Danmarks Tekniske Universitet

Bachelor Projekt

Funktionel Modellering af Matematiske Systemer i F#

Jonas Dahl Larsen (s205829)

14. maj 2024

Indhold

1 Introduktion

Dette projekt fokuserer på funktionel modellering af matematiske systemer ved brug af programmeringssproget F#. I en tid, hvor programmeringssprog som Python dominerer i tekniske og videnskabelige miljøer, undersøger dette projekt potentialet og fordelene ved funktionel programmering i matematiske sammenhænge. I 2023 valgte Danmarks Tekniske Universitet at anvende Python som et hjælpeværktøj i deres grundlæggende matematikkursus "01001 Matematik 1a (Polyteknisk grundlag)". Imidlertid åbner funktionel programmering op for et andet perspektiv og metoder, som kan berige og muligvis forbedre forståelsen af matematiske koncepter hos studerende.

Projektet har til formål at demonstrere, hvordan funktionel programmering, specifikt gennem F#, kan anvendes til at opbygge og manipulere matematiske udtryk og systemer. Ved at introducere læserne til grundlæggende såvel som avancerede funktioner og teknikker i F#, vil rapporten guide dem gennem opbygningen af funktionelle programmer, der kan løse matematiske problemer.

Rapporten vil først og fremmest dykke ned i konstruktionen af et specifikt modul for håndtering af symbolske matematiske udtryk og matrix manipulering, og 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 og logiske principper kan integreres direkte i softwareudvikling gennem funktionel programmering. Dette vil ikke kun fremme en bedre forståelse af teoretiske koncepter gennem praktisk anvendelse, men også demonstrere F#'s kapacitet og effektivitet i behandlingen af matematiske egenskaber.

2 Fundamentale koncepter

2.1 Introduktion til Funktions Programmering

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.

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

Vi begynder derfor med at betragte funktionen for fakultet Ligning 1. Et eksempel på en implementering i F# er givet i Listing 1, som kan sammenlignes med Python-koden i Listing 2, da Python og pseudokode er næsten det samme.

```
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#

```
1 # Fakultet i Python
2 def factorial(n):
3     if n == 0:
4         return 1
5     elif n > 0:
6         return n * factorial(n - 1)
7     else:
8         raise ValueError("Negative argument")
```

Listing 2: Eksempel på Fakultet i Python

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 inputargument n, og i linje 3 starter et match-udtryk. Her er n vores udtryk, og efter with begynder en række mønstre, som udtrykket forsøger at matche på, separeret med '|'. Resultatet af funktionen vil være den kode, der eksekveres efter '→', på den linje, hvor mønsteret er genkendt.

I F#, er det som udgangspunkt ikke nødvendigt at anvende parenteser som i andre programmeringssprog. Derfor vil de kun blive anvendt, hvor det er nødvendigt 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

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

Listing 3: Eksempel på anvendelse af pipe-operatorer i F#

2.2 Typer

I F#, i modsætning til Python, er typer tildelt ved kompileringstidspunktet, ikke under kørsel. Alle udtryk, inklusiv funktioner, har 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. Typen for funktionen beskrives som $Factorial : int \to int$. Vi kan derfor formulere følgende omkring typer[1]:

$$f : T_1 \to T_2$$
$$f(e) : T_2 \iff e : T_1$$

Hvis en funktion kaldes med et argument, der ikke matcher funktionens type, genereres en fejlmeddelelse. Derudover kan en type også bestå af en tuple af typer:

$$f : T_1 * T_2 * .. * T_n \to T_{n+1}$$
$$f(e_1, e_2, .., e_n) : T_{n+1} \iff e_1 : T_1 \land e_2 : T_2 \land .. \land e_n : T_n$$

En tuple, der kun består af to typer, kaldes et par. Givet en funktion $g : T_1 \to T_2 \to T_3$, 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.

2.3 Signatur filer og implementerings filer

En standard F# fil er lavet med .fs extension, denne fil indeholder alt den kode som er nødigt for at kunne køre programmet. En implementerings fil kan have en signatur fil med .fsi extension, denne fil indeholder en beskrivelse af de typer og funktioner i implementerings filen som er tilgængelige for andre filer. En signatur fil kan derfor bruges som et blueprint for andre der ønsker at anvende eller replicere implementerings filen. I andre programmerings sprog vil man anse funktionerne i signatur filen som værende "public"og de funktioner der ikke er i signatur filen, men er i implementerings filen som værende "private".

2.4 Overloading 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.

[1] [6] *Functional Programming Using F#*, s. 14.

2.5 Property Based Testing

Property Based Test (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.

På DTU lærer de matematiske studerende først om logik, hvor det introduceres, 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 de indledende matematiske kurser på DTU lærer man at anvende sandhedstabeller, som demonstrerer gyldigheden af en formel. Eksempelvis vises hvordan 2 er gyldig.

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

Vi kan også bruge PBT til at undersøge, om (2) holder, ved at definere egenskaben som en funktion af P, Q og R, som vist i Listing 3 (4).

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

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

Listing 4: PBT af ligning 2. Begge sider er omgivet af parenteser da = har en højere præcedens end &&

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

Listing 5: Output ved PBT af (2)

Check.Quick er en del af "FsCheck"biblioteket, den tager en funktion som argument, og generere en række tilfældige input til funktionen på baggrund af funktionens type. Hvis funktionen returnere "true"for alle input, vil testen lykkedes. Hvis funktionen returnere "false"for et input, vil testen fejle og give et eksempel på et input der fejlede. I Listing 4 er der anvendt "Check.Quick"til at teste om (2) er gyldig. Funktionen "Check.Quick"returnere "Ok, passed 100 tests."hvilket indikerer at (2) er gyldig. Det vigtigt her at forstå dette ikke er det samme som at bevise at den er gyldig, da ikke alle muligheder er blevet testet. 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 anvende Test Driven Development (TDD)[2] til at teste om ens egenskab forbliver overholdt, under implementering.

[2] [15] *Test-Driven Development*.

3 Symbolske udtryk

Det ønskes at kunne repræsentere simple udtryk som en type i F#. Vil derfor gennemgå en del teori og funktion 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. Tanke gangen her at gennemgå en opbygning af en måde at kunne repræsentere udtryk samt simplificere 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 talgrupper. Typen for tal består af tre konstruktører, henholdsvis for heltal, rationale tal og komplekse tal. Dog er modulet designet med henblik på, at det kan udvides med flere taltyper. Ved udvidelse er det eneste krav til den nye talmængde, at der er definerede matematiske operationer i form af addition, subtraktion, negation, multiplikation og division. Desuden skal tallene inden for addition og multiplikation være associative. Dette gælder for eksempel ikke for en vektor, hvorfor vi senere vil overveje at udvide programmet med en type for vektorer. En udvidelse kunne være for reelle tal, som kan håndtere "floating point errors"[3], men for at undgå at komplicere programmet yderligere vil vi i denne opgave ikke inkludere decimal tal.

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 6: Typen for rationelle tal

Nedenfor er der givet en signaturfil for rational modulet 7. I implementeringsfilen overloades de matematiske operatorer ved hjælp af de klassiske regneregler for brøker[4].

