Coordinate system transformations - calibration with examples

Leverić, Ivan

UNIVERSITY OF ZAGREB

FACULTY OF SCIENCE

DEPARTMENT OF MATHEMATICS

Ivan Leverić

COORDINATE SYSTEM TRANSFORMATIONS - CALIBRATION WITH EXAMPLES

Master thesis

Advisor:
prof.dr.sc Zlatko Drmač

Zagreb, July, 2022

Ovaj diplomski rad obranjen je dana _____ pred ispitnim povjerenstvom u sastavu:

1. _____, predsjednik
2. _____, član
3. _____, član

Povjerenstvo je rad ocijenilo ocjenom _____.

Potpisi članova povjerenstva:

1. _____
2. _____
3. _____

This work is dedicated to my parents and family without whose constant support and care this thesis paper was not possible. At the same time, my thanks also go to my team at the ASML Netherlands BV for guiding me through this project. Special thanks go to my supervisor Davide Lena who was there to help me whenever I needed it.

Contents

Introduction

Differentiation is one of the fundamental problems in numerical mathematics. The solution of many optimization problems and other applications require knowledge of the gradient, the Jacobian matrix or the Hessian matrix of a function. It often happens that this process has to be done repeatedly for a sequence of streaming input data and on a large-scale, making the computation of derivatives a central part of the solution. This is why it is necessary to have an accurate and efficient derivative computation for the practical solutions of different industrial problems.
One of the applications where these solutions are needed is photolithography - a technique that uses light to produce extremely small (nanometer) patterns of suitable materials over a substrate. Most commonly, it is used for fabricating integrated circuits (electronic chips) such as solid-state memories and microprocessors. What happens is the following: a system uses ultraviolet light and transfers it from an optical image called reticle through the number of lenses and projects it on a thin slice of semiconductor - usually a silicon wafer. This wafer is coated by a light-sensitive material, which causes it to change when exposed to light, thus producing a desired pattern on a wafer. To project this patterns (image), a number of coordinate system transformations are associated with the above mentioned hardware parts. This way, the machine knows where the desired point in the upper lever will end up in the lower levels of the machine. Of course, since we are talking about extremely small patterns (nanometer precision), while loading and positioning all the stages and instruments before exposing an image, the errors are unavoidable. These can have an enormous impact on a resulting product, since any mispositioning of the instruments will inevitably result in defects in the exposed image. Thus, correcting for that error is of great importance. It will turn out that computing the Jacobian matrix of a given coordinate system transformation is the solution of this problem. Therefore, the goal of this work is to investigate different methods for computing derivatives and test their performance in the context of coordinate system transformations.
Since a derivative calculation is a topic that is thoroughly researched, there are a lot of methods that we can choose from. It will turn out that there are two types of methods that will require a deeper investigation - numerical methods, including both finite step and complex step approximation, as well as automatic differentiation, hereafter abbreviated

as ”AD”. The two classes of methods have their share of differences. The most glaring one being the technical effort needed to implement the methods. On one hand, numerical methods are usually extremely simple, needing a few lines of code to compute the entire Jacobian matrix. On the other hand, AD is the one where all the complicated comes in. As we are about to see, there are two different modes of AD - forward and reverse, with the latter one being especially challenging. Of course, there is a tradeoff : accuracy wise, AD performs as perfect as one can ask for. It computes the derivative that is exact to roudoff, while numerical methods incur an truncation error in the result. How big is the error is also one of the things we aim to find out.
When testing these methods, two conditions will be required - accuracy and time-wise efficiency. As we are about to see, all methods will produce excellent results, which will make the decision on which one to choose even tougher. It will come down to choosing between two things: sacrificing a bit of time and accuracy for a simple, effortless implementation, or making quite an effort to gain a few milliseconds of execution time.
The thesis is organized as follows. In Chapter 1 we state the mathematical facts and definitions used throughout the work. The chapter is divided into four parts. The first one gives an introduction into coordinate system transformations. The second one gives a review of calculus with an emphasis on differentiation. Third, we review problems that arise when dealing with systems of linear equations using a computer and lastly, the software and implementation methods used in this work are described.
In Chapter 2 we present the problem and different solutions to it. Here we discuss different approaches on how to compute the Jacobian matrix using a computer, and also how and why those methods are implemented. Special attention is given to numerical and automatic differentiation, since it will be shown that those fit the problem best.
In Chapter 3 we present the results obtained on a real application data. The focus is on the accuracy and speed of execution, as we compare different methods. After collecting and visualizing the data, the final conclusions are drawn.
Many thanks to ASML Netherlands BV for giving me the opportunity of doing the Internship and writing this thesis about my work there. Special thanks to all my colleagues for giving me assistance and providing me with everything needed to make this possible. Their guidance and advice is much appreciated and I couldn’t be more proud to be a part of the team.

Chapter 1

Mathematical preliminaries

In this chapter the most important mathematical definitions and theorems used in this work will be reviewed. It is divided in three sections. In the first one, the basic types of transformation matrices will be introduced. Those will be used for coordinate transformation experiments used later. After that, we review the basic notions from the calculus. More specifically, the basic definition of the derivative and the Jacobian matrix, as well as their properties and most important features. Lastly, we will briefly introduce the software environment in which the numerical methods will be implemented. To be more precise, the software used throughout the work will be MATLAB, whose object-oriented nature will be of extreme use.

1.1 Coordinate systems and transformations

In this section we review the basic notions from linear algebra - matrix representation of linear operators, affine spaces and transformations. We will also take a closer look at some basic transformations - translation, rotation and magnification - that will be crucial later. The material here is taken from [6].

Definition 1.1.1. ***(Vector space)***
Let V be a nonempty set of objects called vectors, on which two operations are defined - addition and multiplication by scalars from a field $\mathbb{F}$:

$$+ \colon V \times V \longrightarrow V$$

$$\cdot \colon \mathbb{F} \times V \longrightarrow V$$

We say that V is a vector space over the field $\mathbb{F}$ if:

i) $\forall a, b, c \in V \colon (a + b) + c = a + (b + c)$.

ii) $\exists 0 \in V : a + 0 = 0 + a = a, \forall a \in V.$

iii) $\forall a \in V, \exists -a \in V : a + (-a) = -a + a = 0.$

iv) $\forall a, b \in V : a + b = b + a.$

v) $\forall a \in V, \forall \alpha, \beta \in \mathbb{F} : \alpha(\beta a) = (\alpha\beta)a.$

vi) $\forall a \in V : 1 \cdot a = a.$

vii) $\forall a \in V, \forall \alpha, \beta \in \mathbb{F} : (\alpha + \beta)a = \alpha a + \beta a.$

viii) $\forall a, b \in V, \forall \alpha \in \mathbb{F} : \alpha(a + b) = \alpha a + \alpha b.$

Example 1.1.2. *Let $n \in \mathbb{N}$. The set $\mathbb{R}^n$ is a real vector space where the vectors are n-tuples of real numbers and the operations are defined as follows:*

$$(x_1, \ldots, x_n) + (y_1, \ldots, y_n) = (x_1 + y_1, \ldots, x_n + y_n),$$
$$\alpha(x_1, \ldots, x_n) = (\alpha x_1, \ldots, \alpha x_n),$$

where $\alpha \in \mathbb{R}$, $(x_1, \ldots, x_n), (y_1, \ldots, y_n) \in \mathbb{R}^n$. The defining vector space properties are easily checked.

Definition 1.1.3. ***(Transformation matrix)***
Let $T : \mathbb{R}^n \longrightarrow \mathbb{R}^m$ be a linear transformation and $x \in \mathbb{R}^n$ an arbitrary column vector. There exists the matrix $\boldsymbol{A} \in M_{m,n}(\mathbb{R})$ such that

$$T(x) = \boldsymbol{A}x.$$

The matrix $\boldsymbol{A}$ (that is uniquely determined by T) is called the transformation matrix of T.

The transformation matrix alters the given coordinate system and maps the coordinates of the vector to the new coordinates. It can be also taken as the transformation of space. Most examples here will be three-dimensional - thus using 3×3 transformation matrices. There are multiple types of transformation matrices that are frequently used in all kinds of applications, for example stretching, squeezing, rotation etc. In this work we will mostly concentrate (with few exceptions) on so-called affine transformations. Examples of these transformations are visualized in Figure 1.1.

Definition 1.1.4. *Let $\mathcal{A} \neq \emptyset$ be a set and V a vector space over a field $\mathbb{F}$. Furthermore, let $v : \mathcal{A} \times \mathcal{A} \longrightarrow V$ be a mapping with properties:*

i) $\forall X \in \mathcal{A}, x \in V, \exists! Y \in \mathcal{A} : v(X, Y) = x.$

ii) $\forall X, Y, Z \in \mathcal{A} : v(X, Y) + v(Y, Z) = v(X, Z).$

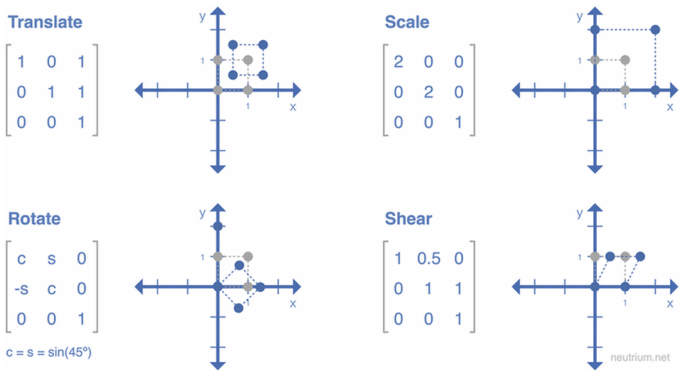

Figure 1.1: 2D affine transformations

Then we say that $(\mathcal{A}, V, v)$ is an affine space over V.

Example 1.1.5. *Using the notations from the Definition 1.1.4, we set $\mathcal{A} = \mathbb{R}$ (as a point set) and $V = \mathbb{R}^1$ (as a vector space). We can define $v(x, y) = y - x$. Similarly, let $\mathcal{A} = \mathbb{R}^n$, $V = \mathbb{R}^n$ and define*

$$v((x_1, x_2, \ldots, x_n), (y_1, y_2, \ldots, y_n)) = (y_1 - x_1, y_2 - x_2, \ldots, y_n - x_n).$$

These are the examples of affine spaces where $d(x, y)$ is the vector in $\mathbb{R}^n$ from point x to point y.

Definition 1.1.6. *Let $\mathcal{A}, \mathcal{B}$ be affine spaces and $f : \mathcal{A} \longrightarrow \mathcal{B}$. f is an affine transformation if it determines a linear map ϕ such that, for any pair of points $P, Q \in \mathcal{A}$*

$$\overrightarrow{f(P)f(Q)} = \phi(\overrightarrow{PQ}).$$

It is not hard to show the following:

Proposition 1.1.7. *An affine transformation of $\mathbb{R}^n$ is a map $f : \mathbb{R}^n \longrightarrow \mathbb{R}^n$ of the form*

$$f(p) = \boldsymbol{A}p + q$$

for all $p \in \mathbb{R}^n$, where $\boldsymbol{A}$ is a linear transformation of $\mathbb{R}^n$ and $q \in \mathbb{R}^n$. If $\det A > 0$, the transformation is orientation-preserving. Otherwise, it is orientation-reversing.

The most important affine transformations are rotations, magnifications and translations. In fact, all affine transformations can be expressed as combinations of those three. Let us briefly introduce all of them.

1.1.1 Translation

Definition 1.1.8. *Let $v \in V$ be a fixed vector. The translation $T\colon V \longrightarrow V$ is defined as $T(p) = p + v$.*

Remark 1.1.9. *If T is a translation and $\mathcal{A} \subseteq V$, then $T(\mathcal{A})$ is the translate of $\mathcal{A}$ by T. This is often written as $\mathcal{A} + v$, $v \in V$.*

Example 1.1.10. ***(Matrix representation)*** *Let $v = \begin{pmatrix} v_x & v_y & v_z \end{pmatrix}^T \in \mathbb{R}^3$ be a fixed vector. We wish to translate a vector $p = \begin{pmatrix} p_x & p_y & p_z \end{pmatrix}^T \in \mathbb{R}^3$ by a vector v. Since there is no way to represent a translation using 3×3 matrix, we have to come up with something different. Luckily, there is a workaround. Write the 3 dimensional vector p using 4 coordinates: $\bar{v} = \begin{pmatrix} v_x & v_y & v_z & 1 \end{pmatrix}^T$ and $\bar{p} = \begin{pmatrix} p_x & p_y & p_z & 1 \end{pmatrix}^T$. To translate an object by a vector $\bar{v}$, each vector $\bar{p}$ can be multiplied by this translation matrix:*

$$A = \begin{bmatrix} 1 & 0 & 0 & v_x \\ 0 & 1 & 0 & v_y \\ 0 & 0 & 1 & v_z \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

The corresponding transformation is given by:

$$T(\bar{p}) = \begin{bmatrix} 1 & 0 & 0 & v_x \\ 0 & 1 & 0 & v_y \\ 0 & 0 & 1 & v_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix} = \bar{p} + \bar{v}.$$

Our result in 3D space is now the first three coordinates of the obtained trasformation $T(p)$.

Proposition 1.1.11. *Let T_v be a translation by vector v. Then it holds that:*

i) $T_v^{-1} = T_{-v}$.

ii) $T_v T_w = T_{v+w}$.

1.1.2 Rotation

A rotation is a type of transformation that takes each point in a figure or a set and rotates it a certain number of degrees around a given point called the *origin*. Rotation matrix is a transformation matrix that is used to perform a rotation. A basic (elemental) rotation

is a rotation about one of the axes of a coordinate system. The following are the 3 basic rotation matrices that rotate vectors by an angle θ about the x, y, z axes respectively

$$\boldsymbol{R}_x(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix} \tag{1.1}$$

$$\boldsymbol{R}_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix} \tag{1.2}$$

$$\boldsymbol{R}_x(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{1.3}$$

In general, all other rotation matrices can be obtained from these three using matrix multiplication. More on rotations and rotation matrices can be found in [4].

1.1.3 Magnification

Definition 1.1.12. *Let* $v = \begin{pmatrix} v_x & v_y & v_z \end{pmatrix}^T \in V$. *Magnification is a transformation* $M_v \colon V \longrightarrow V$ *such that*

$$M_v p = \begin{bmatrix} v_x & 0 & 0 \\ 0 & v_y & 0 \\ 0 & 0 & v_z \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ p_z \end{bmatrix} = \begin{bmatrix} v_x p_x \\ v_y p_y \\ v_z p_z \end{bmatrix}, \quad p = \begin{pmatrix} p_x & p_y & p_z \end{pmatrix}^T \in V$$

1.2 Calculus

In this section we review the basic derivative rules of calculus of functions with several variables. More precisely, we will define a derivative for the multivariate functions along with its important properties. One thing to look for here is the introduction of the Jacobian matrix, since it will be extensively used throughout the work. The definitions and theorems here are taken from [16], [8].

Definition 1.2.1. *Let* V *be a vector space over the field* $\mathbb{F}$. *Norm is a mapping* $\|\cdot\| \colon V \longrightarrow \mathbb{R}$ *with the following properties:*

i) $\forall x \in V, \|x\| \geq 0$.

ii) $\|x\| = 0 \Leftrightarrow x = 0$.

iii) $\forall \alpha \in \mathbb{F}, \forall x \in V, \|\alpha x\| = |\alpha| \|x\|$.

iv) $\forall x, y \in V, \|x + y\| \leq \|x\| + \|y\|$.

