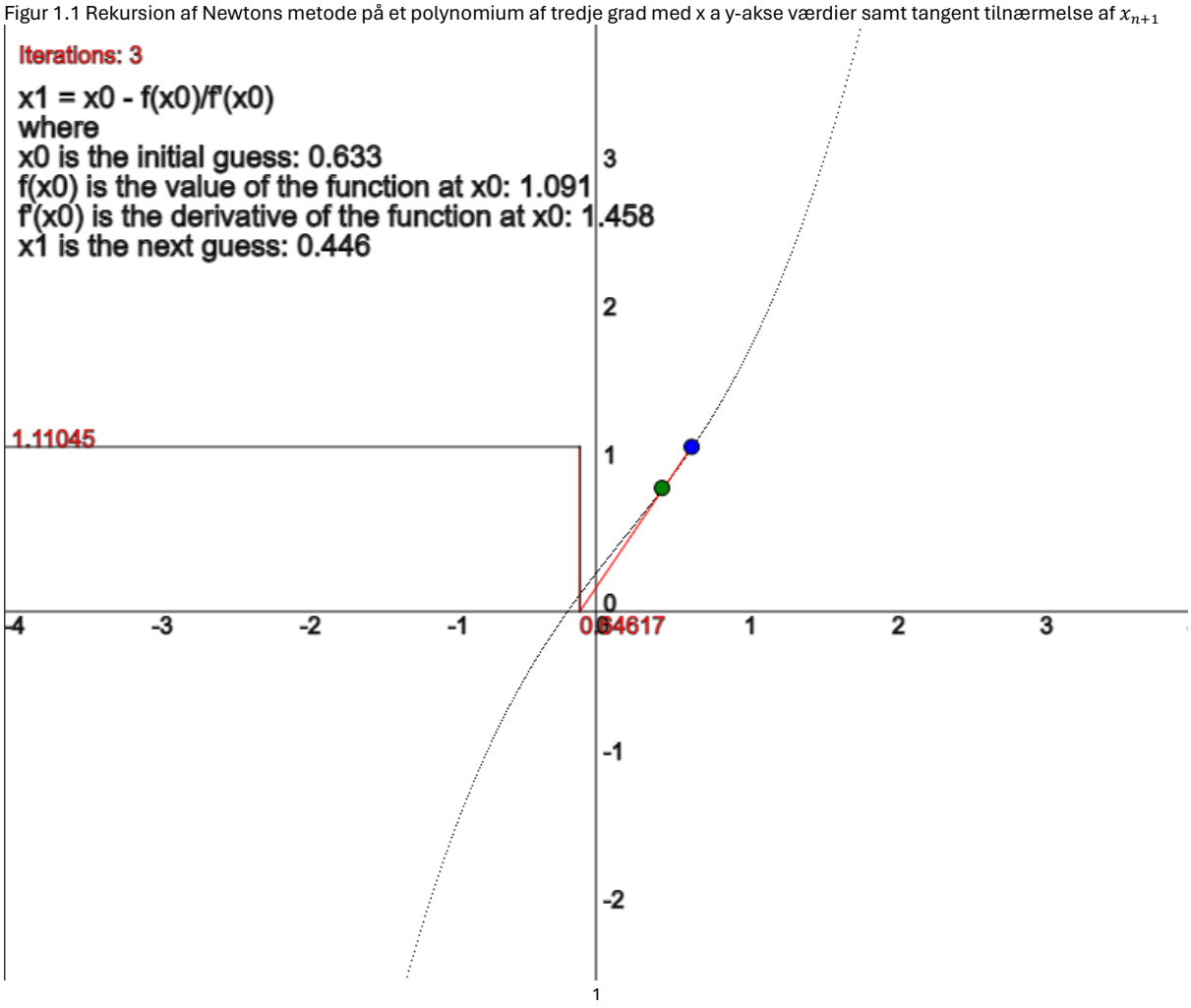
Newton-Raphson metode visualiseret

Det rekursive rod-findende algoritme

Figur 1.1 Rekursion af Newtons metode på et polynomium af tredje grad med x a y-akse værdier samt tangent tilnærmelse af x_{n+1}



[1] https://github.com/mbn-code/newton-raphsons-root-finder (Norengaard Malthe Bang, tilgået 15:00 22/02 2024)

Indholdsfortegnelse

Introduktion

I denne rapport vil vi undersøge og implementere Newton-Raphson metoden, et rod-findende algoritme på et polynomium. Metoden er til at finde rødderne af en reel-værdi funktion og er baseret på principperne iteration (implementeret med recursion) for at tilnærme sig rødderne med en tæt på 0 defineret epsilon, og diskutere styrker og svagheder ved min løsning. Jeg vil også redegøre og formulere newtons metode som en rekursionsligning. Hvordan kan vi effektivt implementere Newton-Raphson metoden i programmering for at finde rødderne af en funktion? Hvordan påvirker valget af det indledende gæt og præcisionen af resultaterne?