```
module rational

[<Sealed>]
type rational =
    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
```

[3] [3] *Floating-point error*.
[4] [13] *Rational Number wikipedia*.

```
15      static member ( / )  : rational * int -> rational
16      static member ( / )  : int * int -> rational
17
18  val make          : int * int -> rational
19  val equal         : rational * rational -> bool
20  val posetive      : rational -> bool
21  val toString      : rational -> string
22  val isZero        : rational -> bool
23  val isOne         : rational -> bool
24  val isInt         : rational -> bool
25  val makeRatInt    : rational -> int
26  val greaterThan   : rational * rational -> bool
27  val isNegative    : rational -> bool
28  val absRational   : rational -> rational
```

Listing 7: 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)[5]. 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 rapportere 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. Derfor er 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 8.

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

Listing 8: Typen for komplekse tal

Vi begynder dermed med at opskrive en række regneregler i Definition 1 for operationer på komplekse tal, og betragter deres tilsvarende implementering i modulet.

Defination 1 (Regneregler for komplekse tal)**.**
Lad $a, b, c, d \in \mathbb{Q}$ *Så er følgende defineret omkring komplekse tal*[6]*:*

1. ***Addition***
   $(a + bi) + (c + di) = (a + c) + (b + d)i$
2. ***Subtraktion***
   $(a + bi) - (c + di) = (a - c) + (b - d)i$
3. ***Multiplikation***
   $(a + bi) \cdot (c + di) = (ac - bd) + (bc + ad)i$

[5] [5] *Greatest common divisor wikipedia.*
[6] [9] *Mathematics 1a*, se. Definition 3.3 s. 54, Definition 3.5 s. 56, Definition 3.8 s. 57, ligning 3-2 s. 58.

4. ***Kvadratisk form***
   $(a + bi) \cdot (a - bi) = a^2 + b^2$
5. ***Konjugering***
   $\overline{a + bi} = a - bi$
6. ***Division***
   $\frac{a+bi}{c+di} = \frac{(a+bi)\cdot(c-di)}{c^2+d^2}$

Ved implementeringen, se Listing 9, af addition, subtraktion, multiplikation samt skalering med et rationelt tal, som kan udledes fra multiplikation ved at lade den imaginære del være 0, er simple operationer, som ikke behøver at defineres i særskilte funktioner, men kan anvendes 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 9: Overskrivning af operationer på komplekse tal

Bortset fra division af to komplekse tal, ligner de resulterende overbelastninger 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 kan vi også opskrive en række egenskaber i 1 som vi ønsker at de skal opfylde. Egenskaberne vil blive testet i sektion 5.1.1.

Egenskab 1 (Egenskaber for tal)**.**
Lad $a, b, c \in$ Number, så gælder følgende egenskaber[7]*:*

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

Vi begynder med at definere en type for tal 10, som indeholder konstruktører for de tal typer vi har definerede samt en for heltal.

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

Listing 10: Typen for Number

Betragtes signatur filen for Number modulet 11, ses det at der igen er defineret overloading af de anvendte matematiske operationer. Derudover er der defineret en række funktioner som kan anvendes på Number typen.

```
module Number
open rational
open complex

type Number =
    | Int of int
    | Rational of rational
    | Complex of complex
with
    static member ( + )  : Number * Number -> Number
    static member ( - )  : Number * Number -> Number
    static member ( * )  : Number * Number -> Number
```

[7] [9] *Mathematics 1a*, se. Side 60 Theorem 3.10 og Theorem 3.11.

```
13      static member ( / )  : Number * Number -> Number
14      static member ( ~- ) : Number -> Number
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 11: 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 12 givet, som tager to tal og en funktion i form af den ønskede binærer operation som parameter. Funktionen vil derefter matche på de to tal og anvende den operation på de to tal.

```
1  // makeRational: Number -> rational
2  let makeRational a =
3      match a with
4      | Int x       -> make(x, 1)
5      | Rational x  -> x
6
7  // operation: Number -> Number -> (rational -> rational -> rational) -> Number
8  let operation a b f =
9      f (makeRational a) (makeRational b) |> Rational
```

