Danmarks Tekniske Universitet

Bachelor Projekt

Funktionel Modellering af Matematiske Systemer i F#

Jonas Dahl Larsen (s205829)

7. maj 2024

Indhold

1 Introduktion **2**

2 Fundamentale koncepter **3**
2.1 Introduktion til Funktions Programmering . . . 3
2.1.1 Typer . . . 4
2.2 Signatur filer og implementerings filer . . . 4
2.3 Overloading operatorer . . . 4
2.4 Property Based Testing . . . 5

3 Symbolske udtryk **6**
3.1 Tal mængder . . . 6
3.1.1 Rationelle tal Mondul . . . 6
3.1.2 Komplekse tal Mondul . . . 7
3.1.3 Tal Mondulet . . . 7
3.2 Matematiske udtryk . . . 9
3.2.1 Polsk notation . . . 9
3.2.2 Udtryk som træer . . . 9
3.2.3 Udtryksmodulet . . . 11
3.3 Evaluering af udtryk . . . 13
3.3.1 Konventering mellem udtryks notation . . . 14
3.4 Simplifikation af udtryk . . . 15
3.5 differentiering af udtryk . . . 15
3.6 PBT af udtryk . . . 15
3.6.1 Homomorfisme af evaluering . . . 15
3.6.2 Invers morphism mellem infix og prefix . . . 15

4 Vektorer og Matricer **15**
4.1 Matrix operationer . . . 16
4.1.1 Matematiske operationer . . . 16
4.1.1.1 Skalering af en matrix . . . 17
4.1.1.2 Addition af matricer . . . 17
4.1.1.3 Matrix multiplikation . . . 18
4.1.1.4 Projektion af en vektor . . . 20
4.2 PBT af matrix operationer . . . 21
4.3 Række-echelon form . . . 22
4.4 Gram-Schmidt . . . 22
4.5 PBT af Gram-Schmidt . . . 23

