Het versnellen van CONTACT-programmatuur met behulp van Fouriertransformaties.

Engelse titel: Speeding up the CONTACT package by means of the Fourier transform.

Verslag ten behoeve van het Delft Institute for Applied Mathematics als onderdeel ter verkrijging van de graad van

BACHELOR OF SCIENCE
in
TECHNISCHE WISKUNDE

doors

PIETER LOOF

Delft, Nederland
Juni 2011

Copyright © 2011 door Pieter Loof. Alle rechten voorbehouden.
BSc verslag TECHNISCHE WISKUNDE

“Het versnellen van CONTACT-programmatuur met behulp van Fouriertransformaties.”
(Engelse titel: “Speeding up the CONTACT package by means of the Fourier transform.”)

PIETER LOOF

Technische Universiteit Delft

Begeleiders
Prof.dr.ir. C. Vuik Dr.ir. E.A.H. Vollebregt

Overige commissieleden
Dr. J.G. Spandaw Dr.ir. H.X. Lin

Juni, 2011 Delft
Contents

1 Introduction 7

2 Problem description 9
   2.1 The whole problem ........................................... 10
   2.2 Notation ...................................................... 11

3 About Fourier transforms and convolutions 13
   3.1 The 1D transform ............................................ 13
   3.2 The 2D transform ............................................ 15
   3.3 The inverse Fourier transforms ............................... 16
   3.4 When to use 1D and 2D transforms? .......................... 16
   3.5 Fast Fourier Transform ....................................... 16
   3.6 Convolutions ................................................ 17

4 A 2D convolution in the Fourier domain 19
   4.1 The convolution in the Fourier domain ....................... 19
   4.2 Is the convolution in the Fourier domain valid and exact? 20
   4.3 The runtime of the improved 2D convolution ................. 21
   4.4 The runtime compared to the standard 2D convolution .... 21
   4.5 Summary of the improved 2D convolution ................... 22

5 Barrowes’ method 23
   5.1 The properties of a Toeplitz matrix .......................... 23
   5.2 From a 2D problem to a 1D convolution ...................... 24
      5.2.1 A Toeplitz matrix is determined by a few specific values 24
      5.2.2 The pattern of the rows .................................. 25
      5.2.3 The convolution revealed .................................. 25
   5.3 The relation with the improved 2D convolution ............. 26
      5.3.1 The length of b and p’ ................................. 26
      5.3.2 The convolution and the final result ..................... 27
      5.3.3 The runtime compared ..................................... 27
   5.4 Summary of Barrowes’ method ................................ 28

6 Runtime and memory usage of the treated methods 29
   6.1 Test results .................................................... 29
   6.2 About the runtime ............................................. 30
   6.3 About the memory usage ....................................... 30
   6.4 Choosing ‘good’ grid sizes ................................... 31
7 Solving the whole problem

7.1 Do not merge independent FFT’s .................................................. 35
7.2 A special property of the influence grids ........................................ 35
7.3 Handling the five subproblems efficiently ...................................... 36
  7.3.1 The five subproblems calculated totally independently ............... 36
  7.3.2 Optimized independent calculation ......................................... 36
  7.3.3 Barrowes’ method: four subproblems combined, version 1 .......... 38
  7.3.4 Barrowes’ method: four subproblems combined, version 2 .......... 39
7.4 The best way of solving a 3D problem ............................................. 40
7.5 An iterative process for the whole problem .................................... 41

8 Further optimization of the subproblems ............................................ 43

8.1 Properties of the Fourier transform for real valued input .................. 43
8.2 Properties of the Fourier transform for symmetric input .................... 44
  8.2.1 A 1D example ............................................................................. 44
  8.2.2 A 2D example ............................................................................. 45
  8.2.3 A different 2D example .............................................................. 46
8.3 Conclusion .................
Chapter 1

Introduction

A program called CONTACT is made for solving the problems that arise when two real world objects make physical contact. Questions might arise, like: what is the size and shape of the contact area, or how are the objects deformed due to the pressure between them? For example, one might think of the problem of the amount of abrasion on train wheels and the railway. This kind of questions and their numerical approximations is what the Dutch company VORtech, situated in Delft, deals with. VORtech creates mathematical software for governments departments and commercial clients. Their software program CONTACT, which is used for several computations that arise when dealing with mechanical contact problem, originates from professor J.J. Kalker at Delft University of Technology. This project is done in cooperation with VORtech and I would like to thank them for making this project available and for all their support during this project.

Problems like these might look easy to understand, but extensive calculations are needed to solve such problems. CONTACT makes a discretization of the problem area and then a lot of mathematical operations like matrix vector products and inner products have to be done. Remark that the exact shape of the area where objects make contact is not known in general. An iterative process is needed in order to update the contact area between each step. Ultimately, the contact area should converge to a certain shape. In every iterative step a deformation due to contact pressure might have an effect on both whole objects and thus on the whole contact area of the objects. So in general a number of iteration steps have to be performed and within each step, each grid point needs the information of the whole contact area in order to be updated. One can easily conclude that the number of calculations grows very quickly when using more grid points. At the moment these calculations are so expensive that a very rough discretization is used in order to get a result in a reasonable time span.

The methods used in CONTACT are not yet optimal. The methods themselves, i.e. convolutions of certain matrices, can be further optimized using faster algorithms like the Fast Fourier Transform. Next to that a part of the calculations can be done in parallel, which can be done by using the graphics processing unit, the GPU, instead of only the central processing unit, the CPU. Not all of these topics will be discussed fully now, for a lot of work has already been done by others. For those interested in the use of the GPU for faster calculations I refer to the work of Michiel de Reus, see [1]. In this article we will take a closer look at the possible optimizations for the convolutions in CONTACT.
Chapter 2

Problem description

Consider the area where two real world objects make contact. When these two objects are stiff, the contact area will be small and approximately flat. This contact area can be discretized in an area of $m_x \times m_y$ tiles. The transformation of the objects depends for each grid point on the whole contact area. The strength of influence can be put into a 2D grid, which will be called the influence grid $B$. In general this influence grid will be about twice as long and wide as the input grid $P$. An example of the size of $B$ for some size of grid $P$ can be seen in Figure 2.1.

![Figure 2.1: An example of the relative size of B and P. P fits in B when P overlaps with the light gray grid point.](image)

For each grid point the influence grid is laid with its center, marked with light gray in Figure 2.1, on that grid point and then the elements of the influence grid and the input grid have to be multiplied elementwise. In general $B$ has to be large enough so that all elements of $P$ overlap with elements of $B$ when one grid point of $P$ overlaps with the center point of $B$. Thus per grid point, $m_x m_y$ multiplications and additions have to be performed, so for the whole grid it takes about $2m_x^2m_y^2$ operations. This can be written as a 2D convolution by flipping one of the two grids around the horizontal and vertical axis. More information about convolutions can be found in Section 3.6.

This 2D convolution can also be written as a matrix vector product. The $m_x \times m_y$ grid can be put into a column vector of length $m_x m_y$, called $p$. When laying the influence grid over the input grid, a sub block of the influence grid overlaps with the input grid. The values of such a sub block are put into one row of the new $m_x m_y \times m_x m_y$ matrix, which will be named $A$. So all the rows of $A$ are formed from the sub blocks of the influence grid. The number of operations in the matrix vector product is still about as large as the number of calculations for the 2D convolution. We only write the convolution in a different form. When converting a 2D
CHAPTER 2. PROBLEM DESCRIPTION

grid to a matrix $A$ in this way, $A$ will be a 2-level block-Toeplitz matrix. If one would work with a 3D grid, then $A$ would be a 3-level block-Toeplitz matrix. In Chapter 5 will be explained how to use the properties of a block-Toeplitz matrix to calculate the product of $A$ and $p$ faster than a standard matrix vector product. But finally we will discover that we do not need this method for optimizing the convolutions of CONTACT. Fourier transforms play an important role when calculating a 2D convolution or the product of $A$ and $p$. Therefore, the usage of Fourier transforms and convolutions will be explained first.

2.1 The whole problem

Because we are working in three dimensions, each grid point of grid $P$ is actually a vector describing the displacement in 3D. So each grid point contains three values for the displacement, one per dimension. The total problem for all the dimensions together can be written as one large matrix vector product. Figure 2.2 shows how the total problem looks when written as a matrix vector product. In this figure, $A_{zz}$, $A_{xx}$, $A_{xy}$, $A_{yx}$ and $A_{yy}$ are block-Toeplitz matrices on their own.

![Figure 2.2: The total matrix vector product for the 3D problem. The sub blocks of the matrix are each block-Toeplitz.](image)

The z-direction is the normal direction and the other two directions are the directions within the flat contact area, also referred to as the tangential direction. Figure 2.2 already shows that the displacement in the normal direction is independent from the displacement in the other two directions. Apparently the other two directions are not independent of each other. These dependencies are also described in [1, Section 2.4, p. 16], where the displacement of grid points is also described with formulas.

This large problem has to be calculated multiple times. The resulting grid is the input for the next calculation. So the total problem is an iteration where a problem as in Figure 2.2 has to be calculated in each step. This iteration continues until the resulting grid converges to a stable solution.

The methods that will be discussed in this article mostly describe how to solve one 2D convolution of two grids or the matrix vector product that can be derived from it. But the total problem is an iteration with in each step five of these 2D convolutions or matrix vector products, namely $A_{zz}p_z$, $A_{xx}p_x$, $A_{yx}p_x$, $A_{xy}p_y$ and $A_{yy}p_y$. Later on we will discuss whether these five similar problems should be calculated independent from each other or not.
2.2 NOTATION

The following notations will be used throughout this article. In general, matrices will be denoted with a capital and vectors with lower-case letters. For matrices or formulas representing Fourier transforms and series the $F$ will be used. The input grid is called $P$ and its size is $m_x \times m_y$. The influence grid will be called $B$. The block-Toeplitz matrix that can be constructed from $B$ will be called $A$ and it has a size of $n \times n$ with $n = m_x m_y$. The vector containing all the values from $P$ will be called $p$. The input grid or vector filled up with extra zeros will be referred to as $P'$ respectively $p'$. And finally, the resulting matrix or vector will be called $U$ respectively $u$. The number of calculations for some methods, which is related to their total runtime, will be denoted with a $r$ with a subscript indicating the method and size.
Chapter 3

About Fourier transforms and convolutions

This chapter tries to give a clear look on using Fourier transforms for discrete matrices and vectors. It is not meant as a laymen introduction on the topic of Fourier transforms, but more on the differences of 1D and 2D transforms and using them. Next to that, a short reminder on the topic of convolutions is given.

The Fourier domain can be seen as a frequency domain. For discrete problems, the Fourier transform is also known as a DFT, a discrete Fourier transform. In general, the Fourier series of a function is not equal to the function itself. It is an integral on an infinite interval and its value might grow to infinity. When dealing with discrete input, the transformation to the Fourier domain is a finite sum over finite values, so its value is also finite. Next to that the Fourier series of a vector of finite length is periodic with its length. In other literature on the subject of Fourier transforms can be found proven that if the Fourier series of some function \( f \) is periodic and has a non-infinite, well defined value, then this Fourier series converges uniformly to \( f \) on the whole domain of \( f \). The proof for the continuous case can be found in [4, Corrolary 2.3, p. 41]. When \( f \) is a continuous function on a real valued interval, it has infinitely many points so the Fourier series only converges to \( f \) with infinitely many terms. But the problems this article deals with contain discrete vectors, so then the Fourier series only needs as many terms as the length of the vector.

3.1 The 1D transform

The Fourier transform \( F(x) \) of a vector \( v[j] \) with length \( n \) is defined\(^1\) as:

\[
F(x) = \sum_{u=0}^{n-1} F[u] e^{i \frac{2\pi}{n} ux}
\]  

(3.1)

Where \( i \) is the imaginary unit with property \( i^2 = -1 \) and the terms \( F[u] \) are defined as:

\[
F[u] = \sum_{j=0}^{n-1} v[j] e^{-i \frac{2\pi}{n} uj} \quad \text{for } u = 0, 1, \ldots, n - 1
\]  

(3.2)

\(^1\)In Formulas (3.1) and (3.2), \( u \) is an integer for the Fourier coefficients and \( j \) is an integer for the spatial domain of the input. But variable \( x \) is a real number.
The Fourier series of a vector with length \( n \) results in a sum of sinusoidal terms, because terms of the form \( e^{iax} \) for real valued \( a \) can be decomposed into \( \cos(ax) + i\sin(ax) \), so a sine and cosine part. This also makes it comprehensible that the Fourier series will be periodic because a finite sum of sine and cosine functions is periodic. Remark that this Fourier series is a continuous function. In this method, the values of \( v[j] \) are arranged on the line of natural numbers and the Fourier series gives a continuous function that goes exactly through all these points. It actually is the same principle as that one can find a continuous, smooth function going through \( n \) points. It actually is the same principle as that one can find a continuous, smooth function independent of \( n \) of lower order cannot be an upper bound for this function. If one defines these \( n \) numbers and the Fourier series gives a continuous function that goes exactly through all these points.

Formula (3.2) are the \( V \) a 1D Fourier transform on each column of \( u \) 1D Fourier transforms. Therefore we do not really need the continuous Fourier series of the complex vector \( F \), meaning that the total time of the calculation can be bounded from above by a function of the form \( cn^2 \) with \( c \) a constant value independent of \( n \). By stating that a function is of a certain order we mean that an other function of lower order cannot be an upper bound for this function. If one defines \( \omega = e^{-\frac{2\pi i}{n}} \), then the Fourier transform of a vector can be written as a matrix vector product. This matrix will be called \( F_n \) and it is shown below.

\[
F_n = \begin{bmatrix}
1 & 1 & 1 & \ldots & 1 \\
1 & \omega & \omega^2 & \ldots & \omega^{n-1} \\
1 & \omega^2 & \omega^4 & \ldots & \omega^{2(n-1)} \\
1 & \omega^3 & \omega^6 & \ldots & \omega^{3(n-1)} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
1 & \omega^{n-1} & \omega^{2(n-1)} & \ldots & \omega^{(n-1)(n-1)}
\end{bmatrix}
\]

So this matrix \( F_n \) represents a 1D Fourier transform. The first row of \( F_n \) multiplied with vector \( v \) gives the first coefficient of the Fourier series, the second row produces the second coefficient etc. So when \( F_n \) is multiplied with a vector, the output vector gives the values of \( F[u] \).

Now, what happens when \( F_n \) is multiplied with a 2D matrix, let us say \( V[j,k] \)? Then the product of \( F_n \) with \( V \) is of course a matrix matrix product. Note that for a matrix matrix product in general, the rows of the first matrix are multiplied with the columns of the second matrix. The first column of the resulting output is computed by multiplying all the rows of the first matrix with the first column of the second matrix, so it does not depend on the values in other columns of the second matrix. So a column in the output only depends on the corresponding column in the second matrix, thus the first matrix works on all the columns of the second matrix independently. Thus the product of \( F_n \) with \( V[j,k] \), being a matrix matrix product, produces a 1D Fourier transform on each column of \( V \) independently. This results in a number of 1D transforms.
3.2 The 2D transform

The 2D Fourier transform also exists. For a discrete matrix \( V[j, k] \) of size \( n \times m \) it is defined\(^2\) as:

\[
F(x, y) = \sum_{u=0}^{n-1} \sum_{w=0}^{m-1} F[u, w] e^{i2\pi(\frac{uy}{n} + \frac{wx}{m})} \quad (3.3)
\]

Where the terms \( F[u, w] \) are defined as:

\[
F[u, w] = \sum_{j=0}^{n-1} \sum_{k=0}^{m-1} V[j, k] e^{-i2\pi(\frac{uj}{n} + \frac{wk}{m})} \quad \text{for } u = 0, 1, \ldots, n - 1 \n\]

\[
\text{for } w = 0, 1, \ldots, m - 1 \quad (3.4)
\]

A 2D transform of a matrix produces a 2D Fourier series, depending on two spatial variables. It contains a summation of sine and cosine functions of \( j \), multiplied with sine and cosine functions of \( k \), resulting in products of sine and cosine functions of \( j \) and \( k \). \( F(x, y) \) is a continuous function, but we are only interested in the coefficients \( F[u, w] \). But this transform needs a lot of calculations. The 2D convolution of a matrix of size \( n \times n \) is \( O(n^4) \) because for each of the \( n^2 \) output values a summation over \( n^2 \) points has to be computed.

