Danmarks Tekniske Universitet


# Bachelorprojekt

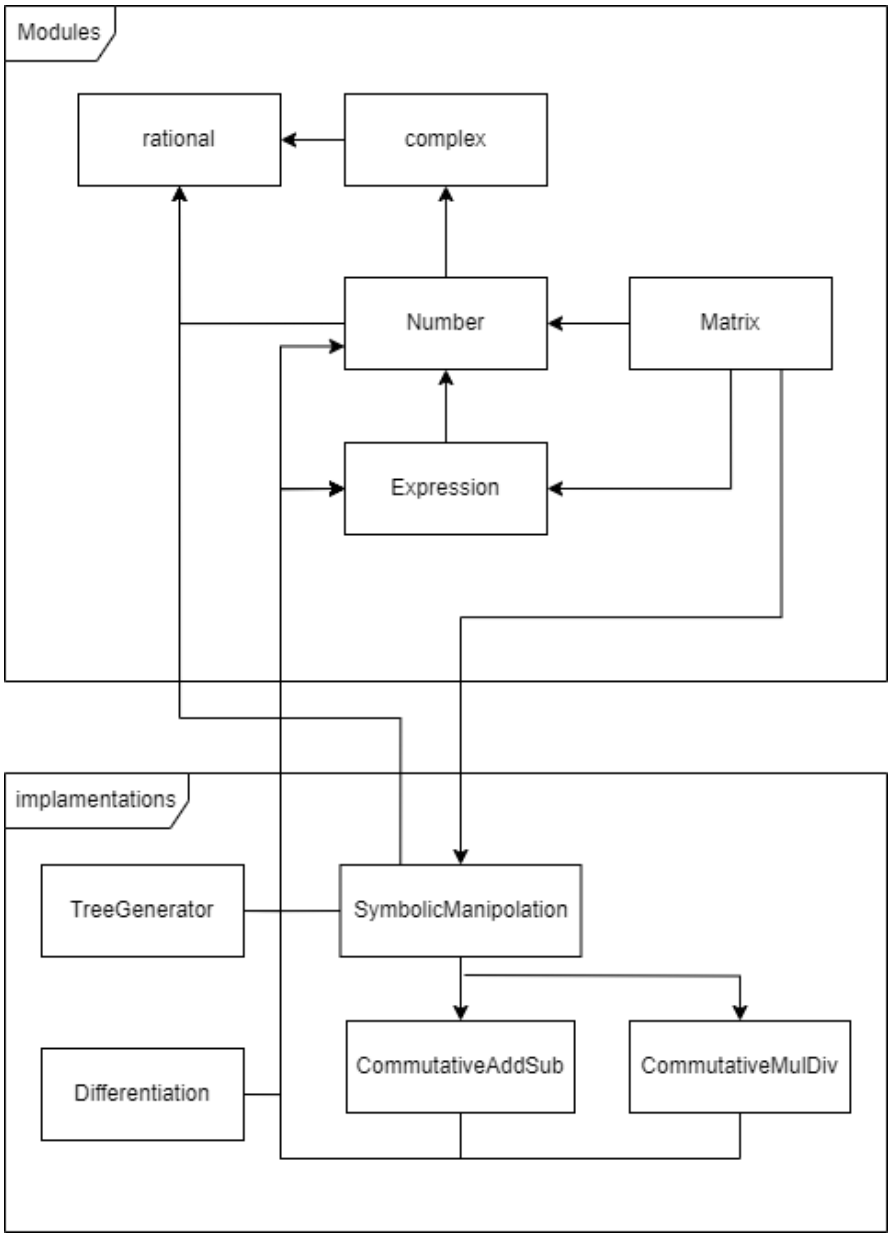
## Funktionel Modellering af Matematiske Systemer i F#

Jonas Dahl Larsen (s205829)

23. maj 2024

# Abstract

This project aims to guide the reader through a detailed modeling of a mathematical program in F#. By building everything from the ground up, the reader will be introduced to the basic concepts of functional programming, and how to use them to solve mathematical problems. The project aims to illustrate a strong correlation between the mathematical concepts and functional programming. Within the mathematical space, the project walks through building a number module consisting of integers, rational numbers, and complex numbers. Then, using the number module to express mathematical expressions, vectors, and matrices. Furthermore, the project will introduce the reader to property-based testing, which is used throughout the build-up to ensure the correctness of the program. The motivation behind the project was to show F# as a possible alternative to Python in fundamental mathematical courses at Denmark's Technical University. The project concludes that using functional programming can benefit the learning process of mathematical concepts.




Figur 1: Visuelt overblik over moduler og implementeringsfiler i programmet. $f_1 \rightarrow f_2$ indikere at $f_1$ åbner eller anvender funktioner fra $f_2$

# Forord

Denne rapport er udarbejdet som et 15 ECTS bachelorprojekt i Matematik og Teknologi ved Danmarks Tekniske Universitet. Arbejdet er udført under vejledning af lektor, Ph.D. Michael Reichhardt Hansen og lektor, Ph.D. Jakob Lemvig.

Ved siden af mit studie arbejder jeg som softwareudvikler hos Secure Spectrum Fondsmæglerselskab A/S, hvor vores kodebase er skrevet i Ruby[1], et dynamisk typet sprog ligesom Python. Denne erfaring danner grundlag for nogle af de konklusioner, vi kommer frem til vedrørende fordelene og ulemperne mellem F# og Python.

Programmet, som rapporten vejleder læseren igennem, kan findes på GitHub: `https://github.com/Larsen00/funktionsprogrammeringForIndledendeMatematik`

[1] [22] *Ruby*.

# Indhold

# 1 Introduktion

I 2023 besluttede Danmarks Tekniske Universitet (DTU) at anvende Python som et hjælpemiddel, der erstattede det matematiske softwareprogram Maple[2] i deres grundlæggende matematikkurser „01001 Matematik 1a“ og „01002 Matematik 1b“. I en tid, hvor programmeringssprog som Python dominerer i tekniske og videnskabelige miljøer[3], undersøger dette projekt fordele og ulemper ved funktionel programmering i matematiske sammenhænge. Funktionel programmering introducerer et andet perspektiv og alternative metoder, som kan berige og muligvis forbedre de studerendes forståelse af matematiske koncepter.

Projektet har til formål at belyse, hvorvidt funktionel programmering, specifikt gennem F#[4], kan anvendes til at opbygge og manipulere matematiske udtryk og systemer, der introduceres i den indledende matematiske undervisning. Ved at introducere læserne til grundlæggende såvel som avancerede funktioner og teknikker i F#, vil rapporten guide dem gennem opbygningen af funktionel modellering af matematiske systemer, der kan løse matematiske problemer fra de grundlæggende matematikkurser på DTU.

Rapporten vil først og fremmest dykke ned i konstruktionen af et specifikt modul for håndtering af symbolske matematiske udtryk og matrixmanipulering, samt deres anvendelser i forskellige matematiske kontekster. Projektets struktur og metodologi har til formål at give læseren en dybdegående forståelse af, hvordan funktionel programmering kan benyttes strategisk i matematiske discipliner, og hvordan det adskiller sig fra mere traditionelle imperative programmeringstilgange.

Gennem en systematisk tilgang til design og implementering af matematiske moduler vil rapporten udforske, hvordan matematiske principper kan integreres direkte i softwareudvikling gennem funktionel programmering. Dette vil ikke kun fremme en bedre forståelse af teoretiske koncepter og konstruktioner gennem praktisk anvendelse, men også demonstrere F#'s kapacitet og effektivitet i behandlingen af matematiske egenskaber.

[2] [11] *Maple*.
[3] [17] *Python: The most popular language*.
[4] [4] *fsharp.org*.

# 2 Fundamentale koncepter

Det forventes, at læseren har kendskab til programmering. Der gives derfor kun en kort beskrivelse af syntaks og notation, så læsere, der ikke er bekendt med F#, kan forstå de eksempler, der løbende vil forekomme i rapporten.

## 2.1 Introduktion til Funktions Programmering

F# er en del af ML (Meta Language) familien af funktionsprogrammeringssprog, som bygger på „typed lambda calculus“[5]. „Typed lambda calculus“ er en udvidelse af „lambda calculus“[6], som er et matematisk system. Det giver et formelt fundament for at definere og manipulere funktioner.

Vi begynder med at betragte funktionen for fakultet Ligning (1).

$$f(n) = \begin{cases} 1 & n = 0 \\ n \cdot f(n-1) & n > 0 \\ \text{undefined} & n < 0 \end{cases} \tag{1}$$

Et eksempel på en implementering af Ligning (1) i F# er givet i Listing 1.

```
1 // Fakultet i F#
2 let rec factorial n =
3     match n with
4     | 0             -> 1
5     | x when x > 0  -> x * factorial (x - 1)
6     | _             -> failwith "Negative argument"
```

Listing 1: Eksempel på Fakultet i F#

I F# anvendes let til at definere en ny variabel eller, i dette tilfælde, en funktion kaldet factorial. Næste nøgleord er rec, hvilket indikerer, at funktionen er rekursiv. Funktionen tager et argument $n$, og i linje 3 starter et match-udtryk. Her er $n$ vores udtryk, og efter with begynder en række mønstre. Med tilhørende udtryk, der svare til de tre tilfælde i Ligning (1).

I F#, er det som udgangspunkt ikke nødvendigt at anvende parenteser som i andre programmeringssprog. Derfor vil de kun blive anvendt, hvor det er passende gennem rapporten, typisk i sammenhænge med kædning af funktioner. For at undgå brugen af parenteser kan man i F# benytte pipe-operatorerne, $|>$ og $<|$, som fører resultatet fra en udledning direkte ind i den næste funktion. Nedenstående eksempel viser tre ækvivalente udtryk, der demonstrerer anvendelsen af disse operatorer.

```
> factorial (factorial 3);;
val it: int = 720

> factorial <| factorial 3;;
val it: int = 720
```

[5] [26] *Typed lambda calculus*.
[6] [10] *lambda calculus*.

```
> factorial 3 |> factorial;;
val it: int = 720
```

Listing 2: Eksempel på anvendelse af pipe-operatorer i F# ved udregning af $(3!)! = 6! = 720$.

Vi kan beskrive evalueringen af udtrykket i Listing 2 som følgende, hvor $e_1 \rightsquigarrow e_2$ betyder, at $e_1$ evalueres til $e_2$:

$$
\begin{aligned}
&\text{factorial}(\text{factorial}(3)) \\
&\quad \rightsquigarrow \text{factorial}(3 \times \text{factorial}(3-1)) \\
&\quad \rightsquigarrow \text{factorial}(3 \times 2 \times \text{factorial}(2-1)) \\
&\quad \rightsquigarrow \text{factorial}(3 \times 2 \times 1 \times \text{factorial}(1-1)) \\
&\quad \rightsquigarrow \text{factorial}(3 \times 2 \times 1 \times 1) \\
&\quad \rightsquigarrow \text{factorial}(6) \\
&\quad \rightsquigarrow 6 \times \text{factorial}(6-1) \\
&\quad \rightsquigarrow 6 \times 5 \times \text{factorial}(5-1) \\
&\quad \rightsquigarrow 6 \times 5 \times 4 \times \text{factorial}(4-1) \\
&\quad \rightsquigarrow 6 \times 5 \times 4 \times 3 \times \text{factorial}(3-1) \\
&\quad \rightsquigarrow 6 \times 5 \times 4 \times 3 \times 2 \times \text{factorial}(2-1) \\
&\quad \rightsquigarrow 6 \times 5 \times 4 \times 3 \times 2 \times 1 \times \text{factorial}(1-1) \\
&\quad \rightsquigarrow 6 \times 5 \times 4 \times 3 \times 2 \times 1 \times 1 \\
&\quad \rightsquigarrow 720
\end{aligned}
$$

## 2.2 Typer

I F# har alle udtryk, inklusiv funktioner, en defineret type. Typen for funktionen i Listing 1 er $int \to int$. Det betyder, at det ikke er muligt at kalde funktionen med et argument, der ikke er af typen $int$.

I matematik kan vi beskrive funktioner som afbildninger ved brug af domæner og co-domæner på følgende måde:

$$
\begin{aligned}
f &: \mathbb{R} \to \mathbb{R} \\
g &: \mathbb{R}^n \to \mathbb{R}
\end{aligned}
$$

De tilsvarende funktioner i F# vil have typerne:

$$
\begin{aligned}
f &: float \to float \\
g &: float * float * \ldots * float \to float
\end{aligned}
$$

Hvor typekonstruktøren „$*$“ svarer til det kartesiske produkt, og refereres til som en tuple. I begge tilfælde skal senere definitioner og funktionsanvendelser overholde disse beskrivelser.

En tuple, der kun består af to typer, kaldes et par. Givet en funktion $f : T_1 \to T_2 \to T_3$, betyder dette, at den tager et udtryk af typen $T_1$, som giver en funktion af typen $T_2 \to T_3$, hvor evalueringen af funktionen resulterer i $T_3$.

Hvis en funktion kaldes med et argument, der ikke matcher funktionens type, genereres en fejlmeddelelse. For F#'s vedkommende kontrolleres dette på oversættelsestidspunktet, hvilket er en af de store forskelle mellem F# og Python, hvor typerne først kontrolleres under kørslen.

### 2.2.1 Induktivt definerede typer

I F# defineres lister og træer induktivt ved brug af rekursive typer. Et eksempel på en rekursivt defineret type for et træ er givet i Listing 3.

```
type Tree =
    | Leaf of int
    | Node of Tree * Tree
```

Listing 3: Eksempel på en rekursivt defineret type for et træ i F#

Rekursive typer lagres som en værdi og ikke en datastruktur.

## 2.3 Property Based Testing

Property Based Testing (PBT) er en teknik til at teste korrekthed af egenskaber som man ved altid skal være opfyldt. Ved PBT genereres en række tilfældige input til en funktion, hvorefter det kontrolleres, om en given egenskab holder. Fokusset ved PBT er de fundamentale egenskaber, som en funktion skal overholde, og ikke de specifikke tilfælde som ved eksempelvis unit tests[7].

På DTU introduceres de studerende til logik, som det første emne i „01001 Matematik 1a“. Her lærer de at en udsagnslogisk formel er gyldig (en tautologi), hvis den altid er sand. Der findes mange teknikker til at påvise gyldigheden af en udsagnslogisk formel. I „01001 Matematik 1a“ anvendes blandt andet sandhedstabeller, som demonstrerer gyldigheden af en formel. Eksempelvis vises hvordan Formel (2) er gyldig.

$$P \wedge (Q \wedge Y) \iff (P \wedge Q) \wedge Y \tag{2}$$

Vi kan også bruge PBT til at undersøge, om Formel (2) holder, ved at definere egenskaben som en funktion af $P, Q$ og $Y$, som vist i Listing 4.

```
#r "nuget: FsCheck"
open FsCheck

// proposition formula: bool -> bool -> bool -> bool
let propositional_formula P Q Y =
    (P && ( Q && Y)) = ((P && Q) && Y)

let propositional_formula_invalid P Q Y =
    P && ( Q && Y) = (P && Q) && Y
```

Listing 4: PBT af Formel (2). Begge sider skal være omgivet af parenteser, for at den er gyldig, da „=“ har en højere præcedens end„ &&“

```
> let _ = Check.Quick propositional_formula;;
Ok, passed 100 tests.
```

Listing 5: Output ved PBT af (2)

[7] [27] *Unit Testing.*

„Check.Quick“ er en del af „FsCheck“-biblioteket. Den tager en funktion som argument og genererer en række tilfældige input til funktionen baseret på funktionens type. Hvis funktionen returnerer „true“ for alle input, vil testen lykkes. Hvis funktionen returnerer „false“ for et input, vil testen fejle og give et eksempel på et input der fejlede. I Formel 4 er „Check.Quick“ anvendt til at teste, om Formel (2) er gyldig. Funktionen „Check.Quick“ returnerer „Ok, passed 100 tests,“ hvilket indikerer, at (2) er gyldig. Det er vigtigt at forstå, at dette ikke er det samme som at bevise, at Formel (2) er gyldig, da testen ikke garanterer, at alle muligheder er testet. Generelt vil der være flere kombinationer af argumenter til en funktion, end de 100 test cases som genereres.

Vi kan også teste den ikke gyldige funktion propositional_formula_invalid, som vist i Listing 6, er den ikke gyldig og vi får givet et eksempel på en input kombination der fejlede.

```
> let _ = Check.Quick propositional_formula_invalid;;
Falsifiable, after 2 tests (0 shrinks) (StdGen (68724885, 297333127)):
Original:
false
true
false
```

Listing 6: Output ved PBT af propositional_formula_invalid

I nogle tilfælde vil det være en fordel at opskrive en PBT før implementeringen af en funktion, som man ved skal overholde en egenskab. På den måde anvendes Test-Driven Development (TDD)[8] til at teste, om ens egenskab forbliver overholdt under implementeringen. I dette projekt vil vi anvende PBT til at validere, at de matematiske egenskaber bliver overholdt af programmet.

## 2.4 Overskrivning af operatorer

I F# er det muligt at overskrive standardoperatorer, så de kan anvendes på egne typer. Denne teknik vil blive benyttet igennem rapporten til at definere matematiske operationer for de typer, vi udvikler. Et eksempel på overskrivning af operatorer er givet i næste sektion.

## 2.5 Signaturfiler og implementeringsfiler

En implementeringsfil i F# er lavet med .fs extension. Implementeringsfilen kan have en tilhørende signaturfil med .fsi extension. Denne fil indeholder en beskrivelse af de typer og funktioner i implementeringsfilen, som er tilgængelige for andre filer. En signaturfil kan derfor bruges som en arbejdstegning for andre, der ønsker at anvende eller replicere implementeringsfilen. Til sidst er der scriptfiler med .fsx extension, som kan bruges til at køre F# kode uden at skulle kompilere den.

Som et eksempel på de forskellige filtyper kan vi se signaturfilen i Listing 7.

```
1 module MyInt
2
3 type MyInt = I of int
4     with
5     static member (*) : MyInt * MyInt -> MyInt
6     static member (-) : MyInt * MyInt -> MyInt
```

[8] [24] *Test-Driven Development*.

```
7
8 val factorial : MyInt -> MyInt
```

Listing 7: Eksempel på en signaturfil *MyInt.fsi*

Ud fra signaturfilen kan vi se, at implementeringsfilen i Listing 8 indeholder en type *MyInt*, hvor der er defineret overskrivning af operatorerne multiplication og subtraction. Derudover er der defineret en funktion factorial.

```
1  module MyInt
2  type MyInt = I of int
3
4  type MyInt with
5      static member (*) (I a, I b) = I (a * b)
6      static member (-) (I a, I b) = I (a - b)
7
8  let rec factorial (n:MyInt) =
9      match n with
10     | I 0 -> I 1
11     | I x when x > 0 -> n * factorial (n - I 1)
12     | _ -> failwith "Negative argument"
```

Listing 8: Eksempel på en implementeringsfil *MyInt.fs*

I denne implementeringsfil finder vi overskrivningerne af operatorerne på typen *MyInt*. Derudover er der også en implementering af en funktion factorial, som virker med typen *MyInt*.

Til sidst kan vi åbne modulet i en scriptfil, som vist i Listing 9.

```
1 #r "../../bin/Release/net7.0/main.dll"
2 open MyInt
3 let a = I 3
4 factorial a |> factorial |> printfn "%A"
```

Listing 9: Eksempel på en scriptfil *MyInt.fsx*

Vi kan derefter køre scriptfilen og få outputtet, som vist i Listing 10.

```
I 720
```

Listing 10: Output ved kørsel af *MyInt.fsx*

## 2.6 Ræsonnering omkring korrekthed

Ræsonnering omkring korrekthed i funktionel programmering indebærer flere begreber, der hjælper med at sikre, at programmer opfører sig som forventet. Et program „virker“ først, når det er korrekt med hensyn til dets specifikationer[9].

Funktionsprogrammering har **Referential Transparency**, hvilket betyder, at en funktion altid vil producere det samme output givet det samme input uden bivirkninger. Dette gør koden mere forudsigelig og lettere at teste, da udtryk kan erstattes af deres værdier uden at ændre programmets adfærd[10]. **Lambda calculus** er blandt andet med til at give funktionsprogrammering „Referential Transparency“.

[9] [14] *ML for the Working Programmer*, Kapitel 6.
[10] [20] *Referential Transparency*.

F# har et **stærkt typesystem**, som sikrer, at funktioner ikke kan kaldes med argumenter af forkerte typer. Dette bliver håndteret på kompileringstidspunktet og reducerer dermed mængden af køretidsfejl.

**Induktivt definerede typer og rekursive funktioner** gør det muligt at bevise korrektheden af programmer ved hjælp af matematiske beviser. For eksempel kan rekursive funktioner bevises korrekte ved hjælp af induktionsbeviser[11].

Disse koncepter kombineret giver et stærkt grundlag for at skrive pålidelig og korrekt software, hvor mange fejl kan fanges tidligt i udviklingsprocessen.

[11] [14] *ML for the Working Programmer*, Kapitel 6.

# 3 Symbolske udtryk

Det ønskes at kunne repræsentere simple udtryk som en type i F#. Vi skal derfor gennemgå en del teori og funktioner som er nødvendige for at kunne dette. Det vil give os et grundlæggende fundament for at kunne udføre matematiske evalueringer som differentiering i F#. Som de fleste andre programmer har F# kun float og int som kan repræsentere tal. Derfor vil vi begynde med at definere et modul som indeholder en type for tal. Tankegangen her at gennemgå en opbygning af en måde at kunne repræsentere udtryk samt reducere dem. Vi begrænset os selv til at kun have matematiske operationer som addition, subtraktion, negation, multiplikation og division.

