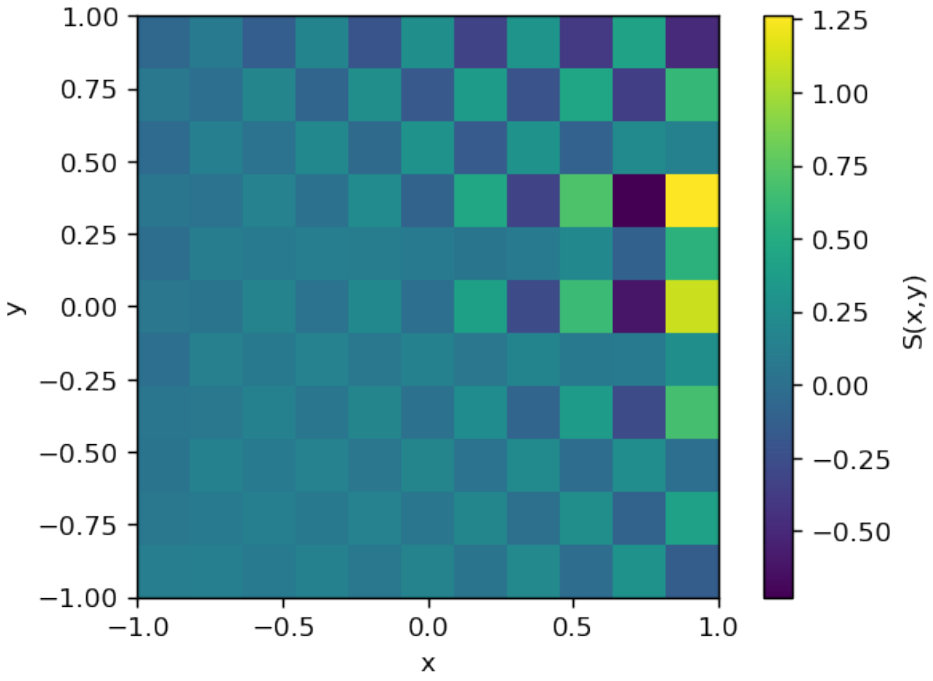

Figure 6.2: Output: Invalid Galerkin-solution due to the piecewise linear HJB Equation with low resolution mesh. The Dirichlet boundary condition was applied only on a single node: $u(0,0) = 0$ (as required by the problem formulation).

As can be seen the generated solution is invalid, which is due to the wrong assumption that piecewise linear PDEs can be solved via the linear Finite Element Method (for a detailed explanation see section 4.3.1). To ensure that the experiment did not only fail due to mesh convergence issues the calculation was also carried out for a finer mesh:

[13]:
```
resolution = 0.02
m = mesh(domain, resolution) # initialize mesh object
S = m.solveWeakForm( Au*v,-F(X)*v, e, E) # solve weak form in defined mesh
S = numpy.flipud(S.T)
fig, ax = plt.subplots(dpi=120)
im = ax.imshow(S,extent=[-1, 1, -1, 1],vmax=S.max(), vmin=S.min())
cbar = plt.colorbar(im)
cbar.set_label('S(x,y)')
ax.set_ylabel('y')
ax.set_xlabel('x')
plt.show()
```

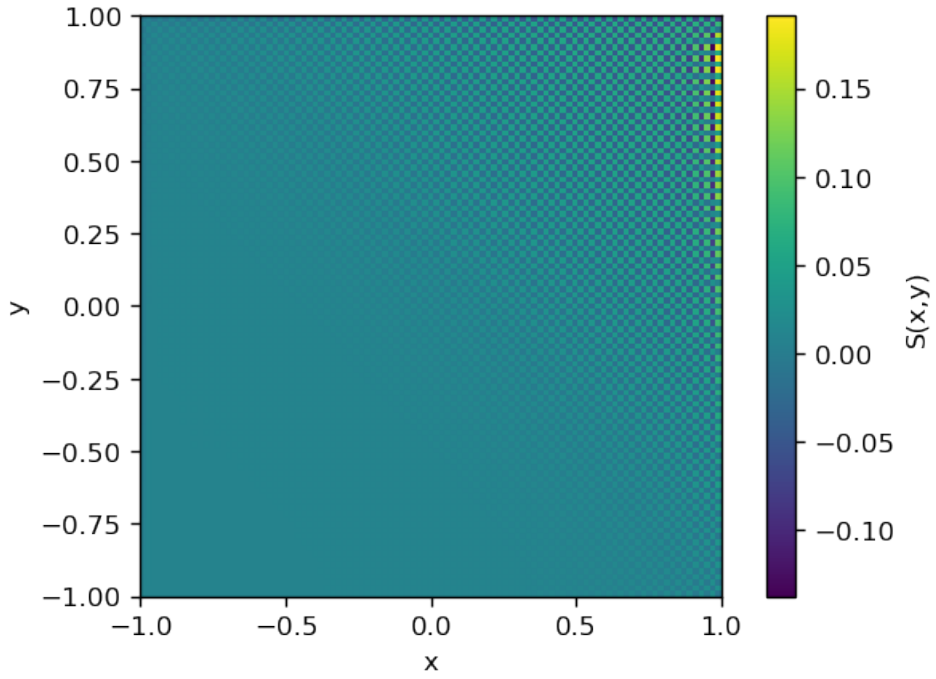

Figure 6.3: Output: Invalid Galerkin-solution due to the piecewise linear HJB Equation with high resolution mesh. The Dirichlet boundary condition was applied only on a single node: $u(0,0) = 0$ (as required by the problem formulation).

6.3.3 The Least Squares Method

The Least Squares Method for linear problems can be viewed as a special case of the Weighted Residual Method where the weight function is the operator equation in dependence of a test function. The method can be derived by squaring the integral operator equation and minimizing it with respect to the design variables. According to that it is the only other method, in addition to Ritz's Method, that is based on the minimization of a functional [Red84]. Therefore, the weighting function can be simply generated by substituting the operator equation with test functions. Of course the resulting element stiffness matrix will be quadratically dependent on $\underline{x}$.

[14]:
```
V_x = numpy.asarray(sympy.symbols('v_x:' + str(dimension)))
w = Au.subs([(S_x[0],V_x[0]),(S_x[1],V_x[1])]) # weight function
k, f = m.ESM_ELV_weak(Au*w,-F(X)*w)
display(k.evalf(1))
```

$$\begin{bmatrix} 0.2x_1^2 - 0.2x_1 + 0.3 & -0.2x_1^2 + 0.5x_1 + 0.3 & -0.2x_1^2 - 0.3x_1 + 0.2 & 0.2x_1^2 + 0.2 \\ -0.2x_1^2 + 0.5x_1 + 0.3 & 1.0x_1^2 + 1.0x_1 + 0.3 & 0.2 - 0.8x_1^2 & -0.2x_1^2 + 0.3x_1 + 0.2 \\ -0.2x_1^2 - 0.3x_1 + 0.2 & 0.2 - 0.8x_1^2 & 1.0x_1^2 - 1.0x_1 + 0.3 & -0.2x_1^2 - 0.5x_1 + 0.3 \\ 0.2x_1^2 + 0.2 & -0.2x_1^2 + 0.3x_1 + 0.2 & -0.2x_1^2 - 0.5x_1 + 0.3 & 0.2x_1^2 + 0.2x_1 + 0.3 \end{bmatrix}$$

[15]:
```
resolution = 0.2
m = mesh(domain, resolution) # initialize mesh object
S = m.solveWeakForm( Au*w,-F(X)*w, e, E) # solve weak form in defined mesh
print(S)
S = numpy.flipud(S.T)
fig, ax = plt.subplots(dpi=120)
im = ax.imshow(S,extent=[-1, 1, -1, 1],vmax=S.max(), vmin=S.min())
cbar = plt.colorbar(im)
cbar.set_label('S(x,y)')
ax.set_ylabel('y')
ax.set_xlabel('x')
plt.show()
```

```
[[0.1  0.1  0.1  0.1  0.1  0.1  0.1  0.1  0.1  0.1  0.1 ]
 [0.1  0.1  0.1  0.1  0.1  0.1  0.1  0.1  0.1  0.1  0.1 ]
 [0.11 0.09 0.1  0.1  0.1  0.1  0.1  0.1  0.1  0.1  0.1 ]
 [0.12 0.08 0.11 0.1  0.1  0.1  0.1  0.1  0.1  0.1  0.1 ]
 [0.12 0.08 0.11 0.09 0.11 0.1  0.08 0.1  0.1  0.1  0.1 ]
 [0.09 0.11 0.09 0.1  0.12 0.   0.11 0.1  0.1  0.1  0.1 ]
 [0.07 0.12 0.09 0.11 0.08 0.1  0.11 0.1  0.1  0.1  0.1 ]
 [0.1  0.1  0.1  0.1  0.1  0.1  0.1  0.1  0.1  0.1  0.1 ]
 [0.1  0.1  0.1  0.1  0.1  0.1  0.1  0.1  0.1  0.1  0.1 ]
 [0.1  0.1  0.1  0.1  0.1  0.1  0.1  0.1  0.1  0.1  0.1 ]
 [0.1  0.1  0.1  0.1  0.1  0.1  0.1  0.1  0.1  0.1  0.1 ]]
```

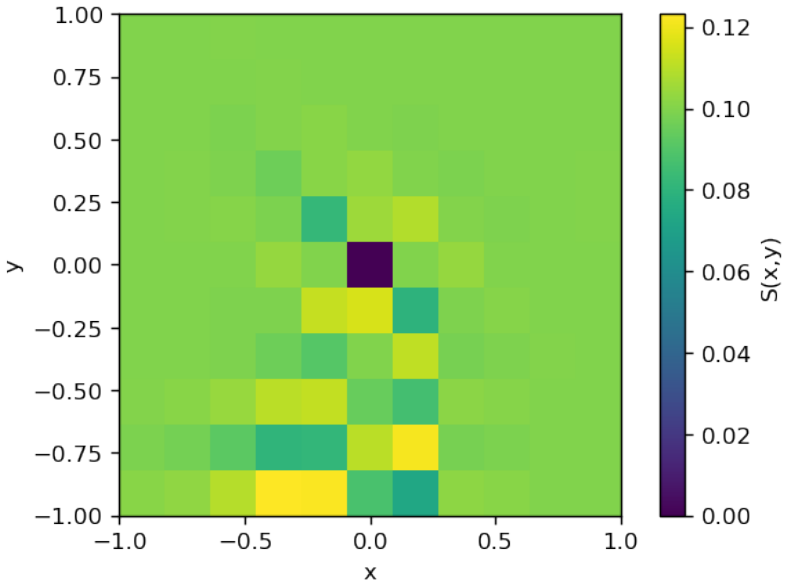

Figure 6.4: Output: Invalid Least-Squares-solution of the piecewise linear HJB Equation with low resolution mesh. The Dirichlet boundary condition was applied only on a single node: $u(0,0) = 0$.

Just like for the Galerkin Method, the solution is obviously invalid.

[16]:
```
resolution = 0.02
m = mesh(domain, resolution) # initialize mesh object
S = m.solveWeakForm( Au*w,-F(X)*w, e, E) # solve weak form in defined mesh
S = numpy.flipud(S.T)

fig, ax = plt.subplots(dpi=120)
im = ax.imshow(S,extent=[-1, 1, -1, 1],vmax=S.max(), vmin=S.min())
cbar = plt.colorbar(im)
cbar.set_label('S(x,y)')
ax.set_ylabel('y')
ax.set_xlabel('x')
plt.show()
```

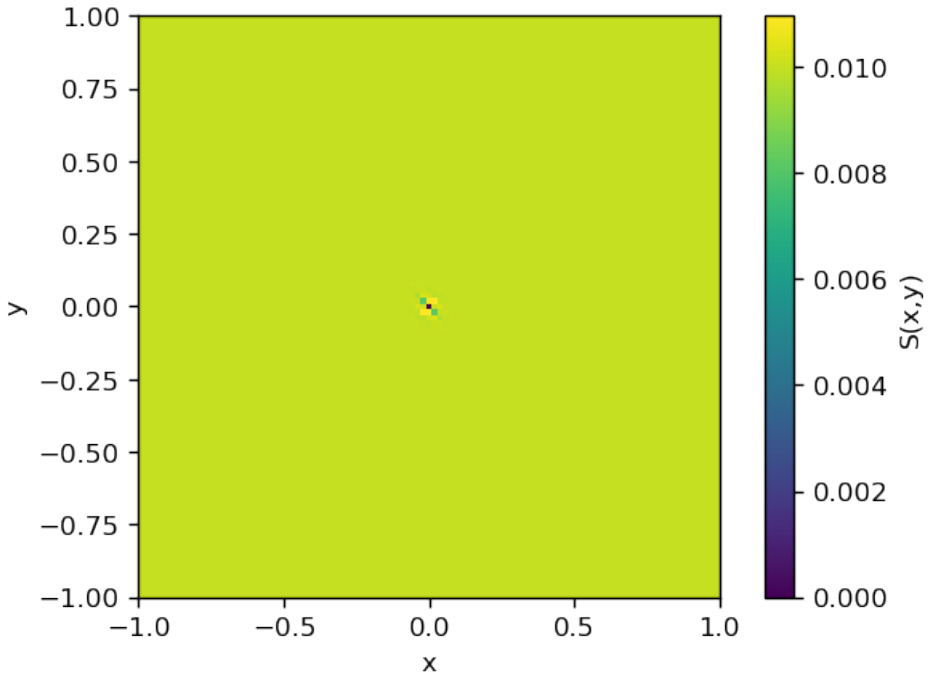

Figure 6.5: Output: Invalid Least-Squares-solution of the piecewise linear HJB Equation with high resolution mesh. The Dirichlet boundary condition was applied only on a single node: $u(0,0) = 0$.