The 2D Fourier transform can be written as two 1D transforms. Here is shown how the Fourier coefficients can be calculated with two successive 1D operations for a matrix \( V[j, k] \). The matrix of Fourier coefficients \( F[u, w] \) for matrix \( V \) is written as \( F_{uw}(V) \) for the sake of a clear notation.

\[
F[u, w] = F_{uw}(V) = \sum_{j=0}^{n-1} \sum_{k=0}^{m-1} V[j, k] e^{-i2\pi(\frac{uj}{n} + \frac{wk}{m})} \quad (3.5)
\]

\[
= \sum_{j=0}^{n-1} \left( \sum_{k=0}^{m-1} V[j, k] e^{-i2\pi \frac{wk}{m}} \right) e^{-i2\pi \frac{uj}{n}} \quad (3.6)
\]

\[
= F_u(F_w(V)) \quad (3.7)
\]

\[
= F_w(F_u(V)) \quad (3.8)
\]

So the 2D transform can be calculated by first applying a 1D transform in one direction followed by a 1D transform in the other direction. The order in which to do these two transforms does not matter. The 2D transform of a matrix can also be expressed as a product of matrices. Define \( \omega_n = e^{-i \frac{2\pi}{n}} \) and \( \omega_m = e^{-i \frac{2\pi}{m}} \). Because one transform works in horizontal direction and the other in vertical direction, the transform of an input matrix \( V \) with \( n \) rows and \( m \) columns to the Fourier domain can be written as:

\[
\begin{bmatrix}
1 & 1 & \ldots & 1 \\
1 & \omega_n & \ldots & \omega_n^{n-1} \\
1 & \omega_n^2 & \ldots & \omega_n^{2(n-1)} \\
\vdots & \vdots & \ddots & \vdots \\
1 & \omega_n^{n-1} & \ldots & \omega_n^{(n-1)(n-1)}
\end{bmatrix}
\begin{bmatrix}
1 & 1 & \ldots & 1 \\
1 & \omega_m & \ldots & \omega_m^{m-1} \\
1 & \omega_m^2 & \ldots & \omega_m^{2(m-1)} \\
\vdots & \vdots & \ddots & \vdots \\
1 & \omega_m^{m-1} & \ldots & \omega_m^{(m-1)(m-1)}
\end{bmatrix}
\]

\[
\text{where } n \times n \text{ and } m \times m
\]

\(^2\)In Formulas (3.3) and (3.4), \( u \) and \( w \) are integers for the Fourier coefficients and \( j \) and \( k \) are the integers for the spatial domain of the input. But variables \( x \) and \( y \) are real numbers.
CHAPTER 3. ABOUT FOURIER TRANSFORMS AND CONVOLUTIONS

The number of calculations with two successive 1D transforms is much lower than computing the full 2D transformation when dealing with large matrices. But it is still twice as much work as applying only one 1D Fourier transform to a matrix. So why would anyone use a 2D Fourier transform for a matrix when a 1D transform is always faster? That will be explained in Section 3.4.

3.3 The inverse Fourier transforms

The inverse Fourier transforms almost look like the transforms themselves, there are only two small differences. The first one is that a minus sign has to be put before the imaginary unit $i$ in Formula (3.2) for the Fourier coefficients. The other difference is a factor $\frac{1}{n}$ where $n$ is the size of the vector that is being transformed. So for the 1D case one can find $v[j]$ back out of $F[u]$ with the following formula:

$$v[j] = \frac{1}{n} \sum_{u=0}^{n-1} F[u] e^{i \frac{2\pi}{n} uj} \quad \text{for} \quad u = 0, 1, \ldots, n-1 \quad (3.9)$$

In the same way, the formula for the 2D inverse transform can be found out of the normal 2D transform by putting a minus sign before $i$ and putting a factor $\frac{1}{nm}$ before the sum, for a matrix size of $n \times m$.

3.4 When to use 1D and 2D transforms?

A 2D transform is useful when dealing with 2D operations on grids. As an example, think of an image, of course 2D, that has to be smoothed. The smoothing can be done by a convolution of a 2D filter and the image, meaning that the center of the 2D filter should be laid on every pixel of the image to produce every pixel of the new image. This can be done faster in the Fourier domain. Because the smoothing has to be done in two directions, the image should be transformed to a 2D Fourier image by applying a 2D Fourier transform. The filter is also transformed and then the two resulting grids can be multiplied elementwise. The resulting value of a certain pixel depends on the values of a 2D field of pixels of the input image. Thus for 2D operations like this, a 2D Fourier transform is needed. Or the problem can be transferred to a larger matrix vector product, which in general still costs about the same amount of calculations, as explained in Chapter 2. When dealing with products of a matrix with a matrix or vector, the value of a certain output position only depends on one row of the first matrix and one column of the matrix or vector it is multiplied with. So a product of matrices with given sizes can be handled by 1D Fourier transforms which is faster than using 2D transforms.

3.5 Fast Fourier Transform

FFT stands for a fast Fourier transform. This is not a special way of transforming. It is just a method to speed up the Fourier transforms. The idea is that a transform over $n$ elements can be split up into two transforms over $\frac{n}{2}$ elements. Instead of approximately $n^2$ multiplications and addition, only $2(\frac{n}{2})^2 = \frac{n^2}{2}$ calculations have to be done. But these two parts can each be split up again. This splitting can be continued, making the needed amount of calculations smaller. We will not go in detail why this method is valid, it can be found in [4, Section 7.1.3, p. 224]. By using the FFT, a Fourier transform of a vector of length $n$ is only $O(n \log n)$ instead of
3.6. CONVOLUTIONS

This method works best when $n$ is a power of 2, because then the successive splitting in two parts is nice and neat. For example, a vector of length $n = 4$ can be split into two times successively, resulting in four single point transformations. For numbers that are not a power of 2, more work has to be done. When the number is a product of two integers, say $n = n_1n_2$, then a split up in FFT’s of length $n_1$ and $n_2$ can be done plus some extra work. For example, $n = 6$ can be split up in transforms of length 2 and 3. When the length is not a power of 2, the FFT will still be much faster than the standard Fourier transform, but a power of 2 is preferred. Even for prime lengths $n$ there are FFT’s of $O(n \log n)$, but they require a bit more work so they might be a bit slower than for non-prime values. The main point to note here is that a Fourier transform of a vector of length $n$ can be achieved in $O(n \log n)$ instead of the expected $O(n^2)$.

The number of calculations needed for a FFT can be described more precisely. According to [6] the number of operations of a few different FFT variants lies between 34000 and 41000 for a vector of length 1024. Now, $1024 \log 1024 = 10240$ which is almost a factor 4 lower, which means that a good guess for the number of operations for a FFT of length $n$ is $4n \log n$. The calculation of the 1D Fourier transform of a matrix is done simply by calculating the 1D Fourier transform for each column of the matrix, as stated in Section 3.1. For a matrix of size $n \times m$ this costs about $m \cdot (4n \log n) = 4mn \log n$ operations.

3.6 Convolutions

In general, a discrete convolution of vectors $f$ and $g$ is defined as

$$(f \ast g)[k] = \sum_{m=-\infty}^{\infty} f[m]g[k-m] \quad (3.10)$$

The formula shows that each next element of $f$ is multiplied with the previous element of $g$, because when $m$ is incremented with 1, $k - m$ goes down by 1. The 2D convolution is just a simple expansion of the 1D convolution. For grids $F$ and $G$ it is defined as:

$$(F \ast G)[k,l] = \sum_{m=-\infty}^{\infty} \sum_{n=-\infty}^{\infty} F[m,n]G[k-m,l-n] \quad (3.11)$$

Sometimes one would like to multiply two vectors or two grids elementwise. This can be rewritten to a convolution. When multiplying two vectors elementwise, just flip one of the vectors. Then the first element of the first vector should be multiplied with the last element of the other vector, the second element of the first vector should be multiplied with the one but last element of the other vector etc. So then it is just a convolution. The same holds for multiplying two grids elementwise. One simply flips one grid around the horizontal and vertical axis and then apply a 2D convolution.

Convolutions can be computed very fast within the Fourier domain. The only restriction is that both vectors or grids should be of the same size. When two vectors or grids are not of the same size, one can simply enlarge the smallest one and fill up the new positions with zeros.
Chapter 4

A 2D convolution in the Fourier domain

As stated in the first part of Chapter 2, a 2D convolution of the influence grid with the input grid has to be calculated. Calculating the 2D convolution directly for a \( m_x \times m_y \) input grid and an influence grid of size \((2m_x-1)\times(2m_y-1)\) will cost almost \(2m_x^2m_y^2\) operations. This convolution can be calculated faster within the Fourier domain. To achieve this, 2D Fourier transforms of the grids have to be computed. The transformed grids must be multiplied elementwise, which comes with a condition that the grids should be of the same size. This means in our case that the input grid has to be enlarged to the same size as the influence grid, by adding zeros. After the elementwise multiplication is done, the result has to be transformed back to the normal domain. From Chapter 3 we learned that a 2D Fourier transform can be calculated with two successive 1D transforms.

Now, an example problem will be shown and worked out to illustrate the method. Consider the influence grid \( B \) and the input grid \( P \) below:

\[
\begin{bmatrix}
6 & 0 & 2 & 2.5 & -1 \\
5 & 3 & 1 & 0 & 7 \\
8 & 0 & 1.3 & 0 & 9 \\
\end{bmatrix}
\]

influence grid \( B \)

\[
\begin{bmatrix}
1 & 2 & 3 \\
4 & 5 & 6 \\
\end{bmatrix}
\]

\( P \)

4.1 The convolution in the Fourier domain

The input grid should be enlarged to the same size as \( B \) by adding zeros on the right side and below. The resulting grid will be called \( P' \).

\[
\begin{bmatrix}
1 & 2 & 3 & 0 & 0 \\
4 & 5 & 6 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
\end{bmatrix}
\]

\( P' \)

Both \( B \) and \( P' \) will be transformed to the Fourier domain with a 2D Fourier transform. This means that \( B \) and \( P' \) are both multiplied with two matrices, as explained in Section 3.2. This results in the matrices \( B' \) and \( P'' \). The two matrices are shown below together with the result of the elementwise multiplication. The values in these matrices are rounded.
CHAPTER 4. A 2D CONVOLUTION IN THE FOURIER DOMAIN

B

\[
\begin{bmatrix}
43.8 & 19.1 + 10.4i & 6.5 + 8.8i & 6.5 - 8.8i & 19.1 - 10.4i \\
-7.7 + 2.0i & -10.4 - 4.0i & 6.2 - 2.6i & 11.8 + 9.4i & -2.5 + 8.3i \\
-7.7 - 2.0i & -2.5 - 8.3i & 11.8 - 9.4i & 6.2 + 2.6i & -10.4 + 4.0i \\
\end{bmatrix}
\]

Then, the resulting matrix \( U' \) can be transformed back to the normal domain with an inverse 2D Fourier transform, which results in the matrix \( U'' \). This matrix \( U'' \) contains the resulting values. But the matrix is too large, it also contains values we are not interested in. This is due to the fact that zeros had to be added to \( P \) in order get this method working. The resulting matrix \( U \) we are looking for can be found in the \( 2 \times 3 \) sub block (the size of \( P \)) at the bottom right of \( U'' \). \( U' \) and the obtained result \( U \) are shown below:

\[
\begin{bmatrix}
88.5 & 103 & 73.2 & 13 & 53.8 \\
53 & 58 & 66 & 31 & 30.5 \\
81 & 122 & 74.3 & 25.6 & 46.9 \\
\end{bmatrix}
\]

One can easily verify that the obtained result \( U \) is correct, by calculating the 2D convolution directly, which should be doable for such a small input grid. In order to do this, one flips \( P \) around the vertical and horizontal axis and then lay this flipped grid onto \( B \) in all possible ways for which all its values overlap with \( B \). For each possible overlapping, there are six possibilities in our example, one calculates the elementwise product of the values lying on each other and add them together. Figure 4.1 shows the calculation for the bottom right value of \( U \) when computing the 2D convolution directly.

\[
\begin{bmatrix}
6 & 0 & 2 & 2.5 & -1 \\
5 & 3 & 1 & 0 & 7 \\
8 & 0 & 1.3 & 0 & 9 \\
\end{bmatrix} \cdot \begin{bmatrix}
6 & 5 & 4 \\
3 & 2 & 1 \\
\end{bmatrix}
\]

\[
6 \cdot 1 + 5 \cdot 0 + 4 \cdot 7 + 3 \cdot 1.3 + 2 \cdot 0 + 1 \cdot 9 = 46.9
\]

Figure 4.1: The calculation of one resulting value of the 2D convolution of \( B \) and \( P \). It is an inner product of the gray area of \( B \) with the flipped grid \( P \).

4.2 Is the convolution in the Fourier domain valid and exact?

\( B \) and \( P' \) are discrete grids with a finite size. As stated in Chapter 3 this means that the Fourier series of these grids are unique and exact. When applying Fourier transforms on these grids, the
4.3. THE RUNTIME OF THE IMPROVED 2D CONVOLUTION

Fourier series contain complex numbers, but this is nothing to be afraid of. When performing multiplications or additions on real valued grids or vectors, the results will also be real valued. Because the Fourier series is unique and exact, there is no risk of ending up with complex values or wrong answers at all.

In the Fourier domain the convolution is done by simply multiplying elementwise, which is only $O(m_x m_y)$ when the grids each contain about $4m_x m_y$ elements. The proof that a convolution is just an elementwise product in the Fourier domain will not be given now, for it can be found in other literature, for example in [4, Proposition 3.1.vi, p. 45]. After the elementwise product is taken, the results only have to be transformed back into the normal domain. There is no loss of information or exactness in this process. It is like integrating a function over a curved domain, where one might transform the problem to a system with polar coordinates, solving it and finally transforming back to the normal system without loss of information or correctness.

4.3 The runtime of the improved 2D convolution

Now that we saw a specific example, the runtime of this method will be calculated. Constructing $P'$ out of $P$ only costs about $m_x m_y$ operations, when the values of $P$ are put into an empty vector of zeros of the appropriate size. This kind of preparations do not cost much time and will be of no importance when dealing with large problems. The main operations of this method are the two Fourier transforms and the inverse Fourier transform. As stated in Section 3.5, the number of operations for a Fourier transform, using FFT, on a grid of size $m_x \times m_y$ is about $4m_x m_y \log m_x$.

In our case, all three Fourier transforms are done on grids with a size of about $2m_x \times 2m_y$. One 2D Fourier transform consists of two successive 1D transforms, one with a matrix of size $2m_x \times 2m_x$ and one with a size of $2m_y \times 2m_y$. So these two transformation costs about $4 \cdot 2m_x 2m_y \log 2m_x + 4 \cdot 2m_x 2m_y \log 2m_y = 16m_x m_y (\log 2m_x + \log 2m_y) = 16m_x m_y \log 4m_x m_y$. So three Fourier transforms costs about $48m_x m_y \log 4m_x m_y$ in total. The elementwise product of $B'$ and $P'$ in the Fourier domain costs $2m_x \cdot 2m_y = 4m_x m_y$ multiplications. The total amount of work, which is of course related to the total runtime, will be $r_{\text{conv}}(n)$ which is $O(m_x m_y \log m_x m_y) = O(n \log n)$ with $n = m_x m_y$. A good approximation of the total amount of work of the algorithm is given by the formula:

$$r_{\text{conv}}(n) = 4n + 48n \log 4n \quad \text{with } n = m_x m_y \quad (4.1)$$

4.4 The runtime compared to the standard 2D convolution

Now we would like to know how fast the described method is compared to computing the 2D convolution directly. For the direct convolution, the number of multiplications and additions is both about $m_x m_y$, so $2m_x^2 m_y^2$ in total. Remark that this formula and the formula of the method of this chapter can both be rewritten with $n = m_x m_y$. The total amount of work for the direct 2D convolution can be written as:

$$r_{\text{direct}}(n) = 2n^2 \quad \text{with } n = m_x m_y \quad (4.2)$$

Then, one can calculate for which value of $n$ this formula and Formula (4.1) give an equal value. $4n + 48n \log 4n = 2n^2$ leads to $n \approx 240$. Note that the formula $2n^2$ increases faster than the other formula when $n$ increases. This means that for a grid with more than 240 points, the method of this chapter is faster than the direct 2D convolution.

It might be the case that some more preparations are needed when implementing the given method, but such extra work will be just $O(n)$ or lower in this case. For example, the extra
work might change the term $4n$ of the given formula to something like $10n$ or maybe $20n$, but even then the intersection point will be around $n = 250$, which is hardly different from the value 240 we just found. The term $48n \log 4n$ starts to dominate quite quickly when $n$ increases, so the bit of extra work does not really matter. It is quite save to conclude that the 2D convolution calculated in the Fourier domain as described, will be faster than the direct calculation of the 2D convolution for input grids with more than 250 points.

4.5 Summary of the improved 2D convolution

Given a 2D grid of size $m_x \times m_y$ and an influence matrix, this method computes the 2D convolution for which it makes use of Fourier transforms. This method works for an arbitrary influence matrix and grid. In our problems the influence matrix is larger than the grid, to be precise, its size is $(2m_x-1) \times (2m_y-1)$. A direct convolution would be $O(m_x^2 m_y^2)$, but the method of this chapter can perform the convolution in $O(m_x m_y \log m_x m_y)$. The amount of memory space needed is about $12m_x m_y$, while the direct convolution needs $9m_x m_y$, so this is not a big difference. Here are the steps that are to be performed when using the described method:

1. Make sure the input grid $P$ has the same size as the influence grid $B$, by adding positions filled with zeros. The resized grid is called $P'$.
2. Calculate the 2D Fourier transform of $P'$ and $B$. Both 2D transforms are to be calculated using two successive 1D FFT’s. This step results in two matrices in the Fourier domain: $B_f$ and $P'_f$.
3. Perform the elementwise product of $B_f$ and $P'_f$, which produces one new matrix $U_f$.
4. Calculate the inverse Fourier transform of $U_f$ with a FFT, which results in matrix $U'$.
5. The values we are interested in can be found in a sub block of the size of $P$ within $U'$.