## 3.1 Tal mængder

Vi begynder med opbygningen af et modul, der kan repræsentere en talmængde. Typen for tal består af tre konstruktører, henholdsvis for heltal, rationale tal og komplekse tal hvor real- og imaginærdelen er rationale.

### 3.1.1 Rationelle tal modul

Repræsentationen af rationale tal kan laves ved hjælp af at danne et par af heltal, hvor det ene heltal er tælleren, og det andet er nævneren.

```
type rational = R of int * int
```

Listing 11: Typen for rationelle tal

Nedenfor er der givet en signaturfil for rational modulet 12. I implementeringsfilen overskrives de matematiske operatorer ved hjælp af de klassiske regneregler for brøker[12].

```
module rational


type rational = R of int * int
    with
    static member ( ~- ) : rational -> rational
    static member ( + )  : rational * rational -> rational
    static member ( + )  : int * rational -> rational
    static member ( - )  : rational * rational -> rational
    static member ( - )  : int * rational -> rational
    static member ( * )  : int * rational -> rational
    static member ( * )  : rational * int -> rational
    static member ( * )  : rational * rational -> rational
    static member ( / )  : rational * rational -> rational
    static member ( / )  : int * rational -> rational
    static member ( / )  : rational * int -> rational
    static member ( / )  : int * int -> rational

val newRational         : int * int -> rational
val equal        : rational * rational -> bool
val positive     : rational -> bool
val toString     : rational -> string
val isZero      : rational -> bool
val isOne        : rational -> bool
```

[12] [19] *Rational Number wikipedia*.

```
25 val isInt         : rational -> bool
26 val makeInt    : rational -> int
27 val greaterThan   : rational * rational -> bool
28 val isNegative    : rational -> bool
29 val absRational   : rational -> rational
```

Listing 12: Signaturfilen for rational-modulet

For at kunne sammenligne og også for nemmere at undgå for store brøker, vil alle rationelle tal blive reduceret til deres simplest form. Dette gøres ved at finde den største fælles divisor (GCD)[13]. Derudover er det vigtigt at være opmærksom på ikke at foretage nul division. Derfor vil implementeringsfilen kaste en „System.DivideByZeroException“, hvis nævneren er eller bliver nul. Signaturfilen indeholder en række funktioner, som anvendes af andre filer. Det vil desuden være nødvendigt at kunne håndtere overflow, idet heltallene, der repræsenterer de rationelle tal, under eller efter operationen kan blive for store til korrekt at blive repræsenteret af 32-bit. Da denne rapport fokuserer på implementeringen af matematiske koncepters og ikke numeriske algoritmer, vil modulet blot slå en fejl, hvis der opstår overflow.

### 3.1.2 Komplekse tal-modul

Vi skal nu dykke lidt mere ned i implementeringen af et modul for komplekse tal. Signaturfilen *complex.fsi* givet i Appendiks 8.1. Først defineres en type for komplekse tal, som består af et rationale tal par, henholdsvis for realdelen og imaginærdelen, se Listing 13.

```
1 type complex = C of rational * rational
```

Listing 13: Typen for komplekse tal

Der kan opskrives en række regneregler i Definition 1 for operationer på komplekse tal.

**Defination 1** (Regneregler for komplekse tal)**.**
*Lad $a, b, c, d \in \mathbb{Q}$ så er følgende defineret omkring komplekse tal*[14]*:*

1. ***Addition***
   $(a + bi) + (c + di) = (a + c) + (b + d)i$
2. ***Subtraktion***
   $(a + bi) - (c + di) = (a - c) + (b - d)i$
3. ***Multiplikation***
   $(a + bi) \cdot (c + di) = (ac - bd) + (bc + ad)i$
4. ***Kvadratisk form***
   $(a + bi) \cdot (a - bi) = a^2 + b^2$
5. ***Konjugering***
   $\overline{a + bi} = a - bi$
6. ***Division***
   $\frac{a+bi}{c+di} = \frac{(a+bi)\cdot(c-di)}{c^2+d^2}$

[13] [7] *Greatest common divisor wikipedia.*
[14] [12] *Mathematics 1a*, se. Definition 3.3 s. 54, Definition 3.5 s. 56, Definition 3.8 s. 57, ligning 3-2 s. 58.

Ved implementeringen, se Listing 14, af addition, subtraktion, multiplikation samt skalering med et rationelt tal. Skaleringen af et rationelt tal, kan udledes fra multiplikation, ved at lade den imaginære del være 0. Disse er simple operationer, som ikke behøver at defineres i særskilte funktioner. De kan defineres direkte på overskrivningen af deres respektive operatorer. Dog er det nødvendigt at definere multiplikation som en funktion, da den skal anvendes af divisionsfunktionen.

```
// complexDivRational: complex -> rational -> complex
let complexDivRational c (n) =
    match c with
    | _ when isZero n ->  raise (System.DivideByZeroException("Complex.
    divRational: Cannot divide by zero!"))
    | C (a, b) -> C (a / n, b / n)

// mulConjugate: complex -> rational
let mulConjugate (C(a, b)) = a*a + b*b

// conjugate: complex -> complex
let conjugate (C (a, b)) = C (a, -b)

// mulComplex: complex -> complex -> complex
let mulComplex (C (a, b)) (C (c, d)) = C(a*c-b*d, b*c+a*d)

// divComplex: complex -> complex -> complex
let divComplex z1 z2 =
    complexDivRational (mulComplex z1 (conjugate z2)) <| mulConjugate z2

type complex with
    static member (+)  (C(a, b), C(c, d)) =  C(a + c, b + d)
    static member (-)  (C(a, b), C(c, d)) =  C(a - c, b - d)
    static member (*)  (n, C(a, b))       =  C(n * a, n * b)
    static member (*)  (C(a, b), n)       =  C(n * a, n * b)
    static member (*)  (z1, z2)           =  mulComplex z1 z2
    static member (/)  (z, n)             =  complexDivRational z n
    static member (/)  (z1, z2)           =  divComplex z1 z2
    static member (~-) (C(a, b))          =  C(-a, -b)
```

Listing 14: Overskrivning af operationer på komplekse tal

Bortset fra division af to komplekse tal, ligner de resulterende overskrivninger på operationerne deres respektive matematiske definitioner. Men når vi nærmere studerer divisionen af to komplekse tal, ser vi, at der blot er brug for få funktioner til at kunne udføre divisionen. Først konjugeres nævneren, derefter multiplikeres resultatet med tælleren. Til sidst divideres resultatet af multiplikationen med kvadratet af nævneren. Da komplekse tal som en talmængde indeholder heltal og rationale tal, vil vi i det følgende afsnit omkring tal modulet anvende komplekse tal til udføre de matematiske operationer i modulet.

### 3.1.3 Tal modulet

Vi har nu beskrevet en måde at kunne repræsentere bruger definere tal på ved brug af typer i F#. Det vil derfor være oplagt at have en type som indeholder alle de typer tal vi ønsker at kunne anvende i de matematiske udtryk vi er ved at opbygge. Fordelen ved at samle dem til en type er at vi kan lave en række funktioner blandt andet matematiske operationer som kan anvendes på alle type tal. Ved at samle dem til en type „Number“ kan vi også opskrive en række egenskaber i 1 som vi ønsker at de skal opfylde. Egenskaberne vil blive valideret i sektion 5.1.1.

**Egenskab 1** (Egenskaber for Number-typen)**.**
*Lad $a, b, c \in$ Number, så gælder følgende*[15]*:*

1. *Addition og multiplikation er* ***associative***
   $a + (b + c) = (a + b) + c$ *og* $a \cdot (b \cdot c) = (a \cdot b) \cdot c$
2. *Addition og multiplikation er* ***kommutative***
   $a + b = b + a$ *og* $a \cdot b = b \cdot a$
3. ***Distributivitet*** *af multiplikation over addition*
   $a \cdot (b + c) = a \cdot b + a \cdot c$
4. *Addition og multiplikation har et* ***neutralt element***
   $a + 0 = a$ *og* $a \cdot 1 = a$
5. ***Omvendt funktion*** *eksisterer til addition*
   $a + (-a) = 0$
6. ***Omvendt funktion*** *eksisterer til multiplikation for* $a \in$ Number $\setminus \{0\}$
   $a \cdot a^{-1} = 1$

Vi begynder med at definere typen for tal i Listing 15, som indeholder konstruktører for de tal typer vi har definerede samt en for heltal.

```
1     type Number = | Int of int | Rational of rational | Complex of complex
```

Listing 15: Typen for Number

Typen Number er designet med henblik på, at den kan udvides med flere taltyper. Ved udvidelse er det et krav til den nye talmængde, at der er definerede matematiske operationer i form af addition, subtraktion, negation, multiplikation og division. Den nye Number type skal derudover forsat opfylde egenskaberne 1. En udvidelse kunne være for reelle tal, som kan håndtere „floating point errors“[16], men for at undgå at komplicere programmet yderligere vil vi i denne opgave ikke inkludere decimal tal.

Betragtes signatur filen for Number modulet i Listing 16, ses det at der igen er defineret overskrivning af de anvendte matematiske operationer. Derudover er der defineret en række funktioner som kan anvendes på Number typen.

```
1  module Number
2  open rational
3  open complex
4
5  type Number =
6      | Int of int
7      | Rational of rational
8      | Complex of complex
9  with
10     static member ( + )  : Number * Number -> Number
11     static member ( - )  : Number * Number -> Number
12     static member ( * )  : Number * Number -> Number
13     static member ( / )  : Number * Number -> Number
14     static member ( ~- ) : Number -> Number
```

[15] [12] *Mathematics 1a*, se. Side 124 Definition 6.1.
[16] [3] *Floating-point error*.

```
15
16 val zero         : Number
17 val one          : Number
18 val two          : Number
19 val isZero       : Number -> bool
20 val isOne        : Number -> bool
21 val isNegative   : Number -> bool
22 val absNumber    : Number -> Number
23 val greaterThan  : Number -> Number -> bool
24 val tryReduce    : Number -> Number
25 val toString     : Number -> string
26 val conjugate    : Number -> Number
27 val inv          : Number -> Number
28 val isInt        : Number -> bool
```

Listing 16: Signatur filen for Number modulet

Ved implementeringen af de matematiske operationer, hvis der eksistere en konstruktør i Number, der repræsentere en tal mængde hvor alle andre konstruktører er delmængder af denne mængde. Er det muligt at definere en enkelt funktion som kan udføre alle binærer operationer. Som et eksempel er funktionen operation i Listing 17 givet, som tager to tal og en funktion i form af den ønskede binærer operation som argumenter. Funktionen konventere de to tal til vores komplekse tal type, hvortil den binære operation udføres på dem.

```
1 // makeComplex: Number -> complex
2 let makeComplex n =
3   match n with
4   | Int x -> newComplex (newRational(x, 1), newRational(0, 1))
5   | Rational x -> newComplex (x, newRational(0, 1))
6   | Complex x -> x
7
8 // operation: Number -> Number -> (complex -> complex -> complex) -> Number
9 let operation a b f =
10     f (makeComplex a) (makeComplex b) |> Complex
```

Listing 17: operation funktionen til udførelse af matematiske operationer

Det vil her til være oplagt, at have en funktion til at reducere tallet efter den matematiske operation, til den simpleste tal type. Den simpleste tal typer er i vores tilfælde er heltal. Dette er gjort ved at anvende funktionen tryMakeInt på alle de matematiske operationers overskrivninger i Listing 18.

```
1 // tryMakeInt: Number -> Number
2 let tryMakeInt r =
3     match r with
4     | Rational a when isInt a -> Int (makeRatInt a)
5     | _ -> r
6
7 type Number with
8   static member (+) (a, b) = operation a b (+) |> tryMakeInt
9   static member (-) (a, b) = operation a b (-) |> tryMakeInt
10  static member (*) (a, b) = operation a b (*) |> tryMakeInt
11  static member (/) (a, b) = operation a b (/) |> tryMakeInt
12  static member (~-) (a)   = neg a |> tryMakeInt
```

Listing 18: Overloadnings funktionerne for Number

Ved at have implementeret de matematiske operationer på denne måde, er det blandt andet muligt at fortage heltales division, da det vil resulterer i et rationalt tal.

Dermed har vi et modul som kan repræsentere tal, samt udføre matematiske operationer på dem. Vi vil nu begynde at betragte hvordan vi kan anvende den i et matematisk udtryk.

## 3.2 Matematiske udtryk

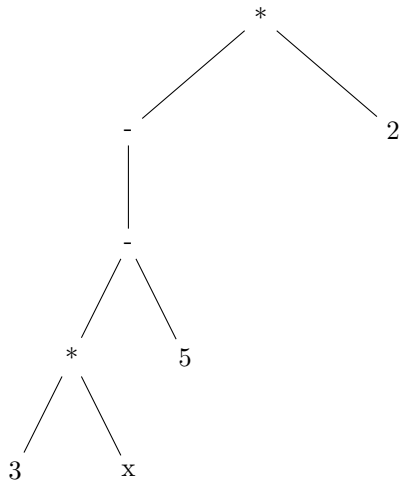
### 3.2.1 Polsk notation

Matematiske udtryk som vi normalt kender dem er skrevet med infix notation. I infix notation skrives en binær operator mellem to operandere, kende tegnet for sproget er at det indeholder parenteser samt præcedens regler. Dette gør det generalt kompliceret at evaluere og håndtere matematiske udtryk i et programmeringssprog. Derfor er det mere oplagt at kunne anvende polsk notation (prefix) istedet, hvor operatoren skrives før operandere eller omvendt polsk notation (postfix). Da de hverken indeholder parenteser eller præcedens regler[17]. Følgende er der givet et simpelt eksempel på de 3 notationer.

$$\begin{aligned} \text{Infix Notation:} &\quad (A+B)\cdot C \\ \text{Prefix Notation:} &\quad \cdot\,+ABC \\ \text{Postfix Notation:} &\quad A\,B\,+\,C\,\cdot \end{aligned}$$

### 3.2.2 Udtryk som træer

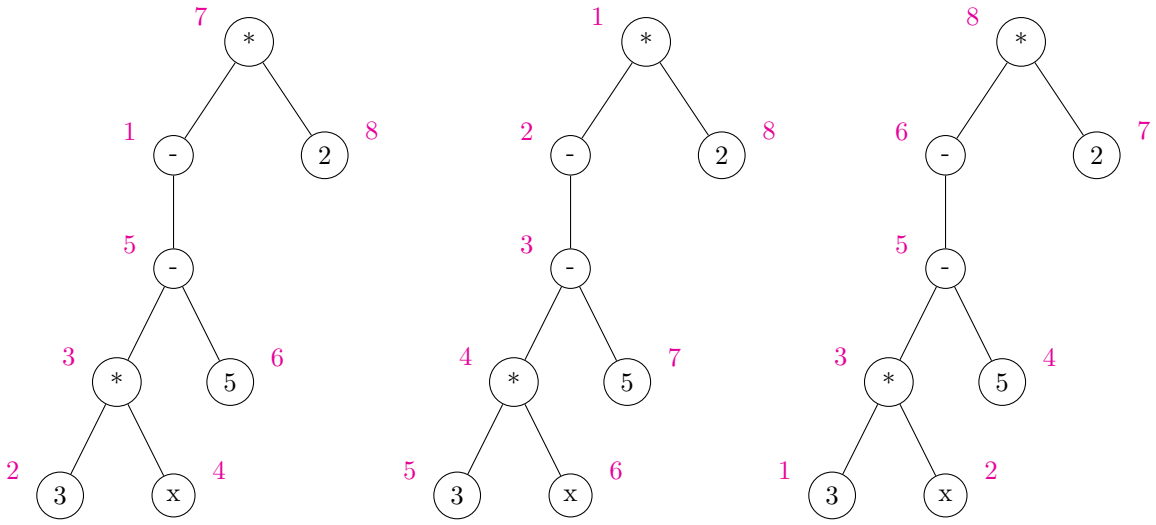
Et matematisk udtryk kan repræsenteres som et binært træ, hvor bladene er operander i det anvendte matematiske rum og alle andre noder er operationer. Som eksempel kan udtrykket $-(3 \cdot x - 5) \cdot 2$ repræsenteres som følgende træ i Figur 2.




Figur 2: Et binært træ der repræsenterer udtrykket $-(3 \cdot x - 5) \cdot 2$

[17][15] *Polsk notation wikipedia.*

Det skal bemærkes der er forskel på den unære og binære operator '−' i træet, den unære betyder negation og den binære er subtraktion. Givet et binært træ for et matematisk udtryk, vil det være muligt omdanne dem til infix, prefix eller postfix notation. Dette kan gøres ved at anvende modificeret Inorder, Preorder eller Postorder Traversal[18], algorithmerne er illustreret i Figur 3.




Modificeret Inorder Traversal: $-(3 \cdot x - 5) \cdot 2$

Preorder Traversal: $\cdot - - \cdot 3\, x\, 5\, 2$

Postorder Traversal: $3\, x\, \cdot 5\, - -2\, \cdot$

Figur 3: Træet fra Figur 2 med forskellige travesal metoder

Vi vil i 3.2.3 betragte hvordan vi kan implementere et modul som kan repræsentere udtryk ved brug af prefix notation. Postorder Traversal bliver blandt andet anvendt til at kunne rekursivt reducere og evaluere udtryk.

Grundet præcedens regler i infix notation, er det nødvendigt at modificere Inorder Traversal, da unære noder altid skal håndteres før dens børn. Desuden vil det også være nødvendigt at implementere regler for at håndtere parenteser, hvis der ønskes et symbolsk udtryk. Den modificeret Inorder Traversal anvendes til at kunne visualisere udtrykket i infix notation.

### 3.2.3 Udtryksmodulet

Efter udviklingen af et modul til repræsentation af talmængder er vi nu klar til at udvide programmet med et modul for matematiske udtryk. Vi starter med at definere en polymorf type for udtryk, givet i Listing 19. Denne type omfatter flere konstruktører, hver tilknyttet specifikke

[18] [25] *Tree Traversal Techniques – Data Structure and Algorithm Tutorials*.

matematiske operationer vi ønsker at implementere. Desuden introducerer vi konstruktøren `N` til at repræsentere numeriske værdier ved at anvende talmængder defineret i Listing 15. Til sidst tilføjer vi konstruktøren `X` for variable. Således lagres matematiske udtryk i en rekursiv datatype, der skal repræsentere udtryks træerne i sektion 3.2.2. Dette skyldes at, hver konstruktør for en operation indeholder et eller to underudtryk af samme type.

```
type Expr<'a> =
    | X of char
    | N of 'a
    | Neg of Expr<'a>
    | Add of Expr<'a> * Expr<'a>
    | Sub of Expr<'a> * Expr<'a>
    | Mul of Expr<'a> * Expr<'a>
    | Div of Expr<'a> * Expr<'a>
```

Listing 19: Typen for Expr

Expr<'a> typen er dermed en polymorfisk type, hvor 'a er typen for den tal mængde hvor vi kan lave brugerdefinerede matematiske operationer. Et exemplar på en Expr<Number> er givet i Listing 20. Her ses det at når udtrykket $-(3 \cdot x - 5) \cdot 2$ visualiseres er det i prefix notation.

```
> tree "-(3*x-5)*2";;
val it: Expr<Number> =
  Mul (Neg (Sub (Mul (N (Int 3), X 'x'), N (Int 5))), N (Int 2))
```

Listing 20: $-(3 \cdot x - 5) \cdot 2$ som et udtryks træ. Funktionen tree bliver beskrevet i 3.4.

Signatur filen indeholder overskrivninger på de matematiske operationer, så de kan anvendes på udtryk. Samt en funktion eval til at evaluere et udtryk beskrevet i 3.3.

```
module Expression
open Number

type Expr<'a> =
  | X of char
  | N of 'a
  | Neg of Expr<'a>
  | Add of Expr<'a> * Expr<'a>
  | Sub of Expr<'a> * Expr<'a>
  | Mul of Expr<'a> * Expr<'a>
  | Div of Expr<'a> * Expr<'a>
with
  static member ( ~- ) : Expr<Number> -> Expr<Number>
  static member ( + ) : Expr<Number> * Expr<Number> -> Expr<Number>
  static member ( - ) : Expr<Number> * Expr<Number> -> Expr<Number>
  static member ( * ) : Expr<Number> * Expr<Number> -> Expr<Number>
  static member ( / ) : Expr<Number> * Expr<Number> -> Expr<Number>

val eval : Expr<Number> -> Map<char,Number> -> Number
val isAdd : (Expr<Number> -> Expr<Number> -> Expr<Number>) -> bool
val isSub : (Expr<Number> -> Expr<Number> -> Expr<Number>) -> bool
val isMul : (Expr<Number> -> Expr<Number> -> Expr<Number>) -> bool
val isDiv : (Expr<Number> -> Expr<Number> -> Expr<Number>) -> bool
val isNeg : (Expr<Number> -> Expr<Number>) -> bool
val isZero : Expr<Number> -> bool
val containsX : Expr<Number> -> Expr<Number> -> bool
```

```
27 val getNumber : Expr<Number> -> Number
28 val getVariable : Expr<Number> -> char
```

Listing 21: Signatur filen for Expression modulet

De overskrevne matematiske operatorer i implementeringsfilen for udtryk laver overflade evalueringer samt reduceringer på deres respektive argumenter. Overfalde evaluering vil sige at de individuelle funktioner kun betragter de to øverste niveauer på de udtryks træer de tager som input, implementeringerne af addition og multiplikation er givet i Listing 22. Lignende funktioner er implementeret for de andre operationer.

```
1  // mul: Expr<Number> -> Expr<Number> -> Expr<Number>
2  let rec mul e1 e2:Expr<Number> =
3    match e1, e2 with
4    |N a, N b                          -> N (a * b)
5    |N a, b | b, N a when isOne a      -> b
6    |N a, _ | _, N a when isZero a     -> N zero
7    |Div(a, b), c | c, Div(a, b)       -> Div (mul a c, b)
8    |Div (a, b), Div (c, d)            -> Div ((mul a c), (mul b d))
9    |Neg a, Neg b                      -> mul a b
10   | _, _                             -> Mul(e1, e2)
11
12 // add: Expr<Number> -> Expr<Number> -> Expr<Number>
13 let rec add e1 e2:Expr<Number>  =
14   match e1, e2 with
15   | N a, N b                                -> N (a + b)
16   | N a, b | b, N a when isZero a           -> b
17   | a, b when a = b                         -> Mul (N two, b)
18   | Neg a, Neg b                            -> neg (add a b)
19   | Neg a, b | b, Neg a                     -> Sub (b, a)
20   | Mul(a, X b), Mul(c, X d)
21   | Mul(X b, a), Mul(c, X d)
22   | Mul(a, X b), Mul(X d, c)
23   | Mul(X b, a), Mul(X d, c) when b = d -> Mul(add a c, X b)
24   | _, _                                    -> Add(e1, e2)
```

Listing 22: Addition og multiplikation af to udtryk

## 3.3 Evaluering af udtryk

Vi vil nu betragte hvordan vi kan evaluere et udtryk, ved hjælp af et miljø som indeholder værdier for variable som er indeholdt i udtrykket. Evalueringen af udtryk skal kunne opfylde følgende homomorfiske egenskaber 2. Egenskaben vil blive valideret i sektion 5.1.2.