Arbejdsspørgesmål:

- Hvordan fungerer Newton Raphson metoden, og hvad er den matematiske baggrund for den?
- Hvordan kan vi implementere Newton Raphson metoden i programmering?
- Hvordan påvirker det indledende gæt og præcision nøjagtigheden og effektiviteten af metoden?

I denne rapport vil vi derfor forsøge at besvarer disse spørgsmål ved brug af teoretisk gennemgang og praktisk implementering. VI vil også diskutere de potentielle udfordringer og begrænsninger ved Newton-Raphson metoden.

Teoriafsnit

Rekursionsligninger udgør en vigtig del af matematik og datalogi, da det kan bruges til at beskrive gentagende og selvreferentielle processer. Dette afsnit vil introducere grundlæggende begreber og metoder så som rekursionsligninger, tangenter, rekursion som princip og polynomier.

Rekursionsligninger

En rekursionsligninger er en matematisk ligning, som beskriver en sekvens eller en funktion ved hjælp af sig selv, ved at have (for det meste) pre-defineret elementer i sekvensen eller funktionen baseret på tidligere elementer i samme sekvens eller funktion.

Newton Raphson

Et eksempel er Newton Raphson metode. Newtons metode er en iterativ (implementeret med rekursion) numerisk teknik til at finde rødder af en funktion. Metoden baseres på at starte med et gæt x0 på en ønsket rod og derefter forbedre gættet gradvist ved brug af tangentlinjen til funktionen ved x0. Beregning ser således ud:

Initialbetingelse x_0 og epsilon : vælg et startgæt x_0, og en ønsket præcision epsilon
Epsilon: epsilon kan defineres hvis der er en ønsket præcision som skal stoppes ved, for at undgå uendelig iteration.
Iterativ proces: *Følgende trin gentages indtil roden er fundet:*
$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$ (Hvor f(x) er den givne funktion, og f'(x) er den afledte)
Konvergens mod epsilon: Iterationen / rekursionen fortsættes indtil den absolutte værdi af y er det samme eller mindre som epsilon, eller man har overskrid et maks iterations punkt, som også kan betegnes matematisk som:
$|y - x_n| \leq epsilon$

Fibonacci (ekstra)

Et eksempel på rekursive ligninger til det 5 fibonacci tal F_5 i matematik ser sådan her ud:

Start med at definere F_1 og F_2 til 1 for at kunne starte talfølgen. Derefter går vi efter formlen $F_n = F_{n-1} + F_{n-2}$ hvor n er initialbetingelserne F_1og F_2. For så at finde det 5 tal i talfølgen ved hjælp af rekursionsligninger beregner vi $F_3, F_4 \, og \, F_5$ Således:
$F_3 = F_{3-1} + F_{3-2} = F_2 + F_1 = 1 + 1 = 2$
$F_4 = F_{4-1} + F_{4-2} = F_3 + F_2 = 2 + 1 = 3$
$F_5 = F_{5-1} + F_{5-2} = F_4 + F_3 = 3 + 2 = 5$
Hvor vi så kan regne ud ved hjælp af rekursionsligningerne at $F_5 = 5$

Tangentens ligning for newtons metode

I Newtons metode bruges tangentens ligning til at finde en tilnærmelse til roden af en funktion rekursivt. Tangentens ligning er givet ved: $y = f'(x_n)(x - x_n) + f(x_n)$, hvor:

- f er funktionen, hvis rod vi prøver at finde
- $f'(x_n)$er den afledte af f ved x_n
- $x_n$er det nuværende gæt (tilnærmelse til roden)

I Newton Raphson metoden bruger vi tangentens ligning til at finde det punkt, hvor tangenten skærer x-aksen, (hvor y = 0) som er vores næste tilnærmelse til roden. Så derfor løser vi tangentens ligning for x når y = 0.
$0 = f'(x_n)(x - x_n) + f'(x_n)$
Som så giver os:
$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$
Som så er vores generelle formel for tilnærmelsen af roden ved $y = 1e - 10$

Empiri

For at implementere Newton-Raphson metoden I programmering rekursivt har man brug for at definere en funktion der kalder sig selv. I mit forsøg på at implementere og visualisere Newtons metode til at tilnærme sig en rod efter et gæt gjorde jeg følgende.