6.4 The Ritz Method

The Ritz Method is usually used when a Problem can be posed as boundary value problem as well as variational problem. The variational problem can then be solved directly by introducing interpolation functions and design variables with respect to which the functional can be minimized. The difference of the following approach is that the integral form of the discretized HJB Equation is minimized by a similar method. This is natural as the HJB was modified in order to be quadratic; after differentiation this leads to a liner system of equations. To the authors knowledge, this is a new approach. The following code formulates the HJB Equation for a quadratic cost functional as introduced in section 4.4.

[17]:
```
from artoc import *

def F(X,U):
    u0 = U[0]
    u1 = U[1]
    x0 = X[0]
    x1 = X[1]
    F = 1
    return F

# dynamic constraint f := f(X,U)
def f(X,U):
    u0 = U[0]
    u1 = U[1]
    x0 = X[0]
    x1 = X[1]
    dx0_dt = x1
    dx1_dt = - x1 + u0 - u1
    X_dot = [dx0_dt, dx1_dt]
    return X_dot

dimension = 2

W = HJB_quadraticCost(f, F, dimension, 2)
display(W)
```

$$S_{x0}x_1 + \frac{S_{x1}^2}{\alpha} + S_{x1}\left(-\frac{2S_{x1}}{\alpha} - x_1\right) + 1$$

`artoc` also provides a function which directly minimizes quadratic functionals and computes the element load vector and element stiffness matrix. Due to the algorithm only the element load vector is explicitly dependent on the global coordinate frame.

[18]:
```
alpha = sympy.Symbol('alpha')
W = W.subs(alpha,1)
k, f = m.ESM_ELV_ritz(W)
display(k)
```

$$\begin{bmatrix} -0.666666666666667 & 0.666666666666667 & -0.333333333333333 & 0.333333333333333 \\ 0.666666666666667 & -0.666666666666667 & 0.333333333333333 & -0.333333333333333 \\ -0.333333333333333 & 0.333333333333333 & -0.666666666666667 & 0.666666666666667 \\ 0.333333333333333 & -0.333333333333333 & 0.666666666666667 & -0.666666666666667 \end{bmatrix}$$

Unfortunately this method did not lead to valid results either. It is now obvious that it is not possible to minimize the error produced by a Finite Element discretization directly. Such practices have to be approached just like in the least squares method where the square of the produced error is minimized.

[19]:
```
domain = [(0,1),(0,1)]
resolution = 0.2
m = mesh(domain, resolution)
S = m.solveRitz(W, e, E)
print(S)
```

```
S = numpy.flipud(S.T)
fig, ax = plt.subplots(dpi=120)
im = ax.imshow(S,extent=[0, 1, 0, 1],vmax=S.max(), vmin=S.min())
cbar = plt.colorbar(im)
cbar.set_label('S(x,y)')
ax.set_ylabel('y')
ax.set_xlabel('x')
plt.show()
```

```
[[ 0.00e+00 -7.85e-01 -1.63e+00 -2.31e+00 -2.84e+00 -3.10e+00]
 [-4.70e-01 -4.57e-01 -1.59e-01 -6.19e-02  2.20e-03 -2.85e-02]
 [-1.90e+09 -1.90e+09 -1.90e+09 -1.90e+09 -1.90e+09 -1.90e+09]
 [ 3.54e+10  3.54e+10  3.54e+10  3.54e+10  3.54e+10  3.54e+10]
 [-2.19e+12 -2.19e+12 -2.19e+12 -2.19e+12 -2.19e+12 -2.19e+12]
 [-4.50e+15 -4.50e+15 -4.50e+15 -4.50e+15 -4.50e+15 -4.50e+15]]
```

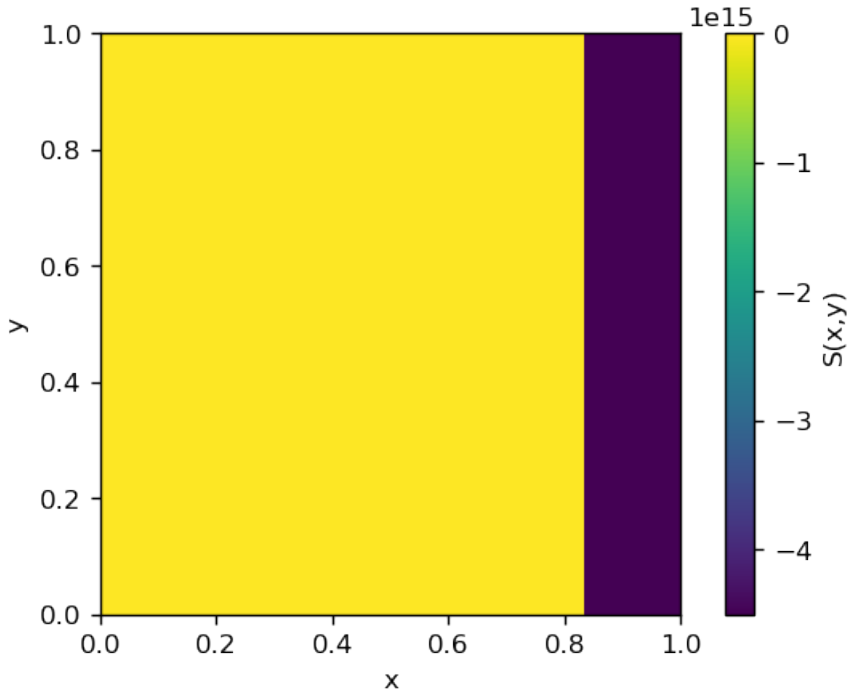

Figure 6.6: Output: Invalid modified Ritz-solution of the piecewise linear HJB Equation with low resolution mesh. The Dirichlet boundary condition was applied only on a single node: $u(0,0) = 0$.

[20]:
```
resolution = 0.02
m = mesh(domain, resolution)
S = m.solveRitz(W, e, E)
S = numpy.flipud(S.T)
fig, ax = plt.subplots(dpi=120)
im = ax.imshow(S,extent=[0, 1, 0, 1],vmax=S.max(), vmin=S.min())
cbar = plt.colorbar(im)
cbar.set_label('S(x,y)')
ax.set_ylabel('y')
ax.set_xlabel('x')
plt.show()
```

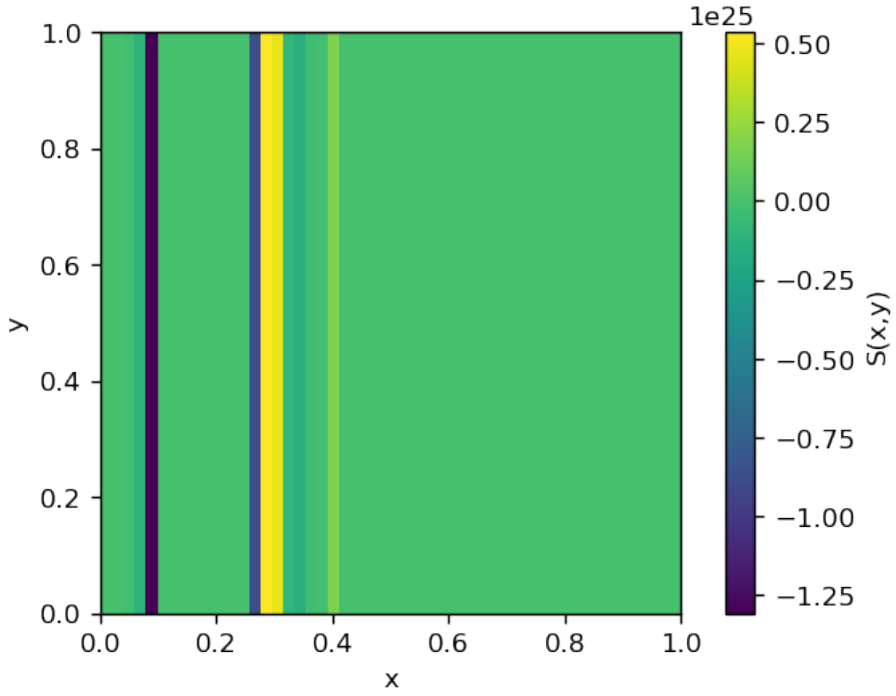

Figure 6.7: Output: Invalid modified Ritz-solution of the piecewise linear HJB Equation with high resolution mesh

As mentioned in section 4.4 the piecewise HJB Equation could also not be solved by the nonlinear Finite Element Method. One could be lead to believe that by using a quadratic functional the resulting PDE of this section would be more suitable for nonlinear methods (due to the quadratic dependencies). Unfortunately a nonlinear Galerkin Method for the above problem was also not successful. The calculations were performed by FEniCS and the error *newton solver did not converge* was encountered.

6.5 A Splitting Method

Consider the following PDEs where equation (6.1) is the Hamilton-Jacobi-Bellman Equation for the optimal control problem of section 4.3.1:

$$|S_{x_1}| + S_{x_0} \cdot x_1 = 0 \tag{6.1}$$
$$S_{x_1} + S_{x_0} \cdot x_1 = 0 \tag{6.2}$$
$$-S_{x_1} + S_{x_0} \cdot x_1 = 0 \tag{6.3}$$

Due to the special features of the magnitude operator the following statements must be true:

- A solution of equation (6.2) is also a solution to equation (6.1) for every $\underline{x}$ for which $S(\underline{x}) \geq 0$ is statisfied.
- A solution of equation (6.3) is also a solution to equation (6.1) for every $\underline{x}$ for which $S(\underline{x}) < 0$ is statisfied.

This gives rise to the opportunity to find solutions to the piecewise liner HJB by splitting it up into two linear problems. After finding a solution to each problem

the resulting strategies can be overlayed to find the solution to the original Optimal Control problem. This procedure shall now be tested for the above problem using the least squares Finite Element Method in FEniCS. First the used functions are defined:

[21]:
```
import numpy
import matplotlib.pyplot as plt
import matplotlib.cm as cm


def toArray(fenicsSolution, length, a, b):
    x = numpy.linspace(a, b, length)
    y = numpy.linspace(a, b, length)
    S = numpy.zeros((length, length))
    for i in range(length):
        for j in range(length):
            S[i,j] = fenicsSolution(x[j], y[i])
    S = numpy.flipud(S)
    return S

def ReLU(array):
    for indices in numpy.ndindex(array.shape):
        if array[indices] < 0.0:
            array[indices] = 0.0
        else:
            pass
    return array

def sgn(array):
    for indices in numpy.ndindex(array.shape):
        if array[indices] < 0.0:
            array[indices] = -1.0
        elif array[indices] > 0.0:
            array[indices] = 1.0
        else:
            pass
    return array

def heav(array):
    for indices in numpy.ndindex(array.shape):
        if array[indices] > 0.0:
            array[indices] = 1.0
        else:
            array[indices] = 0.0
    return array
```

The function `toArray` is introduced to convert the FEniCS solution object to a simple numpy array for processing purposes. The function `ReLU`[2] is a function that returns the input array if the entries are positive. If not, the negative entries are replaced with the value zero. The functions `sgn` and `heav` are the standard sign and Heaviside functions. The following script solves the two linear problems by making use of FEniCS:

[2] `ReLU` stands for *Rectified Linear Unit*

[40]:
```
from fenics import *

a = -1
b = 1
mesh = RectangleMesh(Point(a, a), Point(b, b), 100, 100)
V = FunctionSpace(mesh, 'CG', 2)

# Define boundary condition
tol = 0.04 # tolerance for finding boundary nodes
def boundary(x):
    return near(x[0], 0, tol) and near(x[1], 0, tol)

bc = DirichletBC(V, Constant(0.0), boundary)

# Define variational problem
u = TrialFunction(V)
v = TestFunction(V)
f = Constant(1.0)
y = Expression('x[1]', degree=1)

a1 = (  u.dx(1) - u.dx(0)*y )*(  v.dx(1) - v.dx(0)*y )*dx
a2 = ( -u.dx(1) - u.dx(0)*y )*( -v.dx(1) - v.dx(0)*y )*dx

L1 = f*(  v.dx(1) - v.dx(0)*y )*dx
L2 = f*( -v.dx(1) - v.dx(0)*y )*dx

# Compute solution
u1 = Function(V)
u2 = Function(V)

solve(a1 == L1, u1, bc)
solve(a2 == L2, u2, bc)

plot(u1)
plt.show()
```

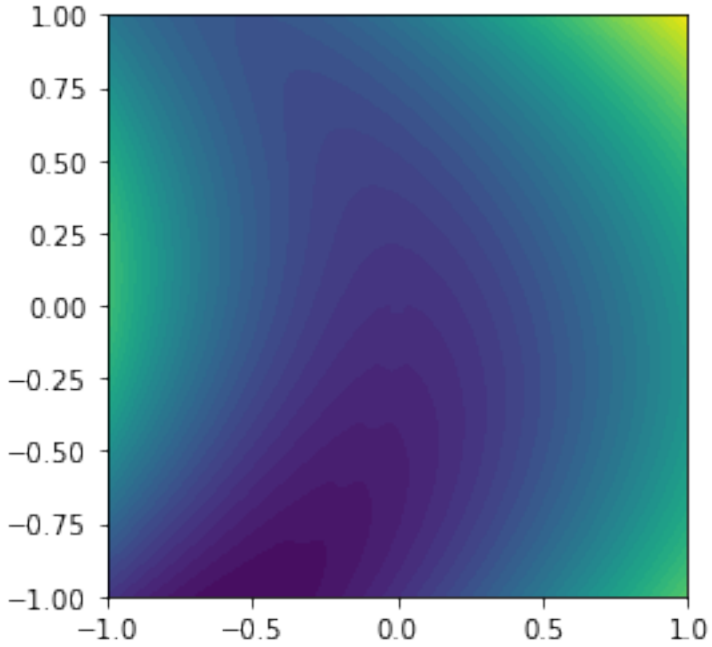

Figure 6.8: Output: FEniCS generated Least-Squares-solution to equation (6.2)