An ordered pair $(V, \|\cdot\|)$ *is called normed space.*

Example 1.2.2. ***(Norms in a vector space*** $\mathbb{F}^n$***)***

i) $(\mathbb{F}, |\cdot|)$ *- absolute value serves as a norm in a field.*

ii) $(\mathbb{F}^n, \|\cdot\|_1)$, $\|(x_1, \ldots, x_n)\|_1 = \sum_{j=1}^n |x_j|$.

iii) $(\mathbb{F}^n, \|\cdot\|_2)$, $\|(x_1, \ldots, x_n)\|_2 = \sqrt{\sum_{j=1}^n |x_j|^2}$.

iv) $(\mathbb{F}^n, \|\cdot\|_\infty)$, $\|(x_1, \ldots, x_n)\|_\infty = \max\{|x_j| \colon j = 1, \ldots, n\}$..

Definition 1.2.3. *Let* V *be a vector space over the field* $\mathbb{F}$ *and* $\|\cdot\|_a$, $\|\cdot\|_b$ *be two norms. We say that the two norms are equivalent if there exist positive constants* $m, M > 0$ *such that for all* $x \in V$

$$c\|x\|_a \leq \|x\|_b \leq C\|x\|_a$$

Theorem 1.2.4. *Let vector space* V *be finite-dimensional. Then all norms are equivalent.*

Corollary 1.2.5. *For* $n \in \mathbb{N}$*, all norms in a vector space* $\mathbb{F}^n$ *are equivalent.*

Remark 1.2.6. *Because of the last corollary, we can use any norm in the given vector space. That's why, henceforward, we are going to use the the notation* $\|\cdot\|$ *if not specified otherwise, meaning that the claims hold for all norms in the observed vector space.*

Definition 1.2.7. *Let* $A \subseteq \mathbb{R}^n$*. Function* $f \colon A \longrightarrow \mathbb{R}^m$ *is differentiable in point* $c \in A$ *if there exists a linear operator* $L \in \mathcal{L}(\mathbb{R}^m, \mathbb{R}^n)$ *such that*

$$\lim_{n\to\infty} \frac{\| f(x) - f(c) - L(x - c) \|}{\| x - c \|} = 0.$$

The linear operator L *is unique and we call it a differential of the function* f *in point* c*. We denote it by* $Df(c)$*.*

Remark 1.2.8. *In case of* $m = n = 1$*,* $Df(c) = f'(c)$*.*

Lemma 1.2.9. *Let* $A \subseteq \mathbb{R}^n$*,* $f \colon \mathbb{R}^n \longrightarrow \mathbb{R}^m$*,* $f = (f_1, \ldots, f_m)$ *and* $c \in A$*. Function* f *is differentiable in point* c *if and only if* f_i *are differentiable for all* $i = 1, \ldots, m$*.*

Proof. In [8]. □

For a function $f\colon A \subseteq \mathbb{R}^n \longrightarrow \mathbb{R}^m$, let us observe its component functions $f_1, \ldots, f_m\colon A \longrightarrow \mathbb{R}$,

$$f(x) = (f_1(x), \ldots, f_m(x)).$$

Each function f_i is a function of n variables. Therefore we can define real functions

$$g_{ij}(h) = f_i(c_1, \ldots, c_{j-1}, c_j + h, c_{j+1}, \ldots, c_n).$$

around a point $c = (c_1, \ldots, c_n) \in A$. These functions we know how to differentiate.

Definition 1.2.10. *We define **partial derivatives** of a function f in point $c \in A$ as*

$$\begin{aligned}\frac{\partial f_i}{\partial x_j} = g'_{ij} &= \lim_{h\to 0} \frac{f_i(c_1, \ldots, c_{j-1}, c_j + h, c_{j+1}, \ldots, c_n)}{h} \\ &= \lim_{h\to 0} \frac{f_i(c + he_j) - f_i(c)}{h}\end{aligned}$$

$i = 1, \ldots, m$, $j = 1, \ldots, n$.

Theorem 1.2.11. ***(Jacobian matrix)***
Let $f\colon A \subseteq \mathbb{R}^n \longrightarrow \mathbb{R}^m$ be differentiable in point $c \in A$. Then all partial derivatives $\frac{\partial f_i}{\partial x_j}(c)$ exist. Furthermore, if we define a matrix $\nabla f(c) \in M_{m,n}(\mathbb{R})$ as

$$\nabla f(c) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1}(c) & \frac{\partial f_1}{\partial x_2}(c) & \ldots & \frac{\partial f_1}{\partial x_n}(c) \\ \frac{\partial f_2}{\partial x_1}(c) & \frac{\partial f_2}{\partial x_2}(c) & \ldots & \frac{\partial f_2}{\partial x_n}(c) \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1}(c) & \frac{\partial f_m}{\partial x_2}(c) & \ldots & \frac{\partial f_m}{\partial x_n}(c) \end{bmatrix}$$

*then $\nabla f(c)$ is a representation of $Df(c)$ in a pair of canonical bases of $\mathbb{R}^n$ and $\mathbb{R}^m$. This matrix is called the **Jacobian matrix** of the function f and it is often denoted by J_f or just J.*

Proof. [16], [8]. □

In Definition 1.2.10 we restricted ourselves to the points in the direction of coordinate axes. There is, of course, every reason to look at the other directions too.

Definition 1.2.12. *Let $f\colon A \subseteq \mathbb{R}^n \longrightarrow \mathbb{R}^m$ and $c \in A$. Furthermore, let $v \in \mathbb{R}^n$ be unit vector. If there exists a limit*

$$\lim_{h\to 0} \frac{f(c + hv) - f(c)}{h},$$

we call it a directional derivative of a function f in a direction of a vector v at the point c and we denote it by $\frac{\partial f}{\partial v}$.

Proposition 1.2.13. *Let $f\colon A \subseteq \mathbb{R}^n \longrightarrow \mathbb{R}^m$ and $v \in \mathbb{R}^n$, $\| v \| = 1$. If f is differentiable at the point c then there exists the directional derivative $\frac{\partial f}{\partial v}(c)$ and*

$$\frac{\partial f}{\partial v}(c) = Df(c)v.$$

Remark 1.2.14. *If we choose $v = e_i$ in Proposition 1.2.13, then it follows*

$$\frac{\partial f}{\partial e_i}(c) = \frac{\partial f}{\partial x_i}(c) = Df(c)e_i = \begin{pmatrix} \frac{\partial f_1}{\partial x_i}(c) \\ \frac{\partial f_2}{\partial x_i}(c) \\ \vdots \\ \frac{\partial f_m}{\partial x_i}(c) \end{pmatrix}.$$

Notice that $\frac{\partial f}{\partial x_i}(c) \in \mathbb{R}^m$ is equal to the i^{th} column of the Jacobian matrix from Theorem 1.2.11.

Especially interesting and important is the case when $m = 1$, i.e. when f is a real multivariate function. Then the Jacobian matrix is a $1 \times n$ dimensional:

$$\nabla f(c) = \begin{pmatrix} \frac{\partial f}{\partial x_1}(c) & \dots \frac{\partial f}{\partial x_n}(c) \end{pmatrix}.$$

This vector is called the **gradient** of f and it is denoted by $grad f(c)$.
Next we shall take a look at basic but important properties of derivatives which will be used throughout this work. Most proofs are straightforward and can be found in [16] or [8]. Especially important will be Theorem 1.2.17.

Proposition 1.2.15. *Let $f\colon A \subseteq \mathbb{R}^n \longrightarrow \mathbb{R}^m$ be differentiable in point $c \in A$. Let $B \subseteq \mathbb{R}^m$ Then the function $\alpha f + \beta g$ is differentiable in c and*

$$D(\alpha f + \beta g)(c) = \alpha Df(c) + \beta Dg(c).$$

Proposition 1.2.16. *Let $f\colon A \subseteq \mathbb{R}^n \longrightarrow \mathbb{R}^m$ and $g\colon A \longrightarrow \mathbb{R}$ be differentiable in point $c \in A$. Then the function gf is differentiable in c and*

$$D(gf(c))h = g(c)(Df(c)h) + f(c)(Dg(c)h), \quad h \in \mathbb{R}^n.$$

Theorem 1.2.17. ***(Chain rule)***
Let $g\colon A \subseteq \mathbb{R}^n \longrightarrow \mathbb{R}^m$ be differentiable in point $c \in A$. Let $B \in \mathbb{R}^m$ be an open subset such that $g(A) \subseteq B$ and $h\colon B \longrightarrow \mathbb{R}^p$. Then the function $f = h \circ g$ is differentiable in c and

$$Df(c) = D(h \circ g)(c) = Dh(g(c))Dg(c).$$

Remark 1.2.18. *In terms of the Jacobian matrices Theorem 1.2.17 can be written as*

$$J_f(x) = J_{h \circ g}(x) = J_h(g(x)J_g,$$

with $(i, j)^{th}$ *element:*

$$J_{ij} = \frac{\partial f_i}{\partial x_j} = \frac{\partial h_i}{\partial g_1}\frac{\partial g_1}{\partial x_j} + \frac{\partial h_i}{\partial g_2}\frac{\partial g_2}{\partial x_j} + \ldots + \frac{\partial h_i}{\partial g_k}\frac{\partial g_k}{\partial x_j}.$$

More generally, if our target function f *is the composite expression of* L *functions*

$$f = f^L \circ f^{L-1} \circ \ldots \circ f^1,$$

the corresponding Jacobian matrix is the product:

$$J = J_{f^L} \cdot J_{f^(L-1)} \cdot \ldots \cdot J_{f^1}. \tag{1.4}$$

Theorem 1.2.19. ***(Mean value theorem for real functions)***
Let $A \subseteq \mathbb{R}^n$ *be open and* $f \colon A \longrightarrow \mathbb{R}$ *differentiable in A. For every segment* $[x, y] \subseteq A$ *there exists a point* $c \in (x, y)$ *such that*

$$f(y) - f(x) = Df(c)(y - x).$$

Corollary 1.2.20. *Let* $A \subseteq \mathbb{R}^n$ *be open and* $f \colon A \longrightarrow \mathbb{R}^m$ *differentiable in A. For every segment* $[x, y] \subseteq A$ *there are points* $c_1, \ldots, c_m \in (x, y)$ *such that*

$$f_i(y) - f_i(x) = Df_i(c_i)(y - x), \quad i = 1, \ldots, m.$$

Theorem 1.2.21. ***(Mean value theorem for vector-valued functions)***
Let $A \subseteq \mathbb{R}^n$ *be open and* $f \colon A \longrightarrow \mathbb{R}^m$ *differentiable in A. If* $\| Df \|_\infty < \infty$ *then for every segment* $[x, y] \subseteq A$

$$\|f(y) - f(x)\| \leq \|Df(c)\|_\infty \|(y - x)\|.$$

1.3 Numerical analysis

In this chapter we are introducing certain problems that are arising when we are working with numerical methods and finite precision arithmetic in a computer. More precisely, we will take a look at the solution of the system of linear equations, and how the changes in the input data affect the final solution. We will introduce perturbations, and give some general results on the conditions and consequences of the given issues. More on this topic can be found in [9]. Before we dive into it, we introduce two different types of error that can emerge when dealing with numerical computing: truncation and roundoff error.

Definition 1.3.1. *Let x be the actual value of some process, and $\hat{x}$ the approximated value calculated using a numerical method. We define the truncation error as $\Delta x = x - \hat{x}$.*

Definition 1.3.2. *Let x be the result produced by a given algorithm using exact arithmetic, and $\tilde{x}$ the result produced by the same algorithm using finite-precision, rounded arithmetic. The difference $\Delta x = x - \tilde{x}$ is called roundoff error.*

Example 1.3.3. *(Truncation error when calculating derivatives).*
Suppose we want to find truncation in calculating the first derivative of a function

$$f : \mathbb{R} \longrightarrow \mathbb{R}, \quad f(x) = 5x^3.$$

at the point $x = 7$. We know that

$$f'(x) = 15x^2,$$
$$f'(7) = 735.$$

The exact first derivative is given by the formula defined in Definition 1.2.7. However, if we are calculating the derivative numerically, $\boldsymbol{h}$ in the definition of exact derivative has to be finite. The error caused by choosing $\boldsymbol{h}$ as finite is a truncation error in mathematical process of differentiation. Let us choose $\boldsymbol{h = 0.1}$. The approximate value is given by:

$$f'(7) = \frac{f(7 + 0.1) - f(7)}{0.1} = 745.55$$

Hence, the truncation error is $TE = |735 - 745.55| = 10.55$

1.3.1 Perturbation theory for systems of linear equations

In general, perturbation theory comprises methods for finding an approximate solution to a problem. It is based on the fact that it is possible to give an approximate description of the system under study using some specially selected "ideal" system which can be correctly and completely studied. One of the criteria of applicability of some part of perturbation theory, depending on the nature of the problem being studied, is that the equations describing the process in question contain a small parameter (or several small parameters), explicitly or implicitly. The requirement is furthermore that if the small parameter is zero, the equation is exactly solvable, so that the problem is reduced to finding the asymptotic behaviour of the best approximation to the true solution, accurately to within $\epsilon, \epsilon^2, ...$
Before describing a problem, we introduce one important definition.

Definition 1.3.4. *We say that a matrix $\boldsymbol{A} \in \mathbb{R}^{n \times n}$ is invertible or non-singular if there exists a matrix $\boldsymbol{B} \in \mathbb{R}^{n \times n}$ such that*

$$\boldsymbol{AB} = \boldsymbol{BA} = \boldsymbol{I}$$

Matrix $\boldsymbol{B}$ is called the inverse of the matrix $\boldsymbol{A}$. Matrices that don't have the inverse are called singular matrices.

We are considering the linear system of equations

$$\boldsymbol{A}x = b$$

where $\boldsymbol{A}$ is a non-singular $n \times n$ system matrix, b is a vector in $\mathbb{R}^n$ and $x \in \mathbb{R}^n$ is a vector whose coefficients $x_1, x_2, \ldots, x_n$ are unknown.
We know from linear algebra that, in theory, solving this system is a simple problem. The solution is given by formula $x = \boldsymbol{A}^{-1}b$, where $\boldsymbol{A}^{-1}$ is a inverse matrix of $\boldsymbol{A}$. Thereby exist explicit formulas for the inverse matrix $\boldsymbol{A}^{-1}$ and the solution x.
Unfortunately, when using a finite precision arithmetic in a computer, it is impossible to get the exact solution, even with the theoretically perfect method and formulas. Also, when working in a real environment and on the real data, we cannot be sure that our input data, i.e. system matrix $\boldsymbol{A}$ and vector b, are exactly correct. These small inaccuracies are called *perturbations*, and always have to be kept in mind when solving practical problems since:

- The data A and/or b may be obtained from measurements, and therefore they are error prone.
- Representation of data as floating point numbers using computers produces errors.

Hence, one always has to emanate from the fact that one solves a perturbed linear system instead of the given one. More precisely, we can say that our approximation of the solution $\tilde{x} = x + \delta x$ satisfies the perturbed system

$$(\boldsymbol{A} + \delta\boldsymbol{A})(x + \delta x) = b + \delta b.$$

Our aim is to examine how perturbations of $\boldsymbol{A}$ and b affect the solution of the system x.
For simplicity, let us first examine the case when $\delta b = 0$. Then the perturbed system has the form

$$(\boldsymbol{A} + \delta\boldsymbol{A})(x + \delta x) = b.$$

The question that immediately arises when dealing with this system is the singularity of the perturbed system matrix $\boldsymbol{A} + \delta\boldsymbol{A}$. From the equality $\boldsymbol{A} + \delta\boldsymbol{A} = \boldsymbol{A}(\boldsymbol{I} + \boldsymbol{A}^{-1}\delta\boldsymbol{A})$ we deduce that the matrix $\boldsymbol{A} + \delta\boldsymbol{A}$ is regular if $\boldsymbol{I} + \boldsymbol{A}^{-1}\delta\boldsymbol{A}$ is regular. The conditions for which we can guarantee the non-singularity of the matrix $\boldsymbol{I} + \boldsymbol{A}^{-1}\delta\boldsymbol{A}$ is given by the following results.