The position of the sub block where the final values can be found depends on the position of the added zeros in the first step. If the extra zeros were added on the right side and bottom of the input grid, then the new values of the grid can be found in the bottom right sub block of $U'$. 
Chapter 5

Barrowes’ method

Consider the situation of taking the convolution of an input grid and an influence grid. As described in Chapter 2 this can be written as a matrix vector product. The matrix has special properties: it is a multilevel block-Toeplitz matrix. Because of that the matrix vector product can be rewritten to a convolution of two 1D vectors. For a good description of Toeplitz matrices and how Barrowes’ method works, I refer to [3]. A good explanation of this method accompanied by an example, can be found in [2]. Barrowes’ method will be worked out for the example we introduced in Chapter 4. The values of all possible $2 \times 3$ sub blocks of influence grid $B$ are put in rows, resulting in matrix $A$. The values of grid $P$ are put in one column vector, which will be called $p$ in this section. Remark that the size of $A$ is $m_xm_y \times m_xm_y$, so in this example the size is $(2 \cdot 3) \times (2 \cdot 3) = n \times n$ for $n = m_xm_y$. The product of $A$ and $p$ is another representation of the 2D convolution. The product is shown below.

$$
\begin{bmatrix}
u_0 \\
 u_1 \\
 u_2 \\
 u_3 \\
 u_4 \\
 u_5 \\
\end{bmatrix} =
\begin{bmatrix}
1 & 0 & 7 & 1.3 & 0 & 9 \\
3 & 1 & 0 & 0 & 1.3 & 0 \\
5 & 3 & 1 & 8 & 0 & 1.3 \\
2 & 2.5 & -1 & 1 & 0 & 7 \\
0 & 2 & 2.5 & 3 & 1 & 0 \\
6 & 0 & 2 & 5 & 3 & 1 \\
\end{bmatrix}
\begin{bmatrix}
p_0 \\
p_1 \\
p_2 \\
p_3 \\
p_4 \\
p_5 \\
\end{bmatrix}
$$

In this chapter it will be shown that Barrowes’ method actually does exactly the same as the described method of the 2D convolution in the Fourier domain.

5.1 The properties of a Toeplitz matrix

In short, the value of the entries in a Toeplitz matrix are constant along each diagonal (from top left to bottom right). A special form of Toeplitz matrices are block-Toeplitz matrices. In such a matrix every block is Toeplitz. On a higher level, the blocks that lie on the same diagonal are exactly the same. Our example matrix $A$ contains four $3 \times 3$ sub blocks that are Toeplitz. Next to that the top left block and the bottom right block are the same, so on the level of blocks these four blocks are also Toeplitz with each other. The special properties of $A$ can be used to calculate the product of $A$ and $p$ faster than the standard method, which multiplies each row of $A$ with $p$. Note that the standard method costs $n^2$ multiplications and almost $n^2$ additions when $A$ is of size $n \times n$, so this standard method is $O(n^2)$. In [3] it is shown that the multiplication of a Toeplitz matrix with a vector can be achieved in $O(n \log n)$. How this method works will be explained in detail for our example. The general idea is that the multiplication can be reduced
to a convolution of a 1D vector and a vector constructed from \( p \), which can be computed within the Fourier domain.

### 5.2 From a 2D problem to a 1D convolution

#### 5.2.1 A Toeplitz matrix is determined by a few specific values

But how do we get from a 2D block-Toeplitz matrix \( A \) to a 1D convolution? The idea is to put all the (possible) different values in a row and see what happens. Remember that blocks within each diagonal are exactly the same, so in our example, the values of the bottom right block are completely determined by the values in the top left block. In general, one only has to look at the first column and row of blocks. Within each block the values on each diagonal are constant, so the values within each block are determined when the values of the first column and row are known. So for our example, we have to look at three blocks and five values per block in order to determine the whole matrix. This can be seen in Figure 5.1.

![Figure 5.1: The matrix \( A \) is fully determined by the light gray values. The values of the dark gray row can be found in the light gray positions enclosed by a box.](image)

If the value of the light gray positions of \( A \) are known, then the whole matrix is determined. These 15 light gray values can be put in a row, ordering them from bottom left to top left to top right, first within each block and then on the higher level for the whole block structure. Let us call this vector \( b \). Figure 5.1 also shows that the values of the fifth row, which are accented dark gray, can be found also within the light gray area on the positions that are enclosed by a box. This is true for each row of any multilevel block-Toeplitz matrix. If one picks a certain value from a certain row, then this value can also be found in the most top left block within the same diagonal of blocks. This is also true for more levels of sub blocks. Within the block on the deepest level, the specified value can be found on every position of the diagonal, also on the most top left position, which is definitely within the light gray area. For example, take the sixth element of the fifth row of \( A \), which has a value of zero. It is located in the bottom right \( 3 \times 3 \) block of \( A \). This zero can also be found in the top left block on the same relative position, thus the third position of the second row. But within the top left block all the values on this specific diagonal are zero, so it can also be found on the second position of the first row. So in general, all values of a certain row can be found by moving top left along the diagonals until a value within the light gray area is reached. With this strategy of finding the same values of a row by moving along diagonals, one can also verify that the values of a row from left to right are in the same order within \( b \), simply because different diagonals never cross each other. Figure 5.2 shows \( b \) and it gives a hint about the structure of the rows of \( A \).
5.2. FROM A 2D PROBLEM TO A 1D CONVOLUTION

Figure 5.2: The rows of $A$ in vector $b$ which was constructed from the different elements of $A$.

5.2.2 The pattern of the rows

With a look at Figure 5.2 it also becomes clear that the values of each row form a certain pattern on the given vector $b$. If one fills the pattern of a row with the values of $p$, then it represents the multiplication of that row with $p$. Now, $p$ can be put into that pattern by inserting two zeros halfway. In general, the insertion of zeros depends on the block structure of $A$. A good way of finding the exact pattern in general is by looking at the bottom row of the matrix. Toeplitz matrices are symmetric around the diagonal from bottom left to top right, so the bottom row of the matrix can also be found in the first column. So the exact pattern to put $p$ in can be found by visiting all the elements of $b$ and $p$, stepping through them from left to right. Each time an element of $b$ is not situated on the most left column of $A$, a zero has to be inserted in the corresponding position in vector $p$. Let us call the resulting zero padded vector $p'$. Intuitively one would like to shift $p'$ along $b$, each time multiplying its values with the values right above it and adding them together. This looks a bit like a convolution already.

5.2.3 The convolution revealed

In our problem we want to multiply $p'$ and $b$ elementwise. This can be written as a convolution by reversing one of the two vectors, as explained in Section 3.6. Here, we choose to flip $b$. The example in Figure 5.3 shows us how the convolution works, by showing the calculation process for element number 14 of the resulting vector. The convolution sum is only calculated from $m = 0$ to 14 because for all other values of $m$, $\text{flip}(b)[m] = 0$ and thus all the other products are equal to zero.

\[
(flip(b) * p')[14] = \sum_{m=0}^{14} (flip(b))[m]p'[14 - m] = 6p_0 + 0p_1 + 2p_2 + 5p_3 + 3p_4 + 1p_5
\]  

But this is exactly the result of multiplying the bottom row of $A$ with $p$. In the same way, element number 13 of the convolution will be the same as the fifth row of $A$ multiplied with $p$, so ultimately one finds that elements number 7, 8, 9, 12, 13 and 14 of the convolution form respectively the multiplications of rows 1, 2, 3, 4, 5 and 6 of $A$ with $p$.

Thus the product of $A$ and $p$ can be written as a 1D convolution of vectors that are a lot smaller than the matrix $A$. Note that the product of $A$ and $p$ can only be written as a 1D
CHAPTER 5. BARROWES’ METHOD

5.3 The relation with the improved 2D convolution

Now, take a closer look at vector $b$ and $p'$ and the matrices $B$ and $P'$ from Chapter 4. $b$ is exactly the same as the rows of $B$ put behind each other. In the same way, $p'$ is the same as the rows of $P'$ behind each other. So the 1D convolution of Barrowes’ method works with the same values as the 2D convolution of Chapter 4. The only difference is that Barrowes’ method puts $B$ and $p'$ into 1D vectors. But the results are the same. The method that calculates the convolution within the Fourier domain is explained in detail for our example in Chapter 4. An important result is that the runtime is $O(n \log n)$ for a matrix $A$ of size $n \times n$. A guess for the total amount of work for Barrowes’ method is:

$$r_{barr(n)} = 4n + 48n \log 4n \quad \text{with } n = m_x m_y$$

$$= r_{conv(n)}$$

One could also see it from the other side. When encountering a problem of multiplying a block-Toeplitz matrix with a vector, one could rewrite it to a 2D convolution. Then, one simply calculates the 2D convolution in the Fourier domain.

5.3.1 The length of $b$ and $p'$

The length of $b$ and thus of $p'$ is bounded by $4n$. This can be seen easily in our example matrix $A$. All different values of $A$ can be found by going from bottom left to top left to top right on all levels of the matrix, the light gray positions in Figure 5.1. In general, say the size of the sub blocks is $m$ so that the number of blocks in a row or column is $\frac{n}{m}$. Then for a two level Toeplitz matrix in general one can find all the different values within $2 \frac{n}{m} - 1$ blocks and within each of these blocks on $2m - 1$ positions. This is bounded from above by $2 \frac{n}{m} 2m = 4n$. So the length of $b$ and $p'$ will be close to $4n$ which is indeed $O(n)$.

There is another way to conclude that the length of the vectors is bounded by $4n$. As stated, the vectors $b$ and $p'$ are just the 1D versions of $B$ and $P'$ from Chapter 4. The size of both $B$ and $P'$ are bounded by $2m_x \times 2m_y = 4m_x m_y = 4n$ so this also holds for $b$ and $p'$. 
5.3. THE RELATION WITH THE IMPROVED 2D CONVOLUTION

For general block-Toeplitz matrices, an upper bound for the length of the extracted vectors in Barrowes’ method will be $2^k$ where $k$ is the number of levels of the block-Toeplitz matrix. This can also be found in [3]. Because our problem involves 2-level block-Toeplitz matrices, the upper bound will be $4n$ for a Toeplitz matrix of size $n^2$.

5.3.2 The convolution and the final result

The 1D convolution of Barrowes’ method is the same as the 2D convolution of Chapter 4 as explained earlier. The only difference is that the matrices of the 2D convolution are now put into 1D vectors. In order to do a convolution in the Fourier domain, both vectors should have the same length so that is why $p'$ contained extra zeros on the right side of element $p_5$ in Figure 5.3. Both $flip(b)$ and $p'$ are transformed to the Fourier domain, then these vectors are multiplied elementwise and then the resulting vector is transformed back to the normal domain, let us call it $u'$, for it has still the same length as $b$. As shown earlier, for our example the elements of the vector $u$ are to be found in elements 7, 8, 9, 12, 13 and 14 of $u'$. In general they can be found by putting the pattern of a row totally on the right side as a mask over $u'$. The process will not be shown in detail because it is quite similar to the process of Chapter 4. So the product of $A$ and $p$ can be computed in just $O(n \log n)$ instead of the standard $O(n^2)$, which results in a significant runtime reduction for large values of $n$.

It is also easy to see where the resulting values $u$ can be found inside $u'$ by looking at the resulting matrices $U'$ and $U$ from Chapter 4. As explained, the final results were to be found in the $2 \times 3$ sub block in the bottom right corner of $U'$. By putting the rows of $U'$ behind each other, which is the same as $u'$, the given pattern appears.

Again, this convolution within the Fourier domain is valid and exact, the explanation is the same as in Section 4.2. Because the method does the same as the method of Chapter 4, the formula for the runtime is also the same. But there might be some extra work involved for Barrowes’ method. When the matrix $A$ and vector $p$ are given, the vector $b$ should be created, which costs some extra time. Next to that the pattern on the rows should be laid on the resulting vector $u'$ to get the vector $u$ we are looking for.

5.3.3 The runtime compared

With a few test cases it can be found that Barrowes’ method is a bit faster than the 2D convolution in the Fourier domain when working with small grids. When handling very large grids, the optimized 2D convolution is a bit faster. There is a good explanation for this runtime difference. When performing the convolution on the grids $B$ and $P'$, six FFT’s have to be performed, while the convolution in Barrowes’ method only needs three FFT’s. Together they should be performing the same number of operations. But for very small grids, the extra time needed to invoke an existing method like a FFT starts to play a significant role. It depends on the program or programming language that is used to perform the calculations, but in general invoking more functions costs more work which is significant when dealing with very small grids. For very large grids, the Fourier transforms are also very large. It also might depend on the computer or program, but in my test cases the 2D convolution in the Fourier domain was a bit faster than Barrowes’ method for very large grids. It might be that handling very large Fourier transforms will relatively slow down a bit for very large sizes, meaning that the work can be done a bit faster when splitting it up into six transforms instead of three. Another main issue is the length of each Fourier transform, because FFT’s fluctuate a bit in speed for different but close lengths. The six FFT’s for the 2D version are performed on other lengths than the three FFT’s in the 1D version so that can influence the relative speed of the FFT’s. In my test cases
the runtime of the two methods differed less than a factor 2 for grids between about 250 and 250000 grid points. The test cases can be found in Chapter 6.

5.4 Summary of Barrowes’ method

Barrowes’ method provides a fast algorithm to calculate the product of a block-Toeplitz matrix and a vector. When the matrix is of size $n \times n$, computing the product directly would be $O(n^2)$. With Barrowes’ method this can be done in just $O(n \log n)$. The memory usage of the direct product would be about $n^2$ because the whole block-Toeplitz matrix is needed. Barrowes’ method only needs at most $4n$ values of the matrix, $4n$ values for $p'$ and $4n$ for the resulting vector so that comes down to a memory space of about $12n$. The steps performed in Barrowes’ method are:

1. Extract all the (possible) different values from the block-Toeplitz matrix $A$ as explained above, store them in a vector $b$ and then flip $b$ to $b'$.
2. Pad the grid vector $p$ with zeros in a way that the described pattern occurs and that the length of the vector is the same as the length of $b$, call it $p'$.
3. Compute the Fourier transforms $\text{flip}(b)^f$ and $p'^f$ by using FFT’s, which results in two vectors in the Fourier domain.
4. Calculate the elementwise product of $\text{flip}(b)^f$ and $p'^f$, this gives a vector $u^f$.
5. Calculate the inverse Fourier transform of $u^f$, called $u'$, again using a FFT.
6. The values we are interested in can be found by laying the described pattern totally on the right side on $u'$ and then look on the positions where the pattern and $u'$ overlap.

As shown earlier, this method actually does the same as the improved 2D convolution. Just as with the improved 2D convolution, Barrowes’ method works for an arbitrary block-Toeplitz matrix and grid vector. The product of $A$ of size $n \times n$ and $p$ can be rewritten to a 2D convolution of matrices $B$ and $P$, where $P$ has size $m_x \times m_y$, when $n = m_x m_y$. If one replaces the occurrences of $n$ above with $m_x m_y$, then one can see that the given speed bound and used memory space are the same as in the summary of the improved 2D convolution method. If one starts with a problem of computing the 2D convolution of two grids, then it would not make much sense to use Barrowes’ method. For then, one first writes the convolution as a large matrix vector product, where most single values of the original problem occur multiple times in the block-Toeplitz matrix, and then Barrowes’ method extracts the original values back from the block-Toeplitz matrix. After that the same operation is done as the improved 2D convolution. So when both Barrowes’ methods and the improved 2D convolution do the same calculations eventually, then one only has to choose whether to put the values into 1D or 2D form.
Chapter 6

Runtime and memory usage of the treated methods

6.1 Testresults

Some test cases were performed in Matlab, version R2009b, to see how fast the treated methods are. The first method is the direct product of $A$ and $p$ from Chapter 5 which costs the same amount of operations as the direct 2D convolution of grid $B$ and grid $P$ of Chapter 4. It is actually the same method but then written into a matrix vector product. The other two methods are the methods described in these two chapters to speed up the calculation. The results can be found in Table 6.1. A graphical representation of the data can be found in Figure 6.1.

The tests were done on square input grids, where the number of rows and columns are powers of 2. The given runtimes are in seconds for a few different amounts of grid points. All the test cases are performed a few times to avoid wrong answers due to start up work or other things the computer might be busy with. Next to that a number of runs is averaged for the smaller grid sizes so that the resulting runtimes do not fluctuate too much. The tests are performed on a computer with Windows 7 on an Intel Xeon W3530 CPU running at 2.80 GHz and a memory of 4 GB.