***Newtons recursive funktion*:**

```
function newtonMethod(f, fDerivative, x0, epsilon) {
  const fx = f(x0);
  const fpx = fDerivative(x0);

  if (Math.abs(fx) < epsilon) {
    console.log("Root found:", x0);
    return x0;
  }

  const x1 = x0 - fx / fpx;

  console.log("Current guess:", x0);
  console.log("New guess:", x1);

  return newtonMethod(f, fDerivative, x1, epsilon);
}
newtonMethod(
  x => x**5+3.5*x**4-2.5*x**3-12.5*x**2+1.5*x+9,
  x => 5*x**4+14*x**3-7.5*x**2-25*x+1.5,
  4,
  1e-10
);
```
[2]

Overstående ses mit forsøg på at implementere Newton Raphson metoden i JavaScript. Efterfølgende har jeg brugt min viden til at udvide programmet, og skrevet visualiseringsdelen med en smuk animation af en parabel i selvgiven grad og et

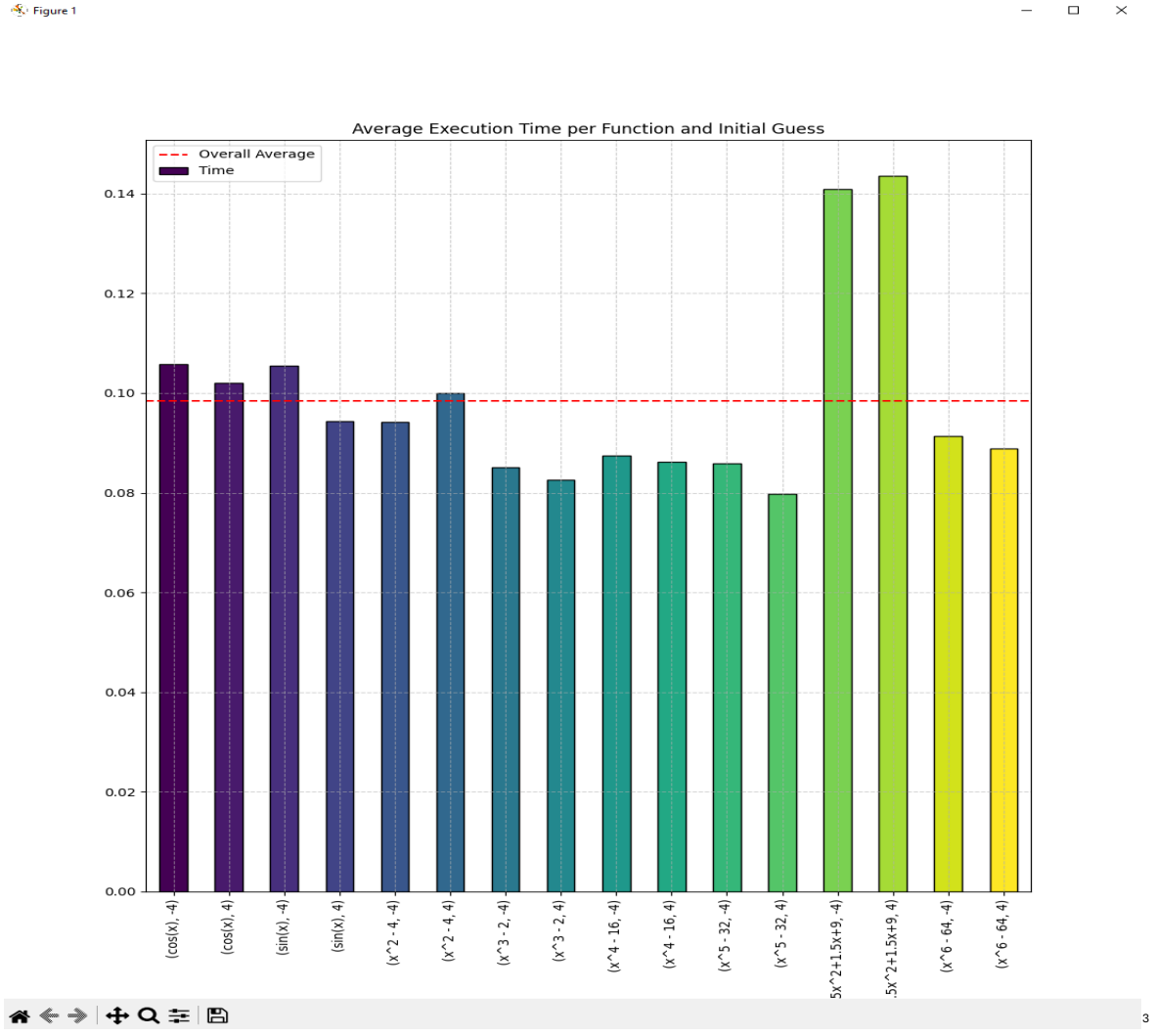
[2] https://github.com/mbn-code/newton-raphsons-root-finder/blob/237e8e4169650cedd211a22dfb04321698a37986/src_js/newton_rec.js#L2-L27