**Egenskab 2** (Homomorfisme af evaluering)**.**
*Lad* $\oplus \in \{+, -, \times, /\}$ *sættet af binære operationer,* $e1$ *og* $e2$ *være udtryk, så gælder følgende:*

$$eval(e1 \oplus e2) = eval(e1) \oplus eval(e2)$$

*Derudover skal det om negation også gælde at:*

$$eval(-e) = -eval(e)$$

Funktionen eval i Listing 23 tager et udtryk og et miljø som input og evaluere udtrykket til en numerisk værdi. Funktionen kører en Postorder Traversal på udtrykket og evaluerer dermed udtrykket nedefra og op, ved at foretage matematiske operationer defineret i Number modulet.

```
// eval: Expr<Number> -> Map<char, Number> -> Number
let rec eval (e:Expr<Number>) (env) =
  match e with
  | X x -> Map.find x env
  | N n -> n
  | Neg a -> - eval a env
  | Add (a, b) -> eval a env + eval b env
  | Sub (a, b) -> eval a env - eval b env
  | Mul (a, b) -> eval a env * eval b env
  | Div (a, b) -> eval a env / eval b env
```

Listing 23: Evaluering af et udtryk

## 3.4 Konventering mellem udtryksformer

Det er ønskes at kunne konvertere udtryk frem og tilbage mellem prefix notation, repræsenteret af Expression-typen, og den standard infix notation. Dette ønske skyldes, at infix notation er lettere for os at læse og skrive. Derfor er det essentielt, at de to konverteringsfunktioner fungerer som hinandens inverse. Dette krav er yderligere uddybet i egenskab 3. Egenskaben bliver valideret i sektion 5.1.4.

**Egenskab 3** (Invers morphism[19] mellem infix og prefix)**.**
*Lad $Q^n$ være mængden af rationelle infix udtryk repræsenteret som en string, med $n$ variable, så defineres følgende:*

$$tree : Q^n \to Expr$$
$$tree^{-1} : Expr \to Q^n$$

*Dermed gælder følgende egenskaber*

$$tree^{-1} \circ tree = id_{Q^n}$$
$$tree \circ tree^{-1} = id_{Expr}$$

*Hvor $id_x$ er identitetsfunktionen på mængden $x$.*

Vi begynder med at betragte den inverse funktion, som konverterer et udtryk til infix notation. Funktionen etf i Listing 24 fortager denne konvertering ved at lave den modificeret Inorder Traversal på udtrykket, som blev beskrevet i 3.2.2. Den modificeret del er at håndtere parentesersamt negation, som var det en binær node i træet hvor det venstre barn er et tomt udtryk.

```
// parenthesis: bool -> string -> string
let parenthesis b f = if b then "(" + f + ")" else f

// etf: Expr<Number> -> bool -> string
let rec etf e p =
    match e with
```

[19] [8] *Inverse function.*

```
    | N a when not <| isInt a -> parenthesis p <| toString a
    | N a    -> toString a
    | X a    -> string a
    | Neg a -> parenthesis p <| "-" + etf a (not p)
    | Add(a, b) -> parenthesis p <| etf a false + "+" + etf b false
    | Sub(a, b) -> parenthesis p <| etf a false + "-" + etf b true
    | Mul(a, b) -> parenthesis p <| etf a true  + "*" + etf b true
    | Div(a, b) -> parenthesis p <| etf a true  + "/" + etf b true


// infixExpression: Expr<Number> -> string
let infixExpression e = etf e false
```

Listing 24: konventering fra expression til infix notation

Funktionen `tree`, som foretager konverteringen fra infix notation til et udtrykstræ, er baseret på algoritmen beskrevet i [1][20]. Først konverteres en udtryksstreng til en liste af tokens. Disse tokens beskriver, om en karakter i udtrykket er en operand, en operator, eller en konstant, hvor en operator også indeholder information om præcedens og associativitet[21]. Typen for disse tokens kan ses i Listing 25. Herefter anvendes to stacks: én for operatorer og én for udtryk. Der anvendes en række regler, som beskrevet i [1], for hvornår der skal udføres pop og push på disse to stacks. Det skal bemærkes, at når en operator pushes til udtryksstacken, da navnet på operatorkonstruktøren på udtrykket står skrevet fra venstre mod højre, vil udtryksstacken være i prefix notation og ikke postfix notation som beskrevet i kilden. Funktionen `tree` er at finde i Appendiks 8.2.

```
type Associative = | Left | Right
type Precedence = int
type Operator = char * Precedence * Associative
type Token =
    | Operand of char
    | Operator of Operator
    | Constant of int
type OperatorList = Operator list
```

Listing 25: Konvertering fra infix til udtrykstræ

## 3.5 Reducering af udtryk

Vi skal nu betragte en systematisk metode til at kunne reducere matematiske udtryk, ved hjælp af simple algebraiske regler. Dette er en nødvendighed at kunne for at bruge udtrykkene i en matematisk sammenhæng, da det vil kunne medføre både en reduktion i kompleksitet og en forbedring i læsbarhed når udtrykkene visualiseres. Før vi betragter metoden, kan vi opskrive en egenskab som reduceringen skal overholde. Egenskaben vil blive valideret i sektion 5.1.3, det er en nødvendighed at evaluere udtrykket før og efter reduktion, da det er en kompleks opgave at skulle sammenligne om to udtryk er ækvivalente.

**Egenskab 4** (Reducering af udtryk)**.**
*Lad e være et udtryk, så gælder følgende:*

$$eval(e) = eval(simplifyExpr(e))$$

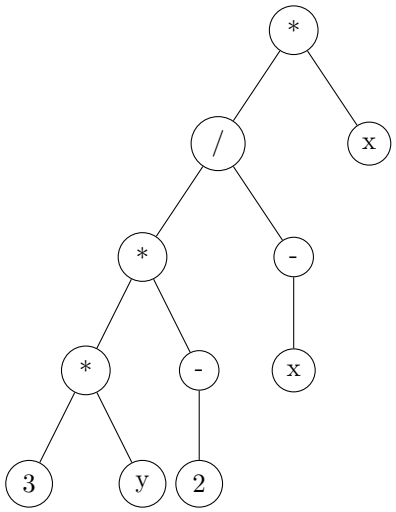
[20] [1] *Convert Infix expression to Postfix expression.*
[21] [13] *Operator Precedence and Associativity in C.*

Vi begynder med at betragte funktionen simplifyExpr i Listing 26, som reducere et udtryk ved at foretage en Postorder Traversal på udtrykket. Hvilket betyder når en node i udtrykstræet skal reduceres, vil dens børn allerede være reduceret.

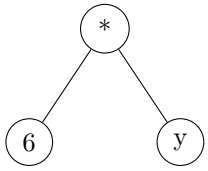
```
//simplifyOperation: Expr<Number> -> Expr<Number> -> (Expr<Number> -> Expr<
    Number> -> Expr<Number>) -> Expr<Number>
let rec simplifyOperation e1 e2 f =
  match f e1 e2 with
  | Neg a ->
    commutativeMulDiv.applyCommutative (Neg a) |> commutativeAddSub.
    applyCommutative
  | Add(a, b) when isAdd f -> commutativeAddSub.applyCommutative (Add(a, b))
  | Sub(a, b) when isSub f -> commutativeAddSub.applyCommutative (Sub(a, b))
  | Mul(a, b) when isMul f -> commutativeMulDiv.applyCommutative (Mul(a, b))
  | Div(a, b) when isDiv f -> commutativeMulDiv.applyCommutative (Div(a, b))
  | Add(a, b) -> simplifyOperation a b (+)
  | Sub(a, b) -> simplifyOperation a b (-)
  | Mul(a, b) -> simplifyOperation a b (*)
  | Div(a, b) -> simplifyOperation a b (/)
  | a -> a

//simplifyExpr: Expr<Number> -> Expr<Number>
let rec simplifyExpr e =
  match e with
  | N a when Number.isNegative a -> Neg (N (Number.absNumber a))
  | N (Rational(R(a, b))) ->
    simplifyOperation (simplifyExpr (N (Int a))) (simplifyExpr (N (Int b)))
    (/)
  | Neg a     -> - (simplifyExpr a)
  | Add(a, b) -> simplifyOperation (simplifyExpr a) (simplifyExpr b) (+)
  | Sub(a, b) -> simplifyOperation (simplifyExpr a) (simplifyExpr b) (-)
  | Mul(a, b) -> simplifyOperation (simplifyExpr a) (simplifyExpr b) (*)
  | Div(a, b) -> simplifyOperation (simplifyExpr a) (simplifyExpr b) (/)
  | _ -> e
```

Listing 26: Reducering af et udtryk

Det er simplifyOperation, som foretager selve reduceringen af et givet udtryk. Funktionen tager som argumenter to udtryk samt den binære operation, der skal anvendes på disse. Funktionen anvendes på udtrykkene, hvorefter overfladisk reducering, som blev beskrevet i 3.2.3, udføres. Hvis overfladisk reducering resulterer i en ændring af den anvendte operation, kalder funktionen sig selv rekursivt med de to udtryk og den nye operation. Hvis overfladisk reducering ikke resulterer i ændringer i operationen, vil funktionen foretage en dybere reducering af de to udtryk. Denne dybere reducering udføres af funktionerne applyCommutative fra filerne *CommutativeAddSub.fs* og *CommutativeMulDiv.fs*, som findes i Appendiks 8.3 og 8.4. Disse funktioner arbejder efter samme princip, hvor de starter med at flade udtrykstræerne ud ifølge de kommutative regler for henholdsvis addition og multiplikation. Derefter sorterer de udtrykstræet, hvilket muliggør at foretage overfladisk når ved at gendanne træet. For multiplikation anvendes samme metode i nævneren for division, og der undersøges, om der er fælles udtryk i det udfladede træ, som fremtræder i nævneren af en division og i det udfladede træ, der indeholder divisionen. Et eksempel på anvendelse af den kommutative multiplikationsreducering på et udtryk er givet i Figur 4, som viser det visuelle input og det resulterende svar fra funktionen, samt Figur 5, der viser det udfladede træ og sorteringen af det fladede træ.

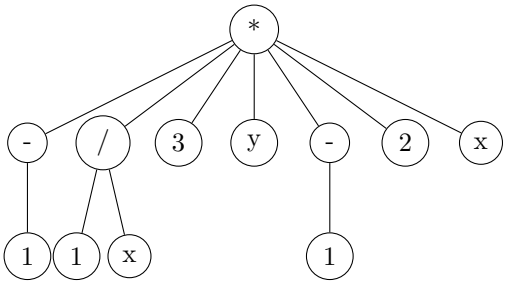



(a) Udtrykket $(3 \cdot y \cdot (-2)/-x) \cdot x$ som et træ

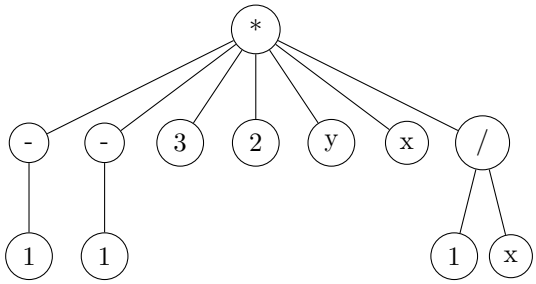



(b) Det resuterende reduceret udtryk, efter at have anvendt CommutativeMulDiv.applyCommutative på udtrykket i Figur 4a

Figur 4: Før og efter reducering af et udtryk ved brug af CommutativeMulDiv.applyCommutative




(a) Det udfladede træ




(b) Det sorterede træ fra venstre mod højre

Figur 5: Det udfladede og sorterede af udtrykket $(3 \cdot y \cdot (-2)/-x) \cdot x$ i processen af kommutativ reducering

Generelt, når det gælder reducering af udtryk, skal man være opmærksom på ikke at ende i et uendeligt loop. Derfor er det vigtigt ikke at definere nogle overfladiske reduceringer, som er hinandens inverse funktioner. Desuden forsøger funktionerne i dette program altid at skubbe negation af udtryk så langt ud som muligt i håb om, at de kan ophæve hinanden. Dette illustreres blandt andet i figur 5, hvor ved udfladning af træet, hvis et af de kommutative udtryk for multiplikation er negativt, fjernes negationen, og der tilføjes i stedet en negation af tallet 1, som ved sortering skubbes til venstre.

## 3.6 Differentiering af udtryk

Vi kan nu betragte den første implementering, der benytter vores udtryksmodul, som samtidig vil understrege vigtigheden af at kunne reducere udtryk. Vi begynder igen med at opskrive nogle algebraiske linearitetsegenskaber, som differentieringen skal overholde. Disse egenskaber vil blive testet i sektion 5.1.5.

**Egenskab 5** (Linearitetsbetingelserne[22])**.**
*Lad $f$ og $g$ være udtryk, og $a$ være tal fra talmodulet, så gælder følgende:*

1. ***Skaleringsreglen***
   $\frac{d}{dx}(af) = a\frac{df}{dx}$
2. ***Sumreglen***
   $\frac{d}{dx}(f+g) = \frac{df}{dx} + \frac{dg}{dx}$
3. ***Subtraktionsreglen***
   $\frac{d}{dx}(f-g) = \frac{df}{dx} - \frac{dg}{dx}$
4. ***Produktreglen***
   $\frac{d}{dx}(fg) = \frac{df}{dx}g + f\frac{dg}{dx}$
5. ***Kvotientreglen***
   $\frac{d}{dx}\left(\frac{f}{g}\right) = \frac{\frac{df}{dx}g - f\frac{dg}{dx}}{g^2}$

Funktionen for differentiering diff i Listing 27, som tager et udtryk og en variabel som input og differentierer udtrykket med hensyn til variablen. Dette er en af de store fordele ved at anvende et funktionelt programmeringssprog, da det ses, hvordan fire af reglerne fra egenskab 5 er implementeret direkte, som de er beskrevet.

```
// diff: Expr<Number> -> char -> Expr<Number>
let rec diff e dx =
    match e with
    | X f when f = dx -> N (Int 1)
    | X _ -> N (Int 0)
    | N _ -> N (Int 0)
    | Neg f -> diff (Mul (N (Int -1), f)) dx
    | Add(f, g) -> Add(diff f dx, diff g dx)
    | Sub(f, g) -> Sub(diff f dx, diff g dx)
    | Mul(f, g) -> Add(Mul(diff f dx, g), Mul(f, diff g dx))
    | Div(f, g) -> Div(Sub(Mul(diff f dx, g), Mul(f, diff g dx)), Mul(g, g))
```

Listing 27: Differentiering af et udtryk

## 3.7 Multivariable polynomier af første grad

Da vi senere i projektet skal betragte matricer, vil vi i den forbindelse også lave en løsning af lineære ligningssystemer i sektion 4.5. Det kræver derfor, at vi kan isolere variable i et multivariabelt polynomium af første grad. Vi betragter derfor nu to simple funktioner til at udføre denne isolation, se Listing 28. Funktionen isolateX undersøger først, om den variabel, som skal isoleres, befinder sig på højre eller venstre side, derefter kaldes funktionen expressionOnX, som fungerer

[22] [2] *Differentiation Rules*.

ved at evaluere til en funktion, der giver den omvendte operation af den operation, som variablen er involveret i. Dertil anvendes den omvendte funktion på begge sider af ligningen, hvor hvis variablen er isoleret, gives et udtrykspar, hvor det første udtryk er den isolerede variabel. Hvis variablen ikke er isoleret, kaldes funktionen rekursivt med de nye højre og venstre sider.

```
// expressionOnX: Expr<'a> -> Expr<'a> -> (Expr<'a> -> Expr<'a>)
let rec expressionOnX hs x =
  match hs with
  | N _ | X _ -> fun e -> e
  | Neg(a) when a = x -> fun e -> Neg e
  | Sub(a, b) when a = x -> fun e -> Add(e, b)
  | Div(a, b) when a = x -> fun e -> Mul(e, b)
  | Div(_, a) when a = x -> fun e -> Mul(e, a)
  | Mul(a, b) | Mul(b, a) when a = x -> fun e -> Div(e, b)
  | Add(a, b) | Add(b, a) | Sub(b, a) when a = x -> fun e -> Sub(e, b)
  | Add(a, b) | Sub(a, b) | Mul(a, b) | Div(a, b) -> fun e -> expressionOnX a
    x e |> expressionOnX b x
  | Neg(a) -> fun e -> expressionOnX a x e

// isolateX: Expr<Number> -> Expr<Number> -> Expr<Number> -> Expr<Number> *
    Expr<Number>
let rec isolateX lhs rhs x =
  let operation =
      if containsX lhs x
          then expressionOnX lhs x
      elif containsX rhs x
          then expressionOnX rhs x
      else
          failwith "Variable not found in either side of the equation"
  match operation lhs |> simplifyExpr, operation rhs |> simplifyExpr with
  | a, b | b, a when a = x -> (a, b)
  | a, b -> isolateX a b x
```

Listing 28: Isolering af variable i et udtryk

Da funktionen kun betragter operationen på den variable, der ønskes isoleret, eksisterer der mange tilfælde, hvor funktionen ikke vil kunne isolere variablen. Men den fungerer til at løse ligninger af formen $a_1x_1 + a_2x_2 + \ldots + a_nx_n = b$, hvilket er tilstrækkeligt for vores formål.

# 4 Vektorer og Matricer

Vi vil nu betragte opbyggelsen af et modul for vektorer og matricer. Eftersom en vektor også kan opfattes som en matrix, vil vi i det følgende, når begge dele omtales, udelukkende referere til matricer. For systematik at kunne håndtere matricer, starter vi med at definere en type for lagringsordning Listing 29.

```
type Order = | R | C
```

Listing 29: Typen for order

Typen Order, anvendes til at angive, om en matrix er i rækkefølge (row-major) eller kolonnefølge (column-major)[23]. En vektor, der er lagret i rækkefølge, kan betragtes som den transponeret kolonnefølge vektor. Vi kan derfor nu definere en type for matricer, ved hjælp af en type for vektore i Listing 30.

```
type Vector = V of list<Number> * Order
type Matrix = M of list<Vector> * Order
```

Listing 30: Typen for Matricer

Derudover er det en fordel at kunne kende dimissionen af en matrix. Derfor er der også defineret en type for dimissionen se Listing 31.

```
// Rows x Cols
type Dimension = D of int * int
```

Listing 31: Typen for dimissionen

## 4.1 Matrix operationer

Der vil i denne sektion beskrives en række funktioner som er nødvendige før vi kan betragte nogle funktion for anvendelse af matricer. Da modulet indeholder mange hjælpe funktioner, vil der fokuseres på de funktioner med matematisk relevans.

Det muligt at definere en funktion til at finde dimissionen af en matrix se Listing 32. Funktionen laver et kald til matrixValidMajor som genere en fejl hvis ikke alle vektorer og matrien har samme lagringsordning. matrixVectorLength finder længden på vektor listen en i matricen.

```
    // dimMatrix : Matrix -> Dimension
let dimMatrix (M(vl, o)) =
    if vl = [] then D (0, 0)
    else
    let _ = matrixValidMajor (M(vl, o))
    let d1 = List.length vl
    let d2 = matrixVectorLength (M(vl, o))
    match o with
    | R -> D (d1, d2)
    | C -> D (d2, d1)
```

Listing 32: Funktion til at finde dimissionen af en matrix

[23] [21] *Row- and column-major order*.

Hvis en matrix er gemt som rækkefølge, vil antallet af rækker være længden af en vektor og antallet af kolonner være længden af vektor listen, og omvendt for kolonnefølge.

