UNIVERSITÄT BAYREUTH

FAKULTÄT FÜR MATHEMATIK, PHYSIK UND INFORMATIK

MATHEMATISCHES INSTITUT

A Feasibility Problem Approach For Reachable Set Approximation

Von der Universität Bayreuth
zur Erlangung des akademischen Grades eines
Doktor der Naturwissenschaften (Dr. rer. nat.)
vorgelegte Abhandlung

vorgelegt von

Thomas Jahn

aus Dillingen/Donau

22. Dezember 2014

Betreuung:
Prof. Dr. L. Grüne

Danksagungen

Diese Arbeit ist während meiner Zeit als wissenschaftlicher Mitarbeiter am Lehrstuhl für Angewandte Mathematik der Universität Bayreuth entstanden – die bis dato schönste und ereignisreichste Zeit meines Lebens. Ich möchte die Gelegenheit nutzen, um mich bei einigen Menschen zu bedanken, die diesen Weg mit mir gegangen sind und/oder maßgeblich bei der erfolgreichen Fertigstellung dieser Arbeit beteiligt waren:

Lars Grüne: Der beste Chef, Doktorvater und Mentor der Welt. Danke für die Inspiration, die Möglichkeiten, die Freiheiten... für Alles!

Karl Worthmann, Jürgen Pannek: Ihr habt mich während meines Studiums auf die richtige Bahn gebracht.

Meine lieben Kollegen am Lehrstuhl: Danke für die schöne Zeit, die Gespräche, die privaten Unternehmungen. Ich habe mich jeden einzelnen Tag auf euch gefreut.

Sigrid Kinder: Die Liebenswürdigkeit in Person. Ich bin mir sicher, dich nie schlecht gelaunt oder genervt erlebt zu haben. In deinem Sekretariat scheint einfach immer die Sonne.

Carl Laird: Thank you very much for the very inspiring conversations about interior point methods and the great time in Texas!

Meine Studenten: Danke für eine Million Gründe, mich nicht mit meiner Dissertation beschäftigen zu müssen. Ich habe sie dankbar angenommen. :-)

Chrissie: Ach, wo soll ich da anfangen. Danke für's Rücken freihalten, wenn ich wieder einen guten Lauf hatte und für's in den Allerwertesten treten, wenn der Lauf wieder mal in's Stocken geraten ist. Alles Weitere gehört sicherlich nicht hier rein.

Meine Eltern: Danke, dass ich ernten kann, was ihr beiden ursprünglich mühsam gesät habt.

Bayreuth, den 22. Dezember 2014

Thomas Jahn

Contents

Introduction

Reachability and controllability analysis for dynamic control systems are powerful tools for numerous applications like trajectory prediction, system verification, collision avoidance or control strategy validation. The computation of reachable sets (and controllability sets) is a central part of this analysis.

During the last decades a lot of approaches for computing these sets have been published. Two popular and fast algorithms for general nonlinear dynamics are based on a level–set computation of a solution of a Hamilton–Jacobi–Isaacs partial differential equation [22, 23] or on solving a family of special optimal control problems [2,3]. Both algorithms are capable of considering lower–dimensional projections of reachable sets. This is very welcome since the handling of higher–dimensional sets within numerical algorithms is not efficient (and aside from that often impossible with current hardware).

For very general problems, the optimal control approach (based on distance functions) of Baier and Gerdts [2, 3] seems to be much more flexible. While the HJI–algorithm is only discussed for handling simple projections to Euclidean coordinate system planes, the optimal control approach offers more freedom owing to the ability of defining submanifolds by nonlinear equality constraints which intersect with the reachable set. It is also possible to restrict the trajectories at any time, not only at the beginning and/or the end. So, for universal nonlinear control systems, this approach currently seems to be the fastest and most promising way of approximating reachable sets. In addition to that, the algorithm is simple enough and quickly implemented by using some generic numerical modules like nonlinear optimizers and differential equation solvers.

However, we will see later in this work that this approach comes with some weaknesses. These are almost completely inherited from the underlying nonlinear program which is the core of the algorithm. The irony of situation is that the nonlinear program (which causes most of the problems) computes a solution of an optimal control problem which is only a makeshift proxy of the essential problem and question: *Is a specific point reachable*?

In this work, we will take the best of all the generic modules of the original algorithm – an optimal control context, nonlinear program and ODE–solver – and merge it to a specialized algorithm for solving thousands of pure reachability problems (instead of optimal control problems), which add up to a reachable set. The final algorithm will efficiently run on manycore computers and meet important requirements which are needed to port the calculation kernels to CUDA hardware.

How to read this work

This work is written in strict top–down design. We start in Chapter 1 by simply analyzing our aimed hardware to get the skills for efficiently implementating parallel algorithms for it.

After developing the theory of our new approach we will discuss the details about modifying and tuning a nonlinear optimizer for our special case in Chapter 2. This will lead to a specific iteration loop containing a linear system of equations which is the major part of the computational effort.

We slightly restrict the generality of the problem setting in Chapter 3 so that this linear system of equations will have a sparse regular structure. After a slight detour to a little parallelization theory and methods for solving linear systems of equations in Chapter 4, we will develop the procedures for efficiently solving the sparse linear system of equations in Chapter 5.

Finally in Chapter 6, we will bring everything together, define a detailed algorithm for computing a reachable set in parallel and will also have a small outlook on some CUDA implementation concepts. As a demonstration we will use this algorithm in Chapter 7 to compute the controllability set of a satellite docking on an uncontrolled target in earth's orbit.

Problem class definition

Generally, we consider the dynamic control system

$$\begin{array}{rcl} \dot{x}(t) & = & f(t, x(t), u(t)) \\ x(t) & \in & \mathcal{X}(t) \\ u(t) & \in & \mathcal{U}(t) \\ t & \in & [0, T] \end{array} \tag{1}$$

with time–dependent state constraints $\mathcal{X}(t) \subset \mathbb{R}^n$ (which by the way implicitly define the initial value of the ordinary differential equation) and control constraints and $\mathcal{U}(t) \subset \mathbb{R}^m$ on a bounded time interval. Without loss of generality we choose the start of the time interval as zero. For any $t \in [0, T]$ we define

$$\mathcal{R}(t) := \{x(t) \in \mathcal{X}(t) \mid \exists\, u(\cdot) : (x(\cdot), u(\cdot)) \text{ solves (1) } \} \tag{2}$$

as the reachable set of (1) at time t.

The definition is quite universal. The reachable set is not restricted to the start or the end of the trajectory which is why the problem formulation can be used to compute reachable sets and controllability sets.

Chapter 1

Hardware considerations

The reachable set algorithm will be constructed with the intent to benefit from parallel computation possibilities of the underlaying hardware. Therefore we will have a look at two common parallelization architectures: A conventional multicore CPU (like Intel i7) and nVidia's CUDA enabled GPU. Both architectures require totally different parallelization techniques. In this section we will consider the main characteristics of these different architectures to enable us to design a suitable algorithm, which will (hopefully) perform well on both processor types.

1.1 Multicore parallelization

In the majority of cases a parallel execution should simply occupy the different cores of a multicore CPU like Intel's Dual–Core or Quad–Core processors. The growing number of cores, a huge amount of (shared) system memory and the easy–to–use libraries like OpenMP or pthreads turn these processors into versatile devices.

Conventional multicore CPUs can run threads with completely different instruction sequences or functions with almost arbitrary memory operations on each of the CPU cores. Starting and synchronizing threads is not very expensive on this hardware, but it will take some time, since the thread management is done by the operating system. Communication between all threads is very easy and can be implemented by using the system's main memory with some mutex protection around memory accesses.

Efficient multicore CPU implementations are often using some producer–consumer or parallel queue processing techniques.

1.2 CUDA computation

Describing CUDA parallelization is much more complicated than describing CPU implementations. This is why we have to look a bit closer at this architecture to find out the main requirements of efficient implementation of algorithms. Unfortunately, this topic is way too huge to deal with all details. Here we will just mention some important facts to motivate some decisions taken later during the algorithm design. For further details, refer to [12, 18] and have a look at the official CUDA documentation, guides and white papers published by nVidia .

1.2.1 Hardware background

First of all, a GPU consists of several cores, the *multiprocessors*. Each multiprocessor, in turn, consists of several processors. The number of processors per multiprocessor depends on the device. For example, a *Tesla C2050* device has 14 multiprocessors, each with 32 processors. Presently, newer devices consist of up to 192 processors per multiprocessor.

Currently, a multiprocessor is designed as a *SIMD* device (single instruction streams, multiple data streams). While all processors can work in parallel, they lack of most of CPU's instruction control mechanisms: Every processor of a multiprocessor has the choice between executing the same instruction or doing nothing. The result of a processor instruction only differs by the data on which this instruction is performed.

The start of a GPU program (device code, called *kernel*) is done within a CPU program (host code) by the nVidia driver. The driver uploads GPU code together with execution parameters to the device and starts the kernel execution, which altogether will take some time.

1.2.2 Thread hierarchy

For a fast and efficient data–to–thread assignment, the threads on a GPU device are strictly organized in a two-level hierarchical way. On the lowest level, up to 512 threads with a three–dimensional enumeration are considered as a so called *thread block*. A thread block is SIMD–executed on a single multiprocessor[1]. All thread blocks are set into a two–dimensional *block grid* which can hold up to 65535 $\times$ 65535 thread blocks.

When the kernel is executed, all thread blocks are processed by all multiprocessors of the device in an undefined order. Hence, a synchronization between different threadblocks is not possible.

1.2.3 Parallel memory access: The difference between faster and slower

As already stated, a single instruction within a thread block works on different data values at the same time. As a consequence a single instruction can produce a very high memory traffic at once. Since memory access is terribly slow compared to computing instructions, coalescing memory access of multiple threads can be *the* crucial factor and often makes the difference between good and bad performance.

To accelerate the memory access, the GPU can merge the memory reads and writes of neighboring threads into single 128 byte transfers... and this is both a blessing and a curse. There are two things that can disturb parallel memory accesses and thereby destroy the efficiency of a GPU program:

- *Misaligned memory access*: A merged 128 byte transfer must always access memory blocks whose starting addresses are multiples of 128 bytes. If a memory access is not aligned, then two 128 byte blocks are copied with twice the transfer time. This problem often occurs during processing matrices with odd row and column numbers.

- *Strided memory access*: This happens, when thread i accesses a data element at address $i \cdot s$, where s is a striding factor. An operation like that aims to leave big regions

[1]Actually, it is not perfectly SIMD–executed, since in particular 32 processors cannot process 512 threads in SIMD style. But it is easier to understand and there is not a big difference from the user's point of view.

of memory between single elements untouched (depending on the size of s). Unfortunately, all requested elements will be copied via several 128 byte transfers with a lot of unwanted data, which might be the death blow for the algorithm's GPU-efficiency. A problem like that easily happens e.g. during matrix multiplication, where s is the number of rows or columns.

1.2.4 Maximizing the kernel execution performance

As mentioned before, the execution of a thread block is tied to a single multiprocessor. The number of threads per block is not tied to the number of processors per multiprocessor. The hardware scheduler makes the processors' activity jump between threads by suspending (marking as inactive) and resuming (marking as active) them. By doing so, the hardware can reduce idle time caused by memory transactions, register latency, and things like that. This mechanism also faciliates the multiprocessor to handle more thread blocks at the same time, which means more threads to switch over and more ways of reducing idle time.

Besides a maximum number, the amount of simultaneously processed thread blocks is limited by the ressources a thread block needs for execution. These ressources are *registers*, *shared memory* and *threads*. A multiprocessor has maximum limits, which vary from device to device. Determined by the physical design, the hard limits per multiprocessor of a *Tesla C2050* are:

Resident thread blocks: 8

Registers: 32768

Shared Memory: up to 48 kb

Threads: 1536

In this example we should try to write kernels with at most 192 threads per block, 6kb of shared memory usage and 21 registers per thread to achieve 100% device occupancy[2]. Of

[2]A kernel produces maximum device occupancy if the multiprocessor can run the maximum amount of possible threads – 1536 threads in this case. 48 kb distributed to 8 blocks are 6 kb per block, 32768 registers distributed to 1546 threads are $\approx$ 21 registers per thread.

course, this is just a vague starting guess. The runtime performance of a kernel indirectly depends on a lot of properties and also the complexity of the algorithm is often tied to the blocksize and memory copy transactions. In practice, optimal kernel performance is lively discussed in the CUDA community and often ends up in trial–and–error. However, a rule of thumb is in order to achieve a minimum of 50% device occupancy for a good idle time reduction.

The parallelization width should be big enough such that all available multiprocessors can run with maximum occupancy for a while.

1.2.5 Multiple CUDA–devices

Multiple CUDA–devices can be used at the same time which can be considered as an additional hierarchical level of parallelization. In this case, all devices work independently and can run their own kernels. The concept is similar to simple multicore CPU scheduling as described in Section 1.1, with a small difference: There is no GPU memory that can be shared between all devices during kernel runtime. If two CUDA–devices need to communicate, the data has to be copied to the CPU host memory followed by a copy transaction to the other CUDA–device. It is obvious that these copying transactions need a lot of time and should be avoided whenever possible.

1.2.6 CUDA in a nutshell

CUDA programming is a very huge topic. There is still much more to learn in addition to this chapter, like warps, efficient shared memory usage, thread branching and stuff like that. Some facts here have been described from a quite simplified point of view. But becoming CUDA experts is not the goal of this work. We want to outline the most restrictive properties of the CUDA hardware in order to have an idea, whether an algorithm has the chance to perform well or not *before* it has been implemented. Thus, when designing a reachable set algorithm, which is also suitable for nVidia's GPUs, we should try to keep some facts in mind:

1. **Stick to the thread hierarchy.** The problem must be distributable to independently running thread blocks, whose instructions are basically being executed in strict SIMD style. If going for multiple CUDA cards, every card must be able to process a part of the algorithm with very rare communication between all parts, since synchronizing multiple cards is comparatively slow.

2. **Size matters after all.** When trying to achieve full occupancy on a *Tesla C2050* you should run 1536 threads per multiprocessor. Since this hardware has 14 multiprocessors and we are able to use a computing server containing four of these cards, we have to feed 1792 hungry processors with at least 86016 threads. An algorithm should be suited for this massive parallelization.

3. **Kernels must be slim.** Huge kernels that require a lot of registers and/or shared memory reduce the device occupancy and along with it the execution speed. Instead of writing one big kernel it could be better to write several smaller ones, and run them serially.

4. **Seriously: Do not use any memory.** As this might not be possible, we should at least try to keep memory transactions at a minimum. The possibility of doing a lot of computation on registers or shared memory will speed up the execution significantly. If a high memory throughput cannot be avoided, design the kernel to make use of coalesced memory transactions by all means and knock on wood.

Chapter 2

A feasibility problem approach

A promising algorithm was first described in 2009 by Baier and Gerdts [2] and theoretically refined in [3]. The algorithm is based on minimizing distance functions via optimally controlled trajectories. The algorithm has very nice properties from the parallelization point of view. Furthermore, the concept allows for some modifications that enables the algorithm to handle dynamical systems of higher dimension under suitable circumstances. One application, which is based on this original algorithm and computes two–dimensional projections of reachable sets of a 7–dimensional system has been published in [29].

In this chapter we will describe the basic idea of Baier end Gerdts and enhance their concept in order to improve numerical speed, reliability and its ability for parallel implementation.

2.1 The original algorithm based on distance functions

2.1.1 Algorithm definition

The idea of this algorithm is quite simple. First of all, we choose a rectangular domain

$$\bar{\mathcal{X}} = [x_1^l, x_1^u] \times \cdots \times [x_n^l, x_n^u] \subset \mathbb{R}^n \tag{2.1}$$

wherein the reachable set is expected and define a discrete equidistant grid

$$\mathcal{G} = \left\{ (x_1, \dots, x_n) \mid x_i = x_i^l + \frac{x_i^u - x_i^l}{G_i - 1} \cdot k, \;\; k = 0, \dots, G_i - 1 \right\} \tag{2.2}$$

that approximately covers $\bar{\mathcal{X}}$. $\prod_{i=1}^n G_i$ is the number of gridpoints.

For each gridpoint $\tilde{x} \in \mathcal{G}$ we compute the optimal control function $\hat{u}(\cdot)$ which minimizes the distance between $\tilde{x}$ and the optimal trajectory point $\hat{x}(t)$. The point $\hat{x}(t)$ will either be equal to a grid point or part of the boundary of $\mathcal{R}(t)$. The set of all points $\hat{x}(t)$ is the resulting approximation of $\mathcal{R}(t)$.

More precisely, for all $\tilde{x} \in \mathcal{G}$ we solve the following optimization problem:

$$\begin{aligned} &\min_{u(\cdot)} \|x(t) - \tilde{x}\|_2^2 \\ &\text{s.th. } (x(\cdot), u(\cdot)) \text{ solve (1)} \end{aligned} \tag{2.3}$$

The approximation $\widetilde{\mathcal{R}}(t)$ is then defined as

$$\widetilde{\mathcal{R}}(t) := \{\hat{x}(t) \mid (\hat{x}(\cdot), \hat{u}(\cdot)) \text{ solve (2.3) for at least one } \tilde{x} \in \mathcal{G}\} \tag{2.4}$$

To allow a simple computation of (2.3), we transform the optimal control problem into a static nonlinear program by approximating $u(\cdot)$ by a piecewise constant function. We choose a time–horizon discretization length N with stepsize $\eta = T/N$. For better readability, we define

$$t_i := i\eta. \tag{2.5}$$

The control functions can now be approximated by

$$u(t) \equiv u_{N,i} \in \mathcal{R}^m \quad \text{for} \quad t \in [t_i, t_{i+1}[, \quad i = 0, \dots, N-1 \tag{2.6}$$

Alltogether, this leads to the distance function based algorithm for computing approximations of reachable sets:

Algorithm 2.1: Computing a reachable set approximation, original distance function approach

```
1: function ReachableSet($\mathcal{G}$)
2:     $\widetilde{\mathcal{R}}_\eta(t) \leftarrow \emptyset$
3:     for all $\tilde{x} \in \mathcal{G}$ do
4:         Solve the nonlinear program
```

$$\begin{aligned} \hat{u}(\cdot) = \underset{u_N}{\operatorname{argmin}}\ & \|x(t) - \tilde{x}\|_2^2 \\ \text{s.th.}\ & \dot{x}(t) = f(t, x(t), u(t)) \\ & u(t) \equiv u_{N,i}, \quad t \in [t_i, t_{i+1}[, \quad i = 0, \dots, N-1 \\ & u_{N,i} \in \mathcal{U}(t_i), \quad i = 0, \dots, N-1 \\ & x(t) \in \mathcal{X}(t) \\ & t \in [0, T] \end{aligned} \tag{2.7}$$

```
5:         $\widetilde{\mathcal{R}}_\eta(t) \leftarrow \widetilde{\mathcal{R}}_\eta(t) \cup \{x_{\hat{u}(t)}(t)\}$
6:     end for
7:     return $\widetilde{\mathcal{R}}_\eta(t)$
8: end function
```

2.1.2 Testing the algorithm

We use the Rayleigh problem [2] for benchmarking the algorithm. It is defined via the two–dimensional initial value problem

$$\begin{aligned} \dot{x}_1(t) &= x_2(t) \\ \dot{x}_2(t) &= -x_1(t) + x_2(t)(1.4 - 0.14 x_2(t)^2) + 4u(t) \\ x_1(0) &= -5 \\ x_2(0) &= -5 \\ u(t) &\in [-1, 1] \\ t &\in [0, 2.5] \end{aligned} \tag{2.8}$$

As a test setting we compute $\widetilde{\mathcal{R}}_\eta(2.5)$ with a little MATLAB implementation. An interior point method with L–BFGS[1] Hessian approximation and accuracy 10^{-5} will solve the non-linear program. Furthermore, we choose $\bar{\mathcal{X}} = [-7, 0] \times [3, 5.5]$ and $G_1 = G_2 = 200$. The optimizer's iteration limit is 100 iterations. Trajectories are approximated with $N = 20$ via a Runge–Kutta method of order 4. To accelerate the computation and improve the optimizer's reliability we traverse the grid row by row and always use the last solution as new initial guess.

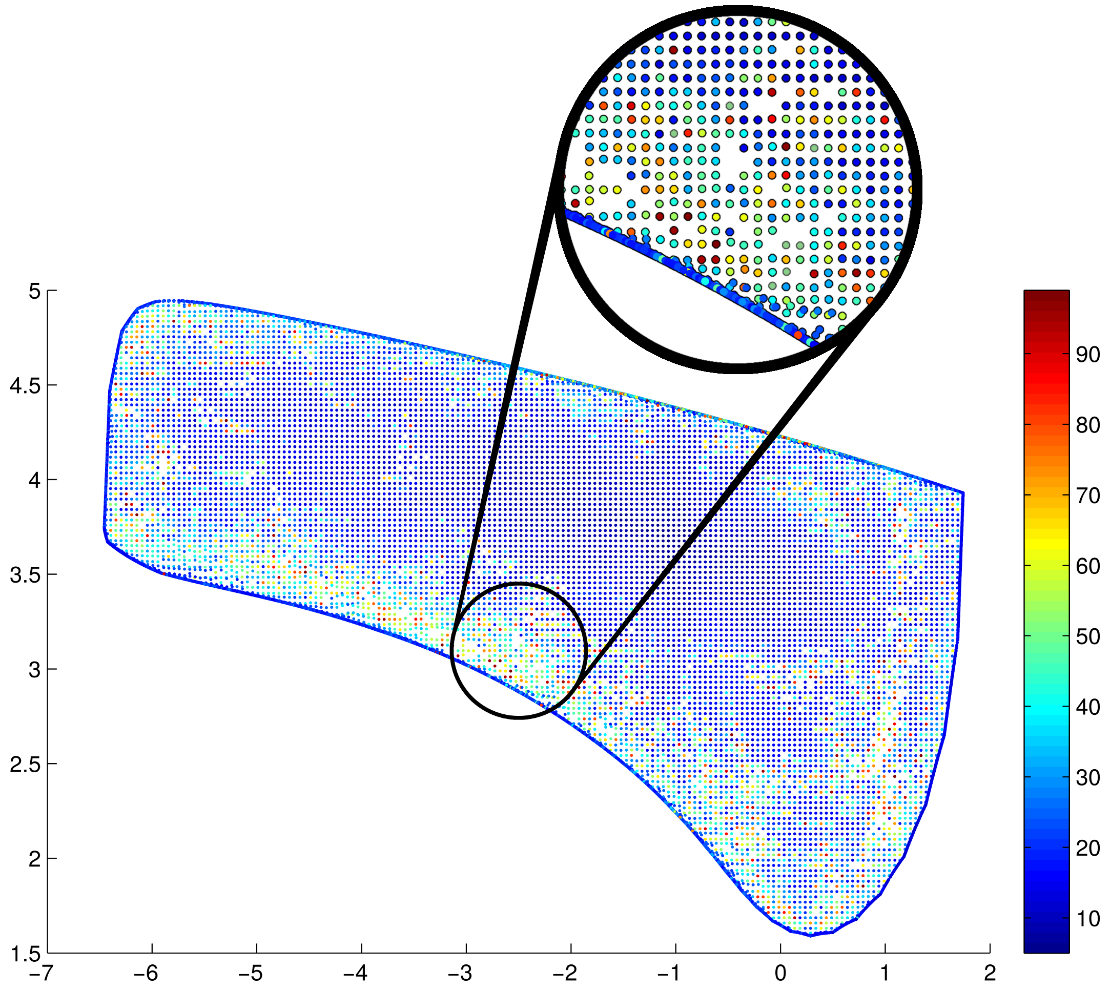


Figure 2.1: Reachable set of the Rayleigh problem (2.8) calculated with Algorithm 2.1 (grid: 200×200, $N = 20$), the colors of the dots indicate the number of required interior point steps to solve the corresponding NLP

The result is shown in Figure 2.1. The contours of the set have been approximated

[1]Limited Memory BFGS–approximation, see [24].

nicely. The cornered style of the line is ascribed to the quite rough stepsize of the trajectory approximation.

A closer look reveals little holes in the interior of the set which should not be there. Some regions of the set require some tricky control functions to be reached and the optimizer struggles to get there. The high number of iteration steps (visualised by the dots' colors) also attract attention. The median of iteration steps per grid point is 18, mean is approx. 21.

2.2 Deriving an improved algorithm

Since the first concepts of the distance function approach have been published, several improvements have been developed. Most of them consider speed enhancements by adaptively skipping grid points with redundant information [3], grid refinement techniques [26] and parallelization. Another work considers fast updates of reachable sets after parameter changes, based on sensitivity analysis [10].

This work is primarily aimed at advancing the speed and reliability of set detection. Strategies, which improve reachable point detection by using global optimization methods have already been considered in [4]. The new approach of this work makes use of the feasibility aspect of the whole problem. This is motivated by the idea that a set of reachable points of a dynamical system coincides with the set of feasible points, as already stated in [3]. We will also reduce the number of the optimizer's iteration steps and create an algorithm that is optimized for parallel hardware. At best it will nicely run on multiple CUDA devices.

2.2.1 Feasible solutions are good enough

Actually, we are not interested in a trajectory solution which hits a specific grid point as close as possible. The important question is: *Can we reach a grid point?* Since the answer doesn't include a precise distance, it should be easier to determine.

Therefore we simply introduce an equality constraint

$$\|x(T) - \tilde{x}\| \leq \varepsilon_{\mathcal{G}} \tag{2.9}$$

to the original optimization problem (2.7), where $\varepsilon_{\mathcal{G}}$ denotes the distance of two neighboring points in the grid. This restriction ensures that the trajectory will pass a cell around $\tilde{x}$ if possible. In exchange, the objective

$$\min_{u_N} \|x(T) - \tilde{x}\|_2^2$$

can be omitted, as it is not interesting *where* the cell is passed.

Simply choosing an objective function equal to zero can cause stability problems. To ensure a regular Hessian matrix of the Lagrangian function, we temporarily introduce the objective

$$\min_{u_N} \frac{1}{2}\xi \, \|u_N - \bar{u}\|_2^2 \tag{2.10}$$

with $\xi > 0$ and u_N is a complete discrete control trajectory as piecewisely defined in (2.6). Let $\bar{u}$ be a parameter which is very close to the final solution of the resulting optimization problem. The KKT conditions (see [24]) can then be written as

$$\begin{aligned} \xi(u_N - \bar{u}) - Dh(u_N)^T y &= 0 \\ h(u_N) &= 0 \end{aligned} \tag{2.11}$$

where y are Lagrangian multipliers, h are active constraints (including (2.9)) and Dh is the Jacobian matrix of h. $Dh(u_N)$ is assumed to have full rank for all u_N with $h(u_N) = 0$. The computation of the search direction within Newton's method requires the solution of the linear system of equations

$$\begin{pmatrix} \xi Id - \sum_k \nabla^2 h_k(u_N) y_k & -Dh(u_N)^T \\ Dh(u_N) & 0 \end{pmatrix} \begin{pmatrix} p_u \\ p_y \end{pmatrix} = \begin{pmatrix} -\xi(u_N - \bar{u}) + Dh(u_N)^T y \\ -h(u_N) \end{pmatrix}. \tag{2.12}$$

Assuming that we can start Newton's method with a very good initial guess, the equations can be simplified by disregarding $u_N - \bar{u}$ which is already almost zero. Hence, the conditions (2.11) and the full rank of $Dh(u_N)$ imply that the components y_k are also very small and

(2.12) can approximately be written as

$$\begin{pmatrix} \xi Id & -Dh(u_N)^T \\ Dh(u_N) & 0 \end{pmatrix} \begin{pmatrix} p_u \\ p_y \end{pmatrix} = \begin{pmatrix} Dh(u_N)^T y \\ -h(u_N) \end{pmatrix}. \tag{2.13}$$

Of course, $Dh(u_N)^T y$ will also be very small but this does not complicate solving the linear system of equations, so we will keep this term and will not worsen the approximation any more.

As the gradient $\xi(u_N - \bar{u})$ of the objective function does not appear on the right hand side (which means, that the current iterate u_N is already considered as optimal with respect to the objective function), the Newton step will focus on satisfying the condition $h(u_N) = 0$. Large values for ξ will reduce the convergence speed of Newton's method, since they shorten the original step size. On the other hand, ξ must be large enough such that

$$\xi Id - \sum_k \nabla^2 h_k(u_N) y_k \approx \xi Id \tag{2.14}$$

is an valid approximation. Numerical experiments in the end of this chapter will reveal that $\xi = 0.1$ is often a reasonable guess.

Since there is no need to compute a BFGS–like approximation (the hessian approximation has been reduced to a simple ξId matrix), the search direction can be computed faster. It is also possibile to make use of the partially sparse structure of the identity, which will speed up the computation further more and reduces memory transfers.

2.2.2 Tuning the convergence criteria

The iteration of the nonlinear program can instantly stop once a feasible solution has been found. There is no need to do final iterations to achieve a good accuracy with respect to an objective function. Hence, we choose a new exit condition based on a penalty criterion of the form

$$P(u_N) := \|h(u_N)\|_\infty < \nu \varepsilon_{\mathcal{G}}, \tag{2.15}$$

where ν is an accuracy parameter.

If (2.9) is the only restriction for the feasibility problem, the nonlinear program exits immediately when the cell around $\tilde{x}$ has been hit, which is perfect. If the feasibility problem also contains some equality constraints which are always active, the penalty criterion (2.15) will need a little more time to trigger. The iteration cannot simply stop when hitting the grid cell, as some equality constraints could still be violated. One could try to design the equality constraints in such a way that they converge quite fast during the Newton iterations, e.g. linear or heavy weighted (and ignore the weights within the penalty criterion).

2.2.3 A queue–based domain grid processing algorithm

The approximated step computation in (2.13) requires a very good initial guess. We can achieve this by reusing the optimal solutions of neighboring reachable cells. The idea is quite simple: After we found a reachable cell, we use the optimal solution to check the reachability of all neighboring cells by storing the cells and the initial guess in a FIFO buffer $\mathcal{F}$. This step is repeated until the buffer ran empty. This, in precise, leads to Algorithm 2.2. Note that an initial guess (u_N, y) is required which must be a valid solution of the feasibility problem.

Algorithm 2.2: Computing a reachable set approximation, new feasibility problem approach

```
1:  function ReachableSet($u_N, y, \mathcal{G}$)                ▷ valid initial guess $u_N, y$
2:      init $\mathcal{F}$                                       ▷ initialize FIFO–buffer
3:      determine grid point $\tilde{x}$ next to $x_{u(\cdot)}(t)$
4:      $\widetilde{\mathcal{R}}_\eta(t) \leftarrow \{\tilde{x}\}$
5:      loop
6:          for all gridpoints $\tilde{x}_i$ adjacent to $\tilde{x}$ with $\tilde{x}_i \notin \widetilde{\mathcal{R}}_\eta(t)$ do
7:              push $(\tilde{x}_i, u_N, y)$ onto $\mathcal{F}$
8:          end for
9:          repeat
10:             if $\mathcal{F} = \emptyset$ then
11:                 return $\widetilde{\mathcal{R}}_\eta(t)$
12:             end if
13:             pop $(\tilde{x}, \tilde{u}_N, \tilde{y})$ from $\mathcal{F}$
```

```
14:             **Solve the feasibility problem:**
                  search $u_N$ (and $y$) that satisfies penalty criterion (2.15) including (2.9),
                  use approximations of (2.13) for NLP iterations and $\tilde{u}_N$, $\tilde{y}$ for
                  warmstarting.
15:         **until** found $u_N$ and $y$
16:         $\widetilde{\mathcal{R}}_\eta(t) \leftarrow \widetilde{\mathcal{R}}_\eta(t) \cup \{\tilde{x}\}$
17:     **end loop**
18: **end function**
```

By doing so, every cell is tried to be reached within several attempts (e.g. up to eight attempts in a 2D set). This leads to a quite reliable reachability detection. In some cases, the trajectory switches discontinuously between two neighboring grid points. This might happen, if an obstacle can be passed on two different sides or a rotating object needs four turns instead of five. These cases can also be handled better by the buffer–approach. One cell can be processed with multiple initial guesses which can be completely different and come from distinct areas of the already computed reachable set.

Unfortunately, this will only work well if there always exists a path between two random reachable points whose corresponding trajectories vary continously. If this is not true, the optimizer has to "jump over a border" with a bad initial guess at least one time. Since this border usually has a lot of attached cells on both sides, hundreds or thousands of attempts will happen. Hence, there is a high probability that the algorithm can detect the complete set, although this is not guaranteed. The probability can be increased further by starting the algorithm with several initial guesses which are distributed over the reachable set.

2.2.4 Early infeasibility assumption

Another key to fast grid processing is early dropping a feasibility problem, if a single grid cell cannot be reached or is hard to reach. A failed feasibility check will be followed by several additional attempts with different initial guesses. So it is not necessary to do hundreds of iterations to finally solve hard feasibility checks. It might be more economic to exit quite early when the problem seems to diverge and hope for better luck during the next try.

A heuristic approach could be to observe the quotient

$$\frac{P(u_N^+)}{P(u_N)} \tag{2.16}$$

where u_N^+ is the next iterate while solving the nonlinear problem. If $P(u_N)$ does not converge to zero, then the quotient will converge to one, which can be (approximately) detected after a few steps. Of course, this can easily lead to false alerts, as problems could just have bad convergence at the beginning of the iterations. But since one grid point will be visited several times by the algorithm, there is no need to make the most out of one feasibility check. Three attempts with 10 iterations are still better than a single attempt with 100 iterations.

In practice we simply terminate a single feasibility check if (2.16) is greater than 0.95 a couple of times, assuming that another attempt will be more promising.

2.3 Numerical results

To test the new concept we make use of the previous example (2.8) and compute the set approximation with the same grid size and optimizer accuracy. We choose $\xi = 0.1$ and as infeasibility assumption we abort the grid point processing after (2.16) has been sctrictly greater than one 10 times in a row. A simple trajectory with zero control has been used as initial trajectory for the queue processing.

The result is quite good. Figure 2.2 demonstrates how the algorithm proceeds. The algorithm starts to detect reachable points near the first trajectory and spreads from there until the whole set approximation has been computed. The figure also shows that no holes have been left, so the concept seems to be more reliable then the original algorithm.

Figure 2.3 shows a comparison between the required iteration steps of the original and the new algorithm. Note that the color bar of the second plot has been modified to a logarithmic scale to emphasize the differences. The new concept produces much lower iteration numbers which are approximately three steps in average and median.

