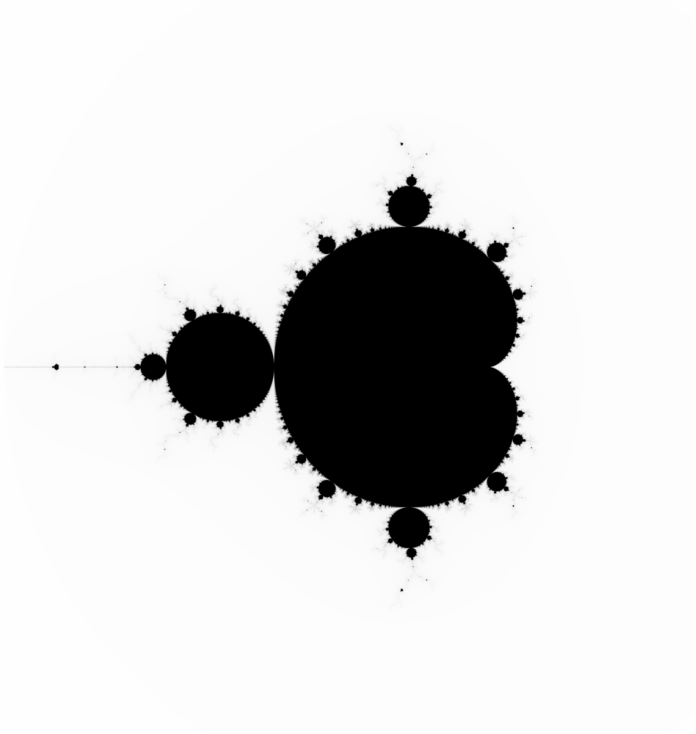
Visualisering af Mandelbrotmængden

[redacted]

[redacted]

Vejledere: [redacted]
Fag: Matematik Programmering B,
Sidetal: ca. 19,8 normalsider,
Skole [redacted]



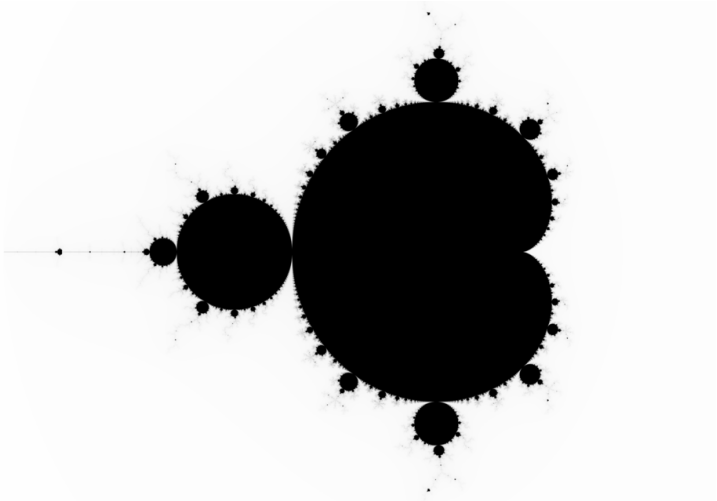
Resumé

I dette projekt gennemgås det matematiske grundlag for fremstillingen af billeder af Mandelbrotfraktalen. Forskellige metoder til generation af disse billeder gennemgås og deres tidsforbrug sammenlignes ved at inspicere forbrugt CPU-tid og realtid. På baggrund heraf konkluderes at særligt brugen af flere tråde til generering er optimal, mens andre metoder kræver vurderinger af hvor relevante de er.

Indhold

1 Indledning

Fraktaler er geometriske former, der gentager sig selv for evigt [19] på smukke og tankevækkende måder. De blev introduceret af Benoit B. Mandelbrot i 1980'erne baseret på Felix Hausdorffs arbejde i begyndelsen af 1900-tallet [5, 18]. Figur 1 viser Mandelbrotfraktalen, en af de mest kendte og fascinerende fraktaler. Mandelbrot opdagede den tidligt i sin karriere, men den ikoniske silhuet blev for alvor kendt med udbredelsen af computeren, der gjorde det muligt at udforske og visualisere fraktalen langt hurtigere end før [19].



Figur 1: Mandelbrotmængden

Jeg har udviklet en softwarepakke til undersøgelse af Mandelbrotfraktalen. Inkluderet heri er programmer til bl.a. at genere zoom-sekvenser og udforske fraktalen ved at zoome og panorere rundt. Dette projekt omhandler effektiviseringen af koden ansvarlig for at fremstille Mandelbrotfraktalen. Målet er at kunne generere Mandelbrotfraktalen så hurtigt at disse programmer kan fungere i realtid, samtidig med at koden forbliver læselig og det matematiske grundlag tydeligt.

Indledningsvist forklares om softwarepakkens struktur i afsnit 2. Dette hjælper med at kontekstualisere problemstillingen. Derpå introducerer afsnit 3 Mandelbrotfraktalen og den nødvendige matematiske baggrundsviden. I afsnit 4 forklares hvordan billeder af fraktalen frembringes på en computer. Afsnit 5 til 8 gennemgår derpå én visualiseringsmetode hver, alle sammen forskellige variationer af escape-time-metoden. I afsnit 9 opsummeres delkonklusionerne og metoderne sammenlignes med henblik på at vurdere fordele og ulemper ved inklusion af forskellige metoder i programmetpakken.

2 Softwarepakkens arkitektur

For at give koden i de følgende afsnit kontekst er det værd at give et generelt overblik over hvordan softwaren er struktureret.

Helt overordnet er hvert program delt i to dele: en bagende og en grænseflade. Grænsefladen er ansvarlig for at fortolke brugerinteraktioner og fremvise fraktalen, og bagenden er ansvarlig for at bestemme, hvilke pixels i det resulterende billede er medlemmer af Mandelbrotmængden. Figur 2 illustrerer dette forhold grafisk. Denne struktur er oplagt, fordi programmerne har det til fælles, at de viser et billede af Mandelbrotfraktalen. Forskellen på dem er hvordan de lader brugeren interagerer med dette billede. Eksempelvis lader `interactive` brugeren panorere rundt og zoome ind, mens `zoom` afspiller en film der gradvist zoomer ind på et bestemt punkt.

Figur 2: Afgrænsning mellem komponenter

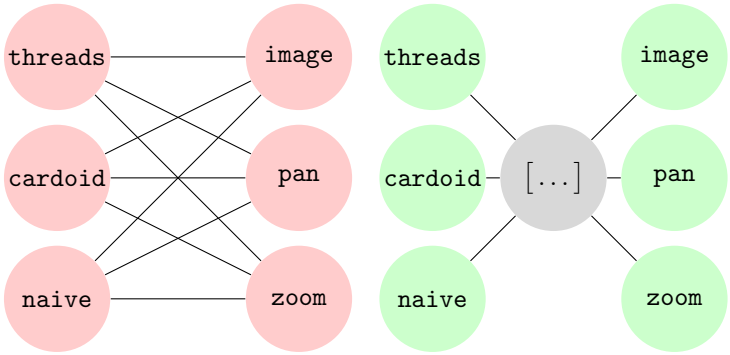
Når grænsefladen ønsker at rendere et billede sendes følgende parametre til bagenden.

1. Et område af det komplekse talplan, som skal undersøges, re_{min}, re_{maks}, im_{min} og im_{maks}.
2. En størrelse på det resulterende billede: s_x og s_y.

Bagenden er så ansvarlig for at inspicere hver pixel og bestemme om det tilsvarende komplekse tal er med i Mandelbrotmængden. Denne information udveksles i form af en matrix med én værdi for hver pixel på billedet, som brugeren ser. Den præcise betydning af værdierne i denne matrix gennemgås i afsnit 4. Matrixen udgør dermed en *smal talje* [3]. Et typisk kendetegn ved smalle taljer er at de tillader $N \cdot M$ forskellige kombinationer med blot $M + N$ moduler. I dette tilfælde er der tale om N grænseflader, M bagender og $N \cdot M$ programmer. Andre eksempler på smalle taljer inkluderer bl.a. IP-pakker, JSON, og tekst. Dette koncept illustreres på figur 3.

Med undtagelse af afsnit 4, der redegør for hvordan koden i grænsefladerne omdanner matrixen til æstetiske og/eller informative billeder, beskæftiger dette projekt sig kun med implementeringen af forskellige bagender. Afsnit 5 til 8 udforsker hver én bagende.

Programmet med bagende `A` og grænsefalde `B` benævnes `A-B`, f.eks. `threads-zoom`.



(a) Uden smal talje (b) Med smal talje

Figur 3: Softwaremoduler i pakken med/uden matrix som smal talje

3 Mandelbrotfraktalen

Mandelbrotfraktalen fremstilles ved at undersøge den komplekse rekursionsligning

$$z_{n+1} = z_n^2 + c \tag{1}$$

I afsnit 3.1 redegøres for komplekse tal og i afsnit 3.2 redegøres for rekursionsligninger. Med denne baggrundsviden forklares ligning (1) i afsnit 3.3.

3.1 Komplekse tal

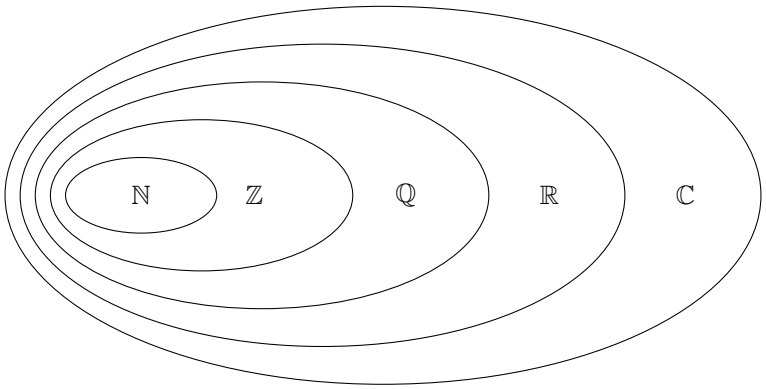
Alle tal er ikke skabt lige. Inden for matematikken skelner man mellem forskellige talmængder [14]. Talmængderne kan indordnes i et hierarki sådan at

$$\mathbb{N} \subsetneq \mathbb{Z} \subsetneq \mathbb{Q} \subsetneq \mathbb{R} \tag{2}$$

Nederst i hierarkiet er de naturlige tal $0, 1, 2, 3, \ldots$ benævnt med $\mathbb{N}$. Inddrages også de negative tal $-1, -2, -3, \ldots$ fås heltalene, benævnt med $\mathbb{Z}$. Dernæst kommer de rationelle tal, $\mathbb{Q}$, som kan beskrives ved $\frac{p}{q}$ hvor $p, q \in \mathbb{Z}$. Øverst i hierarkiet findes de reelle tal, $\mathbb{R}$. De inddrager også irrationelle tal som π, der ikke kan beskrives med en brøk af to heltal, fordi de indeholder decimaler, der fortsætter for evigt uden at gentage sig selv. Det vigtige princip i talhierarkiet er, at alle naturlige tal også er heltal. Ligeledes er alle heltal rationelle tal og alle rationelle tal er reelle tal. Denne relation beskrives både i ligning (2) og figur 4. Formålet med dette afsnit er at udvide talbegrebet med talmængden $\mathbb{C} \supsetneq \mathbb{R}$, også kendt som de komplekse tal.

3.1.1 Imaginære tal

På vejen mod de komplekse tal er det nødvendigt først at introducere de imaginære tal. De imaginære tal tager udgangspunkt i objektet $\sqrt{-1}$ [8, s. 131–132].



Figur 4: Tallenes hierarki illustreret som Venn-diagram

Inden for de reelle tal er denne operation selvfølgelig ulovlig, hvorfor denne disse tal kaldes "imaginære". Dette kaldes *den imaginære enhed* og benævnes som oftest med i. Det gælder dermed at

$$i = \sqrt{-1} \Leftrightarrow i^2 = -1 \tag{3}$$

Et imaginært tal konstrueres ved produktet af et reelt tal og den imaginære enhed, f.eks. $42i$ eller $123i$. Mængden af imaginære tal benævnes $\mathbb{I}$. Bemærk dog at $\mathbb{I}$ ikke indgår direkte på figur 4. Det skyldes at alle reelle tal *ikke* er imaginære tal, med andre ord er $\mathbb{R} \nsubseteq \mathbb{I}$.