[41]:
```
plot(u2)
plt.show()
```

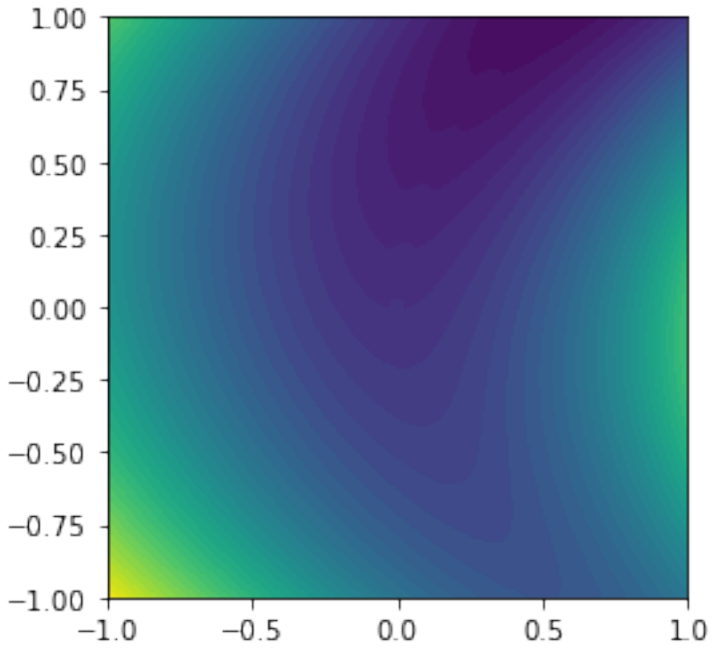

Figure 6.9: Output: FEniCS generated Least-Squares-solution to equation (6.3)

To make use of the output, it has to be defined where the individual solutions are also a solution to equation (6.1). By taking the partial derivative in y-direction and applying the `ReLu` function to the solution of equation (6.2), one obtains the subdomain where

the solution is also a solution to equation (6.1). For the solution to equation (6.3) the procedure works analogously.

[42]:
```
length = 20
S1 = toArray(u1, length, -1, 1)
S2 = toArray(u2, length, -1, 1)

S1_y = -numpy.gradient(S1)[0]
S2_y = -numpy.gradient(S2)[0]

plt.figure(dpi=120)
color_map = plt.imshow(ReLU(S1_y),  extent=[-1, 1, -1, 1])
plt.colorbar()
plt.show()
```

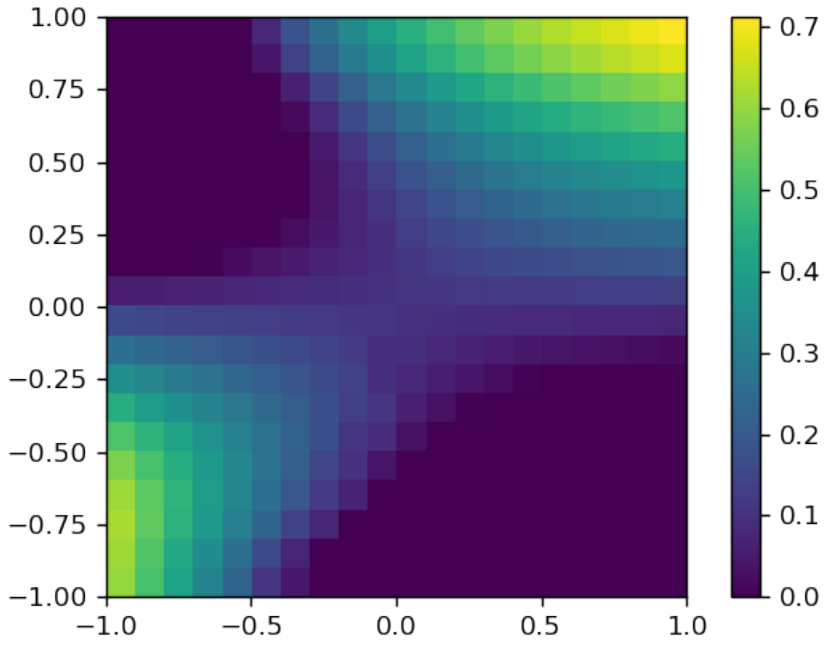

Figure 6.10: Output: Modification of the solution to equation (6.2): $\texttt{ReLU}(\frac{\partial S}{\partial y})$

[43]:
```
plt.figure(dpi=120)
color_map = plt.imshow(ReLU(-S2_y),  extent=[-1, 1, -1, 1])
plt.colorbar()
plt.show()
```

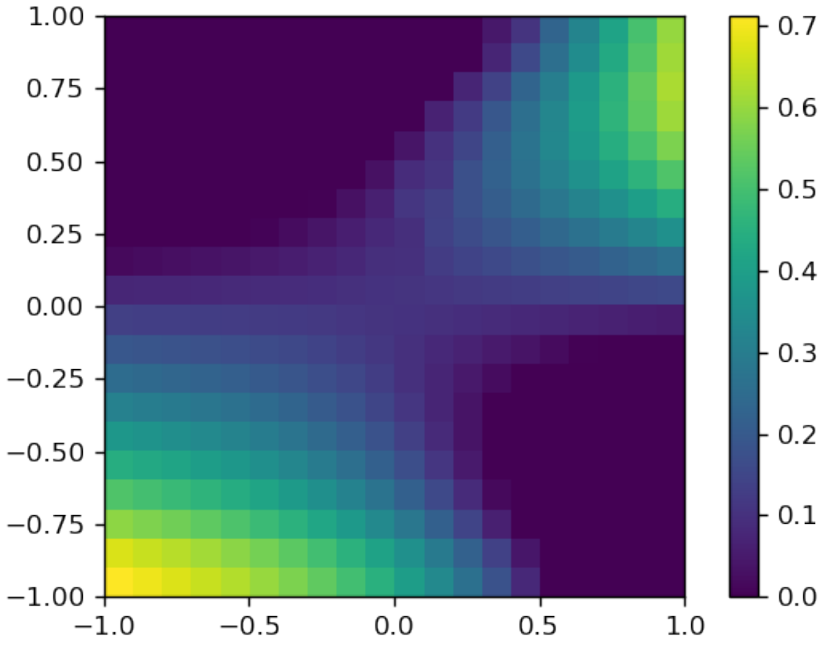

Figure 6.11: Output: Modification of the solution to equation (6.3): $\texttt{ReLU}(-\frac{\partial S}{\partial y})$

Unfortunately, the solutions are not defined everywhere in the domain and are often contradictory. The control law is dependent on the partial derivatives of the solution:

$$u = \texttt{heav}(\frac{\partial S}{\partial y}) \tag{6.4}$$

By obeying each individual solution and overlaying the resulting strategies, the following incomplete strategy is revealed:

[44]:
```
u = -(heav(S1_y)-heav(-S2_y))
u = numpy.flipud(u)


fig, ax = plt.subplots(dpi=120)
im = ax.imshow(u, cmap=cm.RdYlGn,
               origin='lower', extent=[-1, 1, -1, 1],
               vmax=abs(u).max(), vmin=-abs(u).max())
cbar = plt.colorbar(im)
cbar.set_label('$u(x_0,x_1)$')
ax.set_ylabel('velocity $x_1$')
ax.set_xlabel('location $x_0$')
plt.show()
```

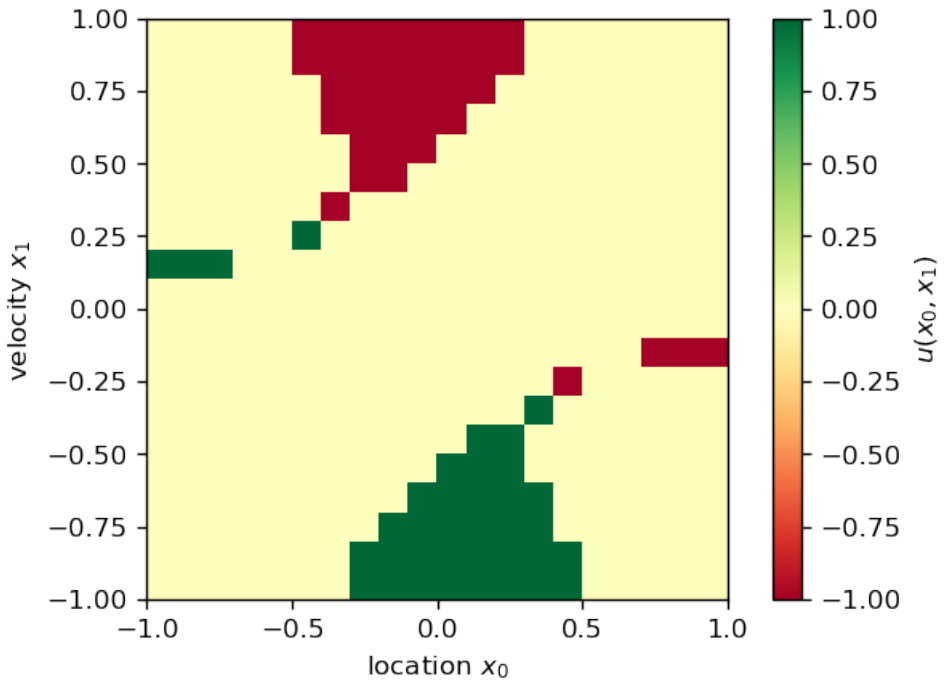

Figure 6.12: Output: Incomplete strategy to satisfy equation (6.1)

As can be seen, the solutions are correct in the regions where they are not contradictory. It is nevertheless not a valid strategy as the dynamics of the problem shows:

[45]:
```
x0_ = numpy.linspace(a, b, length)
x1_ = numpy.linspace(a, b, length)

U = u

x0_dot = numpy.zeros((length, length))
x1_dot = numpy.zeros((length, length))
for i in range(length):
    for j in range(length):
        x0_dot[i,j] = x1_[i]
        x1_dot[i,j] = U[i,j]

ig, ax = plt.subplots(dpi=120)
im = ax.imshow(u, cmap=cm.RdYlGn,
               origin='lower', extent=[-1, 1, -1, 1],
               vmax=abs(u).max(), vmin=-abs(u).max())

ax.quiver(x0_, x1_, x0_dot, x1_dot)
ax.set_aspect('equal', 'box')
ax.set_ylabel('velocity $x_1$')
ax.set_xlabel('location $x_0$')

cbar = plt.colorbar(im)
cbar.set_label('$u(x_0,x_1)$')
plt.show()
```

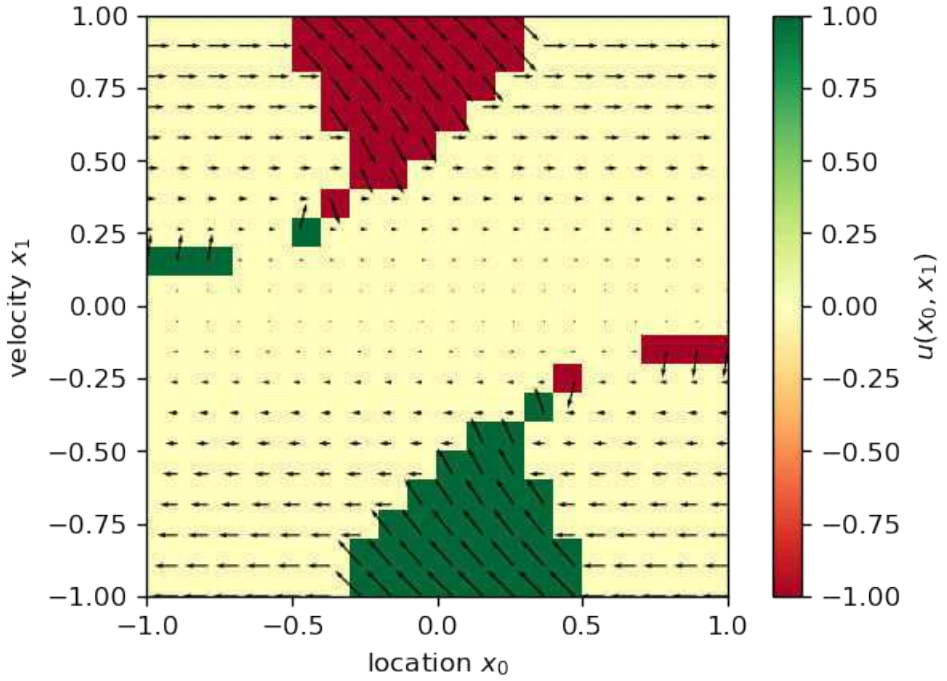

Figure 6.13: Output: Incomplete strategy to satisfy equation (6.1) including the dynamics-vectorfield.