koordinatsystem, hvor nulpunktet tættest på x_0 start gæt findes, som kan ses i følgende link. En visuel repræsentation kan findes ved at downloade filen her eller online her.
En ny graf genereres ved brug af `ctrl + s` eller `command + s` på macOS. Og et nyt selvvalgt gæt på samme graf kan gøres ved et venstre klik på musen et vilkårligt sted i koordinatsystemet.

Det her kode hjælper med til at bevise og repræsentere den teoretiske del af Newton Raphson metoden og hvordan den implementeres i programmering.
Jeg har også implementeret algoritmerne i programmerings sprogene Python og Rust udover JavaScript som kan findes her og her. Som gør det samme, men i forskellige sprog.

Følgende har jeg skrevet programmer i de respektive sprog, som udøver visualisering data af algoritmernes hastighed skrevet i Python og JavaScript. Dette er med til at vise hastigheden af de forskellige sprog og implamentations forskellen, som kan findes her og her med deres respektive analyse filer her og her. Og kan følgende visualiseres som understående figur 1.2:

Figur 1.2 Visuel repræsentation af hastighed af Newton Raphson algoritme implementation i JavaScript på forskellige funktioner.



[3]

[3] https://github.com/mbn-code/newton-raphsons-root-finder/blob/master/newton-raphson-algoritme-hastighed-javascript.png?raw=true

Analyse af empiri på baggrund af teori

Implementation af Newton Raphson demonstreres ved en rekursiv funktion som jeg har skrevet i JavaScript, Python og Rust. For at lave en nem forståelig analyse af den teoretiske metode som implementation vil jeg tage udgangspunkt i newton_rec.js - også set nedenstående.

```
function newtonMethod(f, fDerivative, x0, epsilon) {
  const fx = f(x0);
  const fpx = fDerivative(x0);

  if (Math.abs(fx) < epsilon) {
    console.log("Root found:", x0);
    return x0;
  }

  const x1 = x0 - fx / fpx;

  return newtonMethod(f, fDerivative, x1, epsilon);
}
newtonMethod(
  x => x**5+3.5*x**4-2.5*x**3-12.5*x**2+1.5*x+9,
  x => 5*x**4+14*x**3-7.5*x**2-25*x+1.5,
  4,
  1e-10
);
```

Newton-raphson metode og rekursionsligninger:
Ligesom en rekursionsligning implementeres Newtons metode med en funktion kaldt newtonMethod der tager 4 parameter. De 4 parameter er funktionen f ved x0, afledte funktion fDerivative ved x0, første gæt x0 og vores epsilon tal. Funktionen bruger rekursion til at gentagende gange opdatere gættet på roden, indtil en tilstrækkelig præcision opnås. Denne tilgang er i overensstemmelse med teorien om rekursionsligninger, hvor gentagne opdateringer af variabler bruges til at konvergere mod en løsning.

Tangentens ligning og numerisk tilnærmelse:
I Newton-Raphson metoden er tangentens ligning essentiel. I koden bruges tangentens ligning til at finde det næste gæt x_1 som tilnærmer sig nulpunktet på grafen. dermed afspejler den empiriske implementering nøje teorien om tangentens ligning og dens rolle i numerisk tilnærmelse af rødderne på grafen.