3.1.2 Komplekse tal på rektangulær form

Hvad der i stedet indgår på figur 4 er de komplekse tal, $\mathbb{C}$. Et komplekst tal dannes ved summen af et reelt tal og et imaginært tal. De skrives helt generelt på formen $a + ib$, hvor $a \in \mathbb{R}$ kaldes den reelle del og $b \in \mathbb{R}$, ikke ib, kaldes den imaginære del. Formen $a+ib$ kaldes *rektangulær form*. Ethvert reelt tal $r \in \mathbb{R}$ kan beskrives som et imaginært tal hvor $a = r$ og $b = 0$. Med andre ord er alle andre talmængder delmængder af de komplekse tal. De komplekse tal udvider derfor talhierarkiet fra de reelle tal til de komplekse tal. Mange matematikere mener desuden, at $\mathbb{C}$ er den størst mulige talmængde; at de komplekse tal fuldender talhierarkiet [20].

De sædvanlige matematiske operationer fungerer også på komplekse tal. For at lægge to komplekse tal sammen lægges deres reelle og imaginære dele sammen.

$$\begin{aligned} Z_1 + Z_2 &= (a_1 + ib_1) + (a_2 + ib_2) \\ &= a_1 + ib_1 + a_2 + ib_2 \\ &= a_1 + a_2 + ib_1 + ib_2 \\ &= (a_1 + a_2) + i(b_1 + b_2) \end{aligned}$$

Produktet af to komplekse tal sker ved en lignende forlængelse af de sædvanlige regneregler. Vi husker her på ligning (3).

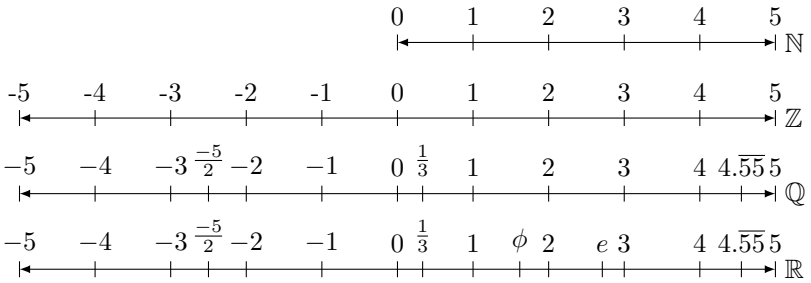
$$\begin{aligned} Z_1 \cdot Z_2 &= (a_1 + ib_1) \cdot (a_2 + ib_2) \\ &= a_1a_2 + ia_1b_2 + ia_2b_1 + i^2b_1b_2 \\ &= a_1a_2 + ia_1b_2 + ia_2b_1 - b_1b_2 \\ &= (a_1a_2 - b_1b_2) + i(a_1b_2 + a_2b_1) \end{aligned}$$

På samme måde fungerer de andre grundlæggende regneoperationer som minus, division, eksponentiering, osv.

3.1.3 Det komplekse talplan

Indtil introduktionen af $\mathbb{C}$ har udvidelsen af talbegrebet betydet en finere inddeling af talrækken som vist på figur 5*. De komplekse tal bryder med dette mønster. Hvor på talrækken placeres $3 + i4$? Hvad med $\pi + i6$? For at inkludere alle komplekse tal udvides talrækken nu til tal*planet*. Den reelle talrække lægges ud på førsteaksen og den imaginære talrække lægges ud på andenaksen. Et komplekst tal, z, ses indtegnet i et sådan koordinatsystem på figur 6.

Denne egenskab ved komplekse tal gør dem ofte fordelagtige at arbejde med, fordi de kan beskrive et punkt med et enkelt tal $x + iy$ frem for et koordinatsæt (x, y). Mandelbrotfraktalen tegnes også i det komplekse talplan.

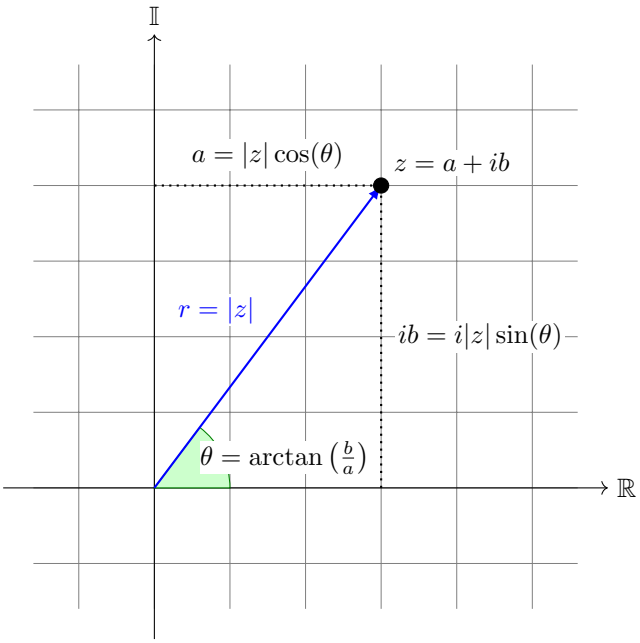


Figur 5: Talmængdernes talrækker.

At se z indtegnet i et koordinatsystem på denne måde giver anledning til at betragte z som en vektor frem for på rektangulær form. I så fald kan z skrives som $\binom{a}{b}$. Den blå pil på figur 6 repræsenterer z på vektorform. Denne observation gør det nemt oplagt at definere modulo/størrelsen af et komplekst tal analogt til længden af en vektor.

$$|z| = \sqrt{a^2 + b^2} \tag{4}$$

*Da det ved introduktionen af de rationelle tal bliver umuligt at afbillede alle tal i et interval fremvises her kun et par eksempler.



Figur 6: Det komplekse talplan

Denne operation kaldes også for tallet zs *modulus*. En anden måde at anskue denne definition på er at for $x \in \mathbb{R}$ er $|x|$ afstanden fra 0 til x langs med tallinjen. På samme måde er $|z|$ afstanden fra 0, der udgør origo i det komplekse talplan, til punktet z.

3.1.4 Komplekse tal på polær form

I afsnit 3.1.2 så vi, at komplekse tal kunne skrives på rektangulær form (dvs. $a+ib$) og i afsnit 3.1.3 så vi den skrevet på vektorform (dvs. $\binom{a}{b}$). I dette afsnit introduceres endnu en måde at anskue et komplekst tal: *polær form*.

Afstanden fra origo til det komplekse tal z kaldes dets *modulo* og skrives som $r = |z|$. Denne størrelse kender vi allerede fra afsnit 3.1.3. Nu introduceres hertil en vinkel, kaldet zs *argument*. Argumentet skrives som $\arg(z)$ eller blot θ og er defineret ved $\theta = \arctan\left(\frac{b}{a}\right)$. Et komplekst tal på polær form beskrives ud fra r og θ.

$$z = r\sin(\theta) + ir\sin(\theta) \tag{5}$$

Denne form ses også repræsenteret på figur 6.

3.2 Rekursionsligninger og talfølger

Nogle gange ønsker man i matematikken at beskrive *iteration*; en diskret proces der gentages. Hertil benyttes rekursionsligninger og talrækker. Da Mandelbrot-

mængden beskrives ud fra en sådan iterativ process, gives her et kort overblik over emnet.

En talfølge er en potentielt uendelig række af tal, Et eksempel på en talfølge er $\{1, 4, 16, \ldots\}$. I denne talfølge er hvert tal fire gange større end det forrige tal. Elementerne i talfølgen benævnes y_0, y_1, y_2, y_3, osv. En rekursionsligning er ganske enkelt en måde at beskrive en talfølge på. For at beskrive førnævnte talfølge, ville man bruge ligningen

$$y_n = y_{n-1} \cdot 4 \text{ hvor } n = 1, 2, 3, \ldots \tag{6}$$

eller tilsvarende*

$$y_{n+1} = y_n \cdot 4 \text{ hvor } n = 1, 2, 3, \ldots$$

Der er dog det lille problem, at ligning (6) faktisk beskriver uendeligt mange talfølger. Den kunne f.eks. også beskrive talfølgen $\{\ldots, 6.25, 25, 100, 400, \ldots\}$. Derfor angiver man ofte en begyndelsesbetingelse, som "stopper" rekursionen og samtidig fastslår et begyndelsespunkt, y_0, for talfølgen. Ligning (7) er en passende begyndelsesbetingelse for ligning (6).

$$y_0 = 1 \tag{7}$$

Det er nu muligt at beregne så mange værdier i talfølgen, som man har tid og tålmodighed til.

$$\begin{aligned} y_0 &= 1 \\ y_1 &= y_0 \cdot 4 \ = 1 \cdot 4 = 4 \\ y_2 &= y_1 \cdot 4 \ = 4 \cdot 4 = 16 \\ y_3 &= y_2 \cdot 4 = 16 \cdot 4 = 64 \\ &\vdots \end{aligned}$$

En talfølge siges at være *konvergent*, hvis den "går mod" et bestemt tal. Mere formelt siges en talfølge $\{y_0, y_1, y_2, \ldots\}$ at være konvergent, hvis der findes et tal y_* som y_n kommer vilkårligt tæt på, hvis blot n er højt nok. Helt konkret siges det, at der for enhver $\epsilon > 0$ findes et naturligt tal N sådan at hvis $n \geq N$ er $|y_* - y_n| < \epsilon$. I så fald siges y_n at konvergere mod y_*. Et eksempel på en konvergent talfølge er den beskrevet af ligning (8).

$$\begin{aligned} y_n &= \frac{y_{n-1}}{y_{n-1} + 1} \\ y_0 &= 1 \end{aligned} \tag{8}$$

Som $n \to \infty$ vil summen med 1 bliver mere og mere ubetydeligt, sådan at $\frac{y_{n-1}}{y_{n-1}+1} \to \frac{y_{n-1}}{y_{n-1}} = 1$. Med andre ord går talfølgen y_n altså mod $y_* = 1$. En

*Fremadrettet vil det for korthedens skyld antages, at $n = 1, 2, 3, \ldots$.

talfølge siges at konvergere på ∞, hvis der for ethvert tal x findes en n sådan at $x_n > x$ og vice versa for $-\infty$.

En førsteordens rekursionsligning, hvor y_n kun afhænger y_{n-1}, kaldes somme tider for et *itereret* system. At bevæge sig fra y_n til y_{n+1} kaldes at *iterere*. Talføljen, som rekursionsligningen beskriver kaldes så systemets *bane*, og værdien y_0 kaldes *begyndelsesværdien* eller *begyndelsespunktet*.

3.3 Mandelbrotmængden

Vi er nu klar til at vende tilbage til ligning (1). Vi kan nu se, at der er tale om en kompleks rekursionsligning.

$$z_n = z_{n-1}^2 + c \text{ hvor } c, z \in \mathbb{C} \tag{9}$$

Mandelbrotmængden, M, defineres nu som alle de punkter på det komplekse talplan, c, hvor talfølgen z_n ikke går mod uendelig, når startværdien er $z_0 = 0$ og n går mod uendelig [8, s. 138–147].

Det er i afbildningen af Mandelbrotmængden på det komplekse talplan at vi finder Mandelbrotfraktalen. Afsnit 4 går mere i dybden, men som udgangspunk kan det siges, at alle punkter der er i M farves sorte for at danne billedet set på figur 1.

Ligning (9) skrives også nogle gange med funktionsnotation som på ligning (10).

$$f_c(z) = z^2 + c \text{ hvor } c, z \in \mathbb{C} \tag{10}$$

Her angives antallet af iterationer som $f_c^n(z)$, så $z_n = f_c^n(z_0)$.