<table>
<thead>
<tr>
<th>$n = m_x^2$</th>
<th>direct product $Ap$</th>
<th>direct convolution $B \ast P$</th>
<th>Barrowes’ method for $Ap$</th>
<th>convolution in Fourier domain for $B \ast P$</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>0.0000011</td>
<td>0.0000012</td>
<td>0.0000078</td>
<td>0.000041</td>
</tr>
<tr>
<td>16</td>
<td>0.0000013</td>
<td>0.0000027</td>
<td>0.000011</td>
<td>0.000047</td>
</tr>
<tr>
<td>256</td>
<td>0.00032</td>
<td>0.00029</td>
<td>0.00013</td>
<td>0.00031</td>
</tr>
<tr>
<td>1024</td>
<td>0.0063</td>
<td>0.0032</td>
<td>0.00038</td>
<td>0.00069</td>
</tr>
<tr>
<td>4096</td>
<td>0.10</td>
<td>0.038</td>
<td>0.0053</td>
<td>0.0042</td>
</tr>
<tr>
<td>16384</td>
<td>out of memory</td>
<td>0.51</td>
<td>0.019</td>
<td>0.022</td>
</tr>
<tr>
<td>65536</td>
<td>out of memory</td>
<td>7.5</td>
<td>0.11</td>
<td>0.078</td>
</tr>
<tr>
<td>1048576</td>
<td>out of memory</td>
<td>&gt; half an hour</td>
<td>3.0</td>
<td>0.77</td>
</tr>
</tbody>
</table>

Table 6.1: The runtime in seconds for four methods for different problem sizes.
6.2 About the runtime

The most general and important fact to notice is that the direct methods are the fastest for tiny grids and the two more complicated methods treated in the previous chapters are faster for larger grids. To be more precise: for grids larger than 250 grid points, the two treated method are faster than the direct methods. For very small grids the runtime for the direct methods is so small that their simple procedure is faster than the more complicated methods. For larger grids the number of calculations is increasing way too fast in the direct methods, so then the more complicated but smarter methods win.

The small difference between Barrowes’ method and the 2D convolution in the Fourier domain is also discussed in Section 5.3.3. The main difference between these two methods is whether the Fourier transforms are performed on \( m_x \times m_y \) grids or vectors of length \( m_x m_y \). As concluded in Section 4.4, the direct 2D convolution and the optimized 2D convolution are about as fast for a grid of 250 grid points. The test results do fit quite well with this conclusion, because the runtime for these two methods (and also for the direct product) are very close to each other for \( n = 256 \). Somewhere around 2500 or 3000 grid points, the two treated methods are about as fast, the improved convolution overtakes Barrowes’ method somewhere around there. A strange fact is that Barrowes’ method is once more a bit faster for \( n = 16384 \). On another computer where the tests were done too, the improved 2D convolution was faster than Barrowes’ method for all test cases with \( n \geq 4096 \). The general tendency is that Barrowes’ method is slower than the improved convolution for large \( n \). As stated earlier, they perform the same calculations, but in a form that is a bit different.

6.3 About the memory usage

As Table 6.2 shows, the direct product needs a lot of memory. The computer could not handle it for 16384 grid points or more. The other three methods have only a memory usage that is
6.4. CHOOSING 'GOOD' GRID SIZES

<table>
<thead>
<tr>
<th>Method</th>
<th>Memory Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>direct product $Ap$</td>
<td>$\sim n^2$</td>
</tr>
<tr>
<td>direct convolution $B \ast P$</td>
<td>$\sim 9n$</td>
</tr>
<tr>
<td>Barrowes’ method for $Ap$</td>
<td>$\sim 12n$</td>
</tr>
<tr>
<td>convolution in Fourier domain for $B \ast P$</td>
<td>$\sim 12n$</td>
</tr>
</tbody>
</table>

Table 6.2: The memory usage of the methods.

linear with respect to $n$ instead of quadratic. Therefore, the computer can handle these methods for very large grid sizes.

As stated, the direct product of $A$ and $p$ does the same as the direct 2D convolution of $B$ and $P$. The only difference is memory usage. The direct 2D convolution only needs about $9m_xm_y$ of memory space, because $P$ contains $m_xm_y$ elements and $B$ and the resulting matrix $U$ contain almost $4m_xm_y$. But the direct product of $A$ and $p$ needs about $m_x^2m_y^2$. That is because $B$ is a lot smaller than $A$. Because these two direct and simple calculation methods are about the same in runtime, one could simply always prefer the direct 2D convolution of $B$ and $P$ over the direct product of $A$ and $p$ because it needs far less memory space. Barrowes’ method and the improved 2D convolution use a bit more memory than the direct convolution, because the matrices or vectors have to be of the same size to perform a convolution in the Fourier domain, so $B$, $P'$ and $U$ all have about $4m_xm_y$ elements. But the bit of extra memory compared to the direct convolution is just a minor difference because the gain in runtime is far more significant and important when dealing with very large grids.

6.4 Choosing ‘good’ grid sizes

Let us take a more precise look at the size of the grids or vectors that have to be transformed to the Fourier domain. As stated several times, the FFT’s are fast when the size to be handled is a product of low prime numbers. If one may choose the size of the grid, it makes sense to choose powers of 2 for the length and width, so assume for now that $m_x$ and $m_y$ are powers of 2. But note that for a $m_x \times m_y$ grid, the grid $B$ will be of size $(2m_x - 1) \times (2m_y - 1)$ and the vectors for Barrowes’ method will be of length $(2m_x - 1)(2m_y - 1)$. Maybe it would be more wise to add one row and column of zeros to $B$, so that its size will be $2m_x \times 2m_y$ which are powers of 2. The vectors for Barrowes’ method could also be enlarged to a length of $4m_xm_y$, which is also a power of 2. Table 6.3 shows some interesting test results about this considerations. The results for Barrowes’ method and the improved 2D convolution are shown again, these are the same results as in Table 6.1. But now both methods are also tested for enlarged matrices so that the sizes for the FFT’s are powers of 2. The results can also be found in Figures 6.2 and 6.3.

The most important aspect to notice here is that the enlarged version is always as fast or faster than the original calculation. A bit more memory is needed but it is just a very small amount. So we can conclude that it is profitable to enlarge the vectors of length $(2m_x - 1)(2m_y - 1)$ to $4m_xm_y$ for Barrowes’ method or to enlarge the grids of the improved 2D convolution method from size $(2m_x - 1) \times (2m_y - 1)$ to $2m_x \times 2m_y$. It might look strange that the speed gain of the enlarged versions varies between a factor 1 and 5 compared to the original versions. Most likely this is caused by the behaviour of FFT’s. It depends quite a bit on the factorization of the size $(2m_x - 1)(2m_y - 1)$ for Barrowes’ method or the factorization of the sizes $(2m_x - 1)$ and $(2m_y - 1)$ for the improved convolution method. This effect is the most likely cause for the remarkable fact that for $n = 65536$ the convolution is faster than Barrowes’ method while the enlarged version of the convolution is still slower than the enlarged version of Barrowes’ method. The fluctuations in the test cases cannot be ignored totally. But there is still a tendency that
the enlarged version of the convolution is faster than the enlarged version of Barrowes’ method for large grids and vice versa for small grids, the same tendency as for the original versions.

The general message here is that one should choose the sizes of the matrices in such a way that all the 1D Fourier transforms are performed on lengths that are the product of low prime numbers, powers of 2 are best.

### Table 6.3: The runtime of the two treated methods compared to versions in which the grids and vectors are enlarged so that their sizes are powers of 2.

<table>
<thead>
<tr>
<th>$n = m^2$</th>
<th>Barrowes’ method for $Ap$</th>
<th>Barrowes’ method enlarged</th>
<th>convolution in Fourier domain, for $B * p$</th>
<th>convolution in Fourier domain, enlarged</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>0.0000078</td>
<td>0.0000078</td>
<td>0.000041</td>
<td>0.000041</td>
</tr>
<tr>
<td>16</td>
<td>0.000011</td>
<td>0.000010</td>
<td>0.000047</td>
<td>0.000045</td>
</tr>
<tr>
<td>256</td>
<td>0.00013</td>
<td>0.000059</td>
<td>0.00031</td>
<td>0.00016</td>
</tr>
<tr>
<td>1024</td>
<td>0.00038</td>
<td>0.00022</td>
<td>0.00069</td>
<td>0.00056</td>
</tr>
<tr>
<td>4096</td>
<td>0.0053</td>
<td>0.0010</td>
<td>0.0042</td>
<td>0.0020</td>
</tr>
<tr>
<td>16384</td>
<td>0.019</td>
<td>0.013</td>
<td>0.022</td>
<td>0.018</td>
</tr>
<tr>
<td>65536</td>
<td>0.11</td>
<td>0.045</td>
<td>0.078</td>
<td>0.047</td>
</tr>
<tr>
<td>1048576</td>
<td>3.0</td>
<td>1.5</td>
<td>0.77</td>
<td>0.48</td>
</tr>
</tbody>
</table>
6.4. CHOOSING 'GOOD' GRID SIZES

Figure 6.2: The runtime of Barrowes' methods compared to its enlarged version, on a logarithmic scale.

Figure 6.3: The runtime of the improved 2D convolution compared to its enlarged version, on a logarithmic scale.
Chapter 7

Solving the whole problem

As explained in Section 2.1, the whole 3D problem can be calculated with five 2D convolutions or five times Barrowes’ method for a block-Toeplitz matrix vector product. A graphical representation in one matrix vector product can be found in Figure 2.2. Now one could ask whether calculating these five subproblems independently is the fastest way of computing the final results or not. This chapter provides an answer to that question.

7.1 Do not merge independent FFT’s

Because the runtime of the treated methods depends heavily on the speed of the FFT’s, we should examine the speed of FFT’s for different input sizes a bit further. As stated in Section 3.5 the speed of an FFT of length \( n \) takes about \( 4n \log n \) basic calculations. Consider for example the problem that two independent convolutions have to be calculated on grids or vectors with a total size of \( n \) elements. For each of these convolutions three FFT’s have to be performed with length \( n \), so that would cost about \( 3 \cdot 4n \log n \) calculations per convolution, so \( 2 \cdot 12n \log n = 24n \log n \) for two convolutions.

Now, what happens with the number of calculations when the two convolutions are calculated together as one problem? In that case, only three FFT’s have to be performed instead of six. But these FFT’s are each now of length \( 2n \). So the total amount of calculations will be about \( 3 \cdot 4 \cdot 2n \log 2n = 24 \log 2n \). The only difference which the previous result is the term \( 2n \) within the log function instead of the term \( n \). So the number of calculations is always larger than the independent calculation of the two convolutions. So when two problems are independent of each other it is wise to compute the results also independently. One could also see it in a different way. Because the runtime of an FFT is larger than \( O(n) \), the amount of calculations increases with more than a factor 2 when the length is doubled. So calculating two FFT’s separately is faster than merging them into one larger FFT. Only for very small grids merging might be faster because less FFT’s have to be invoked which minimizes the extra startup time.

7.2 A special property of the influence grids

Now, take a closer look at Figure 2.2 from Section 2.1. The influence grids \( B_{zz}, B_{xx}, B_{xy}, B_{yx} \) and \( B_{yy} \) from which the block-Toeplitz sub matrices \( A_{zz}, A_{xx}, A_{xy}, A_{yx} \) and \( A_{yy} \) can be constructed, have some special properties. For the problem VORtech wants to solve, these matrices have the following property that is relevant in this chapter:

- \( B_{xy} = B_{yx} \)
7.3 Handling the five subproblems efficiently

Now some different methods for calculating the whole problem will be discussed. We start with the most simple and obvious method: calculating the five subproblems one after the other. After that some faster methods will be discussed.

7.3.1 The five subproblems calculated totally independently

We have five subproblems and at least we have the option of calculating these five problems independently of each other. For now, we assume that the input grids are given and we do not extract the new grid out of the resulting matrix. So for the case of simplicity we will only look at the main steps, involving the FFT’s. We assume for the moment that the grids are used in 2D form. Then we need 2D FFT’s, these will be denoted with $FFT$. Elementwise products will be denoted with ‘$\ast$’. Here are the steps for the convolution of $B_{xx}$ and $P_x$:

1. $B_{xx}^f = FFT2(B_{xx})$.
2. $P_x^f = FFT2(P_x)$.
3. $U_{xx}^f = B_{xx}^f \ast P_x^f$.
4. $U_{xx} = IFFT2(U_{xx}^f)$.

The other four subproblems can be calculated in the same way. Between each two subproblems the memory can be cleared to keep memory usage low. The first guess for the total runtime that one could safely make is that it will be five times as long as the calculation of just one subproblem. In Section 4.3 we have seen that a good guess for the number of calculations of one subproblem with a $m_x \times m_y$ grid is $4n + 48n \log 4n$ with $m_x m_y = n$. Here, the term $48n \log 4n$ dominates for large grids. Calculating the five subproblems independently should take about $20n + 240n \log 4n$. But maybe we can use the given property of the influence grids to speed up the process?

7.3.2 Optimized independent calculation

The five subproblems are not totally independent. For example, one subproblem involves the convolution of $B_{xx}$ and $P_x$ and another subproblem involves the convolution of $B_{yx}$ and $P_x$. For both convolutions the FFT of the zero-padded grid $P_x'$ is needed. The same holds for $P_y'$, its FFT is needed twice. Next to that, $B_{xy}$ and $B_{yx}$ both occur in one subproblem, although these grids are exactly the same, so their Fourier transforms are also the same. Thus when we compute the subproblems totally independently, some FFT’s are calculated twice, three FFT’s to be exact. The number of calculations can be decreased: one could store the result of these FFT’s when they are calculated for the first time and then use it also in the other subproblem. For example, one has to calculate the Fourier transforms of $B_{xx}$ and $P_x'$ in order to perform the elementwise multiplication in the Fourier domain. Also the Fourier transforms of $B_{yx}$ and $P_x'$ have to be calculated. Now, when the Fourier transforms of $B_{xx}$, $B_{yx}$ and $P_x'$ are calculated, one can compute the elementwise product of $B_{xx}^f$ and $P_x'^f$ and directly after that the elementwise product of $B_{yx}^f$ and $P_x'^f$. Of course, the extra amount of memory that is needed to do these two calculations at once is a bit higher. Here are the steps that are needed to calculate the five subproblems without redundant calculations:

1. $B_{xx}^f = FFT2(B_{xx})$. 

2. \( P^I_x = \text{FFT}_2(P_x) \).

3. \( U^I_{xx} = B^I_{xx} \ast P^I_x \).

4. \( U_{xx} = \text{IFFT}_2(U^I_{xx}) \).

5. \( B^I_{yx} = \text{FFT}_2(B_{yx}) \).

6. \( U^I_{yx} = B^I_{yx} \ast P^I_x \).

7. \( U_{yx} = \text{IFFT}_2(U^I_{yx}) \).

8. \( P^I_y = \text{FFT}_2(P_y) \).

9. \( U^I_{xy} = B^I_{yx} \ast P^I_y \).

10. \( U_{xy} = \text{IFFT}_2(U^I_{xy}) \).

11. \( B^I_{yy} = \text{FFT}_2(B_{yy}) \).

12. \( U^I_{yy} = B^I_{yy} \ast P^I_y \).

13. \( U_{yy} = \text{IFFT}_2(U^I_{yy}) \).

14. \( B^I_{zz} = \text{FFT}_2(B_{zz}) \).

15. \( P^I_z = \text{FFT}_2(P_z) \).

16. \( U^I_{zz} = B^I_{zz} \ast P^I_z \).

17. \( U_{zz} = \text{IFFT}_2(U^I_{zz}) \).

The steps look quite similar to computing the steps in Section 7.3.1 five times, one time for each subproblem. But in this optimized method, these five repetitions are mixed up a bit to avoid the redundant FFT’s. Instead of \( 5 \cdot 4 = 20 \) steps, only 17 steps are performed. As a result, the total number of FFT’s is reduced from 15 to just 12. But still five elementwise multiplications in the Fourier domain are computed. This means that our guess for the total number of calculations can be reduced to \( 20n + 12 \cdot 4(4n \log 4n) = 20n + 192n \log 4n \). For large values of \( n \) the total runtime is therefore reduced to the amount of time needed for four subproblems instead of five, because \( 4(4n + 48n \log 4n) = 16n + 192n \log 4n \). Here the dominant part is \( 192n \log 4n \) which is the same as in the formula we just found.