5 Appendiks **25**
5.1 TreeGenerator.fs . . . 25
5.2 Matrix.fs . . . 25

1 Introduktion

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

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

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

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

2 Fundamentale koncepter

2.1 Introduktion til Funktions Programmering

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

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

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

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

Listing 1: Eksempel på Fakultet i F#

```
# 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 '$\rightarrow$', på den linje, hvor mønsteret er genkendt.

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

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

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

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

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

2.1.1 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 $int \rightarrow int$. Det betyder, at det ikke er muligt at kalde funktionen med et argument, der ikke er af typen int. Typen for funktionen beskrives som $Factorial : int \rightarrow int$. Vi kan derfor formulere følgende omkring typer[1]:

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

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:

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

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

2.2 Signatur filer og implementerings filer

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

2.3 Overloading operatorer

I F# er det muligt at overskrive standard operatorer, så man kan anvende dem på egne typer. Det vil igennem rapporten blive anvendt til at definere matematiske operationer på de typer som vi kommer til at bygge.

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

2.4 Property Based Testing

Property Based Test (PBT) er en teknik til at teste korrekthed af egenskaber som man ved altid skal være opfyldt. En PBT test generer en række tilfældige input til en funktion og tester om en egenskab er opfyldt. Hvis en egenskab ikke er opfyldt, vil PBT give et eksempel på en fejl. De matematiske studerende på DTU, begynder med at lære om logik. I den forbindelse lærer man at en udsagnslogisk formel er gyldig (tautologi) hvis den altid er sand. Der eksistere mange teknikker til at vise at en udsagnslogisk formel er gyldig, i DTU's matematiske kursus lærer man at anvende sandhedstabellen. De viser hvordan 2 er gyldig.

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

Vi vil også kunne anvende PBT til at undersøge om (2) er gyldig, ved at udtrykke egenskaben som en funktionen af P, Q og R se Listing 3 4.

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

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

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

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

Listing 5: Output ved PBT af (2)

Check.Quick er en del af "FsCheck"biblioteket, den tager en funktion som argument, og genere en række tilfældige input til funktionen på baggrund af funktionens type. Hvis funktionen returnere "true"for alle input, vil testen lykkedes. Hvis funktionen returnere "false"for et input, vil testen fejle og give et eksempel på et input der fejlede. I Listing 4 er der anvendt "Check.Quick"til at teste om (2) er gyldig. Funktionen "Check.Quick"returnere "Ok, passed 100 tests."hvilket indikerer at (2) er gyldig. Det vigtigt her at forstå dette ikke er det samme som at bevise at den er gyldig, da ikke alle muligheder er blevet testet. I dette tilfælde kan vi regne sandsynligheden for hvor vidt alle kombinationer er testet. $P, Q, R \in \{True, False\}$, derfor er der $2^3 = 8$ mulige kombinationer for input til funktionen. Hvis vi antager at alle kombinationer er lige sandsynlige, er sandsynligheden for at alle kombinationer er blevet testet $1 - (\frac{7}{8})^{100} = 0.999998$. Derfor er det meget sandsynligt at (2) er gyldig.
Det vil generelt ikke være muligt at regne denne sandsynlighed, da vi senere vil anvende PBT til at teste funktioner der tager argumenter som ikke har et endeligt antal kombinationer. Dog vil det stadig give en god indikation hvorvidt en egenskab er overholdt. I nogle tilfælde vil det være en fordel at opskrive en PBT før implementeringen af en funktion som man ved skal overholde en egenskab, på den måde anvende Test Driven Development (TDD)[2] til at teste om ens egenskab forbliver overholdt, under implementering.

[2] *Test-Driven Development*.

3 Symbolske udtryk

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

3.1 Tal mængder

Vi begynder med opbygningen af et mondul som kan repræsentere tal mængder. Typen for tal, består af tre konstruktører, for henholdvis heltal, rationale tal og komplekse tal. Dog er mondulet lavet med henblik på at kunne udvides med flere typer af tal. Måden resten af programmet er lavet på, gør de eneste grav til tal er at der er definerede matematiske operationer i form af addition, subtraktion, negation, multiplikation og division. Samt at tallet inden for addition og multiplikation er associative. Dette gælder blandt andet ikke for en vektor, derfor vil vi senere betragte at udvide programmet med en type for vektorer. En udvidelse kunne være for reele tal, som kan håndtere "floating point errors"[3], men for ikke at komplicere programmet vil vi i denne opgave ikke betragte floats.

3.1.1 Rationelle tal Mondul

Repræsentationen af rationale tal kan laves ved hjælp af danne et par af integers, hvor den ene integer er tælleren og den anden er nævneren.

```
    [language={FSharp},
    label={type_rationel},
    caption={Typen for rationelle tal}]
type rational = R of int * int
```

Nedestående er der givet en signatur fil for rational mondulet 6. i Implementerings filen overloades de matematiske operatorer, ved hjælp af de klassiske regneregler for brøker[4].

```
module rational

[<Sealed>]
type rational =
    static member ( ~- ) : rational -> rational
    static member ( + )  : rational * rational -> rational
    static member ( + )  : int * rational -> rational
    static member ( - )  : rational * rational -> rational
    static member ( - )  : int * rational -> rational
    static member ( * )  : int * rational -> rational
    static member ( * )  : rational * int -> rational
    static member ( * )  : rational * rational -> rational
```

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