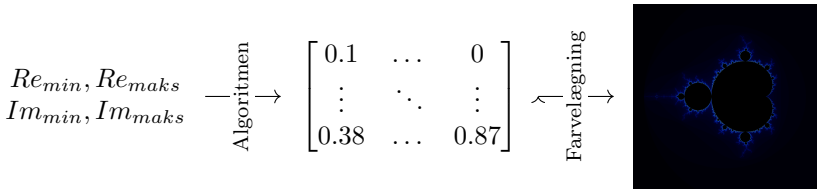
4 Escape-time visualisering

Afsnit 3.3 forklarede hvilke tal, der var medlemmer i Mandelbrotmængden samt at afbildningen af denne på det komplekse talplan danner Mandelbrotfraktalen. I dette afsnit vil vi i stedet gå i dybden med *hvordan* en sådan afbildning genereres på en computer. Her beskrives hvordan computeren går fra et område af det komplekse plan til det færdige billede, en opgave der strækker sig fra bagende til grænseflade.

Samtlige bagender benytter *escape-time visualisering**. Hvor hver pixel i billedet bestemmes det hvorvidt det tilsvarende komplekse tal, c, er et medlem af Mandelbrotmængden eller, hvis det ikke er, hvor hurtigt $f_c(z)$ går mod uendelig. Resultatet heraf er en *pixelværdi*; et tal i intervallet $\mathbb{R} \cup [0; 1]$ som angiver hvor hurtigt $f_c(z) \to \infty$ sådan at $0 \Rightarrow c \notin M$ og $1 \Rightarrow c \in M$. Pixelværdierne gemmes i en matrix af samme dimensioner som det færdige billede.

*Der er ingen særlig grund til dette, udover at det er bekvemt. De kunne lige så godt benytte sig af *distance estimering* som beskrevet i [30]. Desuden er der ingen grund til, at de alle skulle benytte den *samme* visualiseringsmetode.

Dernæst omdannes matrixen af pixelværdier til det endelige billede. Dette sker ved en funktion der oversætter pixelværdier til RGB-farver. Denne funktion er defineret som en del af grænsefladen, men den gennemgås alligevel her i håb om at det giver læseren en bedre forståelse af hele processen. Figur 7 illustrerer processen fra parametre til færdigt billede.



Figur 7: Afgrænsning mellem komponenter

Afsnit 4.1 gennemgår det teoretiske grundlag for første del af processen og afsnit 4.2 gennemgår farvelægningsprocessen. I afsnit 5.1 ses idéerne fra afsnit 4.1 omsat til kode.

4.1 Bestemmelse af medlemskab

Mandelbrotmængden blev i afsnit 3.3 defineret som alle de punkter for hvilke ligning (1) har $|z_n| \not\to \infty$ når $n \to \infty$. Den definition er problematisk, fordi uendeligheder er svære at simulere på computere, der kun har endelige ressourcer. Derfor bestemmes i stedet en tilnærmelse af Mandelbrotmængden baseret på to antagelser, én for hver uendelighed i definitionen på Mandelbrotmængden.

1. Hvis $|z|$ overstiger 2, vil funktionen gå mod uendelig.
2. Hvis funktionen ikke overstiger 2 efter n_{maks} iterationer, vil den aldrig gå mod uendelig.

4.1.1 Indskrænkning af kriterier

Vi beskæftiger os først med antagelse 1. Der er et matematisk grundlag for denne antagelse, som vi vil se med et lille bevis. Antag først at $|z| > \text{maks}\{2, |c|\}$, dvs. $|z| > 2$ og $|z| > c$. Med $|z| > |c|$ og den omvendte trekantsulighed, $|a + b| \geq ||a| - |b||$, ser vi nu, at

$$\begin{aligned}
|f(z)| &= |z^2 + c| && \text{Trekantsuligheden} \\
&\geq ||z^2| - |c|| && |z|^2 > |c| \\
&\geq |z^2| - |c| && \text{uligheden holder da } |z| > |c| \\
&> |z^2| - |z| && |a^n| = |a|^n \\
&> |z|^2 - |z| && \text{Faktorisér } |z| \\
&> |z|(|z| - 1)
\end{aligned}$$

Nu sættes $|z| = 2+\epsilon$, hvor $\epsilon > 0$. Dette er lovligt, fordi vi indledningsvist antog, at $|z| > 2$.

$$\begin{aligned} |f(z)| &> |z|(|z|-1) && \big) |z| = 2+\epsilon \\ &> |z|(2+\epsilon-1) && \big) \textit{Reducér} \\ &> |z|(1+\epsilon) \end{aligned}$$

Hvis vi itererer én gang får vi nedenstående.

$$\begin{aligned} |f^2(z)| &= |f(f(z))| && \big) \textit{Ovenstående ulighed} \\ &> |f(z)|(1+\epsilon) && \big) \textit{Igen} \\ &> (|z|(1+\epsilon))(1+\epsilon) && \big) \textit{Reducér} \\ &> |z|(1+\epsilon)^2 \end{aligned}$$

Herfra kan vi ekstrapolere, at følgende generelt må gælde.

$$|f^n(z)| > |z|(1+\epsilon)^n$$

Eller tilsvarende på talfølgenotation.

$$|z_{n+k}| > |z_n|(1+\epsilon)^k \tag{11}$$

Eftersom $(1+\epsilon) > 1$ følger det af ovenstående ulighed*, at $(1+\epsilon)^k \to \infty$ og $|z_{n+k}| \to \infty$ når $k \to \infty$. Med andre ord vil $|z_n|$ gå mod uendeligt under de forudsætninger vi satte i starten af beviset.

Der er dog ét problem: de forudsætninger, vi satte, er lidt for brede. Vi antog at $|z| > \text{maks}\{2, |c|\}$, men vi ønskede kun at vise, at $|z_n| \to \infty$ hvis blot $|z| > 2$. For at slippe af med betingelsen $|z| > |c|$ undersøger vi nu de to muligheder for $|c|$. Hvis $|c| \leq 2$ gælder $|z| > \text{maks}\{2, |c|\}$ stadig, og den ovenstående argumentation står ved. For $|c| > 2$ må vi huske på, at

$$\begin{aligned} z_0 &= 0 \\ z_1 &= z_0^2 + c = c \\ z_2 &= z_1^2 + c = c(c+1) \end{aligned}$$

Størrelsen på z_2 er således givet ved $|z_2| = |c| \cdot |c+1| > |c|$, og argumentationen ovenfor gælder igen, denne gang blot med udgangspunkt i z_2. Uanset værdien af $|c|$ vil argumentationen i dette bevis således stadig gælde.

Således har vi vist, at talrækken z_n vil gå mod uendelig, hvis en værdi nogenside får en afstand fra origo større end 2.

*Der er et potentielt problem i at værdien af ϵ afhænger af $|z|$, men dette kan hurtigt afværges ved at indse at $|z_n| > \ldots > |z_1| > |z_0| \Leftrightarrow \epsilon_n > \ldots > \epsilon_1 > \epsilon_0$. Hvis ϵ_0 derpå indsættes i ligning (11) vil højresiden af uligheden kun mindskes.

4.1.2 Iterationsgrænsen

Vi kigger nu på antagelse 2. Modsat antagelse 1 er dette ikke en matematisk solid antagelse. Per definitionen på Mandelbrotmængden er det nødvendigt at lade $n \to \infty$ for at tjekke om $z_n \to \infty$, hvilket som nævnt før ikke er muligt på en fysisk computer. I stedet lades n gå mod en øvre grænse n_{maks}. Hvis ikke $|z|$ overstiger 2 inden da, antages det, at $|z|$ ikke vil gå mod uendelig.

Det er ikke trivielt at sætte iterationsgrænsen n_{maks}. Hvis denne grænse er for lav vil værdier for c hvis talrækker holder sig under 2 indtil n_{maks} men senere går mod uendelig fejlagtigt regnes som ikke med i mængden. Den visuelle effekt heraf er at hvirvlerne omkring mængdens grænse forsvinder ved for lave værdier af n_{maks}. Jo højere værdien af n_{maks} er, jo mere korrekt bliver estimatet, fordi n_{maks} kommer tættere på ∞. Min egen empiri indikerer at $n_{maks} = 500$ er tilstrækkeligt højt til at give tilfredsstillende, æstetiske resultater.

Til slut normaliseres n. En pixelværdi skal ligge i intervallet $\mathbb{R} \cap [0; 1]$, men n ligger i intervallet $\mathbb{N} \cap [0; n_{maks}]$. Resultatet er derfor den normaliserede værdi $\frac{n}{n_{maks}}$.

4.2 Farvelægning

Processen med farvelægning oversætter en pixelværdi, t, til farve repræsenteret som en RGB-værdi. Bemærk at t angiver et reelt tal i intervallet $[0; 1]$ og *ikke* antallet af iterationer $n \in [0; n_{maks}]$.

$$f : [0, 1] \to \{0, 1, 2, \ldots, 255\}^3$$

Den simpleste metode til at oversætte t til en RGB farve er at farve alle medlemmer hvide og alle ikke-medlemmer sorte.

$$f(t) = \begin{cases} (255, 255, 255) & \text{hvis } t = 1 \\ (0, 0, 0) & \text{ellers} \end{cases}$$

Denne tilgang er nem at implementere, men information, om hvor hurtigt z_n går mod uendelig for den pågældende c-værdi, går tabt. En mulig løsning hertil er at oversætte alle farvekanaler til $\lfloor t \cdot 255 \rfloor$.

$$f(t) = \left(\lfloor t \cdot 255 \rfloor, \lfloor t \cdot 255 \rfloor, \lfloor t \cdot 255 \rfloor \right)$$

Det giver et monokromt resultat frem for et strengt sort/hvidt resultat. Det giver større indblik i de komplekse dynamikker omkring grænsen til mængden.

Med henblik på at gøre billederne mere visuelt interessante og æstetisk tilfredsstillende introduceres der farver til farvelægningsprocessen. Det er vigtigt at farvelægningsprocessen ikke introducerer visuelle bias, som eksempelvis ukontrolleret luminansvarians eller mangel på opfattelsesmæssig rækkefølge [1, 22].

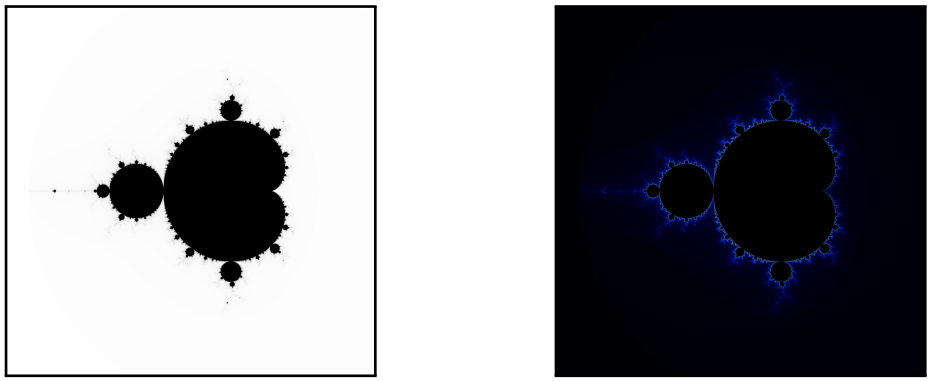
Derfor introduceres ét Bernsteinpolynomium per farvekanal [28, 26] som beskrevet på ligning (12).

$$\begin{aligned} f(t) &= \big(r(t), g(t), b(t)\big) \\ r(t) &= 9 \cdot (1-t) \cdot t^3 \\ g(t) &= 15 \cdot (1-t)^2 \cdot t^2 \\ b(t) &= 8.5 \cdot (1-t)^3 \cdot t \end{aligned} \tag{12}$$

Bernsteinpolynomier er særligt velegnede, da både deres definitionsmængde og værdimængde er $[0, 1]$ og de er tilstrækkeligt glatte [28, 29]. Funktionen på ligning (12) har dog ikke den rigtige værdimængde. Det er nødvendigt at skalere og afrunde funktionsværdierne for at bringe dem ind i det rette interval som vist på ligning (13).

$$\begin{aligned} f(t) &= \big(r(t), g(t), b(t)\big) \\ r(t) &= \lfloor 9 \cdot (1-t) \cdot t^3 \cdot 255 \rfloor \\ g(t) &= \left\lfloor 15 \cdot (1-t)^2 \cdot t^2 \cdot 255 \right\rfloor \\ b(t) &= \left\lfloor 8.5 \cdot (1-t)^3 \cdot t \cdot 255 \right\rfloor \end{aligned} \tag{13}$$

Værdimængden af funktionen på ligning (13) har det rette interval. De kan findes næsten ordret i den vedhæftede fil `ppm.c`. Figur 8 viser en sammenligning af de to farvelægningsmetoder diskuteret i dette afsnit. Mere vil der ikke blive sagt om farvelægningsprocessen.