Lemma 1.3.5. *Let $\boldsymbol{A} \in \mathbb{R}^{n\times n}$ such that $\|\boldsymbol{A}\| < 1$. Then the matrix $\boldsymbol{I} - \boldsymbol{A}$ is non-singular and it holds that*

$$\|(\boldsymbol{I} - \boldsymbol{A})^{-1}\| \leq \frac{1}{1 - \|\boldsymbol{A}\|}.$$

Proof. For all $x \in \mathbb{R}^n$, $x \neq 0$:

$$\|(\boldsymbol{I} - \boldsymbol{A})x\| \geq \|x\| - \|\boldsymbol{A}x\| \geq \|x\| - \|\boldsymbol{A}\|\,\|x\| = (1 - \|\boldsymbol{A}\|)\,\|x\| > 0.$$

Therefore, the linear system

$$(\boldsymbol{I} - \boldsymbol{A})x = 0$$

has the unique solution $x = 0$ and $\boldsymbol{I} - \boldsymbol{A}$ is non-singular. The estimate of the norm of the $(\boldsymbol{I} - \boldsymbol{A})^{-1}$ follows from

$$\begin{aligned} 1 &= \|(\boldsymbol{I} - \boldsymbol{A})^{-1}(\boldsymbol{I} - \boldsymbol{A})\| = \|(\boldsymbol{I} - \boldsymbol{A})^{-1} - (\boldsymbol{I} - \boldsymbol{A})^{-1}A\| \\ &\geq \|(\boldsymbol{I} - \boldsymbol{A})^{-1}\| - \|(\boldsymbol{I} - \boldsymbol{A})^{-1}A\| \\ &\geq \|(\boldsymbol{I} - \boldsymbol{A})^{-1}\| - \|(\boldsymbol{I} - \boldsymbol{A})^{-1}\|\,\|\boldsymbol{A}\| \\ &= (1 - \|\boldsymbol{A}\|)\,\|(\boldsymbol{I} - \boldsymbol{A})^{-1}\|. \end{aligned}$$

□

Corollary 1.3.6. *Let $\boldsymbol{A} \in \mathbb{R}^{n\times n}$ be a non-singular matrix and $\delta\boldsymbol{A} \in \mathbb{R}^{n\times n}$. Assume that*

$$\|\delta\boldsymbol{A}\| \leq \frac{1}{\|\boldsymbol{A}^{-1}\|}.$$

Then $\boldsymbol{A} + \delta\boldsymbol{A}$ is non-singular and

$$\|(\boldsymbol{A} + \delta\boldsymbol{A})^{-1}\| \leq \frac{\|\boldsymbol{A}^{-1}\|}{1` - \|\boldsymbol{A}^{-1}\delta\boldsymbol{A}\|} \leq \frac{\|\boldsymbol{A}^{-1}\|}{1 - \|\boldsymbol{A}^{-1}\|\,\|\delta\boldsymbol{A}\|}$$

Proof. The existence of $(\boldsymbol{A} + \delta\boldsymbol{A})^{-1}$ follows from Lemma 1.3.5 since

$$\|\delta\boldsymbol{A}\| \leq \frac{1}{\|\boldsymbol{A}^{-1}\|} \Rightarrow 1 \geq \|\boldsymbol{A}^{-1}\delta\boldsymbol{A}\|$$

and $\boldsymbol{A} + \delta\boldsymbol{A} = A(\boldsymbol{I} + \boldsymbol{A}^{-1}\delta\boldsymbol{A})$

$$\begin{aligned} \|(\boldsymbol{A} + \delta\boldsymbol{A})^{-1}\| &= \|(\boldsymbol{I} + \boldsymbol{A}^{-1}\delta\boldsymbol{A})^{-1}\boldsymbol{A}^{-1}\| \leq \|\boldsymbol{A}^{-1}\| \cdot \|(\boldsymbol{I} + \boldsymbol{A}^{-1}\delta\boldsymbol{A})^{-1}\| \\ &\leq \frac{\|\boldsymbol{A}^{-1}\|}{1` - \|\boldsymbol{A}^{-1}\delta\boldsymbol{A}\|} \leq \frac{\|\boldsymbol{A}^{-1}\|}{1 - \|\boldsymbol{A}^{-1}\|\,\|\delta\boldsymbol{A}\|} \end{aligned}$$

□

The corollary demonstrates that for a non-singular matrix $\boldsymbol{A}$ the perturbed matrix $\boldsymbol{A} + \delta\boldsymbol{A}$ is also non-singular if the perturbation $\delta\boldsymbol{A}$ is sufficiently small. Having that in mind, let

$$\epsilon = \frac{\|\delta\boldsymbol{A}\|}{\|\boldsymbol{A}\|} \ll 1$$

non-singularity of the matrix $\boldsymbol{A} + \delta\boldsymbol{A}$ is assured if

$$\|\boldsymbol{A}^{-1}\|\,\|\delta\boldsymbol{A}\| = \epsilon(\|\boldsymbol{A}\| \cdot \|\boldsymbol{A}^{-1}\|) < 1, \text{ i.e. } \epsilon < \frac{1}{\|\boldsymbol{A}\|\,\|\boldsymbol{A}^{-1}\|}.$$

Then it holds $\|\boldsymbol{A}^{-1}\delta\boldsymbol{A}\| < 1$ and $(\boldsymbol{A} + \delta\boldsymbol{A})^{-1} = (\boldsymbol{I} + \boldsymbol{A}^{-1}\delta\boldsymbol{A})^{-1}\boldsymbol{A}^{-1}$, so $\tilde{x} = (\boldsymbol{A} + \delta\boldsymbol{A})^{-1}b$ can be written as

$$\tilde{x} = (\boldsymbol{I} + \boldsymbol{A}^{-1}\delta\boldsymbol{A})^{-1}\boldsymbol{A}^{-1}b = (\boldsymbol{I} + \boldsymbol{A}^{-1}\delta\boldsymbol{A})^{-1}x, \text{ i.e. } (\boldsymbol{I} + \boldsymbol{A}^{-1}\delta\boldsymbol{A})\tilde{x} = x.$$

Therefore, $x = \tilde{x} = \boldsymbol{A}^{-1}\delta A\tilde{x}$, and

$$\|x - \tilde{x}\| \leq \|\boldsymbol{A}^{-1}\delta\boldsymbol{A}\|\,\|\tilde{x}\|.$$

Because of $\|\boldsymbol{A}^{-1}\delta\boldsymbol{A}\| \leq \|\boldsymbol{A}^{-1}\|\,\|\delta\boldsymbol{A}\|$, we have

$$\frac{\|x - \tilde{x}\|}{\|\tilde{x}\|} \leq \|\boldsymbol{A}^{-1}\|\,\|\boldsymbol{A}\|\frac{\|\delta\boldsymbol{A}\|}{\|\boldsymbol{A}\|} = \epsilon\|\boldsymbol{A}^{-1}\|\|\boldsymbol{A}\|.$$

This shows us that relative error in the solution $\tilde{x}$ can be magnified by a factor of $\|\boldsymbol{A}^{-1}\|\|\boldsymbol{A}\|$, regards to relative change ϵ in the input matrix A. This discussion serves as a motivation for the following definition.

Definition 1.3.7. *Let $\boldsymbol{A} \in \mathbb{R}^{m\times n}$ be a matrix. We define a condition number of a matrix $\boldsymbol{A}$ as*

$$\kappa(\boldsymbol{A}) = \|\boldsymbol{A}\|\,\|\boldsymbol{A}^{-1}\|$$

if $\boldsymbol{A}$ is regular. Otherwise we set $\kappa(\boldsymbol{A}) = \infty$. If $\kappa(\boldsymbol{A})$ is such that $\kappa(\boldsymbol{A}) \cdot \epsilon \ll 1$, we say that the linear system is well-conditioned. Otherwise, it is ill-conditioned.

Example 1.3.8. *Consider the linear system of equations*

$$\begin{bmatrix} 1 & 1 \\ 1 & 0.999 \end{bmatrix} x = \begin{bmatrix} 2 \\ 1.999 \end{bmatrix}$$

Obviously, the solution is $x = \begin{bmatrix} 1 & 1 \end{bmatrix}^T$. We compute that

$$\boldsymbol{A}^{-1} = \begin{bmatrix} -999 & 1000 \\ 1000 & -1000 \end{bmatrix}$$

and therefore $\kappa_\infty(\boldsymbol{A}) = 4000$.

Let us now take a look at a general case, where $\delta\boldsymbol{A} \neq 0$ and $\delta b \neq 0$. The results are summarized in the following theorem.

Theorem 1.3.9. *Let* $\boldsymbol{A}x = b$, $(\boldsymbol{A} + \delta\boldsymbol{A})(x + \delta x) = b + \delta b$, *where* $\|\delta\boldsymbol{A} \leq \epsilon\|\boldsymbol{A}\|$ *and* $\|\delta b\| \leq \epsilon\|b\|$. *If* $\epsilon\|\boldsymbol{A}^{-1}\|\|\boldsymbol{A}\| < 1$, *then it holds*

$$\frac{\|\delta x\|}{\|x\|} \leq \frac{\epsilon}{1 - \epsilon\|\boldsymbol{A}^{-1}\|\,\|\boldsymbol{A}\|}\left(\frac{\|\boldsymbol{A}^{-1}\|\,\|b\|}{\|x\|} + \|\boldsymbol{A}^{-1}\|\,\|\boldsymbol{A}\|\right) \leq 2\frac{\epsilon\|\boldsymbol{A}^{-1}\|\,\|\boldsymbol{A}\|}{1 - \epsilon\|\boldsymbol{A}^{-1}\|\,\|\boldsymbol{A}\|}.$$

Thereby exist perturbations $\delta\boldsymbol{A}$ *and* δb *such that above inequality is almost achieved. More precisely, there exist* $\delta\boldsymbol{A}$ *and* δb *such that* $\|\delta\boldsymbol{A}\| = \epsilon\|\boldsymbol{A}\|$, $\|\delta b\| = \epsilon\|b\|$, *and*

$$\frac{\|\delta x\|}{\|x\|} \geq \frac{\epsilon}{1 + \epsilon\|\boldsymbol{A}^{-1}\|\|\boldsymbol{A}\|}\left(\frac{\|\boldsymbol{A}^{-1}\|\,\|b\|}{\|x\|} + \|\boldsymbol{A}^{-1}\|\,\|\boldsymbol{A}\|\right).$$

1.4 Software environment

The software that will be used to implement the methods in this work will be MATLAB. Its name is the abbreviation of Matrix laboratory and it is a proprietary multi-paradigm programming language and numerical computing environment. It allows matrix manipulations, plotting of functions and data, implementations of algorithms, etc. Although MATLAB is primarily intended for numerical computing, it is extremely powerful and it allows the use of object-oriented features. In this section we will introduce object-oriented programming. Special attention will be given to a feature called operator overloading, since the most method in this work will use it as a fundamental technology. More on MATLAB object-oriented programming can be found in [13].

1.4.1 Object-oriented programming

As mentioned above, the software environment that will be used throughout this work is MATLAB. More specifically, we will take advantage of its object-oriented nature. Especially useful and important will be a feature called operator overloading. Hereafter, object-oriented programming (OOP) and its basic principles will be introduced.
The main goal of OOP is to make the code simpler. To do this, the program is divided into independent blocks of code that are called objects. An object is a collection of data and functions - called properties and methods respectively. An object can represent anything, from geometric shape to cars or any abstract entities. To use objects, a programmer needs to define classes. In MATLAB, one example could be:

```
classdef square
    properties
        side
    end
    methods
```

```
        function A = area(self)
            A = self.a * self.a;
        end
        function S = scope(self)
            S = 4 * self.a;
        end
    end
end
```

Listing 1.1: MATLAB class example

Objects are actually *instances* of classes. Objects can be characterized by 4 terms:

1. Encapsulation - objects are independent, which means that all the data needed is stored inside the object itself.
2. Abstraction - objects have interface so users can access its properties and methods from outside the object.
3. Inheritance - essentially a copying mechanism. We can create multiple objects that are alike which prevents us from copy-pasting the same code.
4. Polymorphism - objects that are alike and inherited from one another can share functionality while being adapted as necessary.

1.4.2 Operator overloading

One of the most popular and widely used features of OOP is operator overloading. It is used to redefine the operators such as $+, -, \times, /$ etc. These operators by default work only on standard data types such as int, float, char etc. But with this OOP feature we can give this operators additional meaning. The way we do it is by defining certain methods inside our class. For instance, if we want to add our objects, we would overload, or redefine a method called *plus()*. Following is a simple example where operator overloading is a go to technology.

```
classdef fraction
    properties
        numerator
        denominator
    end
    methods
        function obj = fraction(num, den)
        function res = plus(a, b)...
```

```
        function res = times(a, b)...
        % ...
    end
end
```

Listing 1.2: Operator overloading - class fraction

This approach allows us to write simple and natural code such as:

```
>> fraction x(1, 2);
>> fraction y(1,3);
>> x + y
ans =
fraction with properties:
    numerator: 5
    denominator: 6
```

Listing 1.3: Usage of class fraction - $\frac{1}{2} + \frac{1}{3}$

Apart from regular arithmetic and logical operations, it is possible to overload the math functions such as $\sin, \cos, \log$, as well as relational operators such as $==, >, \leq$ etc. For a deeper dive into operator overloading in MATLAB and which functions/operators can be redefined we refer the reader to `https://nl.mathworks.com/help/matlab/matlab_oop/implementing-operators-for-your-class.html`.

Chapter 2

Problem statement and algorithms

In this chapter we will present the problem and its possible solutions i.e. the methods that can be used to solve it. We will start by introducing a framework - an array of coordinate system transformations used in photolithography equipment. First we wil acquaint ourselves with the basics of photolithography. More on this specific topic can be found in [14]. After describing the problem it will turn out that everything revolves around computing the Jacobian matrix, a term defined in Theorem 1.2.11. Therefore, the main goal of this chapter will be to investigate and implement different methods to compute derivatives. Of course, there are many methods that we can choose, but we will see that the best ones are the following:

1. Numerical methods - finite step and complex step approximation
2. Automatic differentiation and its two modes - forward and reverse

All of these methods will be thoroughly explained and investigated in this chapter. We will give a summary of the differences between the methods, as well as its implementations and advantages/disadvantages of each one. More information on the derivatives and its computations can be found in [18], [15] and [3].

2.1 Problem Setting

Photolithography, in its most general form, is a term used for techniques that use light to produce patterned thin films of suitable material over a substrate, such as a silicon wafer. In this context, a wafer is a thin slice of semiconductor, such as crystalline silicon. It serves as the substrate for microelectronic devices built in and upon the wafer. Typically, ultraviolet light is used to transfer a geometric design from an optical mask to a light-sensitive chemical called photoresist coated on the wafer. This photoresist either hardens or

breaks down when exposed to light. The patterned film is then created by removing softer parts of the coating. Photolithography is widely used in different applications. The most common and wide-known use is for fabricating electronic chips (solid-state memories, microprocessors). It is able to create extremely small patterns, down to a few tens of nanometers in size.
The root of the name photolithography has Greek origins. It is compounded from three words:

1. photo - light,
2. litho - stone,
3. graphy - writing.