The last goal of the new design is to create a very big potential to parallelize the algorithm. While the original algorithm can process all grid points at the same time, the parallelization bandwith of the new concept is restricted to the current size of the FIFO buffer. Fortunately, this buffer grows quite fast, even for this simple example. Figure 2.4 shows the varying buffer size during the algorithm execution. The buffer size instantly grows up to 300 grid points and stays around 700 grid points most of the time. This should be more than enough for running efficient CPU and GPU programs.

As conclusion, we can say that the new feasibility based approach seems to be very promising with respect to reliability and iteration speed and is worth to be developed further. The remainder of this thesis is dedicated to the design of a high performance implementation of this concept for general nonlinear problems.

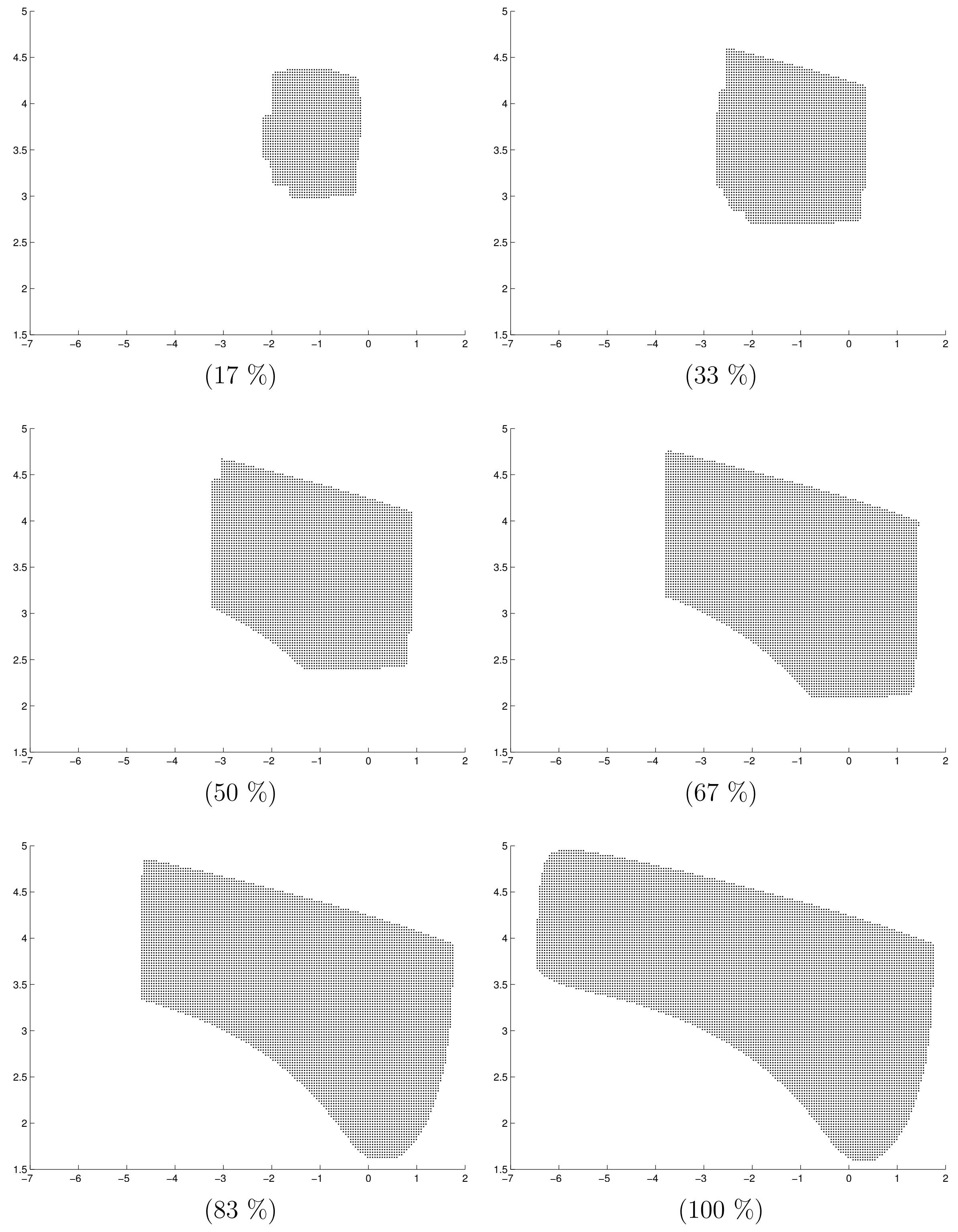
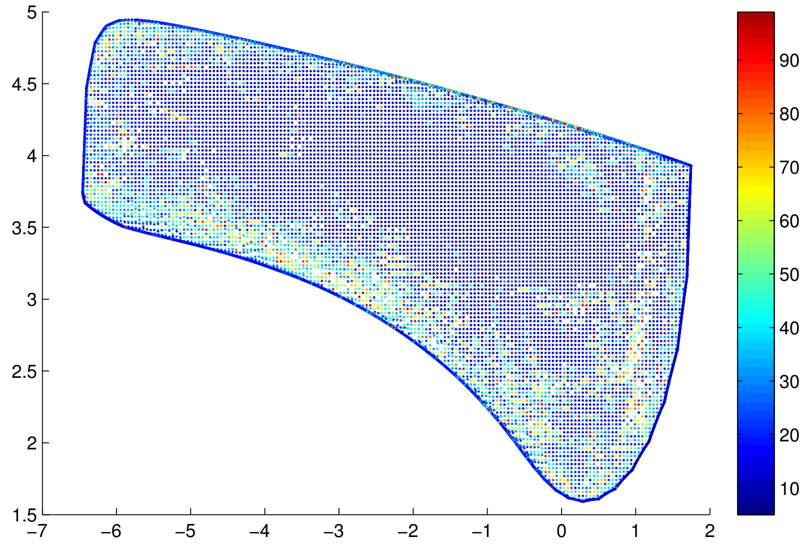
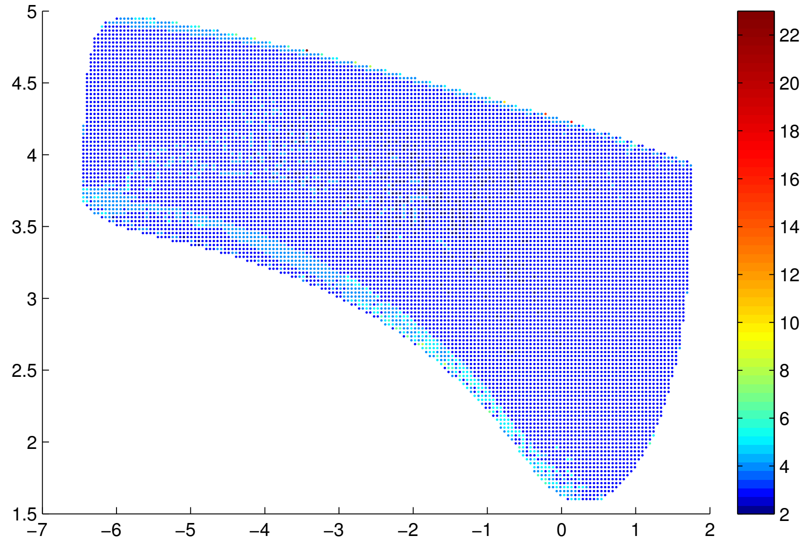


Figure 2.2: Demonstration of the queue processing including the final result of the feasibility based approach



Original algorithm based on distance functions



New algorithm based on feasibility problems

Figure 2.3: Comparison of needed iteration steps per gridpoint

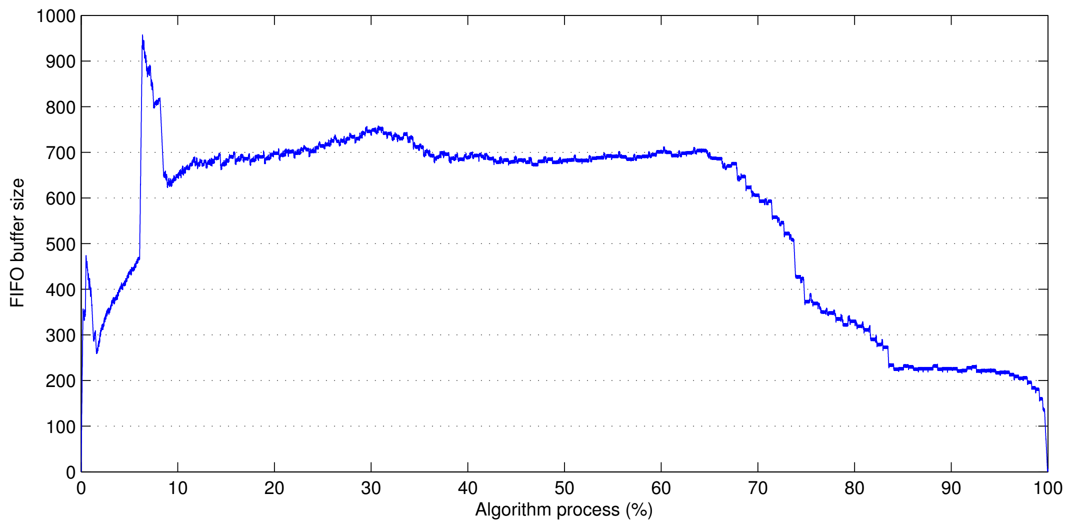


Figure 2.4: Development of the FIFO buffer size (number of queued grid points $\tilde{x}$ with initial guess) during the algorithm execution

Chapter 3

Tuning the interior point method

The resulting nonlinear program (2.10) of the last chapter is unproblematic from a numerical point of view. Any NLP algorithm which is capable of solving nonlinear restricted optimization problems should yield a useful result. In this work we choose the idea of the nonlinear interior point method [24] as a base and specialize this algorithm in solving the feasibility problems in parallel, fast and efficiently. We choose this method for two reasons:

1. While iterating, the structure of the interior point method does not change, even if restrictions are switching beween being active or inactive (since they do not really "switch" discretely). This fact will allow a potential implementation on CUDA hardware, which is known as quite unflexible.

2. Considering the difficulty with warmstarting of interior point methods, the curse is actually a blessing in this case. The barrier parameter $\mu > 0$, which keeps the solution away from active constraints during NLP iterations (and worsens the value of the objective function), works like an accelerator for fulfilling inequality constraints like the grid restriction (2.9).

In this chapter we will analyze the details of the interior point method and see what happens if we use this method for solving feasibility problems. We will develop a detailed problem formulation for the feasibility problem which will lead to memory–saving and sparse matrix algebra with a fixed structure in order to be suitable for CUDA implementations.

3.1 The interior point method

The formulation of the interior point method in this section is very close to *Nocedal* and *Wright* [24]. We refer to the book for more details and proofs of convergence. Here, we will just elaborate the important computational steps of the interior point method and adapt them to our feasibility problem setting.

3.1.1 Problem formulation

We consider the nonlinear program

$$\begin{array}{rrcl} & \min_x J(x) & & \\ \text{s.t.} & h(x) & = & 0 \\ & g(x) & \geq & 0 \end{array} \tag{3.1}$$

where $J(x)$ is the objective function and $h(x)$ and $g(x)$ are equality and inequality constraints.

The interior point method for solving the general nonlinear program (3.1) is based on the so called *barrier problem*

$$\begin{array}{rrcl} & \min_{x,s} J(x) - \mu \sum_i \log s_i & & \\ \text{s.t.} & h(x) & = & 0 \\ & g(x) - s & = & 0 \end{array} \tag{3.2}$$

where $s_i \geq 0$ are slack variables (positive values are assured by the term $-\mu \sum \log s_i$) and $\mu > 0$ is the barrier parameter. The KKT–conditions for solving (3.2) are

$$\begin{array}{rcl} \nabla J(x) - Dh(x)^T y - Dg(x)^T z & = & 0 \\ -\mu S^{-1} e + z & = & 0 \\ h(x) & = & 0 \\ g(x) - s & = & 0 \end{array} \tag{3.3}$$

where $z \geq 0$ and y are Lagrangian multiplicators, S is a matrix with s_i as diagonal elements and $e = (1, \dots, 1)^T$. The barrier approach iteratively searches for a solution of (3.3) while sending the barrier parameter to zero with every iteration step.

3.1.2 An interior point step for feasibility problems

One iteration step of the interior point method basically consists of an iteration of Newton's method, solving (3.3). Due to stability issues, we transform the second equation

$$-\mu S^{-1}e + z = 0 \quad \Leftrightarrow \quad Sz - \mu e = 0. \tag{3.4}$$

Moreover, we can apply the approximations that have been described in Section 2.2.1 and finally compute an approximated step of Newton's method by solving the linear system of equations

$$\begin{pmatrix} \xi Id & 0 & -Dh(x)^T & -Dg(x)^T \\ 0 & Z & 0 & S \\ Dh(x) & 0 & 0 & 0 \\ Dg(x) & -Id & 0 & 0 \end{pmatrix} \begin{pmatrix} p_x \\ p_s \\ p_y \\ p_z \end{pmatrix} = - \begin{pmatrix} -Dh(x)^T y - Dg(x)^T z \\ Sz - \mu e \\ h(x) \\ g(x) - s \end{pmatrix} \tag{3.5}$$

where p_x, p_s, p_y, p_z are search directions.

The next iterate (x^+, s^+, y^+, z^+) of the interior point method step can be obtained by

$$\begin{aligned} x^+ &= x + \alpha_s p_x \\ s^+ &= s + \alpha_s p_s \\ y^+ &= y + \alpha_z p_y \\ z^+ &= z + \alpha_z p_z \end{aligned} \tag{3.6}$$

with α_s and α_z are defined by

$$\begin{aligned} \alpha_s &= \max\{\alpha \in (0,1] \mid s + \alpha p_s \geq (1-\tau)s\} \\ \alpha_z &= \max\{\alpha \in (0,1] \mid z + \alpha p_z \geq (1-\tau)z\} \end{aligned} \tag{3.7}$$

in which $\tau \in (0,1)$ is chosen as $\tau = 0.995$. Shortening the stepsize by the factors α_s and α_z (the so called *fraction to boundary rule*) ensures that the convergence does not stick to an active restriction too early (which would extremely limit the size of the next steps). While iterating, the parameter must be updated such that $\tau \to 1$ holds to reach active restrictions in the end.

3.1.3 Line search method

The step size can optionally be shortened further by a one–dimensional line search towards the search direction. It turns out that the *Armijo condition* is well suited, since it does not need any additional gradient evaluations. For all $\alpha_s \geq 0$ we define a merit function

$$\Phi(\alpha_s) = \sum_i |h_i(x + \alpha_s p_x)| + \sum_{i \in \mathcal{I}} |g_i(x + \alpha_s p_x)| \tag{3.8}$$

whose minimum is a good indicator for feasible solutions. $\mathcal{I}$ is the set of indices of active (or violated) inequality constraints. Assuming that $\Phi(\alpha_s)$ is differentiable at $\alpha_s = 0$ we can write the derivative as

$$\Phi'(0) = \sum_i \nabla h_i(x) p_x \cdot \operatorname{sgn}(h_i(x)) - \sum_{i \in \mathcal{I}} \nabla g_i(x) p_x. \tag{3.9}$$

A step size α_s will be accepted if the *Armijo condition*

$$\Phi(\alpha_s) \leq \Phi(0) + \gamma \alpha_s \Phi'(0) \tag{3.10}$$

holds with, e.g., $\gamma = 10^{-4}$. If not, a backtracking line search is performed by iteratively approximating $\Phi(\alpha_s)$ by a quadratic polynomial and choosing the minimum as new backtracking step until (3.10) holds. The following theorem describes how to choose the next backtracking step size α_s^+.

Theorem 3.1

Let Φ be a merit function with $\Phi'(0) < 0$ and $\alpha_s > 0$ a step size which violates the Armijo condition with $0 < \gamma < \frac{1}{2}$. The next backtracking step size α_s^+ minimizing a quadratic *approximation of Φ is given by*

$$\alpha_s^+ = \frac{-\Phi'(0)\alpha_s^2}{2\left(\Phi(\alpha_s) - \Phi(0) - \Phi'(0)\alpha_s\right)} \tag{3.11}$$

which approximates the minimum of Φ. In particular, $0 < \alpha_s^+ < \alpha_s$ holds.

Proof: Let Φ be approximated by a polynomial of the form

$$P_\Phi(\alpha_s) = a_0 + a_1\alpha_s + a_2\alpha_s^2,$$

where P_Φ fulfills the following interpolation conditions:

$$\begin{aligned} P_\Phi(0) &= a_0 = \Phi(0) \\ P_\Phi(\alpha_s) &= a_0 + a_1\alpha_s + a_2\alpha_s^2 = \Phi(\alpha_s) \\ \dot{P}_\Phi(0) &= a_1 = \Phi'(0) \end{aligned}$$

Then a_2 can be obtained by

$$\begin{aligned} \Phi(0) + \Phi'(0)\alpha_s + a_2\alpha_s^2 &= \Phi(\alpha_s) \\ \Leftrightarrow a_2 &= \frac{1}{\alpha_s^2}\left(\Phi(\alpha_s) - \Phi(0) - \Phi'(0)\alpha_s\right) \end{aligned}$$

Since the Armijo condition (3.10) does not hold, it holds that

$$\Phi(\alpha_s) - \Phi(0) - \Phi'(0)\alpha_s > \Phi(0) + \gamma\alpha_s\Phi'(0) - \Phi(0) - \Phi'(0)\alpha_s = (\gamma - 1)\alpha_s\Phi'(0) \geq 0, \tag{3.12}$$

therefore P_Φ is convex and the minimum α_s^+ is given by

$$\begin{aligned} 0 = \dot{P}_\Phi(\alpha_s^+) &= \Phi'(0) + \tfrac{2}{\alpha_s^2}\left(\Phi(\alpha_s) - \Phi(0) - \Phi'(0)\alpha_s\right)\alpha_s^+ \\ \Leftrightarrow \alpha_s^+ &= \frac{-\Phi'(0)\alpha_s^2}{2\left(\Phi(\alpha_s) - \Phi(0) - \Phi'(0)\alpha_s\right)}. \end{aligned}$$

Furthermore, due to (3.12) and $\Phi'(0) < 0$ it holds that

$$\begin{aligned} 0 &< \frac{-\Phi'(0)\alpha_s^2}{2\left(\Phi(\alpha_s) - \Phi(0) - \Phi'(0)\alpha_s\right)} = \alpha_s^+ \\ &< \frac{-\Phi'(0)\alpha_s^2}{2(\gamma-1)\alpha_s\Phi'(0)} = \frac{1}{2(1-\gamma)}\alpha_s \\ &< \alpha_s. \end{aligned}$$

□

3.1.4 The termination condition

Motivated by (2.15) we simply use the merit function Φ for defining a termination condition. As it becomes zero once a feasible point has been found, we exit the iteration if the condition

$$\Phi(0) = \sum_i |h_i(x)| + \sum_{i\in\mathcal{I}} |g_i(x)| < \nu\varepsilon_G \qquad (3.13)$$

holds with $\nu = 10^{-2}$. This strategy allows a small numerical constraint violation which is of the same magnitude as the grain size of the chosen grid $\mathcal{G}$ and thus tolerable.

3.1.5 Updating the barrier parameter

While performing the iterations of the interior point method, the barrier parameter μ has to be updated such that $\mu \to 0$. We prefer an adaptive selection of μ. This allows to start with larger values for μ (to provide a better stability) and to react quickly to good initial guesses at the same time. The reference [24] describes an adaptive strategy which uses s^Tz (converges to zero) to estimate the progress of the iteration steps. The next barrier parameter μ^+ is defined by

$$\mu^+ = \sigma\frac{s^Tz}{m}, \qquad (3.14)$$

where m is the number of inequality constraints. Here $\sigma > 0$ is a factor which accelerates the descent of μ for small values of σ.

To achieve a uniform convergence of all inequality constraints, the parameter σ can be chosen in such a way that it becomes larger when individual constraints converge too fast to their active border. In every step, we choose σ adaptively as

$$\sigma = 0.1 \min\left\{0.05\frac{1-\xi}{\xi}, 2\right\}^3 \quad \text{with} \quad \xi = \frac{\min_i\{s_i z_i\}}{\frac{1}{m}s^T z} \tag{3.15}$$

which causes the descent of μ to be much slower if the smallest complementary product $s_i z_i$ is far from the average.

A uniform convergence of all inequality constraints is supported by the *Fiacco-McCormick approach.* While iterating, we fix the barrier parameter until

$$\sum_i |h_i(x)| + \sum_{i\in\mathcal{I}} |g_i(x)| < \mu \tag{3.16}$$

holds. Then we update μ as described in (3.14) and continue. This procedure enables slowly converging inequality constraints to close the gap on fast converging ones and prevents individual s_i and z_i from converging to zero too fast.

When updating the barrier parameter, we also update $\tau^+ = \max\{1 - 0.1\mu^+, 0.995\}$.

3.2 Constructing a sparse linear system of equations

The previous section has shown a modification of the interior point method for solving general feasibility problems. We will now apply this method to our optimal control problem. The most expensive part (by far) of the interior point method lies in solving the linear system of equations (3.5). Storing the matrix will need a lot of memory and solving the system a lot of time. We will make this step less expensive by defining the optimal control problem in a way such that Dh and Dg are sparse matrices. This will "sparsify" the whole structure of the matrix in (3.5) as Z and S are simple diagonal matrices.

3.2.1 A full discretization formulation of the feasibility problem

An optimal control problem with the constraints

$$\dot{x}(t) = f(t, x(t), u(t)) \tag{3.17a}$$

$$u(t) \equiv u_i, \quad t \in [t_i, t_{i+1}[, \quad i = 0, \ldots, N-1 \tag{3.17b}$$

$$u_i \in \mathcal{U}(t_i), \quad i = 0, \ldots, N-1 \tag{3.17c}$$

$$x(t) \in \mathcal{X}(t) \tag{3.17d}$$

with $t \in [0, T]$ must be expressed by the constraint functions g and h of the interior point method.

Using an r–stage Runge–Kutta method [6] with $A \in \mathbb{R}^{r \times r}$ and $c, b \in \mathbb{R}^r$, $x(t)$ can be approximated via

$$\begin{aligned} k_{i,j} &= f(t_{i-1} + c_j \eta, x_{i-1} + \eta \sum_{l=1}^{r} a_{j,l} k_{i,l}, u_i), \ j = 1, \ldots, r \\ x_i &= x_{i-1} + \eta \sum_{l=1}^{r} b_l k_{i,l} \end{aligned} \tag{3.18}$$

with $i = 1, \ldots, N$ and $x_i \approx x(t_i)$ and $x_0 = x(t_0)$. We approximately implement the theoretical constraints (3.17a) and (3.17b) by the equations (3.18). Naturally, just doing one Runge–Kutta step per horizon step can be a quite rough approximation. On the other hand, embedding the Runge–Kutta method into the optimizer's iteration loop makes it possible to use high order implicit methods with comparatively few stages and without any additional cost (e.g. Radau 5).

The other constraints (3.17c) and (3.17d) can be modeled via a combination of nonlinear functions

$$r_u(u_i) > 0, \quad i = 1, \ldots, N \tag{3.19a}$$

$$r_x(x_i) > 0, \quad i = 0, \ldots, N \tag{3.19b}$$

$$q_u(u_i) = 0, \quad i = 1, \ldots, N \tag{3.19c}$$

$$q_x(x_i) = 0, \quad i = 0, \ldots, N \tag{3.19d}$$

and reduced box constraints

$$P_{u,i}^U u_i \leq \tilde{u}_i^U, \quad i = 1, \ldots, N \tag{3.20a}$$

$$P^L_{u,i} u_i \geq \tilde{u}^L_i, \quad i = 1, \ldots, N \tag{3.20b}$$

$$P^U_{x,i} x_i \leq \tilde{x}^U_i, \quad i = 0, \ldots, N \tag{3.20c}$$

$$P^L_{x,i} x_i \geq \tilde{x}^L_i, \quad i = 0, \ldots, N \tag{3.20d}$$

where $P^U_{u,i}$, $P^L_{u,i}$, $P^U_{x,i}$ and $P^L_{x,i}$ are permutation matrices, which "select" individual elements of u_i and x_i for being constrained. $\tilde{u}^U_i$ and $\tilde{x}^U_i$ are upper bounds, $\tilde{u}^L_i$ and $\tilde{x}^L_i$ are lower bounds.

For better readability, we use the abbreviations

$$f_{i,j} := f(t_{i-1} + c_j \eta, x_{i-1} + \eta \sum_{l=1}^{r} a_{j,l} k_{i,l}, u_i) \tag{3.21a}$$

$$\partial_u f_i := \begin{pmatrix} \frac{\partial}{\partial u} f_{i,1} \\ \vdots \\ \frac{\partial}{\partial u} f_{i,r} \end{pmatrix} \tag{3.21b}$$

$$\partial_x f_i := \begin{pmatrix} \frac{\partial}{\partial x} f_{i,1} \\ \vdots \\ \frac{\partial}{\partial x} f_{i,r} \end{pmatrix} \tag{3.21c}$$

$$\partial_x f^A_i := \begin{pmatrix} a_{1,1}\eta \frac{\partial}{\partial x} f_{i,1} & \cdots & a_{1,r}\eta \frac{\partial}{\partial x} f_{i,1} \\ \vdots & \ddots & \vdots \\ a_{r,1}\eta \frac{\partial}{\partial x} f_{i,r} & \cdots & a_{r,r}\eta \frac{\partial}{\partial x} f_{i,r} \end{pmatrix} - Id \tag{3.21d}$$

$$B := \begin{pmatrix} \eta b_1 Id & \cdots & \eta b_r Id \end{pmatrix} \tag{3.21e}$$

$$q_{u,i} := q_u(u_i) \tag{3.21f}$$

$$q_{x,i} := q_x(x_i) \tag{3.21g}$$

$$r_{u,i} := r_u(u_i) \tag{3.21h}$$

$$r_{x,i} := r_x(x_i) \tag{3.21i}$$

$$\dot{q}_{u,i} := \dot{q}_u(u_i) \tag{3.21j}$$

$$\dot{q}_{x,i} := \dot{q}_x(x_i) \tag{3.21k}$$

$$\dot{r}_{u,i} := \dot{r}_u(u_i) \tag{3.21l}$$

$$\dot{r}_{x,i} := \dot{r}_x(x_i) \tag{3.21m}$$

in the rest of this thesis.

If we combine all the variables u_i, x_i and $k_{i,j}$ into one vector

$$x := (u_1, \dots, u_N, x_0, \dots, x_N, k_{1,1}, \dots, k_{1,r}, \dots, k_{N,1}, \dots, k_{N,r})^T \in \mathbb{R}^{Nm+(N+1)n+Nnr} \tag{3.22}$$

we can write the constraint functions as

$$h(x) = \begin{pmatrix} f_{1,1} - k_{1,1} \\ \vdots \\ f_{1,r} - k_{1,r} \\ x_0 + \eta \sum_{l=1}^{r} b_l k_{1,l} - x_1 \\ \vdots \\ f_{N,1} - k_{N,1} \\ \vdots \\ f_{N,r} - k_{N,r} \\ x_{N-1} + \eta \sum_{l=1}^{r} b_l k_{N,l} - x_N \\ q_{u,1} \\ \vdots \\ q_{u,N} \\ q_{x,0} \\ \vdots \\ q_{x,N} \end{pmatrix}, \quad g(x) = \begin{pmatrix} \tilde{u}_1^U - P_{u,1}^U u_1 \\ P_{u,1}^L u_1 - \tilde{u}_1^L \\ r_{u,1} \\ \vdots \\ \tilde{u}_N^U - P_{u,N}^U u_N \\ P_{u,N}^L u_N - \tilde{u}_N^L \\ r_{u,N} \\ \tilde{x}_0^U - P_{x,0}^U x_0 \\ P_{x,0}^L x_0 - \tilde{x}_0^L \\ r_{x,0} \\ \vdots \\ \tilde{x}_N^U - P_{x,N}^U x_N \\ P_{x,N}^L x_N - \tilde{x}_N^L \\ r_{x,N} \end{pmatrix} \tag{3.23}$$

with the corresponding Jacobian matrices

$$Dh(x) = \begin{pmatrix}
\partial_u f_1 & & & \partial_x f_1 & & & & & \partial_x f_1^A & & \\
0 & & & Id & -Id & & & & B & & \\
 & \ddots & & & \partial_x f_2 & \ddots & & & & \ddots & \\
 & & \partial_u f_N & & & & \partial_x f_N & & & & \partial_x f_N^A \\
 & & 0 & & & & Id & -Id & & & B \\
\dot{q}_{u,1} & & & & & & & & & & \\
 & \ddots & & & & & & & & & \\
 & & \dot{q}_{u,N} & & & & & & & & \\
 & & & \dot{q}_{x,0} & & & & & & & \\
 & & & & \dot{q}_{x,1} & & & & & & \\
 & & & & & \ddots & & & & & \\
 & & & & & & \dot{q}_{x,N-1} & & & & \\
 & & & & & & & \dot{q}_{x,N} & & &
\end{pmatrix} \tag{3.24a}$$

and

$$
Dg(x) = \begin{pmatrix}
-P^U_{u,1} & & & & & & 0 \\
P^L_{u,1} & & & & & & \vdots \\
\dot{r}_{u,1} & & & & & & \\
& \ddots & & & & & \\
& & -P^U_{u,N} & & & & \\
& & P^L_{u,N} & & & & \\
& & \dot{r}_{u,N} & & & & \\
& & & -P^U_{x,0} & & & \\
& & & P^L_{x,0} & & & \\
& & & \dot{r}_{x,0} & & & \\
& & & & \ddots & & \\
& & & & & -P^U_{x,N} & \\
& & & & & P^L_{x,N} & \vdots \\
& & & & & \dot{r}_{x,N} & 0
\end{pmatrix}. \tag{3.24b}
$$

These matrices have a nice sparse structure which can be exploited when calculating the search direction of the interior point method.

3.2.2 Identifying the non–zero elements

Finally, we will analyze the sparse structure of the linear system of equations (3.5) so that we can configure a solver to keep the zero elements in mind. Before we substitute the matrices Dg and Dh as defined in (3.24a) and (3.24b) we can simplify the problem a little bit by reducing the actual matrix.

The linear system of equations (3.5) can be written as

$$
\begin{aligned}
\xi p_x - Dh(x)^T p_y - Dg(x)^T p_z &= Dh(x)^T y + Dg(x)^T z && (3.25a)\\
Z p_s + S p_z &= \mu e - S z && (3.25b)\\
Dh(x) p_x &= -h(x) && (3.25c)
\end{aligned}
$$

$$Dg(x)p_x - p_s = -g(x) + s. \tag{3.25d}$$

Equations (3.25b) and (3.25d) imply the explicit partial solutions

$$p_s = Dg(x)p_x + g(x) - s \tag{3.26a}$$

$$p_z = -S^{-1}Zp_s - z + \mu S^{-1}e. \tag{3.26b}$$

Substituting p_s and p_z in (3.25a) and defining $\Sigma = S^{-1}Z$ (Σ is also a diagonal matrix), we get

$$\begin{aligned}
& \xi p_x - Dh(x)^T p_y - Dg(x)^T(-\Sigma p_s - z + \mu S^{-1}e) \\
= \quad & \xi p_x - Dh(x)^T p_y - Dg(x)^T(-\Sigma(Dg(x)p_x + g(x) - s) - z + \mu S^{-1}e) \\
= \quad & (\xi Id + Dg(x)^T \Sigma Dg(x))p_x - Dh(x)^T p_y - Dg(x)^T(-\Sigma(g(x) - s) - z + \mu S^{-1}e) \\
= \quad & Dh(x)^T y + Dg(x)^T z \\
\\
\Longrightarrow \quad & (\xi Id + Dg(x)^T \Sigma Dg(x))p_x - Dh(x)^T p_y \\
= \quad & Dh(x)^T y + Dg(x)^T z + Dg(x)^T(-\Sigma(g(x) - s) - z + \mu S^{-1}e) \\
= \quad & Dh(x)^T y + Dg(c)^T(z - \Sigma(g(x) - s) - z + \mu S^{-1}e) \\
= \quad & Dh(x)^T y + Dg(c)^T(-\Sigma g(x) + S^{-1}Zs + \mu S^{-1}e) \\
= \quad & Dh(x)^T y + Dg(c)^T(z - \Sigma g(x) + \mu S^{-1}e)
\end{aligned}$$

Hence, the linear system of equations (3.5) can be solved by calculating p_x and p_y implicitly via

$$\begin{pmatrix} \xi Id + Dg(x)^T \Sigma Dg(x) & Dh(x)^T \\ Dh(x) & 0 \end{pmatrix} \begin{pmatrix} p_x \\ -p_y \end{pmatrix} = \begin{pmatrix} Dh(x)^T y + Dg(c)^T(z - \Sigma g(x) + \mu S^{-1}e) \\ -h(x) \end{pmatrix} \tag{3.27}$$

and p_s and p_z afterwards by using the explicit equations (3.26a) and (3.26b). Note that by inverting the sign of p_y, the matrix of (3.27) becomes symmetric.

The diagonal elements of Σ are given by

$$\Sigma = \begin{pmatrix} \Sigma_{u,1} & & & & & \\ & \ddots & & & & \\ & & \Sigma_{u,N} & & & \\ & & & \Sigma_{x,0} & & \\ & & & & \ddots & \\ & & & & & \Sigma_{x,N} \end{pmatrix} \tag{3.28}$$

with

$$\Sigma_{u,i} := \begin{pmatrix} \sigma_{u,i}^U & & \\ & \sigma_{u,i}^L & \\ & & \sigma_{u,i}^r \end{pmatrix} \quad \text{and} \quad \Sigma_{x,i} := \begin{pmatrix} \sigma_{x,i}^U & & \\ & \sigma_{x,i}^L & \\ & & \sigma_{x,i}^r \end{pmatrix}, \tag{3.29}$$

where $\sigma_{u,i}^U$ is a diagonal matrix containing the components of $S^{-1}Z$ which are associated with the inequality constraints of the upper boxed constraints of u_i, $\sigma_{u,i}^L$ for the lower boxed constraints and $\sigma_{u,i}^r$ for the nonlinear inequality constraints defined by $r_{u,i}$. The matrices $\sigma_{x,i}^U$, $\sigma_{x,i}^L$ and $\sigma_{x,i}^r$ are defined respectively.