This method requires a bit more memory because the steps for the different subproblems are mixed. But just a few more matrices have to be stored at the same time. In the steps above, none of the matrices in the Fourier domain have to be loaded in the memory for more than five steps. When the extra memory usage is not an issue, one could change the order of the steps, as long as the given matrices on the right hand side of each formula are known on the moment that calculation is done.
7.3.3 Barrowes’ method: four subproblems combined, version 1

Barrowes’ method works for a product of a given block-Toeplitz matrix and a vector. Let us say we calculate the convolution of $A_{zz}$ and $P_z$ independently from the rest, but the other four subproblems will be combined into one new problem. We can write it as one larger matrix vector product in the following way:

\[
\begin{pmatrix}
    u_x \\
    u_y
\end{pmatrix} =
\begin{pmatrix}
    A_{xy} & A_{yy} \\
    A_{xx} & A_{yx}
\end{pmatrix}
\begin{pmatrix}
    A_{xx} & A_{yy}
\end{pmatrix}
\begin{pmatrix}
    p_x \\
    p_y
\end{pmatrix}
\]

This is a correct representation of the four subproblems, because $u_x = A_{xx}p_x + A_{xy}p_y$ and $u_y = A_{yy}p_y + A_{yx}p_x$, which is also explained in [1]. Because $B_{xy}$ and $B_{yx}$ are exactly the same, the block-Toeplitz matrices $A_{xy}$ and $A_{yx}$ formed from these grids are also the same. In Figure 7.1 is shown that these two matrices lie on the same diagonal. Therefore, the whole matrix is block-Toeplitz, but now it is a 3-level block-Toeplitz matrix. We can use Barrowes’ method to transform this matrix vector product into a convolution.

We will take a closer look at this problem with a small example. Let us take a $2 \times 2$ input grid, where each grid point consists of a position in 3D space, thus each grid point has a $x$-, $y$- and $z$-component. Then, all the influence grids are of size $3 \times 3$. From these influence grids the corresponding block-Toeplitz matrices can be created, which are of size $4 \times 4$. Let us say that these block-Toeplitz matrices are defined as:

\[
\begin{pmatrix}
    4 & 5 & 7 & 8 \\
    3 & 4 & 6 & 7 \\
    1 & 2 & 4 & 5 \\
    0 & 1 & 3 & 4
\end{pmatrix}
\begin{pmatrix}
    14 & 15 & 17 & 18 \\
    13 & 14 & 16 & 17 \\
    11 & 12 & 14 & 15 \\
    10 & 11 & 13 & 14
\end{pmatrix}
\begin{pmatrix}
    24 & 25 & 27 & 28 \\
    23 & 24 & 26 & 27 \\
    21 & 22 & 24 & 25 \\
    20 & 21 & 23 & 24
\end{pmatrix}
\begin{pmatrix}
    34 & 35 & 37 & 38 \\
    33 & 34 & 36 & 37 \\
    31 & 32 & 34 & 35 \\
    30 & 31 & 33 & 34
\end{pmatrix}
\]

For this example, the matrix vector product for the four combined subproblems is:

When Barrowes’ method is applied to this problem, all the light gray accented values from Figure 7.2 are put in a vector $b$ and this vector is flipped. The zero-padded input vector is $p'$. Then, flipped vector $b$ and $p'$ are convolved. The resulting vector contains the values we wanted to compute.
7.3. HANDLING THE FIVE SUBPROBLEMS EFFICIENTLY

Figure 7.2: The block-Toeplitz matrix for four subproblems together. This matrix is fully determined by the light gray values.

But how much calculations are needed for this method? As already mentioned, the matrix for the combined subproblems is a 3-level block-Toeplitz matrix. Let us say each of the four largest blocks has size $n \times n$, so the total matrix has size $2n \times 2n$. In Section 5.3.1 we saw that the length of the vectors for the 1D convolution is bounded by $(2^3)2n = 16n$ when dealing with a $2n \times 2n$ 3-level block-Toeplitz matrix. But in our case, there are always four blocks on the highest level, because the matrix combines four subproblems. Values from three of these four blocks are needed to create vector $b$. So the length of $b$ and $p'$ is bounded by $3(2^2)n = 12n$. The three FFT's that are needed together with the elementwise product takes about $12n + 3 \cdot 4(12n \log 12n) = 12n + 144n \log 12n$. The fifth subproblem, $A_{zz}p_z$, costs about $4n + 48n \log 4n$ calculations. Thus, a good guess for the total number of calculations to solve all five subproblems is $16n + 144n \log 12n + 48n \log 4n$. By substituting $\log 12n$ for the term $\log 4n$ we can write this as $16n + 192n \log 12n$, which is approximately the same.

Is this number of calculations higher or lower than the number of calculations found in Section 7.3.2? Note that $\log 12n = \log 4n + \log 3$. So we can rewrite the formula we just found:

$$16n + 144n \log 12n + 48n \log 4n$$

$$= 16n + 144n(\log 4n + \log 3) + 48n \log 4n$$

$$= 16n + 192n \log 4n + 144n \log 3$$

$$> 16n + 192n \log 4n + 228n$$

$$= 244n + 192n \log 4n$$

$$> 20n + 192n \log 4n$$

The inequality above shows that the found formula for the number of calculations is larger than the number of calculations for the method of Section 7.3.2. This means that the method of this section is slower than the method of Section 7.3.2, although it is just a small difference.

7.3.4 Barrowes’ method: four subproblems combined, version 2

In [3] is explained that each element of a block-Toeplitz matrix may be a matrix itself, which does not have to be Toeplitz. We could put the chosen four subproblems into one big block-Toeplitz matrix of size $n \times n$ by letting every element be a $2 \times 2$ grid, filled with one value of each of the four subproblems. Now, we build up the matrix for the example introduced in Section 7.3.3.
Chapter 7. Solving the Whole Problem

Figure 7.3: The block-Toeplitz matrix where each element is a 2×2 matrix on its own. The matrix is fully determined by the gray accented elements.

In Figure 7.3 can be seen that each element is a 2×2 grid. The values in it are arranged in a way that each grid is Toeplitz, but that does not help us much. In the convolution that can be extracted from this matrix vector product, the grids are multiplied elementwise with 2×1 grids from the column vector in the Fourier domain. The multiplication of a 2×2 grid with a grid from the column vector is a multiplication of the four values of the 2×2 grid with the two values of the 2×1 grid, resulting in four new values. For example, the product of the top left grid of the matrix with the top grid of the vector results in these four values: 24px0, 14py0, 4px0 and 24py0. So eventually, all four values of each 2×2 grid are multiplied with four other values. One could choose to store only the three different values of each 2×2 grid because each of these grids is Toeplitz, but that will not decrease the number of calculations.

So each multiplication of grids consists of four elementwise products. In our example the convolution consists of vectors with a length of 9 grids, one of those vectors is formed from the gray accented grids from Figure 7.3. Because each grid contains four values, the total number of values in the vectors that are convolved is 36. But if we compute the result of the four subproblems independently, we have four times a convolution on vectors of length 9, so 36 values are convolved with 36 other values. Barrowes’ method for the combined problem just merges these four convolutions into one convolution of the same length on elements that each contain four values. So actually this method does not decrease the number of calculations nor is it simpler than calculation the problems independently. We can conclude that Barrowes’ method on the combined problem is not faster than solving the four subproblems one by one. And it is definitely slower than the method of Section 7.3.2.

7.4 The best way of solving a 3D problem

Now we can conclude that the fastest way of dealing with the whole 3D problem containing five subproblems, is by just solving the five subproblems independently, but not just one after the other. Because some FFT’s occur twice, some effort can be saved by calculating these FFT’s only once and store their result for a while. By performing the calculations in a certain order, as shown in Section 7.3.2, the memory usage increases just a bit. But it reduces the runtime with almost 20%, so the total runtime should be about four times the runtime for a subproblem, although we solve five subproblems. A good guess for the number of calculations is 20n + 192n log 4n where n = mxmy is the number of grid points in each subproblem.
7.5 An iterative process for the whole problem

The total problem, containing five subproblems resulting in five convolutions, has to be solved several times with an iterative process. All the influence grids are the same throughout the whole iteration, but the input grids are updated every step. The resulting grids from the last step is the input for the next step. The number of steps typically lies between 10 and 100 for the contact problems VORtech has to solve. The steps listed in Section 7.3.2 can be used for each step, but there is a lot of redundancy. Because the influence grids are the same during the iteration, one only has to calculate the Fourier transform of these grids once and store them once they are computed. The steps needed for the iterative process using $t$ steps are shown here:

1. $B_{xx}^f = FFT2(B_{xx})$.
2. $B_{yx}^f = FFT2(B_{yx})$.
3. $B_{yy}^f = FFT2(B_{yy})$.
4. $B_{zz}^f = FFT2(B_{zz})$.
5. repeat $t$ times:
   (a) $P_x^f = FFT2(P_x)$.
   (b) $P_y^f = FFT2(P_y)$.
   (c) $P_z^f = FFT2(P_z)$.
   (d) $U_{xx}^f = B_{xx}^f * P_x^f$.
   (e) $U_{yy}^f = B_{yy}^f * P_y^f$.
   (f) $U_{zx}^f = B_{zx}^f * P_z^f$.
   (g) $U_{yx}^f = FFT2(U_{yx}^f)$.
   (h) $U_{yy}^f = FFT2(U_{yy}^f)$.
   (i) $U_{xy}^f = FFT2(U_{xy}^f)$.
   (j) $U_{xy}^f = FFT2(U_{xy}^f)$.
   (k) $U_{yx}^f = FFT2(U_{yx}^f)$.
   (l) $U_{zz}^f = FFT2(U_{zz}^f)$.
   (m) $P_z = IFFT2(U_{zz}^f)$.
   (n) $P_x = U_{xx} + U_{xy}$.
   (o) $P_y = U_{yy} + U_{yx}$.

There are four FFT's outside the loop and eight inside. Each FFT computes the Fourier transform for $4n$ elements, so $16n \log 4n$ per FFT. The Fourier transforms are the bottleneck of the whole computation. The total number of calculations for the FFT's is about $(64 + 128t)n \log 4n$ with $t$ the number of steps.

But is it not possible to do all the steps within the Fourier domain without switching to the normal domain between steps? If that would be possible, the number of Fourier transforms would be brought back to 10, independent of the number of steps. The consequence would be that the running time of the algorithm would be $O(n^2)$ instead of $O(n^2 \log n)$. But this optimization is not possible because of the form of the convolutions that CONTACT needs to
compute. The area on the grid where two physical object make contact is not a rectangle in general, but the algorithm works with matrices that are rectangular. Therefore, some values in the resulting grid have to be altered ‘by hand’ when the result of a step is computed. But altering one value in the resulting grid means that all values of the corresponding matrix in the Fourier domain change, and therefore we cannot do such justifications within the Fourier domain.

Note that the Fourier transform of the different influence grids $B$ are calculated only once, while the Fourier transform of the input grid $P$ has to be computed for every step in the iteration. Chapter 8 treats the optimization of the computation of the Fourier transforms that occur in our problem. The Fourier transform of the influence grids can be optimized a lot, but this can only result in a small reduction in total runtime of the whole iteration because the loop from step 5 is the bottleneck of the process. We can expect a larger reduction in runtime when the Fourier transform of grid $P$ can be done faster.
Chapter 8

Further optimization of the subproblems

Now, one could have noticed that only one special property of the grids of the subproblem is described and used in Section 7.2. But actually the influence grids of the subproblems have some more properties:

- All grids $B$ and $P$ are real-valued.
- $B_{zz}$ is symmetric in the horizontal, vertical and both diagonal axes.
- $B_{xx}$ and $B_{yy}$ are symmetric in the horizontal and vertical axis.
- $B_{xy}$ and $B_{yx}$ are symmetric in both diagonal axis. Next to that they are anti-symmetric around the horizontal and vertical axis.

These properties could not have helped us much for the decision we had to make in Section 7.3. But these properties might be used to speed up the calculation within the subproblems. The symmetric properties of the different influence grids also results in special properties in the resulting Fourier transformed matrices. But the first property of the list above is the most important. Section 8.1 deals with the optimization of real valued input grids. In Section 8.2 the possible optimizations for real input with symmetric properties are treated. Sections 8.2.1 and 8.2.2 are examples of optimizations for grids like $B_{xx}$, $B_{yy}$ and $B_{zz}$, and Section 8.2.3 deals with possible optimization of grids like $B_{xy}$ and $B_{yx}$.

8.1 Properties of the Fourier transform for real valued input

FFT’s are capable of computing the Fourier transform of complex vectors or grids. Real valued vectors or grids have the special property that all complex parts of the elements are zero. The result is that the Fourier transform of a real valued grid is complex conjugated around the center. This is not exactly true when the Fourier transform is calculated by taking the product of a Fourier matrix from Chapter 3 with the grid. The result must be shifted so that the first Fourier coefficient, $F[0,0]$, lies in the center of the matrix in the Fourier domain. The Fourier transforms of Chapter 3 centers the computed matrix on the top left instead of the center. But when shifted correctly the conjugate symmetry is revealed. For example, take a look at the matrix $B^f$ in Chapter 4. When the top left value is shifted to the center, the matrix looks like:
CHAPTER 8. FURTHER OPTIMIZATION OF THE SUBPROBLEMS

This matrix contains exactly the same values as in Chapter 4 but then shifted. Now it can be seen that this matrix is complex conjugated around the center point. So the real part of the matrix is normal symmetric and the imaginary part is anti-symmetric. The grid B from which this matrix was derived is real valued, but has no symmetric properties or whatsoever.

Because the Fourier transform of a real valued grid is complex conjugated, only half of the elements have to be computed, the other half can be derived from it. It depends on the used libraries whether this really results in a faster computation. This is an important optimization, more important than the further optimizations in the next section. As explained in Section 7.5 the Fourier transform of the input matrix has to be calculated for each step of the iteration. As explained in the next section, the Fourier transform of the influence matrices can be further optimized, but these transforms are only calculated once for the whole iteration.

8.2 Properties of the Fourier transform for symmetric input

8.2.1 A 1D example

Now, consider an input grid or vector with a certain symmetry. Here is a simple example of a real valued vector \( v \) with even symmetry around the center. Next to this vector the shifted version of \( v \) is shown.

\[
\begin{bmatrix}
0 & 1 & 2 & 4 & 2 & 1 & 0 \\
v
\end{bmatrix}
\]

\[
\begin{bmatrix}
4 & 2 & 1 & 0 & 0 & 1 & 2 \\
v \text{ shifted}
\end{bmatrix}
\]

Actually, the calculation of the Fourier coefficient with Formula 3.2 from Chapter does not result in the given \( v^f \) when we give \( v \) as input. The reason is the same as in Section 8.1: the Formula does not automatically center around the center of the input vector. To get the obtained result \( v^f \) one should shift the values of \( v \) so that the center value is at the first position of the vector. This is the reason that the shifted version of \( v \) is also displayed. The Fourier transform of the shifted version of \( v \) is displayed below, together with the reshifted version \( v^f \).

\[
\begin{bmatrix}
10 & 6.05 & 1.31 & 1.64 & 1.64 & 1.31 & 6.05 \\
v^f \text{ shifted}
\end{bmatrix}
\]

\[
\begin{bmatrix}
1.64 & 1.31 & 6.05 & 10 & 6.05 & 1.31 & 1.64 \\
v^f
\end{bmatrix}
\]

Next to the fact that this vector has even symmetry, it is also real valued. So this anti-symmetry is now even a normal even symmetry. The reason that the Fourier transform has only real values can be explained by taking a closer look at Formula 3.2. In this formula the term \( e^{-i \frac{2\pi}{n} u j} \) occurs. This can be written as \( \cos(\frac{2\pi}{n} u j) + i \cdot \sin(\frac{2\pi}{n} u j) \). When all the sine parts are zero, then the Fourier transform would only contain real values. This means that the Fourier series is built up from cosine parts only. Because all cosine parts have an even symmetry around the
vertical axis, the whole Fourier series is an even function. So if we have an input function with even symmetry that can be represented exactly with a Fourier series, then the Fourier series is also an even function and thus all the Fourier coefficients are real valued. One should notice that this is in general not true for complex input vectors with even symmetry.

In order to calculate the whole Fourier transform of the example vector \( v \) one can apply a DCT, a discrete cosine transform. Only four coefficients have to be calculated, but all the seven input values are needed for that. The four different values of \( v_DCT \), a discrete cosine transform. Only four coefficients have to be calculated, but all the seven input values are needed for that. The four different values of \( v \) can be calculated with the following formula which is a special kind of DCT, by taking \( n = 7 \). One should use the shifted version of \( v \) as input for the formula.

\[
F[u] = \sum_{j=0}^{n-1} v[j] \cos\left(\frac{2\pi}{n} u j\right) \quad \text{for } u = 0, 1, \ldots, \frac{n-1}{2} \tag{8.1}
\]

Instead of a sine and cosine part, only the cosine part has to be calculated. Next to that, only the first half of the coefficients has to be calculated, only up to \( u = \frac{n-1}{2} \) instead of \( u = n - 1 \). For a grid with \( n^2 \) points the calculation of the Fourier transform with this Formula is still \( O(n^2) \).

We need a special FFT that can solve this equation in \( O(n \log n) \), just as we need FFT’s to calculate Formula 3.2 more efficient.