This process was invented by Alphonse Poitevin in 1855 who combined the process of lithography with light. The same process is used today to print microchips that exist in cellphones, computers and every other electronic devices. Photolithography is quite a complicated procedure that requires many steps. Our focus will be on a step called *exposure*. Here, the photoresist is exposed to a beam of intense light. One of the possible exposure methods uses projection systems. These systems project the mask, usually called reticle, onto the wafer. Essentially, light goes through the reticle and projects the exposed point(s) onto the wafer while passing through a number of projection lenses that magnify/squeeze the light beam. This process is illustrated in Figure 2.1. To project a given point, the machine uses a number of coordinate transformation systems that connect different hardware parts. Of course, since extreme precision is required, during positioning of the wafers and reticles, as well as their stages, there is usually misalignment and mispositioning of the used instruments. These errors can be detrimental to the resulting product. One example of a possible error is illustrated in Figure 2.2.

To be more precise, let $f\colon \mathbb{R}^n \longrightarrow \mathbb{R}^3$ be the above mentioned coordinate system transformation. This is nicely illustrated at Figure 2.3. n input arguments represent the parameters that we wish to correct the error in. Those can be translation, rotation or magnification parameters in any direction in 3D, as well as some other parameters of interest. Notice that we wish those parameters to be 0. Furthermore, we define the *expected position* $x \in \mathbb{R}^n$ of the reticle. This point represents a position of the reticle in a perfect environment - without any positioning errors. Using the given coordinate system transformation f, we yield the expected wafer position $f(x)$. On the other hand, we have *aligned position* $\tilde{x}$ which is the actual position of the reticle. Similarly, at the wafer level there exists the aligned position $f(\tilde{x})$. Finally, $\Delta x = x - \tilde{x}$ represents the error in input data of the given coordinate system transformation. x and $f(x)$ are known a priori, and $f(\tilde{x})$ is measured by high-precision positioning instruments. The only thing we do not have is the error Δx. The goal is to

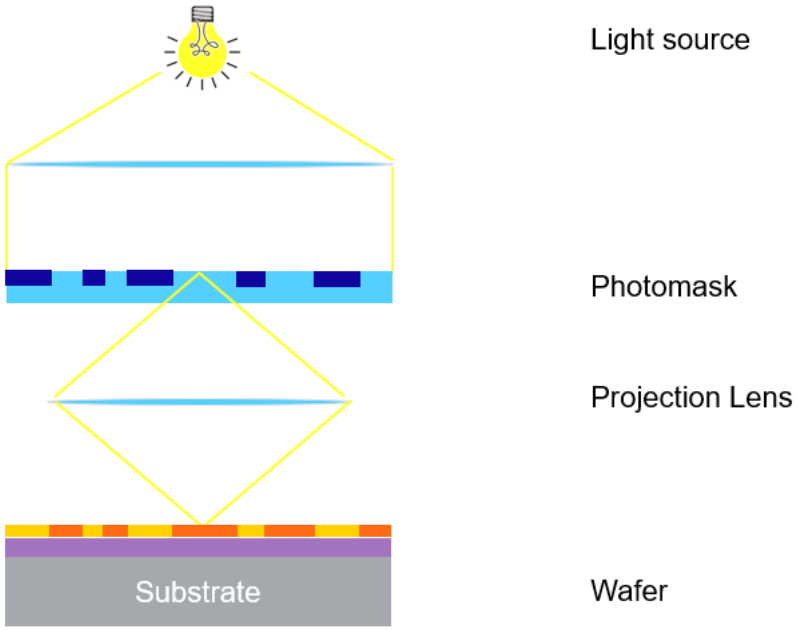

Figure 2.1: Exposure process in photolithography

calculate that error so we can correct for it. To do that we make use of Corollary 1.2.20. More precisely, we use the formula:

$$\Delta y = f(x) - f(\tilde{x}) = J_f \Delta x.$$

The Jacobian matrix describes the variation of f with respect to each of the parameters. So, to calculate Δx, one has to calculate the Jacobian matrix J_f of the coordinate system transformation f. Once we have that, it follows:

$$\Delta x = J_f^{-1} \Delta y$$

There is of course the question whether a matrix J_f is invertible. Thus, a few conditions have to be met:

1. J_f has to be of full rank.
2. J_f has to be well-conditioned, e.g. the parameters are well modelled.

The objective of this work is to calculate the Jacobian matrix in the context of coordinate system transformations. As mentioned above, we will primarily focus on accuracy and efficiency of the observed methods, as both are crucial to obtaining the best result in applications. A more thorough example of how and when this approach is used we refer the reader to [19].

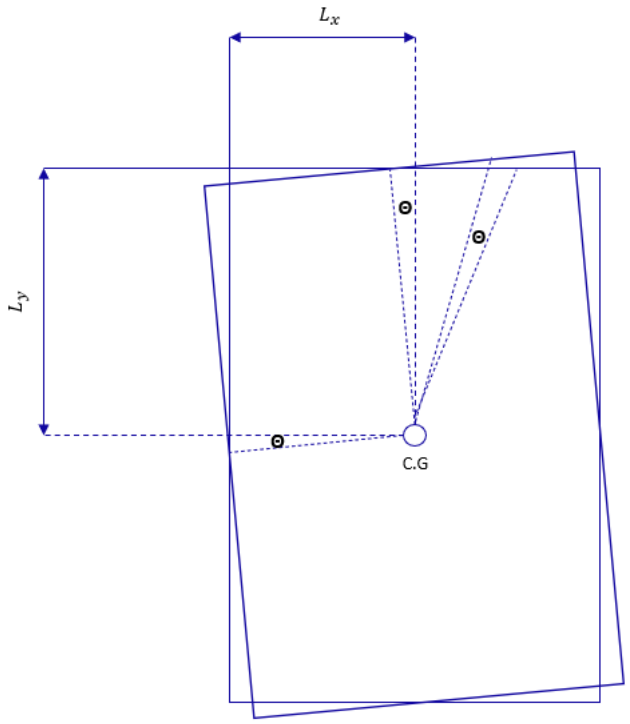

Figure 2.2: Mispositioning of the reticle - adopted from [19].

2.2 Computing derivatives

Numerical simulations arising in large-scale scientific applications often require the evaluation of derivatives of some objective function, as they play a crucial role in numerical computing. Some examples include solution of nonlinear systems of equations, ordinary and partial differential equations and differential-algebraic equations. Derivatives are also ubiquitous in the areas of sensitivity analysis and design optimization. Therefore, the topic of computational differentiation (i.e. some process by which derivatives are obtained with a computer) is thoroughly researched. With that said, further development and investigation of the known and new methods is still an open field of research today. When derivatives are obtained with a computer, errors may be introduced in two ways: truncation and roundoff error, as defined in Definition 1.3.1 and 1.3.2 respectively.

There are multiple approaches to this problem, and all of them have their advantages and disadvantages and appear in a variety of applications. In general, when computing derivatives with a computer, we want to meet the following requirements:

a) Efficiency - the amount of memory and runtime should be minimized.

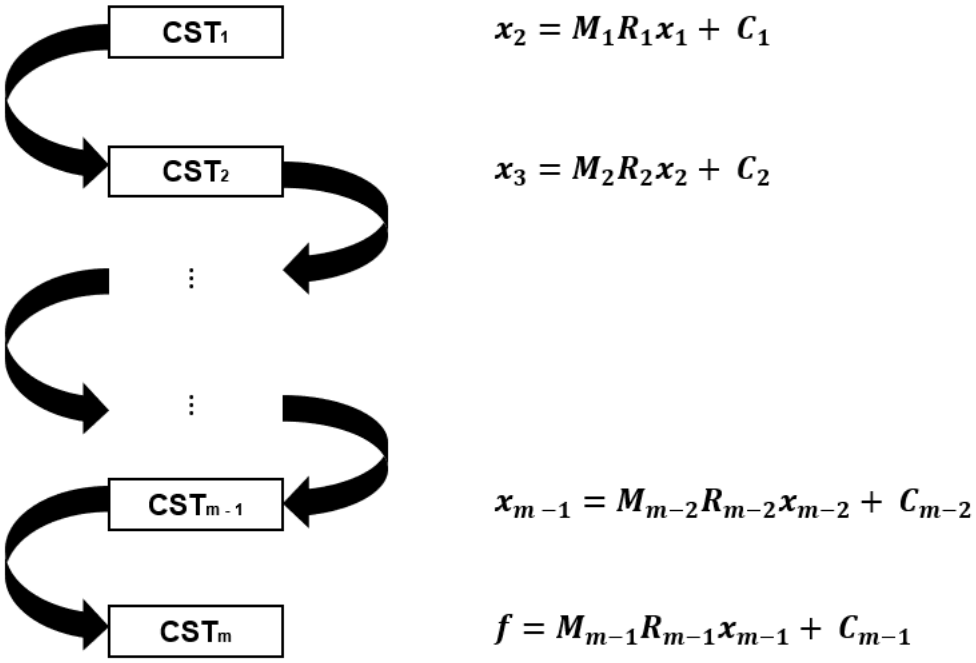

Figure 2.3: Coordinate system transformations.

b) Reliability - the truncation error should be minimized or ideally, non-existent.

c) Scalability - the method should give a correct result for the simplest and most complicated programs.

d) Human effort - it should be easy to use.

The classical, or the old school way to compute derivatives is hand-coded differentiation. Here the derivatives are obtained by deriving the analytic expressions by hand and then coding a subroutine for their computation. The clear advantage of this approach is that the result is exact (to roundoff) and can even be efficient when coded correctly. But this approach is extremely time consuming, as well as error prone for complex cases. Shortly, it is not practical for most problems of interest.
One alternative to hand-coding is symbolic differentiation. Here the idea is to find an explicit expression for the Jacobian matrix J. A big disadvantage of this approach is that the expressions get really complicated really fast, especially for the large-scale problems. It is also extremely painful when one wants to obtain higher-order derivatives. Moreover, because of its tree-based structure, it is often times extremely inefficient due to its rapid growth of underlying expressions. With that said, it is still used widely, for example it is implemented via the software Maple.

2.2.1 Numerical differentiation

A well-known and widely used approach to all kinds of general problems in numerical mathematics is to use approximations rather then exact solutions. Reasons for that are of course computational cost (both memory and time-wise) and the simplicity of implementations. Numerical methods for computing derivatives satisfy both. There is, naturally, an obvious disadvantage of this kind of methods, and that is the truncation error. In case of derivatives, in many applications this loss of accuracy is not acceptable. That is especially true when there are complicated functions and programs, or when there is a need for higher order derivatives. Specifically, there are two methods that we will take a closer look at. Those are the well-known and widely used finite difference approximation and the less-known complex-step approximation.

2.2.2 Finite difference approximation of derivatives

The idea of the finite difference approximation is simple and it uses basic definitions of derivatives from calculus. More precisely, the Definition 1.2.7. Here h represents a perturbation in variable x_j. Imagine h being finite and representable in a computer. Then, what we get is an approximation of the derivative:

$$\frac{\partial f_i}{\partial x_j} \approx \frac{f_i(x + he_j) - f_i(x)}{h}. \tag{2.1}$$

Obviously, there is a truncation error. The question is how good is this approximation, i.e. how big is the error?
Let us write the Taylor series about the base point x, expanded in the variable h

$$f_i(x + he_j) = f_i(x) + \frac{\partial f_i}{\partial x_j}(x) \cdot h + \frac{1}{2}\frac{\partial^2 f_i}{\partial^2 x_j}(x) \cdot h^2 + \frac{1}{3!}\frac{\partial^3 f_i}{\partial^3 x_j}(x) \cdot h^3 + \dots \tag{2.2}$$

from where by dividing the entire equation by h it follows:

$$\frac{f_i(x + he_j) - f_i(x)}{h} = \underbrace{\frac{\partial f_i}{\partial x_j}(x)}_{\text{partial derivative we want}} + \underbrace{\frac{1}{2}\frac{\partial^2 f_i}{\partial^2 x_j}(x) \cdot h + \frac{1}{3!}\frac{\partial^3 f_i}{\partial^3 x_j}(x) \cdot h^2 + \dots}_{\text{error}}$$

We have proven:

Lemma 2.2.1. *For a function $f : \mathbb{R}^n \longrightarrow \mathbb{R}^m$ is valid:*

$$\frac{\partial f_i}{\partial x_j}(x) = \frac{f_i(x + he_j) - f_i(x)}{h} + O(h)$$

$i = 1, \dots, m$, $j = 1, \dots, n$. We say that the dominant error term is on the order of h.

The expression in Lemma 2.2.1 is called Forward Finite Difference Approximation (FFDA). Similarly, we define a Backward Finite Difference Approximation (BFDA)

$$\frac{\partial f_i}{\partial x_j}(x) = \frac{f_i(x) - f_i(x - he_j)}{h} + O(h). \tag{2.3}$$

This follows directly from Taylor series expansion of $f_i(x - he_j)$:

$$f_i(x - he_j) = f_i(x) - \frac{\partial f_i}{\partial x_j}(x) \cdot h + \frac{1}{2}\frac{\partial^2 f_i}{\partial^2 x_j}(x) \cdot h^2 - \frac{1}{3!}\frac{\partial^3 f_i}{\partial^3 x_j}(x) \cdot h^3 + \dots \tag{2.4}$$

Furthermore, if we subtract the equation 2.4 from 2.2 we get a Central Finite Difference Approximation (CFDA)

$$\frac{\partial f_i}{\partial x_j}(x) = \frac{f_i(x + he_j) - f_i(x - he_j)}{2h} + O(h^2). \tag{2.5}$$

The question now is what is the optimal step-size h? One could say that the more we decrease h, we get better results. So why wouldn't we just decrease h? Here comes the biggest disadvantage of these methods - and that's the trade-off between truncation and roundoff error. Namely, every number represented in a computer comes with a built-in roundoff error. That error is of order 10^{-16} and its usually referred to as machine precision ϵ. It is a limit of how close can a computer tell different numbers apart. Effectively what we have is the following:

$$\frac{\partial f_i}{\partial x_j}(x) = \frac{f_i(x + he_j) - f_i(x)}{h} + \underbrace{O(h)}_{\text{Truncation error}} + \underbrace{\frac{\epsilon}{h}}_{\text{Roundoff error}}. \tag{2.6}$$

So, the more we decrease h, the more the truncation error decreases. But on the other hand, decreasing h means increasing the roundoff error. Because of that there is an optimal h for which the cumulative error is minimized. In practice, that number is usually around 10^{-5} or 10^{-6}. This is illustrated in Figure 2.4.

2.2.3 Complex step approximation

As an alternative to a finite step differencing approximation we will introduce another numerical method: complex step differencing approximation. The story and idea behind this method can be found in [11]. Here we are concerned with an analytic function $f\colon \mathbb{C}^n \longrightarrow \mathbb{C}^m$. Put it simply, it means that f is a infinitely differentiable function.
Let $z = x + ih \in \mathbb{C}$. Now let us expand $f_i(z)$, $i \in \{1, \dots, m\}$ in a Taylor series of the coordinate $j \in \{1, \dots, n\}$ of the real axis:

$$f_i(x + ihe_j) = f_i(x) + ih\frac{\partial f_i}{\partial x_j}(x) - \frac{1}{2}h^2\frac{\partial^2 f_i}{\partial^2 x_j}(x) - ih^3\frac{1}{3!}\frac{\partial^3 f_i}{\partial^3 x_j} + \dots$$

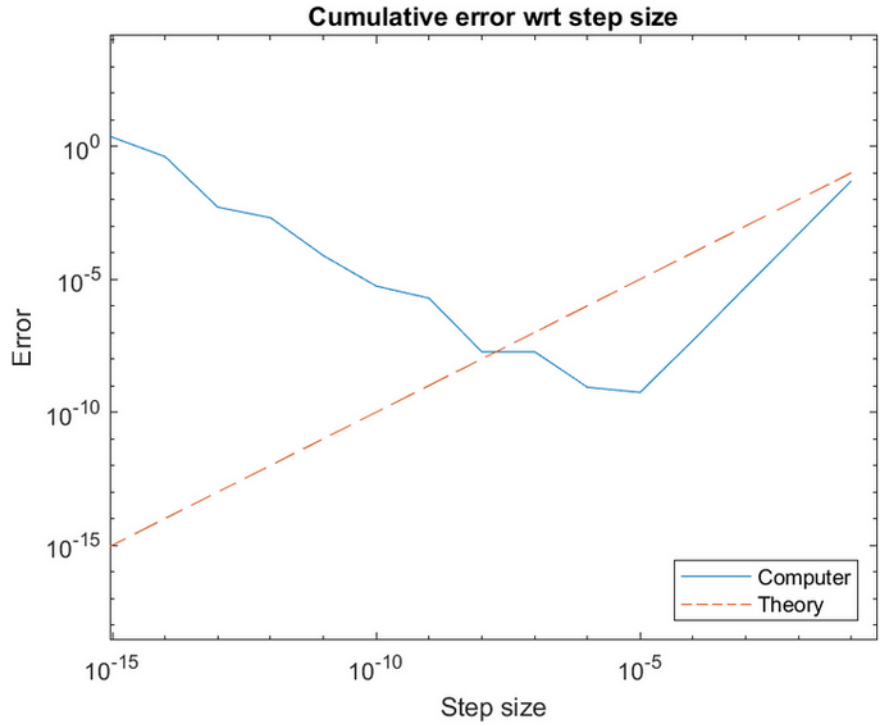

Figure 2.4: Cumulative error plotted against step-size h. Figure has to be observed from right to left, corresponding to decreasing of the step-size h. Blue line shows the magnitude of the error with respect to different step sizes, while the red line shows what would happen if we could represent the numbers in a computer exactly.