## 4.2 Matematiske operationer

I denne sektion bør det bemærkes, at flere listings ikke inkluderer fejlhåndtering; dette er udeladt for at forbedre læsbarheden. De funktioner, der anvendes i det implementerede modul, har passende fejltjek, herunder dimensionstjek på matricerne. Den fulde implementering med fejlhåndtering kan findes i appendiks 8.5.

Før vi implementere funktioner til at udføre de ønskede matematiske operationer, vil vi først definere nogle egenskaber matricerne skal opfylde i egenskab 6.

**Egenskab 6** (Vektor Aksiomer)**.**
*Lad* $c, d \in \mathbb{F}$ *og* $v_i \in \mathbb{F}^n$ *for* $i = 1 \dots m$ *så gælder:*[24]

1. $(v_1 + \cdots + v_{m-1}) + v_m = v_1 + (v_2 + \cdots + v_m)$
2. $c \cdot \left( d \cdot \begin{bmatrix} | & & | \\ v_1 & \cdots & v_m \\ | & & | \end{bmatrix} \right) = (c \cdot d) \cdot \begin{bmatrix} | & & | \\ v_1 & \cdots & v_m \\ | & & | \end{bmatrix}$
3. $c \cdot (v_1 + \cdots + v_m) = c \cdot v_1 + \cdots + c \cdot v_m$

### 4.2.1 Skalering af en matrix

Vi begynder med at betragte en funktion til at skalere en matrix se Listing 33.

```
// scalarVector : Number -> Vector -> Vector
let scalarVector (n:Number) (V (nl, o)) =
    V ((List.map (fun x -> x * n) nl), o)

// scalarMatrix : Matrix -> Number -> Matrix
let scalarMatrix (M (vl, o)) n =
    M ((List.map (fun x -> scalarVector n x) vl), o)
```

Listing 33: Funktion til at skalere en matrix

Det at skalere en matrice er svare til at skalere hvert element i matricen. Derfor ved at have en funktion scalarVector, der skalere hvert element i en givet vektor bliver scalarMatrix at skalere hver vektor i en givet matrice. List.map svare til at lave en list comprehension i Python[25].

### 4.2.2 Addition af matricer

```
// addVector : Vector -> Vector -> Vector
let addVector (V (v1, o1)) (V (v2, _)) =
    V ((List.map2 (+) v1 v2), o1)

// addMatrix : Matrix -> Matrix -> Matrix
let addMatrix (M(vl1, o)) (M(vl2, _)) =
```

[24] [12] *Mathematics 1a*, Theorem 7.2 s. 155.
[25] [16] *Python - List Comprehension*.

```
7       M (List.map2 addVector vl1 vl2, o)
8
9   // subVector : Vector -> Vector -> Vector
10  let subVector x y =
11      scalarVector (-one) y |> addVector x
12
13  // sumRows : Matrix -> Matrix
14  let rec sumRows m =
15      if not <| corectOrderCheck m C
16          then sumRows <| correctOrder m C
17      else
18          let zeroVector = vectorOf zero <| matrixVectorLength m
19          let (M(vl, _)) = m
20          matrix [List.fold (addVector) zeroVector vl]
```

Listing 34: Funktion til at addere matricer og substraktion af vektorer

Addition af vektorer reduceres til at udføre additionen elementvis, som vist i Listing 34, ved brug af List-funktionen map2. Vi kan bruge addVector til at definere matrix addition og subtraktion af vektorer. Vektor addition bruges også til at summere rækkerne i en matrix (sumRows), hvilket vil blive anvendt i implementeringen af matrix produkt i næste sektion og Gram-Schmidt-processen i sektion 4.3. Funktionen bliver yderligere beskrevet i definition 2.

**Defination 2** (Summering af rækker i en matrix)**.**
*Lad $A$ være en matrix med $m$ rækker og $n$ søjler, hvor*

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}$$

*Så gælder om sumRows at*

$$sumRows(A) = v = \begin{bmatrix} \sum_{j=1}^{n} a_{1j} \\ \sum_{j=1}^{n} a_{2j} \\ \vdots \\ \sum_{j=1}^{n} a_{mj} \end{bmatrix}$$

*Dermed er $v_i = \sum_{j=1}^{n} a_{ij}$ for $i = 1, 2, \ldots, m$.*

### 4.2.3 Matrix produkt

**Defination 3** (Matrix-Vektor Produkt)**.**
*Lad $A$ være en matrix med $m$ rækker og $n$ søjler, hvor*

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}$$

*og* $\mathbf{v} = (v_1, v_2, \ldots, v_n)^T$ *være en vektor med* $n$ *elementer. Så er matrix-vektor* $Av$ *produktet defineret som*

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix} = \begin{bmatrix} a_{11}v_1 + a_{12}v_2 + \cdots + a_{1n}v_n \\ a_{21}v_1 + a_{22}v_2 + \cdots + a_{2n}v_n \\ \vdots \\ a_{m1}v_1 + a_{m2}v_2 + \cdots + a_{mn}v_n \end{bmatrix} = \begin{bmatrix} \sum_{j=1}^{n} a_{1j}v_j \\ \sum_{j=1}^{n} a_{2j}v_j \\ \vdots \\ \sum_{j=1}^{n} a_{mj}v_j \end{bmatrix}$$

**Sætning 1** (Matrix-Vektor Produkt)**.**
*Lad* $A$ *være en matrix med* $m$ *rækker og* $n$ *søjler, og lad* $\mathbf{v}$ *være en vektor med* $n$ *elementer. Så gælder der*

$$Av = \textit{sumRows} \begin{bmatrix} | & | & & | \\ a_1v_1 & a_2v_2 & \cdots & a_nv_n \\ | & | & & | \end{bmatrix}$$

*Bevis.* Lad $B$ være resultatet af at skalere søjlerne i matrix $A$ med de tilsvarende elementer i vektoren $\mathbf{v}$. Vi har

$$\begin{aligned} B &= \begin{bmatrix} | & | & & | \\ a_1v_1 & a_2v_2 & \cdots & a_nv_n \\ | & | & & | \end{bmatrix} \\ &= \begin{bmatrix} a_{11}v_1 & a_{12}v_2 & \cdots & a_{1n}v_n \\ a_{21}v_1 & a_{22}v_2 & \cdots & a_{2n}v_n \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}v_1 & a_{m2}v_2 & \cdots & a_{mn}v_n \end{bmatrix} \end{aligned}$$

Ved brug af definition 2 for *sumRows* og definition 3 ses det at

$$\textit{sumRows}(B) = \begin{bmatrix} \sum_{j=1}^{n} a_{1j}v_j \\ \sum_{j=1}^{n} a_{2j}v_j \\ \vdots \\ \sum_{j=1}^{n} a_{mj}v_j \end{bmatrix} = A\mathbf{v}$$

□

Vi kan dermed anvende sætning 1 til at definere en funktion matrixVectorProduct til matrix-vektor produkt se Listing 35. Denne funktion skalerer først søjlerne i matricen med de tilsvarende elementer i vektoren og summer derefter rækkerne i matricen.

```
1 // matrixVectorProduct : Matrix -> Vector -> Matrix
2 let rec matrixVectorProduct (M(vl, _)) (V(nl, _)) =
3     M (List.map2 (fun mc n -> scalarVector n mc) vl nl, C)
4     |> sumRows
```

Listing 35: Funktion til matrix-vektor produkt

**Defination 4** (Matrix produkt)**.**
*Lad $A \in \mathbb{F}^{m\times n}$ og $B \in \mathbb{F}^{n\times \ell}$. Lad søjlerne i $B$ er være givet ved $b_1, \ldots, b_\ell \in \mathbb{F}^n$, dermed*[26]

$$B = \begin{bmatrix} | & & | \\ b_1 & \cdots & b_\ell \\ | & & | \end{bmatrix}.$$

*Så defineres matrixproduktet som*

$$A \cdot B = \begin{bmatrix} | & & | \\ A \cdot b_1 & \cdots & A \cdot b_\ell \\ | & & | \end{bmatrix}.$$

Ud fra definition 4 ses det, at funktionen matrixProduct i Listing 36 til at tage produktet af to matricer, ved at udfører matrix-vektor produkt på hver søjle i matricen. Da matrixVectorProduct evaluere til en matrix, skal der bruges en funktion til at konvertere matricen til en vektor, hvilket matrixToVector gør.

```
// matrixProduct : Matrix -> Matrix -> Matrix
let rec matrixProduct a (M(vlb, _)) =
    let product = List.map (
        fun bv -> matrixVectorProduct a bv |> matrixToVector ) vlb
    M(product, C)
```

Listing 36: Funktion til at tage produktet af to matricer

### 4.2.4 Projektion af en vektor

**Defination 5** (Projektion af en vektor)**.**
*Projektionen af en vektor på en linje defineres i som følgende, hvor $Y = span\{y\}$*[27]*.*

$$proj_Y : V \to V, \quad proj_Y(x) = \frac{\langle x, y\rangle}{\langle y, y\rangle} y \tag{3}$$

*hvor det standard indre produkt er defineret som:*

$$\langle x, y\rangle = y^* x = \sum_{k=1}^{n} x_k \overline{y}_k \tag{4}$$

For at kunne projektere en vektor, som defineret i destination 5 skal vi først kunne konjugere en vektor. Funktionen conjugateVector konjugerer elementerne i en vektor. Derudover defineres en funktion til at multiplicere to vektorer elementvis (vectorMulElementWise).

```
// vectorMulElementWise : Vector -> Vector -> Vector
let vectorMulElementWise (V(u, o1)) (V(v, o2)) =
    V (List.map2 (*) u v, o1)

// conjugateVector : Vector -> Vector
```

[26] [12] *Mathematics 1a*, Definition 7.12 s. 162.
[27] [9] *Mathematics 1b - Functions of several variables*, s. 40.

```
let conjugateVector (V(v, o)) =
    V (List.map conjugate v, o)

// innerProduct : Vector -> Vector -> Number
let innerProduct u v =
    let (V(w, _)) = conjugateVector v |> vectorMulElementWise u
    List.fold (+) zero w

// proj : Vector -> Vector -> Vector
let proj y x =
    scalarVector (innerProduct x y / innerProduct y y) y
```

Listing 37: Funktioner til at projicere en vektor på en anden

Evalueringen af det standard indre produkt innerProduct mellem to vektore, bliver derfor at konjugere den ene vektor og derefter multiplicere elementvis med den anden vektor. Hvortil summen af elementerne i den resulterende vektor er det indre produkt.

Til sidst kan funktionen proj skrives direkte som den er defineret i definition 5.

## 4.3 Gram-Schmidt

Vi kan nu betragte implementeringen af Gram-Schmidt processen. Denne proces kan anvendes rekursivt til at finde en ortonormal basis for et underrum udspændt af en liste af vektorer $v_1, v_2, \ldots, v_n$. Processen kan implementeres rekursivt idet de nye vektorer $w_k$ for $k = 2, 3, \ldots, n$ konstrueres baseret på alle de tidligere vektorer $w_1, \ldots, w_{k-1}$.

Før vi implementerer Gram-Schmidt processen, er vi dog begrænset af vores Number type fra Listing 15, idet $x \in \{\text{Number}\} \not\Rightarrow \sqrt{x} \in \{\text{Number}\}$. Derfor vil vi ikke normalisere vektorerne, hvilket medfører, at vi kun vil finde en ortogonal basis, fremfor en ortonormal basis.

Der er også 2 egenskaber 7 som vi i sektion 5.2.2 vil teste for at sikre at vores implementering af Gram-Schmidt processen er korrekt.

**Egenskab 7** (Gram-Schmidt)**.**
*Lad $w_1, w_2, \ldots, w_n$ være de nye vektore som er dannes udfra gram-schmidt processen på $v_1, v_2, \ldots, v_n$ som er lineært uafhængige vektorer. Så gælder der:*

1. *$w_1, w_2, \ldots, w_n$ er ortogonale.*
2. *$w_1, w_2, \ldots, w_n$ udspænder det samme underrum som $v_1, v_2, \ldots, v_n$. dvs. $span\{v_1, v_2, \ldots, v_n\} = span\{w_1, w_2, \ldots, w_n\}$.*

```
// orthogonalBacis : Matrix -> Matrix
let orthogonalBacis m =
    if not <| corectOrderCheck m C
    then orthogonalBacis (correctOrder m C)
    else

    // Gram_Schmidt : Matrix -> (Vector list -> Matrix) -> Matrix
    let rec Gram_Schmidt vm acc_wm =
        match acc_wm [], vm with
        | x, M([], _) -> x
        | M([], _), M(v1::vrest, o) ->
            Gram_Schmidt (M(vrest,o))
```

```
13              <| fun x -> extendMatrix (M([v1], C)) x
14          | M(w, _), M(vk::vrest, o) ->
15              let (V(wk, _)) = vk - sumProj w vk
16              Gram_Schmidt (M(vrest,o))
17              <| fun x -> extendMatrix (acc_wm wk) x
18
19      // sumProj : Vector list -> Vector -> Vector
20      and sumProj w vk =
21          List.map (fun x -> proj x vk) w
22          |> matrix
23          |> sumRows
24          |> matrixToVector
25
26      Gram_Schmidt m (fun _ -> M([], C))
```

Listing 38: Dannelsen af en ortogonal basis, ved hjælp af Gram-Schmidt processen

Listing 38 viser implementeringen af Gram-Schmidt processen. Lavet udfra beskrivelse side 45 i 'Mathematics 1b'[28].

Funktionen `sumProj` tager en liste med vektorer $w$, som i Gram-Schmidt-processen er de tidligere behandlede vektorer $w_1, \ldots, w_{k-1}$, og en vektor $v_k$ som er den $k$'te vektor. $v_k$ projiceres på alle vektorerne i $w$, hvorefter der tages summen af disse projektioner.

Funktionen Gram_Schmidt, tager en matrix hvor søjlerne er de vektore som ønskes at finde en ortogonal basis for. Der udover tager den en akkumulerende funktion som indeholder de behandlede vektorer. Hvis der ikke er flere vektorer i matricen, gives den akkumulerede funktion. Hvis der ikke er nogle vektorer i akkumulatoren, tages den første vektor fra matricen og tilføjes til akkumulatoren. Hvis der er vektorer i både akkumulatoren og matricen, kaldes sumProj på den akkumulerede liste og den første vektor i matricen. Resultatet trækkes fra den første vektor i matricen, og dette bliver den nye vektor som tilføjes til akkumulatoren.

Funktionen orthogonalBacis tager en matrix og tjekker om matricen er i kolonnefølge, hvis ikke kalder funktionen sig selv, med den korrekte lagringsordning. Ellers kaldes Gram_Schmidt med matricen og en tom akkumulator. Resultatet bliver derfor en matrix med en ortogonal basis for underrummet udspændt af de givne vektorer, givet at vektorerne er lineært uafhængige.

## 4.4 Række-echelon form

I forbindelse med udførelsen af PBT til Gram-Schmidt metoden vil vi blandt andet anvende række-echelon form til at sikre os, at et sæt af vektorer er lineært uafhængige. Derfor vil vi i denne sektion betragte implementeringen af række-echelon form, ud fra „Algorithm 9 for computing a row echelon form of a matrix“ angivet på side 142 af noterne til „01001 Mathematics 1a“[29]. Efter at have udført PBT af Gram-Schmidt metoden, slog den i første omgang fejl, hvilket efter nærmere undersøgelse viste sig at være grundet en mindre fejl i Algorithm 9. Linje 14 burde være „b ← the j'th entry of the i-th row of B“. Listing 39 viser implementeringen af række-echelon form, lavet ud fra psudokoden „Algorithm 9“.

```
1 // rowEchelonForm : Matrix -> Matrix
2 let rec rowEchelonForm A =
```

[28] [9] *Mathematics 1b - Functions of several variables*, s. 45.
[29] [12] *Mathematics 1a*, s. 142.

```
    if not <| corectOrderCheck A R then rowEchelonForm (correctOrder A R)
    else
    let (D(r, c)) = dimMatrix A
    match A with
    |M([], _) -> A
    | _ when isZeroMatrix A -> A
    | M(v::_, _) when r = 1 -> firstNonZero v |> inv |> scalarMatrix A
    | M(_, o) ->
        let (i, j) = firstNonZeroIndexMatrix A 0 (-1 ,-1)
        let (M(B, _)) = swapFirstWith A i
        let b = List.head B |> firstNonZero
        let (M(B, _)) = scalarIthVector (inv b) 0 (M(B, o))
        let R1 = List.head B
        let R2m = List.tail B
        let B = rowOps j 1 r R1 (M(R2m, o))
        let (M(Cm, _)) = rowEchelonForm B
        M(R1::Cm, o)

// rowOps: int -> int -> int -> Vector -> Matrix -> Matrix
and rowOps coloumn i nrows R1 acc_m  =
    if i >= nrows then acc_m else
    let Ri = getMatrixIthVector (i - 1) acc_m
    let b = getVectorIthNumber coloumn Ri
    rowOps coloumn (i + 1) nrows R1 <| replaceMatrixIthVector (i-1) acc_m (Ri
    - b * R1)
```

Listing 39: Funktion til at finde række-echelon form

## 4.5 Lineært ligningssystem

Som den sidste del i anvendelsen af matricer vil vi beskrive, hvordan vi kan løse et lineært ligningssystem af formen $A\mathbf{x} = \mathbf{b}$. Ved at foretage række-echelon form på totalmatricen $[A \mid \mathbf{b}]$ kan vi finde en løsning til ligningssystemet, hvis ranken af $A$ er forskellig fra ranken af den totale matrix[30].

Efter at have foretaget række-echelon form på totalmatricen, skal vi tage matrix-vektor-produktet af ref($A$) og $\mathbf{x}$ for at kunne have $m$ ligninger med $n$ ubekendte, hvis vi antager, at $A \in \mathbb{F}^{m \times n}$.

Den nemmeste tilgang vil være at introducere to nye typer for at repræsentere en vektor og matrix, som kan indeholde variable. Dette kan gøres ved hjælp af vores type for udtryk fra Listing 19. Vi definerer derfor følgende typer i Listing 40.

```
type ExprVector = list<Expr<Number>>
type ExprMatrix = list<ExprVector>
```

Listing 40: Typer for at repræsentere en vektor og matrix med variable

Det er her undladt at definere lagringsordningen for matricerne af udtryk, da vi ikke bruger dem til mere end en mellemregning i løsningen af ligningssystemet. Men alle funktioner, vi har lavet i Listing 30, ville man også kunne lave for udtryksmatricer, hvis man anvender en lagringsordning.

Ved at have udtryksmatricer defineret som en liste af udtryk, kan vi anvende den samme teknik til at foretage matrix-vektor-produktet mellem ref($A$) og $\mathbf{x}$, som vi benyttede i Listing 35. Dette

[30] [12] *Mathematics 1a*, Corollary 6.27 s. 146.

vil resultere i en liste af udtryk, som vi ved hjælp af funktionen isolateX i Listing 28 kan bruge til at isolere $x_i$ i den $i$'te ligning ved at sætte den lig med $b_i$ og indsætte det i ligningerne 1 til $i-1$.

Den fulde implementering er at finde i appendiks 8.5. Det er værd at bemærke, at denne implementering vil resultere i en vektor af `Number`. Derfor vil den fejle, hvis der er flere ubekendte end ligninger, hvilket betyder, at koefficientmatricen $A$ skal have fuld rank.

# 5 PBT af programmet

Vi skal nu validere vores program ved hjælp af PBT, ved at lave funktioner, som kan undersøge de egenskaber der løbende er blevet beskrevet i rapporten.

## 5.1 PBT af udtryk

Før vi begynder at udføre PBT på alle egenskaber vedrørende udtryk, skal vi først bygge en generator for vores talmængde og udtryk. Da vi ikke altid kan sammenligne, om to udtrykstræer er ækvivalente, vil vi i alle vores PBT, hvor vi ønsker at sammenligne træer, gøre dette ved brug af et miljø, som indeholder værdier for variablene, og derefter evaluere udtrykene ved brug af eval-funktionen.