Exploiting the structure of $Dg(x)$ as defined in (3.24b) it holds that

$$\xi Id + Dg(x)^T \Sigma Dg(x) = \begin{pmatrix} \sigma_{u,1} &&&&&&&& \\ & \ddots &&&&&&& \\ && \sigma_{u,N} &&&&&& \\ &&& \sigma_{x,0} &&&&& \\ &&&& \ddots &&&& \\ &&&&& \sigma_{x,N} &&& \\ &&&&&& \xi Id && \\ &&&&&&& \ddots & \\ &&&&&&&& \xi Id \end{pmatrix} \tag{3.30a}$$

with

$$\sigma_{u,i} := \xi Id + P_{u,i}^{UT} \sigma_{u,i}^{U} P_{u,i}^{U} + P_{u,i}^{LT} \sigma_{u,i}^{L} P_{u,i}^{L} + \dot{r}_{u,i}^{T} \sigma_{u,i}^{r} \dot{r}_{u,i} \tag{3.30b}$$

$$\sigma_{x,i} := \xi Id + P_{x,i}^{UT} \sigma_{x,i}^{U} P_{x,i}^{U} + P_{x,i}^{LT} \sigma_{x,i}^{L} P_{x,i}^{L} + \dot{r}_{x,i}^{T} \sigma_{x,i}^{r} \dot{r}_{x,i}. \tag{3.30c}$$

Note that $\sigma_{u,i}$ and $\sigma_{x,i}$ are not necessarily diagonal matrices if nonlinear constraints are used. On the other hand, if only box constraints are used, an operation like $P_{u,i}^{UT} \sigma_{u,i}^{U} P_{u,i}^{U}$ leads to diagonal matrices again. $\sigma_{u,i}^{U}$ can be smaller than the actual dimension of the control, as some of the control components might be unlimited. Hence, $P_{u,i}^{UT} \sigma_{u,i}^{U} P_{u,i}^{U}$ can have lower rank. Of course, the same facts apply to $\sigma_{x,i}$, too.

Besides that, $\sigma_{u,i}$ and $\sigma_{x,i}$ are positive definite and symmetric matrices in any case. The inner σ^U, σ^L and σ^r matrices are diagonal matrices with positive entries. The matrix products generate lower rank symmetric matrices with eigenvalues greater than or equal to zero. Adding ξId shifts the minimal eigenvalue to $\xi > 0$.

With these considerations, we can identify the individual non–zero components of the matrix used in the reduced linear system of equations (3.27). Figure 3.1 shows an exemplary setting with $N = 4$. One can easily decode the sparse structure for arbitrary values of N.

$$\begin{array}{c|cccccccccccccccccccccccccccccc}
 & 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 \\
\hline
1 & \sigma_{u,1} &&&&&&&&&&&&& \partial_u f_1^T &&&&&&&& \dot{q}_{u,1}^T &&&&&&&& \\
2 && \sigma_{u,2} &&&&&&&&&&&&&& \partial_u f_2^T &&&&&&& \dot{q}_{u,2}^T &&&&&&& \\
3 &&& \sigma_{u,3} &&&&&&&&&&&&&&& \partial_u f_3^T &&&&&& \dot{q}_{u,3}^T &&&&&& \\
4 &&&& \sigma_{u,4} &&&&&&&&&&&&&&&& \partial_u f_4^T &&&&& \dot{q}_{u,4}^T &&&&& \\
5 &&&&& \sigma_{x,0} &&&&&&&&& \partial_x f_1^T & Id &&&&&&&&&&& \dot{q}_{x,0}^T &&&& \\
6 &&&&&& \sigma_{x,1} &&&&&&&&& -Id & \partial_x f_2^T & Id &&&&&&&&&& \dot{q}_{x,1}^T &&& \\
7 &&&&&&& \sigma_{x,2} &&&&&&&&&& -Id & \partial_x f_3^T & Id &&&&&&&&& \dot{q}_{x,2}^T && \\
8 &&&&&&&& \sigma_{x,3} &&&&&&&&&&& -Id & \partial_x f_4^T & Id &&&&&&&& \dot{q}_{x,3}^T & \\
9 &&&&&&&&& \sigma_{x,4} &&&&&&&&&&&& -Id &&&&&&&&& \dot{q}_{x,4}^T \\
10 &&&&&&&&&& \xi Id &&&& \partial_x f_1^{AT} & B^T &&&&&&&&&&&&&&& \\
11 &&&&&&&&&&& \xi Id &&&&& \partial_x f_2^{AT} & B^T &&&&&&&&&&&&& \\
12 &&&&&&&&&&&& \xi Id &&&&&& \partial_x f_3^{AT} & B^T &&&&&&&&&&& \\
13 &&&&&&&&&&&&& \xi Id &&&&&&& \partial_x f_4^{AT} & B^T &&&&&&&&& \\
14 & \partial_u f_1 &&&& \partial_x f_1 &&&&& \partial_x f_1^A &&&&&&&&&&&&&&&&&&&& \\
15 &&&&& Id & -Id &&&& B &&&&&&&&&&&&&&&&&&&& \\
16 && \partial_u f_2 &&&& \partial_x f_2 &&&&& \partial_x f_2^A &&&&&&&&&&&&&&&&&&& \\
17 &&&&&& Id & -Id &&&& B &&&&&&&&&&&&&&&&&&& \\
18 &&& \partial_u f_3 &&&& \partial_x f_3 &&&&& \partial_x f_3^A &&&&&&&&&&&&&&&&&& \\
19 &&&&&&& Id & -Id &&&& B &&&&&&&&&&&&&&&&&& \\
20 &&&& \partial_u f_4 &&&& \partial_x f_4 &&&&& \partial_x f_4^A &&&&&&&&&&&&&&&&& \\
21 &&&&&&&& Id & -Id &&&& B &&&&&&&&&&&&&&&&& \\
22 & \dot{q}_{u,1} &&&&&&&&&&&&&&&&&&&&&&&&&&&&& \\
23 && \dot{q}_{u,2} &&&&&&&&&&&&&&&&&&&&&&&&&&&& \\
24 &&& \dot{q}_{u,3} &&&&&&&&&&&&&&&&&&&&&&&&&&& \\
25 &&&& \dot{q}_{u,4} &&&&&&&&&&&&&&&&&&&&&&&&&& \\
26 &&&&& \dot{q}_{x,0} &&&&&&&&&&&&&&&&&&&&&&&&& \\
27 &&&&&& \dot{q}_{x,1} &&&&&&&&&&&&&&&&&&&&&&&& \\
28 &&&&&&& \dot{q}_{x,2} &&&&&&&&&&&&&&&&&&&&&&& \\
29 &&&&&&&& \dot{q}_{x,3} &&&&&&&&&&&&&&&&&&&&&& \\
30 &&&&&&&&& \dot{q}_{x,4} &&&&&&&&&&&&&&&&&&&&&
\end{array}$$

Figure 3.1: Example matrix of (3.27) with $N = 4$

Chapter 4

Parallel direct methods on sparse matrices

In this chapter we will discuss an algorithm for solving symmetric sparse linear systems of equations like (3.27). The first difficult task is the choice of the algorithm, particularly whether using a direct or indirect method.

Of course, there is a serious discussion about the advantages and disadvantages of iterative and direct methods for sparse matrices. For our medium–sized matrices they balance each other. The authors of [27] had a closer look at direct methods and present a comprehensive overview of current software packages with different algorithms. They came to the conclusion, that although a perfect direct method for sparse matrices doesn't really exist, they (the direct methods) can pay off *if the individual problem is well suited.* By the name of this chapter one could have already guessed that I share their opinion.

In the first section of this chapter we will consider some important tools for handling sparse matrices. These tools help to answer the question whether a matrix is well suited for direct solvers, or not. The method of our choice will be the old–school Cholesky method which can be redesigned for good parallel SIMD performance and sparse structures. To our surprise, after some reordering this will be the only required method, even though (3.27) is indefinite.[1]

[1]We will see in the next chapter that solving the linear system of equations (3.27) will result in solving multiple smaller positive definite symmetric systems.

4.1 The basics

A very good book in the context of direct methods for sparse matrices has been written by *Duff*, *Erisman* and *Reid* [7]. This section will pick up their ideas and describe the concepts needed in this work.

4.1.1 Dealing with fill–ins

While iterative methods only modify the iteration vector, direct methods modify elements of the matrix. If things go wrong, zeros of the matrix will be destroyed during elimination steps and generate *fill–ins*. These fill–ins are additional data values, which need further memory. The memory must be allocated and assigned to the matrix element's position on–the–fly. In the worst case, every zero element is eliminated by fill–ins and the matrix becomes dense in the end.

The amount and position of fill–ins can be manipulated by an a priori permutation of the matrix. For example, the following matrix will result in a dense decomposition after applying Cholesky's method[2]:

$$\begin{pmatrix} * & * & * & * & * \\ * & * & 0 & 0 & 0 \\ * & 0 & * & 0 & 0 \\ * & 0 & 0 & * & 0 \\ * & 0 & 0 & 0 & * \end{pmatrix} \Longrightarrow \begin{pmatrix} * & 0 & 0 & 0 & 0 \\ * & * & 0 & 0 & 0 \\ * & * & * & 0 & 0 \\ * & * & * & * & 0 \\ * & * & * & * & * \end{pmatrix} \tag{4.1}$$

Simply permuting the first row to the bottom and the left column to the right prevents every

[2]We will discuss details of Cholesky's method in Section 4.2.

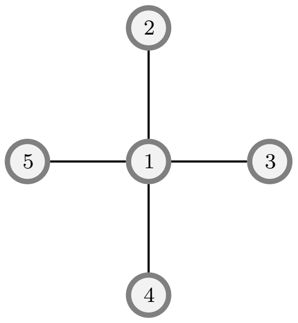
zero from being overwritten:

$$\begin{pmatrix} * & 0 & 0 & 0 & * \\ 0 & * & 0 & 0 & * \\ 0 & 0 & * & 0 & * \\ 0 & 0 & 0 & * & * \\ * & * & * & * & * \end{pmatrix} \Longrightarrow \begin{pmatrix} * & 0 & 0 & 0 & 0 \\ 0 & * & 0 & 0 & 0 \\ 0 & 0 & * & 0 & 0 \\ 0 & 0 & 0 & * & 0 \\ * & * & * & * & * \end{pmatrix} \tag{4.2}$$

The structure of the last matrix (before being decomposed) is called *doubly–bordered block diagonal form* and has the nice property that Gaussian and Cholesky elimination steps do not modify any zero elements.

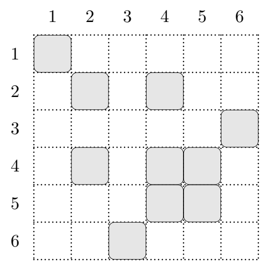
4.1.2 Matrix graphs

A very important tool for analyzing squared matrix structures are matrix graphs. These graphs consist of one node for each row containing the number of the row. If the matrix has an element $a_{i,j} \neq 0$ then the two nodes with the numbers i and j are connected. Since the matrix is symmetric, the connections have no direction. The graph of the first matrix (4.1) looks like this:

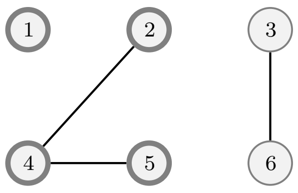


Diagonal elements cause connections of nodes with themselves. We illustrate this case via bold borders of graph nodes. In the case above, every diagonal element is non–zero.

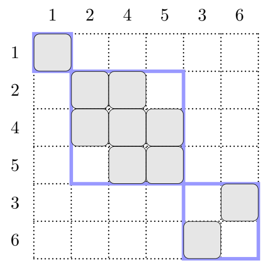
One feature of matrix graphs is that they give information about possible diagonal block submatrices, even if they are not visible at a first glance. For example, the matrix



generates the matrix graph



with three unconnected subgraphs. We can now perform symmetrical permutations on the matrix so that we reorder the rows/columns in a way that the nodes (which are equivalent to rows/columns) of each subgraph are grouped together in the matrix. The result is a matrix with three independent block diagonal matrices (rows 1, 2-4, 5-6):



4.1.3 Nested dissection

The idea of the *(one–way–)dissection* is based on the fact that a linear system of equations can easily be solved if it has doubly–bordered block diagonal form. If a linear system of equations is defined as

$$\begin{pmatrix} A_{1,1} & & & A_{1,N} \\ & \ddots & & \vdots \\ & & A_{N-1,N-1} & A_{N-1,N} \\ A_{1,N}^T & \cdots & A_{N-1,N}^T & A_{N,N} \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_{N-1} \\ x_N \end{pmatrix} = \begin{pmatrix} b_1 \\ \vdots \\ b_{N-1} \\ b_N \end{pmatrix} \tag{4.3}$$

then $(x_1, \ldots, x_N)^T$ can be calculated via Gaussian elimination steps

$$\begin{aligned}
D &= A_{N,N} - \sum_{i=1}^{N-1} A_{i,N}^T A_{i,i}^{-1} A_{i,N} && \text{(4.4a)} \\
c &= b_N - \sum_{i=1}^{N-1} A_{i,N}^T A_{i,i}^{-1} b_i && \text{(4.4b)} \\
x_N &= D^{-1} c && \text{(4.4c)} \\
x_i &= A_{i,i}^{-1} \left(b_i - A_{i,N} x_N \right), \quad i = 1, \ldots, N-1 && \text{(4.4d)}
\end{aligned}$$

where D and c are temporary matrices (see [7]). The matrix inversions in (4.4) can be substituted by solving smaller linear systems of equations with temporary matrices Q_i and R_i:

$$\begin{aligned}
\text{solve} \quad A_{i,i} Q_i &= A_{i,N}, \quad i = 1, \ldots, N-1 && \text{(4.5a)} \\
\text{solve} \quad A_{i,i} R_i &= b_i, \quad i = 1, \ldots, N-1 && \text{(4.5b)} \\
D &= A_{N,N} - \sum_{i=1}^{N-1} A_{i,N}^T Q_i && \text{(4.5c)} \\
c &= b_N - \sum_{i=1}^{N-1} A_{i,N}^T R_i && \text{(4.5d)} \\
\text{solve} \quad D x_N &= c && \text{(4.5e)} \\
x_i &= R_i - Q_i x_N, \quad i = 1, \ldots, N-1 && \text{(4.5f)}
\end{aligned}$$

Since (4.5a) and (4.5b) can make use of the same matrix decomposition, we replace those systems by a single linear system of equations

$$\text{solve} \quad A_{i,i} \left(\begin{array}{c|c} Q_i & R_i \end{array} \right) = \left(\begin{array}{c|c} A_{i,N} & b_i \end{array} \right), \quad i = 1, \ldots, N-1 \qquad \text{(4.5g)}$$

where the notation $\left(\begin{array}{c|c} A & B \end{array} \right)$ indicates a matrix which is concatenated from A and B.

Fortunately, the $N-1$ linear systems of equations (4.5g) and also (4.5f) can be evaluated in parallel since the results do not depend on each other. The evaluation of (4.5e) is a *parallel reduction task* and must be done serially. A matrix with a small $A_{N,N}$ block (the so called *separator block*) needs less serial computational time – something we should keep in mind.

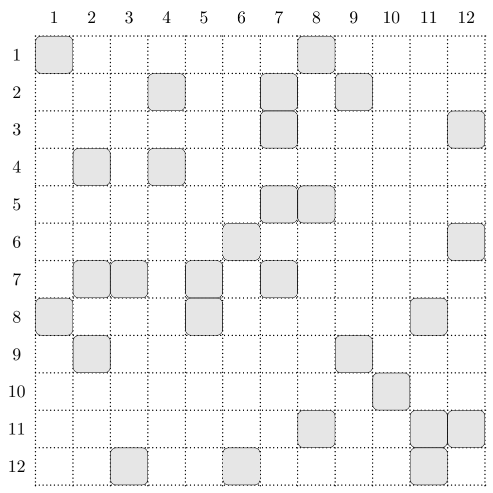
We can use matrix graphs to develop permutations which transform matrices to doubly–bordered block diagonal forms. As shown in Section 4.1.1, unconnected subgraphs of a matrix graph make it possible to permute the matrix to independent diagonal blocks $A_{i,i}$. These subgraphs can be generated by “removing” individual graph nodes. “Re–appending” the corresponding rows/columns of the removed nodes to the lower right end of the matrix results in the desired double–borders. The removed nodes are associated with the lower right block $A_{N,N}$. Since they separate the graph in unconnected parts it becomes clear why $A_{N,N}$ is called separator block.

Figure 4.1 shows an example matrix and the permuted version, which has the intended form. The permutation is described by Figure 4.2. The matrix graph turns out to be quite regular. Removing the three nodes 5, 7 and 3 results in three unconnected subgraphs. The permutation is defined by clustering the rows/columns of each subgraph consecutively followed by the three removed nodes.

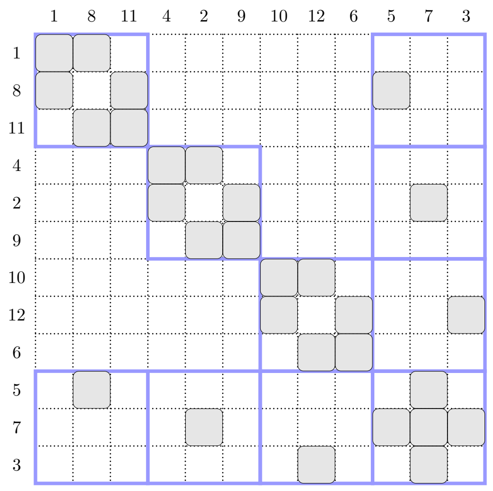
The concept of *nested dissection* is based on the idea that the smaller linear systems of equation (4.5g), in turn, can be solved by doing a dissection on each subgraph. As a result, each diagonal block will also have doubly–bordered block diagonal form. Figure 4.3 demonstrates the second dissection step of each of the diagonal blocks. In a parallel implementation, each of the dark blue highlighted diagonal blocks can be solved in parallel, which means a parallel bandwidth of 6 in this case. The linear system of equations (4.5e) of the separator blocks has to be solved for each diagonal block of the first dissection with a parallel bandwidth of 3 linear systems of equations.

Note that choosing the separator nodes is a very sensitive process. In some sense it corresponds to a pivoting strategy which, of course, can fail when done wrong. Figure 4.4 demonstrates how a bad dissection can cause singular diagonal blocks, even if the original matrix is regular.

Particularly, for larger matrices one is interested in doing the dissection automatically. A short summary of current algorithms is given in [19]. Unfortunately, most of these works only consider graphs which come from numerical solutions of partial differential equations, like [8, 9] by *Alan George*, who originally proposed the nested dissection algorithm. I have not found any work that considers singularity problems which arise due to a bad dissection.



(a) original matrix

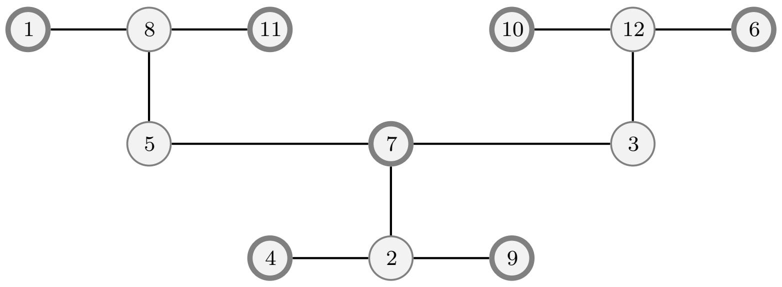


(b) permuted matrix

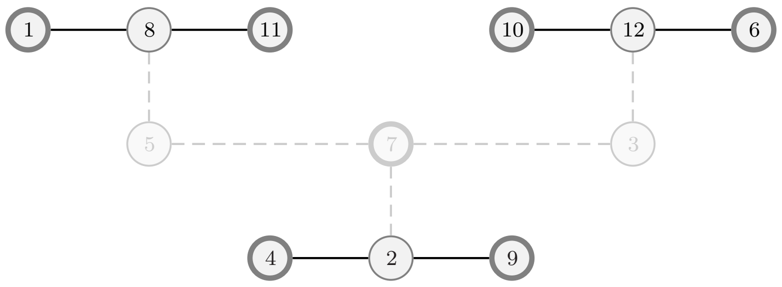
Figure 4.1: Permutation of an example matrix to doubly–bordered block diagonal form

This is why we will not make use of automatic dissection algorithms in this work. Since the considered matrices are not very big and have a regular structure, the dissection can easily be done manually with a pen and a sheet of paper. The manual dissection has some important advantages:

- The nested dissection can be chosen with respect to the later goal of a parallel implementation on CUDA hardware. In particular, we focus on achieving similar diagonal blocks.



(a) original matrix graph



(b) Removing three nodes results in three unconnected subgraphs

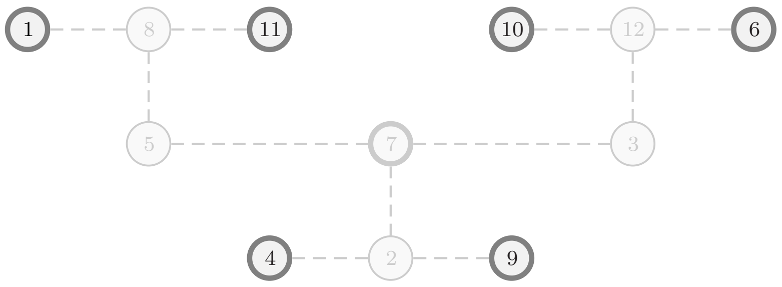
Figure 4.2: Modifications to the matrix graph of Figure 4.1 to achieve a doubly–bordered block diagonal form

- Bad dissections with singular diagonal blocks can be avoided.
- If we know the individual elements of the sparse matrix, the elimination steps can be developed analytically without the need of additional data. A general nested dissection solver needs more or less complex data structures like trees to store the information of the different dissection stages. Trees have a quite ugly and irregular data format in terms of GPU computation (see section 1.2.3).

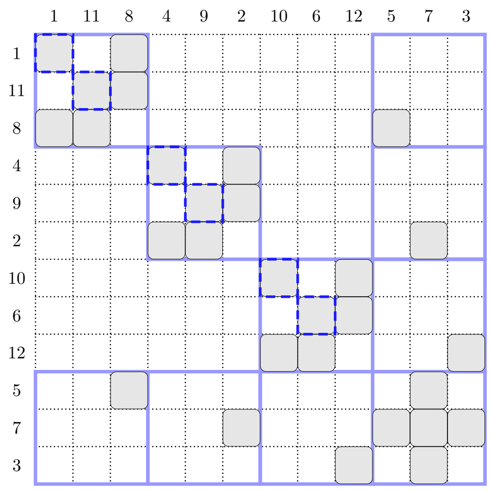
4.1.4 Suitable matrices for direct methods

What is still left is the question whether a matrix is well suited for direct methods or not. This strongly depends on the possibility of choosing a good dissection strategy. The following list is a summary of criteria which boost the effiency of direct methods with nested dissection. The criteria are ordered descending by their importance.

1. Actually, needless to say: $A_{1,1}, \ldots, A_{N-1,N-1}$ must be invertible.



(a) Second dissection stage: Nodes 8, 2 and 12 are removed



(b) Matrix of figure 4.1 permuted by nested dissection,
the diagonal blocks of the second dissection stage are highlighted.

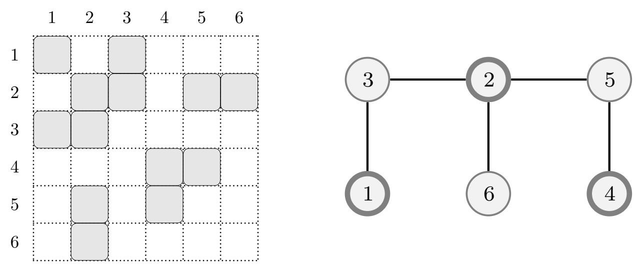
Figure 4.3: Two–way nested dissection of an example matrix

2. The matrices $A_{1,1}, \ldots, A_{N,N}$ must be small since they lead to linear systems of equations with cubic complexity. Fill–ins only appear within these block diagonal matrices.

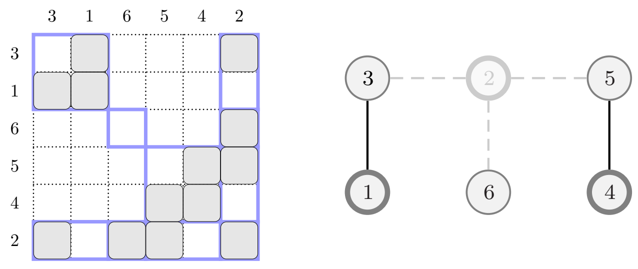
3. A small separator block $A_{N,N}$ reduces the serial overhead during parallel processing.

4. If $A_{1,1}, \ldots, A_{N-1,N-1}$ are positive definite we can use a fast Cholesky implementation for solving (4.5g).

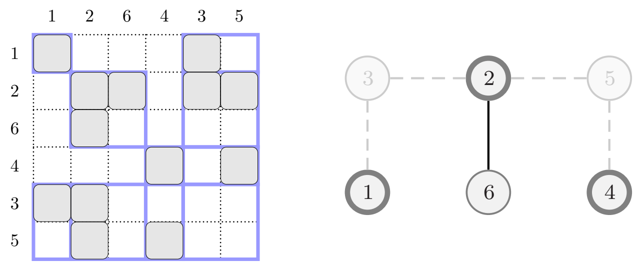
5. In addition to the 4th criterion, if $A_{N,N} = 0$, we can use Cholesky's method to solve (4.5e) after a little modification. Since $A_{i,i}^{-1}$, $i = 1, \ldots, N-1$, is also symmetric and



(a) Original matrix with matrix graph



(b) Bad dissection with singular diagonal block $A_{2,2}$



(c) Good dissection strategy

Figure 4.4: Example of a bad dissection strategy, which leads to a singular diagonal block

positive definite, a Cholesky decomposition $A_{i,i}^{-1} = RR^T$ exists and it holds that

$$D = -\sum_{i=1}^{N-1} A_{i,N}^T A_{i,i}^{-1} A_{i,N} = -\sum_{i=1}^{N-1} A_{i,N}^T R R^T A_{i,N} = -\sum_{i=1}^{N-1} A_{i,N}^T R \left(A_{i,N}^T R\right)^T . \tag{4.6}$$

Hence, D is a negative sum of symmetric positive definite matrices. By multiplying (4.5e) by minus one, the linear system of equations

$$-Dx_N = -c \tag{4.7}$$

becomes positive definite and can be solved by Cholesky's method. We achieve $A_{N,N} =$

0 by not choosing diagonal nodes as separator block.

If all these criteria are satisfied, the matrix is highly suitable for direct methods. We will see in the next chapter that our matrix (3.27) fulfills all these requirements.

4.2 Parallelizing Cholesky's method

A common implementation of Cholesky's method for $A \in \mathbb{R}^{n \times n}$ is shown in Algorithm 4.1. It stores the values of $R \in \mathbb{R}^{n \times n}$ with $A = RR^T$ in the lower triangular elements of A. The loops of the algorithm are designed to run serially, processing the elements one by one.

Algorithm 4.1: A serial implementation of Cholesky's method

```
 1: function CHOLESKY($A$)                                    ▷ $A \in \mathbb{R}^{n \times n}$
 2:     for $i \leftarrow 1, \dots, n$ do
 3:         for $j \leftarrow 1, \dots, i$ do
 4:             $temp \leftarrow A_{i,j}$
 5:             for $k \leftarrow 1, \dots, j-1$ do
 6:                 $temp \leftarrow temp - A_{i,k}A_{j,k}$
 7:             end for
 8:             if $i > j$ then
 9:                 $A_{i,j} \leftarrow temp/A_{j,j}$
10:             else if $temp > 0$ then
11:                 $A_{i,i} \leftarrow \sqrt{temp}$
12:             else
13:                 return failed
14:             end if
15:         end for
16:     end for
17:     return success
18: end function
```

In this section we want to develop an alternative implementation, which performs certain operations in parallel, although it does exactly the same from an analytical point of view.

This can be achieved by systematically reordering individual instructions.

4.2.1 The concept of dependency graphs

As a tool for systematic reordering of instructions we use dependency graphs. They will help us to identify instructions which actually do not depend on each other's results and therefore can be executed in parallel.

The idea is to assign one graph node to each instruction. If an instruction B directly needs the results of another instruction A, an edge pointing from node A to node B is inserted. This indicates the flow of data between both instructions:

$$\begin{tikzcd} \boxed{A} \arrow[r] & \boxed{B} \end{tikzcd}$$

The fully generated graph contains a set of nodes N_0 from which all edges point away since an algorithm must have an entry point. The corresponding instructions can be executed in parallel as they do not need any results from other nodes. We remove the nodes of N_0 from the graph and continue searching for the next set of nodes N_1 from which all edges point away. The execution of every node of N_1 obviously depends on the result of N_0, otherwise these single independent nodes would have already been selected by N_0.

The series $N_0, N_1, \ldots$ defines a new graph containing N_i as supernodes with simultaneously executable instructions. Figure 4.5 demonstrates the construction of N_i via a series of originally executed instructions $A, \ldots, I$.

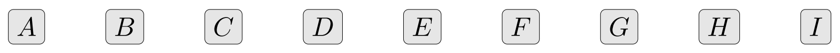
Note that this strategy is very trivial and, of course, does not replace the extensive research about parallel implementation of algorithms currently happening. A simple example which is not fully covered by this easy approach is a sum of elements

$$a = \sum_{i=1}^{4} b_i$$

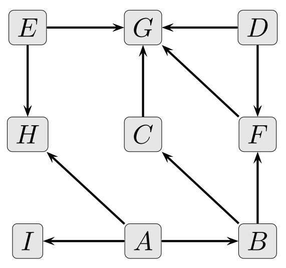
which naively comes with a series of instructions

$$a \leftarrow 0, \quad a \leftarrow a + b_1, \quad a \leftarrow a + b_2, \quad a \leftarrow a + b_3, \quad a \leftarrow a + b_4.$$

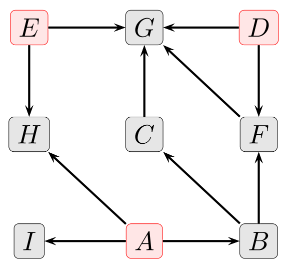
These instructions depend on each other, as each instruction needs the value of a which has



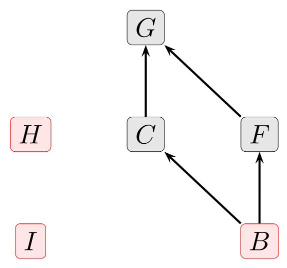
(a) Serial sequence of instructions



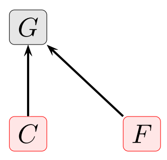
(b) Example dependencies



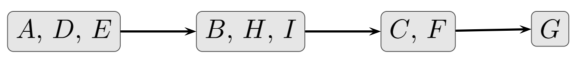
(c) $N_0 = \{A, D, E\}$



(d) $N_1 = \{B, H, I\}$



(e) $N_2 = \{C, F\}$, $N_3 = \{G\}$



(f) Resulting graph with supernodes containing parallel instructions

Figure 4.5: Example parallelization via dependency graph

been altered by the previous instruction. Strangely, the instructions can be reordered, e.g. to

$$a \leftarrow 0, \quad a \leftarrow a + b_1, \quad a \leftarrow a + b_3, \quad a \leftarrow a + b_2, \quad a \leftarrow a + b_4,$$

without changing the final result. This makes us regarding this situation with suspicion. And just to carry it to extremes: One could introduce a temporary variable c and achieve the same result by calculating

$$\begin{matrix} a \leftarrow 0, & a \leftarrow a + b_1, & a \leftarrow a + b_2, \\ c \leftarrow 0, & c \leftarrow c + b_3, & c \leftarrow c + b_4, \end{matrix} \quad a \leftarrow a + c$$

on two parallel program streams.

Obviously, there is a lot more about parallelization which can not be handled efficiently with this simple dependency graph approach. The interested reader is refered to [11], which considers a lot more parallelization techniques in detail. Fortunately, in case of Cholesky's method analyzing the dependency graph helps a lot.

4.2.2 Instruction reordering

We analyze Cholesky's method by simulating the computation steps with $n = 4$ to receive an impression about what actually happens. The resulting 20 instructions of the algorithm are shown in Figure 4.6. The corresponding dependency graph is shown in Figure 4.7. By successively removing the sets of nodes N_0, N_1,... the parallely executable instructions grouped by supernodes can be identified. Figure 4.8 shows the reordered graph.

1. $A_{1,1} \leftarrow \sqrt{A_{1,1}}$
2. $A_{2,1} \leftarrow A_{2,1}/A_{1,1}$
3. $A_{2,2} \leftarrow A_{2,2} - A_{2,1}A_{2,1}$
4. $A_{2,2} \leftarrow \sqrt{A_{2,2}}$
5. $A_{3,1} \leftarrow A_{3,1}/A_{1,1}$
6. $A_{3,2} \leftarrow A_{3,2} - A_{3,1}A_{2,1}$
7. $A_{3,2} \leftarrow A_{3,2}/A_{2,2}$
8. $A_{3,3} \leftarrow A_{3,3} - A_{3,1}A_{3,1}$
9. $A_{3,3} \leftarrow A_{3,3} - A_{3,2}A_{3,2}$
10. $A_{3,3} \leftarrow \sqrt{A_{3,3}}$
11. $A_{4,1} \leftarrow A_{4,1}/A_{1,1}$
12. $A_{4,2} \leftarrow A_{4,2} - A_{4,1}A_{2,1}$
13. $A_{4,2} \leftarrow A_{4,2}/A_{2,2}$
14. $A_{4,3} \leftarrow A_{4,3} - A_{4,1}A_{3,1}$
15. $A_{4,3} \leftarrow A_{4,3} - A_{4,2}A_{3,2}$
16. $A_{4,3} \leftarrow A_{4,3}/A_{3,3}$
17. $A_{4,4} \leftarrow A_{4,4} - A_{4,1}A_{4,1}$
18. $A_{4,4} \leftarrow A_{4,4} - A_{4,2}A_{4,2}$
19. $A_{4,4} \leftarrow A_{4,4} - A_{4,3}A_{4,3}$
20. $A_{4,4} \leftarrow \sqrt{A_{4,4}}$

Figure 4.6: Instructions of Cholesky's method with $n = 4$

Now we can develop a parallel variant of Cholesky's method by analyzing the regularity of the reordered graph. Obviously, there's a loop which serially counts from 1 to n. Every step consists of three different types of supernodes. A single supernode contains equal instructions which are processed on different data indices, which makes it possible to run the whole algorithm on CUDA hardware. A closer look at the indices of the supernodes reveals a parallel implementation of Cholesky's method for arbitrary values n, described by Algorithm 4.2.