Take the imaginary part of both sides and divide by h:

$$\frac{\partial f_i}{\partial x_j} = \frac{Im(f_i(x + ihe_j))}{h} + O(h^2).$$

Therefore, simply by evaluating function f at the imaginary argument $x + ihe_j$ and dividing by h we got an approximation of the partial derivative $\frac{\partial f_i}{\partial x_j}$, $\forall i \in \{1, \ldots, m\}$, $\forall j \in \{1, \ldots, n\}$. Here, the optimal step-size h would be around 10^{-8}. The described method is called Complex Step Differentiation (CSD) and it is often a better alternative compared to FDA.

2.3 Automatic differentiation

In the previous section we discussed various methods for calculating derivatives in a computer finite precision arithmetic. Each one had its advantages and disadvantages. For

example, we have seen that finite step and complex step approximations inevitably yield truncation errors. Even if the step size h is optimally chosen, the values of the derivatives will be accurate to only about $\frac{1}{2}$ or $\frac{1}{3}$ of the significant digits of the observed function f. For higher order derivatives these accuracy problems increase more and more. In contrast, automatic differentiation methods do not incur truncation errors at all and yield exact derivatives up to round-off. This feature is the main advantage of AD. It is good to mention that there are two modes of AD: forward and reverse mode. Both have its advantages and applications, and one of our goals is to compare the two modes. Literature and material used throughout this section: [15], [10], [7], [12].

Remark 2.3.1. *Automatic differentiation does not incur truncation errors, the term defined in Definition 1.3.1. This claim will be proven in the following section. Same cannot be said for the roundoff error defined in Definition 1.3.2, as the latter one is an inevitable consequence of working with the finite precision on the computer.*

Likewise, symbolic differentiation, because of its tree-based structure tends to be too slow and inefficient. On the other hand AD, if implemented carefully and correctly, performs quite fast. In the next sections we will take a closer look at AD and its implementations. Specifically, we will take a look at two modes of AD that are widely used: forward and reverse mode.

2.3.1 How automatic differentiation works

The main idea behind AD is:

> *Even the biggest, most complicated problems must be built from a smaller set of primitive sub-problems.*

In case of differentiation, the tool that makes that possible is the chain rule which we defined in Theorem 1.2.17. This will be our building block for AD.
Given a target function $f\colon \mathbb{R}^n \longrightarrow \mathbb{R}^m$, the corresponding $m\times n$ Jacobian matrix J has $(i, j)^{th}$ component:

$$J_{ij} = \frac{\partial f_i}{\partial x_j}.$$

Now suppose f is a composite function: $f(x) = (h_1 \circ h_2 \circ \ldots \circ h_L)(x) = h_1(h_2(\ldots(h_L(x)))), L \in \mathbb{N}$, with $x \in \mathbb{R}^n$, $f\colon \mathbb{R}^n \longrightarrow \mathbb{R}^m$ and $h_1\colon \mathbb{R}^{l_1} \longrightarrow \mathbb{R}^m, h_2\colon \mathbb{R}^{l_2} \longrightarrow \mathbb{R}^{l_1}, \ldots, h_L\colon \mathbb{R}^n \longrightarrow \mathbb{R}^{l_L}$ and let $u \in \mathbb{R}^n$. One *sweep* of forward AD evaluates the action of the Jacobian matrix J on u; $J \cdot u$. Let us remember the discussion from Remark 1.2.18. There we had a composite

function and verified that its Jacobian is of the form (1.4). From there it follows:

$$\begin{aligned} J &= J_L \cdot J_{L-1} \cdot \ldots \cdot J_1 \cdot u \\ &= J_L \cdot J_{L-1} \cdot \ldots \cdot J_3 \cdot J_2 \cdot u_1 \\ &= J_L \cdot J_{L-1} \cdot \ldots \cdot J_4 \cdot J_3 \cdot u_2 \\ &\ldots \\ &= J_L \cdot u_{L-1} \end{aligned} \tag{2.7}$$

where we have a recursion

$$\begin{aligned} u_1 &= J_1 \cdot u \\ u_l &= J_L \cdot u_{L-1}, \quad l = 2, \ldots, L \end{aligned} \tag{2.8}$$

Hence, given a complex function f, we can break down the action of the Jacobian matrix on a vector into simple components which we evaluate sequentially.

Remark 2.3.2. (2.7) *corresponds to a directional derivative of f with respect to u.*

Let us now consider a vector in the output space $w \in \mathbb{R}^m$. Reverse sweep of AD computes the action of the transposed Jacobian matrix on w, $J^T \cdot w$. Same process that we did in the forward mode above applies here. More precisely, we repeat the procedure from equation 2.7 while replacing J with J^T, and $u \in \mathbb{R}^n$ with above defined $w \in \mathbb{R}^m$. Again, we successfully broke our program or a function into a sequence of elementary operations. We will later see that the right choice of u or w allows us to compute one row or column of the Jacobian matrix, respectively. But before we take a deeper dive into two modes of AD, we will get familiar with a concept that will be crucial in building a theory of AD.

Definition 2.3.3. ***(Computational graph)***
A computational graph is defined as a directed graph where the nodes correspond to mathematical operations.

Remark 2.3.4. *Computational graphs are a way of expressing and evaluating a mathematical expression.*

Example 2.3.5. *Let us consider a function $f(x, y, z) = (x + y)(y + z)$. Now split f into smaller, primitive functions:*

$$\begin{aligned} u &= x + y \\ v &= y + z \\ w &= u \cdot v \end{aligned}$$

The corresponding computational graph is shown in Figure 2.5.

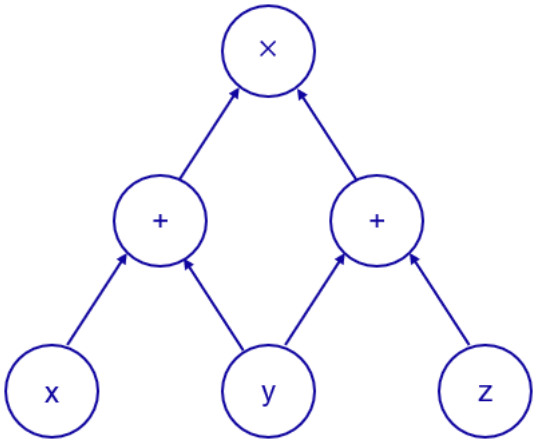

Figure 2.5: Computational graph - function $f(x, y, z) = (x + y)(y + z)$.

.

2.3.2 Forward mode of Automatic Differentiation

As we briefly mentioned, there are two modes of AD - forward and reverse. Here we will first take a look at the intuitively and technically simpler mode which is the forward mode of AD. We have seen in the AD introduction section 2.3.1 that the idea behind AD relies on the chain rule. Also we described that a sweep of forward AD evaluates an action of the Jacobian matrix on a vector in input space. Here, we expand that idea a bit further. Along with the chain rule, forward AD depends on the definition of the dual numbers. That is why, before we dive into the technical details of the forward mode, we start with the introduction of those. The theory and implementation is based on [12] and [7]. More details can be also found in [5].
As mentioned above, first we have to define the dual numbers and study its properties. This concept was first described by W.K. Clifford in 1873, and since then it has been used in many different applications, most notably in differentiation, as we are about to see. The formal definition is the following:

Definition 2.3.6. ***(Dual numbers)***
Let $a, b \in \mathbb{R}$, and ϵ is infinitesimal. We define a dual number $z = a + b\epsilon$.

Remark 2.3.7. ***(Similarities with the complex numbers)***
A complex number is an element of a number system that contains the real numbers and a specific element denoted by i, called an imaginary unit, which satisfies the equation $i^2 = -1$. Every complex number can be expressed in the form $a + bi, a, b \in \mathbb{R}$. The set of

complex numbers is denoted by $\mathbb{C}$.
Similarly, we defined the dual numbers in Definition 2.3.6, where instead of an imaginary i, we introduced an element ϵ such that $\epsilon^2 = 0$. Thus, the dual numbers extend the definition of real numbers by adjoining one new element ϵ.

Example 2.3.8. ***(Matrix representation)***
Using matrices, dual numbers can be represented as

$$a + b\epsilon = a\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix} + b\begin{bmatrix}0 & 1\\ 0 & 0\end{bmatrix}.$$

The sum and product of the dual numbers are then calculated with ordinary matrix addition and multiplication. This correspondence is analogous to the usual matrix representation of the complex numbers:

$$a + bi = a\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix} + b\begin{bmatrix}0 & -1\\ 1 & 0\end{bmatrix}.$$

Like the complex numbers, the dual numbers also satisfy basic arithmetic operations, which is proven in the following proposition:

Proposition 2.3.9. *Let $z = a + b\epsilon$ and $w = c + d\epsilon$ be two dual numbers. It is valid:*

i) $z + w$ *is a dual number and* $z + w = (a + c) + (b + d)\epsilon$.

ii) $z \cdot w$ *is a dual number and* $z \cdot w = ac + (ad + bc)\epsilon$.

iii) if $c \neq 0$, *then* $\frac{z}{w}$ *is a dual number and* $\frac{z}{w} = \frac{a}{c} + \frac{bc - ad}{c^2}$.

Proof. All the proofs follow directly from 2.3.6:

i) $z + w = a + b\epsilon + c + d\epsilon = (a + c) + (b + d)\epsilon$.

ii) $z \cdot w = (a + b\epsilon)(c + d\epsilon) = ac + ad\epsilon + bc\epsilon + \underbrace{bd\epsilon^2}_{=0} = ac + (ad + bc)\epsilon$.

iii) $\frac{z}{w} = \frac{a + b\epsilon}{c + d\epsilon} = \frac{a + b\epsilon}{c + d\epsilon} \cdot \frac{c - d\epsilon}{c - d\epsilon} = \frac{ac - ad\epsilon + bc\epsilon - bd\epsilon^2}{c^2 - d^2\epsilon^2} = \frac{a}{c} + \frac{bc - ad}{c^2}$.

□

Now consider a polynomial

$$P(x) = a_0 + \sum_{n=1}^{N} a_n x^n$$

and a dual number $z = x + \epsilon$. The polynomial P in the indeterminate z has the form:

$$\begin{aligned} P(z) = P(x+\epsilon) &= a_0 + \sum_{n=1}^{N} a_n (x+\epsilon)^n \\ &= a_0 + \sum_{n=1}^{N} a_n x^n + \epsilon \sum_{n=1}^{N} a_n n x^{n-1} \\ &= P(x) + \epsilon \frac{dP}{dx}(x). \end{aligned}$$

The third equality follows from the binomial theorem. Because of the fact that $\epsilon^2 = 0$, we are left only with the first two terms. Last equality follows from the formula for the first derivative of the polynomial.
The fact that $\epsilon^2 = 0$ suggests that the dual numbers can be used to differentiate functions, since in analogy to an infinitesimal dx, quantities of order dx^n with $n \geq 2$ are usually neglected. It turns out that this reasoning is correct. We proved this easily for polynomials. Then, via the Taylor Series, it can be generalized for any analytic function. So, by extending a real function to a dual function one can numerically obtain its derivatives. This serves as a motivation for the following definition:

Definition 2.3.10. *Let $z = a + b\epsilon$ and $f\colon \mathbb{R} \longrightarrow \mathbb{R}$ differentiable in a. We extend the definition of the function f to dual numbers by defining:*

$$f(a + b\epsilon) = f(a) + f'(a)b\epsilon.$$

Proposition 2.3.11. *Definition 2.3.10 is compatible with the properties of derivatives:*

i) if $f(x) = g(x)h(x)$, then $f(a + b\epsilon) = g(a)h(a) + (g'(a)h(a) + g(a)h'(a))b\epsilon$.

ii) if $f(x) = g(h(x))$, then $f(a + b\epsilon) = g(h(a)) + (g'(h(a))h'(a))b\epsilon$.

Proof. Directly from Definition 2.3.10. □

Example 2.3.12. *If $f(x) = \sin(x)$, then*

$$f(x+\epsilon) = \sin(x+\epsilon) = \sin(x)\cos(\epsilon) + \cos(x)\sin(\epsilon) = \sin(x) + \epsilon\cos(x).$$

This follows directly from Taylor expansions of sin *and* cos *functions, and the fact that $\epsilon^2 = 0$.*

Now we see that by evaluating $f(x)$ in its dual form and setting $b = 1$ in Definition 2.3.6 we are able to recover both the function value $f(a)$ as well as its evaluated derivative $f'(a)$ in the form of the coefficient in front of ϵ! Finally, because of 2.3.11 ii) we can easily

propagate gradients across the layers of computation simply by multiplying derivatives with each other.
Now imagine a function with n inputs, $f : \mathbb{R}^n \longrightarrow \mathbb{R}$. Let us remember remark 2.3.2. Notice that the right choice of u allows us to compute a partial derivative of f with respect to one input. Specifically, if $\{e_1, \ldots, e_n\}$ is a canonical basis for $\mathbb{R}^n$, let $u = e_i$. Then:

$$\nabla f_i = \nabla f \cdot u = \nabla f \cdot e_i = \frac{\partial f}{\partial x_i}. \tag{2.9}$$

Here ∇f_i denotes the i^{th} element of the gradient ∇f. Hence we can compute the full ∇f in n forward sweeps. Naturally, we do not compute (2.9) by doing a vector operation, as it would require us to already know ∇f. Instead, forward mode of AD computes a directional derivative at the same time as it performs a forward evaluation trace. Notice that same applies if we had a vector-valued function, $f : \mathbb{R}^n \longrightarrow \mathbb{R}^m$. In that case, n forward sweeps would result in a full $m \times n$ Jacobian matrix.

Example 2.3.13. *Find a derivative of $f(x, y, z) = xy\sin(yz)$ at the point a = (3,-1,2). First we set*

$$\begin{aligned} x &= 3 + \epsilon[1, 0, 0], \\ y &= -1 + \epsilon[0, 1, 0], \\ z &= 2 + \epsilon[0, 0, 1], \end{aligned}$$

which leads to

$$\begin{aligned} t = xy &= -3 + \epsilon[-1, 3, 0], \\ u = yz &= -2 + \epsilon[0, 2, -1], \\ v = \sin(u) &= \sin(-2) + \epsilon\cos(-2)[0, 2, -1], \end{aligned}$$

and finally

$$w = tv = -3\sin(-2) + \epsilon(-\sin(-2),\ 3\sin(-2) - 6\cos(-2),\ 3\cos(-2)).$$

Therefore, we see that

$$\begin{aligned} f(3, -1, 2) &= -3\sin(-2), \\ \nabla f(3, -1, 2) &= [-\sin(-2),\ 3\sin(-2) - 6\cos(-2),\ 3\cos(-2)]. \end{aligned}$$

Figure 2.6 shows a corresponding computational graph.