Listing 12: Number.operation funktionen

Det vil her til være oplagt på alle de matematiske operationer at anvende en funktionen til at forsøge at konvenere tal typen til den simpleste talmængde, som i vores tilfælde er heltal. Dette er gjort ved at anvende funktionen tryMakeInt på alle de matematiske operations overloadnings 13.

```
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 13: Overloadnings funktionerne for Number

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 lignings udtryk.

3.2 Matematiske udtryk

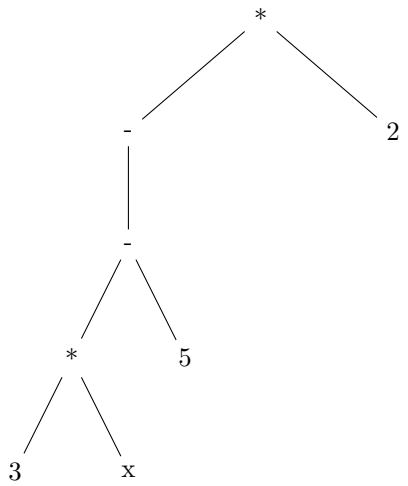
3.2.1 Polsk notation

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[8].

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

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 Figur 1.

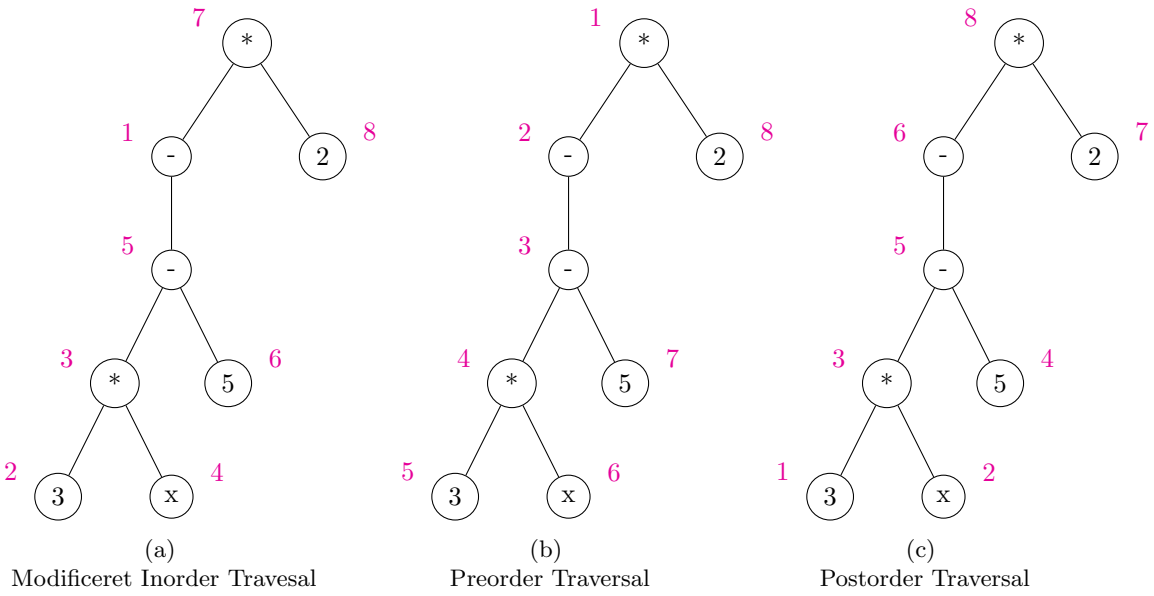


Figur 1: Et binært træ der repræsenterer udtrykket $-(3 \cdot x - 5) \cdot 2$

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

[8] [11] *Polsk notation wikipedia.*

være muligt omdanne dem til infix, prefix eller postfix notation. Dette kan gøres ved at anvende modificeret Inorder, Preorder eller Postorder Traversal[9], algorithmerne er illustreret i Figur 2.



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 2: Træet fra Figur 1 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 simplificere 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, som beskrevet i Listing 14. Denne type omfatter flere konstruktører, hver tilknyttet specifikke matematiske operationer vi ønsker at implementere. Desuden introducerer vi konstruktøren `N` til at repræsentere numeriske værdier ved at anvende talmængder defineret

[9] [16] *Tree Traversal Techniques – Data Structure and Algorithm Tutorials*.

i Listing 10. Til sidst tilføjer vi konstruktøren `X` for variable. Således lagres matematiske udtryk i en træstruktur, se 3.2.2, eftersom 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 14: 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 15. 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 15: $-(3 \cdot x - 5) \cdot 2$ som et udtryks træ. Funktionen tree bliver beskrevet i 3.4.

Signatur filen indeholder overloadings 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 containsX : Expr<Number> -> Expr<Number> -> bool
val getNumber : Expr<Number> -> Number
val getVariable : Expr<Number> -> char
```

Listing 16: Signatur filen for Expression modulet

De overloadede matematiske operatorer i Expressions, laver overflade evalueringer samt simplifikationer 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, mulige implementeringer af addition og multiplikation er givet i Listing 17. Lignende funktioner er implementeret for de andre operationer.

```
// mul: Expr<Number> -> Expr<Number> -> Expr<Number>
let rec mul e1 e2:Expr<Number> =
  match e1, e2 with
  |N a, N b                          -> N (a * b)
  |N a, b | b, N a when isOne a      -> b
  |N a, _ | _, N a when isZero a     -> N zero
  |Div(a, b), c | c, Div(a, b)      -> Div (mul a c, b)
  |Div (a, b), Div (c, d)           -> Div ((mul a c), (mul b d))
  |Neg a, Neg b                      -> mul a b
  | _, _                             -> Mul(e1, e2)

// add: Expr<Number> -> Expr<Number> -> Expr<Number>
let rec add e1 e2:Expr<Number>  =
  match e1, e2 with
  | N a, N b                                -> N (a + b)
  | N a, b | b, N a when isZero a           -> b
  | a, b when a = b                         -> Mul (N two, b)
  | Neg a, Neg b                            -> neg (add a b)
  | Neg a, b | b, Neg a                     -> Sub (b, a)
  | Mul(a, X b), Mul(c, X d)
  | Mul(X b, a), Mul(c, X d)
  | Mul(a, X b), Mul(X d, c)
  | Mul(X b, a), Mul(X d, c) when b = d -> Mul(add a c, X b)
  | _, _                                    -> Add(e1, e2)
```

Listing 17: 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 testet 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 18 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 18: Evaluering af et udtryk

3.4 Konventering mellem udtryks notation

Det er ønkes 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 inverser. Dette krav er yderligere uddybet i egenskab 3. Egenskaben bliver test i sektion 5.1.3.

