Danmarks Tekniske Universitet

Bachelor Projekt

Funktionel Modellering af Matematiske Systemer i F#

Jonas Dahl Larsen (s205829)

15. maj 2024

Indhold

1 Introduktion **3**

2 Fundamentale koncepter **4**
2.1 Introduktion til Funktions Programmering . . . 4
2.2 Typer . . . 5
2.3 Signatur filer og implementerings filer . . . 5
2.4 Overskrivning af operatorer . . . 6
2.5 Property Based Testing . . . 6

3 Symbolske udtryk **7**
3.1 Tal mængder . . . 7
3.1.1 Rationelle tal modul . . . 7
3.1.2 Komplekse tal-modul . . . 8
3.1.3 Tal modulet . . . 9
3.2 Matematiske udtryk . . . 12
3.2.1 Polsk notation . . . 12
3.2.2 Udtryk som træer . . . 12
3.2.3 Udtryksmodulet . . . 13
3.3 Evaluering af udtryk . . . 15
3.4 Konventering mellem udtryks notation . . . 16
3.5 Simplifikation af udtryk . . . 17
3.6 Differentiering af udtryk . . . 20
3.7 Multivariable polynomier af første grad . . . 20

4 Vektorer og Matricer **22**
4.1 Matrix operationer . . . 22
4.2 Matematiske operationer . . . 23
4.2.1 Skalering af en matrix . . . 23
4.2.2 Addition af matricer . . . 23
4.2.3 Matrix produkt . . . 24
4.2.4 Projektion af en vektor . . . 26
4.3 Gram-Schmidt . . . 27
4.4 Række-echelon form . . . 28
4.5 Lineært ligningssystem . . . 29

5 PBT af programmet **30**
5.1 PBT af udtryk . . . 30
5.1.1 Tal modulet . . . 30
5.1.2 Homomorfisme af evaluering . . . 30
5.1.3 Invers morphism mellem infix og prefix . . . 30
5.1.4 Simplifikation af udtryk . . . 30
5.1.5 Differentiering af udtryk . . . 30
5.2 PBT af vektorer og matricer . . . 30
5.2.1 PBT af matrix operationer . . . 30
5.2.2 PBT af Gram-Schmidt . . . 31

6 Diskussion **33**

7 Konklusion **33**

8 Appendiks **35**

8.1 complex.fsi 35
8.2 TreeGenerator.fs 35
8.3 CommutativeAddSub.fs 37
8.4 CommutativeMulDiv.fs 39
8.5 Matrix.fs 41

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 fordele og ulemper ved funktionel programmering i matematiske sammenhænge. I 2023 valgte Danmarks Tekniske Universitet (DTU) at anvende Python som et hjælpeværktøj i deres grundlæggende matematik kursuser „01001 Matematik 1a“ og „01002 Matematik 1b“. Imidlertid åbner funktionel programmering op for et andet perspektiv og andre 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 fra de grundlæggende matematik kurser på DTU.

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

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

Et eksempel på en implementering 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#

```
# Fakultet i Python
def factorial(n):
    if n == 0:
        return 1
    elif n > 0:
        return n * factorial(n - 1)
    else:
        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 ’$\to$’, 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# ved udregning af $(3!)! = 6! = 720$.

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 $Factorial : int \to int$. Det ses dermed tydeligt hvordan f# benytter notationen for afbildninger i mattematik, da den matematiske funktion for fakultet er en afbildning $f : \mathbb{Z} \to \mathbb{Z}$. 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:

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

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. Som eksempel på dette kan vi definere en multivariable funktion $f(x, y) = \sqrt{x! + y!}$ som er en afbildning $f : \mathbb{Z}^2 \to \mathbb{R}$ samt $g(y) = f(3, y) = \sqrt{3! + y!}$ som afbilder $g : \mathbb{Z} \to \mathbb{R}$, de tilsvarende F# funktion kan defineres på følgende måder:

```
// f: int -> int -> float
let f x y = sqrt <| float(factorial x + factorial y)

// g: int -> float
let g = f 3
```

Listing 4: Eksempel typerne for en multivariable funktion i F#

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 implementeringsfil kan have en signaturfil med .fsi extension, denne fil indeholder en beskrivelse af de typer og funktioner i implementeringsfilen som er tilgængelige for andre filer. En signaturfil kan derfor bruges som et blueprint for andre der ønsker at anvende eller replicere implementeringsfilen. I andre programmerings sprog vil man anse funktionerne i signaturfilen, som værende de funktioner der er tilgængelige ved åbning af modulet.

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

2.4 Overskrivning af operatorer

I F# er det muligt at overskrive standardoperatorer, så de kan anvendes på egne typer. Denne teknik vil blive benyttet igennem rapporten til at definere matematiske operationer for de typer, vi udvikler.

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 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 „01001 Matematik 1a" på DTU lærer man at anvende sandhedstabeller, som demonstrerer gyldigheden af en formel. Eksempelvis vises hvordan (2) er gyldig.

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

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 5.

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

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

Listing 5: 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 6: Output ved PBT af (2)

Check.Quick er en del af „FsCheck"-biblioteket. Den tager en funktion som argument og genererer en række tilfældige input til funktionen på baggrund af funktionens type. Hvis funktionen returnerer „true" for alle input, vil testen lykkes. Hvis funktionen returnerer „false" for et input, vil testen fejle og give et eksempel på et input, der fejlede. I Listing 5 er der anvendt „Check.Quick" til at teste, om (2) er gyldig. Funktionen „Check.Quick" returnerer „Ok, passed 100 tests," hvilket indikerer, at (2) er gyldig. Det er vigtigt her at forstå, at dette ikke er det samme som at bevise, at (2) 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 anvendes Test Driven Development (TDD)[2] til at teste, om ens egenskab forbliver overholdt under implementeringen.

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

3 Symbolske udtryk

Det ønskes at kunne repræsentere simple udtryk som en type i F#. Vi skal 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. Tankegangen 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 hvor real- og imaginærdelen er rationale.

3.1.1 Rationelle tal modul

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

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

Listing 7: Typen for rationelle tal

Nedenfor er der givet en signaturfil for rational modulet 8. I implementeringsfilen overloades de matematiske operatorer ved hjælp af de klassiske regneregler for brøker[3].

```
module rational


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

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

[3] [13] *Rational Number wikipedia*.

```
val isInt         : rational -> bool
val makeInt    : rational -> int
val greaterThan  : rational * rational -> bool
val isNegative   : rational -> bool
val absRational  : rational -> rational
```

Listing 8: 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)[4]. 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 9.

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

Listing 9: 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[5]*:*

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

[4] [5] *Greatest common divisor wikipedia*.
[5] [9] *Mathematics 1a*, se. Definition 3.3 s. 54, Definition 3.5 s. 56, Definition 3.8 s. 57, ligning 3-2 s. 58.