2.3.3 Implementing forward mode

To implement the forward mode and dual numbers we will make use of MATLAB's object-oriented features. Specifically, operator overloading introduced in section 1.4. What we

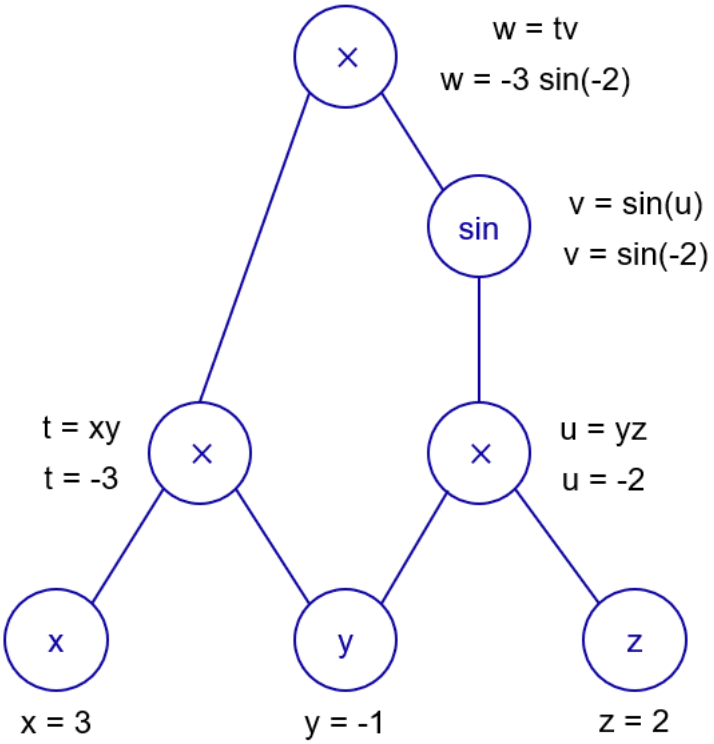

Figure 2.6: Computational graph - function $f(x, y, z) = xy \sin(yz)$.

want to do is represent a dual number with all of its properties. Therefore we need to make sure that all of our elementary operations (addition, multiplication, division) and functions (sin, cos, etc.) are redefined to accept dual numbers. Hence, we define a class:

```
classdef Dual
    properties
        value % function value
        deriv % derivative value or gradient vector
    end
    methods
        function obj = Dual(val, der)...
        function res = plus(a, b)...
        function res = minus(a, b)...
        function res = uminus(a)...
        function res = mtimes(a, b)...
        function res = mrdivide(a, b)...
        function res = sin(a)...
```

```
        function res = cos(a)...
        function res = mpower(a, b)...
        % ...
    end
end
```

Listing 2.1: class Dual definition.

Since any function can be written by combining these operations, it is then possible to write a simple code, such as

```
>> x = Dual(3, 1);
>> sin(x*x)
ans =
Dual with properties:
    value: 1.2364
    deriv: -15.9882
```

Listing 2.2: Forward AD - function $f(x) = x^2$.

In this display, first the Dual object is instantiated. The second argument in the constructor call indicates that we want to take a derivative with respect to variable x. This is particularly important when we are dealing with multivariate functions, as in the below example:

```
>> x = Dual(3,1);
>> y = Dual(-1);
>> z = Dual(2);
>> f = x * y * sin(y * z)
ans =
Dual with properties:
    value: 2.7279
    deriv: 0.9093
```

Listing 2.3: Forward AD - function $f(x, y, z) = xy\sin(yz)$.

Note that the variable f contains both the function value and a partial derivative with respect to the first coordinate, in this case that is $\frac{\partial f}{\partial x}$, both at the point $(3, -1, 2)$. Alternatively, we could have written

```
>> x = Dual([3 -1 2],[1 0 0]);
>> f = x(1) * x(2) * sin(x(2) * x(3));
```

Listing 2.4: Forward AD - alternative approach.

generating the same result. This approach specifically, will be used in coordinate system transformations (section 3.1). All of these examples are made possible with a careful implementation of a Dual constructor, as seen in the following snippet

```
function obj = Dual(val, der)
    obj.value = val;
    if nargin < 2
        obj.deriv = zeros(size(val));
    else
        obj.deriv = der;
    end
end
```

Listing 2.5: Constructor - class Dual.

The simplicity of the methods of interest is obvious. For most elementary functions, corresponding overloads are just one-liner functions that code calculus derivative rules, as in the following sin definition:

```
function res = sin(a)
    res = Dual(sin(a.value), cos(a.value) * a.deriv);
end
```

Listing 2.6: class Dual - overloading sin function.

Notice the silent use of the chain rule!
With two-argument operations we need to be more careful. Namely, in case of multiplication there are two possibilities: either both parameters are a Dual, or just one of them is (the second one is a scalar). Hence:

```
function res = mtimes(a, b)
    if ~isa(a, 'Dual') % a is a scalar
        res = Dual(a * b.value, a * b.deriv);
    elseif ~isa(b, 'Dual') % b is a scalar
        res = Dual(a.value * b, a.deriv * b);
    else % both a and b are Dual
        res = Dual(a.value * b.value,...
            a.deriv * b.value + a.value * b.deriv);
    end
end
```

Listing 2.7: class Dual - overloading operator *.

Aside from basic arithmetic operations and elementary functions, one could also redefine matrix manipulation functions available in Matlab. More specifically, functions such as transpose, horizontal and vertical concatenations, as well as indexing and assignment methods can be customized. This way we could take the full advantage of all MATLAB's power and features that are at disposal.

2.3.4 Reverse mode of Automatic Differentiation

The simplicity of forward AD comes with a big disadvantage which becomes evident when we wish to compute a partial derivative with respect to different parameters. In forward mode, doing so requires running the program (and with it the entire evaluation trace) again and again for each input variable. Obviously, this is very costly for a function with many inputs. A baby example for this is 2.3, where to get the entire gradient, we would have to run the same code (this time for variables y and z) multiple times. So natural question that pops up is can we do better? It turns out that we can, and that is where the reverse mode of AD comes into play. The idea here is to inverte the input-output roles of the vaiables. So instead of computing derivatives with respect to an input, we compute *adjoints* with respect to an output.

Definition 2.3.14. ***(Adjoint of a variable)***
The adjoint of a variable x with respect to another variable z is defined as

$$\bar{x} = \frac{\partial z}{\partial x}.$$

As briefly mentioned in section 2.3.1, for a initial vector $\bar{w} \in \mathbb{R}^m$ and a differentiable function $f\colon \mathbb{R}^n \longrightarrow \mathbb{R}^m$, reverse sweep of AD computes $J^T \cdot w$. Again, the right choice of w allows us to compute one row of the Jacobian matrix. Notice that, for $m = 1$, this row corresponds exactly to the gradient. Particularly, if we pick $w = f_j, \forall\, j = 1, \ldots, m$ where f_j is basis vector for $\mathbb{R}^m$,

$$J_i = J^T \bar{w}$$

we get the full $m \times n$ Jacobian matrix. We already see why the use of reverse AD instead of forward AD would be beneficial in a variety of cases, i.e when $n \gg m$. But how does it work exactly?
First, we make a forward pass through the computational graph and record the intermediate (auxiliary) variables. Then we work our way down the graph and record the adjoints using the chain rule and recorded variables. This is illustrated in Algorithm 3. To better understand this process and give a bit of intuition, the following example is presented.

Example 2.3.15. *We will use the same function as in Example 2.3.13 - $f(x, y, z) = xy\sin(yz)$. First, we do a forward sweep and record the intermediate variables. This is illustrated in Figure 2.6. Observe that we will use the variables in this figure to compute the adjoints. Now we have to do a reverse sweep by working our way down the graph, starting at the root variable w. Remember, we wish to calculate all the adjoints of the output variable w. We start with*

$$\bar{w} = \frac{\partial w}{\partial w} = 1,$$

which is the seed for the reverse pass. Next, we calculate each of the adjoints that w directly or indirectly depends on using the chain rule and basic derivative rules of calculus.

$$\begin{aligned}
\bar{v} = \frac{\partial w}{\partial v} &= t = -3, \\
\bar{t} = \frac{\partial w}{\partial t} &= v = \sin(-2), \\
\bar{u} = \frac{\partial w}{\partial u} = \frac{\partial w}{\partial v}\frac{\partial v}{\partial u} &= \bar{v}\cos(u) = -3\cos(-2), \\
\bar{z} = \frac{\partial w}{\partial z} = \frac{\partial w}{\partial u}\frac{\partial u}{\partial z} &= \bar{u}y = 3\cos(-2), \\
\bar{y} = \frac{\partial w}{\partial y} = \frac{\partial w}{\partial t}\frac{\partial t}{\partial y} + \frac{\partial w}{\partial u}\frac{\partial u}{\partial y} &= \bar{t}x + \bar{u}z = 3\sin(-2) - 6\cos(-2), \\
\bar{x} = \frac{\partial w}{\partial t}\frac{\partial t}{\partial x} &= \bar{t}y = -\sin(-2).
\end{aligned}$$

Notice that in every step the calculated intermediate variables and their values are used. Finally,

$$\begin{aligned}
f(3,-1,2) = w &= -3\sin(-2), \\
\nabla f = \begin{bmatrix} \bar{x} \\ \bar{y} \\ \bar{z} \end{bmatrix} &= \begin{bmatrix} -\sin(-2) \\ 3\sin(-2) - 6\cos(-2) \\ 3\cos(-2) \end{bmatrix}
\end{aligned}$$

2.3.5 Implementing reverse mode

The goal is to do the exact same thing that we did in Example 2.3.15 - store a computational graph in the forward sweep, and use its intermediate values to compute the Jacobian. To do a reverse pass, we have to be able to access two things:

1. The computational graph, and
2. The numerical values of the intermediate variables on that graph.

More precisely, during the forward sweep we build up an internal representation of the computation, and then use it to perform a reverse sweep. This internal representation will be stored in a persistent memory arena, which will be referred to as the *tape*. Again, operator overloading is used. The idea here is that with each new operation, we create new nodes by appending them to an existing, growable array. Notice, this array is actually the *tape*. Also, an integer *index* will be assigned to each node, so we can refer to it in this array. Intuitively, we could think of it as a pointer to another node. Additionally, to connect the nodes, each one will also store indices to their parent node(s) to represent the

dependencies. Conceptually, Figure 2.7 shows how this representation would look like for the function in Example 2.3.15. Note that there is a similarity with the graph 2.6. More

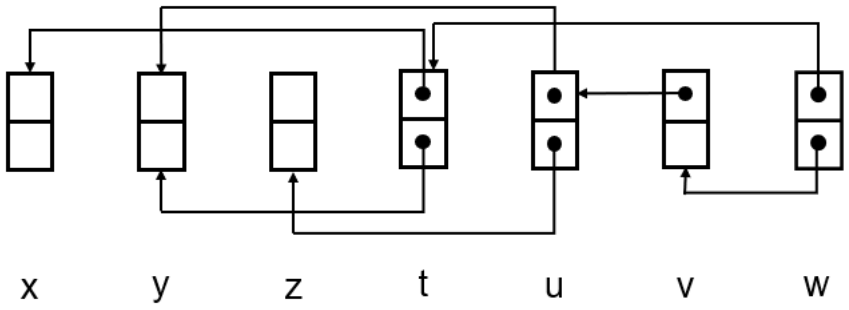

Figure 2.7: Representation of the computational graph in memory - function $f(x, y, z) = xy\sin(yz)$.

generally, given a function $f\colon \mathbb{R}^n \longrightarrow \mathbb{R}^m$, the tape can be divided into 3 parts, as shown in 2.10. At the beginning there are always n input variables created before any operation or evaluation. Following are intermediate variables created and appended to the tape with each new operation. Lastly, there are m variables which represent the output and serve as a root for the reverse sweep.

$$\begin{aligned} \text{Input variables} &\quad \begin{cases} \bar{\mathrm{v}}_1 \\ \vdots \\ \bar{\mathrm{v}}_n \end{cases} \\ \text{Intermediate variables} &\quad \begin{cases} \bar{\mathrm{v}}_{n+1} \\ \vdots \\ \bar{\mathrm{v}}_{n+m} \end{cases} \\ \text{Output variables} &\quad \begin{cases} \bar{\mathrm{v}}_{n+r+1} \\ \vdots \\ \bar{\mathrm{v}}_{n+r+m} \end{cases} \end{aligned} \tag{2.10}$$

To make all of this work properly, we need 3 types of objects:

1. class Node - to store weights (derivatives) and indices to parent nodes (dependencies).

2. array Tape - to store the nodes generated by the evaluation procedure.

3. main class reverseAD - to overload the operators and build the tape.

Let us start with class Node. Its definition is as follows:

```
classdef Node
    properties
        weights
        dependencies
    end
    methods
        obj = Node(w, deps)
    end
end
```

Listing 2.8: class Node definition.

For better understanding on how it works, below is an example on how the node for the variable $a = x * y$ would look like:

```
Node {
    weights: [
        y.value, % ∂a/∂x
        x.value, % ∂a/∂y
    ],
    deps: [ x.index, y.index ]
}
```

Listing 2.9: Node variable a = x * y.

The nodes themselves are stored in a common array - *tape*. Recall that the tape can be thought of as a record of all the operations performed during the evaluation procedure, which in turn contains all the information required to compute its gradient when read in reverse. Because of that, we will define a new class to perform those operations and build the tape. Below is the class definition:

```
classdef reverseAD
    properties
        value
        index
    end
    methods
        obj = reverseAD(val)...
        res = plus(a, b)...
        res = mtimes(a, b)...
        % all the other overloaded operations
        % ...
```

```
    end
end
```

Listing 2.10: class reverseAD definition.

Here's how it works: each operation creates a new instance of the reverseAD class, and assigns the property *value* to the result of the operation. At the same time, function *add_to_tape* is called in every overloaded method of the class. Its job is to create new instances of class *Node*, and append them to the tape. Implementation can be found in Algorithm 1. Property *index*, on the other hand, is incremented automatically. This means

Algorithm 1: function add_to_tape

Input: Weights and indices obtained from performed operation.
Data: tape = Array of Nodes.
1 Create new Node *node* with properties weights and indices.
2 Push *node* to the *tape*.

that internally we keep track of the number of nodes created. Note that number of nodes is equal to the size of the tape since each node is appended to the tape when created. For better understanding, let's take a look at the code of the constructor and a sin function:

```
% constructor
function obj = reverseAD(val)
    obj.value = val;
    obj.index = reverseAD.increment_counter();
end

function res = sin(a)
    res = reverseAD(sin(a.value));
    res.add_to_tape(cos(a.value), a.index);
end
```

Listing 2.11: class reverseAD - constructor and overloaded sin.

As a recap, main tasks of the class reverseAD are:

1. Calculate the expressions.
2. Create new nodes.
3. Build the tape.

Algorithm 2: Reverse sweep

Input: w - seed variable for the reverse sweep.
Output: *gradient* - array containing all adjoints of a variable w.
Data: *tape*.
`// w is of type reverseAD`
$n \leftarrow size(tape)$.
Initialize *gradient* as an array of zeros, size n.
Initialize the root for the reverse sweep: `//` $\partial w/\partial w = 1$.
$gradient(w.index) \leftarrow 1$.
for $i \leftarrow n$ *to* 1 *by* -1 **do**
    foreach dep *in* $tape(i).dependencies$ **do**
        $gradient(dep) \leftarrow gradient(dep) + tape(i).weights(j) * gradient(i)$.