Egenskab 3 (Invers morphism[10] 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 fra en expression til infix notation. Funktionen etf se Listing 19 fortager denne konventering ved at lave en modificeret Inorder Traversal på udtrykket, som beskrevet i 3.2.2. Den modificeret del er at håndtere parenteser samt håndtere 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
    | 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)
```

[10] [7] *Inverse function.*

```
    | 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 19: konventering fra expression til infix notation

Funktionen `tree`, som foretager konverteringen fra infix notation til et udtrykstræ, er baseret på algoritmen beskrevet i [1][11]. 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[12]. Typen for disse tokens kan ses i Listing 20. 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 20: Konvertering fra infix til udtrykstræ

3.5 Simplifikation af udtryk

Vi skal nu betragte en sytematisk metode til at kunne simplificere matematiske udtryk, ved hjælp af simple algebraiske regler. Dette er en nødvendig 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 udtrykene visualiseres. Før vi betragter metoden, kan vi opskrive en egenskab som simplification skal overholde. Egenskaben vil blive testet i sektion 5.1.4, det er en nødvendighed at evaluere udtrykket før og efter simplifikationen da det ikke er kompleks opgave at skulle sammenligne og evaluere to udtryk er ækvivalente.

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

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

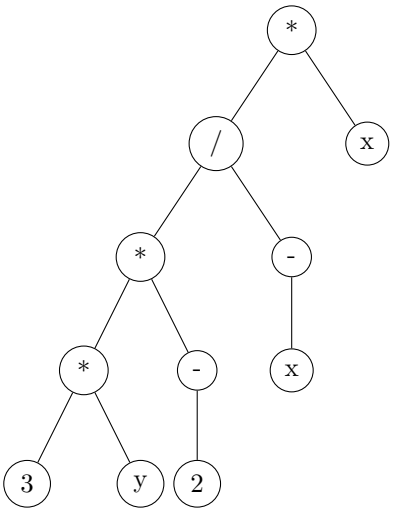
[11] [1] *Convert Infix expression to Postfix expression.*
[12] [10] *Operator Precedence and Associativity in C.*

Vi begynder med at betragte funktionen simplifyExpr i Listing 21, som simplificerer et udtryk ved at foretage en Postorder Traversal på udtrykket. På den måde sikre sig at når en node i udtrykstræet bliver simplificeret, vil dens børn allerede være simplificeret.

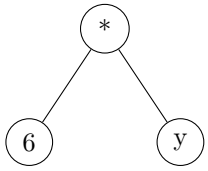
```
1  //simplifyOperation: Expr<Number> -> Expr<Number> -> (Expr<Number> -> Expr<
     Number> -> Expr<Number>) -> Expr<Number>
2  let rec simplifyOperation e1 e2 f =
3  match f e1 e2 with
4  | Neg a ->
5    commutativeMulDiv.applyCommutative (Neg a) |> commutativeAddSub.
     applyCommutative
6  | Add(a, b) when isAdd f -> commutativeAddSub.applyCommutative (Add(a, b))
7  | Sub(a, b) when isSub f -> commutativeAddSub.applyCommutative (Sub(a, b))
8  | Mul(a, b) when isMul f -> commutativeMulDiv.applyCommutative (Mul(a, b))
9  | Div(a, b) when isDiv f -> commutativeMulDiv.applyCommutative (Div(a, b))
10 | Add(a, b) -> simplifyOperation a b (+)
11 | Sub(a, b) -> simplifyOperation a b (-)
12 | Mul(a, b) -> simplifyOperation a b (*)
13 | Div(a, b) -> simplifyOperation a b (/)
14 | a -> a
15
16
17 //simplifyExpr: Expr<Number> -> Expr<Number>
18 let rec simplifyExpr e =
19   match e with
20   | N a when Number.isNegative a -> Neg (N (Number.absNumber a))
21   | N (Rational(R(a, b))) ->
22     simplifyOperation (simplifyExpr (N (Int a))) (simplifyExpr (N (Int b)))
     (/)
23   | Neg a      -> - (simplifyExpr a)
24   | Add(a, b) -> simplifyOperation (simplifyExpr a) (simplifyExpr b) (+)
25   | Sub(a, b) -> simplifyOperation (simplifyExpr a) (simplifyExpr b) (-)
26   | Mul(a, b) -> simplifyOperation (simplifyExpr a) (simplifyExpr b) (*)
27   | Div(a, b) -> simplifyOperation (simplifyExpr a) (simplifyExpr b) (/)
28   | _ -> e
```

Listing 21: Simplifikation af et udtryk

Det er simplifyOperation, som foretager selve simplificeringen af et givet udtryk. Funktionen tager som input to udtryk samt den binære operation, der skal anvendes på disse. Funktionen anvendes på udtrykkene, hvorefter overfladisk simplifikation, som blev beskrevet i 3.2.3, udføres. Hvis overfladisk simplifikation resulterer i en ændring af den anvendte operation, kalder funktionen sig selv rekursivt med de to udtryk og den nye operation. Hvis overfladisk simplifikation ikke resulterer i ændringer i operationen, vil funktionen foretage en dybere simplifikation af de to udtryk. Denne dybere simplifikation 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 multiplikationssimplifikation på et udtryk er givet i Figur 3, som viser det visuelle input og det resulterende svar fra funktionen, samt Figur 4, der viser det udfladede træ og sorteringen af det fladede træ.