(a) Monokrom farvelægning (b) Farverig farvelægning

Figur 8: Farverig farvelægning

5 Den første, naive implementering

Dette afsnit beskriver den simpleste og mest troværdige oversættelse af algoritmen beskrevet i afsnit 4. Denne metode danner basislinjen, som de andre

implementeringer måles imod, hvorfor den gennemgås med særligt høj detaljegrad.

5.1 Implementering

Pseudokode for algoritmen beskrevet i afsnit 4 ses i algoritme 1. Algoritme 1 kan anses for en formalisering af metoden forklaret i afsnit 4. På afsnit 7.2 til 7.2 omregnes pixel-koorinaten til det tilsvarende komplekst tal. De to antagelser, som blev gennemgået i afsnit 4.1.1 og 4.1.2, ses afspejlet i løkken på algoritme 1. På algoritme 1 normaliseres iterationstallet n før det gemmes i pixelmatrixen.

Algorithm 1 Pseudokode for naiv escape-time visualisering

Require: Re_{min}, Re_{maks}, Im_{min}, Im_{maks},
Ensure: $Re_{min} > Re_{maks}$, $Im_{maks} > Im_{min}$,
  for each pixel (P_x, P_y) on the screen **do**
    $a \leftarrow Re_{min} + \frac{P_x}{s_x} \cdot (Re_{maks} - Re_{min})$
    $b \leftarrow Im_{min} + \frac{P_y}{s_y} \cdot (Im_{maks} - Im_{min})$
    $c \leftarrow a + ib$ ▷ Konvertér skærmkoordinater til $c \in \mathbb{C}$
    $z \leftarrow 0$
    $n \leftarrow 0$
    while $|z| < 2 \wedge n \leq n_{maks}$ **do**
      $z \leftarrow z^2 + c$
      $n \leftarrow n + 1$
    end while
    $pixels[P_y][P_x] \leftarrow \frac{n}{n_{maks}}$
  end for

C-koden afspejler på mange måder pseudokoden i algoritme 1. Alle bagender implementerer det samme interface. Dette findes i den vedhæftede fil `backend.h` samt på listing 1. Her ses det, at C-funktionen `mandelbrot` tager fornævnte formelle parameter med undtagelse af n_{maks}. Denne er udeladt, fordi det anskues som en implementeringsdetalje; koden er skrevet med forventning om at der i fremtiden vil være bagender, der ikke benytter escape-time visualisering.

```
1 void mandelbrot(unsigned width, unsigned height, double
  buffer[height][width],
2         double real_min, double real_max,
3         double imag_min, double imag_max);
```

Listing 1: Functionsdekleration for backends

Et uddrag fra implementeringen af algoritmen ses i listing 2. Den fulde kildekode til den naive bagende findes i den vedhæftede fil `naive.c`. De yderste to løkker er de samme som i pseudokoden på algoritme 1. Den eneste store forskel på pseudokoden og implementeringen i C på listing 2 er at løkken på algoritme 1 er faktoriseret ud i C-funktionen `check`. Dette afspejler de to forskellige aspekter af bagenden som udforskes i dette projekt. `mandelbrot` beskæftiger sig med *hvilke* tal

der tjekkes og `check` beskæftiger sig med *hvordan* tallene tjekkes. Sammen med den grovere inddeling i bagende og grænseflade introduceret i afsnit 2 betyder det at koden er meget modulær.

```
1  void mandelbrot(unsigned width, unsigned height, double
     buffer[height][width],
2          double real_min, double real_max,
3          double imag_min, double imag_max)
4  {
5      for (unsigned x = 0; x < width; ++x) {
6          for (unsigned y = 0; y < height; ++y) {
7              double real = real_min + ((double)x/(double)width)  *
     (real_max - real_min);
8              double imag = imag_min + ((double)y/(double)height) *
     (imag_max - imag_min);
9              complex double c = real + imag * I;
10
11             unsigned n_iter = check(c);
12             double value = (double)n_iter / (double)MAX_ITER;
13             buffer[y][x] = value;
14         }
15     }
16 }
```

Listing 2: `mandelbrot`-funktionen itererer hver enkelt pixel

Som sagt findes koden for at tjekke om $z_n \to \infty$ i `check`. Denne funktion ses på listing 3. `MAX_ITER` i koden svarer til variablen n_{maks} fra afsnit 4.1.2. På afsnit 5.1 ses antagelserne fra afsnit 4.1 igen afspejlet i koden, og den kendte formel introduceret i afsnit 3 findes på afsnit 5.1.

```
1  unsigned check(complex double c)
2  {
3      complex double z = 0;
4      unsigned n_iter = 0;
5
6      while (cabs(z) <= 2 && n_iter < MAX_ITER) {
7          z = z * z + c;
8          n_iter += 1;
9      }
10
11     return n_iter;
12 }
```

Listing 3: `check`-funktionen tjekker om $c \in M$

5.2 Om videnskabelig tidtagning

I evalueringen og sammenligningen af de forskellige metoder til frembringelsen af Mandelbrotfraktalen er det nødvendigt at have pålidelige målinger. Kun på baggrund af gentagelige og sammenlignelige målinger kan de forskellige bagenders effektivitet diskuteres.

Almindeligvis sker dette ved analyse af tidskompleksitet, men det er ikke muligt at analysere tidskompleksiteten af denne eller lignende visualiseringsmeto-

de på nogen konstruktiv måde, da tidsforbruget er tæt forbundet med hvilket område af det komplekse talplan der undersøges. Dette fremgår tydeligt, hvis man forsøger at analysere tidskompleksiteten af programmet præsenteret i afsnit 5.1. Tidsforbruget i `mandelbrot` er relativt forudsigeligt. Det havde blot været $O(s_x \cdot s_y)$, hvis arbejdet inde i løkken var konstant. Arbejdet i løkken er dog langt fra konstant. Definitionen på `check` er nemlig mere problematisk. I værste fald evalueres løkken på afsnit 5.1 n_{maks} gange og i bedste fald køres der én gang, hvis $|c| = |z_1| > |2|$. Et forsøg på at opskrive tidskompleksiteten med O-notation ville derfor nødvendigvis indeholde en definition på Mandelbrotmængden selv. Dermed har alle escape-time-baserede visualiseringsmetode den samme worst-case tidskompleksitet, nemlig $O(s_x \cdot s_y \cdot n_{maks})$.

I stedet for en analytisk tilgang benyttes derfor en empirisk tilgang til sammenligning og vurdering af tidsforbrug. En ny grænseflade defineres til at indsamle data om de forskellige bagenders tidsforbrug. Det sker ved at `benchmark` genererer det samme billede flere gange og måler gennemsnitlige tidsforbruget. Alle renderinger foretages med følgende indstillinger.

1. $n_{maks} = 500$
2. $Re_{min} = -2.2$, $Re_{maks} = 1.2$
3. $Im_{min} = -1.7$, $Im_{maks} = 1.7$
4. $s_x = 10,000$, $s_x = 20,000$

Denne metode introducerer mange fejlkilder, som resten af dette afsnit er dedikeret til at minimere. Omvendt kan det dog argumenteres for, at denne metode er mere relevante end analyse af tidskompleksitet, RAM-modellering [12, s. 9–11], og andre forenklede modeller, da den afspejler de virkelige ydelsesegenskaber på en computer*. Målingerne fåretages på en 2021 Macbook Pro med en M1 CPU.

Tidsforbruget måles ved funktionen `clock_gettime`, der defineres af POSIX [13, s. 679–682]. Hvis denne kaldes med `CLOCK_PROCESS_CPUTIME_ID` kan den bruges til at bestemme hvor meget CPU-tid den nuværende proces har brugt. Det fungerer som en form for variabelkontrol. For at opretholde illusionen af multitasking kører styresystemer nemlig mange programmer lidt ad gangen på skift, og er vigtigt at målingerne ikke påvirkes af hvad systemet ellers foretager sig. Derfor er CPU-tid for den specifikke proces en bedre måleenhed end klokkeslættet; den påvirkes ikke af hvilke andre programmer, der kører på computeren.

Det er passende at kigge udelukkende på CPU-tid, fordi visualiseringsmetoderne er *CPU-bundne* processer [11, s. 83], hvilket betyder, at den begrænsende faktor er CPUens hastighed. Det står i kontrast til *IO-bundet* arbejde, der bruger meget tid på at vente på f.eks. måleudstyr, databaser og netværksaktivitet.

*Eksempelvis antager RAM-modellen, at alle operationer tager lige lang tid, hvilket langt fra er tilfældet på en rigtig computer. At hente data fra den primære hukommelse kan tage flere hundrede gange mere tid, hvis den ikke findes i CPUens cache [7].

CPU-tid er dog ikke en ufejlbar måleenhed. Omend den ikke inkluderer tidsforbruget af andre processer på systemet, påvirkes den stadig indirekte af eksekveringen af andre programmer. På moderne computere afhænger CPUens ydeevne i stor del af effektiv cache-udnyttelse [7], og kontekstskift[†]har en negativ indvirkning på cache-udnyttelsen [21]. Hver gang der sker et kontekstskift vil cachen forurenes med data fra den nye proces, og den gamle proces' cache forsvinder.

Foruden at måle CPU-tid frem for klokkeslæt tages også andre metoder i brug for at mindske usikkerheden. Internetforbindelsen slås fra for at minimere effekten af andre processer på systemet. Manglende internetforbindelse stopper nemlig en række baggrundsprocesser fra at starte under testen. Computeren sluttes desuden til strøm for at strømbesparingsmekanismer ikke påvirker computerens ydeevne.

5.3 Evaluering

Escape-time visualisering er relativt nemt at implementere. Med undtagelse af de antagelser, der gøres i afsnit 4.1, følger implementeringen meget nært den tankegang som fører til definitionen af Mandelbrotmængden. Læseligheden hjælpes desuden af C-sprogets indbyggede understøttelse af komplekse tal, da det betyder at koden ligner den bagvedliggende matematik mere. Eksempelvis kan den kendte formel skrives som `z = z*z + c` frem for `z = complex_add(complex_square(z), c)`, som det ville kræves hvis man havde været nødsaget til at definere sin egen datatype.

Resultatet af at køre `naive-benchmark` er et median tidsforbrug på 49,971693 sekunder. Dette kommer til at udgøre basislinjen for de forskellige bagender; alle bagender måles mod denne langsomste algoritme.

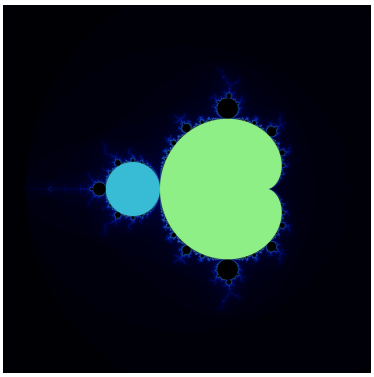
6 Kardioide-tjek

Dette afsnit bygger videre på den naive escape-time visualiseringsmetode introduceret i afsnit 4. Hertil introduceres en optimering, der i visse tilfælde undgår store mængder arbejde.

Mandelbrotmængden har to prominente udfyldte regioner. Den første er den store kardioide[*], og den anden er cirklen til højre for kardioiden med centrum i $-1 + i0$. Disse regioner fylder ofte en stor del af billedet, og da $|z_n| \not\to \infty$ for $c \in M$, spildes der derfor meget tid på at tjekke pixels der falder inden for disse to regioner. Løsningen på dette problem er at tjekke om c falder inden for kardioiden eller cirklen *før* den sædvanlige medlemskabsprøver. Herved spares mange iterationer af `check` arbejde for de pixels der falder inden for disse regioner.

[†]Et *kontekstskift* sker når styresystemet skifter fra at eksekvere én proces til at eksekvere en anden. Mere om det i afsnit 7.1.