```
    static member ( / )  : rational * rational -> rational
    static member ( / )  : int * rational -> rational
    static member ( / )  : rational * int -> rational
    static member ( / )  : int * int -> rational

val make          : int * int -> rational
val equal         : rational * rational -> bool
val posetive      : rational -> bool
val toString      : rational -> string
val isZero        : rational -> bool
val isOne         : rational -> bool
val isInt         : rational -> bool
val makeRatInt    : rational -> int
val greaterThan   : rational * rational -> bool
val isNegative    : rational -> bool
val absRational   : rational -> rational
```

Listing 6: Signatur filen for rational mondulet

For at kunne sammenligne, men også for nemmere at undgå for store brøker, vil alle rationelle tal blive reduceret til deres simpleste from. Dette kan gøres ved at finde den største fælles divisor (GCD)[5]. Der udover er det vigtigt at være opmærksom på man ikke foretager nul division. Derfor vil implementerings filen kaste en "System.DivideByZeroException"hvis nævneren er eller bliver nul. Signatur filen indeholder en række funktioner som bliver anvendt af andre filer.

3.1.2 Komplekse tal Mondul

3.1.3 Tal Mondulet

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. Vi begynder med at definere en type for tal 7, som indeholder konstruktører for de tal typer vi har definerede samt en for heltal.

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

Listing 7: Typen for Number

Betragtes signatur filen for Number mondulet 8, 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 rantionalAndComplex
open rational
open complex

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

[5] *Greatest common divisor wikipedia.*

```
    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




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
```

Listing 8: Signatur filen for Number mondulet

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 9 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.

```
// makeRational: Number -> rational
let makeRational a =
    match a with
    | Int x        -> make(x, 1)
    | Rational x   -> x

// operation: Number -> Number -> (rational -> rational -> rational) -> Number
let operation a b f =
    f (makeRational a) (makeRational b) |> Rational
```

Listing 9: Number.operation funktionen

Det vil her til være oplagt på alle de matematiske operationer at anvende en funktionen til at forsøge at konvenere tal typen til den simpleste talmængde, som i vores tilfælde er heltal. Dette er gjort ved at anvende funktionen tryMakeInt på alle de matematiske operations overloadnings 10.

```
// 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 10: Overloadnings funktionerne for Number

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

3.2 Matematiske udtryk

3.2.1 Polsk notation

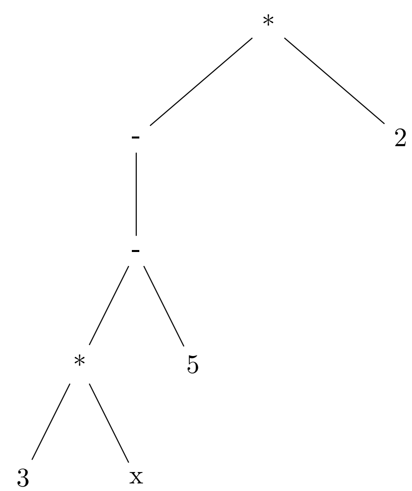
Matematiske udtryk som vi normalt kender dem er skrevet med infix notation. I infix notation skrives en binær operator mellem to operandere, kende tegnet for sproget er at det indeholder parenteser samt præcedens regler. Dette gør det generalt kompliceret at evaluere og håndtere matematiske udtryk i et programmeringssprog. Derfor er det mere oplagt at kunne anvende polsk notation (prefix) istedet, hvor operatoren skrives før operandere eller omvendt polsk notation (postfix). Da de hverken indeholder parenteser eller præcedens regler[6].

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

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.