Præcision og iteration:

I koden bruges der epsilon. Epsilon er den parameter som specificerer præcisionen for løsningen. Algoritmet fortsætter med at lave rekursion indtil en præcision på epsilon er mødt på y-aksen når y går mod 0. Grunden til at det er vigtigt er fordi præcisionen er kilden til pålideligheden af metoden, da en mere præcis løsning giver et mere pålideligt svar.

Sammenligning af programmeringssprog og ydeevne:
Uden at det specifikt går over empiri specifikt på baggrund af teori er det stadig vigtigt at pointere hvor vigtigt sammenligningen af hastigheden og effektiviteten af algoritmen på tværs af forskellige programmeringssprog er. Denne sammenligning er relevant i forståelsen af, hvordan valget af sprog og dets egenskaber kan påvirke den resultere hastighed af ens program. På baggrund af teori burde programmeringssprog gøre det lige hurtigt hvis man ser på mængder af iteration, men ift. Forskellen af hvordan sprogene udøver instruktionerne er der mere end bare teori at kigge på, og derfor er det vigtigt at få det med.

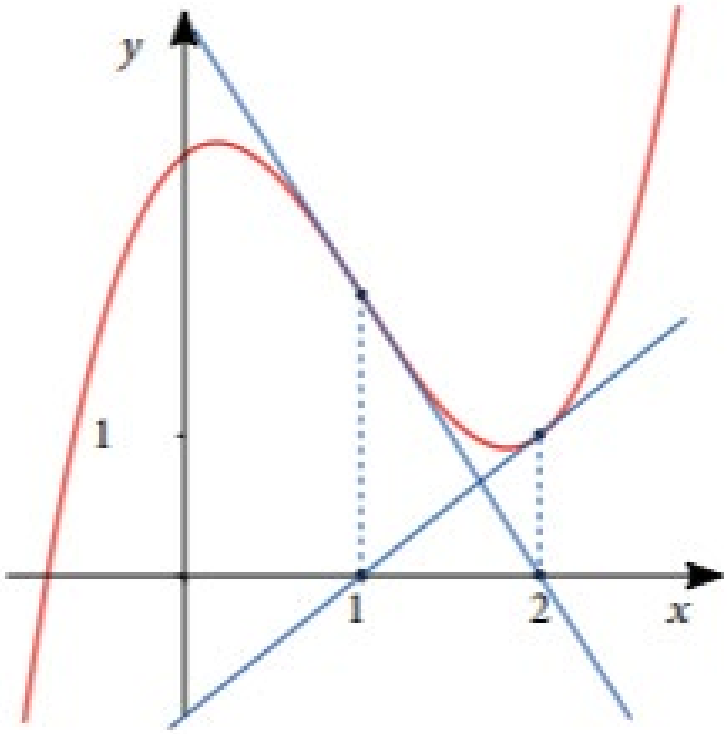
- Diskussion

Min implementation af Newton-Raphson metoden i programmering har flere forskellige styrker og svagheder som jeg nu vil vise med en tabel.

Styrker	Svagheder
Simplicitet. Implementationen er simpel og nem at forstå, med in-line kommentarer som forklarer processen.	**Stackoverflow**: Da der ikke direkte er implementeret en maks iterations tæller kan det lede til uendelig iteration (hvis programmeringssproget ikke har indbygget stackoverflow exception)
Rekursion. Brugen af rekursion for at gentage processen indtil roden er fundet er lige til pas med rekursion da den kalder tidligere brugt elementer.	**Præcision:** Da det implementeret algoritme stopper når den absolutte værdi af fx er mindre end epsilon, kan det ikke garanteres at x er roden af f, men kun at f(x) er tæt på 0. Roden kan være længere væk afhængig af formen på grafen.
Flexibilitet. Funktionen tager parameter som f og fDerivative som bruger selv kan indtaste og få resultatet på selvvalgt funktion. Samt ændring af gæt og præcision.	**Konvergens:** I nogle tilfælde hvor x_0 er for langt væk eller funktionen har flere rødder konvergerer metoden måske ikke til en rod. I stedet kan gættene svinge mellem to værdier eller divergere til det uendelige eller langt væk fra rødderne og derfor gøre kører tiden af programmet meget længere eller skabe stackoverflow problemer. Følgende har Newton Raphson også limitationer som vist understående på figur 1.3.

Den overstående tabel præsenterer styrker og svagheder for min implementation af Newton Raphson metoden.

(Ministeriet for børn, 2016)



Figur 1.3 viser newtons metode som kan fejle i visse tilfælde hvor en konvergering ikke opstår.