Vi begynder med at definere en række generatorer, som kan generere blade i vores udtrykstræer i Listing 41.

```
let max = 3
let min = -3

// noneZeroGen: Gen<int>
let noneZeroGen =
    Gen.oneof [
        Gen.choose(1, max) ;
        Gen.choose(min, -1)]

// numberGen: Gen<Number>
let numberGen =
    Gen.oneof [
        Gen.map2 (fun x y -> newRational(x, y) |> Rational |> tryReduce ) (Gen.choose(min, max)) noneZeroGen;
        Gen.map (fun x -> Int x) (Gen.choose(min, max));
        Gen.map4 (fun a b c d -> newComplex (newRational(a, b), newRational(c, d)) |> Complex |> tryReduce ) (Gen.choose(min, max)) noneZeroGen (Gen.choose(min, max)) noneZeroGen]

// numberInExprGen: Gen<Expr<Number>>
let numberInExprGen =
    Gen.map (fun x -> N x) numberGen

// randomListElement: list<'a> -> Gen<'a>
let randomListElement xlist =
    gen { let! i = Gen.choose(0, List.length xlist - 1)
        return xlist.[i] }

// variableGen: list<char> -> Gen<Expr<'a>>
let variableGen xlist = Gen.map X (randomListElement xlist)

// leafGen: list<char> -> Gen<Expr<Number>>
let leafGen xlist =
    if xlist <> [] then
        Gen.oneof [numberInExprGen; variableGen xlist]
    else
        numberInExprGen

// onlyIntleafGen: list<char> -> Gen<Expr<Number>>
let onlyIntleafGen xlist :Gen<Expr<Number>> =
```

```
38      if xlist <> [] then
39          Gen.oneof [Gen.map (fun x -> N <| Int x) (Gen.choose(-10, 10));
        variableGen xlist]
40      else
41          Gen.map (fun x -> N <| Int x) (Gen.choose(-10, 10))
42
43  // charsSeqGen: char -> char -> seq<Gen<char>>
44  let charsSeqGen c1 c2 = seq { for c in c1 .. c2 do
45                                  yield gen { return c} }
46
47  // charGen: Gen<char>
48  let charGen = gen { return! Gen.oneof (charsSeqGen 'A' 'Z')}
49
50  // smallEnvGen: Gen<Map<char, Number> * list<char>>
51  let smallEnvGen =
52      gen {
53          let! i = Gen.choose (0, 5)
54          let! xlist = Gen.listOfLength i charGen
55          let! ns = Gen.listOfLength i numberGen
56          return (Map.ofList (List.zip xlist ns), xlist) }
57
58  // exprGen: 'a -> int -> ('a -> Gen<Expr<'b>>) -> Gen<Expr<'b>>
59  let rec exprGen xlist n leafType =
60      if n = 0 then
61          leafType xlist
62      else
63          Gen.oneof [
64              // leaf occurs twice becourse leaf is X or N giving the same
        probability for each expression
65              leafType xlist;
66              leafType xlist;
67              Gen.map2 (fun x y -> Add (x, y)) (exprGen xlist (n/2) leafType) (
        exprGen xlist (n/2) leafType);
68              Gen.map2 (fun x y -> Mul (x, y)) (exprGen xlist (n/2) leafType) (
        exprGen xlist (n/2) leafType);
69              Gen.map2 (fun x y -> Div (x, y)) (exprGen xlist (n/2) leafType) (
        exprGen xlist (n/2) leafType);
70              Gen.map2 (fun x y -> Sub (x, y)) (exprGen xlist (n/2) leafType) (
        exprGen xlist (n/2) leafType);
71              Gen.map (fun x -> Neg x) (exprGen xlist (n/2) leafType)]
72
73
74  type SmallEnv = Map<char, Number> * char list
75  type SmallEnvGen =
76      static member SmallEnv() =
77          {new Arbitrary<SmallEnv>() with
78              override _.Generator = smallEnvGen
79              override _.Shrinker _ = Seq.empty}
80
81  type NumberGen =
82  static member Number() =
83      {new Arbitrary<Number>() with
84          override _.Generator = numberGen
85          override _.Shrinker _ = Seq.empty}
```

Listing 41: Generatorene anvendt til PBT af udtryk

Vi begynder med at definere to variable, som alle funktioner har til rådighed, `max` og `min`, som definerer intervallet for de heltal, der kan anvendes i genereringen af Numbers. Den første

funktion, noneZeroGen, er en generator for et tilfældigt heltal fra sættet $S_1$:

$$S_1 = \{x \mid x \in \mathbb{Z} \setminus \{0\}, \quad \texttt{min} \leq x \leq \texttt{max}\}.$$

Vi kan dermed bruge noneZeroGen, da vi ikke ønsker at generere rationale tal med nævneren 0. For at definere en generator for Numbers i form af numberGen, som genererer et gyldigt Number fra sættet $S_5$:

$$\begin{aligned} S_2 &= \{x \mid x \in \mathbb{Z}, \quad \texttt{min} \leq x \leq \texttt{max}\} \\ S_3 &= \left\{ \frac{x}{y} \mid x \in S_1, y \in S_2 \right\} \\ S_4 &= \{x + yi \mid x, y \in S_3\} \\ S_5 &= S_2 \cup S_3 \cup S_4 \end{aligned}$$

Funktionen numberInExprGen er en generator, som ved brug af numberGen konverterer et `Number` til et `Expr<Number>`. Det er disse generatorer, der anvendes til at generere tal til vores talmængde.

Dernæst kommer nogle generatorer til generering af variable. Først har vi randomListElement, som tager en liste af en vilkårlig type og udvælger et tilfældigt element fra listen. Derudover har vi variableGen, som tager en liste af karakterer og bruger randomListElement til at udvælge en af karaktererne og konvertere den til et `Expr<Number>`.

Dermed er det nu muligt at lave en generator, som kan generere enten et tal eller en variabel fra konstruktørerne af `Expr<Number>`. Disse to konstruktører er også bladene i vores udtrykstræer. leafGen genererer et tilfældigt blad i vores udtrykstræ ud fra en liste af karakterer. onlyIntleafGen fungerer ud fra det samme princip, men genererer kun heltal fra intervallet -10 til 10.

Funktionerne charGen og charsSeqGen er generatorer, som sammen genererer en tilfældig karakter mellem 'A' og 'Z'. Endelig har vi en miljøgenerator, smallEnvGen, som genererer et par af et miljø samt liste af variable i miljøet. Miljøet indeholder variable og deres tilsvarende værdier.

Den sidste generator, exprGen, vil i vores PBT tage en liste af variable, som den må generere blade ud fra, samt hvilken generator der skal anvendes til at generere bladene. Derudover vil den maksimale dybde af udtrykket være $\log_2(n)$.

Til sidst defineres tre typer, hvor `SmallEnvGen` og `NumberGen` gør det muligt inden kørsel af vores PBT at registrere generatorerne.

Vi har dermed nu lavet fundamentet, til at kunne teste vores egenskaber vedrørende udtryk.

### 5.1.1 Tal modulet

De 6 egenskaber fra 1 kan nu testes med PBT. Egenskaberne kan oversættes direkte til funktioner, grundet de overskrivninger vi har lavet på number typen i Listing 42.

```
1 #r "../bin/Release/net7.0/main.dll"
2 #r "nuget: FsCheck"
3 open FsCheck
4 open Number
5 open Generators
```

```
6
7  Arb.register<NumberGen>()
8
9  // Addition og multiplikation er associative
10 let associative (a:Number) (b:Number) (c:Number) =
11     a + (b + c) = (a + b) + c && a * (b * c) = (a * b) * c
12
13 // Addition og multiplikation er kommutative
14 let commutative (a:Number) (b:Number) =
15     a + b = b + a && a * b = b * a
16
17 // Distributivitet af multiplikation over addition
18 let distributive (a:Number) (b:Number) (c:Number) =
19     a * (b + c) = a * b + a * c
20
21 // Addition og multiplikation har et neutralt element
22 let neutralAdditive (a:Number) =
23     a + zero = a && a * one = a
24
25 // Omvendt funktion eksisterer til addition
26 let inverseAdditive (a:Number) =
27     a + (-a) = zero
28
29 // Omvendt funktion eksisterer til multiplikation
30 let inverseMultiplicative (a:Number) =
31     let res =
32         try
33             if a * inv a = one then 1 else 0
34         with
35             | :? System.DivideByZeroException -> 2
36     (res = 1 || res = 2)
37     |> Prop.classify (res = 1) "PropertyHolds"
38     |> Prop.classify (res = 2) "DivideByZeroExceptions"
39
40 let _ = Check.Quick associative
41 let _ = Check.Quick commutative
42 let _ = Check.Quick distributive
43 let _ = Check.Quick neutralAdditive
44 let _ = Check.Quick inverseAdditive
45 let _ = Check.Quick inverseMultiplicative
```

Listing 42: *numberPBT.fsx* - funktioner til test af egenskaberne i 1

I inverseMultiplicative anvender vi „Prop.classify“ til at tillade, at egenskaben kan slå bestemte fejl. I dette tilfælde tillades det, at der opstår en „DivideByZeroException“. Dette sker, fordi generatoren godt kan generere 0, som der ikke kan tages en invers af. For at teste ovensånde funktioner, køres *.fsx* filen, og outputtet kan ses i Listing 43.

```
Ok, passed 100 tests.
Ok, passed 100 tests.
Ok, passed 100 tests.
Ok, passed 100 tests.
Ok, passed 100 tests.
Ok, passed 100 tests.
94% PropertyHolds.
6% DivideByZeroExceptions.
```

Listing 43: Outputtet fra PBT af Number typer, ved kørsel af Listing 42

Dermed viser testen at alle egenskaberne fra 1 holder.

### 5.1.2 Homomorfisme af evaluering

Vi skal nu teste egenskaben fra 2. Testen er lavet i Listing 44. Dette er gjort ved, at der genereres et tilfældigt miljø, hvori der samples to udtryk ud fra variablene i miljøet. For alle operatorer testes det, om de overholder egenskaben om homomorfisme.

```
// evalOperation: Expr<Number> -> Expr<Number> -> Map<char, Number> -> (Expr<
    Number> -> Expr<Number> -> Expr<Number>) -> bool
let evalOperation e1 e2 env f =
eval (f e1 e2) env = (getNumber <| f (eval e1 env |> N) (eval e2 env |> N))

let evalPBT ((env ,xlist):SmallEnv) =
    let result =
        try
            let exprList = Gen.sample 1 2 (exprGen xlist 10 leafGen)
            let e1::[e2] = exprList
            let prop = evalOperation e1 e2 env
            let negation = eval (-e1) env = - eval e1 env
            if negation && prop ( + ) && prop ( - ) && prop ( * ) && prop ( /
    ) then 1 else 0
        with
        | :? System.DivideByZeroException as _ -> 2
        | :? System.OverflowException as _ -> 3
    (result = 1 || result = 2 || result = 3)
    |> Prop.classify (result = 1) "Property Holds"
    |> Prop.classify (result = 2) "DivideByZeroExceptions"
    |> Prop.classify (result = 3) "OverflowException"
```

Listing 44: PBT af egenskaben omkring Homomorfisme fra 2

Outputtet fra testen kan ses i Listing 45. Testen viser, at egenskaben holder for alle operationer.

```
> Arb.register<SmallEnvGen>()
- let _ = Check.Quick evalPBT;;
Ok, passed 100 tests.
76% Property Holds.
18% DivideByZeroExceptions.
6% OverflowException.
```

Listing 45: Outputtet fra PBT af egenskaben 2

Vi ser her en større mængde af tests, som bliver klassificeret som „DivideByZeroExceptions“. Dette skyldes, at vores generator godt kan generere udtryk som $1/(0 \cdot X)$ og lignende, hvilket ikke er et lovligt udtryk. Derudover forekommer der også „OverflowExceptions“, hvilket skyldes, at ved evaluering af udtryk kan vi godt ende med en kombination af operationer, som giver rationale tal, der, som beskrevet i sektion 3.1.1, kan resultere i en overflow. Samme form for klassificeringer vil vi løbende se i de kommende tests.

### 5.1.3 Simplifikation af udtryk

Det er vigtigt, at vores simplifikation altid er lig med det oprindelige udtryk. Vi kan derfor teste egenskab 4 ved at simplificere et udtryk og sammenligne evalueringen af det med det oprindelige udtryk ved hjælp af et miljø. Dette er gjort i Listing 46.

```
// compareSimpExpr: Map<char, Number> -> Expr<Number> -> bool
let compareSimpExpr env (e:Expr<Number>) =
    eval (simplifyExpr e) env = eval e env

// simpEqualEval: SmallEnv -> int
let simpEqualEval (env, xlist) =
    try
        if Gen.sample 1 1 (exprGen xlist 10 leafGen)
            |> List.head
            |> compareSimpExpr env
        then 1 else 0
    with
        | :? System.DivideByZeroException as _ -> 2
        | :? System.OverflowException as _ -> 3

// simpPBT: SmallEnv -> Property
let simpPBT (se:SmallEnv) =
    let result = simpEqualEval se
    (result = 1 || result = 2 || result = 3)
    |> Prop.classify (result = 1) "Equal"
    |> Prop.classify (result = 2) "DivideByZeroExceptions"
    |> Prop.classify (result = 3) "OverflowException"
```

Listing 46: PBT af egenskaben 4

Outputtet fra testen kan ses i Listing 47. Testen viser, at egenskaben holder.

```
> Arb.register<SmallEnvGen>()
- let _ = Check.Quick simpPBT;;
Ok, passed 100 tests.
94% Equal.
6% DivideByZeroExceptions.
```

Listing 47: Outputtet fra PBT af egenskaben 4

### 5.1.4 Invers morfisme mellem infix og prefix

Nu skal det undersøges, hvorvidt egenskab 3 holder. Vi beskrev i Listing 24 den inverse funktion til tree-funktionen, som benyttes til at opskrive egenskaben i Listing 48.

```
// generatesCorrectTree: Map<char, Number> -> Expr<Number> -> bool
let generatesCorrectTree env (e:Expr<Number>) =
    eval e env = eval
        (simplifyExpr e
        |> infixExpression
        |> tree
        |> infixExpression
        |> tree ) env

// treeEqualEval: SmallEnv -> int
let treeEqualEval (env, xlist) =
    try
        if Gen.sample 1 1 (exprGen xlist 10 onlyIntleafGen)
            |> List.head
            |> generatesCorrectTree env
        then 1 else 0
    with
```

```
        | :? System.DivideByZeroException as _ -> 2
        | :? System.OverflowException as _ -> 3

// treePBT: SmallEnv -> Property
let treePBT (se:SmallEnv) =
    let result = treeEqualEval se
    (result = 1 || result = 2 || result = 3)
    |> Prop.classify (result = 1) "Equal"
    |> Prop.classify (result = 2) "DivideByZeroExceptions"
    |> Prop.classify (result = 3) "OverflowException"
```

Listing 48: PBT af egenskaben 3

Funktionen generatesCorrectTree tager et miljø samt et udtryk og undersøger, om:

$$\text{eval}(e) = \text{eval}(\text{tree}(\text{tree}^{-1}(\text{tree}(\text{tree}^{-1}(\text{simplifyExpr}(e))))))) \tag{5}$$

Det er gjort på denne måde, da det ikke er muligt igennem vores program at generere matematiske udtryk i infix-notation, som med sikkerhed har gyldig notation. I stedet vil vi anvende vores udtryksgenerator til at generere et udtryk, som vi så ved hjælp af et miljø kan tjekke, at egenskaben 3 holder for alle $x \in$ `Expr<Number>`.

Udtrykket bliver simplificeret, før det konverteres til infix-notation, da der er valgt ikke at placere parenteser rundt om tal i udtrykket, for visualiseringens skyld. Dette betyder, at hvis udtrykket ikke simplificeres, kan der skabes udtryk som `Neg(N -1)`, som i infix-notation er $--1$, hvilket ikke er en gyldig notation uden parenteser. Simplifikationen sørger for, at negative tal repræsenteres ved negationen af et positivt tal.

Outputtet fra kørsel af testen kan ses i Listing 49. Testen viser som forventet, at egenskaben holder for alle udtryk.

```
> Arb.register<SmallEnvGen>()
- let _ = Check.Quick treePBT;;
Ok, passed 100 tests.
94% Equal.
6% DivideByZeroExceptions.
```

Listing 49: Outputtet fra PBT af egenskaben 3

### 5.1.5 Differentiering af udtryk

Selv om differentieringsfunktionen diff er implementeret ud fra egenskaberne i 5, er det stadig vigtigt at teste, om funktionen overholder de samme egenskaber. Dette er gjort i *diffPBT.fsx*, se Listing 50.

```
#r "../bin/Release/net7.0/main.dll"
#r "nuget: FsCheck"
open FsCheck
open Expression
open Number
open Generators
open SymbolicManipolation
open Differentiation
Arb.register<SmallEnvGen>()
```

```
Arb.register<NumberGen>()

// Skaleringsreglen
let scaleP f a env dx =
    eval (diff (a * f) dx) env = eval (a * diff f dx) env

// Sumreglen
let additionP f g env dx =
    eval (diff (f + g) dx) env = eval (diff f dx + diff g dx) env

// Subtraktionsreglen
let subtarctionP f g env dx =
    eval (diff (f - g) dx) env = eval (diff f dx - diff g dx) env

// Produktreglen
let multiplicationP f g env dx =
    eval (diff (f * g) dx) env = eval (diff f dx * g + f * diff g dx) env

// Kvotientreglen
let divisionP f g env dx =
    eval (diff (f / g) dx) env = eval ((g * diff f dx - f * diff g dx) / (g *
    g)) env

// Property test of differentiation
let diffPBT ((env ,xlist):SmallEnv) (a:Number)=
    let result =
        try
            let a = N a
            let dx = if xlist <> [] then List.head xlist else 'X'
            let exprList = Gen.sample 1 2 (exprGen xlist 10 leafGen)
            let e1::[e2] = exprList
            let e1 = simplifyExpr e1
            let e2 = simplifyExpr e2
            if
                scaleP e1 a env dx && scaleP e2 a env dx &&
                additionP e1 e2 env dx &&
                subtarctionP e1 e2 env dx &&
                multiplicationP e1 e2 env dx &&
                divisionP e1 e2 env dx
            then 1
            else 0
        with
        | :? System.DivideByZeroException as _ -> 2
        | :? System.OverflowException as _ -> 3
    (result = 1 || result = 2 || result = 3)
    |> Prop.classify (result = 1) "Propertys Holds"
    |> Prop.classify (result = 2) "DivideByZeroExceptions"
    |> Prop.classify (result = 3) "OverflowException"

printfn "Differentiation property based testing"
let _ = Check.Quick diffPBT
```

Listing 50: *diffPBT.fsx* - funktioner til test af egenskaberne i 5

Outputtet fra kørsel af testen kan ses i Listing 51. Testen viser, at egenskaberne holder.

```
Differentiation property based testing
Ok, passed 100 tests.
72% Property Holds.
```

```
15% DivideByZeroExceptions.
13% OverflowException.
```

Listing 51: Outputtet fra PBT af differentiering af udtryk

## 5.2 PBT af vektorer og matricer

Ligesom ved PBT af udtryk begynder vi med at lave end generatorer for matricer.

```
// vectorGen : int -> Gen<Vector>
let vectorGen n =
    Gen.listOfLength n numberGen |> Gen.map (fun x -> vector x)

// matrixGen : Gen<Matrix>
let matrixGen =
    gen {
        let! row = Gen.choose(1, 6)
        let! col = Gen.choose(1, 6)
        let! vectors = Gen.listOfLength col (vectorGen row)
        return matrix vectors
    }

type MatrixGen =
    static member Matrix() =
        {new Arbitrary<Matrix>() with
            override _.Generator = matrixGen
            override _.Shrinker _ = Seq.empty}
```

Listing 52: Generatorerne anvendt til PBT af matrixoperationer

Først defineres vectorGen, som laver en liste ved hjælp af den tidligere definerede funktion numberGen. Den laver en liste med en given længde af tilfældige tal fra vores talmængde. Dernæst passes listen videre til vector-funktionen, som laver en søjlelagret vektor ud fra listen. Funktionen matrixGen genererer en matrix af tilfældig størrelse, hvor antallet af rækker og kolonner er mellem 1 og 6 ved hjælp af vectorGen. Til sidst kan vi definere en type for matrixgeneratorerne, som gør det muligt at registrere dem før kørsel af PBT.

### 5.2.1 PBT af matrix operationer

Det er nu muligt at opstille nogle PBT af der sikre at matricerne overholder matematiske egenskaber i sætning 6.