Ved implementeringen, se Listing 10, 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 10: 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 „Number“ 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 Number-typen)**.**
Lad $a, b, c \in Number$, så gælder følgende egenskaber[6]*:*

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

Vi begynder med at definere typen for tal i Listing 11, 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 11: Typen for Number

Typen Number er designet med henblik på, at den kan udvides med flere taltyper. Ved udvidelse er det et krav til den nye talmængde, at der er definerede matematiske operationer i form af addition, subtraktion, negation, multiplikation og division. Den nye Number type skal derudover forsat opfylde egenskaberne 1. En udvidelse kunne være for reelle tal, som kan håndtere „floating point errors“[7], men for at undgå at komplicere programmet yderligere vil vi i denne opgave ikke inkludere decimal tal.

Betragtes signatur filen for Number modulet i Listing 12, 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
    static member ( / )  : Number * Number -> Number
    static member ( ~- ) : Number -> Number
```

[6] [9] *Mathematics 1a*, se. Side 124 Definition 6.1.
[7] [3] *Floating-point error*.

```
val zero          : Number
val one           : Number
val two           : Number
val isZero        : Number -> bool
val isOne         : Number -> bool
val isNegative    : Number -> bool
val absNumber     : Number -> Number
val greaterThan   : Number -> Number -> bool
val tryReduce     : Number -> Number
val toString      : Number -> string
val conjugate     : Number -> Number
val inv           : Number -> Number
val isInt         : Number -> bool
```

Listing 12: 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 13 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. Dermed bliver heltal og rationale tal konverteret til vores komplekse tal type, før operationen udføres.

```
// makeComplex: Number -> complex
let makeComplex n =
  match n with
  | Int x -> newComplex (newRational(x, 1), newRational(0, 1))
  | Rational x -> newComplex (x, newRational(0, 1))
  | Complex x -> x

// operation: Number -> Number -> (complex -> complex -> complex) -> Number
let operation a b f =
    f (makeComplex a) (makeComplex b) |> Complex
```

Listing 13: operation funktionen til udførelse af matematiske operationer

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 operationers overloadnings 14.

```
// tryMakeInt: Number -> Number
let tryMakeInt r =
    match r with
    | Rational a when isInt a -> Int (makeRatInt a)
    | _ -> r

type Number with
  static member (+) (a, b) = operation a b (+) |> tryMakeInt
  static member (-) (a, b) = operation a b (-) |> tryMakeInt
  static member (*) (a, b) = operation a b (*) |> tryMakeInt
  static member (/) (a, b) = operation a b (/) |> tryMakeInt
  static member (~-) (a)   = neg a |> tryMakeInt
```

Listing 14: Overloadnings funktionerne for Number

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

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

3.2 Matematiske udtryk

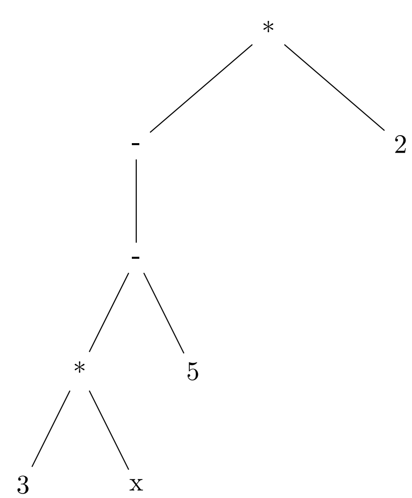
3.2.1 Polsk notation

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

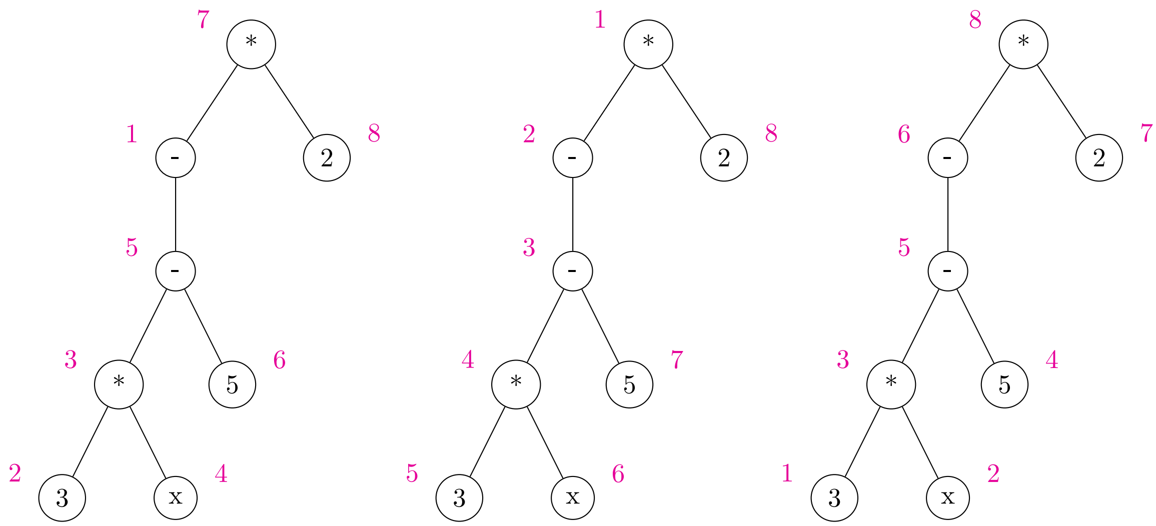
Et matematisk udtryk kan repræsenteres som et binært træ, hvor bladene er operander i det anvendte matematiske rum og alle andre noder er operationer. Som eksempel kan udtrykket $-(3 \cdot x - 5) \cdot 2$ repræsenteres som følgende træ Figur 1.



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

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

Det skal bemærkes der er forskel på den unære og binære operator '$-$' i træet, den unære betyder negation og den binære er subtraktion. Givet et binært træ for et matematisk udtryk, vil det være muligt omdanne dem til infix, prefix eller postfix notation. Dette kan gøres ved at anvende modificeret Inorder, Preorder eller Postorder Traversal[9], algorithmerne er illustreret i Figur 2.



(a) Modificeret Inorder Travesal

(b) Preorder Traversal

(c) Postorder Traversal

$$\begin{aligned} \text{Modificeret Inorder Traversal:} \quad & -(3 \cdot x - 5) \cdot 2 \\ \text{Preorder Traversal:} \quad & \cdot - - \cdot 3\, x\, 5\, 2 \\ \text{Postorder Traversal:} \quad & 3\, x\ \cdot 5\ - -2\, \cdot \end{aligned}$$

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 15. Denne type omfatter flere konstruktører, hver

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

tilknyttet specifikke matematiske operationer vi ønsker at implementere. Desuden introducerer vi konstruktøren `N` til at repræsentere numeriske værdier ved at anvende talmængder defineret i Listing 11. 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 15: 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 16. 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 16: $-(3 \cdot x - 5) \cdot 2$ 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 isZero : Expr<Number> -> bool
val containsX : Expr<Number> -> Expr<Number> -> bool
```