Chapter 7

A new Approach

7.1 Convergence and Convexity

As was shown in the last Chapters the linearization efforts of the HJB turned out to be a dead end. The HJB is nonlinear and has to be treated as such. The Finite Element Method for nonlinear problems requires the variational form to be convex in order to apply methods such as the Newton-Raphson-Algorithm. The requirement is due to the fact that there has to be mathematically accessible information of how to iteratively reduce the residual of the approximation and to approach a numerical solution. In order to achieve the convexity for problems that are nonlinear but not convex such as the piecewise linear HJB, the so called Macaulay Bracket can be applied. This method is common in continuum mechanics and is used to consider geometric constraints such as Hertzian contact and also to model plasticity. These phenomena have one thing in common with the box constraints encountered in Optimal Control Theory: They constitute a special case called Karush-Kuhn-Tucker-Conditions (KKT). These conditions constrain a certain quantity to a minimum and maximum value and have no consequence to the functional when the value of the quantity is somewhere in between these values. The Macaulay Bracket elegantly applies these constraints while maintaining convexity. In modelling plasticity the stress is the constrained to a maximum value whereas in Optimal Control the control vector is constrained. The author ran several FE-Simulations in FEniCS using the described method for simple test problems such as in Chapter 6. The outcome was however disillusioning as it became clear that the boundary conditions delivered by the Optimal Control Problem itself are not sufficient to generate a numerical solution. For the algorithm to converge towards a solution the input to the problem has to be provided along with the boundary value on the whole mathematical boundary of the state space. These boundary values are usually unknown and to find them the canonical equations of the problem would have to be solved for every point on the boundary, which, needless to say, results in extraordinary effort and is therefore not feasible. Note that it is, however, possible to solve the canonical equations with the Macaulay bracket. In fact, by comparing solutions, it seems as if DIDO does the very same thing: Using the Macaulay bracket. It is informative to mention that the boundary conditions in the experiment were found by finding an analytical solution beforehand. Moreover it seemed that even preparing the problem already with the required boundary con-

ditions did not guarantee the convergence of the algorithm. An initial guess of the function had to be applied which was not too close from the analytical solution. Initializing the function with all zero entries (which is usual in FEM) led to convergence issues and often returned the zero-function. Hence it is clear that also by treating the HJB nonlinear as it is, the Finite Element Method will not reliably find solutions. In the following section an alternative to solving the HJB directly is presented, that highly rests upon the theory of the Linear Quadratic Regulator.

7.2 The Linear Quadratic Regulator revisited

It is imperative to understand that the procedure of finding a Linear Quadratic Regulator actually solves the corresponding HJB. It is however unusual to introduce the method by using the HJB as it is easier and more straightforward by deriving it by application of the Euler-Lagrange-Equations or the canonical equations. A derivation using the HJB can however by found in [Nai02] which is also the source for the following derivation. Consider the following Optimal Control Problem:

$$\int_0^\infty \frac{1}{2}\,\underline{u}^\top \underline{\underline{R}}\,\underline{u} + \frac{1}{2}\,\underline{x}^\top \underline{\underline{Q}}\,\underline{x} + \underline{p}^\top(\underline{\underline{A}}\,\underline{x} + \underline{\underline{B}}\,\underline{u} - \dot{\underline{x}})\;\; dt \tag{7.1}$$

Therefore the Control Hamiltonian becomes:

$$H := \frac{1}{2}\underline{u}^\top \underline{\underline{R}}\,\underline{u} + \frac{1}{2}\,\underline{x}^\top \underline{\underline{Q}}\,\underline{x} + \underline{p}^\top(\underline{\underline{A}}\,\underline{x} + \underline{\underline{B}}\,\underline{u}) \tag{7.2}$$

Note that the Problem is convex in $\underline{u}$ if $\underline{\underline{R}}$ is a real, symmetric, positive definite matrix [Nai02]. Therefore, by choosing $\underline{\underline{R}}$ as such, minimization of the Hamiltonian is performed by computing the gradient with respect to $\underline{u}$:

$$\frac{\partial H}{\partial \underline{u}} := \underline{\underline{R}}\,\underline{u} + \underline{\underline{B}}^\top \underline{p} = 0 \quad \Rightarrow \quad \underline{u}^* = -\underline{\underline{R}}^{-1}\underline{\underline{B}}^\top \underline{p} \tag{7.3}$$

Substitution into the control Hamiltonian gives the lower control Hamiltonian:

$$\mathscr{H} := \frac{1}{2}(\underline{\underline{R}}^{-1}\underline{\underline{B}}^\top \underline{p})^\top \underline{\underline{R}}\,\underline{\underline{R}}^{-1}\underline{\underline{B}}^\top \underline{p} + \frac{1}{2}\,\underline{x}^\top \underline{\underline{Q}}\,\underline{x} + \underline{p}^\top(\underline{\underline{A}}\,\underline{x} - \underline{\underline{B}}\,\underline{\underline{R}}^{-1}\underline{\underline{B}}^\top \underline{p}) \tag{7.4}$$

$$\mathscr{H} := \frac{1}{2}\underline{p}^\top \underline{\underline{B}}\,\underline{\underline{R}}^{-1}\underline{\underline{B}}^\top \underline{p} + \frac{1}{2}\,\underline{x}^\top \underline{\underline{Q}}\,\underline{x} + \underline{p}^\top \underline{\underline{A}}\,\underline{x} - \underline{p}^\top \underline{\underline{B}}\,\underline{\underline{R}}^{-1}\underline{\underline{B}}^\top \underline{p} \tag{7.5}$$

$$\mathscr{H} := -\frac{1}{2}\underline{p}^\top \underline{\underline{B}}\,\underline{\underline{R}}^{-1}\underline{\underline{B}}^\top \underline{p} + \frac{1}{2}\,\underline{x}^\top \underline{\underline{Q}}\,\underline{x} + \underline{p}^\top \underline{\underline{A}}\,\underline{x} \tag{7.6}$$

The ingenious step in deriving the algebraic Riccati equation (ARE) is "guessing" the solution to the Lagrange multiplier $\underline{p}$ to be of the following linear form, depending only on $\underline{x}$:

$$\underline{p} = \underline{\underline{P}}\,\underline{x} \tag{7.7}$$

Where $\underline{\underline{P}}$ is a constant symmetric matrix. After substitution, the following HJB has to be satisfied:

$$-\frac{1}{2}\underline{x}^\top \underline{\underline{P}}^\top \underline{\underline{B}}\,\underline{\underline{R}}^{-1}\underline{\underline{B}}^\top \underline{\underline{P}}\,\underline{x} + \frac{1}{2}\,\underline{x}^\top \underline{\underline{Q}}\,\underline{x} + \underline{x}^\top \underline{\underline{P}}^\top \underline{\underline{A}}\,\underline{x} = 0 \tag{7.8}$$

Note that, as every occurring matrix expression has to be multiplied by $\underline{x}$ from both sides, one can argue, that if the condition has to be satisfied for every $\underline{x}$, just the following quadratic matrix equation has to be satisfied by $\underline{\underline{P}}$:

$$-\frac{1}{2}\underline{\underline{P}}^\top \underline{\underline{B}}\,\underline{\underline{R}}^{-1}\underline{\underline{B}}^\top \underline{\underline{P}} + \frac{1}{2}\,\underline{\underline{Q}} + \underline{\underline{P}}^\top \underline{\underline{A}} = \underline{\underline{0}} \tag{7.9}$$

As $\underline{\underline{P}}$ is a symmetric matrix the above equation can multiplied by two and then be rearranged to fit the standard way it is presented in the literature:

$$\underline{\underline{P}}\,\underline{\underline{A}} + \underline{\underline{A}}^\top \underline{\underline{P}} - \underline{\underline{P}}^\top \underline{\underline{B}}\,\underline{\underline{R}}^{-1}\underline{\underline{B}}^\top \underline{\underline{P}} + \underline{\underline{Q}} = \underline{\underline{0}} \tag{7.10}$$

This is called the Algebraic- or Matrix-Riccati-Equation and it can in many cases be solved conveniently by e.g. the Newton-Raphson-Method. Note that the equation is convex in the unknown $\underline{\underline{P}}$ which makes the algorithm convergent and that there are as many equations as unknowns with allows unambiguous solutions. The method is due to its reliability and elegance widespread in control engineering. The only drawback is of course the simple linear form the dynamic equations have to satisfy. It is nevertheless clear that all kinds of information, also information of nonlinear dynamics, could in principle be stored in matrix form. Of course, to hold the information, the matrices to store that kind of information must be significantly bigger than usual linear dynamical systems. In data science it is common to replace a complicated dynamical system with few degrees of freedom with a linear system with an enormous amount of variables that somehow describe the state (for example a neural network; it is actually just a linear dynamical system). The problem with these practices however is, that they are for one extremely calculation-intensive and, more importantly, there is no mathematical guarantee whatsoever, that the obtained system will always be a proper description of the actual dynamics. In the following, a reliable and convenient way to produce these kind of matrices is presented, that relies not on data but on profound mathematical principles. This method will but be calculation-intensive as well.

7.3 The Liouville Equation

As mentioned throughout the previous chapters, an arbitrary dynamical system can be very well interpreted as a vector field. Now consider a particle moving through a state space, with its motion being guided and entirely determined by this vector field. One can easily add further particles to this abstraction that move independently from each other but still are governed by the same dynamics. Extending this idea further and further one ends up with a density distribution over the whole valid region of the state space, that changes continuously and, at each location, represents the probability to encounter a particle at a given time. Of course, as we are dealing now with a multidimensional scalar function and not with a vector, the equations that deliver the motion of the probability-density-function must now be of PDE form. This equation is called *Liouville-Equation*:

$$\dot{\rho} = \nabla \cdot (\rho \underline{f}(\underline{x}, \underline{u})) \tag{7.11}$$

As can be seen the Liouville Equation is a multidimensional linear PDE regardless of the dynamical system. This allows the application of the linear Finite Element Method and more importantly the application of the previously presented FE-Software. After setting the control vector to zero, a Galerkin Method could be applied:

$$\dot{\rho} = \nabla \cdot (\rho \underline{f}(\underline{x}, \underline{0})) \quad \longrightarrow \quad \text{FEA} \quad \longrightarrow \quad \underline{\dot{X}} = \underline{\underline{\hat{A}}}\, \underline{X} \tag{7.12}$$

The application of such a method would then return a linear dynamical system where the probability-density-function would be described by the vector X.

$$\rho(\underline{x}) \approx \sum_i X_i h_i(\underline{x}) \tag{7.13}$$

Of course this vector can store any distribution and therefore also describe a single discrete state. Theoretically the function is then required to be a Dirac distribution. Of course the whole idea describes a bigger problem than was intended here to describe in the first place, but the information we seek is still contained in this system and can therefore be used. In fact we encounter another special case of control Problem namely *PDE constrained Optimal Control*[1]. In some cases, a LQR can be used to control such systems. To do so, the PDE is of course required to be linear. This is, however, not the only requirement as also the way the control vector acts upon such a system has to fit in the LQR formalism. An example where this could be applied would be active damping of an oscillating structure. To apply the LQR formalism to the linearized system we require the initial dynamical system to be linear with respect to the control vector, which, as was described in Chapter 5, can be effectively achieved for any problem in engineering. This is however necessary for the further treatment of

[1] See for example [Trö05] for a detailed introduction.

the problem as will be shown soon. First consider how an arbitrary dynamical system with one control variable u could be treated by the linear Finite Element Method:

$$\dot{\rho} = \nabla \cdot (\rho \underline{f}(\underline{x}, u)) = \nabla \cdot (\rho \underline{a}(\underline{x}) + \rho \underline{b}(\underline{x}) u) \tag{7.14}$$

$$\dot{\underline{X}} = \hat{\underline{\underline{A}}}\, \underline{X} + \hat{\underline{\underline{B}}}\, \underline{X}\, u \tag{7.15}$$

The two computed stiffness matrices shall be named $\hat{\underline{\underline{A}}}$ and $\hat{\underline{\underline{B}}}$ as for the usual linearized systems. Of course we do not have the very same case here as the controlled term is still dependent on the state $\underline{X}$. To overcome this we apply another profound change to the problem by also allowing the control to be a continuous function that can obtain any real value for every location in the state space.

$$\dot{\underline{X}} = \hat{\underline{\underline{A}}}\, \underline{X} + \hat{\underline{\underline{B}}}\, \underline{U} \tag{7.16}$$

This is of course not possible for the actual control that can obtain one real value at each instant. It is nevertheless reasonable that if we allow the state to be non-discrete but continuous we also allow the control to act on every possible state in every possible way at each instant. We continue by formulating the following Optimal Control Problem that fits into the scheme of the Linear Quadratic Regulator and acknowledge that the strategy that is obtained by solving the corresponding ARE actually tries to solve a different problem than we wanted to solve in the first place:

$$\int_0^\infty \frac{1}{2}\, \underline{U}^\top \hat{\underline{\underline{R}}}\, \underline{U} + \frac{1}{2}\, \underline{X}^\top \hat{\underline{\underline{Q}}}\, \underline{X} + \underline{p}^\top (\hat{\underline{\underline{A}}}\, \underline{X} + \hat{\underline{\underline{B}}}\, \underline{U} - \dot{\underline{X}})\ dt \tag{7.17}$$

Note that the matrices $\hat{\underline{\underline{Q}}}$ and $\hat{\underline{\underline{R}}}$ can be chosen in order to resemble cost measures analogously to the linearized case as shall be shown soon. To find out how the solution of this problem is related to the solution of the problem we wanted to solve in the first place, we formulate this reference problem analytically as regular PDE constrained optimal control problem:

$$\int_0^\infty \int_\Omega \frac{1}{2} r(\underline{x}) u^2 \rho + \frac{1}{4} q(\underline{x}) \rho^2 + \lambda (\nabla \cdot (\rho \underline{f}(\underline{x}, u)) - \dot{\rho})\ d\Omega\, dt \tag{7.18}$$

Where $r(\underline{x})$ and $q(\underline{x})$ are given functions and $\rho(\underline{x}), u(\underline{x})$ and $\lambda(\underline{x})$ are the searched functions of the state space Ω. Note that an integral form was chosen; the volume integral over Ω is actually a weak requirement due to later numerical treatment. In general one could formulate the functional just by integration over time, which is a stronger requirement. Now we concentrate on just one term of the functional and apply integration by parts:

$$\int_\Omega \lambda \nabla \cdot (\rho \underline{f}) \; d\Omega = \int_{\partial\Omega} \lambda (\rho \underline{f}) \cdot \underline{n} \; d\Omega - \int_\Omega \nabla\lambda \cdot (\rho \underline{f}) \; d\Omega \tag{7.19}$$