[*]En *kardioide* får sit navn efter det græske ord for 'hjerte', men i grunden har formen mere til fældes med tværsnittet af et æble.



Figur 9: Mandelbrotmængden med kardioiden og cirklen hvor $p = 2$ fremhævet

6.1 Matematisk grundlag

Omend selve implementeringen af optimeringen kun drejer sig om en ændring på omtrent 5 linjer, kræves der en del matematisk forarbejde for at kunne bestemme forskrifter for de to regioner. I dette afsnit gennemgås den fornødne baggrundsviden samt to beviser: ét for hovedkardioiden og ét for cirklen.

6.1.1 p-kredsløb og fikspunkter

I afsnit 3.2 introduceredes talfølger og rekursionsligninger. For at gennemføre nedenstående beviser er det nødvendigt at introducere nogle flere begreber relateret til funktionsanalyse og rekursion.

Et *fikspunkt* er en værdi, som ikke ændrer sig når den itereres [2]. I denne sammenhæng betyder det at $z_{n+1} = z_n$ eller på funktionsnotation, $f(z_*) = z_*$. Bemærk at symbolet z_* bruges til at henvise til fikspunktet for talfølgen z_n.

Idéen om fikspunkter kan udvides til baner. Hvis værdierne i et itereret systems bane gentager sig siges det at have en *periodisk bane*. Det mindste antal iterationer der fører værdien tilbage til sig selv kaldes banes *periode*. Perioden er et positivt og endeligt heltal og benævnes med symbolet p. En bane med perioden p_0 siges at være en periode-p_0-bane. Med matematisk notation vil dette koncept skrives på ligning (14).

$$z_{n+p} = z_n \tag{14}$$

En talfølge som indeholder et kredsløb med $p > 1$ vil selvfølgelig aldrig konvergere på et bestemt tal.

Et fikspunkt kan således anskues som en periodisk bane hvor $p = 1$. I så fald vil $z_{n+1} = z_n$ og $z_n = z_*$ vil aldrig antage en anden værdi. Et fikspunkt skrives også nogle gange med funktionsnotation som på ligning (15).

$$f(z_*) = z_* \tag{15}$$

6.1.2 Stabile og ustabile fikspunkter

Et fikspunkt kan enten være *stabilt* eller *ustabilt*. Disse begreber dækker over hvordan systemet håndterer introduktionen af små ændringer.

Vi taget et eksempel. Antag at systemet z_n har fikspunktet z_* og er defineret ved

$$z_n = g(z_{n-1}) \tag{16}$$

Vi introducerer nu en lille ændring, ϵ, til fikspunktet og fortsætter iterationen herfra. Introduktionen af den lille forstyrrelse svarer således til at definere en ny talrække w_n.

$$w_0 = z_* + \epsilon \tag{17}$$

$$w_n = g(w_{n-1}) \tag{18}$$

Ligning (18) afspejler definitionen på z_n på ligning (16). Den eneste forskel på de to systemer er begyndelsesbetingelsen for w_n starter en smule forskudt fra z_ns fikspunkt. Hvis denne forandring formindskes over tid, dvs. $\lim_{n\to\infty} w_n \to z_*$, siges fikspunktet at være stabilt. Dette kan tænkes som at nærliggende værdier "suges ind" i z_*. Omvendt kaldes et fikspunkt for ustabilt, hvis nærliggende værdier skubbes væk. Et sidste tilfælde er de neutrale punkter, hvor hvilke ingen af disse regler gælder. Nogle nærliggende værdier vil suges ind mens andre vil skubbes væk.

Banachs fikspunktsætning [4, 9, s. 6, 50] giver en måde at bestemme hvorvidt et fikspunkt er stabilt, ustabilt eller neutralt. Hvis $|g'(z_*)|$ er større end, mindre end eller lig 1 er z_* et henholdsvis stabilt, ustabilt eller neutral fikspunkt.

Konceptet bag stabile og ustabile fikspunkter kan ligeledes udvides til periodiske baner med $p > 1$ [6, s. 55–57]. Et kredsløb kan suge nærliggende bane ind på samme måde som fikspunkter gør. Alle punkter i en periodisk bane er enten stabile, ustabile eller neutrale; det er ikke muligt at have en periodisk bane hvor visse punkter er stabile og andre er ustabile. Udvidelsen af Banachs fikspunktsætning til en periode-n-bane omhandler blot $|(g^n)'(z_*)|$ frem for $|g'(z_*)|$.

6.1.3 Medlemmer af hovedkardioiden

Vi har nu den fornødne baggrundsviden til at bestemme et udtryk for hovedkardioiden.

Den store kardioide består af alle de værdier for c hvis bane har et attraktivt fikspunkt. Først bestemmes et udtryk for fikspunkterne for f_c. Vi ved fra afsnit 6.1.2 at et fikspunkt for z_n er givet ved $f_c(z_*) = z_*$. Vi vil dog snart se at der er to mulige fikspunkter for enhver c, hvorfor det i stedet benævnes $z_\pm$ i

dette afsnit.

$$\begin{aligned} f_c(z_\pm) &= z_\pm && \big) \textit{ Udvid } f_c \\ z_\pm^2 + c &= z_\pm && \big) \textit{ Ordén} \\ z_\pm^2 - z_\pm + c &= 0 && \big) \textit{ Andengradsligning} \\ z_\pm &= \tfrac{1 \pm \sqrt{1-4c}}{2} \end{aligned} \tag{19}$$

Afsnit 6.1.3 er således definitionen på fikspunkterne for c. Desuden leder vi udelukkende efter *stabile* fikspunkter. Vi husker fra afsnit 6.1.2, at de stabile fikspunkter for f_c jævnfør Banachs fikspunktsætning opfylder $|f_c'(z)| < 1$. Ved at substituere $z_\pm$ som bestemt ud fra c ind i denne ulighed opnås et udtryk for de c-værdier, som har stabile fikspunkter.

$$\begin{aligned} |f_c'(z)| &< 1 && \big) \textit{ Udvid } f_c \\ |(z^2 + c)'| &< 1 && \big) \textit{ Differentiér mht. } z \\ |2z| &< 1 \\ |2z_\pm| &< 1 && \big) \textit{ Indsæt fikspunkter} \\ |1 \pm \sqrt{1-4c}| &< 1 \end{aligned} \tag{20}$$

Gemt i afsnit 6.1.3 er to uligheder.

$$\begin{aligned} |z_+| &= |1 + \sqrt{1-4c}| < 1 \\ |z_-| &= |1 - \sqrt{1-4c}| < 1 \end{aligned}$$

Vi vil nu se, at z_+ aldrig angiver et stabilt fikspunkt. For at gøre kvadratroden nemmere at arbejde med omskrives udtrykket $1 - 4c$ til polær form.

$$1 - 4c = r\left(\cos\left(\frac{\theta}{2}\right) + i\sin\left(\frac{\theta}{2}\right)\right) \tag{21}$$

Vi kan ekstrapolere fra afsnit 3.1.4 at kvadratroden af et komplekst tal på polær form bestemmes ved at tage kvadratroden af moduloen og halvere argumentet. Der er selvfølgelig to kvadratrødder. Derfor antager vi nu at $-\pi \leq \theta < \pi$, da det vil betyde, at den reelle af $\sqrt{1-4c}$ del bliver positiv* og vice versa for $-\sqrt{1-4c}$.

$$\sqrt{1-4c} = \sqrt{r}\left(\cos\left(\frac{\theta}{2}\right) + i\sin\left(\frac{\theta}{2}\right)\right)$$

*Et kig på enhedscirklen viser, at fordi $-\frac{\pi}{2} \leq \frac{\theta}{2} < \frac{\pi}{2}$ må $\cos\left(\frac{\theta}{2}\right) \geq 0$.

Vi vil nu vise, at z_+ ikke er et stabilt fikspunkt.

$$
\begin{aligned}
|f'(z_+)| &= |1+\sqrt{1-4c}| && \\
&= |1+\sqrt{r}\left(\cos\left(\tfrac{\theta}{2}\right)+i\sin\left(\tfrac{\theta}{2}\right)\right)| && \text{Indsæt nye definition} \\
&= \sqrt{\left(1+\sqrt{r}\cos\left(\tfrac{\theta}{2}\right)\right)^2+\left(\sin\left(\tfrac{\theta}{2}\right)\right)^2} && \text{Udvid } |\dots| \\
&= \sqrt{1+2\sqrt{r}\cos\left(\tfrac{\theta}{2}\right)+r\left(\left(\sin\left(\tfrac{\theta}{2}\right)\right)^2\left(\cos\left(\tfrac{\theta}{2}\right)\right)^2\right)} && \text{Reducér} \\
&= \sqrt{1+2\sqrt{r}\cos\left(\tfrac{\theta}{2}\right)+r} && \sin^2(x)+\cos^2(x)=1
\end{aligned}
$$

Med tanke på at $\cos\left(\frac{\theta}{2}\right) \geq 0$ bliver det nu tydeligt, at uligheden $|f_c'(z_+) < 1|$ aldrig holder.

$$
\begin{aligned}
|f'(z_+)| &= \sqrt{1+2\sqrt{r}\cos\left(\frac{\theta}{2}\right)+r} && (22)\\
&\geq \sqrt{1+r} && (23)\\
&> 1 && (24)
\end{aligned}
$$

Den samme tilgang for z_- giver følgende mindre problematiske udtryk.

$$|f'(z_-)| = \sqrt{1-2\sqrt{r}\cos\left(\frac{\theta}{2}\right)+r}$$

Så de værdier af c for hvilke $|1-\sqrt{1-4c}| < 1$ gælder er medlemmer af den store kardioide.

6.1.4 Medlemmer af cirklen

Vi vender nu vores opmærksomhed mod den anden store udfyldte region; cirklen. Cirklen til højre for den store kardioide består af alle de punkter der har en attraktiv bane med $p = 2$ [8, s. 139–141]. Det er muligt at bestemme de periodiske punkter som indgår i disse baner med samme tilgang som i afsnit 6.1.3. Denne gang bestemmes fikspunkterne $z_\pm$ blot ved $f^2(z_\pm) = z_\pm$.

$$
\begin{aligned}
z_\pm &= f^2(z_\pm) && \text{Iterér } f^2 \\
z_\pm &= f(f(z_\pm)) && \text{Udvid } f \\
z_\pm &= (z_\pm^2+c)^2+c && \text{Udvid, ordén} \\
0 &= z_\pm^4+2cz_\pm^2-z_\pm+c^2+c && (25)
\end{aligned}
$$

Fikspunkterne er således de tal som opfylder afsnit 6.1.4. Her er det dog vigtigt at bemærke, at løsningerne til $f(z_*) = z_*$ selvfølgelig også er løsninger til $f^2(z_*) = f(f(z_*)) = z_*$. Da vi kun er interesserede i at bestemme fikspunkterne til cirklen, er det vigtigt at vi ikke inkluderer disse løsninger i vores resultant.

Vi kan undgå dette problem ved at faktorisere $z_\pm^2 - z_\pm + c$ ud og dele med dette led på begge sider. Herved forkastes løsningerne til $z_\pm^2 - z_\pm + c$.

$$\begin{aligned} 0 &= z_\pm^4 + 2cz_\pm^2 - z_\pm + c^2 + c \\ 0 &= (z_\pm^2 - z_\pm + c)(z_\pm^2 + z_\pm + (c+1)) \\ 0 &= (z_\pm^2 + z_\pm + (c+1)) \end{aligned} \tag{26}$$

Løsningerne til ligning (26) inkluderer således kun fikspunkterne i cirklen. Den kan løses med den sædvanlige løsningsformel for andengradsligninger.

$$z_\pm = \frac{-1 \pm \sqrt{-3-4c}}{2}$$

Vi benytter nu samme stabilitetskriterie som i afsnit 6.1.3. Her husker vi på det faktum at $f(z_+) = f(z_-)$ eftersom at der er tale om binær oscillation frem og tilbage mellem de to fikspunkter.

$$\begin{aligned} 1 &> |(f^2)'(z_+)| \\ 1 &> |f'(f(z_+)) \cdot f'(z_+)| \\ 1 &> |2z_- \cdot 2z_+| \\ 1 &> |(-1 - \sqrt{-3-4c}) \cdot (-1 + \sqrt{-3-4c})| \\ 1 &> |4 + 4c| \\ \frac{1}{4} &> |c - (-1)| \end{aligned} \tag{27}$$

Ligning (27) viser at de stabile fikspunkter danner en cirkel med radius på $\frac{1}{4}$ og centrum i -1.