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

Listing 17: 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, mullige implementeringer af addition og multiplikation er givet i Listing 18. 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 18: 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 19 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 19: 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 20 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
```

[10] [7] *Inverse function.*

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


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

Listing 20: 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 21. 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 21: 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.

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

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

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

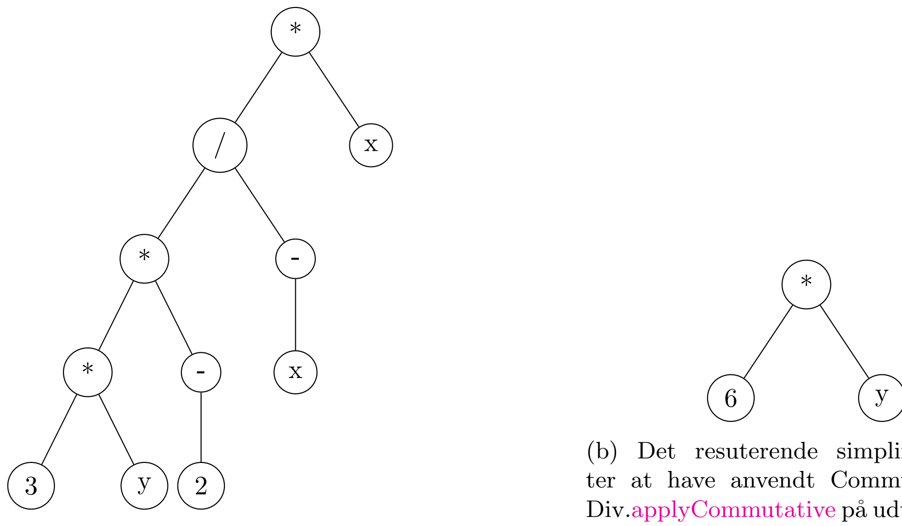
Vi begynder med at betragte funktionen `simplifyExpr` i Listing 22, som simplificerer et udtryk ved at foretage en Postorder Traversal på udtrykket. På den måde sikre sig at når en node i udtrykstræet bliver simplificeret, vil dens børn allerede være simplificeret.

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


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

Listing 22: Simplifikation af et udtryk

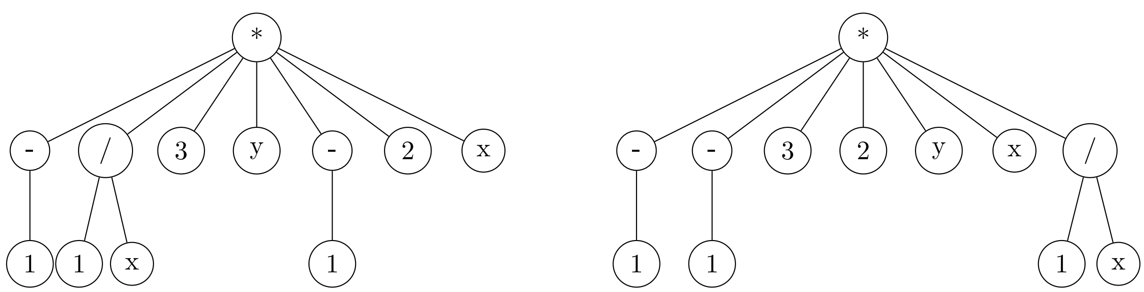
Det er simplifyOperation, som foretager selve simplificeringen af et givet udtryk. Funktionen tager som input to udtryk samt den binære operation, der skal anvendes på disse. Funktionen anvendes på udtrykkene, hvorefter overfladisk simplifikation, som blev beskrevet i 3.2.3, udføres. Hvis overfladisk 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 23, 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 23: 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 multivari-

[13] [2] *Differentiation Rules*.

abelt polynomium af første grad. Vi betragter derfor nu to simple funktioner til at udføre denne isolation, se Listing 24. Funktionen isolateX undersøger først, om den variabel, som skal isoleres, befinder sig på højre eller venstre side, derefter kaldes funktionen expressionOnX, som fungerer 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 24: 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 25.

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

Listing 25: 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 26.

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

Listing 26: 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 27.

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

Listing 27: 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 28. 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 28: 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 $i = 1 \ldots m$ 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 29.

```
// 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 29: 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*.

```
    M (List.map2 addVector vl1 vl2, o)

// subVector : Vector -> Vector -> Vector
let subVector x y =
    scalarVector (-one) y |> addVector x

// sumRows : Matrix -> Matrix
let rec sumRows m =
    if not <| corectOrderCheck m C
        then sumRows <| correctOrder m C
    else
        let zeroVector = vectorOf zero <| matrixVectorLength m
        let (M(vl, _)) = m
        matrix [List.fold (addVector) zeroVector vl]
```

Listing 30: Funktion til at addere matricer og substraktion af vektorer

Addition af vektorer reduceres til at udføre additionen elementvis, som vist i Listing 30, ved brug af List-funktionen map2. Vi kan bruge addVector til at definere matrix addition og subtraktion af vektorer. Vektor addition bruges også til at summere rækkerne i en matrix (sumRows), hvilket vil blive anvendt i implementeringen af matrix produkt i næste sektion og Gram-Schmidt-processen i sektion 4.3. Funktionen bliver yderligere beskrevet i definition 2.

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

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

Så gælder om sumRows at

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

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

4.2.3 Matrix produkt

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

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

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

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

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

$$Av = \mathit{sumRows} \begin{bmatrix} | & | & & | \\ a_1 v_1 & a_2 v_2 & \cdots & a_n v_n \\ | & | & & | \end{bmatrix}$$

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

$$\begin{aligned} B &= \begin{bmatrix} | & | & & | \\ a_1 v_1 & a_2 v_2 & \cdots & a_n v_n \\ | & | & & | \end{bmatrix} \\ &= \begin{bmatrix} a_{11}v_1 & a_{12}v_2 & \cdots & a_{1n}v_n \\ a_{21}v_1 & a_{22}v_2 & \cdots & a_{2n}v_n \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}v_1 & a_{m2}v_2 & \cdots & a_{mn}v_n \end{bmatrix} \end{aligned}$$

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

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

□

Vi kan dermed anvende sætning 1 til at definere en funktion matrixVectorProduct til matrix-vektor produkt se Listing 31. Denne funktion skalerer først søjlerne i matricen med de tilsvarende elementer i vektoren og summer derefter rækkerne i matricen.

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

Listing 31: Funktion til matrix-vektor produkt

Defination 4 (Matrix produkt).
Lad $A \in \mathbb{F}^{m\times n}$ *og* $B \in \mathbb{F}^{n\times \ell}$. *Lad søjlerne i* B *er være givet ved* $b_1, \ldots, b_\ell \in \mathbb{F}^n$, *dermed*[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 32 til at tage produktet af to matricer, ved at udfører matrix-vektor produkt på hver søjle i matricen. Da matrixVectorProduct evaluere til en matrix, skal der bruges en funktion til at konvertere matricen til en vektor, hvilket matrixToVector gør.