Perspektivering

For at kigge på perspektiver ift. Implementationen og algoritmet bliver vi nødt til at kigge på anvendelsen i den virkelige verden, resultater på hastighed og anvendelsesområder ift. Programmeringssprog. Derfor perspektiverer jeg også med andre metoder til at finde rødder.

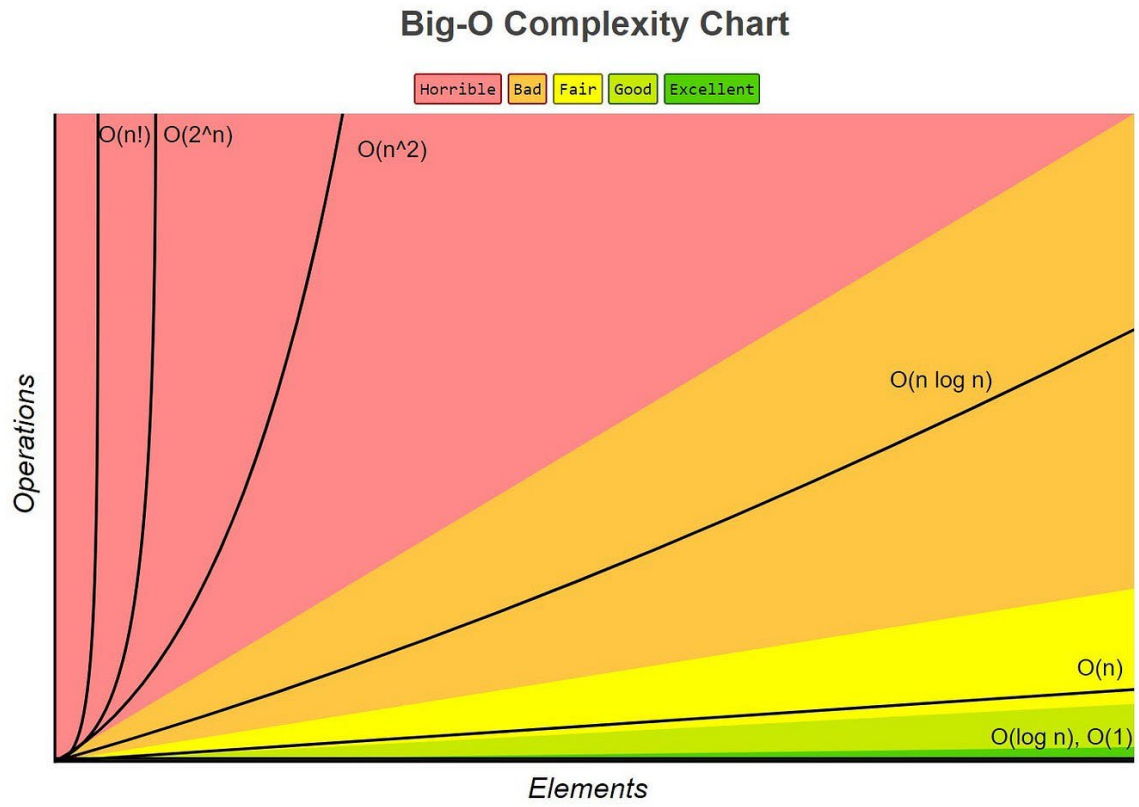
Anvendelsen i den virkelige verden:
Newton Raphson metoden bruges i mange forskellige felter udover programmering Implementationen. Metoden bruges også i statistik, anvendt matematik, numerisk analyse, økonomi, finansiere, management og marketing osv. Som er skrevet i ”Advances in Decision Science” som vist af nedenstående citat fra deres introduktion.

”including statistics, applied mathematics, numerical analysis, economics, finance, management, and marketing, among others” (McAleer, 2019)

Resultater på hastighed ift. Andre Implementationer i virkeligheden:

Min Implementationen i JavaScript, Python og Rust giver et godt eksempel på hvordan metoden vil agere i virkeligheden da Implementationen er meget universel. Min Implementationen er brugt til at finde roden af et polynomie via et gæt på en graf.
Mine resultater på hastighed kan findes i de her bilag: benchmark_data_javascript.xlsx og benchmark_data_python.xlsx som kan analyseres med analyse filerne i mit projekt Analysis_javascript.py og analysis_python.py til deres respektive filer.