It can be arranged later that the boundary term of the integral is always zero by requiring that $\underline{f}(\underline{x}, u)$ is the zero vector on the boundary. This can be achieved by prescribing appropriate values for u at the boundary. Even if it wouldn't be necessary to do so it is nevertheless reasonable, as a mechatronical system shouldn't move further away anyway if it reaches a state that is on the boundary of its operating space.

$$\int_0^\infty \int_\Omega \frac{1}{2} r(\underline{x}) u^2 \rho + \frac{1}{4} q(\underline{x}) \rho^2 + \nabla\lambda \cdot \rho \underline{f}(\underline{x}, u) - \lambda \dot{\rho} \; d\Omega \, dt \tag{7.20}$$

To derive the necessary conditions for the functional to become a minimum the Euler-Lagrange Equation can be used. Applying it with respect to ρ reveals:

$$\frac{1}{2} r(\underline{x}) u^2 + \frac{1}{2} q(\underline{x}) \rho + \nabla\lambda \cdot \underline{f} - \dot{\lambda} = 0 \tag{7.21}$$

As the time integral of the former functional goes to infinity, we can set $\dot{\lambda}$ to zero. This was shown by Kalman [Nai02]. The same assumption is also applied to derive the algebraic Riccati equation. Now consider what happens if the probability to encounter a particle is one at every location in state space, an therefore $\rho(\underline{x}) := 1$. It seems logical that the control will at every location do its best to bring the whole distribution back to a desired state, but the effect has even more properties that are related to the initial problem. Substitution reveals:

$$\frac{1}{2} r(\underline{x}) u^2 + \frac{1}{2} q(\underline{x}) + \nabla\lambda \cdot \underline{f} = 0 \tag{7.22}$$

Now one can choose the functions that were not yet defined to be:

$$r(\underline{x}) := R = const. \tag{7.23}$$

$$q(\underline{x}) := \underline{x}^\top \underline{\underline{Q}} \, \underline{x} \tag{7.24}$$

By substituting these functions one obtains:

$$\frac{1}{2} u^\top R \, u + \frac{1}{2} \, \underline{x}^\top \underline{\underline{Q}} \, \underline{x} + \nabla\lambda \cdot \underline{f} = 0 \tag{7.25}$$

This equation is basically the HJB of the corresponding ODE constrained optimal control problem. This means that by minimizing this Hamiltonian with respect to

the control u, we get analogous analytical expressions for the optimality conditions of the reference PDE problem as for the initial problem. Therefore the solution to the reference problem is similar if and only if $\rho(\underline{x}) := 1$. The next step is to access the information generated by solving the corresponding ARE. To do so, we first have to consider that we allowed the control to be explicitly independent of the state $\underline{X}$. We now have to fix this as it is not true for the dynamical system that was obtained by the finite element method. Therefore we require the following statement to be true:

$$\underline{U} = \underline{\underline{\hat{K}}}\, \underline{X} = \underline{X}\, u \tag{7.26}$$

Now imagine what happens if the probability to encounter a particle is 1 at every location in the state space:

$$\rho(\underline{x}) := 1 \longrightarrow \underline{X} = \underline{1} = (1, 1, 1, 1, 1, 1, 1, ...)^\top \tag{7.27}$$

$$\underline{\underline{\hat{K}}}\, \underline{1} = \underline{1}\, u \tag{7.28}$$

Now the control function will at every location in state space return the feedback law that is needed to bring the density function back to a desired state. But u is in fact a function of the state-space as well. We encounter a situation where a continuous function that lives in a state space is somehow linked to a finite element discretisation of the same space.

$$\underline{\underline{\hat{K}}}\, \underline{1} = \underline{1}\, u(\underline{x}) \tag{7.29}$$

In order to find $u(\underline{x})$ the information in the equation has to be accessed as usual in discretized systems. Consider an arbitrary discrete state $\underline{x}^*$ represented by a continuous function:

$$\rho^*(\underline{x}) := \delta(\underline{x} - \underline{x}^*) \approx \sum_i X_i^* h_i(\underline{x}) \tag{7.30}$$

$$\underline{X}^* = (0, 0, ..., 0, 1, 0, 0, 0, 0, ...)^\top \tag{7.31}$$

To find the value of u at the location $\underline{x}^*$ the following equation is suggested:

$$\underline{X}^{*\top}\, \underline{\underline{\hat{K}}}\, \underline{1} = \underline{X}^{*\top} \underline{1}\, u(\underline{x}^*) \tag{7.32}$$

$$\boxed{u(\underline{x}^*) = \underline{X}^{*\top}\, \underline{\underline{\hat{K}}}\, \underline{1}} \tag{7.33}$$

7.4 Experiment

In the following, the above method is tested for the simple LQR example from section 3.3. First, all necessary numeric packages are imported. In addition, the sign function as well as a simple program to solve the LQR problem are defined:

[1]:
```
from IPython.display import display
import matplotlib.pyplot as plt
import numpy
import sympy
import scipy.linalg
numpy.set_printoptions(precision=2)
numpy.set_printoptions(suppress=True)


def lqr(A,B,Q,R):

    P = numpy.matrix(scipy.linalg.solve_continuous_are(A, B, Q, R))

    K = numpy.matrix(scipy.linalg.inv(R)*(B.T*P))

    eigVals, eigVecs = scipy.linalg.eig(A-B*K)

    return K, P, eigVals, eigVecs


def sgn(array):
    for indices in numpy.ndindex(array.shape):
        if array[indices] < -0.0001:
            array[indices] = -1.0
        elif array[indices] > 0.0001:
            array[indices] = 1.0
        else:
            array[indices] = 0.0
            pass
    return array
```

Next, all key classes and methods of *femco*, which stands for "Finite Element Method enhanced Control", are imported. Most of the functions of this package are similar to the functions presented in Chapter 5. The full source code can be found in Appendix C. First the domain and resolution of the mesh are defined. To find the variational forms of the Liouville equation, the operator equation is simply multiplied by a test function. Therefore just a simple Weighted Residual Method is used, namely Galerkin's Method, as the weight functions are the same as the interpolation functions. This process of obtaining the variational form from the dynamics can later be automated just like for the former program. For the sake of simplicity it is done here 'by hand':

[2]:
```
from femco import *

""" input skript """

# define mesh
domain = [(-1,1),(-1,1)]
resolution = 0.1
```

```
# define weak formulation
dimension = len(domain)    # dimension of the state space
v = sympy.Symbol('v')      # test function
X = numpy.asarray(sympy.symbols('x:' + str(dimension)))
S = sympy.Symbol('S')
V_x = numpy.asarray(sympy.symbols('v_x:' + str(dimension)))
S_x = numpy.asarray(sympy.symbols('S_x:' + str(dimension)))


Au_v = (S_x[0]*X[1] - S_x[1]*X[1] - S)*v

print("weak form A matrix")
print("-----------------------------------------------------")
display(Au_v)
```

```
weak form A matrix
-----------------------------------------------------
```

$$v\left(-S + S_{x0}x_1 - S_{x1}x_1\right)$$

[3]:
```
Bu_v = (S_x[1])*v

print("weak form B matrix")
print("-----------------------------------------------------")
display(Bu_v)
```

```
weak form B matrix
-----------------------------------------------------
```

$$S_{x1}v$$

Next the element stiffness matrices are computed and then assembled into A and B Matrix respectively. To get a better understanding the matrices are visualized:

[4]:
```
m = mesh(domain, resolution) # initialize mesh object
k = m.ESM_weak(Au_v)         # solve weak form in defined mesh
A = m.assemble(k)

print("A matrix")
print("-----------------------------------------------------")
print(A)

# plot
fig, ax = plt.subplots(dpi=120)
im = ax.imshow(A,extent=[-1, 1, -1, 1],vmax=A.max(), vmin=A.min())
cbar = plt.colorbar(im)
cbar.set_label('S(x,y)')
ax.set_ylabel('y')
ax.set_xlabel('x')
plt.show()
```

```
A matrix
--------------------------------------------------
[[-0.   -0.01  0.   ...  0.    0.    0.  ]
 [ 0.02  0.03 -0.01 ...  0.    0.    0.  ]
 [ 0.    0.02  0.03 ...  0.    0.    0.  ]
 ...
 [ 0.    0.    0.   ...  0.03  0.02  0.  ]
 [ 0.    0.    0.   ... -0.01  0.03  0.02]
 [ 0.    0.    0.   ...  0.   -0.01 -0.  ]]
```

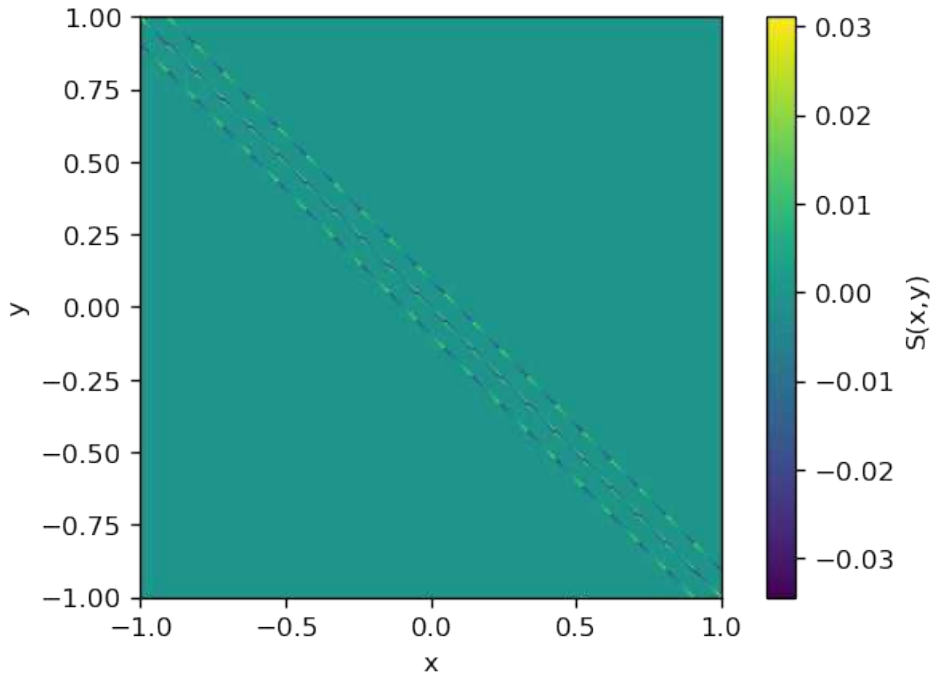

Figure 7.1: Output: $\underline{\underline{\hat{A}}}$-matrix. The color of every pixel corresponds to the numerical value of the entries of the matrix. Note also the band-structure which is due to the efficient numbering of the nodes in the mesh.

[5]:
```
m = mesh(domain, resolution) # initialize mesh object
k = m.ESM_weak(Bu_v)         # solve weak form in defined mesh
B = m.assemble(k)

print("B matrix")
print("--------------------------------------------------")
print(B)

# plot
fig, ax = plt.subplots(dpi=120)
im = ax.imshow(B,vmax=B.max(), vmin=B.min())
cbar = plt.colorbar(im)
plt.show()
```

```
B matrix
--------------------------------------------------
[[-0.02 -0.02  0.    ...  0.    0.    0.  ]
 [ 0.02  0.   -0.02 ...  0.    0.    0.  ]
 [ 0.    0.02  0.   ...  0.    0.    0.  ]
 ...
 [ 0.    0.    0.   ...  0.   -0.02  0.  ]
 [ 0.    0.    0.   ...  0.02  0.   -0.02]
 [ 0.    0.    0.   ...  0.    0.02  0.02]]
```

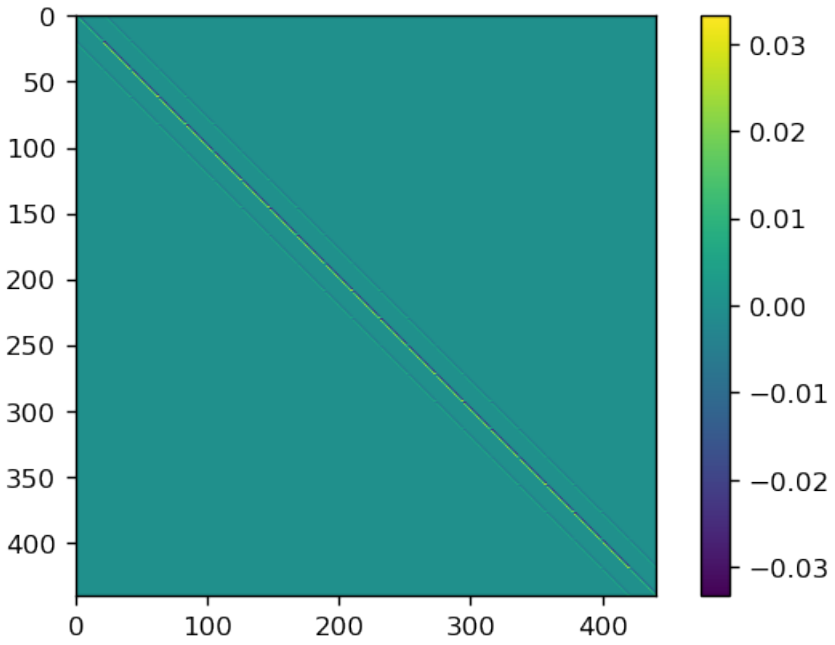

Figure 7.2: Output: $\underline{\underline{\hat{B}}}$-matrix. The color of every pixel corresponds to the numerical value of the entries of the matrix. Note also the band-structure which is due to the efficient numbering of the nodes in the mesh.