Finally, after executing a function or program of interest and simultaneously building a tape, we are able to perform a reverse sweep. Behind the curtain, chain rule is used to calculate the adjoints. Algorithm 2 shows the implementation of the reverse pass. At last, Algorithm 3 shows us a full implementation of the reverse mode of Automatic differentiation. Notice that to compute the entire $m \times n$ Jacobian matrix, we have to run the Algorithm 3 m times.

Algorithm 3: Reverse AD

Data: Target function f.
Input: List of input parameters $inputs$.
Output: Gradient row of the Jacobian matrix.
foreach *parameter in inputs* **do**
    Make an instance of reverseAD object with value *parameter*.
Initialize array *tape*.
foreach *operation in f* **do**
    Perform *operation*.
    Store its result in a new instance of reverseAD object.
    Call routine *add_to_tape*.
Choose *output* variable to get the desired row of the Jacobian.
Call routine *reverse_sweep* with the argument *output* and save the result to *gradient*.

2.3.6 Forward or reverse?

By now one should already see a difference between forward and reverse mode of AD. Suppose that for a function $f\colon \mathbb{R}^n \longrightarrow \mathbb{R}^m$ we wish to compute the entire $m \times n$ Jacobian matrix. The question that we want to answer is which mode of AD should we use? We already know that one sweep of the forward mode computes one column of the Jacobian matrix, and the reverse mode on the contrary computes one row per sweep. Therefore,

Hunch 2.3.16. *Forward mode performs better when $n < m$.*

There is an important thing that should be mentioned here: when comparing the two modes, we ignore the overhead of building the expression graph. More on that will be discussed in the subsection 2.3.7. That said, with the relatively small overhead, the performance of reverse mode AD is superior when $n \gg m$. A proof of that statement is shown in Table 2.1 where the relative runtime of the reverse and forward mode are compared. Reverse mode is extremely useful and highly applicable to high-level modeling.

Dimension of input, n	2	4	8	16
Relative runtime	0.06362	0.58879	3.10207	8.1683

Table 2.1: Relative runtime to compute a Jacobian matrix with reverse mode when compared to forward mode. The table summarizes results of measuring runtimes by differentiating $f\colon \mathbb{R}^n \longrightarrow \mathbb{R}^3$ with various number of inputs n.

2.3.7 Optimizing the performance of reverse mode

There is a big setback time-wise for the reverse mode of AD and that's tape building. The experiment on the same data as in Table 2.1 shows that the ratio of time to build the tape and to build a Jacobian from it is quite big. More precisely:

$$\frac{\textit{Time to build the tape}}{\textit{Time to build the Jacobian from that tape}} \approx 13.$$

Because of that, we don't want to build the tape from scratch every time we wish to compute the Jacobian. The goal is to build the tape (preferably once) and reuse it in multiple reverse sweeps. But there is a natural question that arises: if we have multiple points in which we wish to compute derivatives of the same function, the intermediate values should change from point to point, which which causes changes to the tape?
Luckily, there are solutions for this problem. One is the method called *retaping*. To be able to perform retaping, there is a one condition that has to be met. The control flow of

the function or a program must not change from sweep to sweep. If our target function contains conditional statements, those control the expression graph our program generates. Effectively, we have a different graph for each of the conditions. But if the control flow doesn't change, the expression graph stays the same and doesn't change from sweep to sweep. Therefore, a single tape for the expression graph can be employed to compute AD sweeps at multiple points. More thorough explanations and implementations of this method can be found in [17] or [2].
Another problem that could arise, especially in big-size problems, is peak memory usage. Here, the computational graph gets large and complicated, which takes a toll on memory and with it, on the efficiency of entire program. There are solutions for this specific problem as well. The two possible approaches are: *checkpointing* and reducing the number of operations in a function. More on these and other similar methods can be found in [10].

2.3.8 Alternative implementations

There are multiple ways of implementing AD. Reverse mode especially has its share of different approaches. When it comes to operator overloading approach, the basic idea is always the same; build the tape and then traverse it backwards to get the desired gradient or Jacobian. That said, the tape itself, its implementations and memory representations is a different story. One way that is widely used is to represent the tape as 3-address code, where in addition to intermediate values and indices that we stored (subsection 2.3.5), one stores the performed operations in a third array. So the difference here is that rather than having a one array that stores the intermediate values, the tape is a structure that holds three different arrays. This is nicely illustrated in Figure 2.8. The two arrays storing values and indices can be intuitively thought of as equal to the weights and dependencies in our class Node, although there are subtle differences. To build the operation codes array, the programmer needs to take care of assigning the codes, i.e. integers to different operation. For instance, addition could have the code 1, multiplication 2, sin function 10 and so on. After building the tape this way, the reverse sweep is carried out by a simultaneous interpretation of the three arrays. More details on this approach is in [7].

Aside from using the operator overloading feature (or object-oriented programming in general), there is a completely different way of implementing AD: Source transformation. This is a more classic approach as it uses a preprocessor, which applies differentiation rules to the source code of a computer program that implements the target function. Essentially, it generates a new source code, which calculates the derivatives. Finally, both source codes are then run and evaluated together producing the desired result. There are severe limitations to this approach. One being that it can use the information available only at compile time. This prevents a programmer from using sophisticated programming features such as while loops, object-oriented features, etc. Therefore, implementing source transforma-

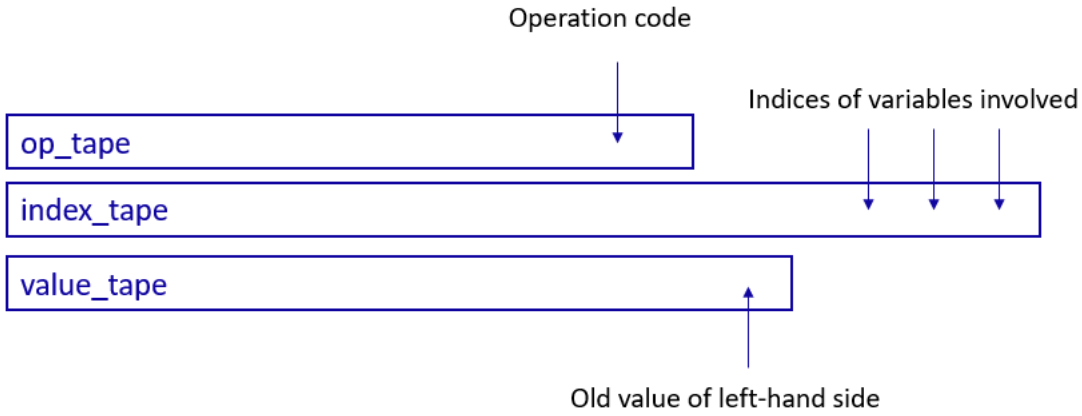

Figure 2.8: Representation of the tape - 3-address code approach.

tion takes a considerable amount of effort and that's why operator overloading is today the appropriate technology. More on this topic can be found in [1] and [3].

Chapter 3

Results and examples

In this chapter the results collected from an application to real data and examples will be presented. One of the main focuses will be on efficiency of the methods. More precisely, average runtimes will be taken to see which methods perform best in a variety of cases. Also, the tests presented here will also give an insight into how good numerical methods are in the specific cases we are going to look at. These results are obtained on the specific coordinate system transformations that were described in Section 2.1.

3.1 Jacobian in context of Coordinate systems transformation

As described in section 2.1, the goal is to compute the Jacobian matrix J of a function $f\colon \mathbb{R}^n \longrightarrow \mathbb{R}^3$ (see Figure 2.3). Since this transformation is close to linear, except of course the rotation which is function of sin and cos, we expect the cumulative error in the numerical methods to be quite small. An important thing to note here is that all the discrepancies and comparisons in this subsection are obtained with respect to AD, since we know that the latter method yields the exact solution. First, we will make an accuracy comparison which will give us an insight into how good and correct the numerical methods are. After that, the results of the time measurement experiments will be presented. Here we will get a real glimpse into how efficient the two AD methods are. Finally, we will wrap up this section with a conclusion and give an opinion and advice on which method would be best utilized in this specific framework.

3.1.1 Accuracy comparison

Before diving into the details of the experiments performed, there is one thing that we should take care of, and it concerns the step-size h when using the numerical methods de-

scribed in section 2.2.1. So first things first, the optimal step size h needs to be chosen for the finite step and complex step approximation methods. Figure 2.4 shows us that expected optimal step size for FDA methods should be around 10^{-5} or 10^{-6}.

That assumption proves correct, as shown in Figure 3.1. Here, the errors for different

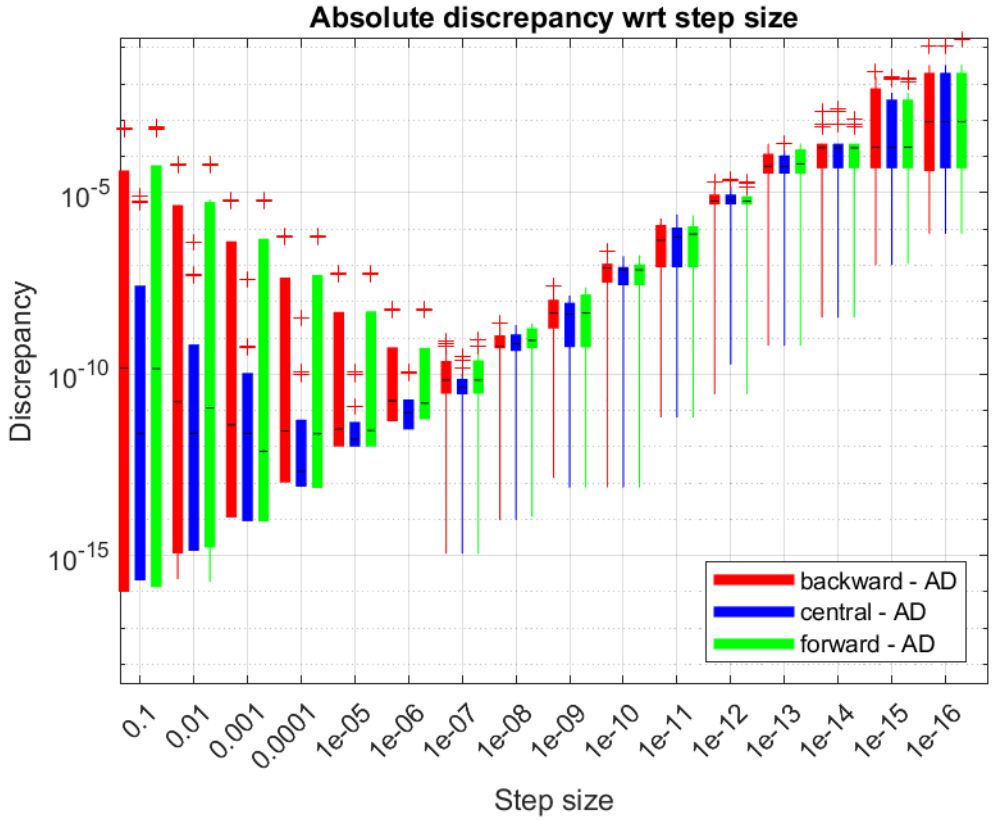

Figure 3.1: Boxplot shows the cumulative error of FDA methods with respect to step size h. It summarizes the results from an experiment for a Jacobian of a function $f\colon \mathbb{R}^{20} \longrightarrow \mathbb{R}^3$. All the discrepancies from all input parameters are collected and grouped by step size h.

input parameters are accumulated with respect to the step size h. There are 16 different step size values chosen, the smallest one being 10^{-16} and the biggest one 0.1. It is obvious that overall best results are obtained for 10^{-5} and 10^{-6}, as expected. Thus, for future tests we choose an optimal step size $h_{FDA} = 10^{-6}$.

The same question pops up when working with complex step approximation. Here, the results are way more accurate than for the FDA, which is expected since the dominant

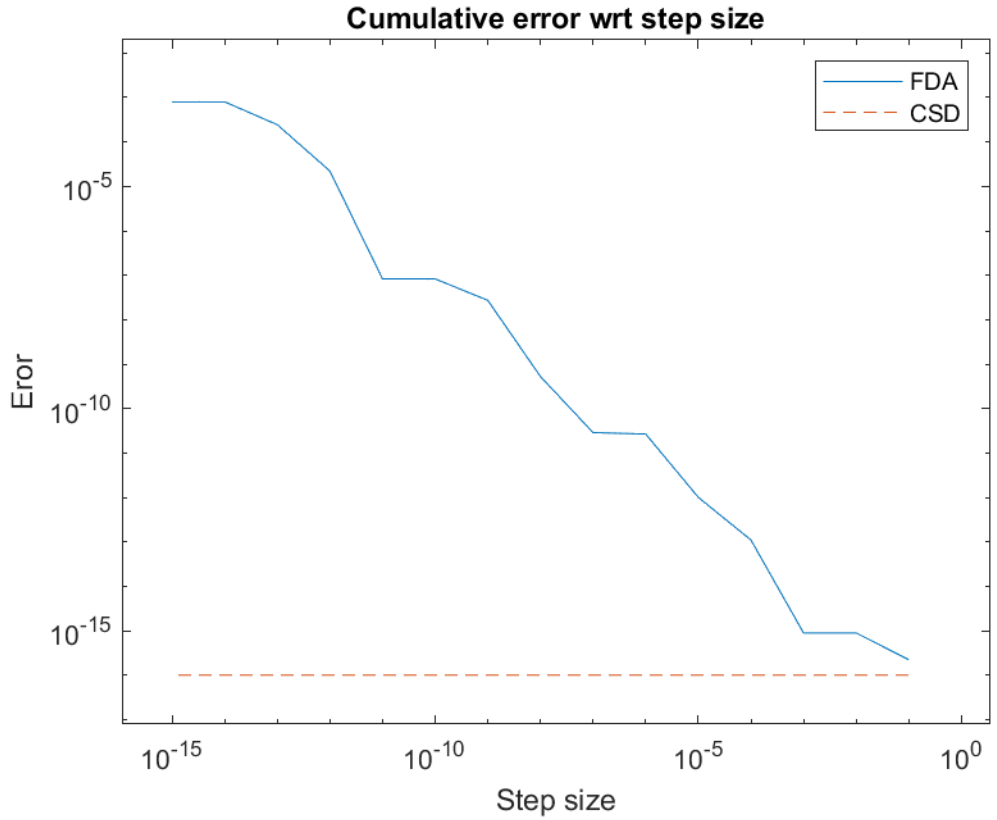

Figure 3.2: Cumulative error wrt to step size - Finite step and complex-step approximation comparison.

error term is on the order of h^2. Also, since CSD isn't a difference based method as the FDA methods are, there is no "large number" + "small number" additions. Thus, the errors due to floating point precision are almost non-existent. One example where this is especially important and where CSD has the upper hand on FDA methods are the linear functions. Suppose that we have a linear function f. Because of the above mentioned addition, smallest error when calculating a derivative of f is when we have the biggest step size, i.e. $h = 0.1$. Luckily, the error when $h = 10^{-6}$ is acceptable. This is shown in Figure 3.2. With that said, using the data from Table 3.1, we choose step size $h_{CSD} = 10^{-8}$.
Now that the step sizes are chosen, we are ready to do the final comparison in terms of accuracy. Both the absolute and relative errors will be taken into account. Again, we make use of MATLAB's boxplot to nicely represent the discrepancies. From Figures 3.3 and 3.4 one can conclude that the numerical methods perform very well. To be more specific, we can see that the error in case of CSD is almost non-existent, while for FDA methods is also

Step size	Maximum discrepancy wrt AD
10^{-1}	2.28×10^{-5}
10^{-2}	2.28×10^{-7}
10^{-3}	2.28×10^{-9}
10^{-4}	2.28×10^{-11}
10^{-5}	2.28×10^{-13}
10^{-6}	2.28×10^{-15}
10^{-7}	5.55×10^{-17}
10^{-8}	5.55×10^{-17}
10^{-9}	5.55×10^{-17}
10^{-10}	5.55×10^{-17}
10^{-11}	5.55×10^{-17}
10^{-12}	5.55×10^{-17}
10^{-13}	1.11×10^{-16}
10^{-14}	1.11×10^{-16}
10^{-15}	1.11×10^{-16}
10^{-16}	5.55×10^{-16}

Table 3.1: Maximum discrepancy of CSD method for each step size h.

acceptable. Table 3.2 shows that the maximum error for FDA is in range of 10^{-9} and 10^{-10}, while for CSD that number goes to -15.