Algorithm 4.2: A parallel implementation of Cholesky's method

```
1: function ParallelCholesky($A$)                ▷ $A \in \mathbb{R}^{n \times n}$
2:     for $i \leftarrow 1, \ldots, n$ do
```

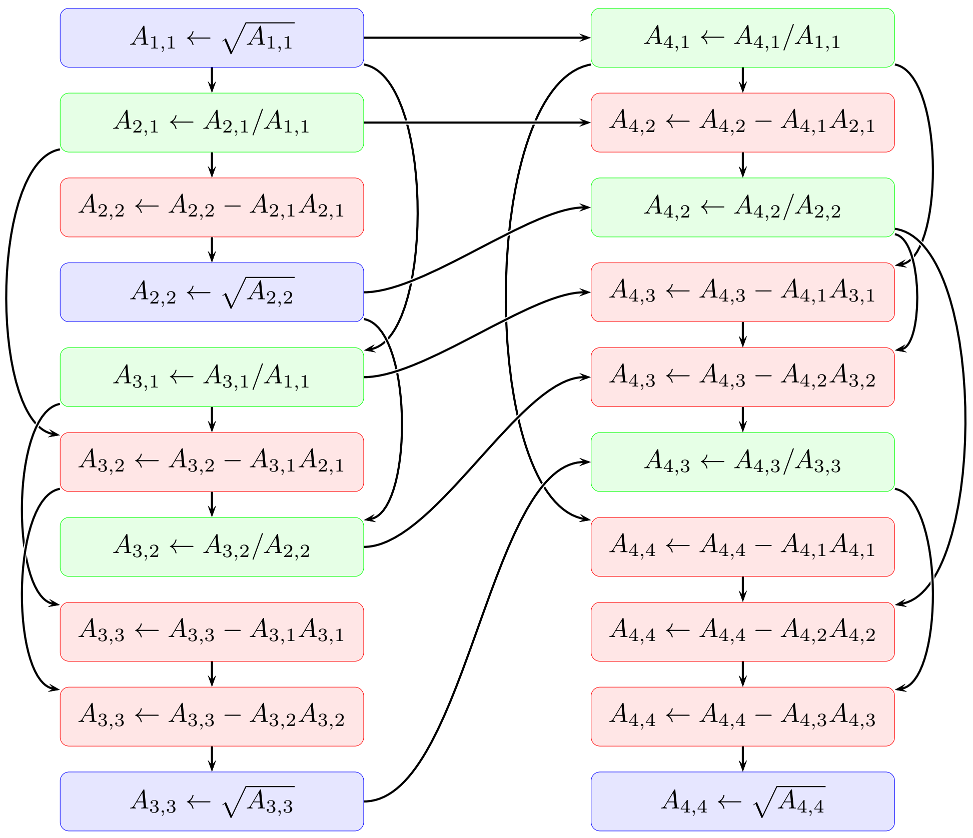

$A_{1,1} \leftarrow \sqrt{A_{1,1}}$
$A_{2,1} \leftarrow A_{2,1}/A_{1,1}$
$A_{2,2} \leftarrow A_{2,2} - A_{2,1}A_{2,1}$
$A_{2,2} \leftarrow \sqrt{A_{2,2}}$
$A_{3,1} \leftarrow A_{3,1}/A_{1,1}$
$A_{3,2} \leftarrow A_{3,2} - A_{3,1}A_{2,1}$
$A_{3,2} \leftarrow A_{3,2}/A_{2,2}$
$A_{3,3} \leftarrow A_{3,3} - A_{3,1}A_{3,1}$
$A_{3,3} \leftarrow A_{3,3} - A_{3,2}A_{3,2}$
$A_{3,3} \leftarrow \sqrt{A_{3,3}}$
$A_{4,1} \leftarrow A_{4,1}/A_{1,1}$
$A_{4,2} \leftarrow A_{4,2} - A_{4,1}A_{2,1}$
$A_{4,2} \leftarrow A_{4,2}/A_{2,2}$
$A_{4,3} \leftarrow A_{4,3} - A_{4,1}A_{3,1}$
$A_{4,3} \leftarrow A_{4,3} - A_{4,2}A_{3,2}$
$A_{4,3} \leftarrow A_{4,3}/A_{3,3}$
$A_{4,4} \leftarrow A_{4,4} - A_{4,1}A_{4,1}$
$A_{4,4} \leftarrow A_{4,4} - A_{4,2}A_{4,2}$
$A_{4,4} \leftarrow A_{4,4} - A_{4,3}A_{4,3}$
$A_{4,4} \leftarrow \sqrt{A_{4,4}}$

Figure 4.7: Dependency graph of Cholesky's method with $n = 4$

```
 3:         **if** $A_{i,i} \leq 0$ **then**
 4:             **return** failed
 5:         **end if**
 6:         $A_{i,i} \leftarrow \sqrt{A_{i,i}}$
 7:         **for all** $j \in \{i+1, \ldots, n\}$ **do**                          ▷ parallel loop
 8:             $A_{j,i} \leftarrow A_{j,i}/A_{i,i}$
 9:         **end for**
10:         **for all** $j \in \{i+1, \ldots, n\},\ k \in \{i+1, \ldots, j\}$ **do**    ▷ parallel loop
11:             $A_{j,k} \leftarrow A_{j,k} - A_{j,i}A_{k,i}$
12:         **end for**
13:     **end for**
14:     **return** success
15: **end function**
```

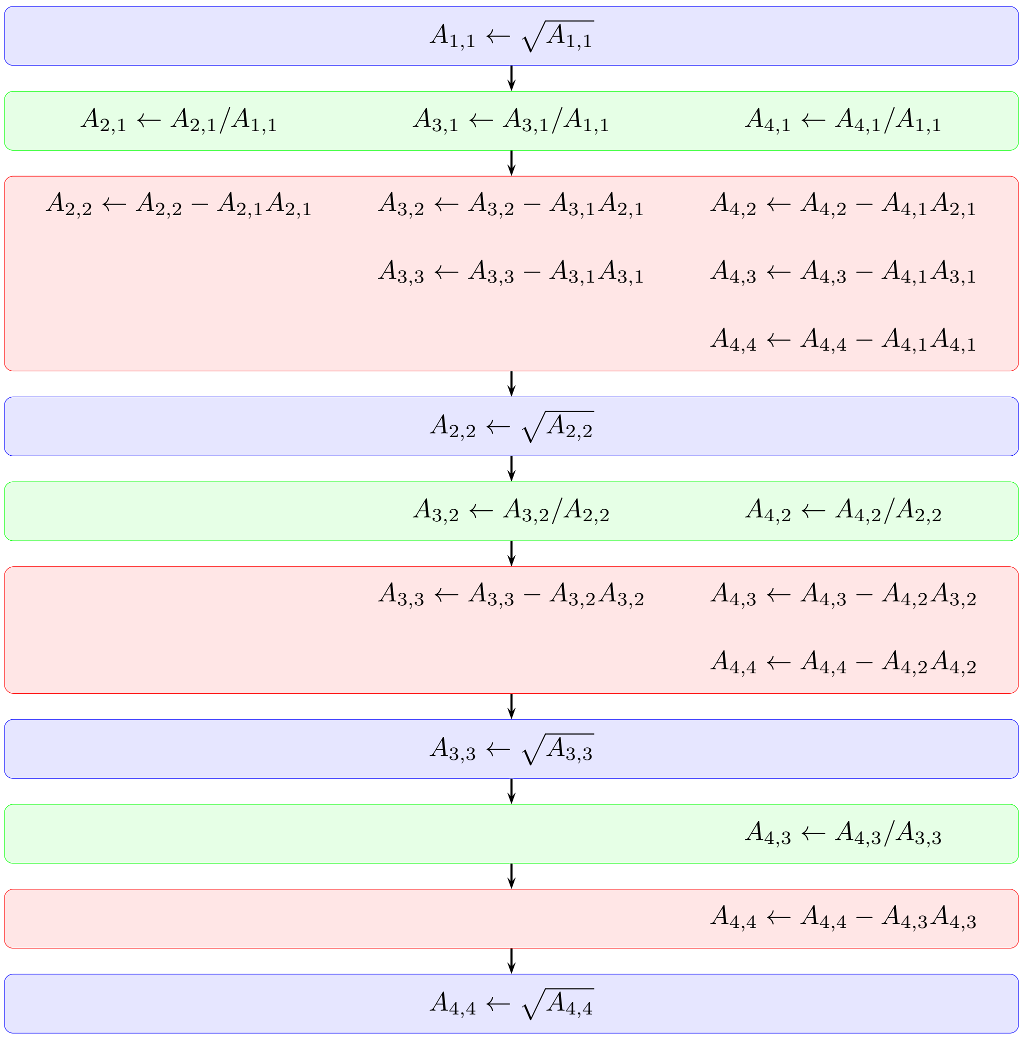

$A_{1,1} \leftarrow \sqrt{A_{1,1}}$
$A_{2,1} \leftarrow A_{2,1}/A_{1,1}$
$A_{3,1} \leftarrow A_{3,1}/A_{1,1}$
$A_{4,1} \leftarrow A_{4,1}/A_{1,1}$
$A_{2,2} \leftarrow A_{2,2} - A_{2,1}A_{2,1}$
$A_{3,2} \leftarrow A_{3,2} - A_{3,1}A_{2,1}$
$A_{4,2} \leftarrow A_{4,2} - A_{4,1}A_{2,1}$
$A_{3,3} \leftarrow A_{3,3} - A_{3,1}A_{3,1}$
$A_{4,3} \leftarrow A_{4,3} - A_{4,1}A_{3,1}$
$A_{4,4} \leftarrow A_{4,4} - A_{4,1}A_{4,1}$
$A_{2,2} \leftarrow \sqrt{A_{2,2}}$
$A_{3,2} \leftarrow A_{3,2}/A_{2,2}$
$A_{4,2} \leftarrow A_{4,2}/A_{2,2}$
$A_{3,3} \leftarrow A_{3,3} - A_{3,2}A_{3,2}$
$A_{4,3} \leftarrow A_{4,3} - A_{4,2}A_{3,2}$
$A_{4,4} \leftarrow A_{4,4} - A_{4,2}A_{4,2}$
$A_{3,3} \leftarrow \sqrt{A_{3,3}}$
$A_{4,3} \leftarrow A_{4,3}/A_{3,3}$
$A_{4,4} \leftarrow A_{4,4} - A_{4,3}A_{4,3}$
$A_{4,4} \leftarrow \sqrt{A_{4,4}}$

Figure 4.8: Supernodes with parallel instructions of Cholesky's method with $n = 4$

4.2.3 Cherry on the cake: A sparse parallel variant with fill–in reduction

The principle of a sparse variant of Cholesky's method is to ignore instructions which actually do not do anything. Instructions of the type

$$A_{j,i} \leftarrow A_{j,i}/A_{i,i} \tag{4.8}$$

can be ignored, if $A_{j,i}$ is a zero element. Besides, we do not need to process an elimination instruction

$$A_{j,k} \leftarrow A_{j,k} - A_{j,i}A_{k,i}, \tag{4.9}$$

if $A_{j,i} = 0$ or $A_{k,i} = 0$. One has to pay attention to fill–ins, which can appear during this step, if $A_{j,k}$ originally is a zero element, but $A_{j,i}A_{k,i} \neq 0$. After a fill–in at $A_{j,k}$ happened, this element has to be considered as a non–zero element for the rest of the algorithm. In practice, a hash table can be used to keep track of all non–zero elements.

If the decomposition is frequently used with a constant matrix structure but different values (like in the case of our interior point implementation), it is worth to generate an instruction scheduler a priori. This scheduler is defined by a series of sets

$$V_1, \ldots, V_{n-1} \quad \text{and} \quad W_1, \ldots, W_{n-1} \tag{4.10}$$

where V_i contains the indices j of all relevant instructions (4.8), and W_i contains the index–tuples (j, k) of all relevant instructions (4.9) in the i–th step. A scheduler can easily be generated by a simple "dry–run" of Cholesky's method. Instead of modifying the matrix A, we collect and store the relevant indices in V_i and W_i and update a temporary hashtable containing the non–zero elements if necessary.

A scheduler also works perfectly with sparse data structures like rows–, columns– and values–arrays. V_i and W_i can then simply hold the position of the elements within these arrays. A fill–in element is easily handled by appending the position to the rows– and columns–array and a zero to the values–array. In this case, one should use a map instead of a hash. It should store pairs of $(i, j) \to l$, where l is the index of the element $A_{i,j}$ within the sparse data structures.

To reduce the number of fill–ins we use an approach called *Approximate Minimum Degree Ordering Algorithm* [1, 28] which performs an intelligent preordering of the matrix. The concept of this approach becomes clear when considering the first parallel elimination step in Figure 4.8 which consists of

$$\begin{aligned} A_{2,2} \leftarrow A_{2,2} - A_{2,1}A_{2,1}, \quad & A_{3,2} \leftarrow A_{3,2} - A_{3,1}A_{2,1}, \quad & A_{4,2} \leftarrow A_{4,2} - A_{4,1}A_{2,1}, \\ & A_{3,3} \leftarrow A_{3,3} - A_{3,1}A_{3,1}, \quad & A_{4,3} \leftarrow A_{4,3} - A_{4,1}A_{3,1}, \\ & & A_{4,4} \leftarrow A_{4,4} - A_{4,1}A_{4,1}. \end{aligned} \tag{4.11}$$

All the factors $A_{j,1}$ and $A_{k,1}$ are represented by edges which are connected to the first node. A fill–in appears, if $A_{j,k}$ is originally zero. This means nodes j and k are not connected as neighbors. If we choose a permutation which brings the node with the smallest degree to the first row/column, most of the products $A_{j,1}A_{k,1}$ will be zero. Hence, the probability of producing non–zero products and, in particular, the probability of producing fill–ins during the first elimination step is reduced significantly.

For the rest of the runtime of Cholesky's method the first node will not be used any more, so we remove it from the graph and repeat. By doing this, a permuation is successively generated.

The crux of the matter is, that removing elimination instructions will reduce the parallelization bandwidth (actually $\mathcal{O}(n^2)$) significantly. On very sparse matrices with $\mathcal{O}(n)$ elements the complexity of the supernodes with elimination instructions even drops to $\mathcal{O}(1)$. For this reason, the sparse parallel variant of Cholesky's method will perform best on "not–so–dense" matrices with $\mathcal{O}(n^2)$ elements but a lot of zeros which ought to be recognized.

Chapter 5

Parallel search direction computation

In Section 3.2 we have found a problem definition which leads to a sparse linear system of equations (3.27). The evaluation of the individual submatrices of this system can be done with a parallel bandwidth of $\mathcal{O}(N)$, where N is the length of the discretization horizon, as $f_{i,j}$, $\partial_u f_i$, $\partial_x f_i$ and the rest of the matrix content can be computed independently.

We will now try to find a permutation in such a way that...

a) ... most of the computation while solving the linear system of equations can be done in parallel,

b) ... we make use of the sparse structure of the matrix with reduced fill–ins,

c) ... all matrix inversions of subsystems can be done by Cholesky's method.

The permutation will be constructed with the help of the tools which have been introduced in Chapter 4.

5.1 Permuting the search direction matrix

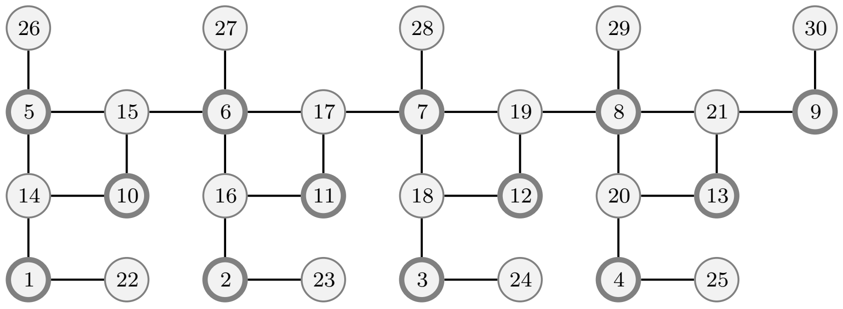
First, we construct the matrix graph based on the example matrix with $N = 4$ on page 40. The full graph is shown by Figure 5.1a. Due to the problem definition the graph has a nice regular structure (except for nodes 9 and 30). This will make it easy to divide the graph into similarly structured blocks in order to satisfy the requirements of CUDA hardware. The first dissection step can be performed either by eliminating nodes (5, 6, 7, 8, 9) or (15, 17, 19, 21). Since we aim for using Cholesky's method in the end, we take (15, 17, 19, 21) as first separator nodes, because they do *not* represent diagonal elements. Hence, this choice of the separator block keeps positive eigenvalues at the block diagonal matrices. The separator block itself is a zero matrix, as the separator nodes are not connected in the original graph. Thus, Cholesky's method can be used to solve the separator block system (4.5e) of the first dissection (see Section 4.1.4).

In the second dissection step we choose (14, 16, 18 20) as separator nodes, which in turn leads to zero–matrices as separator blocks and divides the subgraphs of the first dissection step into almost equally sized subgraphs. As in the first dissection step, we can use Cholesky's method to solve (4.5e).

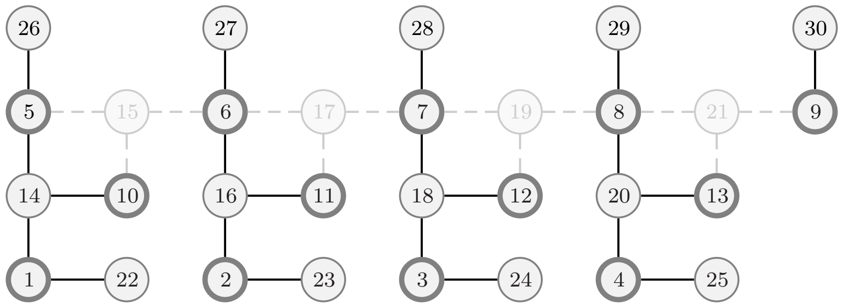
The third and last dissection step is done by eliminating (22,...,30). Doing this is a bit risky, since it does not divide a subgraph into nice diagonal blocks and, hence, will produce some calculation overhead. On the other hand, it will lead to a lot of independent, symmetric and positive definite diagonal block matrices[1] which can all be processed simultaneously by Cholesky's method. The last dissection step will merely permute $\dot{q}_{u,i}$ and $\dot{q}_{x,i}$ out to the double–borders. The sizes of the separator blocks are given by the number of nonlinear equality constraints. If we assume that we do not have "so much" nonlinear equality constrainst per horizon step, it is also safe to assume that the third dissection step will pay off with respect to computational effort. Again, we generate zero matrices as separator blocks, so (4.5e) of the third dissection step can be solved with Cholesky's method.

The fully permuted matrix is shown in Figure 5.2. First and second dissection steps are indicated by solid light blue and dashed dark blue frames. One can easily decode the regularity of the individual indices for arbitrary values of N. Since all block diagonal matrices have the same structure (except the block with $\sigma_{x,4}$) but different values, the whole decomposition is well suited for SIMD architecture like CUDA hardware.

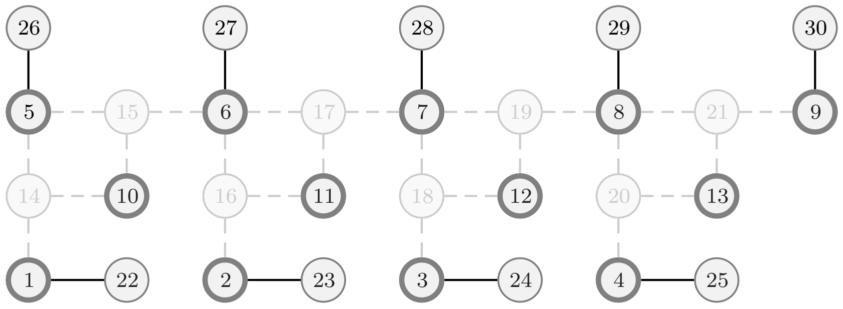
[1]The remaining diagonal nodes are associated with $\sigma_{u,i}$, $\sigma_{x,i}$ and ξId which are symmetric and positive definite as stated in Section 3.2.2.



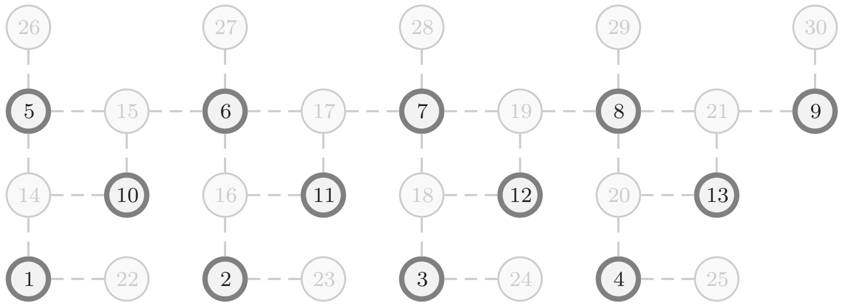
(a) Full matrix graph



(b) First dissection step



(c) Second dissection step



(d) Third dissection step

Figure 5.1: Matrix graph and nested dissection of (3.27) with $N = 4$

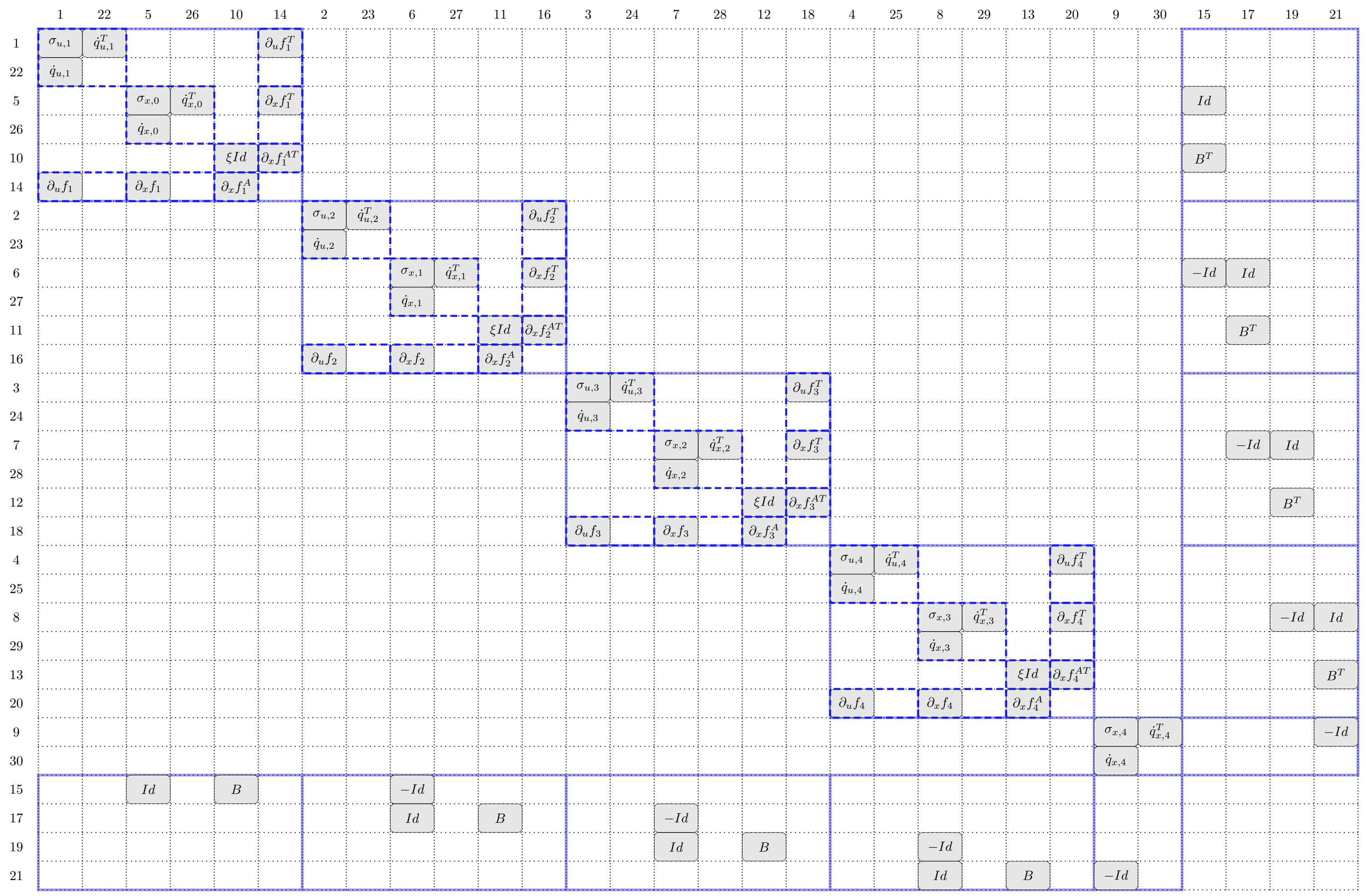

1 22 5 26 10 14 2 23 6 27 11 16 3 24 7 28 12 18 4 25 8 29 13 20 9 30 15 17 19 21
1 $\sigma_{u,1}$ $\dot{q}^T_{u,1}$ $\partial_u f_1^T$
22 $\dot{q}_{u,1}$
5 $\sigma_{x,0}$ $\dot{q}^T_{x,0}$ $\partial_x f_1^T$ Id
26 $\dot{q}_{x,0}$
10 ξId $\partial_x f_1^{AT}$ B^T
14 $\partial_u f_1$ $\partial_x f_1$ $\partial_x f_1^A$
2 $\sigma_{u,2}$ $\dot{q}^T_{u,2}$ $\partial_u f_2^T$
23 $\dot{q}_{u,2}$
6 $\sigma_{x,1}$ $\dot{q}^T_{x,1}$ $\partial_x f_2^T$ $-Id$ Id
27 $\dot{q}_{x,1}$
11 ξId $\partial_x f_2^{AT}$ B^T
16 $\partial_u f_2$ $\partial_x f_2$ $\partial_x f_2^A$
3 $\sigma_{u,3}$ $\dot{q}^T_{u,3}$ $\partial_u f_3^T$
24 $\dot{q}_{u,3}$
7 $\sigma_{x,2}$ $\dot{q}^T_{x,2}$ $\partial_x f_3^T$ $-Id$ Id
28 $\dot{q}_{x,2}$
12 ξId $\partial_x f_3^{AT}$ B^T
18 $\partial_u f_3$ $\partial_x f_3$ $\partial_x f_3^A$
4 $\sigma_{u,4}$ $\dot{q}^T_{u,4}$ $\partial_u f_4^T$
25 $\dot{q}_{u,4}$
8 $\sigma_{x,3}$ $\dot{q}^T_{x,3}$ $\partial_x f_4^T$ $-Id$ Id
29 $\dot{q}_{x,3}$
13 ξId $\partial_x f_4^{AT}$ B^T
20 $\partial_u f_4$ $\partial_x f_4$ $\partial_x f_4^A$
9 $\sigma_{x,4}$ $\dot{q}^T_{x,4}$ $-Id$
30 $\dot{q}_{x,4}$
15 Id B $-Id$
17 Id B $-Id$
19 Id B $-Id$
21 Id B $-Id$

Figure 5.2: Three–way dissected matrix of (3.27) with $N = 4$

5.2 Solving the linear system of equations

We will now identify the computation steps of a manual three–way dissection which results in the matrix shown in Figure 5.2. The five phases of a one–way dissection (4.5g), ..., (4.5f) must be repeated recursively each time when (4.5g) has to be evaluated for an individual dissection level. To be honest, this section might be a little bit uncomfortable to read and one could regret not having used an automatic dissection algorithm. But since the manual work provides a robust nested dissection and eliminates the need of a bulky data structure for hierarchically storing the information about dissection steps, we are encouraged to stay on course.

We assume that the linear system of equations (3.27) is given by

$$\hat{A}p = \hat{b} \tag{5.1}$$

and a permutation matrix P has been constructed with the result that $A = P\hat{A}P^T$ has the desired form of Figure 5.2. Thus, the right hand side of the permuted linear system of equations is defined as $b = P\hat{b}$. The final solution of (3.27) can be obtained by $p = P^T y$, where y is the solution of the permuted system $Ay = b$.

First of all, the dissection phase (4.5g) will be considered for all diagonal blocks of the first dissection level. A close inspection reveals three different blocks $A_{i,i}$, $1 \leq i \leq N+1$. The cases $i = 1$ and $2 \leq i \leq N$ look similar, but differ in the right hand side of the linear system of equations. Case $i = N+1$ falls out of alignment with respect to structure and size. The three cases have to be considered individually. At the end of this chapter we evaluate the remaining dissection phases of the first dissection level and formulate an algorithm which solves (3.27), taking account of all previous results in this chapter.

5.2.1 First dissection level, $2 \leq i \leq N$

Computing (4.5g) of the first dissection level with $2 \leq i \leq N$ is solving the linear system of equations:

$$\begin{pmatrix} \sigma_{u,i} & \dot{q}_{u,i}^T & & & & \partial_u f_i^T \\ \dot{q}_{u,i} & & & & & \\ & & \sigma_{x,i-1} & \dot{q}_{x,i-1}^T & & \partial_x f_i^T \\ & & \dot{q}_{x,i-1} & & & \\ & & & & \xi Id & \partial_x f_i^{AT} \\ \partial_u f_i & & \partial_x f_i & & \partial_x f_i^A & \end{pmatrix} \left(\begin{array}{c|c} Q_i & R_i \end{array} \right) =$$

$$\left(\begin{array}{ccccc|c} & 0 & 0 & & & b_{i,1} \\ & 0 & 0 & & & b_{i,2} \\ \dots \xrightarrow[\text{column}]{(i-1)\text{-st}} & -Id & Id & \dots & & b_{i,3} \\ & 0 & 0 & & & b_{i,4} \\ & 0 & B^T & & & b_{i,5} \\ & 0 & 0 & & & b_{i,6} \end{array} \right) \tag{5.2}$$

To solve this system, we have to solve (4.5g) of the second dissection level for all three diagonal blocks.

First diagonal block: For the first block of (5.2), we solve

$$\begin{pmatrix} \sigma_{u,i} & \dot{q}_{u,i}^T \\ \dot{q}_{u,i} & \end{pmatrix} \left(\begin{array}{c|c} Q_{i,1} & R_{i,1} \end{array} \right) = \left(\begin{array}{c|cccccc} \partial_u f_i^T & \dots & 0 & 0 & \dots & b_{i,1} \\ 0 & \dots & 0 & 0 & \dots & b_{i,2} \end{array} \right) \tag{5.3}$$

in turn by solving (4.5g) of the third dissection level, which leads to the system

$$\sigma_{u,i} \left(\begin{array}{c|c} Q_{i,1,1} & R_{i,1,1} \end{array} \right) = \left(\begin{array}{c|cccccc} \dot{q}_{u,i}^T & \partial_u f_i^T & \dots & 0 & 0 & \dots & b_{i,1} \end{array} \right).$$

We introduce temporary matrices $\tilde{y}_{i,1,1}$, $\tilde{y}_{i,1,2}$, $\tilde{y}_{i,1,3}$ and solve

$$\sigma_{u,i} \begin{pmatrix} \tilde{y}_{i,1,1} & \tilde{y}_{i,1,2} & \tilde{y}_{i,1,3} \end{pmatrix} = \begin{pmatrix} \dot{q}_{u,i}^T & \partial_u f_i^T & b_{i,1} \end{pmatrix},$$

then it holds that

$$\begin{aligned} Q_{i,1,1} &= \tilde{y}_{i,1,1} \\ R_{i,1,1} &= \begin{pmatrix} \tilde{y}_{i,1,2} & \dots & 0 & 0 & \dots & \tilde{y}_{i,1,3} \end{pmatrix}. \end{aligned}$$

The equations (4.5c) and (4.5d) of the third dissection level are given by

$$\begin{aligned} D_{i,1} &= -\dot{q}_{u,i}\tilde{y}_{i,1,1} \\ c_{i,1} &= \begin{pmatrix} 0 & \dots & 0 & 0 & \dots & b_{i,2} \end{pmatrix} - \dot{q}_{u,i}R_{i,1,1} \\ &= \begin{pmatrix} -\dot{q}_{u,i}\tilde{y}_{i,1,2} & \dots & 0 & 0 & \dots & b_{i,2} - \dot{q}_{u,i}\tilde{y}_{i,1,3} \end{pmatrix} \end{aligned}$$

and (4.5e) is given by the linear system of equations

$$\begin{aligned} D_{i,1}y_{i,1,2} &= c_{i,1} \\ (-\dot{q}_{u,i}\tilde{y}_{i,1,1})y_{i,1,2} &= \begin{pmatrix} -\dot{q}_{u,i}\tilde{y}_{i,1,2} & \dots & 0 & 0 & \dots & b_{i,2} - \dot{q}_{u,i}\tilde{y}_{i,1,3} \end{pmatrix}, \end{aligned}$$

where $y_{i,1,2}$ is another temporary matrix. We solve this by introducing $\tilde{y}_{i,1,4}$ and $\tilde{y}_{i,1,5}$ and solving

$$(-\dot{q}_{u,i}\tilde{y}_{i,1,1}) \begin{pmatrix} \tilde{y}_{i,1,4} & \tilde{y}_{i,1,5} \end{pmatrix} = \begin{pmatrix} -\dot{q}_{u,i}\tilde{y}_{i,1,2} & b_{i,2} - \dot{q}_{u,i}\tilde{y}_{i,1,3} \end{pmatrix}.$$

So, $y_{i,1,2}$ is given by

$$y_{i,1,2} = \begin{pmatrix} \tilde{y}_{i,1,4} & \dots & 0 & 0 & \dots & \tilde{y}_{i,1,5} \end{pmatrix}$$

and (4.5f) of the third dissection level is

$$\begin{aligned} y_{i,1,1} &= R_{i,1,1} - Q_{i,1,1} y_{i,1,2} = \\ &= \begin{pmatrix} \tilde{y}_{i,1,2} & \dots & 0 & 0 & \dots & \tilde{y}_{i,1,3} \end{pmatrix} - \tilde{y}_{i,1,1} \begin{pmatrix} \tilde{y}_{i,1,4} & \dots & 0 & 0 & \dots & \tilde{y}_{i,1,5} \end{pmatrix}. \end{aligned}$$

The matrix

$$\begin{pmatrix} y_{i,1,1} \\ y_{i,1,2} \end{pmatrix} = \left(\begin{array}{c|c} Q_{i,1} & R_{i,1} \end{array} \right)$$

is the solution of (5.3), so it holds that

$$\begin{aligned} Q_{i,1} &= \begin{pmatrix} \tilde{y}_{i,1,2} - \tilde{y}_{i,1,1}\tilde{y}_{i,1,4} \\ \tilde{y}_{i,1,4} \end{pmatrix} =: \begin{pmatrix} \tilde{y}_{i,1} \\ \tilde{y}_{i,2} \end{pmatrix} \\ R_{i,1} &= \begin{pmatrix} \dots & 0 & 0 & \dots & \tilde{y}_{i,1,3} - \tilde{y}_{i,1,1}\tilde{y}_{i,1,5} \\ \dots & 0 & 0 & \dots & \tilde{y}_{i,1,5} \end{pmatrix} =: \begin{pmatrix} \dots & 0 & 0 & \dots & \tilde{y}_{i,3} \\ \dots & 0 & 0 & \dots & \tilde{y}_{i,4} \end{pmatrix}, \end{aligned}$$

where $\tilde{y}_{i,1}$, $\tilde{y}_{i,2}$, $\tilde{y}_{i,3}$ and $\tilde{y}_{i,4}$ are matrices which hold the full information of $Q_{i,1}$ and $R_{i,1}$. Alltogether, this leads to Algorithm 5.1.