6.2 Implementering

Implementeringen af denne optimering kræver minimale ændringer. To `if`-sætninger indsættes i starten af funktionen `check` for tjekke at c ikke falder inden for en af de to regioner. Bemærk at det er `check` der ændres, fordi denne optimering omhandler hvordan medlemskab tjekkes.

De to medlemskabsprøver, som blev bestemt i afsnit 6.1.3 og 6.1.4, omformuleres en smule for at gøre dem hurtigere at beregne på computeren. Det drejer sig primært om at eliminere kvadratrødder, da disse er relativt langsomme at beregne. Selvom det ikke påvirker tidskompleksiteten, er det stadig vigtigt at lave denne slags optimeringer, hvis programmet faktisk skal køre hurtigt. Selv en forsinkelse på blot 2 millisekunder for hver pixel på et 1 megapixel stort billede bliver til hele 2 sekunder. Formlen $|1 - \sqrt{1-4c}| < 1$ fra afsnit 6.1.3 omskrives derfor til nedenstående.

$$q\left(q + \left(x - \frac{1}{4}\right)\right) \leq \frac{1}{4}y^2 \text{ hvor } q = \left(x - \frac{1}{4}\right)^2 + y^2 \tag{28}$$

Dette udtryk kommer fra [24], og dets udledning vil ikke blive gennemgået. Ligeledes omskrives udtrykket $|c-(-1)| < \frac{1}{4}$ fra afsnit 6.1.4. Denne omskrivning genkender vi dog straks som formlen for cirklen i $\mathbb{R}^2$-planet.

$$(x+1)^2 + y^2 \leq \frac{1}{16} \tag{29}$$

I begge ligninger henviser x og y selvfølgelig til henholdsvis $Re(c)$ og $Im(c)$.

Listing 4 viser hvordan `check`-funktionen ændres. På afsnit 6.2 indføres prøven fra ligning (28), og på afsnit 6.2 indføres prøven fra ligning (29). Hvis et af udtrykkene er sande er $c \in M$ og `MAX_ITER` returneres. Herved spares eksekveringen af mange tusinde maskininstrukser i løkken på afsnit 6.2.

```
unsigned check(complex double c)
{
    complex double z = 0;
    unsigned n_iter = 0;

    double x = creal(c);
    double y = cimag(c);

    // Check if inside main cardoid.
    double q = (x - 0.25) * (x - 0.25) + y*y;
    if (q * (q + (x - 0.25)) <= 0.25*y*y) {
        return MAX_ITER;
    }

    // Check if inside the period-2 bulb.
    if ((x+1)*(x+1) + y*y <= 0.0625) {
        return MAX_ITER;
    }

    while (cabs(z) <= 2 && n_iter < MAX_ITER) {
        z = z * z + c;
        n_iter += 1;
    }

    return n_iter;
}
```

Listing 4: `check`-funktionen udvidet med karioide-tjek

6.3 Evaluering

Optimeringen beskrevet dette afsnit giver et bemærkelsesværdig formindskelse af tidsforbruget. Med denne metode bruges der ganske få instrukser (konstant tid) på at tjekke de pixels som er i et kredsløb med $p = 1$ eller $p = 2$. Resultatet af `cardoid-benchmark` viser et tidsforbrug på $7,659$ sekunder. Det er en ca. 651% forbedring sammenlignet med `naive-benchmark`.

Det fører til et naturlige spørgsmål om hvorfor man ikke også tjekker for regioner perioder med $p = 3$, $p = 4$, osv. Det skyldes til dels, at en periode-p-banes fikspunkter bestemmes ved løsningen af et 2^p-gradspolynomium [6], men et mere

jordnært argument er, at regionerne med periodelængden $p > 2$ er så små, at det i de fleste tilfælde ikke vil kunne betale sig. Indførslen af flere form-tjek ville nemlig betyde mere arbejde per pixel. Arbejde som vil være spildt på langt de fleste pixels, medmindre der zoomes langt ind på netop denne region. En mulighed for at overkomme denne begrænsning er at bruge Floyds algoritme [16, s. 7] til at registrere periodiske baner, men det kunne snildt fylde sit eget afsnit.

Dette problem gælder på sin vis også for $p = 1$ og $p = 2$. Fordelene ved denne optimering afhænger dog af hvilket område af mængden der undersøges. Hvis området, der undersøges, falder uden for både kardioiden og cirklen giver denne optimering ingen fordele sammenlignet med den naive tilgang. Faktisk er denne tilgang i så fald en smule langsommere, idet unødvendige beregninger bliver foretaget for hver pixel. Disse overflødige beregninger kunne undgås ved at tjekke om området, der undersøges, overlapper med de to former. Hvis det ikke er tilfældet kunne programmet falde tilbage på den gamle definition af `check`, der undgår de (i dette tilfælde) unødvendige beregninger. Denne beslutning kunne spare tid, fordi dette spørgsmål kunne besvares en enkelt gang, frem for hver enkelt pixel.

7 Flere tråde

I frembringelsen af Mandelbrotfraktalen er beregningen af hver pixelværdi uafhængig af hinanden. Det gør det oplagt at beregne flere pixelværdier samtidig for at spare tid. I dette afsnit gennemgås først en specifik tilgang til parallelisering af kode og hvordan denne introduceres i programmet, dernæst evalueres tidsbesparelserne herved.

7.1 Flertrådet programmering

Ordet 'tråd' dækker både over et meget abstrakt koncept og en bestemt implementering heraf.

7.1.1 Abstrakte tråde

Rent konceptuelt kan en tråd betragtes som en sekvens af instrukser der udføres. De fleste simple programmer er *enkelttrådede*, hvilket vil sige at der er en enkelt række instrukser, som udføres. Et eksempel på et sådan enkelttrådet algoritme ses på algoritme 2. Dette er den genkendelige algoritme til bestemmelse af Fibonacci-tal.

Nogle programmer benytter sig af en *flertrådet* udførsel. Det vil sige, at der er flere tråde som kører samtidig. Det sker ved at en tråd forgrener sig og bliver til to tråde. Senere kan de flettes sammen igen. Præcis i hvilken forstand disse tråde kører samtidig vender vi tilbage til senere. Algoritme 3 viser en samme algoritme, som minder om algoritme 2 men benytter flere tråde. På algoritme 3

Algorithm 2 Enkelttrådet algoritme til bestemmelse af Fibonacci-tal

```
procedure FIB(n)
    if n ≤ 1 then
        return n
    else
        a ← FIB(n - 1)
        b ← FIB(n - 2)
        return a + b
    end if
end procedure
```

affødes en ny tråd, som kører funktionen P-FIB med nye parametre. Denne tråd flettes sammen med sin skaber på algoritme 3. På denne måde beregnes a og b samtidig.

Algorithm 3 Flertrådet algoritme til bestemmelse af Fibonacci-tal

```
procedure P-FIB(n)
    if n ≤ 1 then
        return n
    else
        a ← spawn P-FIB(n - 1)
        b ← P-FIB(n - 2)
        sync
        return a + b
    end if
end procedure
```

7.1.2 Styresystemets tråde

Den mere konkrete forståelse at 'tråd' er som et koncept der findes på de fleste styresystemer. Forskellige programmer på en computer kaldes processer [17, 15]. En proces har sit eget stykke virtuel hukommelse, fildeskriptore, m.m., som er isoleret fra andre processer. Hver proces består af én eller flere tråde, der deler ressourcerne i deres fælles proces. Det eneste tråde ikke deler er registre og stakke; hver tråd har sin egen registerfil og sin egen stak. Det tillader tråde at eksekvere forskellige instrukser mens de stadig kan tilgå deres delte ressourcer. Det betyder også, at tråde er langt billigere at oprette end processer [17].

Da der ofte er langt flere tråde end der er CPU-kerner på en given computer, skiftes trådene til at køre på CPUen. Det er styresystemet som sørger for at starte og stoppe processer, sådan at alle tråde får lov at køre ligeligt. Det foregår ved at styresystemet lader en enkelt tråd køre i et lille stykke tid. Når tiden er gået, eller tråden erklærer, at den er færdig med at køre, afbryder styresystemet den nuværende tråd, gemmer dens registre og udskifter dem med registrene

fra den næste tråd. Dette kaldes et *kontekstskift*, og det kan både ske mellem forskellige tråde på tværs af processer*.

Vi vil i dette afsnit beskæftige os med 'tråd' i umiddelbart ovenstående forstand; som objekter i styresystemet, da den faktiske ydeevne afhænger ret meget af den præcise implementering af tråde på den underliggende platform. Tråde i den mere abstrakte forstand kan også være understøttet af f.eks. green threads [25] eller en hændelsesløkke [27].

7.2 Implementering

Den konkrete implementering sker ved en ændring i `mandelbrot`-funktionen*, som den ses på listing 2. Forgreningen og sammenfletningen af tråde defineres i `pthread.h` jævnfør POSIX-standarden [13, s. 315–320]. De to vigtigste funktioner herfra er `pthread_create` og `pthread_join` der står for henholdsvis forgrening og sammenfletning.

Listing 5 viser den nye definition på `mandelbrot`. På afsnit 7.2 defineres antallet af tråde, som arbejdet fordeles på. I dette tilfælde benyttes lige så mange tråde, som computeren har CPU-kerner. Den primære tråd forgrener sig ud til mange arbejdertråde på afsnit 7.2. Hver arbejdertråd modtager en række parametre i form af strukturen `struct thread_params`, som informerer den om hvilket område at billedet den pågældende tråd er ansvarlig for at generere. Først når alle tråde er i gang venter den primære tråd på at de andre tråde færdiggøres på afsnit 7.2. Det er vigtigt at sammenfletningen foregår *efter* samtlige tråde er startet, ellers ville trådene blot køre én ad gangen.

```
1  void mandelbrot(unsigned width, unsigned height, double
     buffer[height][width],
2          double real_min, double real_max,
3          double imag_min, double imag_max)
4  {
5      long num_threads = sysconf(_SC_NPROCESSORS_ONLN);
6      unsigned rows_per_thread = height / num_threads;
7
8      pthread_t threads[num_threads];
9      struct thread_parameters thread_params[num_threads];
10     for (unsigned i = 0; i < num_threads; i++) {
11         thread_params[i] = (struct thread_parameters) {
12             .width = width,
13             .height = height,
14             .buffer = (double *)buffer,
15
16             .y = i * rows_per_thread,
17             .num_rows = (i == num_threads - 1)
18                 ? height - (num_threads - 1) * rows_per_thread
19                 : rows_per_thread,
20
```

*Kontekstskift mellem tråde fra forksellige processer er selvfølgelig markant dyrere, da de kræver at flere ressourcer udskiftes.