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

Listing 32: Funktion til at tage produktet af to matricer

4.2.4 Projektion af en vektor

Defination 5 (Projektion af en vektor).
Projektionen af en vektor på en linje defineres i som følgende, hvor $Y = span\{y\}$[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 33: Funktioner til at projicere en vektor på en anden

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

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

4.3 Gram-Schmidt

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

Før vi implementerer Gram-Schmidt processen, er vi dog begrænset af vores Number type fra Listing 11, idet $x \in \{\text{Number}\} \implies\!\!\!\!\!\!\not\;\;\; \sqrt{x} \in \{\text{Number}\}$. Derfor vil vi ikke normalisere vektorerne, hvilket medfører, at vi kun vil finde en ortogonal basis, fremfor en ortonormal basis.

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

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

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

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

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

```
            <| fun x -> extendMatrix (M([v1], C)) x
        | M(w, _), M(vk::vrest, o) ->
            let (V(wk, _)) = vk - sumProj w vk
            Gram_Schmidt (M(vrest,o))
            <| fun x -> extendMatrix (acc_wm wk) x

    // sumProj : Vector list -> Vector -> Vector
    and sumProj w vk =
        List.map (fun x -> proj x vk) w
        |> matrix
        |> sumRows
        |> matrixToVector

    Gram_Schmidt m (fun _ -> M([], C))
```

Listing 34: Dannelsen af en ortogonal basis, ved hjælp af Gram-Schmidt processen

Listing 34 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.

4.4 Række-echelon form

I forbindelse med udførelsen af PBT til Gram-Schmidt metoden vil vi blandt andet anvende række-echelon form til at sikre os, at et sæt af vektorer er lineært uafhængige. Derfor vil vi i denne sektion betragte implementeringen af række-echelon form, ud fra „Algorithm 9 for computing a row echelon form of a matrix“ angivet på side 142 af noterne til „01001 Mathematics 1a“[20]. Efter at have udført PBT af Gram-Schmidt metoden, slog den i første omgang fejl, hvilket efter nærmere undersøgelse viste sig at være grundet en mindre fejl i Algorithm 9. Linje 14 burde være „b ← the j'th entry of the i-th row of B“. Listing 35 viser implementeringen af række-echelon form, lavet ud fra psudokoden „Algorithm 9“.

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

[19] [8] *Mathematics 1b - Functions of several variables*, s. 45.
[20] [9] *Mathematics 1a*, s. 142.

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

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

Listing 35: Funktion til at finde række-echelon form

4.5 Lineært ligningssystem

Som den sidste del i anvendelsen af matricer vil vi beskrive, hvordan vi kan løse et lineært ligningssystem af formen $A\mathbf{x} = \mathbf{b}$. Ved at foretage række-echelon form på totalmatricen $[A \mid \mathbf{b}]$ kan vi finde en løsning til ligningssystemet, hvis ranken af A er forskellig fra ranken af den totale matrix[21].

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

Den nemmeste tilgang vil være at introducere to nye typer for at repræsentere en vektor og matrix, som kan indeholde variable. Dette kan gøres ved hjælp af vores type for udtryk fra Listing 15. Vi definerer derfor følgende typer i Listing 36.

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

Listing 36: Typer for at repræsentere en vektor og matrix med variable

Det er her undladt at definere lagringsordningen for matricerne af udtryk, da vi ikke bruger dem til mere end en mellemregning i løsningen af ligningssystemet. Men alle funktioner, vi har lavet i Listing 26, ville man også kunne lave for udtryksmatricer, hvis man anvender en lagringsordning.

Ved at have udtryksmatricer defineret som en liste af udtryk, kan vi anvende den samme teknik til at foretage matrix-vektor-produktet mellem ref(A) og $\mathbf{x}$, som vi benyttede i Listing 31. Dette

[21] [9] *Mathematics 1a*, Corollary 6.27 s. 146.

vil resultere i en liste af udtryk, som vi ved hjælp af funktionen isolateX i Listing 24 kan bruge til at isolere x_i i den i'te ligning ved at sætte den lig med b_i og indsætte det i ligningerne 1 til $i-1$.

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

5 PBT af programmet

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 11.

```
// 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 37: 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 38: 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 39: Outputtet fra PBT af vektor Listing 38

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 40: Generatorene anvendt til PBT af Gram-Schmidt

Listing 40 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 41 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 41: 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 42: PBT af Gram-Schmidt processen

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

Listing 43: Output fra PBT af Gram-Schmidt processen

Outputtet fra PBT af Gram-Schmidt processen kan ses i Listing 43. 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 44: 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
    | [] -> []
```

```
    | x::tail when x = ' ' -> mapToToken tail allowUnary allowOperator
    | x::tail when allowOperator && x = '/' || x = '*' -> Operator (x, 2, Left)::mapToToken tail false false
    | x::tail when allowOperator && (x = '+' || (x = '-' && not allowUnary)) -> Operator (x, 1, Left)::mapToToken tail false false
    | x::tail when  (x = '-' && allowUnary) -> Operator ('~', 2, Right)::mapToToken tail true false
    | x::tail when x = '(' -> Operator (x, -1, Left)::mapToToken tail true true
    | x::tail when x = ')' -> Operator (x, -1, Left)::mapToToken tail false true
    | x::_ when System.Char.IsDigit(x) ->
        let (k, tail) = foundInt l ""
        Konstant (int k):: mapToToken tail false true
    | x::tail when System.Char.IsLetter x -> Operand x::mapToToken tail false true
    | x::_ -> failwith ("Invalid syntax at: " + string x)

// Allowing more than one digit in a number
and foundInt l s =
    match l with
    | x::tail when System.Char.IsDigit(x) -> foundInt tail (s + string x)
    | _ -> (s, l)


// pops the last operator from the stack and adds it to the prefix list
let popprefixStack op prefix =
    match op, prefix with
    | x, e1::e2::tail when x = '+' -> Add(e2, e1)::tail
    | x, e1::e2::tail when x = '-' -> Sub(e2, e1)::tail
    | x, e1::e2::tail when x = '*' -> Mul(e2, e1)::tail
    | x, e1::e2::tail when x = '/' -> Div(e2, e1)::tail
    | x, e::tail when x = '~' -> Neg(e)::tail
    | x, _ when x = '(' -> prefix
    |_,_ -> failwith "match not found"

// Runs the algorithm to generate the expression tree
let rec generateExpresion c (stack:OperatorList) prefix =
    match c, stack with
    | [], [] -> prefix
    | [], (s,_,_)::stack_tail  -> generateExpresion c stack_tail (popprefixStack s prefix)
    | Operand x :: tail, _ -> generateExpresion tail stack (X x::prefix)
    | Konstant x :: tail, _ -> generateExpresion tail stack (N (Int x)::prefix)
    | Operator (x, prec, lr)::tail, _
        when x = '('
        -> generateExpresion tail ((x, prec, lr)::stack) prefix
    | Operator (x, _, _)::tail, _
        when x = ')'
        ->  match stack with
            | [] -> generateExpresion tail stack prefix
            | (s,_,_)::stack_tail
                when s = '('
                -> generateExpresion tail stack_tail prefix
            | (s,_,_)::stack_tail
                -> generateExpresion c stack_tail (popprefixStack s prefix)
    | Operator e::tail, [] -> generateExpresion tail (e::stack) prefix
    | Operator e::tail, s::stack_tail
```