Algorithm 5.1: Solving the linear system of equations, $i \leq N$, second dissection level, first block

```
1: function BlockURestr($i$)
2:     $t_1 \leftarrow [\dot{q}_{u,i}^T,\ \partial_u f_i^T,\ b_{i,1}]$
3:     $t_1 \leftarrow$ CholSolve($\sigma_{u,i},\ t_1$)
4:     $D \leftarrow -\dot{q}_{u,i} t_1[1]$
5:     $t_2 \leftarrow [-\dot{q}_{u,i} t_1[2],\ b_{i,2} - \dot{q}_{u,i} t_1[3]]$
6:     $t_2 \leftarrow$ CholSolve($-D,\ -t_2$)
7:     return $[t_1[2] - t_1[1]t_2[1],\ t_2[1],\ t_1[3] - t_1[1]t_2[2],\ t_2[2]]$    ▷ $\tilde{y}_{i,1}$, $\tilde{y}_{i,2}$, $\tilde{y}_{i,3}$ and $\tilde{y}_{i,4}$
8: end function
```

If u_i, $i \geq 2$, has no nonlinear equality constraints, then no third dissection step is required. $\tilde{y}_{i,1,1}$, $\tilde{y}_{i,1,4}$ and $\tilde{y}_{i,1,5}$ have no elements and the algorithm becomes much easier, as shown in

Algorithm 5.2.

Algorithm 5.2: Solving the linear system of equations, $i \leq N$, second dissection level, first block, no equality constraints

```
1: function BlockU($i$)
2:     $t \leftarrow [\partial_u f_i^T,\ b_{i,1}]$
3:     return CholSolve($\sigma_{u,i},\ t_1$)                    ▷ $\tilde{y}_{i,1}$, and $\tilde{y}_{i,3}$
4: end function
```

Second diagonal block: The equation (4.5g) of the second diagonal block of (5.2) is given by

$$\begin{pmatrix} \sigma_{x,i-1} & \dot{q}_{x,i-1}^T \\ \dot{q}_{x,i-1} & \end{pmatrix} \left(\begin{array}{c|c} Q_{i,2} & R_{i,2} \end{array} \right) = \left(\begin{array}{c|cccccc} \partial_x f_i^T & \dots & -Id & Id & \dots & b_{i,3} \\ 0 & \dots & 0 & 0 & \dots & b_{i,4} \end{array} \right). \tag{5.4}$$

Again, we solve this by computing (4.5g) of the third dissection level

$$\sigma_{x,i-1} \left(\begin{array}{c|c} Q_{i,2,1} & R_{i,2,1} \end{array} \right) = \left(\begin{array}{c|cccccc} \dot{q}_{x,i-1}^T & \partial_x f_i^T & \dots & -Id & Id & \dots & b_{i,3} \end{array} \right).$$

We introduce the temporary matrices $\tilde{y}_{i,2,1}$, $\tilde{y}_{i,2,2}$, $\tilde{y}_{i,2,3}$ and $\tilde{y}_{i,2,4}$ and solve

$$\sigma_{x,i-1} \begin{pmatrix} \tilde{y}_{i,2,1} & \tilde{y}_{i,2,2} & \tilde{y}_{i,2,4} & \tilde{y}_{i,2,3} \end{pmatrix} = \begin{pmatrix} \dot{q}_{x,i-1}^T & \partial_x f_i^T & Id & b_{i,3} \end{pmatrix}.$$

Then it holds

$$\begin{aligned} Q_{i,2,1} &= \tilde{y}_{i,2,1} \\ R_{i,2,1} &= \begin{pmatrix} \tilde{y}_{i,2,2} & \dots & -\tilde{y}_{i,2,4} & \tilde{y}_{i,2,4} & \dots & \tilde{y}_{i,2,3} \end{pmatrix} \\ D_{i,2} &= -\dot{q}_{x,i-1}\tilde{y}_{i,2,1} \\ c_{i,2} &= \begin{pmatrix} 0 & \dots & 0 & 0 & \dots & b_{i,4} \end{pmatrix} - \dot{q}_{x,i-1} R_{i,2,1} \\ &= \begin{pmatrix} -\dot{q}_{x,i-1}\tilde{y}_{i,2,2} & \dots & \dot{q}_{x,i-1}\tilde{y}_{i,2,4} & -\dot{q}_{x,i-1}\tilde{y}_{i,2,4} & \dots & b_{i,4} - \dot{q}_{x,i-1}\tilde{y}_{i,2,3} \end{pmatrix}. \end{aligned}$$

Since $\sigma_{x,i-1}^{-1}$ is symmetric, we can save a little bit of computation time by simplifying

$$\begin{aligned}
\dot{q}_{x,i-1}\tilde{y}_{i,2,4} &= \dot{q}_{x,i-1}\sigma_{x,i-1}^{-1}Id = \left(\left(\sigma_{x,i-1}^{-1}\right)^T \dot{q}_{x,i-1}^T\right)^T = \left(\sigma_{x,i-1}^{-1}\dot{q}_{x,i-1}^T\right)^T = \tilde{y}_{i,2,1}^T \\
\Rightarrow c_{i,2} &= \begin{pmatrix} -\dot{q}_{x,i-1}\tilde{y}_{i,2,2} & \dots & \tilde{y}_{i,2,1}^T & -\tilde{y}_{i,2,1}^T & \dots & b_{i,4}-\dot{q}_{x,i-1}\tilde{y}_{i,2,3} \end{pmatrix}.
\end{aligned}$$

The equation (4.5e) is given by

$$\begin{aligned}
D_{i,2}y_{i,2,2} &= c_{i,2} \\
(-\dot{q}_{x,i-1}\tilde{y}_{i,2,1})y_{i,2,2} &= \begin{pmatrix} -\dot{q}_{x,i-1}\tilde{y}_{i,2,2} & \dots & \tilde{y}_{i,2,1}^T & -\tilde{y}_{i,2,1}^T & \dots & b_{i,4}-\dot{q}_{x,i-1}\tilde{y}_{i,2,3} \end{pmatrix},
\end{aligned}$$

where $y_{i,2,2}$ is a temporary matrix. We introduce $\tilde{y}_{i,2,5}$, $\tilde{y}_{i,2,6}$ and $\tilde{y}_{i,2,7}$ and solve

$$(-\dot{q}_{x,i-1}\tilde{y}_{i,2,1})\begin{pmatrix}\tilde{y}_{i,2,5} & \tilde{y}_{i,2,7} & \tilde{y}_{i,2,6}\end{pmatrix} = \begin{pmatrix}-\dot{q}_{x,i-1}\tilde{y}_{i,2,2} & \tilde{y}_{i,2,1}^T & b_{i,4}-\dot{q}_{x,i-1}\tilde{y}_{i,2,3}\end{pmatrix}.$$

The back substitution (4.5f) of the third dissection level is given by

$$\begin{aligned}
y_{i,2,2} &= \begin{pmatrix} \tilde{y}_{i,2,5} & \dots & \tilde{y}_{i,2,7} & -\tilde{y}_{i,2,7} & \dots & \tilde{y}_{i,2,6} \end{pmatrix} \\
y_{i,2,1} &= R_{i,2,1} - Q_{i,2,1}y_{i,2,2} = \\
&= \begin{pmatrix} \tilde{y}_{i,2,2} & \dots & -\tilde{y}_{i,2,4} & \tilde{y}_{i,2,4} & \dots & \tilde{y}_{i,2,3} \end{pmatrix} - \\
&\quad \tilde{y}_{i,2,1}\begin{pmatrix} \tilde{y}_{i,2,5} & \dots & \tilde{y}_{i,2,7} & -\tilde{y}_{i,2,7} & \dots & \tilde{y}_{i,2,6} \end{pmatrix}
\end{aligned}$$

and with

$$\begin{pmatrix} y_{i,2,1} \\ y_{i,2,2} \end{pmatrix} = \left(\begin{array}{c|c} Q_{i,2} & R_{i,2} \end{array} \right)$$

the solution of the second diagonal block (5.4) of the second dissection level is given by

$$
\begin{aligned}
Q_{i,2} &= \begin{pmatrix} \tilde{y}_{i,2,2} - \tilde{y}_{i,2,1}\tilde{y}_{i,2,5} \\ \tilde{y}_{i,2,5} \end{pmatrix} =: \begin{pmatrix} \tilde{y}_{i,5} \\ \tilde{y}_{i,6} \end{pmatrix} \\
R_{i,2} &= \begin{pmatrix} \cdots & -\tilde{y}_{i,2,4} - \tilde{y}_{i,2,1}\tilde{y}_{i,2,7} & \tilde{y}_{i,2,4} + \tilde{y}_{i,2,1}\tilde{y}_{i,2,7} & \cdots & \tilde{y}_{i,2,3} - \tilde{y}_{i,2,1}\tilde{y}_{i,2,6} \\ \cdots & \tilde{y}_{i,2,7} & -\tilde{y}_{i,2,7} & \cdots & \tilde{y}_{i,2,6} \end{pmatrix} \\
&=: \begin{pmatrix} \cdots & -\tilde{y}_{i,7} & \tilde{y}_{i,7} & \cdots & \tilde{y}_{i,9} \\ \cdots & -\tilde{y}_{i,8} & \tilde{y}_{i,8} & \cdots & \tilde{y}_{i,10} \end{pmatrix},
\end{aligned}
$$

which defines $\tilde{y}_{i,5}$, $\tilde{y}_{i,6}$, $\tilde{y}_{i,7}$, $\tilde{y}_{i,8}$, $\tilde{y}_{i,9}$ and $\tilde{y}_{i,10}$. These matrices hold the full information of $Q_{i,2}$ and $R_{i,2}$. A compact algorithm representation of the second block computation is shown in Algorithm 5.3. The modified variant without equality constraints is shown in Algorithm 5.4.

Algorithm 5.3: Solving the linear system of equations, $i \leq N$, second dissection level, second block

```
1: function BlockXRestr($i$)
2:     $t_1 \leftarrow$ CholSolve($\sigma_{x,i-1}$, $Id$)
3:     $t_2 \leftarrow [t_1\dot{q}^T_{x,i-1},\ t_1\partial_x f_i^T,\ t_1 b_{i,3}]$
4:     $D \leftarrow -\dot{q}_{x,i-1}t_2[1]$
5:     $t_3 \leftarrow [-\dot{q}_{x,i-1}t_2[2],\ t_2[1]^T,\ b_{i,4} - \dot{q}_{x,i-1}t_2[3]]$
6:     $t_3 \leftarrow$ CholSolve($-D$, $-t_3$)
7:     return $[t_2[2] - t_2[1]t_3[1],\ t_3[1],\ t_1 + t_2[1]t_3[2],\ -t_3[2],\ t_2[3] - t_2[1]t_3[3],\ t_3[3]]$
                        ▷ $\tilde{y}_{i,5}$, $\tilde{y}_{i,6}$, $\tilde{y}_{i,7}$, $\tilde{y}_{i,8}$, $\tilde{y}_{i,9}$ and $\tilde{y}_{i,10}$
8: end function
```

Algorithm 5.4: Solving the linear system of equations, $i \leq N$, second dissection level, second block, no equality constraints

```
1: function BlockX($i$)
2:     $t \leftarrow$ CholSolve($\sigma_{x,i-1}$, $Id$)
3:     return $[t\partial_x f_i^T,\ t,\ tb_{i,3}]$;          ▷ $\tilde{y}_{i,5}$, $\tilde{y}_{i,7}$ and $\tilde{y}_{i,9}$
4: end function
```

Third diagonal block: The third diagonal block of (5.2) is quite easy since no third dissection level must be evaluated:

$$\begin{aligned}
\left(Q_{i,3} \,\middle|\, R_{i,3} \right) &= \left(\xi Id\right)^{-1} \left(\partial_x f_i^{AT} \,\middle|\, \ldots \;\; 0 \;\; B^T \;\; \ldots \;\; b_{i,5} \right) \\
Q_{i,3} &= \frac{1}{\xi}\partial_x f_i^{AT} \\
R_{i,3} &= \begin{pmatrix} \ldots & 0 & \frac{1}{\xi}B^T & \ldots & \frac{1}{\xi}b_{i,5} \end{pmatrix}
\end{aligned}$$

Merging all blocks of the second dissection level: To compute the solution of (5.2) we first have to evaluate (4.5c) and (4.5d):

$$\begin{aligned}
D_i &= -\begin{pmatrix}\partial_u f_i & 0\end{pmatrix}\begin{pmatrix}\tilde{y}_{i,1}\\ \tilde{y}_{i,2}\end{pmatrix} - \begin{pmatrix}\partial_x f_i & 0\end{pmatrix}\begin{pmatrix}\tilde{y}_{i,5}\\ \tilde{y}_{i,6}\end{pmatrix} - -\frac{1}{\xi}\partial_x f_i^A \partial_x f_i^{AT} \\
&= -\partial_u f_i \tilde{y}_{i,1} - \partial_x f_i \tilde{y}_{i,5} - \frac{1}{\xi}\partial_x f_i^A \partial_x f_i^{AT} \\
c_i &= \begin{pmatrix} \ldots & 0 & 0 & \ldots & b_{i,6} \end{pmatrix} - \begin{pmatrix}\partial_u f_i & 0\end{pmatrix} R_{i,1} - \begin{pmatrix}\partial_x f_i & 0\end{pmatrix} R_{i,2} - \partial_x f_i^A R_{i,3} \\
&= \begin{pmatrix} \ldots & 0 & 0 & \ldots & b_{i,6} \end{pmatrix} - \begin{pmatrix} \ldots & 0 & 0 & \ldots & \partial_u f_i \tilde{y}_{i,3} \end{pmatrix} \\
&\quad - \begin{pmatrix} \ldots & -\partial_x f_i \tilde{y}_{i,7} & \partial_x f_i \tilde{y}_{i,7} & \ldots & \partial_x f_i \tilde{y}_{i,9} \end{pmatrix} \\
&\quad - \begin{pmatrix} \ldots & 0 & \frac{1}{\xi}\partial_x f_i^A B^T & \ldots & \frac{1}{\xi}\partial_x f_i^A b_{i,5} \end{pmatrix} \\
&= \Big(\ldots \;\; \partial_x f_i \tilde{y}_{i,7} \;\; -\partial_x f_i \tilde{y}_{i,7} - \tfrac{1}{\xi}\partial_x f_i^A B^T \;\; \ldots \\
&\qquad\qquad \ldots \;\; b_{i,6} - \partial_u f_i \tilde{y}_{i,3} - \partial_x f_i \tilde{y}_{i,9} - \tfrac{1}{\xi}\partial_x f_i^A b_{i,5} \Big)
\end{aligned}$$

We introduce the temporary matrices $\tilde{y}_{i,11}$, $\tilde{y}_{i,12}$ and $\tilde{y}_{i,13}$, and compute the relevant elements of (4.5e) (of the second dissection level) by solving

$$D_i \begin{pmatrix}\tilde{y}_{i,11} & \tilde{y}_{i,12} & \tilde{y}_{i,13}\end{pmatrix} =$$

$$\begin{pmatrix}\partial_x f_i \tilde{y}_{i,7} & -\partial_x f_i \tilde{y}_{i,7} - \frac{1}{\xi}\partial_x f_i^A B^T & b_{i,6} - \partial_u f_i \tilde{y}_{i,3} - \partial_x f_i \tilde{y}_{i,9} - \frac{1}{\xi}\partial_x f_i^A b_{i,5}\end{pmatrix}.$$

$$y_{i,6} = \begin{pmatrix}\ldots & \tilde{y}_{i,11} & \tilde{y}_{i,12} & \ldots & \tilde{y}_{i,13}\end{pmatrix}.$$

The three iterations of (4.5f) are given by

$$\begin{aligned}
\begin{pmatrix} y_{i,1} \\ y_{i,2} \end{pmatrix} &= \begin{pmatrix} \dots & 0 & 0 & \dots & \tilde{y}_{i,3} \\ \dots & 0 & 0 & \dots & \tilde{y}_{i,4} \end{pmatrix} - \begin{pmatrix} \tilde{y}_{i,1} \\ \tilde{y}_{i,2} \end{pmatrix} \begin{pmatrix} \dots & \tilde{y}_{i,11} & \tilde{y}_{i,12} & \dots & \tilde{y}_{i,13} \end{pmatrix} \\
&= \begin{pmatrix} \dots & -\tilde{y}_{i,1}\tilde{y}_{i,11} & -\tilde{y}_{i,1}\tilde{y}_{i,12} & \dots & \tilde{y}_{i,3} - \tilde{y}_{i,1}\tilde{y}_{i,13} \\ \dots & -\tilde{y}_{i,2}\tilde{y}_{i,11} & -\tilde{y}_{i,2}\tilde{y}_{i,12} & \dots & \tilde{y}_{i,4} - \tilde{y}_{i,2}\tilde{y}_{i,13} \end{pmatrix} \\
\begin{pmatrix} y_{i,3} \\ y_{i,4} \end{pmatrix} &= \begin{pmatrix} \dots & -\tilde{y}_{i,7} & \tilde{y}_{i,7} & \dots & \tilde{y}_{i,9} \\ \dots & -\tilde{y}_{i,8} & \tilde{y}_{i,8} & \dots & \tilde{y}_{i,10} \end{pmatrix} - \begin{pmatrix} \tilde{y}_{i,5} \\ \tilde{y}_{i,6} \end{pmatrix} \begin{pmatrix} \dots & \tilde{y}_{i,11} & \tilde{y}_{i,12} & \dots & \tilde{y}_{i,13} \end{pmatrix} \\
&= \begin{pmatrix} \dots & -\tilde{y}_{i,7} - \tilde{y}_{i,5}\tilde{y}_{i,11} & \tilde{y}_{i,7} - \tilde{y}_{i,5}\tilde{y}_{i,12} & \dots & \tilde{y}_{i,9} - \tilde{y}_{i,5}\tilde{y}_{i,13} \\ \dots & -\tilde{y}_{i,8} - \tilde{y}_{i,6}\tilde{y}_{i,11} & \tilde{y}_{i,8} - \tilde{y}_{i,6}\tilde{y}_{i,12} & \dots & \tilde{y}_{i,10} - \tilde{y}_{i,6}\tilde{y}_{i,13} \end{pmatrix} \\
y_{i,5} &= \begin{pmatrix} \dots & 0 & \frac{1}{\xi}B^T & \dots & \frac{1}{\xi}b_{i,5} \end{pmatrix} - \frac{1}{\xi}\partial_x f_i^{AT} \begin{pmatrix} \dots & \tilde{y}_{i,11} & \tilde{y}_{i,12} & \dots & \tilde{y}_{i,13} \end{pmatrix} \\
&= \begin{pmatrix} \dots & -\frac{1}{\xi}\partial_x f_i^{AT}\tilde{y}_{i,11} & \frac{1}{\xi}(B^T - \partial_x f_i^{AT}\tilde{y}_{i,12}) & \dots & \frac{1}{\xi}(b_{i,5} - \partial_x f_i^{AT}\tilde{y}_{i,13}) \end{pmatrix}
\end{aligned}$$

and (finally) the solution $(\; Q_i \mid R_i \;)$ of (5.2) can be obtained by

$$Q_i = \begin{pmatrix} & -\tilde{y}_{i,1}\tilde{y}_{i,11} & -\tilde{y}_{i,1}\tilde{y}_{i,12} & \\ & -\tilde{y}_{i,2}\tilde{y}_{i,11} & -\tilde{y}_{i,2}\tilde{y}_{i,12} & \\ & -\tilde{y}_{i,7} - \tilde{y}_{i,5}\tilde{y}_{i,11} & \tilde{y}_{i,7} - \tilde{y}_{i,5}\tilde{y}_{i,12} & \\ \dots \xrightarrow[\text{column}]{(i-1)\text{-st}} & -\tilde{y}_{i,8} - \tilde{y}_{i,6}\tilde{y}_{i,11} & \tilde{y}_{i,8} - \tilde{y}_{i,6}\tilde{y}_{i,12} & \dots \\ & -\frac{1}{\xi}\partial_x f_i^{AT}\tilde{y}_{i,11} & \frac{1}{\xi}(B^T - \partial_x f_i^{AT}\tilde{y}_{i,12}) & \\ & \tilde{y}_{i,11} & \tilde{y}_{i,12} & \end{pmatrix}$$

$$=: \begin{pmatrix} & Q_i^{(1,2)} & Q_i^{(1,1)} & \\ & Q_i^{(2,2)} & Q_i^{(2,1)} & \\ \dots \xrightarrow[\text{column}]{(i-1)\text{-st}} & Q_i^{(3,2)} & Q_i^{(3,1)} & \dots \\ & Q_i^{(4,2)} & Q_i^{(4,1)} & \\ & Q_i^{(5,2)} & Q_i^{(5,1)} & \\ & Q_i^{(6,2)} & Q_i^{(6,1)} & \end{pmatrix}$$

$$R_i = \begin{pmatrix} \tilde{y}_{i,3} - \tilde{y}_{i,1}\tilde{y}_{i,13} \\ \tilde{y}_{i,4} - \tilde{y}_{i,2}\tilde{y}_{i,13} \\ \tilde{y}_{i,9} - \tilde{y}_{i,5}\tilde{y}_{i,13} \\ \tilde{y}_{i,10} - \tilde{y}_{i,6}\tilde{y}_{i,13} \\ \frac{1}{\xi}(b_{i,5} - \partial_x f_i^{AT}\tilde{y}_{i,13}) \\ \tilde{y}_{i,13} \end{pmatrix} =: \begin{pmatrix} R_i^{(1)} \\ R_i^{(2)} \\ R_i^{(3)} \\ R_i^{(4)} \\ R_i^{(5)} \\ R_i^{(6)} \end{pmatrix}.$$

5.2.2 First dissection level, $i = 1$

For $i = 1$, the dissection is similar to the case $2 \leq i \leq N$. Only the right hand side of the linear system of equations differs a little bit:

$$\begin{pmatrix} \sigma_{u,i} & \dot{q}_{u,i}^T & & & & \partial_u f_i^T \\ \dot{q}_{u,i} & & & & & \\ & & \sigma_{x,i-1} & \dot{q}_{x,i-1}^T & & \partial_x f_i^T \\ & & \dot{q}_{x,i-1} & & & \\ & & & & \xi Id & \partial_x f_i^{AT} \\ \partial_u f_i & & \partial_x f_i & & \partial_x f_i^A & \end{pmatrix} \left(\begin{array}{c|c} Q_i & R_i \end{array} \right) = \left(\begin{array}{cc|c} 0 & & b_{i,1} \\ 0 & & b_{i,2} \\ Id & & b_{i,3} \\ 0 & \dots & b_{i,4} \\ B^T & & b_{i,5} \\ 0 & & b_{i,6} \end{array} \right) \qquad (5.5)$$

First diagonal block: Processing the first diagonal block of (5.5) is exactly the same as for $2 \le i \le N$. So we compute $Q_{i,1}$ and $R_{i,1}$ via

$$\begin{pmatrix} \sigma_{u,i} & \dot{q}_{u,i}^T \\ \dot{q}_{u,i} & \end{pmatrix} \left(\begin{array}{c|c} Q_{i,1} & R_{i,1} \end{array} \right) = \left(\begin{array}{c|ccc} \partial_u f_i^T & 0 & \dots & b_{i,1} \\ 0 & 0 & \dots & b_{i,2} \end{array} \right) \tag{5.6}$$

with Algorithm 5.1 and/or 5.2.

Second diagonal block: The next block differs a bit from the case $2 \le i \le N$. The linear system of equations is

$$\begin{pmatrix} \sigma_{x,i-1} & \dot{q}_{x,i-1}^T \\ \dot{q}_{x,i-1} & \end{pmatrix} \left(\begin{array}{c|c} Q_{i,2} & R_{i,2} \end{array} \right) = \left(\begin{array}{c|ccc} \partial_x f_i^T & Id & \dots & b_{i,3} \\ 0 & 0 & \dots & b_{i,4} \end{array} \right), \tag{5.7}$$

where the right hand side lacks the $-Id$ element. Since this element has not been explicitly calculated (we have only processed the Id element and used the negative result), we can use the algorithms of the case $2 \le i \le N$ again. The solution is given by

$$\begin{aligned} Q_{i,2} &= \begin{pmatrix} \tilde{y}_{i,2,2} - \tilde{y}_{i,2,1}\tilde{y}_{i,2,5} \\ \tilde{y}_{i,2,5} \end{pmatrix} =: \begin{pmatrix} \tilde{y}_{i,5} \\ \tilde{y}_{i,6} \end{pmatrix} \\ R_{i,2} &= \begin{pmatrix} \tilde{y}_{i,2,4} + \tilde{y}_{i,2,1}\tilde{y}_{i,2,7} & \dots & \tilde{y}_{i,2,3} - \tilde{y}_{i,2,1}\tilde{y}_{i,2,6} \\ -\tilde{y}_{i,2,7} & \dots & \tilde{y}_{i,2,6} \end{pmatrix} \\ &=: \begin{pmatrix} \tilde{y}_{i,7} & \dots & \tilde{y}_{i,9} \\ \tilde{y}_{i,8} & \dots & \tilde{y}_{i,10} \end{pmatrix}, \end{aligned}$$

where $\tilde{y}_{i,5}$, $\tilde{y}_{i,6}$, $\tilde{y}_{i,7}$, $\tilde{y}_{i,8}$, $\tilde{y}_{i,9}$ and $\tilde{y}_{i,10}$ are obtained by Algorithm 5.3 and/or Algorithm 5.4.

Third diagonal block: The last block of (5.5) leads to the same evaluations as for the case $2 \le i \le N$:

$$\left(\begin{array}{c|c} Q_{i,3} & R_{i,3} \end{array} \right) = \left(\xi Id \right)^{-1} \left(\begin{array}{c|ccc} \partial_x f_i^{AT} & B^T & \dots & b_{i,5} \end{array} \right)$$

$$\begin{aligned}
Q_{i,3} &= \frac{1}{\xi}\partial_x f_i^{AT} \\
R_{i,3} &= \begin{pmatrix} \frac{1}{\xi}B^T & \dots & \frac{1}{\xi}b_{i,5} \end{pmatrix}.
\end{aligned}$$

Merging all blocks of the second dissection level: We put everything together to solve (5.5)

$$\begin{aligned}
D_i &= -\begin{pmatrix}\partial_u f_i & 0\end{pmatrix}\begin{pmatrix}\tilde{y}_{i,1} \\ \tilde{y}_{i,2}\end{pmatrix} - \begin{pmatrix}\partial_x f_i & 0\end{pmatrix}\begin{pmatrix}\tilde{y}_{i,5} \\ \tilde{y}_{i,6}\end{pmatrix} - -\frac{1}{\xi}\partial_x f_i^A \partial_x f_i^{AT} \\
&= -\partial_u f_i \tilde{y}_{i,1} - \partial_x f_i \tilde{y}_{i,5} - \frac{1}{\xi}\partial_x f_i^A \partial_x f_i^{AT} \\
c_i &= \begin{pmatrix} 0 & \dots & b_{i,6}\end{pmatrix} - \begin{pmatrix}\partial_u f_i & 0\end{pmatrix} R_{i,1} - \begin{pmatrix}\partial_x f_i & 0\end{pmatrix} R_{i,2} - \partial_x f_i^A R_{i,3} \\
&= \begin{pmatrix} 0 & \dots & b_{i,6}\end{pmatrix} - \begin{pmatrix} 0 & \dots & \partial_u f_i \tilde{y}_{i,3}\end{pmatrix} - \begin{pmatrix}\partial_x f_i \tilde{y}_{i,7} & \dots & \partial_x f_i \tilde{y}_{i,9}\end{pmatrix} \\
&\quad - \begin{pmatrix} \frac{1}{\xi}\partial_x f_i^A B^T & \dots & \frac{1}{\xi}\partial_x f_i^A b_{i,5}\end{pmatrix} \\
&= \begin{pmatrix} -\partial_x f_i \tilde{y}_{i,7} - \frac{1}{\xi}\partial_x f_i^A B^T & \dots & b_{i,6} - \partial_u f_i \tilde{y}_{i,3} - \partial_x f_i \tilde{y}_{i,9} - \frac{1}{\xi}\partial_x f_i^A b_{i,5}\end{pmatrix}
\end{aligned}$$

We compute $\tilde{y}_{i,12}$ and $\tilde{y}_{i,13}$ by solving

$$D_i \begin{pmatrix}\tilde{y}_{i,12} & \tilde{y}_{i,13}\end{pmatrix} = \begin{pmatrix} -\partial_x f_i \tilde{y}_{i,7} - \frac{1}{\xi}\partial_x f_i^A B^T & b_{i,6} - \partial_u f_i \tilde{y}_{i,3} - \partial_x f_i \tilde{y}_{i,9} - \frac{1}{\xi}\partial_x f_i^A b_{i,5}\end{pmatrix}$$

$$y_{i,6} = \begin{pmatrix}\tilde{y}_{i,12} & \dots & \tilde{y}_{i,13}\end{pmatrix}.$$

Note that we skipped $\tilde{y}_{i,11}$. This leads to a consistent index numbering with respect to the case $2 \le i \le N$. The three iterations of (4.5f) are given by

$$\begin{aligned}
\begin{pmatrix} y_{i,1} \\ y_{i,2}\end{pmatrix} &= \begin{pmatrix} 0 & \dots & \tilde{y}_{i,3} \\ 0 & \dots & \tilde{y}_{i,4}\end{pmatrix} - \begin{pmatrix}\tilde{y}_{i,1} \\ \tilde{y}_{i,2}\end{pmatrix}\begin{pmatrix}\tilde{y}_{i,12} & \dots & \tilde{y}_{i,13}\end{pmatrix} \\
&= \begin{pmatrix} -\tilde{y}_{i,1}\tilde{y}_{i,12} & \dots & \tilde{y}_{i,3} - \tilde{y}_{i,1}\tilde{y}_{i,13} \\ -\tilde{y}_{i,2}\tilde{y}_{i,12} & \dots & \tilde{y}_{i,4} - \tilde{y}_{i,2}\tilde{y}_{i,13}\end{pmatrix}
\end{aligned}$$

$$\begin{aligned}
\begin{pmatrix} y_{i,3} \\ y_{i,4} \end{pmatrix} &= \begin{pmatrix} \tilde{y}_{i,7} & \cdots & \tilde{y}_{i,9} \\ \tilde{y}_{i,8} & \cdots & \tilde{y}_{i,10} \end{pmatrix} - \begin{pmatrix} \tilde{y}_{i,5} \\ \tilde{y}_{i,6} \end{pmatrix} \begin{pmatrix} \tilde{y}_{i,12} & \cdots & \tilde{y}_{i,13} \end{pmatrix} \\
&= \begin{pmatrix} \tilde{y}_{i,7} - \tilde{y}_{i,5}\tilde{y}_{i,12} & \cdots & \tilde{y}_{i,9} - \tilde{y}_{i,5}\tilde{y}_{i,13} \\ \tilde{y}_{i,8} - \tilde{y}_{i,6}\tilde{y}_{i,12} & \cdots & \tilde{y}_{i,10} - \tilde{y}_{i,6}\tilde{y}_{i,13} \end{pmatrix} \\
y_{i,5} &= \begin{pmatrix} \frac{1}{\xi}B^T & \cdots & \frac{1}{\xi}b_{i,5} \end{pmatrix} - \frac{1}{\xi}\partial_x f_i^{AT} \begin{pmatrix} \tilde{y}_{i,12} & \cdots & \tilde{y}_{i,13} \end{pmatrix} \\
&= \begin{pmatrix} \frac{1}{\xi}(B^T - \partial_x f_i^{AT}\tilde{y}_{i,12}) & \cdots & \frac{1}{\xi}(b_{i,5} - \partial_x f_i^{AT}\tilde{y}_{i,13}) \end{pmatrix}.
\end{aligned}$$

The solution $(\; Q_i \mid R_i \;)$ of (5.5) can be obtained by

$$\begin{aligned}
Q_i &= \begin{pmatrix} -\tilde{y}_{i,1}\tilde{y}_{i,12} \\ -\tilde{y}_{i,2}\tilde{y}_{i,12} \\ \tilde{y}_{i,7} - \tilde{y}_{i,5}\tilde{y}_{i,12} \\ \tilde{y}_{i,8} - \tilde{y}_{i,6}\tilde{y}_{i,12} \\ \frac{1}{\xi}(B^T - \partial_x f_i^{AT}\tilde{y}_{i,12}) \\ \tilde{y}_{i,12} \end{pmatrix} \cdots =: \begin{pmatrix} Q_i^{(1,1)} \\ Q_i^{(2,1)} \\ Q_i^{(3,1)} \\ Q_i^{(4,1)} \\ Q_i^{(5,1)} \\ Q_i^{(6,1)} \end{pmatrix} \cdots \\
R_i &= \begin{pmatrix} \tilde{y}_{i,3} - \tilde{y}_{i,1}\tilde{y}_{i,13} \\ \tilde{y}_{i,4} - \tilde{y}_{i,2}\tilde{y}_{i,13} \\ \tilde{y}_{i,9} - \tilde{y}_{i,5}\tilde{y}_{i,13} \\ \tilde{y}_{i,10} - \tilde{y}_{i,6}\tilde{y}_{i,13} \\ \frac{1}{\xi}(b_{i,5} - \partial_x f_i^{AT}\tilde{y}_{i,13}) \\ \tilde{y}_{i,13} \end{pmatrix} =: \begin{pmatrix} R_i^{(1)} \\ R_i^{(2)} \\ R_i^{(3)} \\ R_i^{(4)} \\ R_i^{(5)} \\ R_i^{(6)} \end{pmatrix}.
\end{aligned}$$

Thanks to the consistent choice of indices for both cases $2 \le i \le N$ and $i = 1$, we can formulate a universal algorithm for the block merging step and $i \le N$. The implementation is shown in Algorithm 5.5.