As can be seen the matrices are both sparse which will eventually result in less computation effort as for "dense" matrices. To get a better understanding of what these matrices actually do and how nonlinear dynamic behaviour can be predicted from it we continue by multiplying the distribution vector of a single discrete state with the $\underline{\underline{\hat{A}}}$-matrix. This will of result in the rate of change of the distribution. It is informative to compare figure 7.3 to figure 3.1:

[9]:
```
X = numpy.zeros((max(B.shape),1))

p = 155

X[p] = 1

X_dot =  numpy.dot(A,X)

dx_dt = numpy.reshape(X_dot,(int(numpy.sqrt(A.shape[0])),int(numpy.sqrt(A.
↪shape[0]))),order='C')
```

```
dx_dt = dx_dt.T
dx_dt = numpy.flipud(dx_dt)

fig, ax = plt.subplots(dpi=120)
im = ax.imshow(dx_dt, extent=[-1, 1, -1, 1],vmax= X_dot.max(), vmin=X_dot.min())
cbar = plt.colorbar(im)
cbar.set_label('rho(x,y)')
ax.set_ylabel('y')
ax.set_xlabel('x')
plt.show()
```

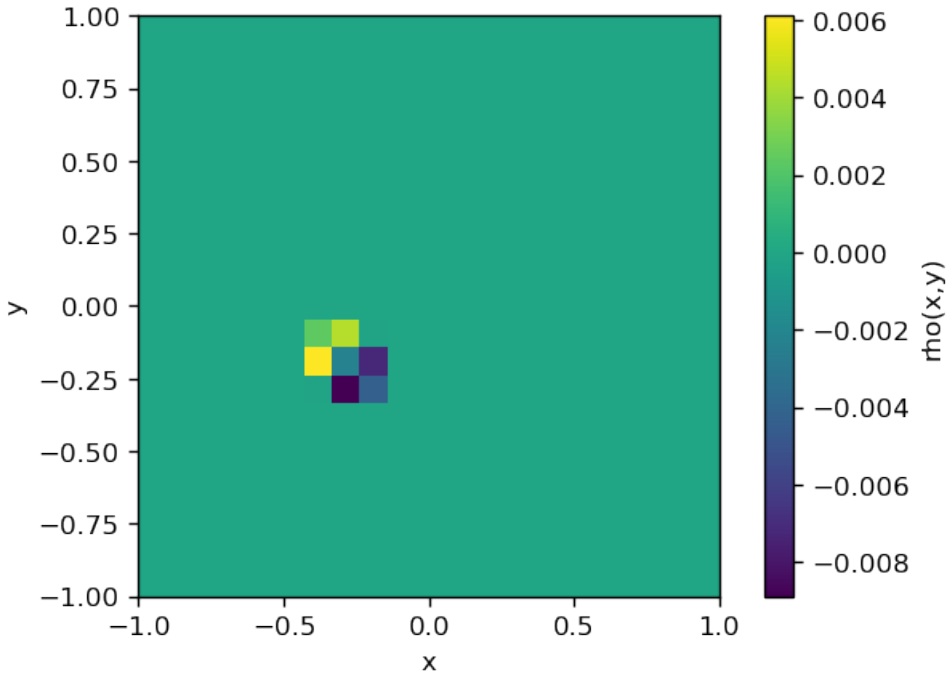

Figure 7.3: Output: Change in the probability-density-function for a Dirac-distribution. Obviously the distribution is drawn to the left upper center of the state space (as prescribed by the dynamics-vectorfield).

Next the $\hat{\underline{\underline{R}}}$ and $\hat{\underline{\underline{Q}}}$ matrices need to be defined. For $\hat{\underline{\underline{R}}}$ it is reasonable to use an identity matrix that is multiplied by a scaling factor that needs to be determined experimentally. The author suggests to use the volume of the state space divided by the number of nodes in the mesh as this leads to usable results for many different mesh resolutions. To find $\hat{\underline{\underline{Q}}}$ it is useful to define it as if it was just for the regular linearized systems and then use the function `Q_to_FEM_Q()`. This function basically computes the value of $\underline{x}^\top \mathbf{Q}\, \underline{x}$ for every node and writes the value to the corresponding diagonal entry of the $\hat{\underline{\underline{Q}}}$-matrix that will be used with the FEM linearization of the Liouville equation. Therefore one obtains a cost measure that is almost similar to the simple linear case. For more information the reader is referred to Appendix C.

[6]:
```
R = numpy.identity(B.shape[0])*(m.state_space_vol/m.number_nodes)
Q = numpy.array([[1, 0],
                 [0, 1]])
Q = m.Q_to_FEM_Q(Q)
```

Next, the so called "controllability" is checked for the obtained matrices. This is also a common practice when making use of LQR. The controllability would, for example, not be satisfied if the control variable has no direct or indirect influence on certain state variables.

[7]:
```
import control
from numpy.linalg import matrix_rank
print(matrix_rank(control.ctrb(A, B)) == A.shape[0])
```

```
True
```

As the system is obviously controllable, we conclude by applying `scypy's` functions to iteratively find the $\hat{\underline{\underline{K}}}$ matrix that contains all the informations on how to control the system:

[8]:
```
K, P, E, eigVecs = lqr(A, B, Q, R)
print(K)
```

```
[[ -8.63  10.19  -2.26 ...   0.     0.     0.  ]
 [ -9.94  -3.57   9.37 ...   0.     0.     0.  ]
 [ -4.79 -11.36  -2.92 ...  -0.    -0.    -0.  ]
 ...
 [  0.     0.     0.    ...   2.92  11.36   4.79]
 [ -0.    -0.    -0.    ...  -9.37   3.57   9.94]
 [ -0.    -0.    -0.    ...   2.26 -10.19   8.63]]
```

Now the final formula of the previous section will be applied to find a strategy:

[12]:
```
U = numpy.zeros((max(B.shape),1))

one = numpy.ones((max(B.shape),1))
for p in range(max(B.shape)):
    X = numpy.zeros((max(B.shape),1))
    X[p] = 1

    Uc = -numpy.dot(K,one)

    u = Uc.T@X
    U[p] = u

U = numpy.reshape(U,(int(numpy.sqrt(A.shape[0])),int(numpy.sqrt(A.
↪shape[0]))),order='C')

U = U.T
U = numpy.flipud(U)

ctrllaw = U
fig, ax = plt.subplots(dpi=120)
im = ax.imshow(ctrllaw, extent=[-1, 1, -1, 1],vmax= U.max(), vmin=U.min())
cbar = plt.colorbar(im)
cbar.set_label('rho(x,y)')
ax.set_ylabel('y')
ax.set_xlabel('x')
plt.show()
```

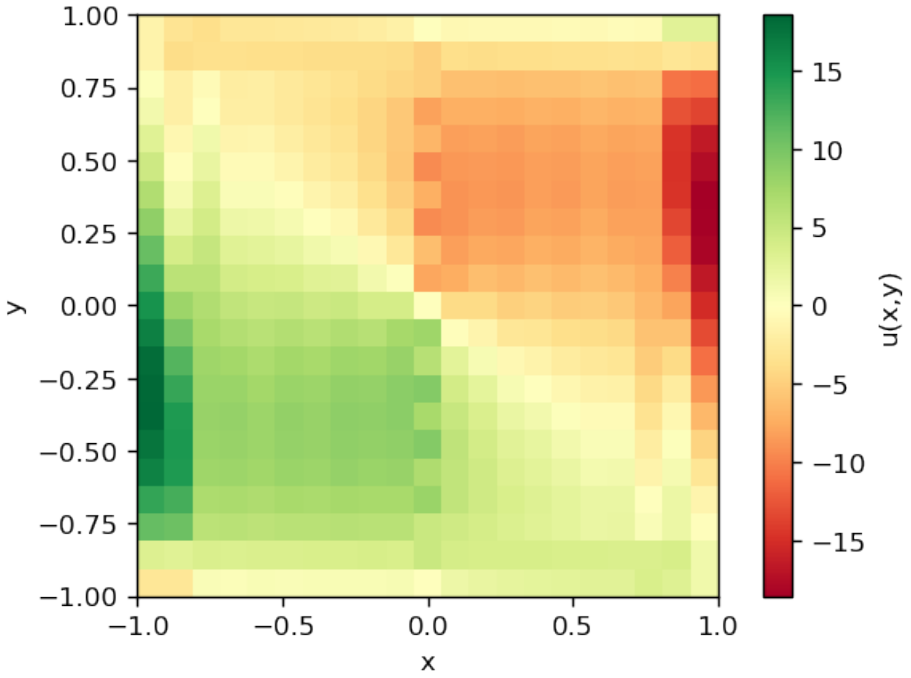

Figure 7.4: Output: Feedback control law or *strategy* generated by the previously described method for the problem from section 3.3. To increase the accuracy multiple FE-Methods and variational forms should be tested.

Compare the produced strategy to figure 3.2. The strategies are not exactly the same but they are obviously related. Especially at the boundary the here produced strategy seems to lack accuracy which is probably because u was not chosen in order for the boundary integral of the previous section to vanish. This could however be arranged easily with more sophisticated software-tools and would probably lead to more concise results. Another way this could be achieved is by finding a better variational form and better fitting FE-Method. This is, however, beyond the scope of this thesis as the aim was only to prove the principal possibility to solve the HJB by the Finite Element Method. For further investigation of the produced strategy, the sign function is applied:

[13]:
```
U = sgn(U)
ctrllaw = U
fig, ax = plt.subplots(dpi=120)
im = ax.imshow(ctrllaw, extent=[-1, 1, -1, 1],vmax= U.max(), vmin=U.min())
cbar = plt.colorbar(im)
cbar.set_label('rho(x,y)')
ax.set_ylabel('y')
ax.set_xlabel('x')
plt.show()
```

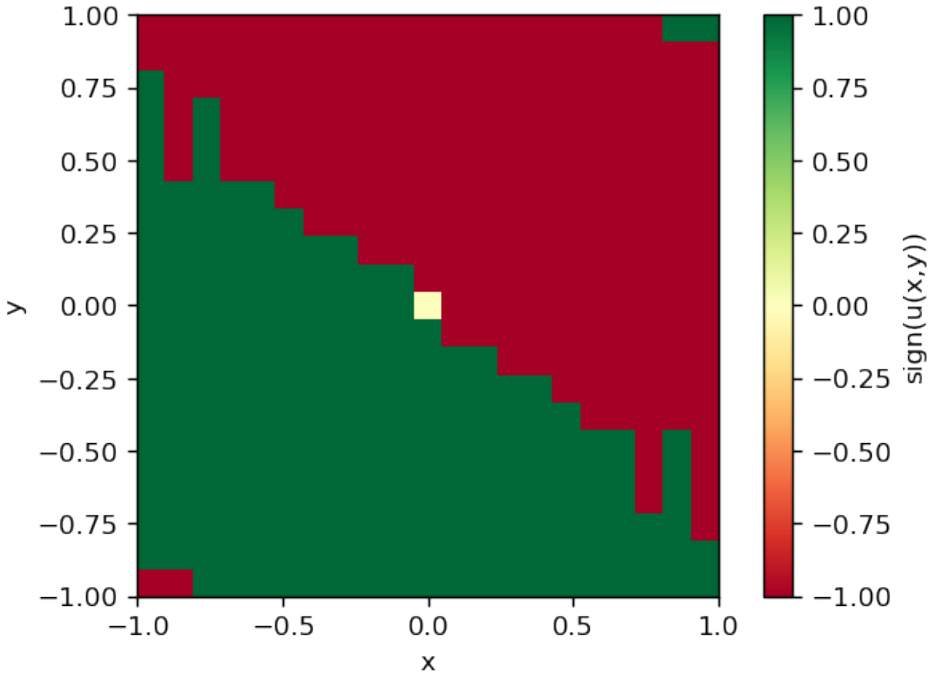

Figure 7.5: Output: Sign function applied to the feedback control law displayed in figure 7.4.

As can be seen the control effort becomes zero as soon as the state reaches its desired location which is necessary and expected. Note also that the strategy in figure 3.2 and the currently reviewed strategy (figure 7.5) both change the sign of the control-law at the same locations (with exclusion of the errors at the boundary of course).

Chapter 8

Conclusion

The author has, as part of this thesis, deepened his knowledge in variational calculus as well as in the Finite Element Method with the aim of deriving a versatile way of solving the Hamilton-Jacobi-Bellman Equation numerically to enable feedback control of nonlinear dynamical systems. To do so, a Finite Element software was developed that can, under certain restrictions regarding element type, be applied to PDEs of arbitrary dimension. The software developed to demonstrate this is, to the authors knowledge, the only FEM-Software that is capable of these calculations. Furthermore, a new method was derived, that is capable of solving feedback control problems of nonlinear dynamics. This was achieved by formulating the problem in such a way that it has a unique solution. There remains, however, a lot of research work to do in order to find a better variational formulation (or a more suitable FE-Method) in order to improve the generated feedback control laws. After this hurdle has been cleared there remains only one fundamental restriction factor: The number of state variables that can be taken into account. As the number of the degrees of freedom of the system of equations rises roughly exponentially with the number of state variables it is clear that the boundary of the physically manageable amount of information can soon be reached. The author is nevertheless convinced that the future of Control Theory is based on solving multidimensional boundary value problems and sees the development of the Finite Element software and the variational methods presented in this thesis as first step in the right direction.