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

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

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

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

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

Listing 45: TreeGenerator.fs

8.3 CommutativeAddSub.fs

```
module commutativeAddSub
open Expression
open Number


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

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

```
    | Add (a, b) -> flatTree a @ flatTree b
    | Sub (a, b) -> flatTree a @ flatTree (Neg b)
    | Neg (Add(a, b)) -> flatTree (Neg a) @ flatTree (Neg b)
    | Neg (Sub(a, b)) -> flatTree (Neg a) @ flatTree b
    | X a -> [Mul(N one, X a)]
    | Div (a, b) -> [a / b]
    | Neg (Neg a) -> flatTree a
    | N _ | Div _ | Mul _ | Neg _-> [e]



let rec sort l = List.sortBy (fun e -> expressionSortRank e) l

// Reduces a sorted commutative list for addition
let rec reduceNumbers l =
    // printfn "rn %A" l
    match l with
    | [] -> []
    | N a :: N b :: tail -> reduceNumbers (N a + N b :: tail)
    | N a :: Neg (N b) :: tail  -> reduceNumbers (N a - N b :: tail)
    | Neg (N a) :: Neg (N b) :: tail -> reduceNumbers (Neg (N a + N b) :: tail
    )
    | Neg (N a) :: tail -> Neg (N a) :: tail
    | N a :: tail -> N a :: tail
    | _ -> l

// given a list of commutative expressions, and a variable, sums the
    coerfficients of the variable
let rec sumInstancesOfVarible x l =
    match x, l with
    | _, [] -> ([] ,x)
    | Mul(N n1, X x1), Mul(N n2, X x2) :: tail
    | Mul(N n1, X x1), Mul(X x2, N n2) :: tail
    | Mul(X x1, N n1), Mul(N n2, X x2) :: tail
    | Mul(X x1, N n1), Mul(X x2, N n2) :: tail
        when x1 = x2    -> sumInstancesOfVarible (Mul(N (n1 + n2), X x1)) tail
    | _, head :: tail   ->
                        let (l_new, x_new) = sumInstancesOfVarible x tail
                        (head :: l_new, x_new)




let rec reduceVariables l =
    match l with
    | [] -> []
    | Mul(N a, X b) :: tail
    | Mul(X b, N a) :: tail
        ->
        let (l_new, x_new) = sumInstancesOfVarible (Mul(N a, X b)) tail
        x_new :: reduceVariables l_new
    | head :: tail -> head :: reduceVariables tail

let rec rebuldTree l =
    // printfn "rt %A" l
    match l with
    | [] -> N zero
    | Neg x::tail -> (rebuldTree tail) - x
    | Mul (a, b)::tail -> (rebuldTree tail) + (a * b)
```

```
    | Div (a, b)::tail -> (rebuldTree tail) + (a / b)
    | x::tail -> rebuldTree tail + x

let applyCommutative e =
    match e with
    | Sub _ | Add _ -> flatTree e |> sort |> reduceNumbers |> reduceVariables |> rebuldTree
    | _ -> e
```

Listing 46: CommutativeAddSub.fs

8.4 CommutativeMulDiv.fs

```
module commutativeMulDiv
open Expression
open Number


/////////////////////////////////////////////////////////////////
/// Commutative RULES For multiplication and division ///////////
/////////////////////////////////////////////////////////////////

// rank when sorting a commutativ expression
let expressionSortRank e1 =
    match e1 with
    | Neg _ -> 1
    | N _ -> 2
    | X _ -> 3
    | Add _ -> 4
    | Sub _ -> 5
    | Mul _ -> 6
    | Div _ -> 7

// sorts the commutativ list
let rec sort l = List.sortBy (fun e -> expressionSortRank e) l


// determines the sign of a commutativ list
let rec signList l s =
    match l with
    | [] -> s
    | Neg _::tail -> signList tail (-1*s)
    | _::tail -> signList tail s


// flattens the tree to a commutativ list
let rec flatTree e =
    match e with
    | N _ | X _ | Add _ | Sub _ -> [e]
    | Neg a -> Neg (N one) :: flatTree a
    | Mul (a, b) -> flatTree a @ flatTree b
    | Div (N a, N b) -> [N (a / b)]
    | Div (Neg a, b) | Div (a, Neg b) -> Neg (N one) :: flatTree (Div (a, b))
    | Div (a, N b) -> N (one / b) :: flatTree a
    | Div (a, b) -> Div (N one, b) :: flatTree a


// multiplys all Div elements in a commutativ list
```

```
let rec mulDivElements l =
    // division will always be in the end of a sorted list, and numarator will be 1
    match l with
    | [] -> []
    | Div (_, b) :: Div (_, c) :: tail -> mulDivElements (Div(N one, Mul(b, c)) :: tail)
    | x::tail -> x :: mulDivElements tail

// removes a element from a list
let rec removeElem e l =
    match  l with
    | [] -> []
    | x::tail when x = e -> tail
    | x::tail -> x :: removeElem e tail

// reduces a commutativ list
let rec reduce l =
    let sorted = divCancelling (sort l)
    if signList sorted 1 > 0 then rebuildTree sorted else Neg (rebuildTree sorted)

// initiates the division cancelling
and divCancelling l =
    // printfn "DivCancelling: %A" l
    match List.rev l with
    | [] -> l
    | Div(_, b)::tail ->
                        let (numerator, denominator) = cancelEquality  (List.rev tail) (sort (flatTree b))
                        sort (flatTree (reduce numerator / reduce denominator))
    | _ -> l

// cancels out equal elements in the numerator and denominator
and cancelEquality nu de =
    // printfn "cancelEquality: %A / %A" nu de
    match nu with
    | [] -> ([], de)
    | n::ntail when List.contains n de ->  cancelEquality ntail (removeElem n de)
    | n::ntail -> let (numerator, denominator) = cancelEquality ntail de
                  (n::numerator, denominator)



// rebuilds a commutativ list to a tree
and rebuildTree l =
    match l with
    | [] -> N one
    | Neg (N a)::tail when Number.isOne a -> rebuildTree tail
    | Neg a::tail -> rebuildTree (a::tail)
    | N a::N b::tail -> rebuildTree (N (a * b) :: tail)
    | N a :: Add(b, c) :: tail -> rebuildTree (Add(reduce (flatTree (Mul(N a, b))), reduce (flatTree (Mul(N a, c)))) :: tail)
    | N a :: Sub(b, c) :: tail -> rebuildTree (Sub(reduce (flatTree (Mul(N a, b))), reduce (flatTree (Mul(N a, c)))) :: tail)
    | a::tail -> a * rebuildTree tail

```