Algorithm 5.5: Solving the linear system of equations, $i \le N$, first dissection level

```
1: function BlockMerge($i, v, w$)
```

```
                                   ▷ $v = [\tilde{y}_{i,1},\ \tilde{y}_{i,2},\ \tilde{y}_{i,3},\ \tilde{y}_{i,4}]$
                                   ▷ $w = [\tilde{y}_{i,5},\ \tilde{y}_{i,6},\ \tilde{y}_{i,7},\ \tilde{y}_{i,8},\ \tilde{y}_{i,9},\ \tilde{y}_{i,10}]$
 2:     $D \leftarrow -\partial_u f_i v[1] - \partial_x f_i w[1] - \frac{1}{\xi}\partial_x f_i^A \partial_x f_i^{AT}$
 3:     if $i \geq 2$ then
 4:         $t \leftarrow [-\partial_x f_i w[3] - \frac{1}{\xi}\partial_x f_i^A B^T,\ b_{i,6} - \partial_u f_i v[3] - \partial_x f_i w[5] - \frac{1}{\xi}\partial_x f_i^A b_{i,5},\ \partial_x f_i w[3]]$
 5:     else
 6:         $t \leftarrow [-\partial_x f_i w[3] - \frac{1}{\xi}\partial_x f_i^A B^T,\ b_{i,6} - \partial_u f_i v[3] - \partial_x f_i w[5] - \frac{1}{\xi}\partial_x f_i^A b_{i,5}]$
 7:     end if
 8:     $t \leftarrow$ CholSolve($-D$,$-t$)
                                   ▷ 2nd and 4th component of $Q1$, $Q2$ and $R$
                                   ▷ can have zero dimension if $u_i$ or $x_i$
                                   ▷ have no nonlinear equality constraints.
 9:     $Q1 \leftarrow \begin{bmatrix} -v[1]t[1] \\ -v[2]t[1] \\ w[3] - w[1]t[1] \\ w[4] - w[2]t[1] \\ \frac{1}{\xi}(B^T - \partial_x f_i^{AT} t[1]) \\ t[1] \end{bmatrix}$, $R \leftarrow \begin{bmatrix} v[3] - v[1]t[2] \\ v[4] - v[2]t[2] \\ w[5] - w[1]t[2] \\ w[6] - w[2]t[2] \\ \frac{1}{\xi}(b_{i,5} - \partial_x f_i^{AT} t[2]) \\ t[2] \end{bmatrix}$
10:     if $i \geq 2$ then
11:         $Q2 \leftarrow \begin{bmatrix} -v[1]t[3] \\ -v[2]t[3] \\ -w[3] - w[1]t[3] \\ -w[4] - w[2]t[3] \\ -\frac{1}{\xi}\partial_x f_i^{AT} t[3] \\ t[3] \end{bmatrix}$
12:     else
13:         $Q2 \leftarrow [\,]$                          ▷ empty placeholder...
14:     end if
15:     return $[Q1,\ Q2,\ R]$
16: end function
```

5.2.3 First dissection level, $i = N + 1$

The last diagonal block of the first dissection level is given by

$$\begin{pmatrix} \sigma_{x,i-1} & \dot{q}_{x,i-1}^T \\ \dot{q}_{x,i-1} & \end{pmatrix} \left(\begin{array}{c|c} Q_i & R_i \end{array} \right) = \left(\begin{array}{cc|c} \dots & -Id & b_{i,3} \\ & 0 & b_{i,4} \end{array} \right). \tag{5.8}$$

For the second dissection level, we have to solve

$$\sigma_{x,i-1} \left(\begin{array}{c|c} Q_{i,2,1} & R_{i,2,1} \end{array} \right) \quad = \quad \left(\begin{array}{c|ccc} \dot{q}_{x,i-1}^T & \dots & -Id & b_{i,3} \end{array} \right).$$

Since this problem is similar to (5.4), we choose a similar enumeration of $Q_{i,2,1}$ and $R_{i,2,1}$. Otherwise, the indices would interfere with the second dissection level enumeration of the previous cases. We solve this equation by introducing temporary matrices $\tilde{y}_{i,2,1}$, $\tilde{y}_{i,2,3}$ and $\tilde{y}_{i,2,4}$ and solving

$$\sigma_{x,i-1} \begin{pmatrix} \tilde{y}_{i,2,1} & \tilde{y}_{i,2,4} & \tilde{y}_{i,2,3} \end{pmatrix} = \begin{pmatrix} \dot{q}_{x,i-1}^T & Id & b_{i,3} \end{pmatrix}.$$

Then it holds that

$$\begin{aligned} Q_{i,2,1} &= \tilde{y}_{i,2,1} \\ R_{i,2,1} &= \begin{pmatrix} \dots & -\tilde{y}_{i,2,4} & \tilde{y}_{i,2,3} \end{pmatrix} \end{aligned}$$

and the separator block is given by

$$\begin{aligned} D_{i,2} &= -\dot{q}_{x,i-1}\tilde{y}_{i,2,1} \\ c_{i,2} &= \begin{pmatrix} \dots & 0 & b_{i,4} \end{pmatrix} - N_{i,2,1}^T R_{i,2,1} \\ &= \begin{pmatrix} \dots & \tilde{y}_{i,2,1}^T & b_{i,4} - \dot{q}_{x,i-1}\tilde{y}_{i,2,3} \end{pmatrix}, \end{aligned}$$

since

$$\dot{q}_{x,i-1}\tilde{y}_{i,2,4} \quad = \quad \dot{q}_{x,i-1}\sigma_{x,i-1}^{-1} Id = \left(\left(\sigma_{x,i-1}^{-1}\right)^T \dot{q}_{x,i-1}^T \right)^T = \left(\sigma_{x,i-1}^{-1}\dot{q}_{x,i-1}^T\right)^T = \tilde{y}_{i,2,1}^T.$$

We solve the linear system of equations

$$\begin{aligned} y_{i,2,2} &= D_{i,2}^{-1} c_{i,2} \\ &= (-\dot{q}_{x,i-1}\tilde{y}_{i,2,1})^{-1} \begin{pmatrix} \dots & \tilde{y}_{i,2,1}^T & b_{i,4} - \dot{q}_{x,i-1}\tilde{y}_{i,2,3} \end{pmatrix} \end{aligned}$$

of the separator block by introducing temporary matrices $\tilde{y}_{i,2,6}$, $\tilde{y}_{i,2,7}$ and solving

$$\begin{aligned} (-\dot{q}_{x,i-1}\tilde{y}_{i,2,1}) \begin{pmatrix} \tilde{y}_{i,2,7} & \tilde{y}_{i,2,6} \end{pmatrix} &= \begin{pmatrix} \tilde{y}_{i,2,1}^T & b_{i,4} - \dot{q}_{x,i-1}\tilde{y}_{i,2,3} \end{pmatrix} \\ \Rightarrow \quad y_{i,2,2} &= \begin{pmatrix} \dots & \tilde{y}_{i,2,7} & \tilde{y}_{i,2,6} \end{pmatrix}. \end{aligned}$$

With the backsubstitution step

$$\begin{aligned} y_{i,2,1} &= R_{i,2,1} - Q_{i,2,1} y_{i,2,2} = \\ &= \begin{pmatrix} \dots & -\tilde{y}_{i,2,4} & \tilde{y}_{i,2,3} \end{pmatrix} - \\ &\quad \tilde{y}_{i,2,1} \begin{pmatrix} \dots & \tilde{y}_{i,2,7} & \tilde{y}_{i,2,6} \end{pmatrix} \end{aligned}$$

the solution of (5.8) is given by

$$\begin{aligned} Q_i &= \begin{pmatrix} \dots & \begin{matrix} -\tilde{y}_{i,2,4} - \tilde{y}_{i,2,1}\tilde{y}_{i,2,7} \\ \tilde{y}_{i,2,7} \end{matrix} \end{pmatrix} =: \begin{pmatrix} \dots & \begin{matrix} Q_i^{(3,2)} \\ Q_i^{(4,2)} \end{matrix} \end{pmatrix} \\ R_i &= \begin{pmatrix} \tilde{y}_{i,2,3} - \tilde{y}_{i,2,1}\tilde{y}_{i,2,6} \\ \tilde{y}_{i,2,6} \end{pmatrix} =: \begin{pmatrix} R_i^{(3)} \\ R_i^{(4)} \end{pmatrix}. \end{aligned}$$

A compact implementation of these computation steps is shown in Algorithm 5.6. If no nonlinear equality constraints are defined for x_N, the computation becomes a lot easier since no second dissection step is required any more (Algorithm 5.7).

Algorithm 5.6: Solving the linear system of equations, $i = N + 1$, first dissection level

```
1: function BlockXRestrLast
2:     $t_1 \leftarrow$ CholSolve($\sigma_{x,N}$, $Id$)
3:     $t_2 \leftarrow [t_1 \dot{q}_{x,N}^T,\ t_1 b_{N+1,3}]$
4:     $D \leftarrow -\dot{q}_{x,N} t_2[1]$
5:     $t_3 \leftarrow [t_2[1]^T,\ b_{N+1,4} - \dot{q}_{x,N} t_2[2]]$
6:     $t_3 \leftarrow$ CholSolve($-D$, $-t_3$)
7:     return [ $[-t_1 - t_2[1]t_3[1],\ t_3[1]],\ [t_2[2] - t_2[1]t_3[2],\ t_3[2]]$ ]        ▷ $Q_{N+1}, R_{N+1}$
8: end function
```

Algorithm 5.7: Solving the linear system of equations, $i = N + 1$, first dissection level, no equality constraints

```
1: function BlockXLast
2:     $t \leftarrow$ CholSigma($x$, $N$, $Id$)
3:     return $[-t,\ tb_{N+1,3}]$;        ▷ $Q_{N+1}, R_{N+1}$
4: end function
```

5.2.4 Finalizing the first dissection level

Now that we have computed (4.5g) for each diagonal block of the first dissection level, we can construct D and c of (4.5c) and (4.5d). Just to call to mind, those are equations (iterator bounds and separator block equal to zero has already been applied)

$$D = -\sum_{i=1}^{N+1} A_{i,N+2}^T Q_i \tag{4.5c}$$

$$c = b_{N+2} - \sum_{i=1}^{N+1} A_{i,N+2}^T R_i. \tag{4.5d}$$

In the case of $2 \leq i \leq N$, it holds that

$$A_{i,N+2}^T Q_i = \xrightarrow[\text{row}]{(i-1)\text{-st}} \begin{pmatrix} & & \vdots & & & \\ 0 & 0 & -Id & 0 & 0 & 0 \\ 0 & 0 & Id & 0 & B & 0 \\ & & \vdots & & & \end{pmatrix} \begin{pmatrix} & Q_i^{(1,2)} & Q_i^{(1,1)} & \\ & Q_i^{(2,2)} & Q_i^{(2,1)} & \\ \cdots \xrightarrow[\text{column}]{(i-1)\text{-st}} & Q_i^{(3,2)} & Q_i^{(3,1)} & \cdots \\ & Q_i^{(4,2)} & Q_i^{(4,1)} & \\ & Q_i^{(5,2)} & Q_i^{(5,1)} & \\ & Q_i^{(6,2)} & Q_i^{(6,1)} & \end{pmatrix}$$

$$= \begin{pmatrix} & & \vdots & \\ & & \downarrow \ (i-1)\text{-st row} & \\ \cdots \xrightarrow[\text{column}]{(i-1)\text{-st}} & -Q_i^{(3,2)} & -Q_i^{(3,1)} & \cdots \\ & Q_i^{(3,2)} + BQ_i^{(5,2)} & Q_i^{(3,1)} + BQ_i^{(5,1)} & \\ & & \vdots & \end{pmatrix}$$

$$A_{i,N+2}^T R_i = \xrightarrow[\text{row}]{(i-1)\text{-st}} \begin{pmatrix} & & \vdots & & & \\ 0 & 0 & -Id & 0 & 0 & 0 \\ 0 & 0 & Id & 0 & B & 0 \\ & & \vdots & & & \end{pmatrix} \begin{pmatrix} R_i^{(1)} \\ R_i^{(2)} \\ R_i^{(3)} \\ R_i^{(4)} \\ R_i^{(5)} \\ R_i^{(6)} \end{pmatrix} = \xrightarrow[\text{row}]{(i-1)\text{-st}} \begin{pmatrix} \vdots \\ -R_i^{(3)} \\ R_i^{(3)} + BR_i^{(5)} \\ \vdots \end{pmatrix},$$

if $i = 1$,

$$A_{i,N+2}^T Q_i = \begin{pmatrix} 0 & 0 & Id & 0 & B & 0 \\ & & \vdots & & & \end{pmatrix} \begin{pmatrix} Q_i^{(1,1)} & \\ Q_i^{(2,1)} & \\ Q_i^{(3,1)} & \\ Q_i^{(4,1)} & \cdots \\ Q_i^{(5,1)} & \\ Q_i^{(6,1)} & \end{pmatrix} = \begin{pmatrix} Q_i^{(3,1)} + BQ_i^{(5,1)} & \cdots \\ \vdots & \end{pmatrix}$$

$$A_{i,N+2}^T R_i = \begin{pmatrix} 0 & 0 & Id & 0 & B & 0 \\ & & \vdots & & & \end{pmatrix} \begin{pmatrix} R_i^{(1)} \\ R_i^{(2)} \\ R_i^{(3)} \\ R_i^{(4)} \\ R_i^{(5)} \\ R_i^{(6)} \end{pmatrix} = \begin{pmatrix} R_i^{(3)} + BR_i^{(5)} \\ \vdots \end{pmatrix}$$

and if $i = N+1$,

$$\begin{aligned} A_{i,N+2}^T Q_i &= \begin{pmatrix} & \vdots \\ -Id & 0 \end{pmatrix} \begin{pmatrix} & Q_i^{(3,2)} \\ \dots & \\ & Q_i^{(4,2)} \end{pmatrix} = \begin{pmatrix} & \vdots \\ \dots & -Q_i^{(3,2)} \end{pmatrix} \\ A_{i,N+2}^T R_i &= \begin{pmatrix} & \vdots \\ -Id & 0 \end{pmatrix} \begin{pmatrix} R_i^{(3)} \\ R_i^{(4)} \end{pmatrix} = \begin{pmatrix} \vdots \\ -R_i^{(3)} \end{pmatrix}. \end{aligned}$$

Alltogether, one can write

$$D = -\sum_{i=1}^{N+1} A_{i,N+2}^T Q_i = \begin{pmatrix} \tilde{A}_1 & \tilde{B}_2 & & \\ \tilde{C}_2 & & & \\ & & \ddots & \\ & & & \tilde{B}_N \\ & & \tilde{C}_N & \tilde{A}_N \end{pmatrix} \tag{5.9}$$

$$c = b_{N+2} - \sum_{i=1}^{N+1} A_{i,N+2}^T R_i = b_{N+2} - \begin{pmatrix} \tilde{D}_1 \\ \vdots \\ \tilde{D}_N \end{pmatrix} \tag{5.10}$$

with

$$\tilde{A}_i := -Q_i^{(3,1)} - BQ_i^{(5,1)} + Q_{i+1}^{(3,2)} \tag{5.11}$$

$$\tilde{B}_i := Q_i^{(3,1)} \tag{5.12}$$

$$\tilde{C}_i := -Q_i^{(3,2)} - BQ_i^{(5,2)} \tag{5.13}$$

$$\tilde{D}_i \;:=\; R_i^{(3)} + BR_i^{(5)} - R_{i+1}^{(3)}. \tag{5.14}$$

Finally, we solve

$$Dy_{N+2} = c. \tag{5.15}$$

We can make use of the very nice regular structure of D. In Section 4.2.3, we have already seen a concept of how to handle sparsity patterns within Cholesky's method. The size of the submatrices of (5.9) is $n \times n$, thus the parallel approach could be worthwhile for bigger systems. D is symmetrical by design, hence we only use $\tilde{B}_i$ and save the computation of $\tilde{C}_i$.

For the last phase (4.5f) of the first dissection level

$$y_i = R_i - Q_i y_{N+2}, \quad i = 1, \dots, N+1 \tag{5.16}$$

we write y_{N+2} as

$$\left(y_{N+2,1}, \ \dots \ , y_{N+2,N}\right)^T := y_{N+2}. \tag{5.17}$$

Therewith, the final solution of our linear system of equation (3.27) is given by

$$y = \begin{pmatrix} \left(R_1^{(j)} - Q_1^{(j,1)} y_{N+2,1}\right)_{j=1,\dots,6} \\ \left(R_2^{(j)} - Q_2^{(j,2)} y_{N+2,1} - Q_2^{(j,1)} y_{N+2,2}\right)_{j=1,\dots,6} \\ \vdots \\ \left(R_N^{(j)} - Q_N^{(j,2)} y_{N+2,N-1} - Q_N^{(j,1)} y_{N+2,N}\right)_{j=1,\dots,6} \\ \left(R_{N+1}^{(j)} - Q_{N+1}^{(j,2)} y_{N+2,N}\right)_{j=3,\dots,4} \\ y_{N+2} \end{pmatrix}. \tag{5.18}$$

A compact pseudocode program of the overall algorithm for evaluating (3.27) is shown in Algorithm 5.8. While writing a real implementation of this algorithm, one has to take care of the sizes of $Q_i^{(2,1)}$, $Q_i^{(4,1)}$, $Q_i^{(2,2)}$, $Q_i^{(4,2)}$, $R_i^{(2)}$ and $R_i^{(4)}$ (and all related factors). These matrices have various numbers of rows, as the number of nonlinear equality constraints may differ between individual horizon steps. A real implementation must also be able to handle zero–height matrices, which occur if no equality constraints are defined for several horizon steps.

Algorithm 5.8: Solving the linear system of equations for search direction computation

```
 1: function SearchDir
 2:     $b \leftarrow P\hat{b}$
 3:     for all $i \in \{1, \dots, N\}$ do                              ▷ parallel evaluation
 4:         $v \leftarrow$ BlockURestr($i$) or BlockU($i$)
 5:         $w \leftarrow$ BlockXRestr($i$) or BlockX($i$)
 6:         $\begin{bmatrix} Q_i^{(1,1)} & Q_i^{(1,2)} & R_i^{(1)} \\ Q_i^{(2,1)} & Q_i^{(2,2)} & R_i^{(2)} \\ Q_i^{(3,1)} & Q_i^{(3,2)} & R_i^{(3)} \\ Q_i^{(4,1)} , & Q_i^{(4,2)} , & R_i^{(4)} \\ Q_i^{(5,1)} & Q_i^{(5,2)} & R_i^{(5)} \\ Q_i^{(6,1)} & Q_i^{(6,2)} & R_i^{(6)} \end{bmatrix} \leftarrow$ BlockMerge($i, v, w$)    ▷ $Q_1^{(j,2)}$ has zero dimension
 7:     end for
 8:     $\begin{bmatrix} Q_{N+1}^{(3,2)} , & R_{N+1}^{(3)} \\ Q_{N+1}^{(4,2)} & R_{N+1}^{(4)} \end{bmatrix} \leftarrow$ BlockXRestrLast or BlockXLast
 9:     for all $i \in \{1, \dots, N\}$ do                              ▷ parallel evaluation
10:         $\tilde{A}_i \leftarrow -Q_i^{(3,1)} - BQ_i^{(5,1)} + Q_{i+1}^{(3,2)}$
11:         $\tilde{D}_i \leftarrow -R_i^{(3)} - BR_i^{(5)} + R_{i+1}^{(3)}$
12:         if $i \geq 2$ then
13:             $\tilde{B}_i \leftarrow Q_i^{(3,1)}$
14:         end if
15:     end for
16:     $y_{N+2} \leftarrow$ solution of (5.15) defined by $A_i$, $B_i$ and $D_i$ (sparse Cholesky's method)
17:     Evaluate the remaining parts of $y$ as defined in (5.18)
18:     return $P^T y$
19: end function
```

Chapter 6

Algorithm implementation

Finally, we have everything we need to implement a fast solver for reachable set problems. It's time to bring everything together, so turn on the text editor and put the final touches on the Makefiles.

In this chapter, we will start with an implementation which is written in Fortran and C++. It will extensively make use of CPU multi–core parallelization. The core of the algorithm will be split into smaller modules. Actually, this is not necessary, but we will use this fragmentation for benchmarking and identifying the module which is most time–consuming of all. Eventually, we will exemplarily port the according module to a CUDA–kernel.

6.1 Parallelization concepts

Until now, we established a general basis by simply meeting as many parallelization requirements as possible, be it for CPU or CUDA hardware. We will now have to do it properly and develop a precise implementation strategy. The strategy aims for running the algorithm *parallely on several CUDA devices* (e.g. several Tesla cards). Since individual CUDA devices are completely independent of each other, a CPU implementation of this strategy can easily be derived by applying the same parallel distribution to the CPU cores and running the individual CUDA–kernel grids as simple serial loops.

Running several CUDA devices in parallel can be done by initializing and using the individual devices within different CPU threads (e.g. POSIX pthread). Each card has its own device memory. The whole system can be considered as a parallelization model *without shared memory*, since a fast data transfer via *GPUdirect* can only be done as peer–to–peer transactions between two devices. During an instant data synchronization of several devices, a single device would have to copy its data a couple of times, depending on the total number of CUDA devices.

On the other hand we can think of a scenario where data synchronization is required, but not instantly. Then, we could make use of the CPU main memory which is accessible by all devices and use a *shared memory* parallelization model. A GPU device can upload its data to the CPU memory where other devices can download and use it. Assuming that memory transfers of different devices usually do not happen at the same time, a single thread is synchronized basically after two memory transactions.

6.1.1 Parallel queue processing

An approach which does not need instant data synchronization is processing the FIFO–buffer of Algorithm 2.2 in parallel by one *worker* per CUDA device.

Each worker pops items from the global FIFO–buffer (which is stored on the CPU memory). The locally stored problem is processed and the result is pushed onto the FIFO–buffer again. Since the number of iterations per problem vary, the workers will not finish their local problems at the same time and colliding accesses to the global buffer can mostly be avoided.

Of course, the ordering of the original FIFO–buffer will be disturbed by the parallel processing. In our case, this is not a big problem. We used a FIFO ordering such that the problems drop out again as soon as possible which keeps the total size of the buffer small. The parallel work will merely cause some local permutations of individual problems which is still fine.

A parallel queue processing must be implemented carefully. While processing, some problems can arise when the buffer runs low. If a worker cannot pop any more problems due to an empty buffer, it must not quit instantly, since another worker could push some new data to the buffer. Therefore we need a mechanism which allows a worker to go to sleep and be awakened again by another worker which has just produced new buffer data. When the last worker comes to an empty buffer (all other workers are already sleeping), it has to wake all other workers so that they can quit the program. An implementation of this mechanism is shown in Algorithm 6.1, which is a modification of Algorithm 2.2.

Algorithm 6.1: Computing a reachable set approximation, parallel queue processing

```
1: function ReachableSetParallel($u_N, y$, ref $\widetilde{\mathcal{R}}_\eta(t)$)    ▷ valid initial guess $u_N, y$
2:     $working \leftarrow$ number of workers
3:     init $\mathcal{F}$                                              ▷ initialize FIFO–buffer
4:     Determine grid point $\tilde{x}$ next to $x_{u(\cdot)}(t)$
5:     $\widetilde{\mathcal{R}}_\eta(t) \leftarrow \{\tilde{x}\}$
6:     for all gridpoints $\tilde{x}_i$ adjacent to $\tilde{x}$ do
7:         push $(\tilde{x}_i, u_N, y)$ onto $\mathcal{F}$
8:     end for
9:     parallel $w \in \{1, \ldots,$ number of workers$\}$
10:        lock mutex
11:        loop
12:            while $\mathcal{F} = \emptyset$ do
13:                $working \leftarrow working - 1$
14:                wait                  ▷ unlock mutex, wait for broadcast, lock mutex
15:                if $working = 0$ then        ▷ woken up by the last worker, quitting...
16:                    unlock mutex
17:                    return
18:                end if
19:                $working \leftarrow working + 1$
20:            end while
21:            pop $(\tilde{x}, \tilde{u}_N, \tilde{y})$ from $\mathcal{F}$
22:            unlock mutex                  ▷ next waiting worker is allowed to continue
23:            $(u_N, y) \leftarrow$ solve the feasibility problem like in Algorithm 2.2
24:            lock mutex
25:            if solution $(u_N, y)$ has been found then
26:                for all gridp. $\tilde{x}_i$ adjacent to $\tilde{x}$ with $\tilde{x}_i \notin \widetilde{\mathcal{R}}_\eta(t)$ do
27:                    push $(\tilde{x}_i, u_N, y)$ onto $\mathcal{F}$
```

```
28:                 end for
29:                 $\widetilde{\mathcal{R}}_\eta(t) \leftarrow \widetilde{\mathcal{R}}_\eta(t) \cup \{\tilde{x}\}$
30:             end if
31:             broadcast                                        ▷ wake waiting workers
32:             if $\mathcal{F} = \emptyset$ and $working = 1$ then
33:                 $working \leftarrow 0$
34:                 unlock mutex
35:                 return
36:             end if
37:         end loop
38:     end parallel
39: end function
```

In this implementation we protect shared variables like *working* or the FIFO–buffer with a mutex. The **wait** command blocks the current worker until another worker calls the **broadcast** command. Prior to this, it unlocks the mutex. When the worker wakes up again, it asks for a mutex lock before continuing. By doing so, we can guarantee that lines 12 to 21 are not processed by multiple workers, though all workers wake up at the same time. Since the buffer could be emptied yet again by another worker who got the mutex lock earlier, we wrap the **wait** command in a loop so that we can go to sleep again without doing anyting.

The commands **wait** and **broadcast** are doing exactly the same as the POSIX pthread library functions `pthread_cond_wait()` and `pthread_cond_broadcast()`. The result $\widetilde{\mathcal{R}}_\eta(t)$ of the algorithm is returned via call–by–reference since parallel worker threads cannot have return values.

6.1.2 Accelerating the linear algebra via CUDA

In the last section we have chosen one worker per CUDA device. Nevertheless, most of the code shown in Algorithm 6.1 still runs on the CPU. Simple queue processing is not a job for CUDA hardware.

GPU parallelization happens in a much more subtle way inside the feasibility solver. We can use the hardware to accelerate basic linear algebra operations like matrix multiply–add–operations and Cholesky decompositions. The changing sizes of non–rectangular matrices during the algorithm runtime make it impossible to write one big GPU–kernel for all computational jobs. The configuration of the kernel execution grid is usually tied to the size of the matrices. Hence, we have to write many small kernels for individual calculation steps. An example workflow is shown in Figure 6.1.

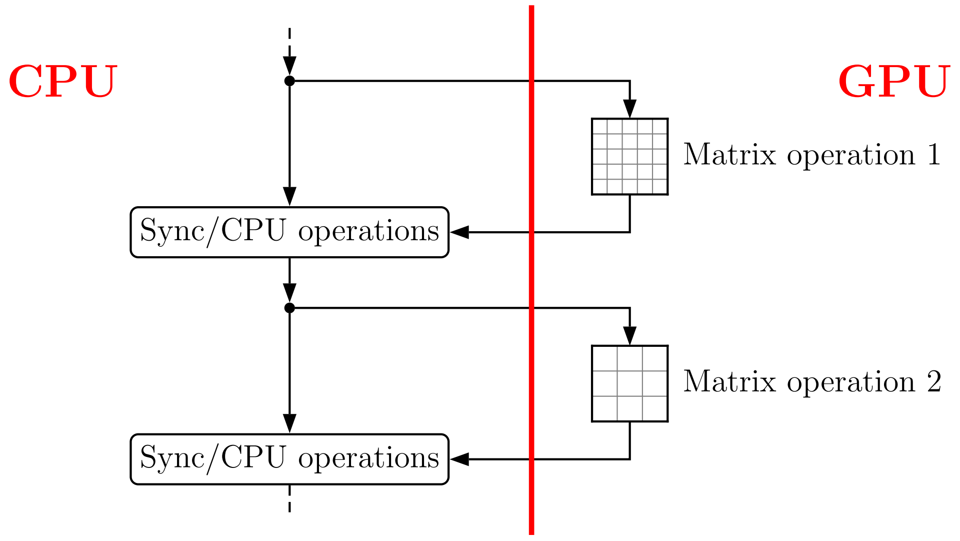


Figure 6.1: Linear algebra computation with two kernels on different grids within a single worker

One could actually come up with the idea of using the cuBLAS library[1] instead of writing own kernels for linear algebra operations. Unfortunately, this library is only efficient when working with large matrices (thousands of matrix entries) and ours are comparatively small. Just to name but a few values: A dynamical system with state dimension $n = 10$, control dimension $m = 4$ and Runge Kutta stages $s = 3$ would lead to matrices with the sizes 4×30, 10×30, 30×30 within the linear algebra operations.

What we can do is to increase the size of the kernel execution grids by processing multiple problems by one worker in parallel. We can modify Algorithm 6.1 such that one worker actually pops a bunch of problems to its local device memory, processes all of them in parallel

[1]The cuBLAS library comes with the CUDA SDK and provides CUDA–accelerated basic linear algebra functions.

and pushes all results back to the buffer. The device kernels must be designed to handle multiple matrix operations on different memory areas at once. Figure 6.2 demonstrates an example workflow of this concept. The number of simultaneously processed problems is limited by the total memory of a single CUDA device.

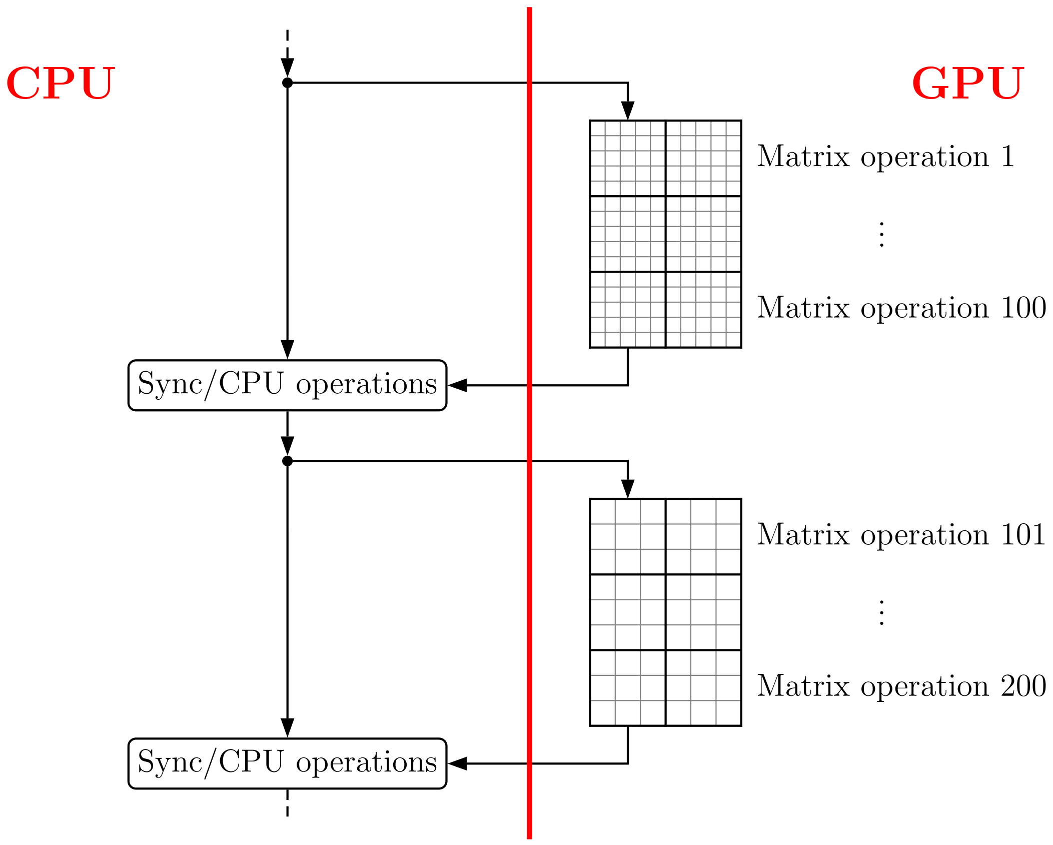


Figure 6.2: Linear algebra computation with two kernels processing multiple problems in parallel within a single worker

6.1.3 Dynamic problem replacing

The concept shown in Figure 6.2 has two disadvantages. First, the buffer must contain enough problems to feed several hungry workers, each with hundreds of problems. Of course, this depends on the actual reachable set. A three–dimensional nice and solid set will produce a much larger number of pending problems than a one–dimensional submanifold. The latter might be completely unsuitable for this approach. The example in Figure 2.4 on page 24 shows that the buffer size can grow very fast, even for two–dimensional problems. So we will simply accept that disadvantage.

The second problem is much more relevant. It will often happen that most of the problems, which are assigned to a worker, have already been finished long before the last problem is solved (the numbers of iteration steps can greatly differ). So the actual grid size of the kernels will decrease very fast, most of the time running just a few problems before the results can be pushed back to the FIFO–buffer.

We can handle this issue, if we enable the solver to instantly replace finished problems by new ones. Actually, we merge iteration steps of the interior point method with the buffer processing algorithm. Instead of executing the feasibility solver as a black box in Algorithm 6.1, a few solver iterations are computed until some finished problems drop out. For every finished problem, we pop a new one from the FIFO–buffer and execute some interior point initializion code before we start the core–iterations on hundreds of problems again. We define this approach more precisely in Algorithm 6.2.