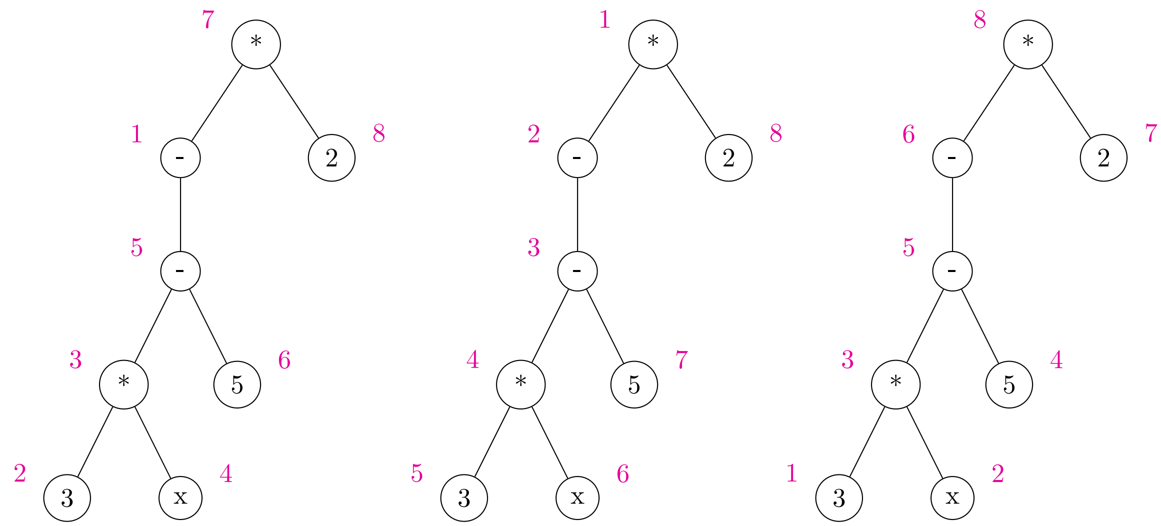
[6] *Polsk notation wikipedia*.



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

Det skal bemærkes der er forskel på den unære og binære operator '$-$' i træet, den unære betyder negation og den binære er subtraktion. Givet et binært træ for et matematisk udtryk, vil det være muligt omdanne dem til infix, prefix eller postfix notation. Dette kan gøres ved at anvende modificeret Inorder, Preorder eller Postorder Traversal[7], algorithmerne er illustreret i Figur 2.

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



(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 mondul som kan repræsentere udtryk ved brug af prefix notation. Postorder Traversal blive 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 med et modul for matematiske udtryk. Vi starter med at definere en polymorf type for udtryk, som beskrevet i Listing 11. Denne type omfatter flere konstruktører, hver tilknyttet specifikke matematiske operationer vi ønsker at implementere. Desuden introducerer vi konstruktøren `N` til at repræsentere numeriske værdier ved at anvende talmængder defineret i Listing 7. 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 11: 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 12. 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 12: $-(3 \cdot x - 5) \cdot 2$ som et udtryks træ. Funktionen tree bliver beskrevet i 3.3.1.

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

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

Listing 13: Signatur filen for Expression mondulet

De overloadede matematiske operatorer i Expressions, laver overflade evalueringer samt simplifikationer på deres respektive argumenter. Overfalde evalueing vil sige at de individuellen 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 14.

```
// 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
  | 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)

// mul: Expr<Number> -> Expr<Number> -> Expr<Number>
let 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
  | _, _                             -> Mul(e1, e2)
```

Listing 14: 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 1. Egenskaben vil blive testet i sektion 3.6.1.

Egenskab 1 (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 15 tager et udtryk og et miljø som input og evaluere udtrykket til en numerisk værdi. Funktionen kører en Preorder Traversal på udtrykket og evaluerer dermed udtrykket nedefra og op, ved at foretage matematiske operationer defineret i Number mondulet.

```
// 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 15: Evaluering af et udtryk

3.3.1 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 2. Egenskaben bliver test i sektion 3.6.2.

Egenskab 2 (Invers morphism[8] 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 16 fortager denne konventering ved at lave en modificeret Inorder Traversal på udtrykket, som beskrevet i 3.2.2. Den modificeret del er at håndtere parenteser samt håndtere negation som var det en binær node i træet hvor det venstre barn er et tomt udtryk.

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

// etf: Expr<Number> -> bool -> string
let rec etf e p =
    match e with
    | N a -> toString a
    | X a -> string a
    | Neg a -> "-" + etf a true |> parenthesis p
    | Add(a, b) ->
      parenthesis p <| etf a false + "+" + etf b false |> parenthesis p
    | Sub(a, b) ->
      parenthesis p <| etf a false + "-" + etf b true  |> parenthesis p
    | Mul(a, b) ->
      parenthesis p <| etf a true  + "*" + etf b true  |> parenthesis p
    | Div(a, b) ->
      parenthesis p <| etf a true  + "/" + etf b true  |> parenthesis p

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

Listing 16: konventering fra expression til infix notation

Funktionen `tree`, som foretager konverteringen fra infix notation til et udtrykstræ, er baseret på algoritmen beskrevet i [1][9]. Først konverteres en udtryksstreng til en liste af tokens. Disse

[8] *Inverse function.*
[9] *Convert Infix expression to Postfix expression.*

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[10]. Typen for disse tokens kan ses i Listing 17. 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 5.1.