Hvis vi skal perspektivere til tids kompleksiteten af algoritmet kan vi se på den nedenstående figur 1.4
(Prado, 2019)



[4] Figur 1.4 viser Big-O tids kompleksitet

1. O(1) - Konstant tidskompleksitet: Operationer tager samme mængde tid uanset inputstørrelsen.

2. O(log n) - Logaritmisk tidskompleksitet: Operationstiden vokser logaritmisk med inputstørrelsen. Eksempel: Binærsøgning i en sorteret liste.

3. O(n) - Lineær tidskompleksitet: Operationstiden vokser lineært med inputstørrelsen. Eksempel: Lineær søgning gennem en usorteret liste.

4. O(n log n) - Lineærithmisk tidskompleksitet: Operationstiden vokser logaritmisk med inputstørrelsen, multipliceret med en lineær faktor. Eksempel: Mange effektive sorteringsalgoritmer som fletningssortering og quicksort.

5. O(n^2) - Kvadratisk tidskompleksitet: Operationstiden vokser kvadratisk med inputstørrelsen. Eksempel: Indlejrede løkker, der itererer over en todimensional liste.

6. O(n^c) - Polynomisk tidskompleksitet: Operationstiden vokser som en polynomisk funktion af inputstørrelsen, hvor c er en konstant.

[4] https://miro.medium.com/v2/resize:fit:720/format:webp/1*5ZLci3SuR0zM_QlZOADv8Q.jpeg (tilgået 23/02 2024 13:00)

7. O(2^n) - Eksponentiel tidskompleksitet: Operationstiden fordobles med hver tilføjelse til inputstørrelsen. Eksempel: Brute-force søgealgoritmer.

8. O(n!) - Fakultet tidskompleksitet: Operationstiden vokser faktorielt med inputstørrelsen. Eksempel: Permutations- eller kombinationsalgoritmer.[5] (Wikipedia, 2024)

Overstående har vi eksempler på de mest basale tids komplexiteter fra Big-O Complexity chart. De beskriver hvordan forskellige tider af algoritmeanalyse og repræsentere forskellige væksthastigheder ift. Input størrelse. Det kan vi så bruge til at beskrive Newton Raphson metoden, men det er ikke så lige frem, som bare at give den en titel som "O(1)" da det er mere kompleks end det.

Tidskompleksiteten af Newton-Raphson metoden afhænger typisk af, hvor hurtigt den konvergerer mod løsningen og det initiale gæt. Som vi kan læse på siden "Newton's method" (Newton's method, 2009) under "Computational complexity" forklares der at hvis vi har et godt startgæt, er kompleksiteten $O((\log n)F(n))$, hvor $F(n)$er omkostningen ved at beregne $f(x)/f'(x)$ med vores n-cifret præcision i epsilon.

Men, hvis funktionen kan evalueres med variabel præcision, kan vi forbedre algoritmen ved at bruge en passende mængde af præcisions trin ift. Grafen. I det man gør det, reduceres kompleksiteten til $O(F(n))$ hvor det er forudsat at den vokser superlineart. Superlineær voksning betyder bare at når n stiger, vokser $F(n)$ hurtigere end n selv. Som så skal betyde at i praksis, jo større input størrelsen bliver, jo mere stiger omkostningerne for operationen. Det her har stor indflydelse på ydeevnen af algoritmet, da omkostningerne (tid og plads) kan stige hurtigt og dermed påvirke den samlede køretid af programmet.

Sammenligning af andre metoder:
Der findes selvfølgelig en alternativ række af metoder til at finde nulpunkter af funktioner, og derfor vil jeg nu herunder beskrive nogle af dem, med perspektivering til Newton Raphson.

Bisektionsmetoden $O(log\ n)$: metoden er brugt til at garantere konvergering, men er generelt langsommere end Newton-Raphson metoden. (GeeksforGeeks, 2021)

Sekantmetoden $O(log\ n)til\ O(n)$: Denne metode er hurtigere end bisektionsmetoden, men den kan konvergere langsomt eller slet ikke konvergere for visse funktioner. Og er derfor tættere på Newton raphson metoden. (GeeksforGeeks, 2021)

Regula falsi metoden $O(log\ n)til\ O(n)$: Denne metode er en sammensmeltning af bisektionsmetoden og sekantmetoden. Den er hurtigere end bisektionsmetoden og mere robust end sekantmetoden. (GeeksforGeeks, 2021)

[5] https://en.wikipedia.org/wiki/Big_O_notation (tilgået 23/02 2024)

Sammenlignet med Newton Raphson metoden der typisk har tidskompleksitet på $O((\log n)F(n))$hvor $F(n)$ er omkostningen ved at beregne $f(x)/f'(x)$med n cifret præcision af epsilon, kan vi se at disse andre metoder har lignede tidskompleksitet, men Newton Raphson metoden kan være mere flexibel og effektiv i visse tilfælde når præcision er vigtig.