*At det netop er `mandelbrot`-funktionen der ændres afspejler at denne optimering omhandler i hvilke pixels der tjekkes.

```
21              .real_min = real_min,
22              .real_max = real_max,
23              .imag_min = imag_min,
24              .imag_max = imag_max,
25          };
26
27          int ret = pthread_create(&threads[i], NULL, check_thread,
    &thread_params[i]);
28          if (ret != 0) {
29              perror("spawn thread");
30              abort();
31          }
32      }
33
34      for (unsigned i = 0; i < num_threads; i++) {
35          pthread_join(threads[i], NULL);
36      }
37  }
```

Listing 5: `mandelbrot`-funktionen udvidet til at håndtere flere tråde

Funktionen `pthread_create` tager en pointer til den funktion som skal eksekveres i den nye tråd. I dette tilfælde er det `check_thread`. Den kører igennem hver række, som tråden er ansvarlig for og kører `check` på hver pixel. Funktionen ses på listing 6. Foruden de to yderste løkker er indholdet det samme som i løkken på listing 2. Dette afspejler, at den eneste ændring er i måden arbejdet fordeles; oversættelsen fra pixel-koordinater til komplekse tal (afsnit 7.2 til 7.2) og beregningen af pixelværdier (afsnit 7.2 til 7.2) forbliver den samme.

```
1  void *check_thread(void *ptr)
2  {
3      struct thread_parameters *p = ptr;
4
5      for (unsigned y = p->start_y; y < p->start_y + p->num_rows;
    ++y) {
6          for (unsigned x = 0; x < p->width; ++x) {
7              double real = p->real_min +
    ((double)x/(double)p->width)  * (p->real_max - p->real_min);)
8              double imag = p->imag_min +
    ((double)y/(double)p->height) * (p->imag_max - p->imag_min);
9              complex double c = CMPLX(real, imag);)
10
11             unsigned n_iter = check(c);
12             double value = (double)n_iter / (double)MAX_ITER;
13             p->buffer[y * p->width + x] = value;
14         }
15     }
16
17     return NULL;
18 }
```

Listing 6: Hver tråd kører `check_thread`-funktionen

Trådene deler som beskrevet i afsnit 7.1 det same virtuelle adresserum. Det vil sige, at når hver tråd skriver til hukommelsen på afsnit 7.2 er det den samme

del af den primære hukommelse de tilgår. I mange flertrådede programmer er dette et problem, men i dette tilfælde er hver tråd kun ansvarlig for sine egne rækker. Der er derfor ingen risiko for at to tråde skriver eller læser til samme stykke hukommelse samtidig.

7.3 Evaluering

Umiddelbart ser den flertrådede bagende ikke ud til at give nogle fordele. `thread-benchmark` viser et tidsforbrug på 71,902 sekunder. Det høje tal skyldes at `benchmark` som sagt i afsnit 5.2 måler CPU-tid. Når `clock_gettime` kaldes med `CLOCK_PROCESS_CPUTIME_ID` returneres den mængde CPU-tid som processen. Det vil sige, at hvis 2 tråde kører i 5 sekunder hver har de brugt 10 sekunders CPU-tid. Derfor er det, beklageligvis nødvendigt at måle tidsforbruget for denne bagende i realtid, dvs. ved at kalde `clock_gettime` med `CLOCK_PROCESS_REALTIME_ID`. Dette introducerer en vis unøjagtighed, men ikke meget. For basisimplementeringen betyder forskellen på CPU-tid og realtid kun en forskel på 1,56% i det rapporterede tidsforbrug. Målt i realtid rapporterer `threads-benchmark` et tidsforbrug på blot 9,502 sekunder; en markant forbedring.

Et oplagt opfølgende spørgsmål på disse tal er: hvor mange tråde giver det mindste tidsforbrug? Indledningsvist kan vi sætte nogle ydre grænser. Det er klart at én enkelt tråd ikke giver den optimale ydeevne, da det ikke udnytter samtlige CPU-kerner på computeren. Omvendt er én tråd per pixel også ineffektivt, da det vil forårsage utallige unødige kontekstskift. På afsnit 7.2 i listing 5 benyttes lige så mange tråde som der er CPU kerner. Eftersom trådenes arbejde er CPU-bundet skulle dette teoretisk set være det optimale antal tråde. Det undgår unødige kontekstskift samtidig med at hver enkelt CPU-kerne kan køre på 100% udnyttelse. Det viser sig imidlertid ikke at holde stik i realiteten. Figur 10 viser sammenhængen mellem antal tråde og ydeevne på test-computerens specifikke kombination af hardware og styresystem. Heraf fremgår det, at det optimale antal tråde på denne maskine er omkring 64. Der er mange mulige forklaringer på misforholdet mellem det optimale antal tråde og antallet af CPU'kerner.

En mulig forklaring er, at 64 tråde med nær 100% CPU-udnyttelse tvinger computeren fra "lower power mode" til "performance mode". M1-chippen indeholder 4 "firestorm" kerner, der fokuserer på ydeevne og 4 "icestorm" kerner, der fokuserer på at minimere strømforbruget [10]. Det er muligt, at 64 tråde er tilstrækkelig belastning til at styresystemet vurderer, at det er passende at aktivere "firestorm" kernerne.

En anden mulig forklaring er, at en kombination af den specifikke scheduling-algoritme, der benyttes, og hastigheden på det forskellige cache-lag mellem CPU-en og den primære hukommelse påvirker hvor hurtigt instrukserne kan skrive til billed-bufferen. I virkeligheden skyldes det formentlig en kombination af disse såvel som mange andre faktorer, som ændrer sig alt efter om computeren er sat til strøm, hvilken version af styresystemet der køres, m.m.

Antallet af tråde er altså en vigtig parameter at justere, men også én som er svær

Antal tråde	Tidsforbrug (s)
2	25,978
4	25,666
16	19,468
32	13,502
64	9,908
128	9,785
256	10,078

Figur 10: Tidsforbrug ved forskellige antal tråde

at finjustere for mennesker, der ikke har fuld forståelse af computerens mange dele og deres indbyrdes forhold. Det fornuftigste ville derfor formentlig være at køre en runtime test som beskrevet af [23]. Ved en runtime test foretages en række test, når programmet først starter op. Testene har til formål at bestemme de specifikke ydelsesegenskaber på slutbrugerens kombination af hardware og software. Baseret på testresultaterne vælges de optimale parametre automatisk.

8 Spejling i $\mathbb{R}$-aksen

Ser man på et billede af Mandelbrotfraktalen bliver det tydeligt, at den er spejlet i den reelle akse*. Ud fra denne observation er det muligt at undlade at beregne hele rækker af pixelværdier, hvis den pågældende række findes på den anden side af $\mathbb{R}$-aksen. I dette afsnit gennemgås et kort bevis for at denne observation holder stik, samt implementeringen af en `mandelbrot`-funktion, der benytter sig af denne observation til at spare tid.

8.1 Matematisk grundlag

Når et komplekst tal er "spejlet i $\mathbb{R}$-aksen" er der tale om tallets *konjugat*. I afsnit 8.1.1 gennemgås regnereglerne for konjugater og i afsnit 8.1.2 bruges disse til et kort bevis for at Mandelbrotmængden er spejlet i $\mathbb{R}$-aksen.

8.1.1 Konjugater

Hvis et tal "spejles i den reelle akse" er der tale om et konjugat. Hvis et komplekst tal er givet ved

$$z = a + ib$$

vil dets konjugat være givet ved

$$\overline{z} = a - ib$$

*Husk fra afsnit 3.1.3, at den reelle talrække udgør førsteaksen i det komplekse talplan. Den imaginære talrække udgør andenaksen.

Notationen $\overline{z}$ bruges til at angive konjugater, og de beregnes ved at negere den imaginære del [9]. De sædvanlige regneregler for komplekse tal gælder stadig. Analogt til definitionen af summen af komplekse tal i afsnit 3.1.2 er $\overline{z_1} + \overline{z_2} = \overline{z_1 + z_2}$.

$$\begin{aligned}\overline{z_1} + \overline{z_2} &= a_1 - ib_1 + a_2 - ib_2 \\ &= a_1 + a_2 - ib_2 - ib_1 \\ &= (a_1 + a_2) - i(b_2 + b_1) \\ &= \overline{z_1 + z_2}\end{aligned}$$

Ligeledes er $\overline{z_1} \cdot \overline{z_2} = \overline{z_1 \cdot z_2}$. Da denne udledning også minder meget om den tilsvarende i afsnit 3.1.2, springes et par af de mere trivielle trin over.

$$\begin{aligned}\overline{z_1} \cdot \overline{z_2} &= \overline{(a_1 + ib_1)} \cdot \overline{(a_2 + ib_2)} \\ &= (a_1 - ib_1) \cdot (a_2 - ib_2) \\ &= (a_1 a_2 - b_1 b_2) - i(a_1 b_2 + a_2 b_1)\end{aligned}$$

Disse to regneregler er de eneste vi har brug for til at gennemføre beviset i næste afsnit.

8.1.2 z_n for cs konjugat

Vi har nu den fornødne baggrundsviden til at bevise at Mandelbrotfraktalen er symmetrisk i den reelle akse. Dette svarer til at bevise, at $f_{\overline{c}}^n(0) = \overline{f_c^n(0)}$, dvs. at hvis c spejles i den reelle tallinje så vil talfølgen z_n også. Hvis dette er sandt, betyder det at $c \in M \Rightarrow \overline{c} \in M$.

Det er åbenlyst at $f_c(0) = 0 = f_{\overline{c}}(0)$. Antag nu at $f_{\overline{c}}(k) = \overline{f_c(k)}$. Vi ønsker nu at vise, at $f_{\overline{c}}(k+1) = \overline{f_c(k+1)}$, idet vi husker på regnereglerne defineret i afsnit 8.1.1.

$$\begin{aligned}f_{\overline{c}}(k+1) &= \left(f_{\overline{c}}(k)\right)^2 + \overline{c} && \big) \textit{ Antagelse} \\ &= \left(\overline{f_c(k)}\right)^2 + \overline{c} && \big) \ \overline{z_1} \cdot \overline{z_2} = \overline{z_1 \cdot z_2} \\ &= \overline{\left(f_c(k)\right)^2} + \overline{c} && \big) \ \overline{z_1} + \overline{z_2} = \overline{z_1 + z_2} \\ &= \overline{\left(f_c(k)\right)^2 + c} && \big) \textit{ Definition på } f_c \\ &= \overline{f_c(k+1)}\end{aligned}$$

Per induktionsprincippet har vi nu vist, at $f_{\overline{c}}(n) = \overline{f_c(n)}$ for alle $n \in \mathbb{N}$ og dermed også at $f_{\overline{c}}^n(0) = \overline{f_c^n(0)}$.

8.2 Implementering

Algoritme 4 demonstrerer de nødvendige ændringer til algoritme 1 for at udnytte symmetrien. Linje 1 til 9 begrænser arealet der genereres til enten kun

det over eller under den reelle talrække, hvad end der er størst. Det er vigtigt at det største område renderes, da der ellers vil opstå et hul omkring den reelle talrække. Algoritmer 4 til 4 begrænser arealet, der genereres, til enten kun det over eller under den reelle talrække, hvad end der er størst. På algoritmer 4 til 4 er der ingen ændringer, med undtagelse af at kun rækkerne fra y_{min} til y_{maks} gennemgås. Slutteligt spejles rækkerne på algoritmer 4 til 4. Det sker ved funktionen copyRowFromTo($matrix, row_{from}, row_{to}$), der kopierer en række pixelværdier fra row_{from} til row_{to}. I den tilsvarende C-kode er dette implementeret med funktionen `memcpy`, der indgår som en del af C-sprogets standardbibliotek. Funktionen er som oftest højt optimeret og kan flytte op mod en gibibyte i sekundet.

Algorithm 4 Pseudokode for escape-time visualisering med symmetri-tjek

Require: Re_{min}, Re_{maks}, Im_{min}, Im_{maks},
Ensure: $Re_{min} > Re_{maks}$, $Im_{maks} > Im_{min}$,
  $y_{min} \leftarrow 0$
  $y_{maks} \leftarrow s_y$
  if $Im_{maks} < 0 \land Im_{min} > 0$ **then** ▷ Symmetri-tjek
    if $Im_{maks} + Im_{min} \geq 0$ **then**
      $y_{maks} \leftarrow \frac{Im_{maks} \cdot s_y}{Im_{maks} - Im_{min}} + 1$
    else
      $y_{min} \leftarrow \frac{Im_{maks} \cdot s_y}{Im_{maks} - Im_{min}}$
    end if
  end if
  for each pixel ($P_x, P_y \in [y_{min}; y_{maks}]$) on the screen **do** ▷ Samme som normalt
    $a \leftarrow Re_{min} + \frac{P_x}{s_x} \cdot (Re_{maks} - Re_{min})$
    $b \leftarrow Im_{min} + \frac{P_y}{s_y} \cdot (Im_{maks} - Im_{min})$
    $c \leftarrow a + ib$ ▷ Konvertér skærmkoordinater til $c \in \mathbb{C}$
    $z \leftarrow 0$
    $n \leftarrow 0$
    while $|z| < 2 \land n \leq n_{maks}$ **do**
      $z \leftarrow z^2 + c$
      $n \leftarrow n + 1$
    end while
    $pixels[P_y][P_x] \leftarrow \frac{n}{n_{maks}}$
  end for
  for $y \in [0; y_{min}]$ **do** ▷ Kopiér de andre rækker
    copyRowFromTo($pixels, y_{min} + y_{min} - y, y$)
  end for
  for $y \in [y_{min}; s_y]$ **do**
    copyRowFromTo($pixels, y_{maks} + y_{maks} - y, y$)
  end for

8.3 Evaluering

Den nye bagende navngives `symmetry`. Programmet `symmetry-benchmark` viser, at denne metode tager 25,300 sekunder om at rendere målebilledet. Det er cirka dobbelt så hurtigt som `naive`, hvilket giver mening, da der udføres nogenlunde halvt så meget arbejde, fordi målebilledets område deles horisontalt af den reelle talrække. Det viser desuden, at prisen for de ekstra kald til `memcpy` er forholdsvis lav sammenlignet med arbejdet i `check`.