Vi kan dermed nu lave definere egenskaberne fra 6 som nogle funktioner i Listing 53.

```
//vectorCom : Matrix -> bool
let vectorCom m =
    sumRows m = sumRows (flip m)

//vectorScalarAss : Matrix -> Number -> Number -> bool
let vectorScalarAss (m:Matrix) (n1:Number) (n2:Number) =
    n1 * (n2 * m) = (n1 * n2) * m

//vectorAssCom : Matrix -> Number -> bool
let vectorAssCom m (c:Number) =
```

```
11      c * (sumRows m) = sumRows (c * m)
```

Listing 53: Egenskaberne fra sætning 6 som funktioner

Listing 54 viser outputtet fra kørsel af testene. Testene viser, at egenskaberne holder for alle matricer.

```
> Arb.register<MaxtrixGen>()
- let _ = Check.Quick vectorCom
- let _ = Check.Quick vectorScalarAss
- let _ = Check.Quick vectorAssCom;;
Ok, passed 100 tests.
Ok, passed 100 tests.
Ok, passed 100 tests.
```

Listing 54: Outputtet fra PBT af vektor Listing 53

### 5.2.2 PBT af Gram-Schmidt

Udfordringen ved at lave en PBT af Gram-Schmidt er, at vektorsættet skal være lineært uafhængigt. Derfor laves der en generator, som ved at udføre tilfældige rækkeoperationer på en diagonal matrix med $m$ rækker og $n$ søjler, hvorom det gælder, at $n \geq m$. Dette sikrer os, at rækkerne i matricen er lineært uafhængige. Hertil vil den transponerede matrix have lineært uafhængige søjler, som vi kan bruge til at teste Gram-Schmidt processen[31]. Listing 55 viser de forskellige generatorer, som anvendes til PBT af Gram-Schmidt-processen.

```
1  // performRowOperationGen: Matrix -> Gen<Matrix>
2  let performRowOperationGen m =
3      let (D(n, _)) = dimMatrix m
4      gen {
5          let! i = Gen.choose(1, n)
6          let! j = match i with
7                       | 1 -> Gen.choose(2, n)
8                       | _ when i = n -> Gen.choose(1, n-1)
9                       | _ -> Gen.oneof [Gen.choose(1, i-1); Gen.choose(i+1, n)]
10         let! a = numberGen
11         return rowOperation i j a m }
12
13 // multipleRowOperationsGen: Matrix -> int -> Gen<Matrix>
14 let rec multipleRowOperationsGen m count =
15     if count <= 0 then Gen.constant m
16     else
17         gen {
18             let! newMatrix = performRowOperationGen m
19             return! multipleRowOperationsGen newMatrix (count - 1)
20         }
21
22 // getDiagonalMatrixGen: int -> Gen<Matrix>
23 let getDiagonalMatrixGen maxRows =
24     gen {
25         let! m = Gen.choose(2, maxRows)
26         let! n = Gen.choose(m, m + 3)
27         return fullrankedDiagonalMatrix m n }
```

[31] [18] *Rank (linear algebra).*

```
28
29 // getIndependentBasisGen: Gen<Matrix>
30 let getIndependentBasisGen =
31     gen {
32         let! A = getDiagonalMatrixGen 5
33         let! numberOfOperations = Gen.choose(1, 10)
34         let! span = multipleRowOperationsGen A numberOfOperations
35         return span |> transposeMatrix  }
36
37 type IndependentBasisMatrix = Matrix
38 type IndependentBasisMatrixGen =
39     static member IndependentBasisMatrix() =
40         {new Arbitrary<Matrix>() with
41             override _.Generator = getIndependentBasisGen
42             override _.Shrinker _ = Seq.empty}
```

Listing 55: Generatorene anvendt til PBT af Gram-Schmidt

Funktionen performRowOperationGen tager en matrix, hvori der udvælges to tilfældige rækker, $i$ og $j$, hvorefter der udføres en rækkeoperation på $R_j$, således at $R_j \leftarrow R_j - aR_i$, hvor $a$ er et tilfældigt tal. multipleRowOperationsGen gentager denne proces et givet antal gange. Funktionen getDiagonalMatrixGen laver en diagonal matrix med $m$ rækker og $n$ søjler, hvor $n \geq m$. getIndependentBasisGen genererer en matrix, hvis søjler er lineært uafhængige. Vi kan dermed definere en funktionen til at undersøge om en matrix er en ortogonal basis, som vist i Listing 56.

```
1 // isOrthogonalBacis : Matrix -> bool
2 let rec isOrthogonalBacis m =
3     (transposeMatrix m |> conjugateMatrix) * m |> isDiagonalMatrix
```

Listing 56: Funktion til at tjekke om søjlerne i en matrix er en ortogonal basis

Dernæst skal vi bruge en funktion til at tjekke, om en matrix er en ortogonal basis. isOrthogonalBacis i Listing 56 tjekker, om alle vektorerne i en matrix er ortogonale med hinanden. Dette kan gøres ved hjælp af følgende sætning:

**Sætning 2.**
*Lad $A$ have søjlerne $v_1, v_2, \ldots, v_n$, som udgør en ortogonal basis. Per definition gælder det, at det indre produkt mellem to ortogonale vektorer $v_i$ og $v_j$ er:*

$$\langle v_i, v_j \rangle = 0 \quad for \quad i \neq j \tag{6}$$

*Lad så $D$ være en kvadratisk diagonal matrix. Da gælder det, at:*

$$D = A^* A \tag{7}$$

*Bevis.* Fra definitionen af matrixprodukt 4 ved vi, at:

$$A^* A = \begin{bmatrix} | & & | \\ A^* \cdot v_1 & \cdots & A^* \cdot v_n \\ | & & | \end{bmatrix} \tag{8}$$

Fra definitionen af matrix-vektorprodukt 3 ved vi, at:

$$A^* v_i = \begin{bmatrix} - & \overline{v}_1 & - \\ & \vdots & \\ - & \overline{v}_n & - \end{bmatrix} \begin{bmatrix} v_{1i} \\ \vdots \\ v_{ni} \end{bmatrix} = \begin{bmatrix} \sum_{j=1}^{m} \overline{v}_{j1} v_{ji} \\ \sum_{j=1}^{m} \overline{v}_{j2} v_{ji} \\ \vdots \\ \sum_{j=1}^{m} \overline{v}_{jn} v_{ji} \end{bmatrix} \tag{9}$$

Hvilket ifølge definitionen af indre produkt 5 betyder

$$A^* v_i = \begin{bmatrix} \langle v_i, v_1 \rangle \\ \langle v_i, v_2 \rangle \\ \vdots \\ \langle v_i, v_n \rangle \end{bmatrix} \tag{10}$$

Hvilket betyder, at:

$$A^* A = \begin{bmatrix} \langle v_1, v_1 \rangle & \langle v_2, v_1 \rangle & \cdots & \langle v_n, v_1 \rangle \\ \langle v_1, v_2 \rangle & \langle v_2, v_2 \rangle & \cdots & \langle v_n, v_2 \rangle \\ \vdots & \vdots & \ddots & \vdots \\ \langle v_1, v_n \rangle & \langle v_2, v_n \rangle & \cdots & \langle v_n, v_n \rangle \end{bmatrix} \tag{11}$$

Derfor, hvis $m \geq n$, så gælder det, at $A^* A = D$, da hvis $m < n$, ville $v_1, v_2 \cdots v_n$ ikke være lineært uafhængige vektorer. □

Derudover skal vi også undersøge, om spannet før og efter Gram-Schmidt processen forbliver det samme. Dette gøres ved brug af sætning 3.

**Sætning 3.**
*Lad*

$$W = [\mathbf{w}_1, \mathbf{w}_2, \ldots, \mathbf{w}_n], \quad \mathbf{w}_k \in \mathbb{F}^m \text{ for } k = 1, \ldots, n$$

*hvor søjlerne af $W$ er de resulterende vektorer efter at have udført Gram-Schmidt processen uden normalisering på*

$$V = [\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n], \quad \mathbf{v}_k \in \mathbb{F}^m \text{ for } k = 1, \ldots, n.$$

*Da gælder det, at span$(V)$ = span$(W)$. Dette kan undersøges ved at verificere, at*

$$\Phi([V\ W]) = [\Phi(V)\ \Phi(W)]$$

*og*

$$\Theta([W\ V]) = [\Theta(W)\ \Theta(V)]$$

*hvor $\Phi : \mathbb{F}^{m \times s} \to \mathbb{F}^{m \times s}$ er den afbildning, som bringer $V$ i række-echelon form, dvs.*

$$\Phi(V) = \begin{bmatrix} 1 & \varphi_{1,2} & \varphi_{1,3} & \cdots \\ 0 & 1 & \varphi_{2,3} & \cdots \\ 0 & 0 & 1 & \cdots \\ 0 & 0 & 0 & \ddots \\ \vdots & \vdots & \vdots & \\ 0 & 0 & 0 & \end{bmatrix} \in \mathbb{F}^{m \times n}$$

*samt* $\Theta : \mathbb{F}^{m \times s} \to \mathbb{F}^{m \times s}$ *er den afbildning, som bringer* $W$ *i række-echelon form.*

*Dette medfører, at* $\Phi(W)$ *og* $\Theta(V)$ *er øvre trekantsmatricer med 1-taller i diagonalerne.*

*Bevis.* Hvis $W \subseteq V$, gælder det, at $\text{span}(W) \subseteq \text{span}(V)$. Hvis derudover $V \subseteq W$, så medfører det, at $\text{span}(V) = \text{span}(W)$[32].

$W$ tilhører $\text{span}(V)$, hvis $\mathbf{w}_k$ for $k = 1, \ldots, n$ er en linearkombination af vektorerne i $V$. Da $\mathbf{w}_k$ er resultatet af Gram-Schmidt processen uden normalisering, gælder det, at

$$\mathbf{w}_1 = \mathbf{v}_1$$

og

$$\mathbf{w}_k = \mathbf{v}_k - \sum_{j=1}^{k-1} \text{proj}_{\mathbf{w}_j}(\mathbf{v}_k) = \mathbf{v}_k - \sum_{j=1}^{k-1} \frac{\langle \mathbf{v}_k, \mathbf{w}_j \rangle}{\langle \mathbf{w}_j, \mathbf{w}_j \rangle} \mathbf{w}_j \quad \text{for } k = 2, \ldots, n.$$

Ved at anvende Gram-Schmidt processen på vektorerne i $V$ fås:

$$\begin{cases} \mathbf{w}_1 = \mathbf{v}_1, \\ \mathbf{w}_2 = \mathbf{v}_2 - c_{2,1}\mathbf{w}_1 = \mathbf{v}_2 - c_{2,1}\mathbf{v}_1, \\ \mathbf{w}_3 = \mathbf{v}_3 - (c_{3,2}\mathbf{w}_2 + c_{3,1}\mathbf{w}_1) = \mathbf{v}_3 - (c_{3,2}\mathbf{v}_2 + (c_{3,1} - c_{3,2}c_{2,1})\mathbf{v}_1), \\ \quad \vdots \end{cases}$$

hvilket betyder, at $W$ er en linearkombination af $V$ på formen:

$$\mathbf{w}_k = \mathbf{v}_k - \sum_{j=1}^{k-1} \alpha_{k,j}\mathbf{v}_j, \quad k = 1, \ldots, n.$$

$\Phi$ udfører en serie af rækkeoperationer og er derfor lineær i rækkerne. Dermed gælder det, at:

$$\Phi(\mathbf{w}_k) = \Phi(\mathbf{v}_k - \sum_{j=1}^{k-1} \alpha_{k,j}\mathbf{v}_j) = \Phi(\mathbf{v}_k) - \sum_{j=1}^{k-1} \alpha_{k,j}\Phi(\mathbf{v}_j), \quad k = 1, \ldots, n.$$

[32] [5] *A Second Course in Linear Algebra*, Theorem 1.4.10 s. 25.

hvor

$$\Phi(\mathbf{v}_k) = \begin{bmatrix} \varphi_{k,1} \\ \vdots \\ \varphi_{k,k-1} \\ 1 \\ 0 \\ \vdots \end{bmatrix}, \quad \text{for} \quad k = 2 \dots n \quad \text{og} \quad \Phi(\mathbf{v}_1) = \begin{bmatrix} 1 \\ 0 \\ \vdots \end{bmatrix}$$

Derudover, da $\sum_{j=1}^{k-1} \alpha_{k,j} \Phi(\mathbf{v}_j)$ er summen af skalerede søjler 1 til $k-1$ i $\Phi(V)$, gælder det, at

$$\sum_{j=1}^{k-1} \alpha_{k,j} \Phi(\mathbf{v}_j) = \begin{bmatrix} \beta_{k,1} \\ \vdots \\ \beta_{k,k-1} \\ 0 \\ \vdots \end{bmatrix}$$

$$\Phi\left(\begin{bmatrix} V & W \end{bmatrix}\right) = \begin{bmatrix} A \\ B \end{bmatrix}$$

hvor $B \in \mathbb{F}^{(m-n)\times 2n}$ er en nulmatrix og

$$A = \begin{bmatrix} 1 & \varphi_{1,2} & \varphi_{1,3} & \cdots & \varphi_{1,n} & 1 & \psi_{1,2} & \psi_{1,3} & \cdots & \psi_{1,n} \\ 0 & 1 & \varphi_{2,3} & \cdots & \varphi_{2,n} & 0 & 1 & \psi_{2,3} & \cdots & \psi_{2,n} \\ 0 & 0 & 1 & \cdots & \varphi_{3,n} & 0 & 0 & 1 & \cdots & \psi_{3,n} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 & 0 & 0 & 0 & \cdots & 1 \end{bmatrix}$$

Da $A_{n+1..2n}$ danner en øvre trekantsmatrix, betyder det, at $W \subseteq V$ og dermed, at $\text{span}(W) \subseteq \text{span}(V)$. Men ydermere, eftersom $A_{n+1..2n}$ også har 1'ere i diagonalen, betyder det, at vi kan bruge den samme serie af rækkeoperationer, som bruges til at bringe $V$ i række-echelon form, til at bringe $W$ i række-echelon form. Derfor gælder det, at

$$\Phi([W\ V]) = [\Phi(W)\ \Phi(V)]$$

Hvilket betyder, at $V \subseteq W$ og dermed gælder det, at $\text{span}(V) = \text{span}(W)$. □

I beviset til sætning 3 viste vi at man vil kunne bruge de samme række operationer til at bringe $W$ og $V$ i række-echelon form. Dog da vi ikke gemmer rækkeoperation i vores implamentation vil vi nøjes med at lave en funktion hasSameSpanGS i Listing 57, som undersøger om at $\Phi(W)$ og $\Theta(V)$ er øvre trekantsmatricer med 1-taller i diagonalerne.

```
1 // hasSameSpan : Matrix -> Matrix -> bool
2 let hasSameSpanGS mV mW =
3     let (D(r, c)) = dimMatrix mV
4     if (D(r, c)) <> dimMatrix mW then failwith "Matrices must have the same
      dimension"
5     else
```

```
6
7       let Phi m1 m2 =
8           let s1, s2 =
9               extendMatrixWithMatrix m1 m2
10              |> rowEchelonForm
11              |> splitMatrix C (c-1)
12          isUpperTriangularDiagonal1 s1 && isUpperTriangularDiagonal1 s2
13
14      Phi mV mW && Phi mW mV
```

Listing 57: Funktion til at undersøge om søjlerne i to matricer har samme span

Dermed kan vi opskrive en en PBT af Gram-Schmidt processen i Listing 58.

```
let gramSchmidtIsOrthogonal (m:independentBacisMatrix) =
    let res =
        try
            let um = orthogonalBacis m
            if  isOrthogonalBacis um && hasSameSpanGS m um then
                1
            else
                0
        with
            | :? System.DivideByZeroException as _ -> 2
            | :? System.OverflowException as _ -> 3

    (res = 1 || res = 2 || res = 3)
    |> Prop.classify (res = 1) "PropertyHolds"
    |> Prop.classify (res = 2) "DivideByZeroExceptions"
    |> Prop.classify (res = 3) "OverflowException"
```

Listing 58: PBT af Gram-Schmidt processen

```
> Arb.register<independentBacisMatrixGen>()
- let _ = Check.Quick gramSchmidtIsOrthogonal;;
Ok, passed 100 tests.
69% PropertyHolds.
31% OverflowException.
```

Listing 59: Output fra PBT af Gram-Schmidt processen

Outputtet fra PBT af Gram-Schmidt processen kan ses i Listing 59. Som sædvanligt indikerer testen kun korrekthed, men medfører ikke garanti for det.

Som nævnt i sektion 4.3 var der en slåfejl i Algorithm 9 for echelonform i noterne til „01001 Mathematics 1a“[33]. I forbindelse med at finde frem til denne fejl blev der lavet en PBT af echelonform af generatoren for lineært uafhængige vektorsæt. Egenskaben i 8 skal sikre, at både generatorens implementering er korrekt, og at funktionen til at finde echelonformen er det.

**Egenskab 8.**
*Lad A være en matrix, hvor søjlerne er lineært uafhængige. Da gælder det, at A har fuld rank.*

Vi kan dermed opskrive en PBT af generatoren i Listing 60.

[33] [12] *Mathematics 1a.*

```
// independentBacisMatrixHasFullRank : independentBacisMatrix -> bool
let independentBacisMatrixHasFullRank (m:independentBacisMatrix) =
    let res =
        try
            if hasFullRank m then 1 else 0
        with
            | :? System.OverflowException -> 2
    (res = 1 || res = 2)
    |> Prop.classify (res = 1) "PropertyHolds"
    |> Prop.classify (res = 2) "OverflowException"
```

Listing 60: PBT af generatoren for lineært uafhængige vektorsæt

Outputtet fra PBT af generatoren for lineært uafhængige vektorsæt kan ses i Listing 61. Testen viser, at egenskaben holder.

```
> Arb.register<independentBacisMatrixGen>()
- let _ = Check.Quick independentBacisMatrixHasFullRank;;
Ok, passed 100 tests.
87% PropertyHolds.
13% OverflowException.
```

Listing 61: Output fra PBT af generatoren for lineært uafhængige vektorsæt

# 6 Diskussion

Vi begyndte rapporten med at informere læseren om, at Python er blevet indført som et hjælpemiddel i matematikkurserne på DTU. En af fordelene ved Python er, at det er et meget mere udbredt programmeringssprog, hvilket gør det nemmere at finde hjælp og vejledninger til opbygning af et program eller løsning af en opgave. Under 1% af udviklere i 2023 anvender F#, hvilket er markant mindre end de næsten 50%, som bruger Python[34]. Derfor har det været en udfordring i udviklingen af dette program at finde vejledning på internettet til de problemstillinger, der er opstået undervejs.

Derimod, eftersom F# er et stærkt typet sprog, har det været markant nemmere at finde fejl i programmet inden det bliver kørt, hvilket er en af de store fordele ved F# frem for Python. Python er dynamisk typet, hvilket tillader at kalde funktioner med argumenter uden at specificere, hvilken type argumentet skal have. Programmer virker derfor kun, hvis argumentet har de metoder, som funktionen forventer. Det medfører, at man ofte skal tjekke, om en metode er til stede på et objekt, hvilket vi ikke behøver i F#. Derudover kan objekter få tilføjet metoder under kørslen, hvilket kan have sine fordele, men som udvikler medfører det flere problemstillinger, blandt andet at når et objekt ikke har den forventede metode, vil programmet først fejle under kørslen.

Mangel på typer sammenlignet med F# betyder også, at det tager længere tid og kræver flere kommentarer at forstå, hvad et program gør. Denne udfordring har man ikke med F#, så længe funktionsnavnet er sigende for, hvad funktionen gør. Kombineret med at kunne se typen for funktionen, behøver man ofte ikke at læse selve koden for at forstå, hvad funktionen gør. Dette medfører, at man som udvikler kan være mere effektiv.

Vedrørende syntaksen af F# sammenlignet med Python, som er kendt for at have en mere læsbar syntaks, hvilket er en af årsagerne til, at det er et mere begyndervenligt sprog. Især da F#'s syntaks er forholdsvis anderledes end klassiske imperative programmeringssprog. Dog ligger syntaksen for mange af funktionerne i dette program meget tæt op ad matematiske notationer, især grundet muligheden for nemt at kunne overskrive operatorer på egne typer. Netop det, at mange af funktionerne ligner de matematiske funktioner gør, at implementeringen af dem burde medføre en bedre forståelse af, hvorfor mange af de matematiske metoder, den studerende lærer at udføre i hånden, er korrekte. Dette inkluderer især matrixoperationer, og hvordan matrix-matrix produkt er bygget på matrix-vektor produkt, hvor vi har gennemgået en række funktioner, som ved implementering gennem funktionsprogrammering gavner forståelsen af at udføre matrix-matrix produkt i hånden. Sammenhængen mellem matematiske domæner og typerne i F# gør, at man også kan forstå domæner fra et andet synspunkt, hvilket kan være med til at give en bedre forståelse herom.

Når det kommer til selve programmet, er der også både nogle udvidelser og forbedringer, der kunne laves. Først og fremmest, matricemodulet kunne være bygget med følgende typer som vist i Listing 62.

```
type Vector = V of list<Expr<Number>> * Order
type Matrix = M of list<Vector> * Order
```

Listing 62: Eksempel på alternative typer for matrixmodulet

[34] [23] *Most used programming languages among developers worldwide as of 2023*.

Dette ville ikke have betydet store ændringer i funktionerne for modulet uden at ændre på funktionaliteten. Desuden ville det have gjort det nemmere at implementere løsninger til et lineært ligningssystem.

Modulet for komplekse tal kunne også have haft en polymorfisk type som vist i Listing 63.

```
type Complex<'a, 'b> = C of 'a * 'b
```

Listing 63: Eksempel på alternative typer for komplekse tal modulet

Som ville medføre, at Number-typen ville blive:

```
type Number =
  | Int of int
  | Rational of Rational
  | Complex of Complex<Number, Number>
```

Listing 64: Eksempel på alternative typer for Number-typen

Denne ændring ville selvfølgelig medføre, at Number ville blive en rekursivt defineret type, hvilket skulle håndteres ansvarligt.

# 7 Konklusion

Det primære fokuspunkt for rapporten er, at læseren skal opnå en bedre forståelse af matematiske koncepter gennem opbygning af et matematisk program ved hjælp af funktionsprogrammering. Flere studerende oplever, at rekursive funktioner generelt er et svært koncept, og derudover mangler nogle en forståelse for programdesign, for eksempel ved at genbruge kode frem for at ligge den i metoder. Ved at introducere de studerende til rekursivitet gennem et funktionelt programmeringssprog som F# ved hjælp af matematiske koncepter, som de møder i kurserne „01001 Matematik 1a“ og „01002 Matematik 1b“, vil det gavne de studerendes forståelse af både matematikken og programmering. Derudover vil kendskabet til funktionsprogrammering forbedre de studerendes evne til at designe bedre programmer også i andre sprog som Python.

Selvom syntaksen for F# kan være sværere at lære for nogle, er gevinsten ved at lære den markant. Implementeringen af dette program, fra hvordan man udfører operationer af komplekse tal til matrixmanipulation, vil gavne forståelsen af matematikken. De studerende vil forstå, hvorfor diverse funktioner virker, hvilket ikke altid er tilfældet i Python, hvor mange funktioner kan håndteres som en sort boks.

Konklusionen er, at F# måske ikke vil kunne erstatte Python i matematikkurserne på DTU, men det vil kunne være et godt supplement til at forbedre de studerendes forståelse af matematikken og programmering.

# Litteratur


[1] *Convert Infix expression to Postfix expression.* URL: https://www.geeksforgeeks.org/%20convert-infix-expression-to-postfix-expression/ (hentet 17.02.2024).

[2] *Differentiation Rules.* URL: https://en.wikipedia.org/wiki/Differentiation_rules (hentet 10.05.2024).

[3] *Floating-point error.* URL: https://en.wikipedia.org/wiki/Floating-point_error_mitigation (hentet 30.03.2024).

[4] *fsharp.org.* URL: https://fsharp.org/ (hentet 30.03.2024).