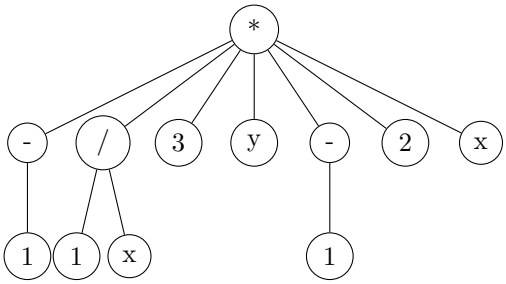


(a) Udtrykket $(3 \cdot y \cdot (-2)/-x) \cdot x$ som et træ

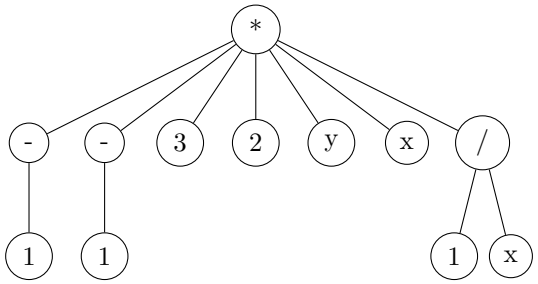


(b) Det resuterende simplificerede efter at have anvendt CommutativeMulDiv.applyCommutative på udtrykket i Figur 3a

Figur 3: Før og efter simplifikation af et udtryk ved brug af CommutativeMulDiv.applyCommutative



(a) Det udfladede træ



(b) Det sorterede træ fra venstre mod højre

Figur 4: Det udfladede og sorterede af udtrykket $(3 \cdot y \cdot (-2)/-x) \cdot x$ i processen af kommutativ simplifikation

Generelt, når det gælder simplificering af udtryk, skal man være opmærksom på ikke at ende i et uendeligt loop. Derfor er det vigtigt ikke at definere nogle overfladiske simplificeringer, 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 4, 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 simplificere 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[13])**.**
Lad f og g være udtryk, og a og b 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 22, 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 22: 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 23. Funktionen isolateX undersøger først, om den variabel, som skal isoleres, befinder sig på højre eller venstre side, derefter kaldes funktionen expressionOnX, som fungerer

[13] [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 23: 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 24.

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

Listing 24: Typen for order

Typen Order, anvendes til at angive, om en matrix er i rækkefølge (row-major) eller kolonnefølge (column-major)[14]. 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 25.

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

Listing 25: 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 26.

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

Listing 26: 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 27. 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 27: Funktion til at finde dimissionen af en matrix

[14] [14] *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:*[15]

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 28.

```
// 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 28: 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[16].

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, _)) =
```

[15] [9] *Mathematics 1a*, Theorem 7.2 s. 155.
[16] [12] *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 29: Funktion til at addere matricer og substraktion af vektorer

Addition af vektorer reduceres til at udføre additionen elementvis, som vist i Listing 29, 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 multiplikation i næste sektion og Gram-Schmidt-processen i sektion 4.4. 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 multiplikation

Defination 3 (Matrix-Vektor Multiplikation)**.**
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 Multiplikation)**.**
Lad A *være en matrix med* m *rækker og* n *søjler, og lad* $\mathbf{v}$ *være en vektor med* n *elementer. Så gælder der*

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

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

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

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

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

□

Vi kan dermed anvende Sætning 1 til at definere en funktion matrixMulVector til matrix-vektor multiplikation se Listing 30. Denne funktion skalerer først søjlerne i matricen med de tilsvarende elementer i vektoren og summer derefter rækkerne i matricen.

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

Listing 30: Funktion til matrix-vektor multiplikation

Defination 4 (Matrix multiplikation)**.**
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[17]

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

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 31 til at multiplicere to matricer, derfor indebærer matrix-vektor multiplikation på hver søjle i matricen. Da matrixMulVector 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 -> matrixMulVector a bv |> matrixToVector ) vlb
    M(product, C)
```

Listing 31: Funktion til at multiplicere 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 $Y = span\{y\}$[18]*.*

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

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
```

[17] [9] *Mathematics 1a*, Definition 7.12 s. 162.
[18] [8] *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 32: 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 Række-echelon form

4.4 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 10, 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
```

```
11          | M([], _), M(v1::vrest, o) ->
12              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 33: Dannelsen af en ortogonal basis, ved hjælp af Gram-Schmidt processen

Listing 33 viser implementeringen af Gram-Schmidt processen. Lavet udfra beskrivelse side 45 i 'Mathematics 1b'[19].

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.

[19] [8] *Mathematics 1b - Functions of several variables*, s. 45.

4.5 linært lignings system

5 PBT af programmet

5.1 PBT af udtryk

5.1.1 Tal modulet

5.1.2 Homomorfisme af evaluering

5.1.3 Invers morphism mellem infix og prefix

5.1.4 Simplifikation af udtryk

5.1.5 Differentiering af udtryk

5.2 PBT af vektorer og matricer

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. Først defineres en generator for matricer, som generere matricer med tilfældige tal fra vores talmængde 10.