Basically, this algorithm starts like its predecessor:

Algorithm 6.2: Computing a reachable set approximation, multiple worker, each processing multiple problems

```
1:  function ReachableSetMultiProblem($u_N, y$, ref $\widetilde{\mathcal{R}}_\eta(t)$)   ▷ valid init. guess $u_N, y$
2:      $working \leftarrow$ number of workers
3:      init $\mathcal{F}$                                   ▷ initialize FIFO–buffer
4:      Determine grid point $\tilde{x}$ next to $x_{u(\cdot)}(t)$
5:      $\widetilde{\mathcal{R}}_\eta(t) \leftarrow \{\tilde{x}\}$
6:      for all gridpoints $\tilde{x}_i$ adjacent to $\tilde{x}$ do
7:          push $(\tilde{x}_i, u_N, y)$ onto $\mathcal{F}$
8:      end for
9:      parallel $w \in \{1, \ldots,$ number of workers$\}$
10:         init $\mathcal{L}_w$                      ▷ initialize a local array for each worker
11:         lock mutex
12:         loop
13:             while $\mathcal{F} = \emptyset$ do
14:                 $working \leftarrow working - 1$
```

```
15:         wait                              ▷ unlock mutex, wait for broadcast, lock mutex
16:         if $working = 0$ then                ▷ woken up by the last worker, quitting...
17:             unlock mutex
18:             return
19:         end if
20:         $working \leftarrow working + 1$
21:     end while
```

Nothing has changed so far apart from initializing an empty local storage $\mathcal{L}_w$ for each worker. At this point, we modify the original algorithm such that several problems are popped from the buffer until the local memory of the worker's CUDA device is full. For each newly added problem we have to execute some initialization code of the interior point method (variable initialization, first function evaluations, etc...). Actually, we do not simply store $(\tilde{x}, \tilde{u}_N, \tilde{y})$ at $\mathcal{L}_w$ but the data structure which defines the whole interior point process for the individual problem which is based on $(\tilde{x}, \tilde{u}_N, \tilde{y})$.

```
22:     for all free cells of $\mathcal{L}_w$ do
23:         pop $(\tilde{x}, \tilde{u}_N, \tilde{y})$ from $\mathcal{F}$
24:         Store $(\tilde{x}, \tilde{u}_N, \tilde{y})$ on a free cell of $\mathcal{L}_w$
25:         Interior point method pre–processing for $(\tilde{x}, \tilde{u}_N, \tilde{y})$
26:     end for
27:     unlock mutex
```

We now insert the interior point iterations which process all problems which are stored on the workers local memory $\mathcal{L}_w$. The linear algebra operations of these iteration steps can be merged to big kernel calls like shown in Figure 6.2. The loop instantly stops, if a single problem was successfully solved or is considered as infeasible.

```
28:     repeat
29:         parallel for all problems $(\tilde{x}, \tilde{u}_N, \tilde{y})$ stored in $\mathcal{L}_w$
30:             SEARCHDIR                                        ▷ see Algorithm 5.8
31:             Step correction by *fraction to boundary rule* and line search
32:             Apply step and update barrier parameter
33:         end parallel
```

```
34:             **until** some problems finished or failed
```

The rest of the algorithm remains almost untouched. Pushing the new problems to the FIFO–buffer must be done for each new feasible solution.

```
35:             lock mutex
36:             for all finished problems $(\tilde{x}, \tilde{u}_N, \tilde{y})$ stored in $\mathcal{L}_w$ do
37:                 if solution $(u_N, y)$ has been found then
38:                     for all gridpoints $\tilde{x}_i$ adjacent to $\tilde{x}$ with $\tilde{x}_i \notin \widetilde{\mathcal{R}}_\eta(t)$ do
39:                         push $(\tilde{x}_i, u_N, y)$ onto $\mathcal{F}$
40:                     end for
41:                     $\widetilde{\mathcal{R}}_\eta(t) \leftarrow \widetilde{\mathcal{R}}_\eta(t) \cup \{\tilde{x}\}$
42:                 end if
43:                 Free the specific cell of $\mathcal{L}_w$
44:             end for
45:             broadcast                                         ▷ wake waiting workers
46:             if $\mathcal{F} = \emptyset$ and $working = 1$ then
47:                 $working \leftarrow 0$
48:                 unlock mutex
49:                 return
50:             end if
51:         end loop
52:     end parallel
53: end function
```

6.2 CUDA implementation

Algorithm 6.2 is the final version of the reachable set approximation algorithm. It is able to keep several CUDA devices busy and creates a sufficiently large parallelization bandwidth for each device kernel. A pure C++/Fortran implementation of this algorithm is included in the attached CD. It uses the pthread library for running parallel worker threads.

An efficient CUDA implementation of the whole algorithm with custom–made device kernels is extremely time–consuming and not part of this work. But since experiments have shown that the function BlockMerge (Algorithm 5.5), which is part of the SearchDir function (line 30 of Algorithm 6.2) takes most of the computational time[2], we will exemplarly show how a CUDA–version of BlockMerge could be implemented.

BlockMerge must be called for every horizon step (N times per interior point iteration step). By construction, the function calls of the individual horizon steps do not depend on each other and can be executed in parallel.

The function BlockMerge consists of three distinct parts:

1. Prepare D and t by evaluating terms of the type

$$A = B - CD - EF - GH$$

   where the result $A \in \mathbb{R}^{n \cdot s \times n \cdot s}$ will be symmetric (n is the state dimension, s is the number of Runge–Kutta–stages).

2. Compute the Cholesky–decomposition and solve a linear system of equations by using forward– and backward–substitution.

3. Compute the rest of $Q1$, $Q2$ and R by evaluating terms of the type

$$A = B - CD.$$

6.2.1 Matrix Multiply–Add–Operations

The technique of efficiently multiplying matrices on CUDA hardware is well explained within Nvidia's programming guide [25]. Our job is to modify the code, such that p multiplications (problems) can be processed by a single block, if the matrix is small. As the size of a thread block we choose 16×16 since this will lead to nice occupancy on our *Tesla C2050* devices. If the size of A is larger than 16×16, we iterate over submatrices until all elements of A have been calculated (see [25]). If the total number of elements in A is less than 256, we increase

[2]In the next chapter we consider a realistic example where BlockMerge takes 95% of the total computational time.

p. In detail, we calculate the number of grid blocks b and the number of threads per block t by

$$\begin{aligned} t_p &= \min\{16, n \cdot s\}^2 && \text{threads per problem,} \\ p &= \lfloor 256/t_p \rfloor && \text{problems per block,} \\ t &= \min\{p, \text{size}(\mathcal{L}_w)\} \cdot t_p && \text{threads per block,} \\ b &= \lceil \text{size}(\mathcal{L}_w)/p \rceil && \text{blocks.} \end{aligned}$$

The individual threads are mapped to the matrix elements as shown in Figure 6.3. Some threads have to compute two or four elements of the resulting matrix serially in this example while others idle after computing one element. By adjusting the size of the thread block to the individual setting, the idle time can be reduced. If a smaller setting with $p > 1$ is given, a single thread block processes several matrices in parallel (see Figure 6.4).

0	16	32	48	64	80	96	112	128	144	160	176	192	208	224	240	0	16	32	48
1	17	33	49	65	81	97	113	129	145	161	177	193	209	225	241	1	17	33	49
2	18	34	50	66	82	98	114	130	146	162	178	194	210	226	242	2	18	34	50
3	19	35	51	67	83	99	115	131	147	163	179	195	211	227	243	3	19	35	51
4	20	36	52	68	84	100	116	132	148	164	180	196	212	228	244	4	20	36	52
5	21	37	53	69	85	101	117	133	149	165	181	197	213	229	245	5	21	37	53
6	22	38	54	70	86	102	118	134	150	166	182	198	214	230	246	6	22	38	54
7	23	39	55	71	87	103	119	135	151	167	183	199	215	231	247	7	23	39	55
8	24	40	56	72	88	104	120	136	152	168	184	200	216	232	248	8	24	40	56
9	25	41	57	73	89	105	121	137	153	169	185	201	217	233	249	9	25	41	57
10	26	42	58	74	90	106	122	138	154	170	186	202	218	234	250	10	26	42	58
11	27	43	59	75	91	107	123	139	155	171	187	203	219	235	251	11	27	43	59
12	28	44	60	76	92	108	124	140	156	172	188	204	220	236	252	12	28	44	60
13	29	45	61	77	93	109	125	141	157	173	189	205	221	237	253	13	29	45	61
14	30	46	62	78	94	110	126	142	158	174	190	206	222	238	254	14	30	46	62
15	31	47	63	79	95	111	127	143	159	175	191	207	223	239	255	15	31	47	63
0	16	32	48	64	80	96	112	128	144	160	176	192	208	224	240	0	16	32	48
1	17	33	49	65	81	97	113	129	145	161	177	193	209	225	241	1	17	33	49
2	18	34	50	66	82	98	114	130	146	162	178	194	210	226	242	2	18	34	50
3	19	35	51	67	83	99	115	131	147	163	179	195	211	227	243	3	19	35	51

Figure 6.3: Indices of threads, processing the elements of a 20×20 resulting matrix ($p = 1$)

0	9	18	27	36	45	54	63	72
1	10	19	28	37	46	55	64	73
2	11	20	29	38	47	56	65	74
3	12	21	30	39	48	57	66	75
4	13	22	31	40	49	58	67	76
5	14	23	32	41	50	59	68	77
6	15	24	33	42	51	60	69	78
7	16	25	34	43	52	61	70	79
8	17	26	35	44	53	62	71	80

81	90	99	108	117	126	135	144	153
82	91	100	109	118	127	136	145	154
83	92	101	110	119	128	137	146	155
84	93	102	111	120	129	138	147	156
85	94	103	112	121	130	139	148	157
86	95	104	113	122	131	140	149	158
87	96	105	114	123	132	141	150	159
88	97	106	115	124	133	142	151	160
89	98	107	116	125	134	143	152	161

Figure 6.4: Indices of threads, processing the elements of two 9×9 resulting matrices ($p = 2$)

6.2.2 Cholesky's method

An appropriate scheduler for Cholesky's method can be generated as shown in Section 4.2. The data of the scheduler must be stored on CUDA device memory within integer arrays:

$ndiv_i$ defines how many division instructions of the type

$$A_{j,i} \leftarrow A_{j,i}/A_{i,i}$$

happen in the i–th iteration of Cholesky's method.

$scdiv_l$ defines all division instructions consecutively via j–indices. They can be assigned to the i–th iteration by adding up the entries of $ndiv$.

$nelim_i$ defines how many elimination instructions of the type

$$A_{j,k} \leftarrow A_{j,k} - A_{j,i}A_{k,i},$$

happen in the i–th iteration of Cholesky's method.

$scelim_l$ is defined like $scdiv_l$ but stores all (j, k)–tuples in a row.

A CUDA implementation of Cholesky's method is shown in Algorithm 6.3 on the next pages. It is important to understand that device kernels always run in parallel by $blocksize \times$

blockgridsize threads. When writing kernels, these threads must be assigned to data elements. It must be possible to change the data assignment or pause/resume threads during kernel runtime. Defining this behaviour within pseudocode could make the algorithm look confusing due to tons of cases, parallel loops and indices. To handle this high parallelism and thread management, we write the pseudocode such that the whole program is executed on all threads simultaneously and introduce *threadfilters* for nice CUDA–pseudocode which are defined as follows.

Definition 6.1

Let $D \subset \mathbb{N}$ be compact, $\Gamma : D \to \mathbb{N}^s$ be an injective function and i be the total index of a thread within the kernel–blockgrid, i.e.

$$i = blockindex \cdot blocksize + threadindex.$$

Γ defines the behaviour of a kernel thread with index i such that

a) if $i \in D$: thread passes the filter, $\Gamma(i) = (j_1, \dots, j_s)$ is an index tupel which assigns thread i to data elements defined by $j_1, \dots, j_s$,

b) if $i \notin D$: thread i idles.

We call Γ a "threadfilter"*.*

For Cholesky's method, we define the following threadfilter where t_p is the number of threads per problem, p is the number of problems per block, $t = t_p \cdot p$ is the number of threads per block, b is the number of blocks and $D = [0, \text{size}(\mathcal{L}_w) \cdot t_p] \cap \mathbb{N}$:

$$\mathcal{P} \; : \; D \to \mathbb{N}^3, \quad i \mapsto \begin{pmatrix} j_1 \\ j_2 \\ j_3 \end{pmatrix} := \begin{pmatrix} \left\lfloor \frac{i}{t_p} \right\rfloor \\ \left\lfloor \frac{i}{t_p} \right\rfloor \mod p \\ i \mod t_p \end{pmatrix} \tag{6.1}$$

The element $j_1 \in \{0, \dots, \text{size}(\mathcal{L})-1\}$ will be the total index of the problem, $j_2 \in \{0, \dots, p-1\}$ will be the relative index of the problem within the current block and $j_3 \in \{0, \dots, t_p - 1\}$ will be the relative index of the current thread with respect to the current problem.

A new

```
filter $expr$
    $instruction$
    ...
    $instruction$
end filter
```

environment in the pseudo language describes the application of filters on threads. If *expr* (which depends on a thread index i) is a valid expression, the corresponding thread may process the instructions inside the **filter** environment. If not, the thread has to wait until all other threads have finished processing. At the end of the environment, the threads are barrier–synchronized.

We use two types of expressions: $j \leftarrow \Gamma$ means that the threadfilter Γ is evaluated for every thread and the result is stored in a variable j which is defined locally per thread. The **filter** will block any thread for whose index Γ is not defined. The second type is a simple condition which depends on the thread's index. If the condition is true for a single tread, it passes the filter, otherwise it idles.

For easier handling, we use indices starting at zero in this algorithm, even for matrix elements $A_{i,j}$. Algorithm 6.3 requires the state dimension n to be smaller than or equal to the block size t_p to work properly. During runtime, scheduler indices are cached on shared memory, so that every problem which is processed in the same threadblock benefits from one memory access. Let's have a look at the details...

Algorithm 6.3: A CUDA implementation of Cholesky's method

```
1: function CUDACholesky($ndiv$, $scdiv$, $nelim$, $scelim$, $A^{(0)}$, $A^{(1)}$, $\ldots$)
2:     local addr $A$                      ▷ a local pointer for every thread
3:     shared array $C(n,p)$               ▷ shared memory for storing columns
4:     shared array $I(n)$                 ▷ shared memory for storing $j$–indices of $scdiv$
5:     shared array $J(t_p)$               ▷ shared memory for storing $(j,k)$–tuples of $scelim$
6:     shared $c_1 \leftarrow 0$           ▷ cursor for reading $scdiv$
7:     shared $c_2 \leftarrow 0$           ▷ cursor for reading $scelim$
```

Keep in mind that this code is processed by $t \cdot b$ threads in parallel. So far, we declared the types of memory. Shared memory has a threadblock scope such that every threadblock has its own shared data. The variable A is defined locally. We now apply the filter $\mathcal{P}$, defined in (6.1), which blocks out a few threads (since $p \cdot b > \text{size}(\mathcal{L}_w)$ in general) and assigns threads to primary data indices.

```
 8:    filter $(j, k, l) \leftarrow \mathcal{P}$
 9:        $A \leftarrow$ addr of $A^{(j)}$                 ▷ ...for better readability
10:        for $i \leftarrow 0, \ldots, n-1$ do
```

This is the main loop of the algorithm. We now read the i–th column of the matrix into the shared memory. Every problem processed by a thread block has its own shared memory column (index k). We read out the column of A which is a pointer to the matrix of the j–th problem. Since we just need to read the elements $i, \ldots, n$, the first threads of each problem are blocked.

```
11:            filter $i \leq l < n$
12:                $C_{l,k} \leftarrow A_{l,i}$      ▷ copy column $i$ to shared memory for each problem
13:            end filter
14:            if $C_{i,k} \leq 0$ then
15:                Mark problem $j$ as failed
16:                return                            ▷ ...not positive definite
17:            end if
```

If the j–th problem has a indefinite matrix we block the threads which are assigned to this problem for the rest of the function execution. All other threads must continue as only the j–th problem might be affected. After that, we load the next indices of the division scheduler into the shared memory. Only the first $ndiv_i$ threads of the first problem of each thread block are used to copy the indices. All other threads can use them since they are shared. After that we increment the cursor which is processing the *scdiv* array by a single thread per thread block (first thread of the first problem per block).

```
18:            filter $k = 0 \wedge l < ndiv_i$
19:                $I_l \leftarrow scdiv_{c_1+l}$      ▷ load division instructions of scheduler
20:            end filter
```

```
21:         filter $k = 0 \wedge l = 0$
22:             $c_1 \leftarrow c_1 + ndiv_i$
23:             $c_3 \leftarrow 0$
24:         end filter
25:         filter $l = 0$
26:             $C_{i,k} \leftarrow \sqrt{C_{i,k}}$        ▷ square root of the diagonal element
27:         end filter
```

But before we do the division step, we have to take the square root of the i–th diagonal element. Only a single thread per problem is assigned to this step. We let the first thread of each problem do the job.

```
28:         filter $l < ndiv_i$
29:             $C_{I_l,k} \leftarrow C_{I_l,k}/C_{i,k}$        ▷ ...the division instruction
30:         end filter
31:         filter $i \leq l < n$
32:             $A_{l,i} \leftarrow C_{l,k}$
33:         end filter
```

After the division step, the column is completed and can be moved to the global memory again. The elimination step can consist of much more than t_p elimination instructions (since its order is $\mathcal{O}(n^2)$). So we iterate over all scheduled tuples for the i–th step with t_p threads in parallel and increment the cursor by that value. t_p is generally not a multiple of $nelim_i$, so there must be a bit of safeguarding.

```
34:         while $c_3 < nelim_i$ do
35:             filter $c_3 + l < nelim_i$
36:                 filter $k = 0$
37:                     $J_l \leftarrow scelim_{c_2+l}$    ▷ threads of first problem load elimination instr.
38:                 end filter
39:                 local $(\tilde{j}, \tilde{k}) \leftarrow J_l$        ▷ every thread loads elimination instr...
40:                 $A_{\tilde{j},\tilde{k}} \leftarrow A_{\tilde{j},\tilde{k}} - C_{\tilde{j},k} C_{\tilde{k},k}$        ▷ ... and process it
41:             end filter
42:             filter $k = 0 \wedge l = 0$
```

```
43:                 $c_2 \leftarrow c_2 + \min\{nelim_i, t_p\}$            ▷ increase cursors
44:                 $c_3 \leftarrow c_3 + \min\{nelim_i, t_p\}$
45:             **end filter**
46:         **end while**
47:     **end for**
48:     Mark problem $j$ as *successful*
49: **end filter**
50: **return**
51: **end function**
```

We can read the column–values for the elimination instructions from the shared memory. But the read–write access at $A_{\tilde{j},\tilde{k}}$ can be anywhere left from the i–th column in the lower diagonal part of the matrices. As we assume the matrices to be too large for shared memory caching, we have to count on coalesced global memory transactions. To ensure as much coalesced[3] memory reads and writes as possible, we must *sort the (j,k)–tuples of the scheduler by k and with second priority by j*. By doing so, the elimination steps will be processed column–by–column by t_p threads in parallel. This is the ordering of the matrix in the memory which ensures coalesced memory transactions.

An intelligent preordering to reduce fill–ins can easily be applied to the algorithm by permuting the matrix indices before accessing the elements. This can already be done by the scheduler to save memory accesses to the permutation array during runtime. A further improvement is to skip zero–elements while copying a column to the shared memory. By "densing" them to a small shared memory array, the algorithm can process even larger, but sparse, matrices. The scheduler should be constructed such that it uses the densed indices on the shared memory cache.

A detailed CUDA–implementation of Algorithm 6.3 (including the improvements of the previous paragraph) can be found on the CD which is attached to this work. This implementation also includes CUDA kernels for forward– and backward–substitution. Parallel CUDA implementations of these substitutions can be developed in the same way as demonstrated on Cholesky's method (dependency graph, scheduler, etc...) and will not be discussed here.

[3]See Section 1.2.3.

The only interesting thing which is different to the implementation of Cholesky's method is the fact that forward substitution processes column–data in parallel while backward substitution accesses the matrices row–by–row in parallel. Actually, this is a problem, since a matrix is stored column–by–column in the memory and a parallel access to a row cannot be coalesced. Therefore we simply copy and transpose the lower triangular part of the Cholesky decomposition to the upper right area of the matrix. This generates a symmetric matrix and the row–by–row access of the backward–substitution can be implemented as coalesced column–operations. An efficient matrix transpose implementation is described in [16].

6.2.3 Cloning intermediate results on device memory

Since we only implement a small part of the algorithm exemplarily on CUDA hardware, we have to clone all required matrices on device memory as they are needed by CPU *and* GPU. This affects almost every intermediate result like $\tilde{y}_{i,1}, \dots, \tilde{y}_{i,10}$ and also function evaluations $\partial_x f_i$, $\partial_u f_i$, $\partial_x f_i^A$ and parts of the right–hand side of the linear system of equations $b_{i,5}$, $b_{i,6}$ for all $i = 1, \dots, N$. The results $Q1$, $Q2$ and R must be copied back to CPU memory. Yes, all in all this is indeed as ugly as it sounds and will significantly lower the overall efficiency of the CUDA implementation.

Fortunately, we can abate the pain by using *Pinned (page–locked) memory* and *Asynchronous memory transfers*. Usually, the OS virtualizes the system memory by using *pages*. Those pages can be moved within the physical memory and also be swapped to disk without changing pointers which are actually used by active programs. This is good, since the OS can optimize the memory usage and provide large free areas.

But due to this fact, data must be copied *twice*[4] during a memory transfer from a CPU host to GPU device (also in the other direction). First, a memory block is copied from the paged system memory to a small area which is not paged by the OS. This operation will be managed by the CPU. After that, a DMA controller moves the memory block to the CUDA device memory which works asynchronously, so the CPU is not occupied and can do something else.

The CUDA driver provides the functionality of allocating host memory such that it will

[4]This is done automatically by the CUDA driver.

not be paged by the OS (as we said above: the pinned or page–locked memory). By doing so, the DMA controller can directly access the memory and CPU work is almost unnecessary for the whole transfer. The big advantage is that a memory transfer and host code execution can be done in parallel. In our case, computing $\tilde{y}_{i,1}, \dots, \tilde{y}_{i,10}$ and all other matrices which are needed by the kernel need some time. E.g. we can start copying $\tilde{y}_{i,1}$ to device memory while already computing $\tilde{y}_{i,2}$ on the CPU. The additional time which is needed for cloning the data will be minimal.

The disadvantage of using page–locked memory is serious. Using too much of pinned memory unables the OS to rearrange the memory pages which produces small free "holes". These holes are useless for bigger allocations. The efficiently usable memory will be reduced and allocations start to fail much earlier even if there is actual free host memory left. To make sure that a big amount of pinned memory can be used without destabilizing the whole system, the user should have at least twice as much host memory as device memory (which is a rule of thumb). More information about page–locked memory can be found in the CUDA programming guide [25] and in NVidia's developer blog [14].

6.2.4 Concurrent kernel execution

Since the introduction of Fermi–architecture (like on our Tesla C2050), a CUDA device has three processing engines which run completely independently from each other:

- **H2D–Engine:** Copies data from host memory to device memory. Only one transfer can be processed at a time.
- **Kernel–Engine:** Executes device kernels. Multiple device kernels can be executed simultaneously as long as they "fit" on the GPU (they share ressources like shared memory, registers, etc.).
- **D2H–Engine:** Copies data from device memory to host memory. Only one transfer can be processed at a time.

All of these engines work asynchronously such that a host program can start these engines and instantly continues running host code. To feed these engines, asynchronous CUDA–API calls can be executed on several *streams*. When calling API functions on the same stream, the host code will block when another action (like a memory transfer or kernel execution) is currently processed on that stream. To feed all three engines in parallel, the API calls must be done on different streams. Even if using several streams, the host code can block when an engine is currently occupied in processing other streams. For example, a Host–to–Device API function call will block if another Host–to–Device transfer is currently processed. A kernel execution may block if the Kernel–Engine has no more ressources of running an additional kernel. Figure 6.5 shows an example execution with three streams and how it is actually executed on the three processing engines.

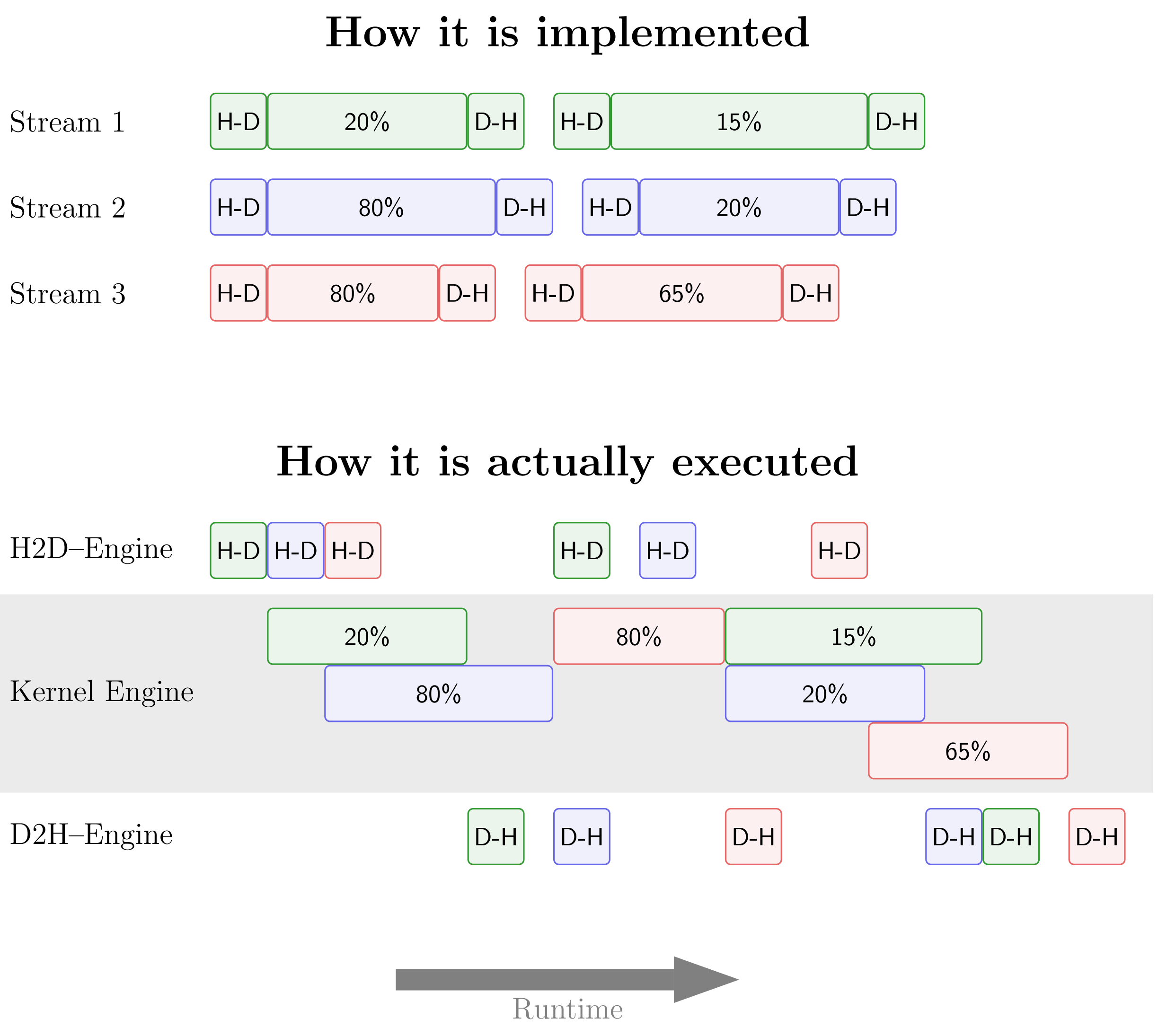


Figure 6.5: Example processing of three streams by the H2D–, Kernel– and D2H–Engine of a CUDA device. The percentage is a simplification of required kernel ressources.

Efficient asynchronous API calls must be implemented very carefully. Unintented blocking on API calls can happen very easily. A complete guide to good CUDA scheduling can be found in the NVidia developer blog [15].

In our case, we have to run the kernels for each horizon step (namely N times). So we assign each horizon step to one of up to 16 CUDA streams (which is the maximum on Fermi–architecture) since every horizon step can be processed independently during the BlockMerge calls. The host–to–device memory transfers (which is the cloning of the intermediate results) has already been explained in the last subchapter. To optimize the device–to–host transfers, we have to arrange the resulting matrices for every processed problem such that they can be copied by a single memory transaction per horizon step.

Chapter 7

A real world application: The DEOS mission

We finally want to test the whole implementation with an example from real life. A nice and very challenging application is the so called *DEOS mission* (Deutsche Orbitale Servicing Mission).

The goal of this mission is to launch a satellite which is able to dock on uncontrolled and inactively tumbling obstacles (targets) in orbit and initiate a controlled reentry. Optimally controlled docking maneuvers have already been discussed in [5,20,21]. In this work, we will use this problem setting to compute a controllability set, i.e. the set of all starting points from which a successful docking maneuver can be carried out within a given time span.

7.1 Problem definition

Since a detailed description of the satellite model can be looked up in [5,20,21], we will just use the results in this work.

We define a LVLH (local vertical, local horizontal coordinate) system such that the origin lies within the center of the target. The x–axis is an extension of the straight line which goes through the center of the earth and the center of the target, pointing away from earth. The y–axis points in the direction of orbital movement of the target and the z–axis is chosen such that it completes the orthogonal tripod (see Figure 7.1).

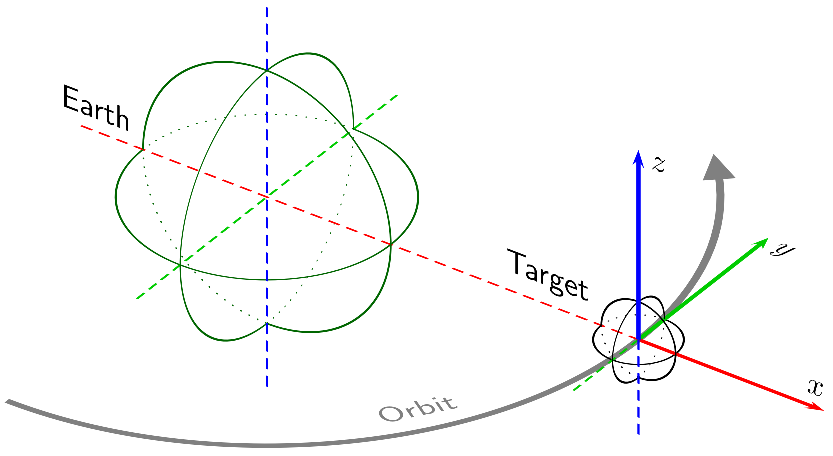


Figure 7.1: Definition of the LVLH system

We assume, that the target travels with constant orbital speed and height. Hence, we can model the satellite's position dynamics via *Clohessy-Wiltshire equations*

$$\ddot{x} = 2n\dot{y} + 3n^2x + \frac{v_x}{M} \tag{7.1a}$$
$$\ddot{y} = -2n\dot{x} + \frac{v_y}{M} \tag{7.1b}$$
$$\ddot{z} = -n^2z + \frac{v_z}{M} \tag{7.1c}$$

where n is the mean motion of the target, M is the mass of the satellite and (v_x, v_y, v_z) is a three dimesional acceleration control vector made by thrusters with respect to the LVLH system.

The orientation dynamics of the satellite is expressed via quaternions (q_i, q_j, q_k, q_l), where (q_i, q_j, q_k) is the vector part and q_l is the scalar part. We use a BFC (body–fixed coordinates) system which coincides with the LVLH system if the satellite is unrotated and positioned at the LVLH–origin. As indices of the BFC system we choose $\{1, 2, 3\}$ as opposed to the $\{x, y, z\}$ indices which describe the elements of the LVLH system. The orientation dynamics can be modelled as

$$\dot{\omega}_1 = \frac{1}{J_{11}}\left(\omega_2\omega_3(J_{22} - J_{33}) + m_1\right) \tag{7.1d}$$
$$\dot{\omega}_2 = \frac{1}{J_{22}}\left(\omega_1\omega_3(J_{33} - J_{11}) + m_2\right) \tag{7.1e}$$

$$\dot{\omega}_3 = \frac{1}{J_{33}}\left(\omega_1\omega_2(J_{11}-J_{22})+m_3\right) \tag{7.1f}$$

$$\dot{q}_i = \frac{1}{2}\left(q_j\omega_3 - q_k\omega_2 + q_l\omega_1\right) \tag{7.1g}$$

$$\dot{q}_j = \frac{1}{2}\left(-q_i\omega_3 + q_k\omega_1 + q_l\omega_2\right) \tag{7.1h}$$

$$\dot{q}_k = \frac{1}{2}\left(q_i\omega_2 - q_j\omega_1 + q_l\omega_3\right) \tag{7.1i}$$

$$\dot{q}_l = \frac{1}{2}\left(-q_i\omega_1 - q_j\omega_2 - q_k\omega_3\right) \tag{7.1j}$$

where $\omega_1, \omega_2, \omega_3$ are angular velocities, (m_1, m_2, m_3) is the momentum control vector with respect to the satellite and J_{11}, J_{22}, J_{33} are the diagonal elements of the inertia tensor which is assumed to coincide with the principal axis of the satellite.

Alltogether we get a 13–dimensional first order differential equation which describes the dyamics of the satellite. The target can be modelled with an additional differential equation. By construction, the target does not change its position. Hence, we just use the orientation dynamics for the target simulation which is a 7–dimensional first order differential equation.

To model the docking condition, we introduce the docking points d^S and d^T of the satellite and the target (Figure 7.2). A successful docking maneuver requires matching docking points $d^S = d^T$ and velocity of docking points $\dot{d}^S = \dot{d}^T$ for a short moment. In order to formally define this condition in the coordinate system we are using the rotation matrix

$$R = \begin{pmatrix} q_i^2 - q_j^2 - q_k^2 + q_l^2 & 2(q_iq_j - q_kq_l) & 2(q_iq_k + q_jq_l) \\ 2(q_iq_j + q_kq_l) & -q_i^2 + q_j^2 - q_k^2 + q_l^2 & 2(q_jq_k - q_iq_l) \\ 2(q_iq_k - q_jq_l) & 2(q_jq_k + q_iq_l) & -q_i^2 - q_j^2 + q_k^2 + q_l^2 \end{pmatrix} \tag{7.2}$$

which transforms BFC vectors to LVLH vectors. The docking condition can then be modelled as

$$d^S - d^T = \begin{pmatrix} x \\ y \\ z \end{pmatrix} + R^S \begin{pmatrix} d_1^S \\ d_2^S \\ d_3^S \end{pmatrix} - R^T \begin{pmatrix} d_1^T \\ d_2^T \\ d_3^T \end{pmatrix} = 0 \tag{7.3a}$$

$$\dot{d}^S - \dot{d}^T = \begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \end{pmatrix} + R^S \begin{pmatrix} \omega_1^S \\ \omega_2^S \\ \omega_3^S \end{pmatrix} \times R^S \begin{pmatrix} d_1^S \\ d_2^S \\ d_3^S \end{pmatrix} - R^T \begin{pmatrix} \omega_1^T \\ \omega_2^T \\ \omega_3^T \end{pmatrix} \times R^T \begin{pmatrix} d_1^T \\ d_2^T \\ d_3^T \end{pmatrix} = 0 \quad (7.3b)$$

where the S and T superscripts indicate the components of the dynamical systems of satellite and target respectively.