[5] Stephan Ramon Garcia og Roger A. Horn. *A Second Course in Linear Algebra.* Cambridge University Press, 2017. ISBN: 9781107103818.

[6] *Github repository for the project.* URL: https://github.com/Larsen00/funktionsprogrammeringForIndledend (hentet 24.05.2024).

[7] *Greatest common divisor wikipedia.* URL: https://en.wikipedia.org/wiki/Greatest_common_divisor (hentet 30.03.2024).

[8] *Inverse function.* URL: https://en.wikipedia.org/wiki/Inverse_function (hentet 06.05.2024).

[9] Ole Christensen og Jakob Lemvig. *Mathematics 1b - Functions of several variables.* URL: https://01002.compute.dtu.dk/_assets/notesvol2.pdf (hentet 09.02.2024).

[10] *lambda calculus.* URL: https://en.wikipedia.org/wiki/Lambda_calculus (hentet 20.05.2024).

[11] *Maple.* URL: https://www.maplesoft.com/products/Maple/ (hentet 19.05.2024).

[12] *Mathematics 1a.* URL: https://01001.compute.dtu.dk/_assets/enotesvol1.pdf (hentet 09.02.2024).

[13] *Operator Precedence and Associativity in C.* URL: https://www.geeksforgeeks.org/operator-precedence-and-associativity-in-c/ (hentet 17.02.2024).

[14] Lawrence C. Paulson. *ML for the Working Programmer.* Cambridge University Press, 1992. ISBN: 0521422256.

[15] *Polsk notation wikipedia.* URL: https://en.wikipedia.org/wiki/Polish_notation.

[16] *Python - List Comprehension.* URL: https://www.w3schools.com/python/python_lists_comprehension.asp (hentet 31.03.2024).

[17] *Python: The most popular language.* URL: https://datascientest.com/en/python-the-most-popular-language#:~:text=Python%20is%20also%20the%20most%20widely%20used%20language%20for%20Data,libraries%20and%20other%20numerical%20algorithms. (hentet 19.05.2024).

[18] *Rank (linear algebra).* URL: https://en.wikipedia.org/wiki/Rank_(linear_algebra)#:~:text=The%20column%20rank%20of%20A,row%20rank%20are%20always%20equal. (hentet 18.05.2024).

[19] *Rational Number wikipedia.* URL: https://en.wikipedia.org/wiki/Rational_number (hentet 30.03.2024).

[20] *Referential Transparency.* URL: https://en.wikipedia.org/wiki/Referential_transparency (hentet 22.05.2024).

[21] *Row- and column-major order.* URL: https://en.wikipedia.org/wiki/Row-_and_column-major_order (hentet 31.03.2024).

[22] *Ruby.* URL: https://www.ruby-lang.org/en/ (hentet 19.05.2024).

[23] Statista. *Most used programming languages among developers worldwide as of 2023*. URL: https://www.statista.com/statistics/793628/worldwide-developer-survey-most-used-languages/ (hentet 30.03.2024).
[24] *Test-Driven Development*. URL: https://en.wikipedia.org/wiki/Test-driven_development (hentet 31.03.2024).
[25] *Tree Traversal Techniques – Data Structure and Algorithm Tutorials*. URL: https://www.geeksforgeeks.org/tree-traversals-inorder-preorder-and-postorder/ (hentet 30.03.2024).
[26] *Typed lambda calculus*. URL: https://en.wikipedia.org/wiki/Typed_lambda_calculus (hentet 20.05.2024).
[27] *Unit Testing*. URL: https://en.wikipedia.org/wiki/Unit_testing (hentet 19.05.2024).

# 8 Appendiks

Hele projektet kan findes på Github, på følgende reference [6]. Udvalgte filer som omtaltes fra projektet er vedlagt i følgende sektioner.

## 8.1 complex.fsi

```
module complex
open rational

type complex = C of rational * rational
with
    static member ( + )  : complex * complex -> complex
    static member ( - )  : complex * complex -> complex
    static member ( * )  : rational * complex -> complex
    static member ( * )  : complex * rational -> complex
    static member ( * )  : complex * complex -> complex
    static member ( / )  : complex * rational -> complex
    static member ( / )  : complex * complex -> complex
    static member ( ~- ) : complex -> complex

val newComplex : rational * rational -> complex
val isGreater : complex * complex -> bool
val realPart : complex -> rational
val isReal : complex -> bool
val isZero : complex -> bool
val toString : complex -> string
val isNegative : complex -> bool
val absComplex : complex -> complex
val conjugate : complex -> complex
```

Listing 65: complex.fsi

## 8.2 TreeGenerator.fs

```
module TreeGenerator
open Number
open Expression

type Associative = | Left | Right
type Precedence = int
type Operator = char * Precedence * Associative
type Token =
    | Operand of char
    | Operator of Operator
    | Konstant of int
type OperatorList = Operator list

// Converts a infix string to a list of tokens
let rec infixToTokenList s =
    mapToToken (Seq.toList s) true false

// maps a token to its corresponding token type
and mapToToken l allowUnary allowOperator=
    match l with
    | [] -> []
```

```
22      | x::tail when x = ' ' -> mapToToken tail allowUnary allowOperator
23      | x::tail when allowOperator && x = '/' || x = '*' -> Operator (x, 2, Left
        )::mapToToken tail false false
24      | x::tail when allowOperator && (x = '+' || (x = '-' && not allowUnary))
        -> Operator (x, 1, Left)::mapToToken tail false false
25      | x::tail when  (x = '-' && allowUnary) -> Operator ('~', 2, Right)::
        mapToToken tail true false
26      | x::tail when x = '(' -> Operator (x, -1, Left)::mapToToken tail true
        true
27      | x::tail when x = ')' -> Operator (x, -1, Left)::mapToToken tail false
        true
28      | x::_ when System.Char.IsDigit(x) ->
29          let (k, tail) = foundInt l ""
30          Konstant (int k):: mapToToken tail false true
31      | x::tail when System.Char.IsLetter x -> Operand x::mapToToken tail false
        true
32      | x::_ -> failwith ("Invalid syntax at: " + string x)
33
34  // Allowing more than one digit in a number
35  and foundInt l s =
36      match l with
37      | x::tail when System.Char.IsDigit(x) -> foundInt tail (s + string x)
38      | _ -> (s, l)
39
40
41  // pops the last operator from the stack and adds it to the prefix list
42  let popprefixStack op prefix =
43      match op, prefix with
44      | x, e1::e2::tail when x = '+' -> Add(e2, e1)::tail
45      | x, e1::e2::tail when x = '-' -> Sub(e2, e1)::tail
46      | x, e1::e2::tail when x = '*' -> Mul(e2, e1)::tail
47      | x, e1::e2::tail when x = '/' -> Div(e2, e1)::tail
48      | x, e::tail when x = '~' -> Neg(e)::tail
49      | x, _ when x = '(' -> prefix
50      |_,_ -> failwith "match not found"
51
52  // Runs the algorithm to generate the expression tree
53  let rec generateExpresion c (stack:OperatorList) prefix =
54      match c, stack with
55      | [], [] -> prefix
56      | [], (s,_,_)::stack_tail  -> generateExpresion c stack_tail (
        popprefixStack s prefix)
57      | Operand x :: tail, _ -> generateExpresion tail stack (X x::prefix)
58      | Konstant x :: tail, _ -> generateExpresion tail stack (N (Int x)::prefix
        )
59      | Operator (x, prec, lr)::tail, _
60          when x = '('
61          -> generateExpresion tail ((x, prec, lr)::stack) prefix
62      | Operator (x, _, _)::tail, _
63          when x = ')'
64          ->  match stack with
65              | [] -> generateExpresion tail stack prefix
66              | (s,_,_)::stack_tail
67                  when s = '('
68                  -> generateExpresion tail stack_tail prefix
69              | (s,_,_)::stack_tail
70                  -> generateExpresion c stack_tail (popprefixStack s prefix)
71      | Operator e::tail, [] -> generateExpresion tail (e::stack) prefix
72      | Operator e::tail, s::stack_tail
```

```
        ->  match e, s with
            | (x, precX, lr), (y, precY, _)
                when precX < precY || (precX = precY && lr = Left)
                -> generateExpresion c stack_tail (popprefixStack y prefix)
            | _, _ -> generateExpresion tail (e::stack) prefix

// Converts a infix string to a expression tree
let tree s =
    match generateExpresion (infixToTokenList s) [] [] with
    | [] -> failwith "Tree is empty"
    |tree::_ -> tree

// Adds parenthesis to a string if a boolean is true
let parenthesis b f =
    if b then "(" + f + ")" else f

// Converts a expression tree to a infix string
let rec etf e p =
    match e with
    | N a when not <| isInt a -> parenthesis p <| toString a
    | N a -> toString a
    | X a -> string a
    | Neg a ->  parenthesis p <| "-" + etf a (not p)
    | Add(a, b) -> parenthesis p <| etf a false + "+" + etf b false
    | Sub(a, b) -> parenthesis p <| etf a false + "-" + etf b true
    | Mul(a, b) -> parenthesis p <| etf a true + "*" + etf b true
    | Div(a, b) -> parenthesis p <| etf a true + "/" + etf b true

// Converts a expression tree to a infix string
let infixExpression e = etf e false
```

Listing 66: TreeGenerator.fs

## 8.3 CommutativeAddSub.fs

```
module CommutativeAddSub
open Expression
open Number


// rank when sorting a commutative expression
let expressionSortRank e1 =
    match e1 with
    | N _ -> 1
    | Neg (N _) -> 2
    | X _ -> failwith "Variables should not be in the list" // all veriables
    are multiplied with 1
    | Add _ -> failwith "Addition should not be in the list" // all addition
    should be reduced
    | Sub _ -> failwith "Subtraction should not be in the list" // all
    subtraction should be reduced
    | Mul _ -> 6
    | Div _ -> 7
    | Neg _ -> 8

// Flattens a expression tree with respect to addition and subtraction
let rec flatTree e =
    match e with
```

```
21      | Add (a, b) -> flatTree a @ flatTree b
22      | Sub (a, b) -> flatTree a @ flatTree (Neg b)
23      | Neg (Add(a, b)) -> flatTree (Neg a) @ flatTree (Neg b)
24      | Neg (Sub(a, b)) -> flatTree (Neg a) @ flatTree b
25      | X a -> [Mul(N one, X a)]
26      | Div (a, b) -> [a / b]
27      | Neg (Neg a) -> flatTree a
28      | N _ | Div _ | Mul _ | Neg _-> [e]
29
30
31
32  let rec sort l = List.sortBy (fun e -> expressionSortRank e) l
33
34  // Reduces a sorted commutative list for addition
35  let rec reduceNumbers l =
36      // printfn "rn %A" l
37      match l with
38      | [] -> []
39      | N a :: N b :: tail -> reduceNumbers (N a + N b :: tail)
40      | N a :: Neg (N b) :: tail  -> reduceNumbers (N a - N b :: tail)
41      | Neg (N a) :: Neg (N b) :: tail -> reduceNumbers (Neg (N a + N b) :: tail
        )
42      | Neg (N a) :: tail -> Neg (N a) :: tail
43      | N a :: tail -> N a :: tail
44      | _ -> l
45
46  // given a list of commutative expressions, and a variable, sums the
        corefficients of the variable
47  let rec sumInstancesOfVarible x l =
48      match x, l with
49      | _, [] -> ([] ,x)
50      | Mul(N n1, X x1), Mul(N n2, X x2) :: tail
51      | Mul(N n1, X x1), Mul(X x2, N n2) :: tail
52      | Mul(X x1, N n1), Mul(N n2, X x2) :: tail
53      | Mul(X x1, N n1), Mul(X x2, N n2) :: tail
54          when x1 = x2    -> sumInstancesOfVarible (Mul(N (n1 + n2), X x1)) tail
55      | _, head :: tail   ->
56                          let (l_new, x_new) = sumInstancesOfVarible x tail
57                          (head :: l_new, x_new)
58
59
60
61
62  let rec reduceVariables l =
63      match l with
64      | [] -> []
65      | Mul(N a, X b) :: tail
66      | Mul(X b, N a) :: tail
67          ->
68          let (l_new, x_new) = sumInstancesOfVarible (Mul(N a, X b)) tail
69          x_new :: reduceVariables l_new
70      | head :: tail -> head :: reduceVariables tail
71
72  let rec rebuldTree l =
73      // printfn "rt %A" l
74      match l with
75      | [] -> N zero
76      | Neg x::tail -> (rebuldTree tail) - x
77      | Mul (a, b)::tail -> (rebuldTree tail) + (a * b)
```

```
78      | Div (a, b)::tail -> (rebuldTree tail) + (a / b)
79      | x::tail -> rebuldTree tail + x
80
81  let applyCommutative e =
82      match e with
83      | Sub _ | Add _ -> flatTree e |> sort |> reduceNumbers |> reduceVariables
        |> rebuldTree
84      | _ -> e
```

Listing 67: CommutativeAddSub.fs

## 8.4 CommutativeMulDiv.fs

```
1   module CommutativeMulDiv
2   open Expression
3   open Number
4
5
6   ////////////////////////////////////////////////////////////////
7   /// Commutative RULES For multiplication and division ///////////
8   ////////////////////////////////////////////////////////////////
9
10  // rank when sorting a commutativ expression
11  let expressionSortRank e1 =
12      match e1 with
13      | Neg _ -> 1
14      | N _ -> 2
15      | X _ -> 3
16      | Add _ -> 4
17      | Sub _ -> 5
18      | Mul _ -> 6
19      | Div _ -> 7
20
21  // sorts the commutativ list
22  let rec sort l = List.sortBy (fun e -> expressionSortRank e) l
23
24
25  // determines the sign of a commutativ list
26  let rec signList l s =
27      match l with
28      | [] -> s
29      | Neg _::tail -> signList tail (-1*s)
30      | _::tail -> signList tail s
31
32
33  // flattens the tree to a commutativ list
34  let rec flatTree e =
35      match e with
36      | N _ | X _ | Add _ | Sub _ -> [e]
37      | Neg a -> Neg (N one) :: flatTree a
38      | Mul (a, b) -> flatTree a @ flatTree b
39      | Div (N a, N b) -> [N (a / b)]
40      | Div (Neg a, b) | Div (a, Neg b) -> Neg (N one) :: flatTree (Div (a, b))
41      | Div (a, N b) -> N (one / b) :: flatTree a
42      | Div (a, b) -> Div (N one, b) :: flatTree a
43
44
45  // multiplys all Div elements in a commutativ list
```

```
46 let rec mulDivElements l =
47     // division will always be in the end of a sorted list, and numarator will
        be 1
48     match l with
49     | [] -> []
50     | Div (_, b) :: Div (_, c) :: tail -> mulDivElements (Div(N one, Mul(b, c)
       ) :: tail)
51     | x::tail -> x :: mulDivElements tail
52
53 // removes a element from a list
54 let rec removeElem e l =
55     match  l with
56     | [] -> []
57     | x::tail when x = e -> tail
58     | x::tail -> x :: removeElem e tail
59
60 // reduces a commutativ list
61 let rec reduce l =
62     let sorted = divCancelling (sort l)
63     if signList sorted 1 > 0 then rebuildTree sorted else Neg (rebuildTree
       sorted)
64
65 // initiates the division cancelling
66 and divCancelling l =
67     // printfn "DivCancelling: %A" l
68     match List.rev l with
69     | [] -> l
70     | Div(_, b)::tail ->
71                           let (numerator, denominator) = cancelEquality  (List.
       rev tail) (sort (flatTree b))
72                           sort (flatTree (reduce numerator / reduce denominator)
       )
73     | _ -> l
74
75 // cancels out equal elements in the numerator and denominator
76 and cancelEquality nu de =
77     // printfn "cancelEquality: %A / %A" nu de
78     match nu with
79     | [] -> ([], de)
80     | n::ntail when List.contains n de ->  cancelEquality ntail (removeElem n
       de)
81     | n::ntail -> let (numerator, denominator) = cancelEquality ntail de
82                   (n::numerator, denominator)
83
84
85
86 // rebuilds a commutativ list to a tree
87 and rebuildTree l =
88     match l with
89     | [] -> N one
90     | Neg (N a)::tail when Number.isOne a -> rebuildTree tail
91     | Neg a::tail -> rebuildTree (a::tail)
92     | N a::N b::tail -> rebuildTree (N (a * b) :: tail)
93     | N a :: Add(b, c) :: tail -> rebuildTree (Add(reduce (flatTree (Mul(N a,
       b))), reduce (flatTree (Mul(N a, c)))) :: tail)
94     | N a :: Sub(b, c) :: tail -> rebuildTree (Sub(reduce (flatTree (Mul(N a,
       b))), reduce (flatTree (Mul(N a, c)))) :: tail)
95     | a::tail -> a * rebuildTree tail
96
```

```
97
98  // applies the commutativ rules to a tree
99  let applyCommutative e:Expr<Number> =
100     match e with
101     | Mul _ | Div _ -> reduce (flatTree e)
102     | _ -> e
```