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

type NumberGen =
    static member Number() =
        {new Arbitrary<Number>() with
            override _.Generator = numberGen
            override _.Shrinker _ = Seq.empty}
```

Listing 34: Generatorene anvendt til PBT af matrix operationer i

Vi kan dermed nu lave definere egenskaberne fra 6 som nogle funktioner, og teste dem med PBT.

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

Listing 35: Egenskaberne fra sætning 6 som funktioner

```
- Arb.register<MaxtrixGen>()
- Arb.register<NumberGen>()
- 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 36: Outputtet fra PBT af vektor Listing 35

5.2.2 PBT af Gram-Schmidt

Udfrodringen ved at lave en PBT af Gram-Schmidt er at vektorsættet skal være lineært uafhængige. Derfor laves der en generator som ved at udføre tilfældige række operationer på en diagonal matrix, kan generere en matrix med lineært uafhængige vektorer. #TODO : Lav et bevis for det beholder enskaben for lineært uafhængighed.

```
// getBacismatrixGen : int -> Gen<Matrix>
let getBacismatrixGen n =
    Gen.map (fun x -> standardBacis x) (Gen.choose (2, n))

// performRowOperationGen : Matrix -> Gen<Matrix>
let performRowOperationGen m =
    let (D(n, _)) = dimMatrix m
    gen {
        let! i = Gen.choose(1, n)
        let! j = match i with
            | 1 -> Gen.choose(2, n)
            | _ when i = n -> Gen.choose(1, n-1)
            | _ -> Gen.oneof [
                    Gen.choose(1, i-1);
                    Gen.choose(i+1, n)]
        let! a = numberGen
        return rowOperation i j a m }


// multipleRowOperationsGen : Matrix -> int -> Gen<Matrix>
let rec multipleRowOperationsGen m count =
    if count <= 0 then Gen.constant m
    else
        gen {
            let! newMatrix = performRowOperationGen m
            return! multipleRowOperationsGen newMatrix (count - 1)
        }

```

```
// getIndependetBacisGen : Gen<Matrix>
let getIndependetBacisGen =
    gen {
        let! m = getBacismatrixGen 5
        let! numberOfOperations = Gen.choose(1, 10)
        let! span = multipleRowOperationsGen m numberOfOperations
        return span }

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

Listing 37: Generatorene anvendt til PBT af Gram-Schmidt

Listing 37 viser de forskellige generatorer, som anvendes til PBT (Property-Based Testing) af Gram-Schmidt-processen. Først genereres en tilfældig basis matrix. Dernæst 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 Number. Denne proces gentages et tilfældigt antal gange.

Dernæst skal vi bruge en funktion til at tjekke om en matrix er en ortogonal basis. isOrthogonalBacis i Listing 38 tjekker om alle vektorerne i en matrix er ortogonale, ved at tjekke om søjle v_i er ortogonal med v_{i+1}, for alle $i \in [1, n-1]$ hvor n er længden på søjlerne. To søjler er ortogonale hvis deres indreprodukt er 0.

```
// isOrthogonalBacis : Matrix -> bool
let rec isOrthogonalBacis (M(vl, o)) =
    if not <| corectOrderCheck (M(vl, o)) C
    then isOrthogonalBacis <| correctOrder (M(vl, o)) C
    else
    match vl with
    | [] -> true
    | _::[] -> true
    | v::vnext::vrest -> innerProduct v vnext = zero && isOrthogonalBacis (M(
    vnext::vrest, o))
```

Listing 38: Funktion til at tjekke om søjlerne i en matrix er en ortogonal basis

PB testen gramSchmidtIsOrthogonal bliver derfor blot at tjekke om en matrix bestående af lineært uafhængige vektorer, der udspænder et underrum, er orthogonale efter Gram-Schmidt processen er blevet anvendt. Grundet tilfældige matematiske operationer, opstår der en støre mængde opstå overflow fejl, derfor godtages disse men klassificeres som overflow.

```
let gramSchmidtIsOrthogonal (m:IndependetBacis) =
    let res =
        try
            if orthogonalBacis m |> isOrthogonalBacis then 1 else 0
        with
            | :? System.OverflowException -> 2
    (res = 1 || res = 2)
    |> Prop.classify (res = 1) "PropertyHolds"
    |> Prop.classify (res = 2) "OverflowException"
```

Listing 39: PBT af Gram-Schmidt processen

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

Listing 40: Output fra PBT af Gram-Schmidt processen

Outputtet fra PBT af Gram-Schmidt processen kan ses i Listing 40. Som sædvanligt indikere testen kun korrekthed, men ikke garanteret korrekthed.

6 Diskussion

7 Konklusion

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

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

[6] Michael R. Hansen og Hans Rischel. *Functional Programming Using F#.* Cambridge University Press, 2013. ISBN: 9781107019027, 1107019028.

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

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

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

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

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

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

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

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

[15] *Test-Driven Development.* URL: https://en.wikipedia.org/wiki/Test-driven_development (hentet 31.03.2024).

[16] *Tree Traversal Techniques – Data Structure and Algorithm Tutorials.* URL: https://www.geeksforgeeks.org/tree-traversals-inorder-preorder-and-postorder/ (hentet 30.03.2024).

8 Appendiks

Hele projektet kan findes på Github, på følgende reference [4]. 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 41: 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
```

```
73          ->  match e, s with
74              | (x, precX, lr), (y, precY, _)
75                  when precX < precY || (precX = precY && lr = Left)
76                  -> generateExpresion c stack_tail (popprefixStack y prefix)
77              | _, _ -> generateExpresion tail (e::stack) prefix
78
79  // Converts a infix string to a expression tree
80  let tree s =
81      match generateExpresion (infixToTokenList s) [] [] with
82      | [] -> failwith "Tree is empty"
83      |tree::_ -> tree
84
85  // Adds parenthesis to a string if a boolean is true
86  let parenthesis b f =
87      if b then "(" + f + ")" else f
88
89  // Converts a expression tree to a infix string
90  let rec etf e p =
91      match e with
92      | N a when not <| isInt a -> parenthesis p <| toString a
93      | N a -> toString a
94      | X a -> string a
95      | Neg a ->  parenthesis p <| "-" + etf a (not p)
96      | Add(a, b) -> parenthesis p <| etf a false + "+" + etf b false
97      | Sub(a, b) -> parenthesis p <| etf a false + "-" + etf b true
98      | Mul(a, b) -> parenthesis p <| etf a true + "*" + etf b true
99      | Div(a, b) -> parenthesis p <| etf a true + "/" + etf b true
100
101 // Converts a expression tree to a infix string
102 let infixExpression e = etf e false
```