### 8.2.2 A 2D example

Now, an example for the 2D case will be given. In this example, grid \( V \) is symmetric around the vertical and horizontal axis as. This is quite interesting because the grids \( B_{xx}, B_{yy} \) and \( B_{zz} \) also have this property. Let \( V \) be defined as below. Next to \( V \) the shifted version is shown, this shifted version will be the input for the FFT. The shifting is done by moving the center to the top left, so all values are shifted two positions to the left and upwards.

\[
\begin{bmatrix}
1 & 2 & 4 & 2 & 1 \\
2.5 & 3 & 6 & 3 & 2.5 \\
5 & 7 & 9 & 7 & 5 \\
2.5 & 3 & 6 & 3 & 2.5 \\
1 & 2 & 4 & 2 & 1 \\
\end{bmatrix}
\quad \begin{bmatrix}
9 & 7 & 5 & 5 & 7 \\
6 & 3 & 2.5 & 2.5 & 3 \\
4 & 2 & 1 & 1 & 2 \\
4 & 2 & 1 & 1 & 2 \\
6 & 3 & 2.5 & 2.5 & 3 \\
\end{bmatrix}
\]

This results in the Fourier transform for the shifted version of \( V \). Next to it \( V^f \) is shown, the version of the Fourier transform that is shifted back, so that one can see the symmetry. The values of both matrices are rounded.

\[
\begin{bmatrix}
87 & 20.1 & 8.9 & 8.9 & 20.1 \\
27.3 & 1.7 & 0.2 & 0.2 & 1.7 \\
11.7 & 1.3 & -2.7 & -2.7 & 1.3 \\
11.7 & 1.3 & -2.7 & -2.7 & 1.3 \\
27.3 & 1.7 & 0.2 & 0.2 & 1.7 \\
\end{bmatrix}
\quad \begin{bmatrix}
-2.7 & 1.3 & 11.7 & 1.3 & -2.7 \\
0.2 & 1.7 & 27.3 & 1.7 & 0.2 \\
8.9 & 20.1 & 87 & 20.1 & 8.9 \\
0.2 & 1.7 & 27.3 & 1.7 & 0.2 \\
-2.7 & 1.3 & 11.7 & 1.3 & -2.7 \\
\end{bmatrix}
\]

Note that \( V^f \) contains nine different values, the other values can be derived from these nine values. These nine values can also be found using a formula similar to Formula 8.1. We just expand this formula for the 2D case. If one fills in this formula with \( n, m = 5 \) for the 2D example, one finds the values of the bottom right \( 3 \times 3 \) block of \( V^f \). These values are all the values we need to know to determine the whole matrix \( V^f \). The formula for the 2D case is:
\[ F[u, w] = \sum_{j=0}^{n-1} \sum_{k=0}^{m-1} V[j, k] \cos(2\pi \left( \frac{uj}{n} + \frac{wk}{m} \right)) \]

for \( u = 0, 1, \ldots, \frac{n-1}{4} \)

for \( w = 0, 1, \ldots, \frac{m-1}{4} \) \hspace{1cm} (8.2)

Now, what happens when the input grid \( V \) is not only symmetric in the horizontal and vertical axis, but also in both diagonal axes, like \( B_{zz} \)? When this problem is solved with the formula above for the shifted version of \( V \), this results in a square block of values of about one quarter of the original size, just as in the example above. The difference is that this resulting block of values is symmetric around the diagonal from top left to bottom right. So in this case one only has to calculate roughly half of the values compared to the formula above. The only question is whether an FFT exists for calculating the values we need. Next to that, this can only be applied to \( B_{zz} \).

### 8.2.3 A different 2D example

Now we treat a different example, a grid with the same properties as \( B_{xy} \) and \( B_{yx} \). As shown in the beginning of this chapter, this grids are symmetric in the diagonals and anti-symmetric in the horizontal and vertical axis. The center row and center column are totally zero. An example grid \( V \) with these properties is:

\[
\begin{bmatrix}
1 & 2 & 0 & -2 & -1 \\
2 & 4 & 0 & -4 & -2 \\
0 & 0 & 0 & 0 & 0 \\
-2 & -4 & 0 & 4 & 2 \\
-1 & -2 & 0 & 2 & 1
\end{bmatrix}
\]

\[
\begin{bmatrix}
0 & 0 & 0 & 0 & 0 \\
0 & 4 & 2 & -2 & -4 \\
0 & 2 & 1 & -1 & -2 \\
0 & -2 & -1 & 1 & 2 \\
0 & -4 & -2 & 2 & 4
\end{bmatrix}
\]

The Fourier transform of the shifted grid and the reshifted version are:

\[
\begin{bmatrix}
0 & 0 & 0 & 0 & 0 \\
-2.2 & -2.2 & 2.2 & 24.8 \\
0 & 0 & 0 & 0 & 0 \\
-2.2 & -2.2 & 2.2 & 24.8 \\
0.2 & 2.2 & -0.2 & -2.2 & -2.2 \\
0 & 0 & 0 & 0 & 0
\end{bmatrix}
\]

\[
\begin{bmatrix}
-0.2 & -2.2 & 0 & 2.2 & 0.2 \\
-2.2 & -2.4 & 0 & 24.8 & 2.2 \\
0 & 0 & 0 & 0 & 0 \\
2.2 & 24.8 & 0 & -24.8 & -2.2 \\
0.2 & 2.2 & -0.2 & -2.2 & -0.2
\end{bmatrix}
\]

If we apply Formula 8.2 to this example, then it produces exactly the bottom right \( 3 \times 3 \) block of \( V^f \). This is actually quite remarkable because the rows and columns of \( V \) have an odd symmetry. One might expect that the real part of the Fourier transform equals zero, but this is not the case. Remark that \( V \) has even symmetry around the diagonals. Apparently, a grid with the given properties can also be described with a series of only cosine functions. If we apply Formula 8.2 and replace the cosine with a sine function, then all resulting values are smaller than \( 10^{-14} \), so approximately zero. This result already indicates that all the sine parts of the Fourier series are zero.

### 8.3 Conclusion

We can now conclude that all five grids \( B_{zz}, B_{xx}, B_{xy}, B_{yx} \) and \( B_{yy} \) can be computed by applying Formula 8.2 to their shifted versions. This formula calculates only about a quarter of the Fourier coefficients and it only calculates the real part of the Fourier series. Therefore, this
formula can be potentially up to eight times faster than the standard DFT. But note that there has to be a special FFT that calculates this formula efficiently, that is the only way we can really decrease the calculation time. A second important aspect is that the resulting matrices should be symmetrically enlarged in the proper way in order to get the whole Fourier transforms of the grids. Or we need a special way of using the smaller matrix for the elementwise product with another, larger Fourier transformed matrix. Inevitable, this costs some extra time.

The input grid \( P \) is real valued but does not have symmetric properties. Its Fourier transform can be calculated by calculating only half of the elements and then derive the other half from the found results. Thus, an optimal computation of the Fourier transform of \( P \) can be up to two times faster. Reducing the runtime with a factor 2 here is more important for our problem than reducing the calculation of the Fourier transforms of the influence grids with a factor 8. The reason is explained in Section 7.5: the Fourier transform of the input grid has to be calculated for each step of the total iteration, but the Fourier transform of the influence grids are only calculated once.
CHAPTER 8. FURTHER OPTIMIZATION OF THE SUBPROBLEMS
Chapter 9

Implementation

Now that we have given a theoretical backbone, the optimizations can be implemented. Of course there are a lot of different programming languages that one could use. Next to that a lot of libraries exist already. The program CONTACT is based on Fortran, and within Fortran we could make use of the Intel® Math Kernel Library, which will be referred to as the Intel library or Intel functions from now on. In this chapter a short explanation will be given about using the Intel library to compute a convolution in the Fourier domain. Detailed information about using Intel’s library can be found in [5].

9.1 Intel’s convolution functions

The Intel library contains a set of functions to compute a convolution directly. Here is a piece of code that is used in CONTACT to calculate a convolution:

```fortran
> status = vsldconvnewtask(task, VSL_CONV_MODE_AUTO,&
   ndim, SHAPE(x), SHAPE(y), SHAPE(z))
> status = vsldconvexec(task, x, xstride,&
   y, ystride, z, zstride)
> status = vslconvdeletetask(task)
```

First, a convolution task is created. Such a task is necessary to compute a convolution. The parameter task is a pointer to this task. The other lines of code also get this pointer name as an input parameter. There are two ways the convolution can be computed: directly or within the Fourier domain. The parameter VSL_CONV_MODE_AUTO tells the library to automatically choose which option to use. One can also choose to explicitly chose one of the two. Which option is chosen in automatic mode depends on the size of the input. The other input parameters are used to define the shape of the input and output. ndim defines the number of dimensions of the input arrays, which mostly equals 2 for the problems CONTACT is solving. The last three parameters provide the dimension lengths of the two input arrays and the output array. Because the number of dimensions equals 2, each of these last three parameters are vectors containing two natural numbers. After the task is created, the values of certain parameters can be set. This is only needed when the default convolution is not the one you need, but for our problem the default functionality is fine.

The second line of code gives the order to calculate the convolution. The input arrays are x and y and the output array is z. Strides can be used to determine which part of the input is to be convolved. Each stride parameter is a vector of natural numbers and its length must be equal to the number of dimensions of the arrays. When one just wants all elements of x convolved
with all elements of $y$ and the number of dimensions is 2, then the first element of each stride vector must be a 1 and the second element must be the number of columns. When this line of code is executed, the result is stored in array $z$. When this is done, the convolution task can be deleted. When one wants to use this task another time, then one simply does not remove the task in between. The variable called status is an integer that holds the error code. If the value of this variable equals zero after the calculation is done, then no errors occurred during the calculation. When it has an other value, then one should check a list of error codes to find out what is going wrong.

9.1.1 Multiple convolutions with the same influence grid

In CONTACT, the influence grids are used as input for convolutions multiple times. In each step of the iteration, the output is convolved with the same influence matrix. The Intel library has another form of the convolution functions which are made for such cases, it is called the X-form. The idea is that the user provides one input array when creating the task, named $x$. For the execution, one should just give one other input array, then this array will be convolved with $x$. One can do as much convolutions of input arrays with $x$ as one needs, as long as the task is not deleted. The library will make use of the fact that the vector $x$ is the same all the time, which makes a bit more optimization possible. The function for creating a convolution task in the X-form is vsldconvnewtaskx, so it has an extra 'x' at the rear, compared to the normal form of the function. In the same way, the execute command for the X-form is vsldconvexecx. Because these functions need other parameters than the normal forms of these functions, the X-form is shown here:

```plaintext
> status = vsldconvnewtaskx(task, VSL_CONV_MODE_AUTO,&
    ndim, SHAPE(x), SHAPE(y), SHAPE(z),&
    x, xstride)
> status = vsldconvexecx(task, y, ystride, z, zstride)
> status = vslconvdeletetask(task)
```

The first function call needs two more parameters: the input array $x$ and its stride. This parameters should be given in the same way as one would in the function vsldconvexec we saw earlier. The second function call now only needs input array $y$, output array $z$ and their strides. The principle of this piece of code is quite similar to the one of the previous code block, the only difference is that we the Intel library that it should use $x$ as one input array each time this task is executed. The library uses this extra information and therefore, calculating multiple convolutions with the X-form is a bit faster than computing the same convolutions with the normal convolution functions.

Two functions are implemented in CONTACT that are based on this last block of code. One of them contains these three lines of code, this function is called when the first convolution is calculated. The other function only contains the second line of code, this function is used for all the other convolutions that are done with the same influence grid. The reason that two different functions were made, despite the fact that they are almost the same, is that the input parameters are not exactly the same. The first function needs to return the convolution task variable and the second function needs this task variable as an input parameter.

9.2 Intel’s FFT functions

Another option is to use the FFT functions to calculate a convolution. This is more low-level because then one has to do the needed Fourier transforms and the elementwise product within
the Fourier domain manually. Here is a piece of code in which the Forward and backward transformations are done.

```c
> status = DftiCreateDescriptor(DSC_FORW, DFTI_DOUBLE, DFTI_REAL,&
   ndim, SHAPE(x))
> status = DftiSetValue(DSC_FORW, DFTI_CONJUGATE_EVEN_STORAGE,&
   DFTI_COMPLEX_COMPLEX)
> status = DftiSetValue(DSC_FORW, DFTI_PLACEMENT, DFTI_NOT_INPLACE)
> status = DftiSetValue(DSC_FORW, DFTI_OUTPUT_STRIDES, strides_out)
> status = DftiCommitDescriptor(DSC_FORW)
> status = DftiComputeForward(DSC_FORW, x_1D, f_x_1D)
> status = DftiFreeDescriptor(DSC_FORW)
>
> status = DftiCreateDescriptor(DSC_BACK, DFTI_DOUBLE, DFTI_REAL,&
   ndim, SHAPE(x))
> status = DftiSetValue(DSC_BACK, DFTI_CONJUGATE_EVEN_STORAGE,&
   DFTI_COMPLEX_COMPLEX)
> status = DftiSetValue(DSC_BACK, DFTI_PLACEMENT, DFTI_NOT_INPLACE)
> status = DftiSetValue(DSC_BACK, DFTI_INPUT_STRIDES, strides_out)
> status = DftiCommitDescriptor(DSC_BACK)
> status = DftiComputeBackward(DSC_BACK, f_u_1D, u_1D)
> status = DftiFreeDescriptor(DSC_BACK)
```

9.2.1 The forward transform

In the first half, the Fourier transform of two input arrays is calculated and in the second half, one resulting array is transformed back to the normal domain. A task has to be created to be able to compute a Fourier transform, called a descriptor here, which is done in the first line of code. The first parameter is a pointer to the task. The second parameter, DFTI_DOUBLE, defines the precision to work with. The only other option for the precision is DFTI_SINGLE, which is less precise. The third parameter states whether the input will be real valued or complex valued. We choose for real input, because the arrays CONTACT works with are all real valued. Computations on real valued input can be handled quicker than on complex valued input, so it is wise to set this option to ‘real’ whenever possible. The last parameter defines the shape of the input, which is a vector with the length along each dimension.

Lines two to four are used to alter the task to our needs. In line two the storage method of the result is defined. In line three is stated that the result should be stored in a separate array, the default is that the input array is overwritten with the output. This is necessary because the input is real valued and we need the resulting array to be complex valued in order to do an elementwise product within the Fourier domain easily. The fourth line defines the output strides. One should mention that the output array is not as large as the input array, because the result is stored in a clever way. Because the result is complex conjugated, only half of the resulting elements is stored. The strides define the length of the dimensions of the complex output array. For an explanation about strides or about Intel’s library in general, I refer to [5].

Once the descriptor is committed, the actual Fourier transforms can be calculated. This is done in code lines six and seven. The input arrays \(x\) and \(y\) are given to the function in one-dimensional form as \(x_{1D}\) and \(y_{1D}\). The second parameter of the functions are the arrays in which the result of the transformation will be stored. Note that both \(x_{1D}\) and \(y_{1D}\) have
the same number of elements, so the descriptor can handle them both. When two arrays are to be convolved within the Fourier domain, they should have the same size and shape because only then the elementwise multiplication in the Fourier domain can be done. If the two arrays are not of the same size, one should simply enlarge the smallest one with zeros until the input arrays have equal shapes.

9.2.2 The backward transform

In between the two halves of code the elementwise product of \( f_x \) and \( f_y \) is computed and stored in complex array \( f_u \). This array has to be transformed back to the normal domain. We need a new descriptor for the backward transformation because the shape of the complex array is not equal to the shape of the real input that was used for the forward transformation. The descriptor that is created in the first line of the second half of the code looks quite a lot like the descriptor for the forward domain, actually only the name of the descriptor is different because it is a different task. Note that the third parameter is \texttt{DFTI_REAL} again, despite the fact that we will give a complex array as input for the backward transform. This parameter just says that the array in the normal domain is real valued. With this information the library knows that the input array for the backward transformation is complex conjugated and the result will be real valued again. For a similar reason the storage method and the placement of the result are defined the same as in the forward transform. A difference between this descriptor and the descriptor for the forward domain is that the output strides of the forward domain are now defined to be the input strides for the backward domain. This tells the library what shape the complex array is.

After all the settings are done, the backward transformation is calculated and the result is stored in the real valued parameter \( u \). This result can be reshaped back to a 2D array. Then this should be the correct result of the convolution.