Method	CSD	BFDA	CFDA	FFDA
Maximum discrepancy	2.14×10^{-10}	6.35×10^{-9}	1.18×10^{-10}	6.36×10^{-9}

Table 3.2: Maximum absolute discrepancy of numerical methods when compared to AD.

3.1.2 Timing and efficiency

Before diving into runtimes of the different methods, there is one important thing to mention. Let us recall the discussion from Subsection 2.3.7. There we proved that there is a setback when using the reverse mode - tape building. Therefore, we ignore the overhead of building the expression graph when measuring the runtime of reverse AD, having in mind that with the relatively small overhead the performance of reverse AD is superior when compared to the other methods, as we are about to prove.

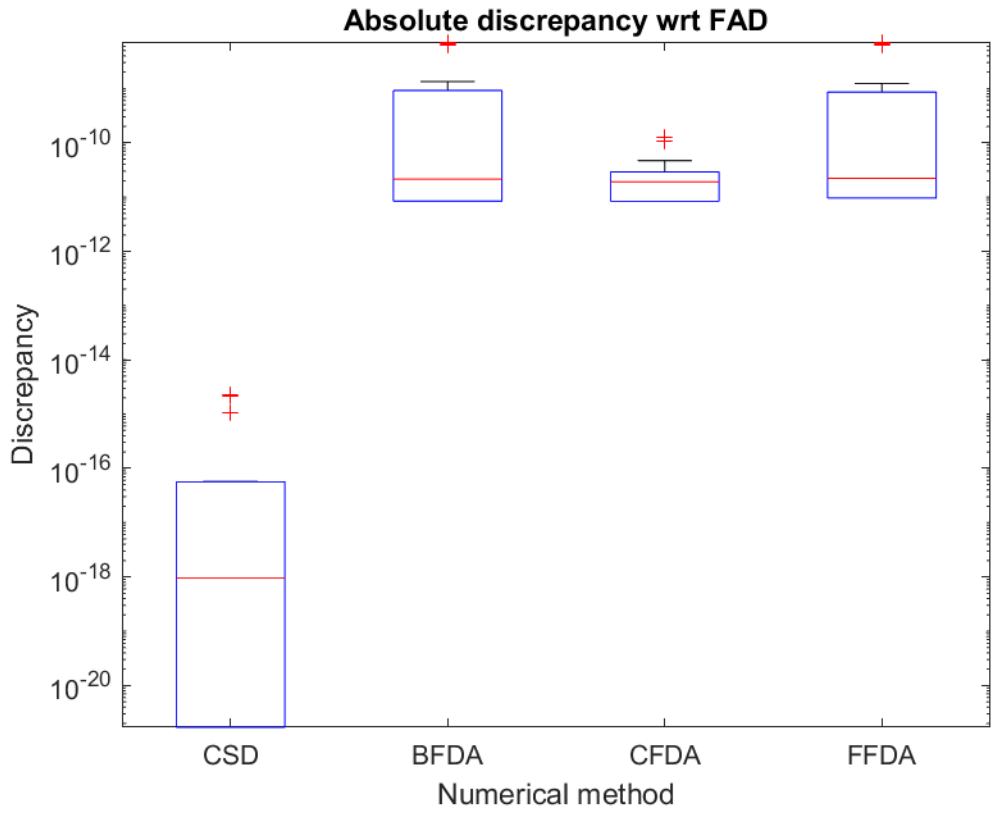

Figure 3.3: Absolute discrepancy - numerical methods when compared to AD.

As far as the numerical methods go, we expect CSD to be slower then FDA methods. Reason for that being that the complex method performs complex arithmetic as well as unused computations. For instance, the real part is calculated but never used.
To measure runtime, we calculate the Jacobian 1000 times using each method, and then take the average. Results are shown in Table 3.3. Immediately we see that forward mode

Method	BFDA	CFDA	FFDA	CSD	FAD	RAD
Time per call (in milliseconds)	2.4	4	2.3	2.9	13	1.2

Table 3.3: Runtimes to compute the Jacobian. This experiment is done by calculating the Jacobian matrix of a function $f\colon \mathbb{R}^{20} \longrightarrow \mathbb{R}^3$ 1000 times. Results in the table show the average runtime for each of the different methods.

of AD does not yield acceptable results in terms of efficiency. That is an expected behavior (see the discussion in subsection 2.3.6). On the other hand, reverse mode gives the best

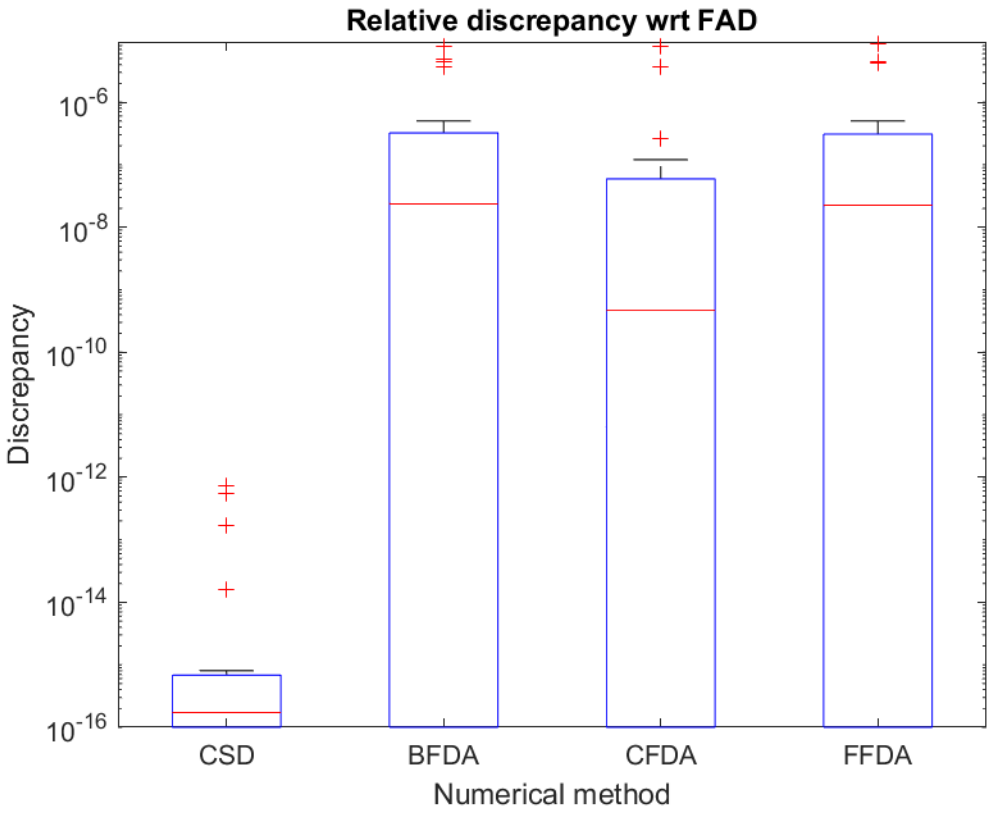

Figure 3.4: Relative discrepancy $\frac{\text{FDA/CSD - AD}}{\text{AD}}$.

results for the specified parameter set at the specific reference point. The results for the numerical methods are as expected - with BFDA and FFDA yielding the best result.

3.1.3 Conclusion

Looking back at the conducted tests, there are a few things to conclude. First, it is obvious that forward mode of AD is not a good option for this specific problem. Even though exact, forward AD is inefficient and takes too much time to build the Jacobian matrix. On the other hand, reverse AD seems to solve the time efficiency issue, since it is the fastest method of all. With that said, reverse AD should be the obvious choice. The only problem with reverse AD is the software effort that needs to be made in order to make it work properly. As seen in the subsection 2.3.5, it takes a considerable amount of time and effort to implement it and integrate it into your own system. For that reason, one could opt in for a simpler solution which makes use of numerical methods. As shown in the discussions above, numerical methods are, in this specific case, really accurate as well as efficient.

Taken that into account, with its straightforward implementation, FFDA and BFDA seem like a good choice as well. One could even make the case for CSD, where the error is shown to be almost non-existent. To conclude this section, there is a choice to be made between reverse AD and first-order FDA methods. If one is willing to put in effort and time in implementing reverse AD, it is a no brainer. But in most cases and examples with different parameters, FDA methods can be more than acceptable, which makes them a good alternative.

Bibliography

[1] Michael Bartholomew-Biggs, Steven Brown, Bruce Christianson, and Laurence Dixon, *Automatic differentiation of algorithms*, Journal of Computational and Applied Mathematics **124** (2000), no. 1-2, 171–190.

[2] Bradley M Bell, *Cppad: a package for c++ algorithmic differentiation*, Computational Infrastructure for Operations Research **57** (2012), no. 10.

[3] Christian H Bischof and H Martin Bücker, *Computing derivatives of computer programs.*, Techn. rep., Argonne National Lab., IL (US), 2000.

[4] Rebecca M Brannon, *A review of useful theorems involving proper orthogonal matrices referenced to three dimensional physical space*, Albuquerque: Sandia National Laboratories (2002).

[5] Shaun A Forth, *An efficient overloaded implementation of forward mode automatic differentiation in matlab*, ACM Transactions on Mathematical Software (TOMS) **32** (2006), no. 2, 195–222.

[6] James E Gentle, *Matrix algebra*, Springer texts in statistics, Springer, New York, NY, doi **10** (2007), 978–0.

[7] Andreas Griewank and Andrea Walther, *Evaluating derivatives: principles and techniques of algorithmic differentiation*, SIAM, 2008.

[8] Serge Lang, *Calculus of several variables*, Springer Science & Business Media, 2012.

[9] Zhi Quan Luo and Paul Tseng, *Perturbation analysis of a condition number for linear systems*, SIAM Journal on Matrix Analysis and Applications **15** (1994), no. 2, 636–660.

[10] Charles C Margossian, *A review of automatic differentiation and its efficient implementation*, Wiley interdisciplinary reviews: data mining and knowledge discovery **9** (2019), no. 4, e1305.

[11] Cleve Moler, *Complex step differentiation*, 2013.

[12] Richard D Neidinger, *Introduction to automatic differentiation and matlab object-oriented programming*, SIAM review **52** (2010), no. 3, 545–563.

[13] Andy H Register, *A guide to matlab object-oriented programming*, Chapman and Hall/CRC, 2007.

[14] Larry F Thompson, *An introduction to lithography*, ACS Publications, 1983.

[15] John E. Tolsma and Paul I. Barton, *On computational differentiation*, Computers I& Chemical Engineering **22** (1998), no. 4, 475–490, ISSN 0098-1354, https://www.sciencedirect.com/science/article/pii/S0098135497002640.

[16] Šime Ungar, *Matematička analiza u r^n*, Golden marketing-Tehnička knjiga, 2005.

[17] Andrea Walther and Andreas Griewank, *Getting started with adol-c.*, Combinatorial scientific computing **181** (2009), 202.

[18] Stephen Wright, Jorge Nocedal, et al., *Numerical optimization*, Springer Science **35** (1999), no. 67-68, 7.

[19] Bausan Yuan, Susumu Makinouchi, and Hideyaki Hashimoto, *Stage control with reduced synchronization error and settling time*, 2001, US Patent 6,260,282.

Sažetak

Kao i mnoge industrije, fotolitografija zahtjeva preciznost i učinkovitost u računanju derivacija u svrhu optimizacije finalnog proizvoda. Točnije, korištena transformacija koordinatnih sustava je optimizirana ispravljanjem grešaka u ulaznim podacima, što zahtjeva najbolje moguće metode za računanje Jacobijeve matrice dane transformacije.
U ovom radu prezentirali smo razne metode za računanje derivacija u smislu transformacija koordinatnih sustava. Upoznali smo razlike u performansama i implentacijama različitih pristupa. Pokazalo se da jednostavnost numeričkih metoda nije bila kobna, pošto je točnost istih bila više nego zadovoljavajuća. Međutim, automatsko deriviranje nam je pokazalo da egzaktnost i efikasnost mogu ići jedno s drugim. Ta tvrdnja je bila posebice točna za obrnuti AD, koja se pokazala najbržom metodom od svih. Jedina manjkavost te metode je zahtjevnost tehničke implementacije koja je potrebna za izvedbu iste. Zbog toga je potrebno donijeti odluku - uložiti vrijeme i trud za malo poboljšanje u efikasnosti ili prihvatiti malo losiju metodu, u svrhu jednostavnije izvedbe.

Summary

Like many industrial applications, photolithography requires accurate and efficient derivative computation to optimize its resulting product. More specifically, its coordinate system transformation is optimized by error correction in input data, which requires the best possible method to compute the Jacobian matrix of a given transformation.
In this work we presented different methods for computing derivatives in the context of coordinate system transformations. We have seen the implementation, as well as the performance differences between the various approaches. It turned out that the simplicity of the numerical methods - finite step and complex step approximations - didn't come back to bite us, since accuracy wise they proved to be more then acceptable. With that said, AD showed us that an exact and efficient solution can come hand in hand. That was especially true when reverse AD was used as it proved to be the fastest method of all. One knock on the reverse AD is the technical effort required for its implementation, as it forces one to make a decision between a software effort and a slight improvement in efficiency or a simpler, but in general worse, solution.

Biography

I was born on September 4, 1997 in Pula, Croatia. After graduating from dr. Mate Demarin's primary school in Medulin, Coratia, I enrolled in the Pula Science and Mathematics High School. In 2016, I finished my high school education and enrolled in undergraduate studies in Mathematics at the Faculty of Science in Zagreb. In 2019, I obtained a bachelor's degree in Mathematics, *univ. bacc. math.*, and I continued with graduate studies in Applied Mathematics at the same faculty. During the second year of my graduate studies, I started working as a software developer in the company Mireo d.d. in Zagreb, Croatia, where I was a member of the navigation team for a year. After that, I moved to Eindhoven, Netherlands where I started an internship at the company ASML Netherlands BV, during which I conduct research, which results in this Master thesis.

Životopis

Rođen sam 4. rujna 1997. godine u Puli. Nakon završavanja Osnovne škole dr. Mate Demarina u Medulinu, upisujem Prirodoslovno-matematičku gimnaziju Pula. 2016. godine, završavam srednjoškolsko obrazovanje i upisujem preddiplomski studij Matematike na Prirodoslovno-matematičkom fakultetu u Zagrebu. 2019. godine, stječem titulu prvostupnika Matematike, *univ. bacc. math.*, te nastavljam s diplomskim studijem Primjenjena Matematika. Tijekom druge godine diplomskog studija, zapošljavam se kao software developer u kompaniji Mireo d.d. u Zagrebu, gdje godinu dana radim u navigacijskom timu. Nakon toga, odlazim u Nizozemsku gdje počinjem studentsku praksu u kompaniji ASML, tijekom koje obavljam istraživanje čiji je rezultat ovaj diplomski rad.