Listing 42: TreeGenerator.fs

8.3 CommutativeAddSub.fs

```
1   module commutativeAddSub
2   open Expression
3   open Number
4
5
6   // rank when sorting a commutative expression
7   let expressionSortRank e1 =
8       match e1 with
9       | N _ -> 1
10      | Neg (N _) -> 2
11      | X _ -> failwith "Variables should not be in the list" // all veriables
        are multiplied with 1
12      | Add _ -> failwith "Addition should not be in the list" // all addition
        should be reduced
13      | Sub _ -> failwith "Subtraction should not be in the list" // all
        subtraction should be reduced
14      | Mul _ -> 6
15      | Div _ -> 7
16      | Neg _ -> 8
17
18  // Flattens a expression tree with respect to addition and subtraction
19  let rec flatTree e =
20      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 43: 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 44: 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 // multiplies a matrix and a vector A.v = b - Definition 7.10
200 let rec matrixMulVector 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 matrixMulVector (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 -> matrixMulVector 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)  = matrixMulVector 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 // Inner product of two vectors
283 let innerProduct u v =
284     let (V(w, _)) = conjugateVector v |> vectorMulElementWise u
285     List.fold (+) zero w
286
287 // Projection of a vector on another vector
288 let proj y x =
289     scalarVector (innerProduct x y / innerProduct y y) y
290
291 // Dot product of two vectors
292 let dotProduct (V(u, ou)) (V(v, ov)) =
293     if List.length u <> List.length v
294     then failwith "dotProduct: Vectors most have same length."
295     else
296
297     match ou, ov with
298     | R, C -> innerProduct (V(u, C)) (V(v, C))
299     | _ -> failwith "Missing implementation for this case."
300
301
302 // Orthogonal bacis using the Gram-Schmidt process
303 let rec orthogonalBacis m =
304     if not <| corectOrderCheck m C
305     then orthogonalBacis (correctOrder m C)
306     else
307
308     // Gram-Schmidt process but without the normalization
309     let rec Gram_Schmidt vm acc_wm =
310         match acc_wm [], vm with
311         | x, M([], _) -> x
312         | M([], _), M(v1::vrest, o) ->
313             Gram_Schmidt (M(vrest,o))
314             <| fun x -> extendMatrix (M([v1], C)) x
315         | M(w, _), M(vk::vrest, o) ->
316             let (V(wk, _)) = vk - sumProj w vk
317             Gram_Schmidt (M(vrest,o))
318             <| fun x -> extendMatrix (acc_wm wk) x
319
320     // Sum all the projections of vk on w1 to wk-1
321     and sumProj w vk =
322         List.map (fun x -> proj x vk) w
323         |> matrix
324         |> sumRows
325         |> matrixToVector
326
327     Gram_Schmidt m (fun _ -> M([], C))
328
329 // checks is if every vector has inner product of zero with the next vector
330 let rec isOrthogonalBacis (M(vl, o)) =
331     if not <| corectOrderCheck (M(vl, o)) C then isOrthogonalBacis <|
    correctOrder (M(vl, o)) C
332     else
333     match vl with
334     | [] -> true
335     | _::[] -> true
```