```

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

Listing 47: CommutativeMulDiv.fs

8.5 Matrix.fs

```
module Matrix
open Number
open Expression
open SymbolicManipolation

// row or coloumn major ordere: default of a vector and matric is Column major
    order.
type Order = | R | C

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

// a "Normal" is C major and a Transposed one is R major
type Vector = V of list<Number> * Order
type Matrix = M of list<Vector> * Order

// Construct a vector
let vector nl = V (nl, C)

// The dimension of a vector
let dimVector (V(v, o)) =
    let len = List.length v
    match o with
    | R -> D (1, len)
    | C -> D (len, 1)

// Makes sure that a matrix has the same major order as its vectors
let matrixValidMajor (M(m, o)) =
    match List.head m, o with
    | V (_, R), R -> true
    | V (_, C), C -> true
    | _, _ -> failwith "Matrix's vector's dont have same major order"

// The list length of a vector
let vectorLength (V(v, _)) = List.length v

// The list length of a matrix's vectors
let matrixVectorLength (M(m, _)) =
    match m with
    | [] -> 0
    | x::_ -> vectorLength x

// The dimension of a matrix
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)

// Multiplication of a scalar and a vector
let scalarVector (n:Number) (V (nl, o)) =
    V ((List.map (fun x -> x * n) nl), o)

// Multiplication of a scalar and a matrix
let scalarMatrix (M (vl, o)) n =
    M ((List.map (fun x -> scalarVector n x) vl), o)

// Addision two vectors
let addVector x y =
    let (V (v1, o1)) = x
    let (V (v2, o2)) = y
    if o1 <> o2
    then failwith "Vectors must have the same major order"
    elif dimVector x <> dimVector y
    then failwith "Vectors must have the same dimension"
    else
    V ((List.map2 (+) v1 v2), o1)

// Subtraction of two vectors
let subVector x y =
    scalarVector (-one) y |> addVector x


// Construct a vector of n with length len
let vectorOf n len = V ((List.init len (fun _ -> n)), C)

// Construct a matrix of n with dimension d
let matrixOf n (D (r, c)) = M ((List.init c (fun _ -> vectorOf n r)), C)

// Transposes a vector
let transposeVector (V(v, o)) =
    match o with
    | R -> V(v, C)
    | C -> V(v, R)

// adds a dimension to a vector
let extendVector (V(v, o)) n =
    V(v @ [n], o)

// Extend a matrix with a number list
let extendMatrix (M(m, o)) nl =
    if m <> [] && nl <> [] && List.length nl <> matrixVectorLength (M(m, o))
    then failwith "The list must have the same length as the matrix's vectors"
    elif nl = [] then M(m, o)
    else
    M (m @ [(V (nl, o))], o)

// extend a matrix with a matrix
let extendMatrixWithMatrix (M(m1, o1)) (M(m2, o2)) =
    if o1 <> o2 then failwith "Matrices must have the same major order"
    elif matrixVectorLength (M(m1, o1)) <> matrixVectorLength (M(m2, o2)) then
     failwith "Matrices must have the same dimension"
```

```
    else M (m1 @ m2, o1)


// Alternates the Major order
let alternateOrder o =
    match o with
    | R -> C
    | C -> R

// Alternates the Major order of a matrix
let alternateOrderMatrix (M(m, o)) =
    M (m, alternateOrder o)

// The i'th number of a vector
let getVectorIthNumber i (V(v, _)) = v.[i]

// The i'th vector of a matrix
let getMatrixIthVector i (M(m, _)) = m.[i]

// replaces the i'th vector of a matrix
let replaceMatrixIthVector i (M(m, o)) v =
    M (m.[0..i-1] @ [v] @ m.[i+1..], o)

// pops a vector
let seperateFistNumberFromVector (V(v, o)) =
    (v.Head, V(v.Tail, o))

// first element of a vector
let headVector (V(v, _)) = List.head v

// Gives a vector of all the first elements of the vectors in a matrix
let rec firstElemetsVectors (M(m, o)) v_acc m_acc =
    match m with
    | [] -> (v_acc, M(m_acc, o))
    | x::xs ->
            let (n, tail) = seperateFistNumberFromVector x
            firstElemetsVectors (M(xs, o)) (v_acc @ [n]) (m_acc @ [tail])

// Helper function for changeOrderMatrix
let rec chaingingOrderMatrix (M(m, o)) m_acc =
    match matrixVectorLength (M(m, o)) with
    | 0  -> m_acc
    | _  ->
        let (v, m_new) = firstElemetsVectors (M(m, o)) [] []
        extendMatrix m_acc v |> chaingingOrderMatrix m_new

// Changes the order of a matrix
let changeOrderMatrix (M(m, o)) =
    alternateOrderMatrix (M([], o)) |> chaingingOrderMatrix (M(m, o))

// Makes sure that two matrices have the same major order, if diffrent then
    changes the order of the second matrix
let giveMatrixHaveSameOrder (M(_, o1)) (M(m2, o2)) =
    if o1 <> o2 then changeOrderMatrix (M(m2, o2)) else M(m2, o2)

// Transposes a matrix
let transposeMatrix (M(vl, o)) =
    M(List.map (fun x -> transposeVector x) vl, alternateOrder o)