9.2.3 Multiple convolutions with the same influence grid

In CONTACT, several convolutions with the same influence grid have to be calculated. Remark that one only has to compute the Fourier transform of the influence grid once. Once it is computed, it can be stored and used the next time it is needed in a convolution. One can use the same lines of code as are given in the start of this section. There is only one difference: when \( x \) is the array representing the influence grid and \( y \) is the other input, then one can omit the line below after the first convolution is computed.

\[
> \text{status} = \text{DftiComputeForward(DSC\_FORW, } x, f_x)\]

After the forward transformation of \( y \) is done, the elementwise multiplication in the Fourier domain can be computed. Remark that the Fourier transform of \( x \) is needed for this. Therefore, one should store \( f_x \) when it is calculated the first time, and this variable should be available when calculating the other convolutions. Just as with convolution based functions, two functions are implemented in CONTACT that are based on this last block of code. One function contains the code line above, the other does not contain that line. Next to that there is a difference in input and output parameters for these two functions. The function that is called for calculating the first convolution asks for the influence grid as input parameter and it returns its Fourier transform, called \( f_x \). The second function can be used for all the other convolutions for which the same influence grid is involved. This second function asks for the Fourier transform of the influence grid as input, and the influence grid in the normal domain is not needed anymore.
9.3. INTERMEDIATE RESULTS

By calculating the Fourier transform of the influence grid only once instead of for each convolution, a speedup of around 20% can be achieved. This was the case for a test problem that uses around 5000 grid points and computes about 250 convolutions. When larger grid sizes are used, the speedup can be a bit higher, because then the Fourier transforms become the runtime bottleneck more and more within the whole convolution function. The theoretical possible speedup is limited to 33% because two Fourier transforms are calculated for each convolution instead of three, except for the very first convolution of course.

9.3 Intermediate results

Two new methods for calculating convolutions are integrated in CONTACT. One is based on the Intel convolution functions and one on the Intel FFT functions. The Intel function calls for these two methods are shown and explained in Sections 9.1 and 9.2. After the implementation and optimizations that are given in these sections, some runtime tests were performed. The size of each grid point can also affect the runtime. The size of the grid points is chosen in such a way that the total area of the whole discretization is about the same size for the different tests. The following rules are applied on the test cases: when the number of grid points in a dimension is below 100, then the length of each grid point is 0.1 in that dimension. For 128 grid points in a row or column, the length of each grid is 0.06 in that direction, and for 256 grid points each grid point has a length of 0.03.

9.3.1 Accuracy of the different methods

The results of the two new methods for calculating a convolution are quite precise. For some test cases, the output grid of the original program has been compared to the output of the methods based on FFT’s and convolution functions. Both new methods work with double precision and the result is printed in single precision. The resulting values of the output grid is exactly the same for all three methods, so the new methods compute the resulting values with a precision of at least eight decimals.

Another point to consider is the number of convolutions the methods need before the result stabilizes. The original, direct method needs 241 convolutions before the stopping condition is met. The method based on convolution functions and the FFT based method need 247 respectively 245 convolutions before the result has stabilized properly. The fact that the number of convolutions hardly differs for the different methods is a good indication that the new methods are behaving as expected, they are definitely not taking a detour before getting to the result.

9.3.2 Runtime results

The test case is about the calculation of the displacement in the normal direction of the potential contact area of two objects. A grid of 71 × 81 is used for it. But also grid sizes of 70 × 81 and 72 × 81 are tested. The reason for this will be made clear soon. Around 250 convolutions are needed before the result has converged enough to a stable solution. Some test results can be found in Table 9.1. In this table, the total runtime on spend on the calculation of the convolutions is shown for three methods. The ‘direct’ method refers to the method that was used in CONTACT already before the possibility of calculating convolutions in the Fourier domain was proposed. The method called ‘convolution’ refers to the calculation of the convolution using the convolution functions of Intel. And finally, the method called ‘fft’ is the method based on Intel’s FFT functions.

A few things become clear from this table. The first aspect is that the two new methods are faster than the old, direct method. The FFT based method is around ten times faster
than the direct method for a grid size of around 5600 grid points. This fits quite well with the test results that were performed in Matlab. In Table 6.1 one can see that for the test case with 4096 grid points, the convolution with FFT’s is around ten times faster than the direct convolution, 0.0042 seconds versus 0.038 seconds for one convolution. When one multiplies the number 0.0042 of Table 6.1 with a factor 250, one obtains a good guess for the total runtime for 250 convolutions; 0.0042 · 250 ≈ 1, so a guess for the total runtime for the convolutions would be 1 second. The runtime results in this section for the new convolution methods are of the same order of magnitude, which is a good hint that no big mistakes were made in the implementation.

A second important aspect is that the runtime for the FFT based method can change with more than a factor 2 when only a very small adjustment in the number of grid points is made. The runtime of the other two methods is hardly affected by a small variation in the number of grid points. The reason is that FFT’s perform best when the number of elements is a number with small prime divisors. A brief explanation of this aspect of FFT’s can be found in Section 3.5. Next to that, some test results on this subject can be found in Section 6.4. The question might arise how large the variation in runtime can be for small adjustments to the number of grid points. Therefore, the convolution based on the FFT functions of Intel is tested for a whole bunch of grid sizes that are quite similar. The grid sizes $70 \times 81$, $71 \times 81$ up to $85 \times 81$ are tested on runtime. The results can be found in Figure 9.1. From this graph one can conclude that the runtime for a ‘bad’ grid size can be up to 3 times as high as for a ‘good’ grid size.

The next step in the optimization process is to alter the FFT based convolution function so that it calculates FFT’s on arrays with a length and width that are powers of 2. Choosing an input grid with a length and width that are powers does not work directly. As stated earlier, the influence grid has a size of $(2^n - 1) \times (2^m - 1)$ when the input grid has size $n \times m$. When one chooses $n$ and $m$ to be powers of 2, then $(2^n - 1)$ and $(2^m - 1)$ are not powers of 2. In the FFT based function, the input grid is padded with zeros to make sure it has the same size as the influence grid. The FFT’s then all work on arrays with the same size as the influence grid. Now, for the example in Table 9.1, the FFT based function is relatively slow for a grid of size $70 \times 81$. For this grid size, the FFT works on matrices of size $139 \times 161$. Note that 139 is a prime number, so its prime divisors cannot be called small. The other two grid sizes result in a matrix with a side of length $141 = 3 \cdot 47$ or $143 = 11 \cdot 13$. Their prime divisors are not very small, but at least a bit smaller than 139, especially for the number 143. Therefore, the FFT based method is a bit faster for the largest grids in the given test case. So, one definitely should pay attention when choosing the size of a grid when FFT’s are involved.

### 9.4 The FFT based convolution for ‘good’ grid sizes

As explained in Section 6.4, it is wise to choose the length and width of the input grid to be powers of 2 and to enlarge the influence grid with one row and column so that its sizes are also
powers of 2. So the next optimization is to enlarge the input grid with one row and column and perform some test results with it. This optimization is only applied to the FFT based function and not to the method based on Intel’s convolution functions. There are two reasons for this. The first reason is that the speed of the convolution function does hardly swing, as can be seen in Table 9.1. Therefore, choosing a good grid size is of no importance for the convolution based function. The second reason is that the FFT based function already can be 4 times faster than the other new method, so it looks more promising to further optimize the FFT based method.

The FFT based function can be altered so that it makes the influence grid one row and column larger, so that its dimensions are exactly twice as long as the dimensions of the input grid. This can be done by defining a matrix of the desired size and then copy the influence grid into this new matrix. The input grid is enlarged with zeros so that it has the same size as the new influence matrix. One should pay attention though when grabbing the result out of the matrix resulting from the convolution. As a reminder: in Chapter 4 is explained that the matrix resulting from a convolution in the Fourier domain is larger than the desired output grid. In the former versions of the convolution in the Fourier domain, the resulting output grid of the proper size was to be found in the bottom right corner of the resulting matrix. Thanks to the extra row and column of zeros, the resulting grid is not exactly in the bottom right corner of the resulting matrix. When one deletes the bottom row and most right column, then the output grid is in the bottom right corner again. One should make sure the right part of the output is taken from the resulting matrix when implementing this optimization.

9.4.1 Runtime results

Now we are going to compare the former FFT based convolution method with its new form in which the FFT input matrices are enlarged a bit so that they are twice as long and width as the input grid. We expect this latest adaption to bring a further reduction in runtime when
choosing a grid size of which the length and width are powers of 2. We could also compare these
methods for other grid sizes, but it is likely that none of the two methods will be always faster
than the other. The reason is that the speed of the FFT’s can alternate rapidly, depending on
what prime divisors the size of the matrices have. Of course we do want to know whether the
enlarged version is faster than the former version of the FFT based function. Some runtime
results can be found in Table 9.2.

<table>
<thead>
<tr>
<th>grid size</th>
<th>former FFT method</th>
<th>enlarged FFT method</th>
</tr>
</thead>
<tbody>
<tr>
<td>71 × 81</td>
<td>0.70</td>
<td>0.48</td>
</tr>
<tr>
<td>64 × 128</td>
<td>1.95</td>
<td>0.52</td>
</tr>
<tr>
<td>128 × 128</td>
<td>2.25</td>
<td>1.36</td>
</tr>
<tr>
<td>256 × 128</td>
<td>10.99</td>
<td>4.33</td>
</tr>
</tbody>
</table>

Table 9.2: The runtime in seconds of the former FFT based convolution function compared to
the new version in which the matrix sizes are enlarged a bit. The length of each grid point in
each dimension is taken 0.1 when there are less than 100 grid points behind each other in that
dimension, and 0.06 for 128 grid points and 0.03 for 256 grid points.

In this table, the 'former FFT method' is the same function as 'FFT method' from Table
9.1. The runtime of 0.70 seconds for a grid size of 71 × 81 from Table 9.1 can also be found in
Table 9.2. When performing tests like this several times, the runtime is not exactly the same
every time. But the difference does not get much larger than 0.01 second, so the given runtimes
are quite reliable. The small differences can be due to the fact that the computer on which the
tests are performed can be a bit busier or less busy with other things while running the tests.
This could result in minor differences, but of course a few test cases were repeated a few more
times to exclude the possibility that different computer speeds lead to unreliable results.

From Table 9.1 it becomes clear that the enlarged FFT based method is faster than its
former version. This is at least true for grids of which the length and width are powers of 2. For
a grid size of 71 × 81 the new method is also a little bit faster, but not very much. In theory
one might be able to find a grid size for which the enlarged version is somewhat slower than
the former version. It depends on the height of the prime divisors for the length of the input
for the FFT operations. The enlarged version is an optimization especially made for input grids
with dimension sizes that are powers of 2, so for such well chosen input the enlarged FFT based
version is always faster than the former version. For other input sizes, the enlarged version is
in general faster than the former version, but not necessarily always. There is a simple reason
why the enlarged version is mostly faster than the former version: in the former version the
input arrays for the FFT’s have an odd length and in the enlarged version the arrays have an
even length. Therefore, the array length for the enlarged version has at least one 2 in its prime
factorization, which is not true for the former version. Because of this, the factorization of the
length of the input arrays for the newest method is on average lower and it has more potency if
one can choose the size of the input grid.

9.5 Practical advise for choosing the grid size

When applying numerical methods, one mostly has the option to choose the size of the input
grid. But chances are that a minimum number of grid points is needed to meet some quality
requirements. For example, one might encounter a situation in which a minimal grid size of
70 × 70 is required. This means that the input for the FFT’s have a minimal size of 140 × 140
for our enlarged FFT based method. As we saw in Section 9.4 it is wise to choose powers of 2
as dimension lengths for the FFT’s. But this would mean that the smallest grid size one could
use is 128 × 128, which results in FFT’s of size 256 × 256, which is quite a bit larger than a size of 140 × 140. We already know that FFT’s are fastest when the array length has a low prime factorization. Powers of 2 are best, but it might be wise to choose sizes of which the factorization consists of the first few prime numbers if this means that a smaller grid can be used. For the example we just introduced, we could take a grid size of 72 × 72, because the factorization of 2 · 72 is 2 · 2 · 2 · 2 · 3 · 3. These prime numbers are very low and it results in FFT’s with a size of just 144 × 144.

So which grid sizes are wise to choose in general, next to sizes that are powers of 2? To find this out, the grid sizes 64 × 64, 65 × 65 . . . 128 × 128 were tested on runtime, while other options such as the size of each grid point are kept constant. The results can be found in Figure 9.2.

From Figure 9.2 it becomes clear that choosing prime numbers as the dimension sizes of the grid gives the worst results. Sizes that have prime divisors larger than 19 are in general slower than sizes with only prime divisors below 19. The optimal line connects all the grid sizes that can be the best choice when some minimal grid size is needed. Given a minimal grid size, go to the corresponding point on the optimal line and then follow the line to the right until one finds the next point on the line. Then, that grid size is the optimal grid size for that specific problem. So from left to right, the optimal line goes through the next lowest point each time. The optimal line does not cross any sizes that have prime divisors larger than 19. If one calculates the prime divisors of the numbers that lie on the optimal line, then it becomes clear that most of these numbers have only 2, 3 and 5 in their factorization. There are also a few optimal numbers that contain some prime divisors 7, 11 and 13. There are just two cases in which 17 or 19 is a prime

![Figure 9.2: The runtime in seconds of a testproblem for the optimal FFT based method for grid sizes 64 × 64 up to 128 × 128. The size of the grid points is taken constant on 0.1 × 0.1 throughout the whole test.](image)
divisor: $102 = 2 \times 3 \times 17$ and $112 = 2 \times 3 \times 19$. Next to this relatively large prime divisors, the other primes are very low.

So if one has to choose a grid size as small as allowed, then a good advice is to pick a size of which the prime factorization contains no numbers larger than 19. For this test case, given a minimal required grid size, the optimal grid size is the smallest allowed number from the list: 64, 65, 66, 72, 75, 80, 81, 84, 88, 90, 96, 100, 102, 108, 112, 114, 120, 128. Of course it could happen that for sizes below $64 \times 64$ or above $128 \times 128$ there is a good grid size with a prime divisor that is a bit higher than 19, but only using sizes with prime divisors below 19 gives a runtime that is hardly larger than the the runtime for the optimal choice. In case such a chosen grid size is not optimal, it cannot be more than a few percent slower than the optimal grid size. A more simple way with quite a good result is to choose grid sizes of which the prime factorization only contain the first few prime numbers, for example only 2, 3, 5 and 7. Figure 9.3 shows a part of the data from Figure 9.2, but now the numbers with a factorization with no prime numbers higher than 7 are accented.

![Figure 9.3](image-url)

Figure 9.3: The runtime in seconds of a test problem for the optimal FFT based method for the optimal grid sizes between $64 \times 64$ up to $128 \times 128$. The size of the grid points is taken constant throughout the whole test. The sizes without prime divisors larger than 7 are accented.

As one can see, most of the numbers on the optimal line have no prime divisors larger than 7. The ones that have higher prime divisors are mostly followed by an other grid size without prime divisors larger than 7 that is hardly any slower. The numbers with prime divisors larger than 7 lie relatively more to the top left than the other numbers, which means that their proportion runtime / number of grid points is a bit worse. In this test there are two grid sizes with a prime divisor larger than 7 that are faster than the next size without prime divisors larger than 7: size $65 \times 65$ and $114 \times 114$. If one disregards the option $65 \times 65$ and chooses $72 \times 72$ instead,
then the runtime is \((0.24/0.22 - 1) \times 100 \approx 9\%\) larger. If one chooses 120 \times 120 instead of 114 \times 114, the runtime is \((0.95/0.93 - 1) \times 100 \approx 2\%\) larger. The fact that the grid size 72 \times 72 is almost 10\% slower than size 65 \times 65 can be caused by the relatively large difference between the number 72 and 65. A grid of 72 \times 72 contains about 22\% more grid points than a size of 65 \times 65. Using the first grid size larger than a certain minimum size, which factorization only contains prime numbers 2, 3, 5 and 7, already gives a good size which is in practice not more than 10\% slower than the best gridsize one could choose. If one can find a good grid size where the length of the dimensions is only a few percent larger than the minimal required size, then the computation will be not more than a few percent slower.

## 9.6 Runtime conclusions

Test results for the runtime of the old implementation have been compared to different versions of new methods already. In order to get a good insight in the speed of the different methods, a table of runtime results is given now for grids with powers of 2 for the number of rows and columns.

<table>
<thead>
<tr>
<th>grid size</th>
<th>grid size</th>
<th>grid size</th>
<th>grid size</th>
<th>grid size</th>
<th>grid size</th>
</tr>
</thead>
<tbody>
<tr>
<td>old, direct method</td>
<td>0.0011</td>
<td>0.013</td>
<td>0.209</td>
<td>3.35</td>
<td>77.35</td>
</tr>
<tr>
<td>convolution method</td>
<td>0.0050</td>
<td>0.017</td>
<td>0.119</td>
<td>1.27</td>
<td>12.84</td>
</tr>
<tr>
<td>former FFT method</td>
<td>0.0024</td>
<td>0.007</td>
<td>0.034</td>
<td>0.77</td>
<td>2.25</td>
</tr>
<tr>
<td>optimal FFT method</td>
<td>0.0016</td>
<td>0.004</td>
<td>0.017</td>
<td>0.14</td>
<td>1.36</td>
</tr>
</tbody>
</table>

Table 9.3: The runtime in seconds for the old method, the method based on convolution functions and the former and optimal version of the FFT based method. The length of each grid point in each dimension is taken 0.03 for 256 grid points, 0.06 for 128 grid points, 0.1 for 64 grid points and twice as large for each halving of the grid points.