```
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 17: Konvertering fra infix til udtrykstræ

3.4 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.

3.5 differentiering af udtryk

3.6 PBT af udtryk

3.6.1 Homomorfisme af evaluering

3.6.2 Invers morphism mellem infix og prefix

4 Vektorer og Matricer

Vi vil nu betragte et modul for vektorer og matricer. Da en Vector også kan betragtes som en matrix, vil vi herfra når der omtalles begge kun referare til en matrix. For at kunne håndtere matricer på en systematik måde begynder vi med at defienre en type for major order.

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.

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

Listing 18: Typen for order

[10] *Operator Precedence and Associativity in C.*

Typen Order (se Listing 18), anvendes til at angive, om en matrix er i rækkefølge (row-major) eller kolonnefølge (column-major)[11]. En vektor, der er lagret i kolonnefølge, kan betragtes som den transponeret rækkefølge vektor. Vi kan derfor nu definere en type for matricer, ved hjælp af en type for vektore (Listing 19).

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

Listing 19: 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 20).

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

Listing 20: Typen for dimissionen

.-

4.1 Matrix operationer

Der vil i denne sektion beskrives en række funktioner som er nødvendige før vi kan betragte nogle rekursive algoritmer som kan anvendes på en matrice. 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 21). Funktionen laver et kald til matrixValidMajor genere en fejl hvis ikke alle vektorer og matrien har samme lagringsordning. matrixVectorLength finder længden af en vektor 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 21: Funktion til at finde dimissionen af en matrix

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.

4.1.1 Matematiske operationer

I denne section bør bemærkes flere Listings ikke anvender fejlhåndtering, dette er udelukket for læsbarhedens skyld. De anvendte funktioner i mondulet har passende dimensions tjek på matricerne se evt. appendiks 5.2, der sikre at operationerne altid er lovlige.

[11] *Row- and column-major order*.

4.1.1.1 Skalering af en matrix

Vi begynder med at betragte en funktion til at skalere en matrix (se Listing 22).

```
// 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 22: 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[12].

```
// 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, _)) =
    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 23: Funktion til at addere matricer og substraktion af vektorer

4.1.1.2 Addition af matricer

Addition af vektorer kommer ned til at fortage additionen elementvis i Listing 23, ved brug af List funktionen map2. Vi kan bruge addVector til at definere, matrix addition og subtraktion af vektorer sidst bruges den også til at summere rækkerne i en matrice (sumRows) som vil blive brugt i implementeringen af Gram-Schmidt processen i sektion 4.4, funktionen bliver yderligere beskrevet i defination 1.

[12] *Python - List Comprehension.*

Defination 1 (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 $i = 1, 2, \ldots, m$.

4.1.1.3 Matrix multiplikation

Defination 2 (Matrix-Vektor Multiplikation).
Lad A være matrix med m rækker og n søjler, hvor

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

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

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

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

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

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

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

Ved brug af definition 1 for *sumRows* og definition 2 ses det at

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

□

Vi kan dermed anvende Sætning 1 til at definere en funktion matrixMulVector for matrix-vektor multiplikation (se Listing 24). Som først skalere søjlerne i matricen med de tilsvarende element i vektoren, og derefter summe rækkerne i matricen.