```

```
// Adds two matrices elemtwise
let addMatrix m1 m2 =
    let m3 = giveMatrixHaveSameOrder m1 m2
    if dimMatrix m1 <> dimMatrix m3
    then failwith "addMatrix: Matrices must have the same dimension"
    else
    let (M(vl1, o)) = m1
    let (M(vl3, _)) = m3
    M (List.map2 addVector vl1 vl3, o)

// Construct a matrix
let matrix vl =
    let rec mc vl m =
        match vl with
        | [] -> m
        | (V(x, _))::xs -> mc xs (extendMatrix m x)
    mc vl (M([], C))

// Changes the order of af matrix to x
let correctOrder (M(m, o)) x =
    if o = x then M(m, o) else changeOrderMatrix (M(m, o))

// Boolean value of if a matrix has the correct order
let corectOrderCheck (M(_, o)) x =
    o = x

// Sum the rows of a matrix
let rec sumRows m =
    if not <| corectOrderCheck m C
    then sumRows <| correctOrder m C
    else
    let zeroVector = vectorOf zero <| matrixVectorLength m
    let (M(vl, _)) = m
    matrix [List.fold (addVector) zeroVector vl]



// matrix vector product  A.v = b - Definition 7.10
let rec matrixVectorProduct m v =
    let (D(rv, _)) = dimVector v
    let (D(_, cm)) = dimMatrix m
    if rv <> cm
    then failwith "matrixVector: the number of columns of the matrix has to be the same as the number of entries in the vector."
    elif not <| corectOrderCheck m C then matrixVectorProduct (correctOrder m C) v
    else
    let (M(vl, _)) = m
    let (V(nl, _)) = v
    M (List.map2 (fun mc n -> scalarVector n mc) vl nl, C)
    |> sumRows

// Converts a matrix to a vector if possible
let matrixToVector m =
    match dimMatrix m, m with
    | D(r, c), M(v::_, _) when r = 1 || c = 1 -> v
    | _, _ -> failwith "mactrixToVector: Matrix is not a vector"

// Multiplies two matrices - Definition 7.12
```

```
let rec matrixProduct a b =
    let (D(_, ca)) = dimMatrix a
    let (D(rb, _)) = dimMatrix b
    if ca <> rb
    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"
    elif not <| corectOrderCheck b C then matrixProduct a (correctOrder b C)
    else
    let (M(vlb, _)) = b
    let product = List.map (
            fun bv -> matrixVectorProduct a bv |> matrixToVector ) vlb
    M(product, C)


type Vector with
    static member (+) (v1, v2) = addVector v1 v2
    static member (-) (v1, v2) = subVector v1 v2
    static member (*) (n, v)   = scalarVector n v
    static member (*) (v, n)   = scalarVector n v
    static member (~-) (v)     = scalarVector (-one) v


type Matrix with
    static member (+) (m1, m2) = addMatrix m1 m2
    static member (+) (m, n)   = dimMatrix m |> matrixOf n |> addMatrix m
    static member (+) (n, m)   = dimMatrix m |> matrixOf n |> addMatrix m
    static member (*) (n, m)   = scalarMatrix m n
    static member (*) (m, n)   = scalarMatrix m n
    static member (*) (m, v)   = matrixVectorProduct m v
    static member (*) (m1, m2) = matrixProduct m1 m2
    static member (/) (m, n)   = scalarMatrix m (inv n)


// Horisontal flip of a matrix
let flip m =
    let (M(m_new, _)) = correctOrder m C
    List.rev m_new |> matrix


// extracts the first vector of a matrix
let extractfirstVector (M(m, o)) =
    match m with
    | [] -> failwith "Matrix is empty"
    | x::xs -> x, M(xs, o)

// Last vector of a matrix
let rec extractlastVector m =
    let (v, mf) = flip m |> extractfirstVector
    (v, flip mf)


// Multiplies two vectors element wise
let vectorMulElementWise (V(u, o1)) (V(v, o2)) =
    if o1 <> o2
    then failwith "Vectors must have the same major order"
    elif List.length u <> List.length v
    then failwith "Vectors must have the same dimension"
    else
    V (List.map2 (*) u v, o1)
```

```

// Conjugates a vector
let conjugateVector (V(v, o)) =
    V (List.map conjugate v, o)

// Inner product of two vectors
let innerProduct u v =
    let (V(w, _)) = conjugateVector v |> vectorMulElementWise u
    List.fold (+) zero w

// Projection of a vector on another vector
let proj y x =
    scalarVector (innerProduct x y / innerProduct y y) y

// Dot product of two vectors
let dotProduct (V(u, ou)) (V(v, ov)) =
    if List.length u <> List.length v
    then failwith "dotProduct: Vectors most have same length."
    else

    match ou, ov with
    | R, C -> innerProduct (V(u, C)) (V(v, C))
    | _ -> failwith "Missing implementation for this case."


// Orthogonal bacis using the Gram-Schmidt process
let rec orthogonalBacis m =
    if not <| corectOrderCheck m C
    then orthogonalBacis (correctOrder m C)
    else

    // Gram-Schmidt process but without the normalization
    let rec Gram_Schmidt vm acc_wm =
        match acc_wm [], vm with
        | x, M([], _) -> x
        | M([], _), M(v1::vrest, o) ->
            Gram_Schmidt (M(vrest,o))
            <| fun x -> extendMatrix (M([v1], C)) x
        | M(w, _), M(vk::vrest, o) ->
            let (V(wk, _)) = vk - sumProj w vk
            Gram_Schmidt (M(vrest,o))
            <| fun x -> extendMatrix (acc_wm wk) x

    // Sum all the projections of vk on w1 to wk-1
    and sumProj w vk =
        List.map (fun x -> proj x vk) w
        |> matrix
        |> sumRows
        |> matrixToVector

    Gram_Schmidt m (fun _ -> M([], C))



// Checks if a vector is a zero vector
let isZeroVector (V(v, _)) =
    List.forall (fun x -> Number.isZero x) v

// Checks if a matrix is a zero matrix
```

```
let isZeroMatrix (M(m, _)) =
    List.forall (fun x -> isZeroVector x) m

// fist non zero element of a vector
let firstNonZero (V(v, _)) =
    if isZeroVector (V(v, C)) then failwith "firstNonZero: Vector does not
    have a non zero element"
    else
    List.find (fun x -> not (Number.isZero x)) v

// Index of the first non zero element of a vector
let firstNonZeroIndex (V(v, _)) =
    if isZeroVector (V(v, C)) then -1
    else
    List.findIndex (fun x -> not (Number.isZero x)) v

// string a vector
let stringVector (V(v, o)) =
    let space  = if o = C then "\n" else " "
    List.map (fun x -> toString x) v |> String.concat space

// string a matrix
let rec stringMatrix m =
    if not <| corectOrderCheck m R then stringMatrix (correctOrder m R)
    else
    let (M(vl, _)) = m
    List.map (fun x -> stringVector x) vl |> String.concat "\n"


// Index of the first non zero element of a matrix
let rec firstNonZeroIndexMatrix (M(m, o)) idx (r, c) =
    if m = [] then (r, c) else
    let fnxi = List.head m |> firstNonZeroIndex
    match m with
    | [] -> (r, c)
    | _::vt when fnxi >= 0 && (fnxi < c || c = -1) -> firstNonZeroIndexMatrix
    (M(vt, o)) (idx + 1) (idx, fnxi)
    | _::vt -> firstNonZeroIndexMatrix (M(vt, o)) (idx + 1) (r, c)


// Switches two vectors in a matrix
let swapFirstWith (M(m, o)) i =
    if i = 0
    then M(m, o)
    else
    let rec swapper h m i idx acc_m =
        match m with
        | [] -> failwith "swapFirstWith: Index out of range"
        | x::xs when idx = i -> x :: acc_m @ (h::xs)