Listing 68: CommutativeMulDiv.fs

## 8.5 Matrix.fs

```
1   module Matrix
2   open Number
3   open Expression
4   open SymbolicManipolation
5
6   // row or coloumn major ordere: default of a vector and matric is Column major
        order.
7   type Order = | R | C
8
9   // Rows x Cols
10  type Dimension  = D of int * int
11
12  // a "Normal" is C major and a Transposed one is R major
13  type Vector = V of list<Number> * Order
14  type Matrix = M of list<Vector> * Order
15
16  // Construct a vector
17  let vector nl = V (nl, C)
18
19  // The dimension of a vector
20  let dimVector (V(v, o)) =
21      let len = List.length v
22      match o with
23      | R -> D (1, len)
24      | C -> D (len, 1)
25
26  // Makes sure that a matrix has the same major order as its vectors
27  let matrixValidMajor (M(m, o)) =
28      match List.head m, o with
29      | V (_, R), R -> true
30      | V (_, C), C -> true
31      | _, _ -> failwith "Matrix's vector's dont have same major order"
32
33  // The list length of a vector
34  let vectorLength (V(v, _)) = List.length v
35
36  // The list length of a matrix's vectors
37  let matrixVectorLength (M(m, _)) =
38      match m with
39      | [] -> 0
40      | x::_ -> vectorLength x
41
42  // The dimension of a matrix
43  let dimMatrix (M(vl, o)) =
44      if vl = [] then D (0, 0)
45      else
46      let _ = matrixValidMajor (M(vl, o))
```

```
47      let d1 = List.length vl
48      let d2 = matrixVectorLength (M(vl, o))
49      match o with
50      | R -> D (d1, d2)
51      | C -> D (d2, d1)
52
53  // Multiplication of a scalar and a vector
54  let scalarVector (n:Number) (V (nl, o)) =
55      V ((List.map (fun x -> x * n) nl), o)
56
57  // Multiplication of a scalar and a matrix
58  let scalarMatrix (M (vl, o)) n =
59      M ((List.map (fun x -> scalarVector n x) vl), o)
60
61  // Addision two vectors
62  let addVector x y =
63      let (V (v1, o1)) = x
64      let (V (v2, o2)) = y
65      if o1 <> o2
66      then failwith "Vectors must have the same major order"
67      elif dimVector x <> dimVector y
68      then failwith "Vectors must have the same dimension"
69      else
70      V ((List.map2 (+) v1 v2), o1)
71
72  // Subtraction of two vectors
73  let subVector x y =
74      scalarVector (-one) y |> addVector x
75
76
77  // Construct a vector of n with length len
78  let vectorOf n len = V ((List.init len (fun _ -> n)), C)
79
80  // Construct a matrix of n with dimension d
81  let matrixOf n (D (r, c)) = M ((List.init c (fun _ -> vectorOf n r)), C)
82
83  // Transposes a vector
84  let transposeVector (V(v, o)) =
85      match o with
86      | R -> V(v, C)
87      | C -> V(v, R)
88
89  // adds a dimension to a vector
90  let extendVector (V(v, o)) n =
91      V(v @ [n], o)
92
93  // Extend a matrix with a number list
94  let extendMatrix (M(m, o)) nl =
95      if m <> [] && nl <> [] && List.length nl <> matrixVectorLength (M(m, o))
      then failwith "The list must have the same length as the matrix's vectors"
96      elif nl = [] then M(m, o)
97      else
98      M (m @ [(V (nl, o))], o)
99
100 // extend a matrix with a matrix
101 let extendMatrixWithMatrix (M(m1, o1)) (M(m2, o2)) =
102     if o1 <> o2 then failwith "Matrices must have the same major order"
103     elif matrixVectorLength (M(m1, o1)) <> matrixVectorLength (M(m2, o2)) then
       failwith "Matrices must have the same dimension"
```

```
104     else M (m1 @ m2, o1)
105
106
107 // Alternates the Major order
108 let alternateOrder o =
109     match o with
110     | R -> C
111     | C -> R
112
113 // Alternates the Major order of a matrix
114 let alternateOrderMatrix (M(m, o)) =
115     M (m, alternateOrder o)
116
117 // The i'th number of a vector
118 let getVectorIthNumber i (V(v, _)) = v.[i]
119
120 // The i'th vector of a matrix
121 let getMatrixIthVector i (M(m, _)) = m.[i]
122
123 // replaces the i'th vector of a matrix
124 let replaceMatrixIthVector i (M(m, o)) v =
125     M (m.[0..i-1] @ [v] @ m.[i+1..], o)
126
127 // pops a vector
128 let seperateFistNumberFromVector (V(v, o)) =
129     (v.Head, V(v.Tail, o))
130
131 // first element of a vector
132 let headVector (V(v, _)) = List.head v
133
134 // Gives a vector of all the first elements of the vectors in a matrix
135 let rec firstElemetsVectors (M(m, o)) v_acc m_acc =
136     match m with
137     | [] -> (v_acc, M(m_acc, o))
138     | x::xs ->
139             let (n, tail) = seperateFistNumberFromVector x
140             firstElemetsVectors (M(xs, o)) (v_acc @ [n]) (m_acc @ [tail])
141
142 // Helper function for changeOrderMatrix
143 let rec chaingingOrderMatrix (M(m, o)) m_acc =
144     match matrixVectorLength (M(m, o)) with
145     | 0  -> m_acc
146     | _  ->
147         let (v, m_new) = firstElemetsVectors (M(m, o)) [] []
148         extendMatrix m_acc v |> chaingingOrderMatrix m_new
149
150 // Changes the order of a matrix
151 let changeOrderMatrix (M(m, o)) =
152     alternateOrderMatrix (M([], o)) |> chaingingOrderMatrix (M(m, o))
153
154 // Makes sure that two matrices have the same major order, if diffrent then
    changes the order of the second matrix
155 let giveMatrixHaveSameOrder (M(_, o1)) (M(m2, o2)) =
156     if o1 <> o2 then changeOrderMatrix (M(m2, o2)) else M(m2, o2)
157
158 // Transposes a matrix
159 let transposeMatrix (M(vl, o)) =
160     M(List.map (fun x -> transposeVector x) vl, alternateOrder o)
161
```

```
162 // Adds two matrices elemtwise
163 let addMatrix m1 m2 =
164     let m3 = giveMatrixHaveSameOrder m1 m2
165     if dimMatrix m1 <> dimMatrix m3
166     then failwith "addMatrix: Matrices must have the same dimension"
167     else
168     let (M(vl1, o)) = m1
169     let (M(vl3, _)) = m3
170     M (List.map2 addVector vl1 vl3, o)
171
172 // Construct a matrix
173 let matrix vl =
174     let rec mc vl m =
175         match vl with
176         | [] -> m
177         | (V(x, _))::xs -> mc xs (extendMatrix m x)
178     mc vl (M([], C))
179
180 // Changes the order of af matrix to x
181 let correctOrder (M(m, o)) x =
182     if o = x then M(m, o) else changeOrderMatrix (M(m, o))
183
184 // Boolean value of if a matrix has the correct order
185 let corectOrderCheck (M(_, o)) x =
186     o = x
187
188 // Sum the rows of a matrix
189 let rec sumRows m =
190     if not <| corectOrderCheck m C
191     then sumRows <| correctOrder m C
192     else
193     let zeroVector = vectorOf zero <| matrixVectorLength m
194     let (M(vl, _)) = m
195     matrix [List.fold (addVector) zeroVector vl]
196
197
198
199 // matrix vector product  A.v = b - Definition 7.10
200 let rec matrixVectorProduct m v =
201     let (D(rv, _)) = dimVector v
202     let (D(_, cm)) = dimMatrix m
203     if rv <> cm
204     then failwith "matrixVector: the number of columns of the matrix has to be
         the same as the number of entries in the vector."
205     elif not <| corectOrderCheck m C then matrixVectorProduct (correctOrder m
        C) v
206     else
207     let (M(vl, _)) = m
208     let (V(nl, _)) = v
209     M (List.map2 (fun mc n -> scalarVector n mc) vl nl, C)
210     |> sumRows
211
212 // Converts a matrix to a vector if possible
213 let matrixToVector m =
214     match dimMatrix m, m with
215     | D(r, c), M(v::_, _) when r = 1 || c = 1 -> v
216     | _, _ -> failwith "mactrixToVector: Matrix is not a vector"
217
218 // Multiplies two matrices - Definition 7.12
```

```
219 let rec matrixProduct a b =
220     let (D(_, ca)) = dimMatrix a
221     let (D(rb, _)) = dimMatrix b
222     if ca <> rb
223     then failwith "matrixProduct: matrix product A .* B is defined only if the
     number of columns of A is the same as the number of rows of B"
224     elif not <| corectOrderCheck b C then matrixProduct a (correctOrder b C)
225     else
226     let (M(vlb, _)) = b
227     let product = List.map (
228             fun bv -> matrixVectorProduct a bv |> matrixToVector ) vlb
229     M(product, C)
230
231
232 type Vector with
233     static member (+) (v1, v2) = addVector v1 v2
234     static member (-) (v1, v2) = subVector v1 v2
235     static member (*) (n, v)  = scalarVector n v
236     static member (*) (v, n)  = scalarVector n v
237     static member (~-) (v)    = scalarVector (-one) v
238
239
240 type Matrix with
241     static member (+) (m1, m2) = addMatrix m1 m2
242     static member (+) (m, n)  = dimMatrix m |> matrixOf n |> addMatrix m
243     static member (+) (n, m)  = dimMatrix m |> matrixOf n |> addMatrix m
244     static member (*) (n, m)  = scalarMatrix m n
245     static member (*) (m, n)  = scalarMatrix m n
246     static member (*) (m, v)  = matrixVectorProduct m v
247     static member (*) (m1, m2) = matrixProduct m1 m2
248     static member (/) (m, n)  = scalarMatrix m (inv n)
249
250
251 // Horisontal flip of a matrix
252 let flip m =
253     let (M(m_new, _)) = correctOrder m C
254     List.rev m_new |> matrix
255
256
257 // extracts the first vector of a matrix
258 let extractfirstVector (M(m, o)) =
259     match m with
260     | [] -> failwith "Matrix is empty"
261     | x::xs -> x, M(xs, o)
262
263 // Last vector of a matrix
264 let rec extractlastVector m =
265     let (v, mf) = flip m |> extractfirstVector
266     (v, flip mf)
267
268
269 // Multiplies two vectors element wise
270 let vectorMulElementWise (V(u, o1)) (V(v, o2)) =
271     if o1 <> o2
272     then failwith "Vectors must have the same major order"
273     elif List.length u <> List.length v
274     then failwith "Vectors must have the same dimension"
275     else
276     V (List.map2 (*) u v, o1)
```

```
277
278 // Conjugates a vector
279 let conjugateVector (V(v, o)) =
280     V (List.map conjugate v, o)
281
282 // Conjugates a matrix
283 let conjugateMatrix (M(m, o)) =
284     M (List.map conjugateVector m, o)
285
286 // Inner product of two vectors
287 let innerProduct u v =
288     let (V(w, _)) = conjugateVector v |> vectorMulElementWise u
289     List.fold (+) zero w
290
291 // Projection of a vector on another vector
292 let proj y x =
293     scalarVector (innerProduct x y / innerProduct y y) y
294
295 // Dot product of two vectors
296 let dotProduct (V(u, ou)) (V(v, ov)) =
297     if List.length u <> List.length v
298     then failwith "dotProduct: Vectors most have same length."
299     else
300
301     match ou, ov with
302     | R, C -> innerProduct (V(u, C)) (V(v, C))
303     | _ -> failwith "Missing implementation for this case."
304
305
306 // Orthogonal bacis using the Gram-Schmidt process
307 let rec orthogonalBacis m =
308     if not <| corectOrderCheck m C
309     then orthogonalBacis (correctOrder m C)
310     else
311
312     // Gram-Schmidt process but without the normalization
313     let rec Gram_Schmidt vm acc_wm =
314         match acc_wm [], vm with
315         | x, M([], _) -> x
316         | M([], _), M(v1::vrest, o) ->
317             Gram_Schmidt (M(vrest,o))
318             <| fun x -> extendMatrix (M([v1], C)) x
319         | M(w, _), M(vk::vrest, o) ->
320             let (V(wk, _)) = vk - sumProj w vk
321             Gram_Schmidt (M(vrest,o))
322             <| fun x -> extendMatrix (acc_wm wk) x
323
324     // Sum all the projections of vk on w1 to wk-1
325     and sumProj w vk =
326         List.map (fun x -> proj x vk) w
327         |> matrix
328         |> sumRows
329         |> matrixToVector
330
331     Gram_Schmidt m (fun _ -> M([], C))
332
333
334
335 // Checks if a vector is a zero vector
```

```
336 let isZeroVector (V(v, _)) =
337     List.forall (fun x -> Number.isZero x) v
338 
339 // Checks if a matrix is a zero matrix
340 let isZeroMatrix (M(m, _)) =
341     List.forall (fun x -> isZeroVector x) m
342 
343 // fist non zero element of a vector
344 let firstNonZero (V(v, _)) =
345     if isZeroVector (V(v, C)) then failwith "firstNonZero: Vector does not
    have a non zero element"
346     else
347     List.find (fun x -> not (Number.isZero x)) v
348 
349 // Index of the first non zero element of a vector
350 let firstNonZeroIndex (V(v, _)) =
351     if isZeroVector (V(v, C)) then -1
352     else
353     List.findIndex (fun x -> not (Number.isZero x)) v
354 
355 // string a vector
356 let stringVector (V(v, o)) =
357     let space  = if o = C then "\n" else " "
358     List.map (fun x -> toString x) v |> String.concat space
359 
360 // string a matrix
361 let rec stringMatrix m =
362     if not <| corectOrderCheck m R then stringMatrix (correctOrder m R)
363     else
364     let (M(vl, _)) = m
365     List.map (fun x -> stringVector x) vl |> String.concat "\n"
366 
367 // bool on index
368 let rec vectorElementIs (V(nl, o)) i b =
369     match nl with
370     | [] -> false
371     | x::_ when i = 0 -> b x
372     | _::xs -> vectorElementIs (V(xs, o)) (i - 1) b
373 
374 // Index of the first non zero element of a matrix
375 let rec firstNonZeroIndexMatrix (M(m, o)) idx (r, c) =
376     if m = [] then (r, c) else
377     let fnxi = List.head m |> firstNonZeroIndex
378     match m with
379     | [] -> (r, c)
380     | _::vt when fnxi >= 0 && (fnxi < c || c = -1) -> firstNonZeroIndexMatrix
    (M(vt, o)) (idx + 1) (idx, fnxi)
381     | _::vt -> firstNonZeroIndexMatrix (M(vt, o)) (idx + 1) (r, c)
382 
383 
384 // Switches two vectors in a matrix
385 let swapFirstWith (M(m, o)) i =
386     if i = 0
387     then M(m, o)
388     else
389     let rec swapper h m i idx acc_m =
390         match m with
391         | [] -> failwith "swapFirstWith: Index out of range"
392         | x::xs when idx = i -> x :: acc_m @ (h::xs)
```

```
393         | x::xs -> swapper h xs i (idx + 1) (acc_m @ [x])
394
395     match m with
396     | [] -> failwith "swapFirstWith: Matrix is empty"
397     | h::tail -> M(swapper h tail i 1 [], o)
398
399 // Muliplies the i'th vector with a scalar
400 let scalarIthVector c i (M(m, o)) =
401     let rec sIV c m i idx acc_m =
402         match m with
403         | [] -> failwith "scalarIthVector: Index out of range"
404         | x::xs when idx = i -> acc_m @ c * x :: xs
405         | x::xs -> sIV c xs i (idx + 1) (acc_m @ [x])
406
407     M(sIV c m i 0 [], o)
408
409 // Alters row j with Rj <- Rj - c * Ri
410 let rec rowOperation i j c m =
411     if i = j then failwith "rowOperation: Row i and j must be diffrent j <> i"
412     elif not <| corectOrderCheck m R then rowOperation i j c <| correctOrder m
        R
413     else
414     replaceMatrixIthVector (j-1) m <| getMatrixIthVector (j-1) m - c *
    getMatrixIthVector (i-1) m
415
416
417 // row echelon form of a matrix
418 let rec rowEchelonForm A =
419     if not <| corectOrderCheck A R then rowEchelonForm (correctOrder A R)
420     else
421     let (D(r, c)) = dimMatrix A
422     match A with
423     |M([], _) -> A
424     | _ when isZeroMatrix A -> A
425     | M(v::_, _) when r = 1 -> firstNonZero v |> inv |> scalarMatrix A
426     | M(_, o) ->
427         let (i, j) = firstNonZeroIndexMatrix A 0 (-1 ,-1)
428         let (M(B, _)) = swapFirstWith A i
429         let b = List.head B |> firstNonZero
430         let (M(B, _)) = scalarIthVector (inv b) 0 (M(B, o))
431         let R1 = List.head B
432         let R2m = List.tail B
433         let B = rowOps j 1 r R1 (M(R2m, o))
434         let (M(Cm, _)) = rowEchelonForm B
435         M(R1::Cm, o)
436
437 // Ri <- Ri - b * R1 - posible error i mat 1 notes Algorithm 1 (should be b <-
        the jth entry og the ith row if B)
438 and rowOps coloumn i nrows R1 acc_m  =
439     if i >= nrows then acc_m else
440     let Ri = getMatrixIthVector (i - 1) acc_m
441     let b = getVectorIthNumber coloumn Ri
442     rowOps coloumn (i + 1) nrows R1 <| replaceMatrixIthVector (i-1) acc_m (Ri
    - b * R1)
443
444
445 // A standard bacis vector of length n with 1 at i
446 let standardBacisVector n i =
447     let rec sbv idx =
```

```
448         match idx with
449         | _ when idx < 1 -> []
450         | 1 when i <> 1 -> [zero]
451         | x when x = i -> one :: sbv (x - 1)
452         | _ -> zero :: sbv (idx - 1)
453     vector <| sbv n
454 
455 // A standard bacis matrix of F^n
456 let standardBacis n =
457     let rec sb idx =
458         match idx with
459         | _ when idx < 1 -> []
460         | 1 -> [standardBacisVector n 1]
461         | _ -> standardBacisVector n idx :: sb (idx - 1)
462     matrix <| sb n
463 
464 // fullranked diagonal matrix
465 let fullrankedDiagonalMatrix m n =
466     if m > n then failwith "fullrankedDiagonalMatrix: number of rows must be
    less than or equal to the number of columns"
467     else
468     let rec frdm i frm =
469         match i with
470         | _ when i < 0 -> failwith "fullrankedDiagonalMatrix: i must be
    greater than or equal to 0"
471         | 0 -> frm
472         | _ ->
473             let (V(zeroV, _)) = vectorOf zero m
474             frdm (i - 1) <| extendMatrix frm zeroV
475     frdm (n - m) <| standardBacis m
476 
477 
478 // Checks if a matrix is upper triangular
479 let rec isUpperTriangular (M(vl, o)) =
480     if not <| corectOrderCheck (M(vl, o)) R then isUpperTriangular (
    correctOrder (M(vl, o)) R)
481     else
482     let (D(_, m)) = dimMatrix (M(vl, o))
483     let rec iut vl idx =
484         match vl with
485         | [] -> true
486         | x::xs when idx >= m -> isZeroVector x && iut xs (idx + 1)
487         | x::xs -> firstNonZeroIndex x = idx && iut xs (idx + 1)
488     iut vl 0
489 
490 // is upper triangular matrix with diagonal elements of 1
491 let rec isUpperTriangularDiagonal1 (M(vl, o)) =
492     if not <| corectOrderCheck (M(vl, o)) R then isUpperTriangular (
    correctOrder (M(vl, o)) R)
493     else
494     let (D(_, m)) = dimMatrix (M(vl, o))
495     let rec iut vl idx =
496         match vl with
497         | [] -> true
498         | x::xs when idx >= m -> isZeroVector x && iut xs (idx + 1)
499         | x::xs ->
500             let fnz = firstNonZeroIndex x
501             vectorElementIs x fnz isOne &&
502             fnz = idx &&
```

```
503             iut xs (idx + 1)
504     iut vl 0
505
506 // rank of a matrix
507 let rank m =
508     let (D(r, c)) = dimMatrix m
509     let maxRank = if r < c then r else c
510     let rec ra m rowi =
511         if rowi > maxRank || isZeroMatrix m then
512             rowi
513         else
514             match m with
515             | M([], _) -> rowi
516             | M(v::vl, o) ->
517                 let fnz = firstNonZeroIndex v
518                 if fnz < rowi then
519                     ra (M(vl, o)) rowi
520                 else
521                     ra (M(vl, o)) (rowi + 1)
522     ra (correctOrder (rowEchelonForm m) R) 0
523
524 // determines if a matrix has full rank
525 let hasFullRank m =
526     let (D(r, c)) = dimMatrix m
527     let maxRank = if r < c then r else c
528     rank m = maxRank
529
530 // determines if a list of numbers is all zero
531 let isZeroNumberList nl =
532     List.forall (fun x -> Number.isZero x) nl
533
534 // determines if a matrix is Identity matrix
535 let isDiagonalMatrix m =
536     let (D(r, c)) = dimMatrix m
537     if r <> c then false
538     else
539     let rec iim m =
540         let nl, (M(vl, o)) = firstElemetsVectors m [] []
541         match vl, nl with
542         | [], [] -> true
543         | v::vTail, n::nTail ->
544             isZeroVector v && isZeroNumberList nTail && not (Number.isZero n)
    && iim (M(vTail, o))
545         | _, _ -> false
546     iim m
547
548 // checks is if every vector has inner product of zero with the next vector
549 let rec isOrthogonalBacis m =
550     (transposeMatrix m |> conjugateMatrix) * m |> isDiagonalMatrix
551
552 // split a matric into two matrices
553 let rec splitMatrix o i m =
554     if not <| corectOrderCheck m o then splitMatrix o i (correctOrder m o)
555     else
556     let (M(vl, _)) = m
557
558     let rec sm vl idx acc =
559         match vl with
560         | [] -> failwith "splitMatrix: Index out of range"
```

```
561         | x::xs when idx = 0 -> matrix <| acc @ [x], matrix xs
562         | x::xs -> sm xs (idx - 1) (acc @ [x])
563     sm vl i []
564 
565 // Determines if a matrix has a zero column
566 let rec dontHaveZeroCols m =
567     if not <| corectOrderCheck m C then dontHaveZeroCols (correctOrder m C)
568     else
569     let (M(vl, _)) = m
570     List.forall (fun x -> not <| isZeroVector x) vl
571 
572 
573 // Determines if a matrix has the same span as another matrix
574 let hasSameSpanGS mV mW =
575     let (D(r, c)) = dimMatrix mV
576     if (D(r, c)) <> dimMatrix mW then failwith "Matrices must have the same
    dimension"
577     else
578 
579     let Phi m1 m2 =
580         let s1, s2 =
581             extendMatrixWithMatrix m1 m2
582             |> rowEchelonForm
583             |> splitMatrix C (c-1)
584         isUpperTriangularDiagonal1 s1 && isUpperTriangularDiagonal1 s2
585 
586     Phi mV mW && Phi mW mV
587 
588 
589 
590 
591 
592 
593 
594 ////////////////////////////////
595 /// Functions to solve Ax = b ///
596 ////////////////////////////////
597 
598 type ExprVector = list<Expr<Number>>
599 type ExprMatrix = list<ExprVector>
600 
601 
602 
603 // multiplies a expression with a vector and returns a Expr list
604 let scalarWithExpr (V(nl, _)) e=
605         List.map (fun x -> N x * e) nl
606 
607 // matrix product with expresion vector
608 let matrixProductExprList vl el =
609     let znl = List.head vl |> vectorLength |> vectorOf zero
610     let zeroExprList = scalarWithExpr znl (N zero) // zero expr vector
611     List.map2 (fun mc n -> scalarWithExpr n mc) el vl
612     |> List.fold (fun a b -> (List.map2 (+) a b)) zeroExprList
613 
614 // Creates a vector of n variables
615 let rec charVector n =
616     match n with
617     | _ when n <= 0 -> []
618     | _ -> X (char (n - 1)) :: charVector (n - 1)
```

```
619
620 // inserts an eviroment into a vector
621 let rec vectorEnv n env =
622     match n with
623     | _ when n <= 0 -> V([], C)
624     | _ ->
625         let (V(nl, _)) = vectorEnv (n - 1) env
626         Map.find (char (n - 1)) env :: nl |> vector
627
628 // solves the system of lineary equations
629 let rec solveEquations el bl cl =
630         match el, bl, cl with
631         | [], [], [] -> Map.empty
632         | e::es, b::bs, _ when isZero e && isZero b ->
633             solveEquations es bs cl
634         | e::es, b::bs, c::cs ->
635                               let env = solveEquations es bs cs
636                               let (lhs, rhs) = isolateX (insertEnv e env) b c
637                               Map.add (getVariable lhs) (getNumber rhs) env
638         | _, _, _ -> failwith "solveEquations: The number of equations and
    variables must be the same"
639
640 // Solves the equation Ax = b
641 let matrixEquation A (V(nlb, ob)) =
642     let (D(r, c)) = dimMatrix A
643     if r <> List.length nlb ||  ob = R then
644         failwith "matrixEquation: b must be a column vector or have same
    length as rows of A"
645
646     // Perform row echelon form on the total matrix
647     let ef_t = correctOrder (rowEchelonForm <| extendMatrix A nlb) C
648
649     // Extract the last vector of the matrix
650     let (V(ef_b, _), M(ef_vl, o)) = extractlastVector ef_t
651
652     // Check if the system of equations has a solution
653     if hasFullRank ef_t then
654         failwith "matrixEquation: The system of equations has no solution"
655     elif not <| hasFullRank (M(ef_vl, o)) then
656         failwith "matrixEquation: The system of equations has infinite
    solutions"
657
658     let varlist = charVector c
659     let b = scalarWithExpr (vector ef_b) (N one)
660     solveEquations (matrixProductExprList ef_vl varlist) b varlist |>
    vectorEnv c
```

Listing 69: Matrix.fs