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

```
392
393     match m with
394     | [] -> failwith "swapFirstWith: Matrix is empty"
395     | h::tail -> M(swapper h tail i 1 [], o)
396
397 // Muliplies the i'th vector with a scalar
398 let scalarIthVector c i (M(m, o)) =
399     let rec sIV c m i idx acc_m =
400         match m with
401         | [] -> failwith "scalarIthVector: Index out of range"
402         | x::xs when idx = i -> acc_m @ c * x :: xs
403         | x::xs -> sIV c xs i (idx + 1) (acc_m @ [x])
404
405     M(sIV c m i 0 [], o)
406
407 // row echelon form of a matrix
408 let rec rowEchelonForm A =
409     if not <| corectOrderCheck A R then rowEchelonForm (correctOrder A R)
410     else
411     let (D(r, c)) = dimMatrix A
412     match A with
413     |M([], _) -> A
414     | _ when isZeroMatrix A -> A
415     | M(v::_, _) when r = 1 -> firstNonZero v |> inv |> scalarMatrix A
416     | M(_, o) ->
417         let (i, j) = firstNonZeroIndexMatrix A 0 (-1 ,-1)
418         let (M(B, _)) = swapFirstWith A i
419         let b = List.head B |> firstNonZero
420         let (M(B, _)) = scalarIthVector (inv b) 0 (M(B, o))
421         let R1 = List.head B
422         let R2m = List.tail B
423         let B = rowOps j 1 r R1 (M(R2m, o))
424         let (M(Cm, _)) = rowEchelonForm B
425         M(R1::Cm, o)
426
427 // Ri <- Ri - b * R1 - posible error i mat 1 notes Algorithm 1 (should be b <-
      the jth entry og the ith row if B)
428 and rowOps coloumn i nrows R1 acc_m  =
429     if i >= nrows then acc_m else
430     let Ri = getMatrixIthVector (i - 1) acc_m
431     let b = getVectorIthNumber coloumn Ri
432     rowOps coloumn (i + 1) nrows R1 <| replaceMatrixIthVector (i-1) acc_m (Ri
    - b * R1)
433
434
435 // A standard bacis vector of length n with 1 at i
436 let standardBacisVector n i =
437     let rec sbv idx =
438         match idx with
439         | _ when idx < 1 -> []
440         | 1 when i <> 1 -> [zero]
441         | x when x = i -> one :: sbv (x - 1)
442         | _ -> zero :: sbv (idx - 1)
443     vector <| sbv n
444
445 // A standard bacis matrix of F^n
446 let standardBacis n =
447     let rec sb idx =
448         match idx with
```

```
449         | _ when idx < 1 -> []
450         | 1 -> [standardBacisVector n 1]
451         | _ -> standardBacisVector n idx :: sb (idx - 1)
452     matrix <| sb n
453
454 // fullranked diagonal matrix
455 let fullrankedDiagonalMatrix n m =
456     if n > m then failwith "fullrankedDiagonalMatrix: number of rows must be
    less than or equal to the number of columns"
457     else
458     let rec frdm i frm =
459         match i with
460         | _ when i < 0 -> failwith "fullrankedDiagonalMatrix: i must be
    greater than or equal to 0"
461         | 0 -> frm
462         | _ ->
463             let (V(zeroV, _)) = vectorOf zero n
464             frdm (i - 1) <| extendMatrix frm zeroV
465     frdm (m - n) <| standardBacis n
466
467
468 // Alters row j with Rj <- Rj - c * Ri
469 let rec rowOperation i j c m =
470     if i = j then failwith "rowOperation: Row i and j must be diffrent j <> i"
471     elif not <| corectOrderCheck m R then rowOperation i j c <| correctOrder m
     R
472     else
473     replaceMatrixIthVector (j-1) m <| getMatrixIthVector (j-1) m - c *
    getMatrixIthVector (i-1) m
474
475 // Checks if a matrix is upper triangular
476 let rec isUpperTriangular (M(vl, o)) =
477     if not <| corectOrderCheck (M(vl, o)) R then isUpperTriangular (
    correctOrder (M(vl, o)) R)
478     else
479     let (D(_, m)) = dimMatrix (M(vl, o))
480     let rec iut vl idx =
481         match vl with
482         | [] -> true
483         | x::xs when idx >= m -> isZeroVector x && iut xs (idx + 1)
484         | x::xs -> firstNonZeroIndex x = idx && iut xs (idx + 1)
485     iut vl 0
486
487 // determines if a matrix has full rank
488 let hasFullRank m =
489     rowEchelonForm m |> isUpperTriangular
490
491 // split a matric into two matrices
492 let rec splitMatrix o i m =
493     if not <| corectOrderCheck m o then splitMatrix o i (correctOrder m o)
494     else
495     let (M(vl, _)) = m
496
497     let rec sm vl idx acc =
498         match vl with
499         | [] -> failwith "splitMatrix: Index out of range"
500         | x::xs when idx = 0 -> matrix <| acc @ [x], matrix xs
501         | x::xs -> sm xs (idx - 1) (acc @ [x])
502     sm vl i []
```

```
503
504 // Determines if a matrix has a zero column
505 let rec dontHaveZeroCols m =
506     if not <| corectOrderCheck m C then dontHaveZeroCols (correctOrder m C)
507     else
508     let (M(vl, _)) = m
509     List.forall (fun x -> not <| isZeroVector x) vl
510
511
512 // Determines if a matrix has the same span as another matrix
513 let hasSameSpan m1 m2 =
514     let (D(r, c)) = dimMatrix m1
515     if (D(r, c)) <> dimMatrix m2 then failwith "Matrices must have the same
    dimension"
516     else
517
518     let hSS m1 m2 =
519         let s1, s2 = extendMatrixWithMatrix m1 m2 |> rowEchelonForm |>
    splitMatrix C (c-1)
520         hasFullRank s1 && dontHaveZeroCols s2 && hasFullRank s2
521
522     hSS m1 m2 && hSS m2 m1
523
524
525
526
527
528 ////////////////////////////////
529 /// Functions to solve Ax = b ///
530 ////////////////////////////////
531
532 type ExprVector = list<Expr<Number>>
533
534
535 // multiplies a expression with a vector and returns a Expr list
536 let scalarWithExpr (V(nl, _)) e=
537         List.map (fun x -> N x * e) nl
538
539 // matrix multiplication with expresion vector
540 let matrixMulExprList vl el =
541     let znl = (vectorOf zero (List.length el))
542     let zeroExprList = scalarWithExpr  znl (N zero) // zero expr vector
543     List.map2 (fun mc n -> scalarWithExpr n mc) el vl
544     |> List.fold (fun a b -> (List.map2 (+) a b)) zeroExprList
545
546 // Creates a vector of n variables
547 let rec charVector n =
548     match n with
549     | _ when n <= 0 -> []
550     | _ -> X (char (n - 1)) :: charVector (n - 1)
551
552 // inserts an eviroment into a vector
553 let rec vectorEnv n env =
554     match n with
555     | _ when n <= 0 -> V([], C)
556     | _ ->
557         let (V(nl, _)) = vectorEnv (n - 1) env
558         Map.find (char (n - 1)) env :: nl |> vector
559
```

```
560 // solves the system of lineary equations
561 let rec solveEquations el bl cl =
562         match el, bl, cl with
563         | [], [], [] -> Map.empty
564         | e::es, b::bs, c::cs ->
565                               let env = solveEquations es bs cs
566                               let (lhs, rhs) = isolateX (insertEnv e env) b c
567                               Map.add (getVariable lhs) (getNumber rhs) env
568         | _, _, _ -> failwith "solveEquations: The number of equations and
    variables must be the same"
569
570 // Solves the equation Ax = b
571 let Axequalb A (V(nlb, ob)) =
572     let (D(r, c)) = dimMatrix A
573     if r <> c then failwith "Axequalb: A must be a square matrix"
574     elif r <> List.length nlb ||  ob = R then failwith "Axequalb: b must be a
    column or have same length as rows of A"
575     else
576     let (V(ef_b, _), M(ef_vl, o)) = correctOrder (rowEchelonForm <|
    extendMatrix A nlb) C |> extractlastVector
577     if not <| isUpperTriangular (M(ef_vl, o)) then failwith "Axequalb: There
    dont exitst a single solution"
578     else
579     let varlist = charVector c
580     let b = scalarWithExpr (vector ef_b) (N one)
581     solveEquations (matrixMulExprList ef_vl varlist) b varlist
582     |> vectorEnv c
```

Listing 45: Matrix.fs