        | x::xs -> swapper h xs i (idx + 1) (acc_m @ [x])

    match m with
    | [] -> failwith "swapFirstWith: Matrix is empty"
    | h::tail -> M(swapper h tail i 1 [], o)

// Muliplies the i'th vector with a scalar
let scalarIthVector c i (M(m, o)) =
    let rec sIV c m i idx acc_m =
        match m with
```

```
        | [] -> failwith "scalarIthVector: Index out of range"
        | x::xs when idx = i -> acc_m @ c * x :: xs
        | x::xs -> sIV c xs i (idx + 1) (acc_m @ [x])

    M(sIV c m i 0 [], o)

// Alters row j with Rj <- Rj - c * Ri
let rec rowOperation i j c m =
    if i = j then failwith "rowOperation: Row i and j must be diffrent j <> i"
    elif not <| corectOrderCheck m R then rowOperation i j c <| correctOrder m
     R
    else
    replaceMatrixIthVector (j-1) m <| getMatrixIthVector (j-1) m - c *
    getMatrixIthVector (i-1) m


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

// Ri <- Ri - b * R1 - posible error i mat 1 notes Algorithm 1 (should be b <-
     the jth entry og the ith row if B)
and rowOps coloumn i nrows R1 acc_m  =
    if i >= nrows then acc_m else
    let Ri = getMatrixIthVector (i - 1) acc_m
    let b = getVectorIthNumber coloumn Ri
    rowOps coloumn (i + 1) nrows R1 <| replaceMatrixIthVector (i-1) acc_m (Ri
    - b * R1)


// A standard bacis vector of length n with 1 at i
let standardBacisVector n i =
    let rec sbv idx =
        match idx with
        | _ when idx < 1 -> []
        | 1 when i <> 1 -> [zero]
        | x when x = i -> one :: sbv (x - 1)
        | _ -> zero :: sbv (idx - 1)
    vector <| sbv n

// A standard bacis matrix of F^n
let standardBacis n =
    let rec sb idx =
```

```
        match idx with
        | _ when idx < 1 -> []
        | 1 -> [standardBacisVector n 1]
        | _ -> standardBacisVector n idx :: sb (idx - 1)
    matrix <| sb n

// fullranked diagonal matrix
let fullrankedDiagonalMatrix n m =
    if n > m then failwith "fullrankedDiagonalMatrix: number of rows must be less than or equal to the number of columns"
    else
    let rec frdm i frm =
        match i with
        | _ when i < 0 -> failwith "fullrankedDiagonalMatrix: i must be greater than or equal to 0"
        | 0 -> frm
        | _ ->
            let (V(zeroV, _)) = vectorOf zero n
            frdm (i - 1) <| extendMatrix frm zeroV
    frdm (m - n) <| standardBacis n


// Checks if a matrix is upper triangular
let rec isUpperTriangular (M(vl, o)) =
    if not <| corectOrderCheck (M(vl, o)) R then isUpperTriangular (correctOrder (M(vl, o)) R)
    else
    let (D(_, m)) = dimMatrix (M(vl, o))
    let rec iut vl idx =
        match vl with
        | [] -> true
        | x::xs when idx >= m -> isZeroVector x && iut xs (idx + 1)
        | x::xs -> firstNonZeroIndex x = idx && iut xs (idx + 1)
    iut vl 0

// determines if a matrix has full rank
let hasFullRank m =
    rowEchelonForm m |> isUpperTriangular


// checks is if every vector has inner product of zero with the next vector
let rec isOrthogonalBacis m =
    let isob (M(vl, o)) =
        match vl with
        | [] -> true
        | _::[] -> true
        | v::vnext::vrest -> innerProduct v vnext = zero && isOrthogonalBacis (M(vnext::vrest, o))
    isob <| correctOrder m C && hasFullRank m



// split a matric into two matrices
let rec splitMatrix o i m =
    if not <| corectOrderCheck m o then splitMatrix o i (correctOrder m o)
    else
    let (M(vl, _)) = m

    let rec sm vl idx acc =
```

```
        match vl with
        | [] -> failwith "splitMatrix: Index out of range"
        | x::xs when idx = 0 -> matrix <| acc @ [x], matrix xs
        | x::xs -> sm xs (idx - 1) (acc @ [x])
    sm vl i []

// Determines if a matrix has a zero column
let rec dontHaveZeroCols m =
    if not <| corectOrderCheck m C then dontHaveZeroCols (correctOrder m C)
    else
    let (M(vl, _)) = m
    List.forall (fun x -> not <| isZeroVector x) vl


// Determines if a matrix has the same span as another matrix
let hasSameSpan m1 m2 =
    let (D(r, c)) = dimMatrix m1
    if (D(r, c)) <> dimMatrix m2 then failwith "Matrices must have the same dimension"
    else

    let hSS m1 m2 =
        let s1, s2 = extendMatrixWithMatrix m1 m2 |> rowEchelonForm |> splitMatrix C (c-1)
        hasFullRank s1 && dontHaveZeroCols s2 && hasFullRank s2

    hSS m1 m2 && hSS m2 m1





////////////////////////////////
/// Functions to solve Ax = b ///
////////////////////////////////

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



// multiplies a expression with a vector and returns a Expr list
let scalarWithExpr (V(nl, _)) e=
        List.map (fun x -> N x * e) nl

// matrix product with expresion vector
let matrixProductExprList vl el =
    let znl = List.head vl |> vectorLength |> vectorOf zero
    let zeroExprList = scalarWithExpr znl (N zero) // zero expr vector
    List.map2 (fun mc n -> scalarWithExpr n mc) el vl
    |> List.fold (fun a b -> (List.map2 (+) a b)) zeroExprList

// Creates a vector of n variables
let rec charVector n =
    match n with
    | _ when n <= 0 -> []
    | _ -> X (char (n - 1)) :: charVector (n - 1)

// inserts an eviroment into a vector
```

```
let rec vectorEnv n env =
    match n with
    | _ when n <= 0 -> V([], C)
    | _ ->
        let (V(nl, _)) = vectorEnv (n - 1) env
        Map.find (char (n - 1)) env :: nl |> vector

// solves the system of lineary equations
let rec solveEquations el bl cl =
        match el, bl, cl with
        | [], [], [] -> Map.empty
        | e::es, b::bs, _ when isZero e && isZero b ->
            solveEquations es bs cl
        | e::es, b::bs, c::cs ->
                                let env = solveEquations es bs cs
                                let (lhs, rhs) = isolateX (insertEnv e env) b c
                                Map.add (getVariable lhs) (getNumber rhs) env
        | _, _, _ -> failwith "solveEquations: The number of equations and
    variables must be the same"

// Solves the equation Ax = b
let matrixEquation A (V(nlb, ob)) =
    let (D(r, c)) = dimMatrix A
    if r <> List.length nlb ||  ob = R then
        failwith "matrixEquation: b must be a column vector or have same
    length as rows of A"

    // Perform row echelon form on the total matrix
    let ef_t = correctOrder (rowEchelonForm <| extendMatrix A nlb) C

    // Extract the last vector of the matrix
    let (V(ef_b, _), M(ef_vl, o)) = extractlastVector ef_t

    // Check if the system of equations has a solution
    if hasFullRank ef_t then
        failwith "matrixEquation: The system of equations has no solution"
    elif not <| hasFullRank (M(ef_vl, o)) then
        failwith "matrixEquation: The system of equations has infinite
    solutions"

    let varlist = charVector c
    let b = scalarWithExpr (vector ef_b) (N one)
    solveEquations (matrixProductExprList ef_vl varlist) b varlist |>
    vectorEnv c
```

Listing 48: Matrix.fs