Ligesom kardiode-tjek-optimeringen i afsnit 6 afhænger fordelene ved denne metode dog også af hvilket område der undersøges. Det område `benchmark` undersøger giver optimale forhold for algoritmen, eftersom at billedet er centreret omring $\mathbb{R}$-aksen. Det betyder, at kun halvdelen af alle rækker skal renderes. Besparelserne er proportionale med hvor tæt $\mathbb{R}$-aksen er på midten af områder der undersøges.

Ulig `cardoid` men lig `threads` indebærer denne optimering store ændringer i `mandelbrot`-funktionen. Disse ændringer bringer koden længere væk fra de bagvedliggende matematiske koncepter, men den forbedrede tid retfærdiggør denne.

9 Konklusion

Mandelbrotmængden er defineret som alle de komplekse tal c, hvor hvilke systemet

$$z_n = z_{n-1}^2 + c$$

ikke går mod uendelig, når n går mod uendelig. Mandelbrotfraktalen opstår ved afbildningen af Mandelbrotmængden på det komplekse talplan. På computeren kan billeder af bl.a. Mandelbrotfraktalen fremstilles ved escape-time-visualisering. Ved escape-time-visualisering indskrænkes definitionen på Mandelbrotmængden for at gøre det muligt at bestemme medlemskab i mængden på endelig tid. Der er blevet gennemgået 3 forskellige variationer på den grundlæggende escape-time-algoritme, der hver især byggede oven på forskellige observationer om Mandelbrotmængden.

I afsnit 5.1 blev den basale implementering gennemgået. Denne satte basislinjen, som optimeringerne blev målt imod. I afsnit 6 fremvistes en optimering, der bestod i for hver pixel først at tjekke for medlemskab i hovedkardioiden eller cirklen med $p = 2$. Den flertrådede implementering fra afsnit 7 benyttede flere tråde til at beregne flere rækker på samme tid. Slutteligt blev det i afsnit 8 observeret at beregningen af mange rækker kunne spares væk, da Mandelbrotfraktalen er symmetrisk i den reelle akse.

Med undtagelse af introduktionen af flere tråde, havde hver optimering både fordele og ulemper. Regionstjekkene i afsnit 6 sparer uhyre mange iterationer af den dyre løkke i `check`, men hvis området, der undersøges, ikke overlapper med de to regioner er de ekstra beregninger blot dødvægt. Det samme gælder i mindre grad for optimeringen introduceret i afsnit 8; hvis området ikke skæres

af $\mathbb{R}$-aksen spildes et par operationer. Der er dog tale om langt færre spildte operationer, da disse foregår i `mandelbrot`-funktionen og dermed ikke gentages for hver enkelt pixel.

På baggrund af disse overvejelser er det tydeligt, at det altid kan betale sig, at bruge flere tråde, men at det præcise antal skal finjusteres per-maskine, evt. som beskrevet i afsnit 7.3. Ligeledes kan det altid betale sig at inkludere symmetritjekkene da de koster relativt lidt, hvis de ikke er relevante. Det samme kan dog ikke siges om kardioide-tjekkene. Da besparelserne er så store, når den er relevant, vil inklusion i den interaktive grænseflade være selvfølgelig, men i zoom-generatoren, hvor meget små områder undersøges og der generes mange billeder, vil det ekstra arbejde per pixel gøre det ubæredygtigt.

Dette projekt gav grundlag for vurdering af om de omtalte optimeringer er passende i et givent program. Der blev desuden opstillet et rammeværk for at vurdere fremtidige optimeringer.

Litteratur

[1] David Borland og Russell M. Taylor Ii. „Rainbow Color Map (Still) Considered Harmful". I: *IEEE Computer Graphics and Applications* 27.2 (mar. 2007), s. 14–17. ISSN: 0272-1716, 1558-1756. DOI: 10.1109/MCG.2007.323435. URL: https://ieeexplore.ieee.org/document/4118486/ (hentet 13.12.2023).

[2] Jeffrey R. Chasnov. *8.1: Fixed Points and Stability*. Mathematics LibreTexts. 5. nov. 2021. URL: https://math.libretexts.org/Bookshelves/Differential_Equations/Differential_Equations_(Chasnov)/08%3A_Nonlinear_Differential_Equations/8.01%3A_Fixed_Points_and_Stability (hentet 15.12.2023).

[3] Andy Chu. *The Internet Was Designed With a Narrow Waist*. Oil Shell. 26. feb. 2022. URL: https://www.oilshell.org/blog/2022/02/diagrams.html (hentet 20.12.2023).

[4] Krzysztof Ciesielski. „On Stefan Banach and Some of His Results". I: *Banach Journal of Mathematical Analysis* 1.1 (2007), s. 1–10. ISSN: 1735-8787. DOI: 10.15352/bjma/1240321550. URL: http://projecteuclid.org/euclid.bjma/1240321550 (hentet 21.12.2023).

[5] Lokenath Debnath. „A Brief Historical Introduction to Fractals and Fractal Geometry". I: *International Journal of Mathematical Education in Science and Technology* 37.1 (15. jan. 2006), s. 29–50. ISSN: 0020-739X, 1464-5211. DOI: 10.1080/00207390500186206. URL: http://www.tandfonline.com/doi/abs/10.1080/00207390500186206 (hentet 20.12.2023).

[6] Robert L. Devaney. *A First Course in Chaotic Dynamical Systems: Theory and Experiment*. Second edition. A Chapman & Hall Book. Boca Raton London New York: CRC Press, Taylor & Francis Group, 2020. 318 s. ISBN: 978-0-367-23599-4.

[7] Ulrich Drepper. „What Every Programmer Should Know about Memory“. I: 2007. URL: https://api.semanticscholar.org/CorpusID:6440708.

[8] Michael Frame, Amelia Urry og Steven H. Strogatz. *Fractal Worlds: Grown, Built, and Imagined*. New Haven: Yale University Press, 2016. 515 s. ISBN: 978-0-300-19787-7.

[9] Jesper Frandsen. *Komplekse tal og fraktaler*. 1. Ausg., 2. Aufl. Herning: Systime, 1995. 144 s. ISBN: 978-87-7783-188-1.

[10] Andrei Frumusanu. *The 2020 Mac Mini Unleashed: Putting Apple Silicon M1 To The Test*. URL: https://www.anandtech.com/show/16252/mac-mini-apple-m1-tested (hentet 20.12.2023).

[11] P.S. Gill. *Operating Systems Concepts*. Firewall Media, 2006. ISBN: 978-81-7008-913-1. URL: https://books.google.dk/books?id=eQ0Z1JWI7AwC.

[12] Michael T. Goodrich og Roberto Tamassia. *Algorithm Design: Foundations, Analysis, and Internet Examples*. New Delhi: Wiley-India, 2011. ISBN: 978-81-265-0986-7.

[13] „IEEE Standard for Information Technology–Portable Operating System Interface (POSIX(TM)) Base Specifications, Issue 7“. I: *IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008)* (31. jan. 2018), s. 1–3951. DOI: 10.1109/IEEESTD.2018.8277153.

[14] Christian Ulrik Jensen. *Tallegeme*. Den Store Danske. 2. feb. 2009. URL: https://denstoredanske.lex.dk/tallegeme (hentet 11.12.2023).

[15] *KLID: Dansk-gruppens og KLIDs engelsk/dansk edb-ordliste*. KLID: En forening for professionelle Linux-interessenter i Danmark. URL: http://www.klid.dk/dansk/ordlister/ordliste.html (hentet 19.12.2023).

[16] Donald Ervin Knuth. *The Art of Computer Programming*. 3rd ed. Reading, Mass: Addison-Wesley, 1997. 3 s. ISBN: 978-0-201-89683-1 978-0-201-89684-8 978-0-201-89685-5.

[17] Monocasual Laboratories. *A Gentle Introduction to Multithreading*. Internal Pointers. URL: https://www.internalpointers.com/post/gentle-introduction-multithreading (hentet 19.12.2023).

[18] Benoit Mandelbrot. „How Long Is the Coast of Britain? Statistical Self-Similarity and Fractional Dimension“. I: *Science* 156.3775 (5. maj 1967), s. 636–638. ISSN: 0036-8075, 1095-9203. DOI: 10.1126/science.156.3775.636. URL: https://www.science.org/doi/10.1126/science.156.3775.636 (hentet 20.12.2023).

[19] Benoit B. Mandelbrot. *The Fractal Geometry of Nature*. San Francisco: W.H. Freeman, 1982. 460 s. ISBN: 978-0-7167-1186-5.

[20] Michael Jensen og Klaus Marthinus og Bernt Hansen. „Komplekse tal“. I: *MAT A htx*. Systime, 2023, p409. ISBN: 978-87-616-9361-7. URL: https://mathtxa.systime.dk/?id=409 (hentet 11.12.2023).

[21] Jeffrey C. Mogul og Anita Borg. „The Effect of Context Switches on Cache Performance“. I: *ACM SIGPLAN Notices* 26.4 (2. apr. 1991), s. 75–84. ISSN: 0362-1340, 1558-1160. DOI: 10.1145/106973.106982. URL: https://dl.acm.org/doi/10.1145/106973.106982 (hentet 16.12.2023).

[22] Kenneth Moreland. „Diverging Color Maps for Scientific Visualization“. I: *Advances in Visual Computing*. Red. af George Bebis m.fl. Bearb. af David Hutchison m.fl. Bd. 5876. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, s. 92–103. ISBN: 978-3-642-10519-7 978-3-642-10520-3. DOI: 10.1007/978-3-642-10520-3_9. URL: http://link.springer.com/10.1007/978-3-642-10520-3_9 (hentet 13.12.2023).

[23] Robert P. Munafo. *Runtime Benchmarking*. Muency. 20. okt. 1996. URL: https://mrob.com/pub/muency/runtimebenchmarking.html (hentet 20.12.2023).

[24] *Plotting Algorithms for the Mandelbrot Set*. I: *Wikipedia*. 11. dec. 2023. URL: https://en.wikipedia.org/w/index.php?title=Plotting_algorithms_for_the_Mandelbrot_set&oldid=1189360425#Cardioid_/_bulb_checking (hentet 18.12.2023).

[25] Kevin Rosendahl. „Green Threads in Rust“. I: 1.1 (dec. 2017), s. 7.

[26] Paul Silisteanu. *The Mandelbrot Set in C++11*. Solarian Programmer. 28. feb. 2013. URL: https://solarianprogrammer.com/2013/02/28/mandelbrot-set-cpp-11/ (hentet 13.12.2023).

[27] *The Node.Js Event Loop, Timers, and Process.nextTick() — Node.Js*. URL: https://nodejs.org/en/guides/event-loop-timers-and-nexttick (hentet 21.12.2023).

[28] Eric W. Weisstein. *Bernstein Polynomial*. URL: https://mathworld.wolfram.com/ (hentet 13.12.2023).

[29] Eric W. Weisstein. *Smooth Function*. URL: https://mathworld.wolfram.com/ (hentet 13.12.2023).

[30] Dr Lindsay Robert Wilson. „Distance Estimation Method for Drawing Mandelbrot and Julia Sets“. I: (20. nov. 12).