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

Listing 24: Funktion til matrix-vektor multiplikation

Defination 3 (Matrix multiplikation[13])**.**
Lad $A \in \mathbb{F}^{m\times n}$ *og* $B \in \mathbb{F}^{n\times \ell}$*. Antag at søjlerne i* B *er givet ved* $b_1, \ldots, b_\ell \in \mathbb{F}^n$*, dermed*

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

Så defineres matrice produktet som

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

Udfra defination 3, ses det at funktion matrixProduct i Listing 25 til at multiplicere to matricer, bliver derfor at lave matrix-vektor multiplikation på hver søjle i matricen. Da matrixMulVector returnere en matrix, skal der bruges en funktion til at konvertere matricen til en vektor, hvilket matrixToVector gør.

[13] *Mathematics 1a*.

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

Listing 25: Funktion til at multiplicere matricer

4.1.1.4 Projektion af en vektor

Som beskrevet i afsnit 2.3 'Projections onto a line' i 'Mathematics 1b'[14], kan projektionen af en vektor defineres som følgende, hvor $Y = \text{span}\{y\}$.

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

Med det standard indreprodukt

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

Den første funktion vi skal bruge er derfor en funktionen til at konjugere en vektor conjugateVector, det gøres ved at konjugere elementerne i vektoren. Udover dette defineres en funktion til at multiplikere to vektorer element vis vectorMulElementWise.

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

// conjugateVector : Vector -> Vector
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 26: Funktioner til projektere 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.

Sidst kan funktionen proj skrives direkte som den er defineres i ligning 3.

[14] *Mathematics 1b - Functions of several variables*.

4.2 PBT af matrix operationer

Det er nu muligt at opstille nogle PBT af der sikre at matricerne overholder matematiske egenskaber i sætning 3. Først defineres en generator for matricer, som generere matricer med tilfældige tal fra vores talmængde 7.

```
// 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 27: Generatorene anvendt til PBT af matrix operationer i

Egenskab 3 (Vektor Aksiomer[15])**.**
Lad $c, d \in \mathbb{F}$ og $v_i \in \mathbb{F}^n$ for $i = 1 \dots m$ så gælder:

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. $c \cdot (v_1 + \cdots + v_m) = c \cdot v_1 + \cdots + c \cdot v_m$

Vi kan dermed nu lave definere egenskaberne fra sætning 3 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
```

[15] theorem 7.2 i mat 1 noterne

```
let vectorAssCom m (c:Number) =
    c * (sumRows m) = sumRows (c * m)
```

Listing 28: Egenskaberne fra sætning 3 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 29: Outputtet fra PBT af vektor Listing 28

4.3 Række-echelon form

4.4 Gram-Schmidt

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

Før vi implementerer Gram-Schmidt processen, er vi dog begrænset af vores Number type 7, idet $x \in \{\text{Number}\} \not\Longrightarrow \sqrt{x} \in \{\text{Number}\}$. Derfor vil vi ikke normalisere vektorerne, hvilket medfører, at vi kun vil finde en ortogonal basis, fremfor en ortonormal basis.

```
// 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 30: Dannelsen af en ortogonal basis, ved hjælp af Gram-Schmidt processen

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 vektores 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.

4.5 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 31: Generatorene anvendt til PBT af Gram-Schmidt

Listing 31 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 32 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 32: 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 33: PBT af Gram-Schmidt processen

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

Listing 34: Output fra PBT af Gram-Schmidt processen

Outputtet fra PBT af Gram-Schmidt processen kan ses i Listing 34. Som sædvanligt indikere testen kun korrekthed, men ikke garanteret korrekthed.

5 Appendiks

5.1 TreeGenerator.fs

5.2 Matrix.fs

Litteratur

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

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

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

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

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

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

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

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

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

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

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

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

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

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