- Konklusion

Newton Raphson metoden, en effektiv algoritme til at finde rødderne af en reelværdi funktion. I denne rapport har vi dykket ned i Newton Raphson metoden ved at kombinere teoretisk forståelse med praktisk implementering i programmeringssprog som JavaScript, Python og Rust hvor vi har undersøgt metoden fra flere vinkler og identificeret både styrker og svagheder ved Implementationen.

Newton Raphson metoden udnytter brugen af rekursion og tangentens ligning til at iterativt tilnærme sig en rod eller rødderne af en funktion. Implementationen af metoden i programmering demonstrerer og præsentere hvordan teorien omsættes i praksis, og hvordan valget af sprog og implementeringsdetaljer kan påvirke ydeevnen af algoritmen.

I gennemgangen af kompleksitet og styrker / svagheder har vi set, at metoden er fleksibel og kan tilpasse forskellige funktioner og præcision. Dog er det vigtigt at være opmærksom på forskellige problemer som konvergeringsproblemer gennemgået i styrker og svagheder og behovet for et passende startgæt.

Derudover har vi diskuteret sammenligningen af hastighed og effektivitet på tværs af forskellige programmeringssprog som giver os en god indsigt i Implementationen i forskellige sprog, og hvordan Implementationen kan se ud i den virkelige verden.

Ved at perspektivere har vi også fået indsigt i hvilke områder metoden er anvendt såsom statistik, økonomi og numerisk analyse, og dens tidskompleksitet ift. Andre metoder brugt såsom bisektionsmetoden, sekantmetoden og regula falso metoden for at belyse alternative styrker og svagheder som komparativt til Newton Raphson metoden.

Samlet set har denne rapport indvirket i at give et dybdegående indblik i Newton Raphson metoden, fra dens teoretiske grundlag til dens praktiske implementering i programmering og anvendelse og sammenligning med andre metoder. Ved at kombinere teori og praksis har vi opnået en dybere forståelse af algoritmens potentiale og begrænsninger, samt hvilke faktorer der skal overvejes ved valg af metode og implementering. Du føres nu hen til GitHub artiklen skrevet af Malthe Bang Norengaard for en omfattende gennemgang af opsætning af programmet til eget brug. (Norengaard, 2024) Og alt kode og andet understøttende materiale kan findes som bilag på Lectio under navn ”newton-raphsons-root-finder-master.zip”.

- Litteraturliste

Referencer

GeeksforGeeks. (2021, 12 31). *Difference between Newton Raphson Method and Regular Falsi Method*. Retrieved from geeksforgeeks: https://www.geeksforgeeks.org/difference-between-newton-raphson-method-and-regular-falsi-method/

McAleer, M. (2019, 12 23/02/2024). *Applications-of-the-Newton-Raphson-Method-in-Decision-Sciences-and-Education*. Retrieved from iads: https://iads.site/wp-content/uploads/papers/2019/Applications-of-the-Newton-Raphson-Method-in-Decision-Sciences-and-Education.pdf

Ministeriet for børn, u. o. (2016). Matematik A Højere teknisk eksamen forberedelsesmateriale. In m. f. ligestilling, *Matematik A* (p. 17). Ministeriet for børn undervisning og ligestilling.

Newton's method. (2009, 6 11). Retrieved from citizendium: https://en.citizendium.org/wiki/Newton%27s_method#Computational_complexity

Norengaard, M. B. (2024, 2 23). *Newton-Raphson Method Implementation*. (M. B. Norengaard, Producer, & mbn-code.dk) Retrieved 2 23, 2024, from github: https://github.com/mbn-code/newton-raphsons-root-finder

Prado, K. S. (2019, mar 4). *Understanding time complexity with Python examples*. Retrieved from towardsdatascience: https://towardsdatascience.com/understanding-time-complexity-with-python-examples-2bda6e8158a7

Wikipedia. (2024, feb 4). *Big_O_notation*. Retrieved from wikipedia: https://en.wikipedia.org/wiki/Big_O_notation