### 9.6.1 Optimal new method versus old method

The most important aspect to notice is that the optimal version of the FFT based method is the fastest method to calculate series of convolutions. The only exception is that the old method is a bit faster for a grid of 8 \times 8. From this table we can safely conclude that the optimal FFT method is always faster than the old method when the number of grid points exceeds one hundred. For a grid with about 1000 grid points the optimal method is about a factor 10 faster than the old method. For a grid with about 4000 grid points the speedup factor is already more than a factor 20. For a grid with about 32000 grid points the optimal new method is about 90 times faster than the old method. This is true for grid sizes of which the dimension lengths are powers of two. For grid sizes that have numbers with only low primes in their factorization, the corresponding size is a bit slower, but in practice their runtimes will be seldom more than 10\% higher compared to taking a power of 2 for the number of grid points. So for such grid sizes the runtime is almost as good, and definitely way better than the old, direct convolution.

### 9.6.2 The real speedup in CONTACT

As explained in the previous section, the new method for calculating convolutions is between equally fast and about a hundred times faster for the tested grid sizes. Within CONTACT,
there is a lot of code around the convolutions. The convolutions are the bottleneck, but thanks to the new method these convolutions are computed a lot faster. Consequently, the runtime code around the convolutions is of more significance. The function in which all the convolutions are computed takes a bit longer than the actual computation of the convolutions, this function is called NormCG. This function plus some more code lies within one function for calculating the displacement of the contact area in the normal direction. Now, we compare the runtime of function NormCG with the total runtime of the test case that was used throughout this chapter. The total runtime for the function NormCG and the total runtime of the normal displacement are shown in Table 9.4 for a few different grid sizes.

<table>
<thead>
<tr>
<th>Grid Size</th>
<th>Norm</th>
<th>Total</th>
<th>Norm</th>
<th>Total</th>
<th>Norm</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 × 16 grid</td>
<td>0.014</td>
<td>0.018</td>
<td>0.222</td>
<td>0.246</td>
<td>3.54</td>
<td>3.73</td>
</tr>
<tr>
<td>32 × 32 grid</td>
<td>0.025</td>
<td>0.049</td>
<td>0.025</td>
<td>0.049</td>
<td>0.22</td>
<td>0.42</td>
</tr>
<tr>
<td>64 × 64 grid</td>
<td>0.22</td>
<td>0.42</td>
<td>80.66</td>
<td>83.08</td>
<td></td>
<td></td>
</tr>
<tr>
<td>128 × 128 grid</td>
<td>2.39</td>
<td>4.82</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 9.4: The runtime in seconds for the NormCG function, abbreviated with 'norm' and for the total normal problem for a test case where only the displacement in the normal direction of the contact area is calculated. The old method and the optimal new method are compared for a few different grid sizes.

From Table 9.4 one can see how much time the code around the function NormCG takes. For a grid of 64 × 64, the total runtime is about 0.2 seconds larger than the time for NormCG and for a grid of 128 × 128 the total runtime is about 2.4 seconds larger. This extra runtime is not really significant for the old method, but for the new method this extra runtime is about half of the total runtime. Therefore, the total normal problem in CONTACT in is practice about 10 times faster with the new method compared to the old method for a grid of 64 × 64 and close to 20 times faster for a grid of 128 × 128. For a grid of 256 × 128, which is not shown in the table, the speedup is around a factor 30. If one compares the total runtime for the new FFT based method with the results from Table 9.3, then one can see that the total runtimes for the new method in Table 9.4 are about a factor 3 larger. The difference in runtime for the old method in these two tables is relatively small because for the old method the convolutions are totally the bottleneck of the whole problem. So we can conclude that the speedup for the total normal problem within CONTACT is about 3 times less than the obtained speedup of the convolution functions.

Figure 9.4 shows a graph of the speedup factor for different grid sizes, so that one can get a good idea of the real speedup of the normal displacement problem of CONTACT. In Section 4.4 we concluded that the old and new method should be about equally fast for a grid with about 240 grid points. As the graph makes clear, the old and new method are in reality equally fast when the total number of grid points lies close to one hundred. The fact that this number is lower than we expected theoretically, can be explained. First of all, the theoretical runtime formulas were guesses. Next to that, some optimizations were implemented in the new method. And the real speed of the new method depends on the speed of the Intel functions.

9.6.3 The runtime order of the old and new methods

One might ask the question whether the runtime order of the old method and the new method correspond to the theoretical orders we found. Remember that for a discretization with $n^2$ points, the order of the old method is $n^4$ and all the new methods should have order $O(n^2 \log n)$. So when the number of grid points increases with a factor 2, then we expect the old method
9.6. RUNTIME CONCLUSIONS

![Figure 9.4: The speedup factor of the runtime for the optimal FFT based method compared to the runtime for the old, direct method for different number of grid points.](image)

...to become about 4 times slower and the new methods to become about $2 \log \sqrt{2} = 2.5$ times slower.

However, there is another aspect that influences the runtime results. The total number of iteration steps increases if one increases the number of grid points, so more convolutions have to be calculated when taking a larger grid. The number of convolutions is a little above 200 for a grid size of $64 \times 64$ and just below 500 for a grid size of $256 \times 128$. When checking the order of the methods, in which this property is not incorporated, we should compensate for this aspect. We can do this by dividing the runtimes by the number of convolutions that are performed.

When we take a look at Table 9.3, we can check the order of runtime increase on sizes $64 \times 64$, $64 \times 128$, $128 \times 128$ and $256 \times 128$, because each next size has twice as much grid points as the previous one. We compensate for the number of convolutions by dividing the runtimes by the number of convolutions and then the runtime for the largest three grid sizes are divided by the runtimes for a grid size with half the number of points. This results in three slowdown factors for each method. The slowdown factors of the old method and the optimal new method are shown in Table 9.5.

As this table makes clear, the old method gets about a factor 4 slower when doubling the total number of grid points in the discretization. The given slowdown factors are not very constant. One should keep in mind that the runtime for different grid sizes can be influenced more or less significantly by aspects like (excessive) memory usage, usage of the cache or parameter passing etc. The average of the three given slowdown factors is 3.8, the closest natural number is 4. This fits quite well in the theoretical expectation that the slowdown should be about a factor 4 when doubling the number of grid points.

For the optimal FFT method the slowdown factors are just a bit larger than 2. the slowdown values lie quite close to the expected factor 2.5. The slowdown factors for the convolution based method are quite similar to the ones of the optimal FFT method, it clearly is of the same order.
The slowdown factors of the intermediate FFT based method are a bit more swingy. This is caused by the factorizations of the length for the FFT’s. For example: for a grid of $64 \times 64$ the intermediate method calculates FFT’s of size $127 \times 127$ and then this method is relatively slow because $127$ is a prime number. For a grid size of $128 \times 128$ the size of the FFT’s is $255 \times 255$. The number $255$ has factorization $3 \cdot 5 \cdot 17$, so thanks to these relatively low prime numbers the method is relatively fast. Therefore, the slowdowns from grid size $64 \times 64$ up to $128 \times 128$ are relatively low. The prime factorization of the number $2 \cdot 256 - 1 = 511$ contains a bit higher prime number, therefore the last of the three slowdown factors for this method is relatively large. The average of the three slowdown factors is $2.1$, which is again just a bit above a factor $2$.

So finally, we can conclude that the runtime results of Table 9.3 do fit quite well to the theoretical orders we gave earlier, they are close enough to conclude that the given runtime orders are correct.

<table>
<thead>
<tr>
<th>grid size from</th>
<th>grid size from</th>
<th>grid size from</th>
</tr>
</thead>
<tbody>
<tr>
<td>$64 \times 64$ to $64 \times 128$</td>
<td>$64 \times 128$ to $128 \times 128$</td>
<td>$128 \times 128$ to $256 \times 128$</td>
</tr>
<tr>
<td>old, direct method</td>
<td>5.0</td>
<td>2.7</td>
</tr>
<tr>
<td>convolution method</td>
<td>2.2</td>
<td>2.4</td>
</tr>
<tr>
<td>intermediate FFT method</td>
<td>1.7</td>
<td>1.0</td>
</tr>
<tr>
<td>optimal FFT method</td>
<td>2.4</td>
<td>2.2</td>
</tr>
</tbody>
</table>

Table 9.5: The slowdown factor of the runtime for the old method, the convolution method and the versions of the FFT based method when doubling the number of grid points a few times.
Chapter 10

Conclusions

Calculations that are used in problems regarding contact mechanics often include the computation of a lot of convolutions. This is at least true for the software CONTACT that the Dutch company VORtech uses. It contains some influence grids which form is determined by some real world properties. Such influence grids are convolved multiple times with input grids that represent the deformation in the contact area where two objects touch each other. These computation of such convolutions is quite expensive, when an input grid contains \( n^2 \) grid points, then a convolution with an influence grid takes about \( n^4 \) multiplications and additions. This article made clear that convolutions can be calculated faster if one computes them within the Fourier domain, the runtime is then \( O(n^2 \log n) \). When using the Fourier domain, the bottleneck of the computation is the transformation to the Fourier domain and the reverse transformation back to the normal domain. This would normally be an \( O(n^4) \) operation for general problem as just described, but thanks to fast Fourier transforms, also known as FFT’s, these transforms can be done in just \( O(n^2 \log n) \). This is the main idea of the method that is used to speed up the convolutions of the CONTACT software.

10.1 Main results

The method of computing convolutions within the Fourier domain is implemented in CONTACT. The Intel Math Kernel Library is used for this. The speedup of the new method compared to the old method depends on the number of grid points in the discretization of the contact area. The two methods are about equally fast for a discretization with about 100 grid points. For grids with less grid points the old method is faster than the new one, but for larger grids the new method is faster. For a discretization with around 4000 grid points, for example a grid of size 64 × 64, the new method is about 20 times faster than the old, direct method. When the number of grid points increases, the speedup is larger. When the number of grid points lies around 32000, for example a grid of size 256 × 128, the speedup is about a factor 90. But within the CONTACT code for computing the normal displacement, there is quite some extra code around the convolution functions. The costs of this extra runtime is of high significance for the new method. The convolutions are optimized so far, that the time needed for their computation is only one third of the total runtime instead of about 95%. Therefore, the real speedup of the whole normal problem within CONTACT is about a factor 3 smaller than the speedups of the new convolution method we just gave.
10.2 Forget about Barrowes’ method

Barrowes’ method provides an algorithm to speed up the computation of a matrix vector product when the matrix is a (block-)Toeplitz matrix. An original idea of VORtech was to write convolutions as a matrix vector product, which results in block-Toeplitz matrices and vectors, and then use Barrowes’ method. Then, the matrix and vector for such a convolution are relatively large compared to the original grids because there is redundant data. Barrowes’ then provides a trick that would result in relatively small vectors that are to be convolved within the Fourier domain. But it turned out that these small vectors are just a 1D representation of the grids we started with. So Barrowes’ method is useless here, because first the problem is written in a larger form and then the original problem is extracted from it. Therefore, we can forget about Barrowes’ method for now. What we learned from Barrowes’ method is that it actually provides the transformation from a block-Toeplitz matrix vector product (back) to a convolution.

10.3 Important aspects for an optimal implementation of FFT’s

When implementing Fourier transformation functions, one has a lot of options and possibilities to choose from. For a fast implementation, several aspects are important. The most important aspects are given now. First some aspects concerning optimizations for computing several problems with several convolutions are treated, and then some aspects concerning the optimization of every single convolution.

10.3.1 Solve independent problems independently

Problems that are independent should be solved independently. So one should not combine several convolutions when it is not necessary. The reason is that computing convolutions in the Fourier domain is $O(n^2 \log n)$ where $n^2$ is the total number of values. If one merged two convolutions, then $n^2$ is two times larger, so the total runtime increases with order $2 \log \sqrt{2}$. When the two independent problems are solved separately, then the total runtime is just two times larger than the time needed for solving one problem.

10.3.2 Use redundancy when computing multiple convolutions

In the CONTACT software a certain influence grid has to be convolved with an input grid a few hundred times. The input grid changes with each convolution, but the influence grid stays the same throughout the process. Therefore, once the Fourier transform of an influence grid is computed, one should store its Fourier transform and use it for all the other convolutions. This saves close to one third of the number of FFT’s that have to be computed. Of course one should keep in mind that increased memory usage and parameter passing for the Fourier transform of the influence grid can affect the speedup negatively a bit.

10.3.3 Use redundancy in the Fourier transformed grids

When a real valued grid is transformed to the Fourier domain, it results in a complex matrix with the same size. Complex numbers can contain more data than real numbers because a complex number is actually built up from two real numbers. The Fourier transform of a real valued grid always results in a complex matrix that is complex conjugated. Therefore, one only has to know one half of the matrix in order to determine all its values. This property can be used when calculating an elementwise product of two grids in the Fourier domain. The Intel functions make use of this property because only half of the elements of the Fourier transformed
grids is stored. More ways of using properties of the Fourier transforms of several different influence matrices are mentioned in Chapter 8. But these other possible optimizations are of small importance because they can only be used for a small number of computations. Next to that, they are likely to speed up elementwise products in the Fourier domain, but the FFT’s themselves are the bottleneck of a convolution that is done within the Fourier domain.

The Intel library contains functions that one can use to implement the needed Fourier transforms and elementwise products, but also high level functions exists that can calculate a convolution with only a few lines of code. For our problems, implementing the convolution ourselves results in a faster method compared to using high level functions. The main reason is that it enables us to make use of all the aspects of our problems so that the convolution is optimized for the contact problems VORtech wants to solve. When trying to create your own method for calculating convolutions using the Intel library, it is wise to compare implementations with high level and lower level functions.

### 10.3.4 Equal grid sizes are necessary

A convolution of two grids can only be done in the Fourier domain when both grids have the same size. This is due to the fact that a convolution is an elementwise multiplication in the Fourier domain. Elementwise products can only be executed on matrices with equal size. When the two grids do not have equal sizes, one should enlarge the smallest one by adding zeros before transforming the grids to the Fourier domain. Within CONTACT the influence grid is larger than the input grid, so the input grid should be enlarged so that it has the same size as the influence grid.

### 10.3.5 Enlarge the input for the FFT’s a little bit

For a grid size $m_x \times m_y$ of the input grid, the corresponding influence grid has size $(2m_x - 1) \times (2m_y - 1)$. Then, the size of the convolutions is at least $(2m_x - 1) \times (2m_y - 1)$. But the Fourier transforms are in general faster when an extra row and column of zeros is added, so that the size on which the FFT’s operate becomes $2m_x \times 2m_y$. This is due to the fact that FFT’s are fastest when the number of elements is a number with only low prime numbers in its factorization. When the size of the matrices is enlarged to $2m_x \times 2m_y$, then the numbers $2m_x$ and $2m_y$ already have the lowest prime number in their factorizations, which is not the case for $(2m_x - 1)$ and $(2m_y - 1)$ because these numbers are odd. Therefore, one should enlarge the grids that are to be convolved with one row and column of zeros before transforming them to the Fourier domain.

### 10.3.6 Choose the discretization wisely

When the previous aspect is implemented, one should choose the number of rows and columns in the discretization of the contact area wisely. The length of both dimensions should be numbers of which the factorization contains only low prime numbers. Numbers with primes larger than 19 in their factorization should be avoided. This prime number 19 is not a hard border between good and bad choices for the number of rows and columns, it is just the result of a test case for grid sizes between $64 \times 64$ and $128 \times 128$. This test case results in the following advice: when a certain minimal grid size between $64 \times 64$ and $128 \times 128$ is required, one should pick the smallest allowed number of rows and columns from the list $64, 65, 66, 72, 75, 80, 81, 84, 88, 90, 96, 100, 102, 108, 112, 114, 120, 128$. All the numbers in this list have no prime numbers larger than 19 in their factorizations. There are also sizes with very low prime numbers that are not optimal, for example 126 which has factorization $2 \cdot 3 \cdot 3 \cdot 7$. The reason is that the number 128
is hardly any larger, but this is more optimal because 128 is a power of 2. So numbers built from low prime factors, say lower than 19, can be an optimal choice but they are not always the best choice. Therefore, it is hard to give a simple formula to find the best choice for the grid size. As a rule of thumb, a good compromise is to choose the smallest allowed grid size with a factorization containing no primes larger than 3, or one could take 5 or 7 as an upper boundary instead. This results in a choice for the grid size that results in a calculation that will practically not be more than 10% slower than the optimal choice. However, if one is totally free in the choice of the grid size, then one should pick dimension sizes that are powers of 2, for example a grid of $64 \times 128$.

10.4 Recommendations

A significant speedup is achieved already by implementing the Fourier based convolutions into the CONTACT software. But this new method is mainly useful for calculating the normal displacement of a contact area. The total displacement problem contains some other kinds of calculations that might be optimized further as well. Another way of improving the speed of the CONTACT software is to use the graphics processing unit, or GPU, of a computer instead of just the central processing unit. The GPU is highly optimized for parallel computing which can be useful when dealing with large operations like convolutions. For more information on this topic I refer to [1].
Bibliography


[6] URL: http://cnx.org/content/m12060/latest/