Appendix A

Notation

If not defined differently, every vecor is a column vector:

$$\underline{a} = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} \quad \underline{b} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} \tag{A.1}$$

The standard vector dot-product will be used:

$$\underline{a} \cdot \underline{b} = \underline{a}^\top \underline{b} = a_1 b_1 + a_2 b_2 + a_3 b_3 \tag{A.2}$$

If a vector product isn't explicitly denoted with a dot or the *Transpose-Operator* then it shall be computed as *Hadamard Product*:

$$\underline{a}\ \underline{b} = \begin{bmatrix} a_1 b_1 \\ a_2 b_2 \\ a_3 b_3 \end{bmatrix} \tag{A.3}$$

The multiplication of column-vector and row-vector results in an two dimensional matrix:

$$\underline{a}\ \underline{b}^\top = \begin{bmatrix} a_1 b_1 & a_1 b_2 & a_1 b_3 \\ a_2 b_1 & a_2 b_2 & a_2 b_3 \\ a_3 b_1 & a_3 b_2 & a_3 b_3 \end{bmatrix} \tag{A.4}$$

Empowerisation is <u>not</u> executed as *Hadmard Product*:

$$\underline{a}^2 = a_1^2 + a_2^2 + a_3^2 = \underline{a}^\top \underline{a} \tag{A.5}$$

A *Nabla-Operator* with a vector as subscript denotes the gradient in the corresponding vector space:

$$\nabla_{\underline{a}} = \begin{bmatrix} \partial/\partial a_1 \\ \partial/\partial a_2 \\ \partial/\partial a_3 \end{bmatrix} \tag{A.6}$$

Hadamard-Product is computed before dot operations:

$$\nabla^2 u(\underline{x}) = \nabla \cdot \nabla u(\underline{x}) = \frac{\partial^2 u(\underline{x})}{\partial x_1^2} + \frac{\partial^2 u(\underline{x})}{\partial x_2^2} + \dots \tag{A.7}$$

Appendix B

ARTOC Toolbox

[1]:
```
"""
-------------------------------------------------------------------------
A.R.T.O.C. - Adaptive Real-Time Optimal Control
author: wolfgang flachberger (c)
em@il: wolfgang.flachberger@stud.unileoben.ac.at
08-03-2020
-------------------------------------------------------------------------
NOTATION:

vectors: (usually one dimensional numpy arrays of shape (n,) )
np.outer(x,x) = xx^(T)
np.dot(x,x) = x^(T)x
x*x = xx (hadamard)

matrices:
np.dot(M,M) = MM

the author recommends the use of column vectors of shape (n,1)
numpy.dot() computes for this strict notation not the actual
dot pruduct but the regular vector/matrix multiplication:
np.dot(x,x) = ERROR
np.dot(x.T,x) = x^(T)x
np.dot(x,x.T) = xx^(T)

use numpy.linalg.multidot((x.T,M,x)) to multiply multiple vectors/matrices.
-------------------------------------------------------------------------
"""

import numpy
import sympy
from sympy.functions import sign
from numpy.linalg import inv

class mesh:
    """creates mesh object
       arguments:
           domain: feasable domain of the state space: takes a list of n tuples
                   (a,b) that bound the domain in the corresponding dimension
                   (the domain is an n-dimensional box --> "box constraints")
           resolution: side length of the finite n-dimensional cube elements
    """
    def __init__(self, domain, resolution):

        self.domain = domain
        self.resolution = resolution

        """-----------------------------------------------------------------"""
        dimension = len(domain) # dimension of state space
        nodes_per_element = 2**dimension
        # calculate number of finite cube elements to cover domain
        nr_el_dim = [] # list with number of elements necessary per axis
        nr_nd_dim = [] # list with number of nodes necessary per axis
        for index in range(dimension): # for every dimension
            length = domain[index][1] - domain[index][0] # take length of "box"
            nr_el_dim.append(int(length/resolution)) # the domain is just
            # approximated --> the Jacobian matrix is therfore not necessary
            # because all elements are of the same size and shape
            nr_nd_dim.append(nr_el_dim[index] + 1)
            pass
        nr_el_dim = tuple(nr_el_dim)
        nr_nd_dim = tuple(nr_nd_dim)
        number_elements = numpy.prod(nr_el_dim) # total amount of elements
        number_nodes = numpy.prod(nr_nd_dim) # total amount of gloabal nodes
        nodes = numpy.arange(number_nodes,dtype=int)
        node_names_global = numpy.reshape(nodes,(nr_nd_dim),order='C')
        v = sympy.Symbol('v') # test funtion
        X = numpy.asarray(sympy.symbols('x:' + str(dimension)))
        XI = numpy.asarray(sympy.symbols('xi:' + str(dimension)))
        S_x = numpy.asarray(sympy.symbols('S_x:' + str(dimension)))
```

```
        V_x = numpy.asarray(sympy.symbols('v_x:' + str(dimension)))
        """-----------------------------------------------------------------"""

        self.dimension = dimension
        self.nodes_per_element = nodes_per_element
        self.number_elements = number_elements
        self.number_nodes = number_nodes
        self.nr_el_dim = nr_el_dim
        self.nr_nd_dim = nr_nd_dim
        self.node_names_global = node_names_global
        self.v = v
        self.X = X
        self.XI = XI
        self.S_x = S_x
        self.V_x = V_x

        T = self.coincidenceTable()
        self.T = T
        h = self.interpolationFunctions()
        self.h = h
        dh_dx = self.interpolationFunctionsDerivatives()
        self.dh_dx = dh_dx

        """-----------------------------------------------------------------"""

    def coincidenceTable(self):
        """ coincidence_table[element,node] """
        coincidence_table = numpy.zeros((self.number_elements,self.nodes_per_element), dtype=int, order='C')
        local_to_global = numpy.zeros((2,)*self.dimension)
        i = 0
        for element_indices in numpy.ndindex(self.nr_el_dim):
            for node_indices in numpy.ndindex((2,)*self.dimension):
                location = tuple([sum(x) for x in zip(element_indices,node_indices)])
                local_to_global[node_indices] = self.node_names_global[location]
                pass
            array = numpy.reshape(local_to_global, self.nodes_per_element)
            coincidence_table[i] = array
            i += 1
            pass
        return coincidence_table

    def interpolationFunctions(self):
        """ interpolationfunction[node] """
        i = 0
        h = sympy.ones(self.nodes_per_element,1)
        for element_indices in numpy.ndindex((2,)*self.dimension):
            for axis in range(self.dimension):
                if element_indices[axis] == 0:
                    factor = -1
                else:
                    factor = 1
                h[i] = h[i] * (1 + factor * self.XI[axis] * 2/self.resolution)
            i += 1
        h = h/self.nodes_per_element
        return h

    def interpolationFunctionsDerivatives(self):
        """ interpolationfunction[node, derivativeaxis] """
        dh_dX = sympy.ones(self.nodes_per_element, self.dimension)
        for index in range(self.nodes_per_element):
            for axis in range(self.dimension):
                dh_dX[index,axis] = sympy.diff(self.interpolationFunctions()[index], self.XI[axis])
        return dh_dX

        """-----------------------------------------------------------------"""

    def ESM_ELV_weak(self, Au_v, F_v):
        # evaluate componets of k_ij (not integrated yet)
        k_ij = sympy.zeros(self.nodes_per_element,self.nodes_per_element)
        for i in range(self.nodes_per_element):
            for j in range(self.nodes_per_element):
                sub = []
                for k in range(self.dimension):
                    sub.append((self.S_x[k], self.dh_dx[i,k]))
                    sub.append((self.V_x[k], self.dh_dx[j,k]))
                    sub.append((self.v, self.h[j]))
                k_ij[i,j] = Au_v.subs(sub)
        # prepare substitution for coordinate transformation
        x_to_xi = []
        for j in range(self.dimension):
            x_to_xi.append( (self.X[j], self.XI[j] + self.X[j]) )
        # transform to local coordinate frame and integrate
        for indices in numpy.ndindex(k_ij.shape):
            k_ij[indices] = k_ij[indices].subs(x_to_xi)
            for i in range(self.dimension):
                k_ij[indices] = sympy.integrate(k_ij[indices],(self.XI[i],-self.resolution/2,self.resolution/2))
        # evaluate componets of f_j (not integrated jet)
        f_j = sympy.zeros(self.nodes_per_element,1)
        for j in range(self.nodes_per_element):
            sub = []
            for k in range(self.dimension):
                sub.append((self.V_x[k], self.dh_dx[j,k]))
                sub.append((self.v, self.h[j]))
            f_j[j] = F_v.subs(sub)
        # prepare substitution for transformation
        x_to_xi = []
        for j in range(self.dimension):
            x_to_xi.append( (self.X[j], self.XI[j] + self.X[j]) )
        # transform and integrate
```

```
        for j in range(self.nodes_per_element):
            f_j[j] = f_j[j].subs(x_to_xi)
            for dim in range(self.dimension):
                f_j[j] = sympy.integrate(f_j[j],(self.XI[dim], -self.resolution/2, self.resolution/2))
        return k_ij, f_j

    def ESM_ELV_ritz(self, W ):
        # initialize ESM and ELV
        k_ij = sympy.zeros(self.nodes_per_element,self.nodes_per_element)
        f_j = sympy.zeros(self.nodes_per_element,1)
        # finite element approximation with "s" as design variables
        s = numpy.asarray(sympy.symbols('s:' + str(self.nodes_per_element)))
        p = sympy.zeros(self.dimension,1)
        for i in range(self.nodes_per_element):
            p += self.dh_dx.T[:,i]*s[i]
        # prepare substitution for coordinate transformation
        S_x_to_p = []
        for j in range(self.dimension):
            S_x_to_p.append( (self.S_x[j], p[j]) )
        W = W.subs(S_x_to_p)
        # THE FOLLOWING CODE WORKS ONLY IN 2 DIMENSIONS
        for i in range(self.nodes_per_element):
            f_j[i,0]  = sympy.diff(W,s[i]).subs([(s[0],0),(s[1],0),(s[2],0),(s[3],0)])
            k_ij[i,0] = sympy.diff(W,s[i]).subs([(s[0],1),(s[1],0),(s[2],0),(s[3],0)])-f_j[i,0]
            k_ij[i,1] = sympy.diff(W,s[i]).subs([(s[0],0),(s[1],1),(s[2],0),(s[3],0)])-f_j[i,0]
            k_ij[i,2] = sympy.diff(W,s[i]).subs([(s[0],0),(s[1],0),(s[2],1),(s[3],0)])-f_j[i,0]
            k_ij[i,3] = sympy.diff(W,s[i]).subs([(s[0],0),(s[1],0),(s[2],0),(s[3],1)])-f_j[i,0]
        f_j = - f_j

        # prepare substitution for coordinate transformation
        x_to_xi = []
        for j in range(self.dimension):
            x_to_xi.append( (self.X[j], self.XI[j] + self.X[j]) )

        # transform to local coordinate frame and integrate
        for indices in numpy.ndindex(k_ij.shape):
            k_ij[indices] = k_ij[indices].subs(x_to_xi)
            for i in range(self.dimension):
                k_ij[indices] = sympy.integrate(k_ij[indices],(self.XI[i],-self.resolution/2,self.resolution/2))

        # transform and integrate
        for j in range(self.nodes_per_element):
            f_j[j] = f_j[j].subs(x_to_xi)
            for dim in range(self.dimension):
                f_j[j] = sympy.integrate(f_j[j],(self.XI[dim], -self.resolution/2, self.resolution/2))

        return k_ij, f_j

    def assemble(self, k_ij, f_j):
        # initialize transport arrays
        k = numpy.zeros((self.nodes_per_element, self.nodes_per_element))
        f = numpy.zeros((self.nodes_per_element, 1))

        # initialize stiffnessmatrix and loadvector of the whole system
        K_ = numpy.zeros((self.number_nodes,self.number_nodes))
        F_ = numpy.zeros((self.number_nodes,1))

        el = 0 # element counter
        for e_index in numpy.ndindex(self.nr_el_dim): # for every element

            # prepare substitution for element
            evaluate = []
            e_location = numpy.asarray(e_index) * self.resolution + numpy.ones(self.dimension)*self.resolution/2
            for i in range(self.dimension):
                evaluate.append( (self.X[i], e_location[i] ))

            # evaluate k_ij and f_j for element
            for indices in numpy.ndindex((self.nodes_per_element, self.nodes_per_element)):
                k[indices] = k_ij[indices].subs(evaluate)

            for j in range(self.nodes_per_element):
                f[j,0] = f_j[j].subs(evaluate)

            # assemble K_ and F_
            for k_index in numpy.ndindex((self.nodes_per_element, self.nodes_per_element)):
                i = self.T[el,k_index[0]]
                j = self.T[el,k_index[1]]
                K_[i,j] += k[k_index] #round(k[k_index],1)

            for f_index in range(self.nodes_per_element):
                F_[self.T[el,f_index],0] += f[f_index,0] #round(f[f_index,0],1)

            el += 1 # next element

        return K_, F_


    def boundaryNodes(self, e):
        node_locations_global = numpy.zeros(self.nr_nd_dim + (self.dimension,))
        initialPoint = []
        for i in range(self.dimension):
            initialPoint.append(self.domain[i][0])
        initialPoint = numpy.asarray(initialPoint)
        for indices in numpy.ndindex(self.nr_nd_dim):
            step = numpy.asarray(list(indices))
            node_locations_global[indices] = initialPoint + step * self.resolution
        boundary_nodes = []
        boundary_node_indices = []
        length = len(e(node_locations_global[indices]))
```

```
        for indices in numpy.ndindex(self.nr_nd_dim):
            if e(node_locations_global[indices]) == list((0,)*length):
                boundary_nodes.append(self.node_names_global[indices])
                boundary_node_indices.append(indices)
        if boundary_nodes == []:
            print("ERROR: no boundary nodes found.")
        return boundary_nodes, boundary_node_indices, node_locations_global


    def applyBoundaryCondition(self, K_, F_, E, bNodes, bNodeInd, nodeLoc):
        U_ = numpy.zeros((self.number_nodes,1))
        i = 0
        for node in bNodes:
            U_[node] = E(nodeLoc[bNodeInd[i]])
            i += 1
        delta_F = numpy.dot(K_,U_)
        F_ = F_ - delta_F

        # delete rows with fixed U values from the system of equations
        K_ = numpy.delete(K_, bNodes, 0)
        K_ = numpy.delete(K_, bNodes, 1)
        F_ = numpy.array([numpy.delete(F_, bNodes)]).T

        # solve system of equations
        U_ = numpy.dot( inv(K_), F_ )

        # insert boundary values
        bvals =[]
        for i in bNodeInd:
            bvals.append(E(nodeLoc[i]))

        sorted_bNodes = sorted(bNodes)
        for i in range(len(bNodes)):
            U_ = numpy.insert(U_, sorted_bNodes[i], 0.0)

        c = 0
        for i in bNodes:
            U_[i] = bvals[c]
            c += 1

        U_ = numpy.array([U_]).T

        S = numpy.reshape(U_,(self.nr_nd_dim),order='C')

        return S


    def solveWeakForm(self, Au_v, F_v, e, E):
        k_ij, f_j = self.ESM_ELV_weak(Au_v, F_v)
        K_, F_ = self.assemble(k_ij, f_j)
        bNodes, bNodeInd, nodeLoc = self.boundaryNodes(e)
        solution = self.applyBoundaryCondition(K_, F_, E, bNodes, bNodeInd, nodeLoc)
        return solution


    def solveRitz(self, W, e, E):
        k_ij, f_j = self.ESM_ELV_ritz( W )
        K_, F_ = self.assemble( k_ij, f_j)
        bNodes, bNodeInd, nodeLoc = self.boundaryNodes(e)
        solution = self.applyBoundaryCondition(K_, F_, E, bNodes, bNodeInd, nodeLoc)
        return solution


if __name__ == '__main__':
    mesh()



def HJB_piecewiseLinear(f, dimension, u_max):
    """returns operator equation (Au) for the picewise linear HJB
    """
    # symbols
    nr_controls = len(u_max)
    U = numpy.asarray(sympy.symbols('u:' + str(nr_controls))) # contol vector
    X = numpy.asarray(sympy.symbols('x:' + str(dimension)))
    S_x = numpy.asarray(sympy.symbols('S_x:' + str(dimension)))

    # pontryagin's principle
    H = numpy.dot(S_x,f(X,U)) # control hamiltonian

    # prepare substitution u = u*(x,p)
    U_to_U_star = []
    for i in range(nr_controls):
        U_to_U_star.append( (U[i], -u_max[i]*((sign(sympy.diff(H, U[i]))+1)*0.5)) )

    HJB = H.subs(U_to_U_star) # lower control hamiltonian
    HJB = sympy.simplify(HJB)
    return HJB


def HJB_quadraticCost(f, F, dimension, nr_controls):
    """returns HJB for a quadratic cost functional
       0 = alpha*u^2 + F(X,U) + S_x * f(X,U))
    """
    # symbols
    U = numpy.asarray(sympy.symbols('u:' + str(nr_controls))) # contol vector
    X = numpy.asarray(sympy.symbols('x:' + str(dimension)))
    S_x = numpy.asarray(sympy.symbols('S_x:' + str(dimension)))
    alpha = sympy.Symbol('alpha')
```

```
H = numpy.dot(S_x,f(X,U)) + F(X,U)

# prepare substitution u = u*(x,p)
U_to_U_star = []
for i in range(nr_controls):
    U_to_U_star.append( (U[i], -(1/alpha) * sympy.diff(H, U[i])))

HJB = (alpha/2)*numpy.dot(U,U) + F(X,U) + numpy.dot(S_x,f(X,U))

HJB = HJB.subs(U_to_U_star) # lower control hamiltonian
return HJB
```