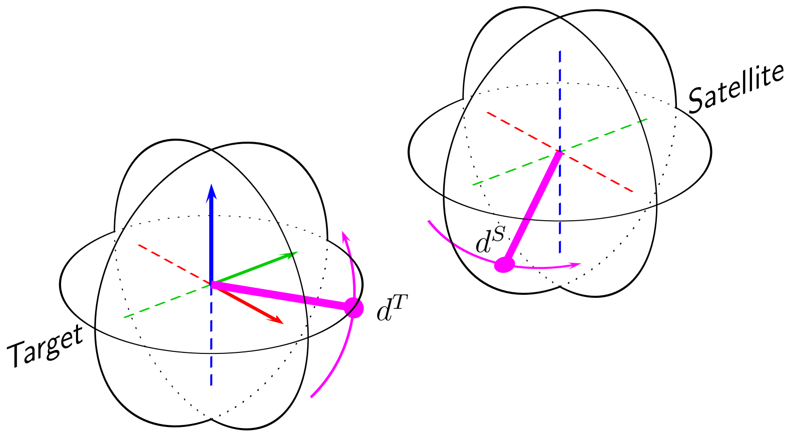


Figure 7.2: Docking points and their moving directions (due to body rotation) of satellite and target

In addition to the docking condition we want to avoid a collision and have to restrict the control values to their physical limit. So we introduce a safety distance $\delta_{\min}$ and define state constraints

$$x^2 + y^2 + z^2 \geq \delta_{\min}^2 \quad (7.4)$$

for all trajectory points. For reasons of numerical stability, we choose

$$\delta_{\min} = 0.95(\|d^S\| + \|d^T\|) \quad (7.5)$$

which provides a little bit of freedom while searching for a docking point. The control variables are restricted by box constraints

$$-v_{\max} \leq v_1, v_2, v_3 \leq v_{\max}, \quad -m_{\max} \leq m_1, m_2, m_3 \leq m_{\max} \quad (7.6)$$

and the acceleration control variables of the LVLH system in (7.1) can be obtained by

$$\begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix} = R^S \begin{pmatrix} v_1 \\ v_2 \\ v_3 \end{pmatrix}. \tag{7.7}$$

The satellite must stand still and unrotated when starting the docking maneuver. Hence, the start of the trajectory is restricted by

$$0 = \dot{x} = \dot{y} = \dot{z} = \dot{\omega}_1 = \dot{\omega}_2 = \dot{\omega}_3 \tag{7.8a}$$

$$0 = q_i = q_j = q_k \tag{7.8b}$$

$$1 = q_l. \tag{7.8c}$$

The goal of this chapter is to compute the approximation controllability set

$$\mathcal{R}(0) := \left\{ \mathbf{x}(0) \in \mathbb{R}^{13} \;\middle|\; \begin{array}{ll} \mathbf{x}(\cdot) & \text{solves (7.1) with control function } \mathbf{u}(\cdot) \\ \mathbf{x}(0) & \text{fulfills (7.8)} \\ \mathbf{x}(T) & \text{fulfills (7.3), (7.4)} \\ \mathbf{u}(\tau) & \text{fulfills (7.6)}, \tau \in [0,T] \end{array} \right\} \tag{7.9}$$

where $\mathbf{x}$ is the full state vector

$$\mathbf{x} := (x, y, z, \dot{x}, \dot{y}, \dot{z}, \omega_1, \omega_2, \omega_3, q_i, q_j, q_k, q_l)^T \tag{7.10}$$

and $\mathbf{u}$ the full control vector

$$\mathbf{u} := (v_1, v_2, v_3, m_1, m_2, m_3)^T. \tag{7.11}$$

Condition (7.8) restricts $\mathcal{R}(0)$ to a three–dimensional submanifold, since $\dot{x}$, $\dot{y}$, $\dot{z}$, ω_1, ω_2, ω_3,

q_i, q_j, q_k, q_l are fixed. Thus, computing the projection

$$\left\{ \begin{pmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \mathbf{x}_3 \end{pmatrix} \in \mathbb{R}^3 \;\middle|\; \mathbf{x} \in \widetilde{\mathcal{R}}_\eta(0) \right\} \tag{7.12}$$

preserves the full set information and can be done with moderate requirements to storage memory.

7.2 Finding the first feasible trajectory

Since the algorithm is heavily dependent on a good initial guess we have to find a single valid trajectory. One could try to simply run an optimal control problem and use its solution as initial guess. But particularly on larger control horizons these optimal control algorithms might struggle and take a long time.

In this section we discuss a concept of finding an initial guess which is based on the specific anatomy of the satellite example and may be not very useful for a general case. At first, we try to find a valid docking position for the satellite. After that we simulate the satellite's trajectory backwards in time and try to stabilize the state such that it converges to a valid vector in sense of initial condition (7.8). We will not satisfy the initial condition exactly but we can start the reachable set algorithm with a trajectory which is at least very close to all constraints.

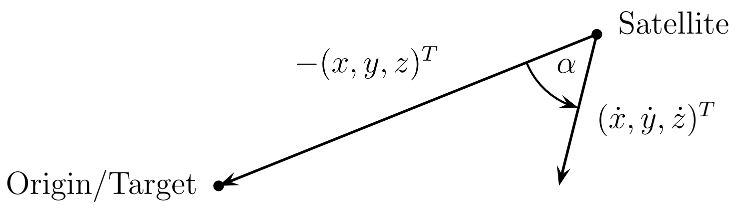
7.2.1 Computing a valid docking point

The first step will be to find a valid state which satisfies the docking condition (7.3). As there is no unique solution for this problem, we define a nonlinear optimization problem

$$\min_{\mathbf{x}} \lambda \sum_{i=1}^{3} \mathbf{x}_i^2 + \sum_{i=4}^{12} \mathbf{x}_i^2 + (\mathbf{x}_{13} - 1)^2 \tag{7.13}$$

with respect to (7.3) and (7.4) as constraints. The optimal value of the objective function of this optimization problem is not important for the solution, since we are just searching for a feasible docking point. But the objective function pushes the docking point towards a state which might already be very close to the starting condition (7.8). The weight λ of the (not so important) position (x, y, z) can be chosen quite small but is necessary for a well conditioned problem formulation.

Since the safety distance $\delta_{\min}$ is a little relaxed towards the target, it is possible to find state vectors $\mathbf{x}$ which fulfill the docking condition and the safety distance condition, but have velocity components that let the satellite come from the *inside* of the target. To avoid this case, we have to use an additional restriction which makes the velocity vector of the satellite point towards the target. We develop this condition with the help of simple geometry:



Obviously, the satellite is moving towards (or at least parallel with) the target if $\alpha \leq 90$. The inequality constraint

$$\left\langle \begin{pmatrix} x \\ y \\ z \end{pmatrix}, \begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \end{pmatrix} \right\rangle \leq 0 \tag{7.14}$$

forces this condition.

7.2.2 Generating an almost–feasible trajectory

After computing a valid docking point we use the concept of MPC (model predictive control, [13]) to generate a backwards trajectory which ends (well, actually starts) near a valid starting point. Therefore we choose a cost function

$$l(\mathbf{x}, \mathbf{u}) = (\mathbf{x} - a)^T \Lambda_x (\mathbf{x} - a) + \lambda_u \mathbf{u}^T \mathbf{u} \tag{7.15}$$

where a is an offset vector, Λ_x is a positive definite diagonal matrix for applying individual costs each element of $\mathbf{x}$, λ_u is a weight for the controls. We choose a such that the controlled trajectory is pulled towards (7.8) at a position where we assume the center of the controllability set.

With a MPC horizon $\widetilde{N} \leq N$ we can define a moving horizon subproblem for horizon steps $i = N - \widetilde{N}, \ldots, 0$

$$\min_{\substack{\mathbf{x}_{i,\ldots,i+\widetilde{N}-1} \\ \mathbf{u}_{i,\ldots,i+\widetilde{N}-1}}} \sum_{j=i+1}^{i+\widetilde{N}-1} l(\mathbf{x}_j, \mathbf{u}_j) + \beta l(\mathbf{x}_i, \mathbf{u}_i)$$

$$\text{s.th.} \quad \left.\begin{array}{l} \mathbf{x}_{j+1} = \Phi(\eta, \mathbf{x}_j, \mathbf{u}_j) \\ \mathbf{x}_j, \mathbf{u}_j \text{ fulfill (7.4) and (7.6),} \end{array}\right. \quad j = i, \ldots, i + \widetilde{N} - 1$$

where $\mathbf{x}_j, \mathbf{u}_j$ are the state and control vectors at timestep j, β is a scaling factor to emphasize the subproblem's last element of the backwards trajectory (this coincides with the *terminal weights* of the common MPC algorithm [13]) and $\Phi(\eta, \mathbf{x}, \mathbf{u})$ is a numerical method to solve the corresponding differential equation η seconds in forward time, beginning at $\mathbf{x}$ with applied control $\mathbf{u}$. It is recommened to use the same Runge–Kutta method like in the reachable set algorithm in order to use the temporary stage variables for the initial guess generation.

Note that the MPC horizon is shifted backwards while the trajectories of the subproblems are generated forwards in time. In the first MPC step, we choose $\mathbf{x}_N = d^S$ as a valid docking point and terminal condition of the subproblem. After that, in the i–th MPC step, $\mathbf{x}_{i+\widetilde{N}}$ is not in the set of optimized variables any more and defines the new constant terminal condition.

By solving the subproblems, we iteratively generate a full initial guess for the reachable set algorithm. The quality of the resulting initial guess heavily depends on the proper choice of the weights Λ_x and λ_u and the MPC horizon length $\widetilde{N}$. Position components of the offset vector a can also worsen the result if they are far away from the reachable set.

Since the dynamics are very insensitive to the controls (respectively, the satellite's controls are very weak compared to the mass), the derivatives within the optimization problem are quite flat. The nonlinear program must be built carefully and the time step η should be big enough to handle this issue. Altogether, sometimes this method seems to require a few attempts until a good parameter setting has been found for a given target orientation.

7.3 Numerical results

The CD contains a pure CPU implementation (C++ with Fortran kernels) and another implementation which has a CUDA version of the functions BlockX(Restr), BlockU(Restr) and BlockMerge. We will compare the runtime of both variants in order to get an idea of the efficiency of the CUDA implementation.

Our testing environment has the following parameters:

Model parameters	
orbit radius $[m]$	$r_x = 7071000$
gravitational constant $[N(\frac{m}{kg})^2]$	$G = 398 \cdot 10^{12}$
mean motion $[\frac{1}{s}]$	$n = \sqrt{\frac{G}{r_x^3}}$
satellite mass $[kg]$	$M = 200$
maximum thrust $[N]$	$v_{\max} = 0.15$
maximum torque $[Nm]$	$m_{\max} = 1$
satellite angular mass in x $[\frac{kg^2}{m}]$	$J^S_{11} = 2000$
satellite angular mass in y $[\frac{kg^2}{m}]$	$J^S_{22} = 5000$
satellite angular mass in z $[\frac{kg^2}{m}]$	$J^S_{33} = 2000$
target angular mass in x $[\frac{kg^2}{m}]$	$J^T_{11} = 1000$
target angular mass in y $[\frac{kg^2}{m}]$	$J^T_{22} = 2000$
target angular mass in z $[\frac{kg^2}{m}]$	$J^T_{33} = 1000$
docking point of satellite $[m]$	$(d^S_1, d^S_2, d^S_3) = (0, -1, 0)$
docking point of target $[m]$	$(d^T_1, d^T_2, d^T_3) = (0, 1, 0)$
minimum distance $[m]$	$\delta_{\min} = 1.9$

Simulation parameters	
horizon length	$N = 30$

Simulation parameters	
step size $[s]$	$\eta = 3$
ODE method	Radau IA, order 5, $s = 3$ (see [17])
target trajectory start (angular velocity)	$(\omega_1^T, \omega_2^T, \omega_3^T) = (-0.02, 0.0349, 0.057453)$
target trajectory start (orientation)	$(q_i^T, q_j^T, q_k^T, q_l^T) = (-0.05, 0, 0, 0.99875)$

Initial guess computation	
docking point position weights	$\lambda = 1$
state weights	$\Lambda_x = \mathrm{diag}(10^{-4}, 10^{-4}, 10^{-4},$ $10, 10, 10, 10^3, 10^3, 10^3, 10^4, 10^4, 10^4, 10^4)$
control weight	$\lambda_u = 10^{-4}$
state offset	$a = (0, -1.2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1)$
end cost weight	$\beta = 1$
MPC horizon length	$\widetilde{N} = 7$
optimizer	Ipopt v3.11.8

Reachable set	
domain	$\bar{\mathcal{X}} = [-2, 6] \times [-5, 1.5] \times [-3.5, 4.2]$
grid size	$(G_1, G_2, G_3) = (64, 64, 64)$
approx. grain size	$\varepsilon_{\mathcal{G}} = 0.125$

Feasibility check	
initial barrier parameters	$\mu_0 = 0.99$, $\tau_0 = 0.005$
relative accuracy	$\nu = 10^{-2}$
warmstart slack variable shift	$+10 \cdot \nu\varepsilon_{\mathcal{G}}$
warmstart barrier parameter	$\mu = \max\{\min\{\mu_0 \cdot 10^4, 0.9\}, 10^{-5}\}$
hessian approximation factor	$\xi = 0.2$
max. interior point iteration steps	100

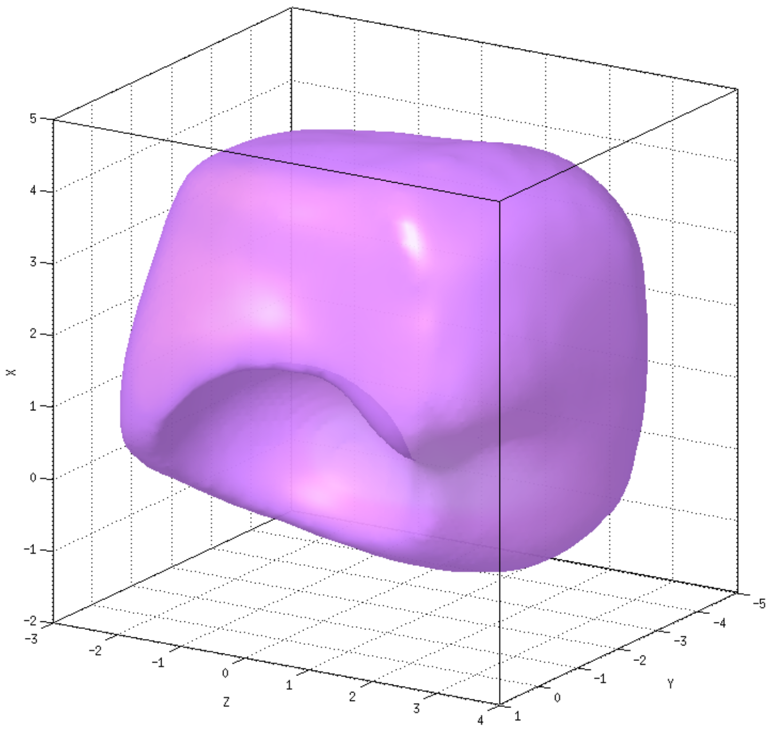
Feasibility check	
divergence check steps	10
local buffer (CPU variant)	size($\mathcal{L}_w$) = 10
local buffer (CUDA variant)	size($\mathcal{L}_w$) = 100

Runtime Environment	
CPUs	2× Intel Xeon E5620, 4 cores, 2.4 GHz
host memory	24 GB
operating system	Ubuntu 14.04 LTS, x64
CUDA devices	4× Nvidia Tesla C2050
device memory per device	3 GB per device
CUDA driver version	6.0

Figures 7.3 and 7.4 show the result of the algorithm. For the visualization, reachable grid points have been considered as cubes. A mesh of triangles has been computed which covers all visible faces of the cubes. The surface was smoothened by performing two steps:

1. Each vertex has been replaced by its average over all vertices which are connected with the original vertex (indirectly) over up to two edges.
2. Each vertex was moved up to half of the cell size such that the total length of all edges is minimal.

These smoothening steps increase the maximum discretization error by $\frac{3}{2}\varepsilon_{\mathcal{G}}$, which is still tolerable. A nice gouraud shading adds a vivid finishing. The figure also illustrates the minimum distance $\delta_{\min} = 1.9$ to the target (located at the origin) as a ball shape which is cut out of the set.



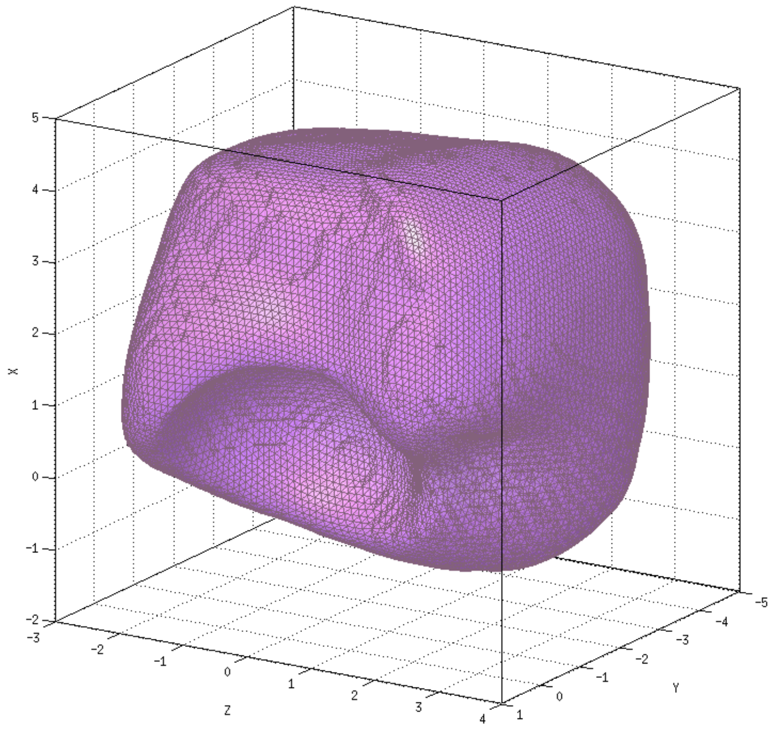


Figure 7.3: Smoothened illustration of the controllability set (front)

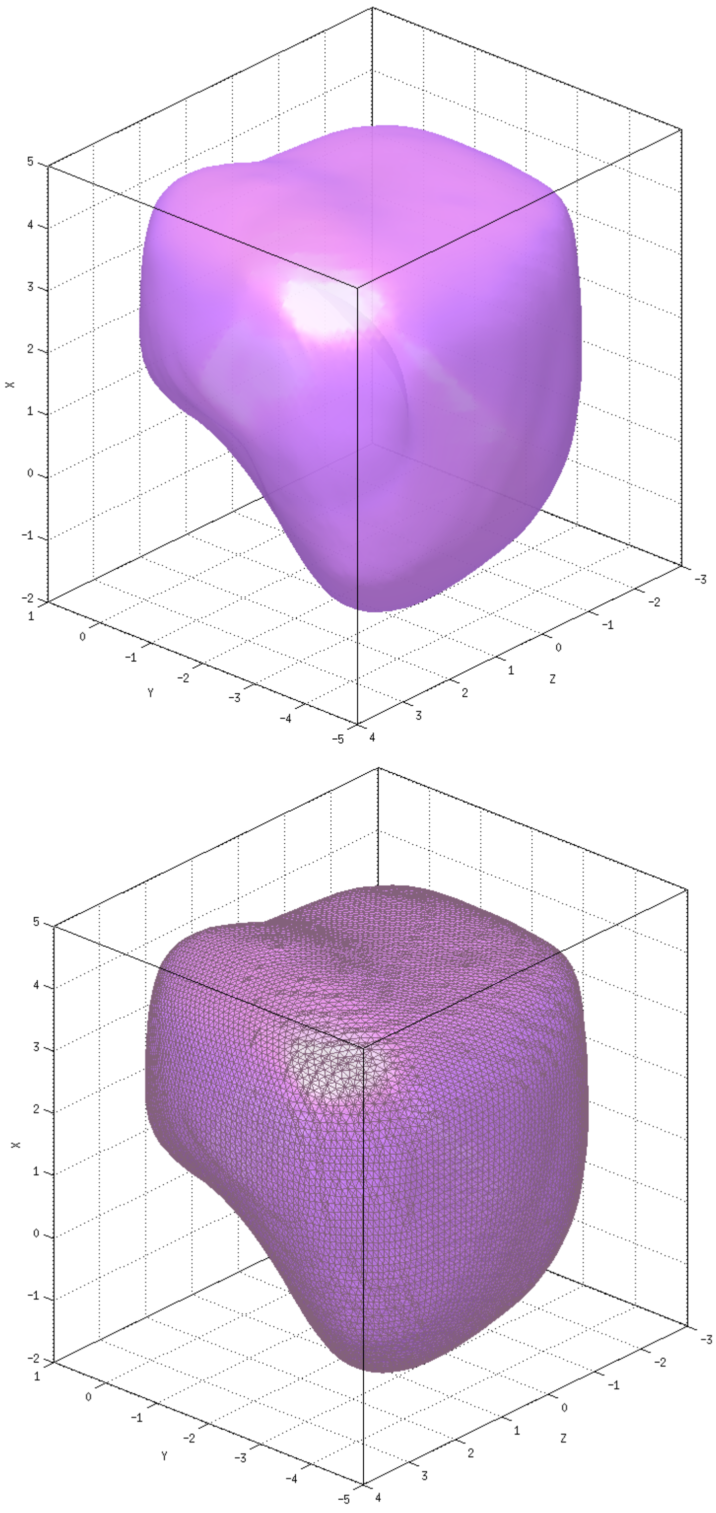


Figure 7.4: Smoothened illustration of the controllability set (back)

We will first do some time measurements on the CPU variant. The following table shows the runtime in hours while using a different number of CPU cores. The *func. time* column displays the time spent on executing the BLOCKX(RESTR), BLOCKU(RESTR) and BLOCKMERGE functions in order to compare them to the CUDA implementation afterwards. We also compute the ratio of the function execution time and the total runtime. The efficiency of parallelization is given by

$$\text{effiency} = \frac{\text{runtime of single core variant}}{\text{runtime of parallel variant} \times \text{number of cores}} \times 100\%$$

cores	**Intel Xeon E5620**			
	total time [h]	*func. time [h]*	*ratio*	*efficiency*
1	10:50	9:42	89.5%	100.0%
2	5:22	4:48	89.4%	100.8%
3	3:36	3:13	89.3%	100.1%
4	2:42	2:24	89.0%	100.6%
5	2:14	2:00	88.9%	96.5%
6	1:53	1:41	89.0%	95.7%
7	1:37	1:27	89.0%	95.0%
8	1:25	1:15	88.6%	95.5%

An efficiency value $> 100\%$ might be confusing at a first glance. Changing the number of queue workers will change the order of processed problems (see Section 6.1.1). This will change pathes of recursive spreading within the set (see Section 2.2.3 and Figure 2.2) and the number of actually computed feasibility checks. As a consequence, the efficiency value will be a bit “blurry”. In our experiments it seems like, the single–core run had more failed feasibility checks than the run on two cores, so the calculated efficiency value is greater than 100%.

The noticeable performance decrease when activating te fifth core could be caused by memory bus conflicts as the second CPU starts working. But this is just a guess.

The same measurement is done with up to four Tesla cards and the CUDA variant of the algorithm:

devices/	**Nvidia Tesla C2050** (with Intel Xeon E5620 host)			
cores	*total time [h]*	*func. time [h]*	*ratio*	*efficiency*
1	2:54	1:35	54.8%	100.0%
2	1:31	0:48	52.4%	95.4%
3	1:01	0:32	52.2%	95.2%
4	0:46	0:25	54.9%	94.3%

The following table compares the speedup factors (CPU runtime divided by GPU runtime) for different numbers of cores and devices.

GPU \ CPU	1	2	3	4	5	6	7	8
1	$\times$3.7	$\times$1.9	$\times$1.2	$\times$0.9	$\times$0.8	$\times$0.6	$\times$0.6	$\times$0.5
2	$\times$7.1	$\times$3.5	$\times$2.4	$\times$1.8	$\times$1.5	$\times$1.2	$\times$1.1	$\times$0.9
3	$\times$10.7	$\times$5.3	$\times$3.5	$\times$2.7	$\times$2.2	$\times$1.9	$\times$1.6	$\times$1.4
4	$\times$14.1	$\times$7.0	$\times$4.7	$\times$3.5	$\times$2.9	$\times$2.5	$\times$2.1	$\times$1.8

Since a multicore CPU and multiple GPU cards are completely different hardware architectures, it is not easy to draw inferences from the runtime of the algorithm about the actual efficiency. The good news are: The CUDA–implementation does really fail as it can be considered as faster than the CPU version (this is not self–evident in the context of CUDA programming!). But with regard to the real costs of the CUDA implementation (time of development, electricity cost, cost of purchase,...) the CPU variant can be clearly considered as the cheaper way to go.

Chapter 8

Conclusions

Compared to the original distance function approach, the new algorithm is fast, very reliable[1] (as shown in Chapter 2) and promising. As long as the resulting set has a modest dimension, high dimensional dynamics (like a satellite in earth orbit) can be considered and processed in acceptable time which is a great result.

8.1 Tuning the parameters

Unfortunately, the reliability of the algorithm depends on a good choice of some specific parameters like ξ (see Section 2.2.1). Besides, an interior point method always comes with some difficulties when using warm starting data. Slack variables must be shifted a little and μ_0 (see Section 3.1.5) should be chosen wisely.

A good start of further research could be to adapt the parameters automatically to the problem while proceeding the buffer. Since we are actually solving thousands of almost similar problems, the algorithm could "learn" somehow which parameter values lead to the best results. For example, one could vary the value of ξ while processing a few problems to get a feeling of which modification of ξ could lead to better convergence.

[1]...once an initial guess has been found.

8.2 Improving the CUDA performance

Technically, the bad CUDA performance was not very surprising since we have constantly broken Rule Nr. 4: we used memory[2]. Of course we *had* to use memory, as we had to clone a lot of matrices to the device memory. But this is not the point. Since we made the algorithm for a general setting, we designed the kernels in a way that they can handle large matrices. While a matrix multiplication kernel was designed to run well on large data structures, the big size destroyed the efficiency of the CUDA–implementation of Cholesky's method and the forward– and backward–substitution. Same columns and single elements have to be accessed on the device memory very often during a single kernel execution.

The best case would be to load the whole data which is needed by a kernel into shared memory, do all calculations and pass it back to the device memory. But this requires quite small system dimensions such that $n \cdot s \times n \cdot s$ matrices and the right hand side of a linear system of equations completely fit into the shared memory (e.g. $n = 3, s = 2$). On the downside this will reduce the parallelization bandwidth of Cholesky's method. But this would be a small price for the gained runtime improvement.

8.3 Future development

One could easily modify the algorithm such that it does not simply calculate a single set for a given time, but the whole reachable tube on the given horizon. Each timestep could have its own grid and initial guess buffer. A single found feasible trajectory could mark a cell on each grid and provide its initial guess data for adjoining cells. After the reachable set of the last timestep has been found, the algorithm could continue processing the grid of the previous timestep, which has already a lot of marked reachable cells and initial guesses. With each timestep, the algorithm will finish much faster. Figure 8.1 illustrates this strategy.

It might also be useful to try some adaptive–grid strategies (e.g. like shown in [26] by W. Riedl) in Algorithm 6.2. In the current version, the queue manager pops a new initial guess and a corresponding grid point $\tilde{x}$ from the buffer. Within this step, the manager initializes the required memory and applies boxed constraints of the trajectory which restrict it to the

[2]see Section 1.2.6

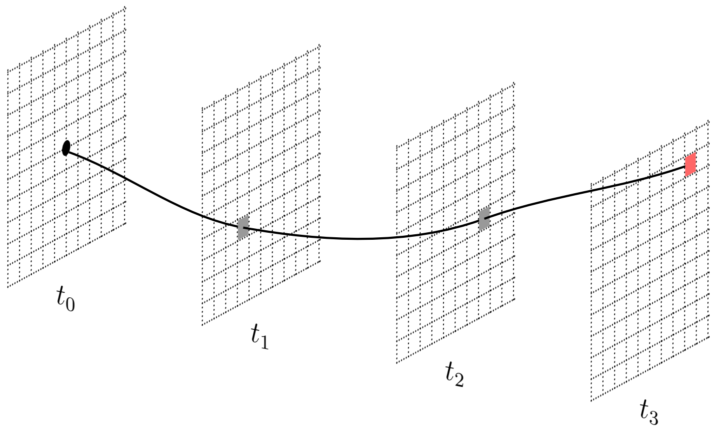


Figure 8.1: Computing a reachable tube: A single trajectory marks multiple grids generates multiple initial guesses

neighborhood of $\tilde{x}$ at a considered time step. Hence, the feasibility check routines (which are the core of the algorithm) actually don't care about the size of the neighborhood. As a consequence, adaptive grid strategies could be implemented easily without changing much of the program code.

List of Figures

List of Algorithms

Bibliography

[1] Amestoy, P. R., Davis, T. A., and Duff, I. S. An approximate minimum degree ordering algorithm. *SIAM J. Matrix Anal. Appl. 17*, 4 (Oct. 1996), 886–905.

[2] Baier, R., and Gerdts, M. A computational method for non-convex reachable sets using optimal control. In *Proceedings of the European Control Conference (ECC) 2009, Budapest (Hungary), August 23–26, 2009* (Budapest, 2009), European Union Control Association (EUCA), pp. 97–102.

[3] Baier, R., Gerdts, M., and Xausa, I. Approximation of reachable sets using optimal control algorithms. *Numerical Algebra, Control and Optimization 3*, 3 (2013), 519–548.

[4] Bodenschatz, M. Berechnung erreichbarer Mengen mit globalen Optimierungsverfahren. Diploma thesis, University of Bayreuth, 2014.

[5] Chudej, K., Michael, J., and Pannek, J. Modeling and Optimal Control of a Docking Maneuver with an Uncontrolled Satellite. In *Preprints MATHMOD 2012 Vienna - Full Paper Volume* (2012), I. Troch and F. Breitenecker, Eds., Argesim-Verlag, p. Report No. S38.

[6] Deuflhard, P., and Bornemann, F. *Scientific computing with ordinary differential equations*, vol. 42 of *Texts in Applied Mathematics.* Springer Verlag, New York, 2002.

[7] Duff, I., Erisman, A., and Reid, J. *Direct Methods for Sparse Matrices.* Oxford University Press, 1986.

[8] George, A. An automatic one-way dissection algorithm for irregular finite element problems. *SIAM Journal on Numerical Analysis 17*, 6 (1980), pp. 740–751.

[9] George, A., and Liu, J. An automatic nested dissection algorithm for irregular finite element problems. *SIAM Journal on Numerical Analysis 15*, 5 (1978), 1053–1069.

[10] Gerdts, M., and Xausa, I. Collision avoidance using reachable sets and parametric sensitivity analysis. In *System modeling and optimization. 25th IFIP TC 7 conference on system modeling and optimization, CSMO 2011, Berlin, Germany, September 12–16, 2011. Revised Selected Papers*, vol. 391 of *IFIP Adv. Inf. Commun. Technol.* Springer, Heidelberg–Dordrecht–London–New York, 2013, pp. 491–500.

[11] Grama, A., Gupta, A., Karypsis, G., and Kumar, V. *Introduction to Parallel Computing*, 2nd ed. The Benjamin/Cummings Publishing Company, Inc., 2003.

[12] Grüne, L., and Jahn, T. U. Computing reachable sets via barrier methods on SIMD architectures. In *European Congress on Computational Methods in Applied Sciences and Engineering (ECCOMAS 2012)* (Vienna, Austria, September 2012), no. 1518, Vienna University of Technology. CD-ROM, Paper No. 1518, 20 pages.

[13] Grüne, L., and Pannek, J. *Nonlinear Model Predictive Control.* Springer–Verlag London Limited, 2011.

[14] Harris, M. How to Optimize Data Transfers in CUDA C/C++, Parallel Forall (Blog). http://devblogs.nvidia.com/parallelforall/how-optimize-data-transfers-cuda-cc/, December 24 2012.

[15] Harris, M. How to Overlap Data Transfers in CUDA C/C++, Parallel Forall (Blog). http://devblogs.nvidia.com/parallelforall/how-overlap-data-transfers-cuda-cc/, December 13 2012.

[16] Harris, M. An Efficient Matrix Transpose in CUDA C/C++, Parallel Forall (Blog). http://devblogs.nvidia.com/parallelforall/efficient-matrix-transpose-cuda-cc/, February 18 2013.

[17] Hermann, M. *Numerik gewöhnlicher Differentialgleichungen: Anfangs- und Randwertprobleme [Numerics of Ordinary Differential Equations: Initial and boundary value problems]*. Oldenbourg Verlag, Munich, 2004.

[18] Jahn, T. Implementierung numerischer Algorithmen auf CUDA–Systemen. Diploma thesis, University of Bayreuth, 2010.

[19] Khaira, M. S., Miller, G. L., and Sheffler, T. J. Nested dissection: A survey and comparison of various nested dissection algorithms. Tech. rep., 1992.

[20] Michael, J. Optimale Steuerung eines Satelliten zur Weltraumschrottbeseitigung. Master thesis, University of Bayreuth, 2011.

[21] Michael, J., Chudej, K., Gerdts, M., and Pannek, J. Optimal Rendezvous Path Planning to an Uncontrolled Tumbling Target. In *Automatic Control in Aerospace* (2013), pp. 347–352.

[22] Mitchell, I. M., Bayen, A. M., and Tomlin, C. J. A time-dependent Hamilton-Jacobi formulation of reachable sets for continuous dynamic games. *IEEE Trans. Automat. Contr. 50*, 7 (2005), 947–957.

[23] Mitchell, I. M., and Tomlin, C. J. Overapproximating reachable sets by Hamilton-Jacobi projections. *J. Sci. Comput. 19*, 1-3 (Dec. 2003), 323–346.

[24] Nocedal, J., and Wright, S. J. *Numerical Optimization*, 2nd ed. Springer Verlag, 2006.

[25] NVIDIA. CUDA C Programming Guide, August 2014. v6.5 – PDF file – http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf.

[26] Riedl, W. Optimization-based subdivision algorithm for reachable sets. Submitted to *Proceedings in Applied Mathematics and Mechanics*.

[27] Scott, J. A., and Hu, Y. Experiences of sparse direct symmetric solvers. *ACM Trans. Math. Softw. 33*, 3 (Aug. 2007).

[28] Stöcklein, M. Eine Variante des Cholesky-Verfahrens für dünn besetzte Matrizen. Bachelor thesis, University of Bayreuth, 2013.

[29] Xausa, I., Baier, R., Gerdts, M., Gonter, M., and Wegwerth, C. Avoidance trajectories for driver assistance systems via solvers for optimal control problems. In *Proceedings on the 20th International Symposium on Mathematical Theory of Networks and Systems (MTNS 2012), July 9–13, 2012, Melbourne, Australia* (Melbourne, Australia, 2012), Think Business Events. CD-ROM, Paper No. 294, full paper.