Appendix C

FEMCO Toolbox

[ ]:
```
"""

    ---------------------------------------------------------------
    |       femco - Finite Element Method enhanced Control       |
    |       author: wolfgang flachberger (c)                      |
    |       em@il: wolfgang.flachberger@unileoben.ac.at           |
    |       30-11-2020                                            |
    ---------------------------------------------------------------

"""
from numpy.linalg import multi_dot as dot
import numpy
import sympy


class mesh:
    """creates mesh object
       arguments:
           domain: feasable domain of the state space: takes a list of n tuples
                   (a,b) that bound the domain in the corresponding dimension
                   (the domain is an n-dimensional box --> "box constraints")
           resolution: side length of the finite n-dimensional cube elements
    """
    def __init__(self, domain, resolution):

        self.domain = domain
        self.resolution = resolution

        """"""----------------------------------------------------------------"""
        dimension = len(domain) # dimension of state space
        nodes_per_element = 2**dimension
        # calculate number of finite cube elements to cover domain
        nr_el_dim = [] # list with number of elements necessary per axis
        nr_nd_dim = [] # list with number of nodes necessary per axis
        state_space_vol = 1
        for index in range(dimension): # for every dimension
            length = domain[index][1] - domain[index][0] # take length of "box"
            nr_el_dim.append(int(length/resolution)) # the domain is just
            # approximated --> the Jacobian matrix is therfore not necessary
            # because all elements are of the same size and shape
            nr_nd_dim.append(nr_el_dim[index] + 1)
            state_space_vol = state_space_vol * length
            pass
        e_loc_zeroind = []
        for index in range(dimension):
            e_loc_zeroind.append(domain[index][0])
        e_loc_zeroind = numpy.asarray(e_loc_zeroind)
        nr_el_dim = tuple(nr_el_dim)
        nr_nd_dim = tuple(nr_nd_dim)
        number_elements = numpy.prod(nr_el_dim) # total amount of elements
        number_nodes = numpy.prod(nr_nd_dim) # total amount of gloabal nodes
        nodes = numpy.arange(number_nodes,dtype=int)
        node_names_global = numpy.reshape(nodes,(nr_nd_dim),order='C')
        S = sympy.Symbol('S')
        v = sympy.Symbol('v') # test funtion
        X = numpy.asarray(sympy.symbols('x:' + str(dimension)))
        XI = numpy.asarray(sympy.symbols('xi:' + str(dimension)))
        S_x = numpy.asarray(sympy.symbols('S_x:' + str(dimension)))
        V_x = numpy.asarray(sympy.symbols('v_x:' + str(dimension)))
        """"""----------------------------------------------------------------"""

        self.state_space_vol = state_space_vol
        self.e_loc_zeroind = e_loc_zeroind
        self.dimension = dimension
        self.nodes_per_element = nodes_per_element
        self.number_elements = number_elements
        self.number_nodes = number_nodes
        self.nr_el_dim = nr_el_dim
        self.nr_nd_dim = nr_nd_dim
```

```
        self.node_names_global = node_names_global
        self.X = X
        self.XI = XI
        self.S = S
        self.S_x = S_x
        self.v = v
        self.V_x = V_x

        T = self.coincidenceTable()
        self.T = T
        h = self.interpolationFunctions()
        self.h = h
        dh_dx = self.interpolationFunctionsDerivatives()
        self.dh_dx = dh_dx

        """-----------------------------------------------------------------"""

    def coincidenceTable(self):
        """ coincidence_table[element,node] """
        coincidence_table = numpy.zeros((self.number_elements,self.nodes_per_element), dtype=int, order='C')
        local_to_global = numpy.zeros((2,)*self.dimension)
        i = 0
        for element_indices in numpy.ndindex(self.nr_el_dim):
            for node_indices in numpy.ndindex((2,)*self.dimension):
                location = tuple([sum(x) for x in zip(element_indices,node_indices)])
                local_to_global[node_indices] = self.node_names_global[location]
                pass
            array = numpy.reshape(local_to_global, self.nodes_per_element)
            coincidence_table[i] = array
            i += 1
            pass
        return coincidence_table

    def interpolationFunctions(self):
        """ interpolationfunction[node] """
        i = 0
        h = sympy.ones(self.nodes_per_element,1)
        for element_indices in numpy.ndindex((2,)*self.dimension):
            for axis in range(self.dimension):
                if element_indices[axis] == 0:
                    factor = -1
                else:
                    factor = 1
                h[i] = h[i] * (1 + factor * self.XI[axis] * 2/self.resolution)
            i += 1
        h = h/self.nodes_per_element
        return h

    def interpolationFunctionsDerivatives(self):
        """ interpolationfunction[node, derivativeaxis] """
        dh_dX = sympy.ones(self.nodes_per_element, self.dimension)
        for index in range(self.nodes_per_element):
            for axis in range(self.dimension):
                dh_dX[index,axis] = sympy.diff(self.interpolationFunctions()[index], self.XI[axis])
        return dh_dX

        """-----------------------------------------------------------------"""

    def ESM_weak(self, Au_v):
        # evaluate componets of k_ij (not integrated jet)
        k_ij = sympy.zeros(self.nodes_per_element,self.nodes_per_element)
        for i in range(self.nodes_per_element):
            for j in range(self.nodes_per_element):
                sub = []
                for k in range(self.dimension):
                    sub.append((self.S_x[k], self.dh_dx[i,k]))
                    sub.append((self.V_x[k], self.dh_dx[j,k]))
                    sub.append((self.S, self.h[i]))
                    sub.append((self.v, self.h[j]))
                k_ij[i,j] = Au_v.subs(sub)
        # prepare substitution for coordinate transformation
        x_to_xi = []
        for j in range(self.dimension):
            x_to_xi.append( (self.X[j], self.XI[j] + self.X[j]) )
        # transform to local coordinate frame and integrate
        for indices in numpy.ndindex(k_ij.shape):
            k_ij[indices] = k_ij[indices].subs(x_to_xi)
            for i in range(self.dimension):
                k_ij[indices] = sympy.integrate(k_ij[indices],(self.XI[i],-self.resolution/2,self.resolution/2))
        return k_ij

    def assemble(self, k_ij):
        # initialize transport arrays
        k = numpy.zeros((self.nodes_per_element, self.nodes_per_element))

        # initialize stiffnessmatrix and loadvector of the whole system
        K_ = numpy.zeros((self.number_nodes,self.number_nodes))

        el = 0 # element counter
        for e_index in numpy.ndindex(self.nr_el_dim): # for every element

            # prepare substitution for element
            evaluate = []
            e_location = self.e_loc_zeroind + numpy.asarray(e_index) * self.resolution + numpy.ones(self.dimension)*self.resolution/2
            for i in range(self.dimension):
                evaluate.append( (self.X[i], e_location[i] ))

            # evaluate k_ij and f_j for element
            for indices in numpy.ndindex((self.nodes_per_element, self.nodes_per_element)):
```

```
                k[indices] = k_ij[indices].subs(evaluate)

            # assemble K_ and F_
            for k_index in numpy.ndindex((self.nodes_per_element, self.nodes_per_element)):
                i = self.T[el,k_index[0]]
                j = self.T[el,k_index[1]]
                K_[i,j] += k[k_index] #round(k[k_index],1)

            el += 1 # next element

        return K_

    def stateToFEM(self, x):
        """ takes a column vector of a state in state space and transforms
            it to the corresponding location in FEM/Probability space
            (returns a column vector)
        """
        x = (x-numpy.array([self.e_loc_zeroind]).T)/self.resolution
        loc = tuple([int(numpy.round(number)) for number in x])
        node = self.node_names_global[loc]
        X = numpy.zeros((self.number_nodes,1),dtype=int)
        X[node] = 1
        return X

    def FEMtoState(self, X):
        node_name = numpy.where(X == 1)[0][0]
        node_indices = numpy.where(self.node_names_global == node_name)
        x = numpy.asarray(node_indices) * self.resolution + numpy.array([self.e_loc_zeroind]).T
        return x

    def Q_to_FEM_Q(self, Q):
        size = self.number_nodes
        I = numpy.identity(size, dtype=int)
        Q_ = numpy.zeros((size,size))
        for i in range(size):
            point = numpy.array([I[:,i]]).T
            x = self.FEMtoState(point)
            Q_[i,i] = dot((x.T, Q, x))[0][0]
        return Q_

if __name__ == '__main__':
    mesh()
```

List of Figures

Bibliography

[HC24] David Hilbert and Richard Courant. *Methoden der mathematischen Physik: Erster Band*. Springer-Verlag Berlin Heidelberg New York 1968, 1924.

[HC37] David Hilbert and Richard Courant. *Methoden der mathematischen Physik: Zweiter Band*. Springer-Verlag Berlin Heidelberg New York 1968, 1937.

[HL63] Jack K Hale and Joseph P LaSalle. "Differential equations: Linearity vs. nonlinearity". In: *SIAM Review* 5.3 (1963), pp. 249–272.

[Isa65] Rufus Isaacs. *Differential games: a mathematical theory with applications to warfare and pursuit, control and optimization*. Courier Corporation, 1965.

[Bry75] Arthur Earl Bryson. *Applied optimal control: optimization, estimation and control*. Routledge, 1975.

[Red84] JN Reddy. *Energy and variational principles in applied mechanics*. John Wiley and Sons New York, 1984.

[Nai02] D Subbaram Naidu. *Optimal control systems*. CRC press, 2002.

[Trö05] Fredi Tröltzsch. *Optimale Steuerung partieller Differentialgleichungen*. Vol. 2. Springer, 2005.

[Ros09] I Michael Ross. *A primer on Pontryagin's principle in optimal control*. Collegiate Publ., 2009.

[LMW+12] Anders Logg, Kent-Andre Mardal, Garth N. Wells, et al. *Automated Solution of Differential Equations by the Finite Element Method*. Springer, 2012. ISBN: 978-3-642-23098-1. DOI: 10.1007/978-3-642-23099-8.

[Lev14] Mark Levi. *Classical mechanics with calculus of variations and optimal control: an intuitive introduction*. Vol. 69. American Mathematical Soc., 2014.

[Aln+15] Martin S. Alnæs et al. "The FEniCS Project Version 1.5". In: *Archive of Numerical Software* 3.100 (2015). DOI: 10.11588/ans.2015.100.20553.

[RN16] Stuart Russell and Peter Norvig. "Artificial intelligence: a modern approach". In: (2016).

[LM19] Hans Petter Langtangen and Kent-Andre Mardal. *Introduction to numerical methods for variational problems*. Vol. 21. Springer Nature, 2019.