```
// NEWTON-RAPHSON
template <class T>
auto inline newton_raphson(const T value)
    const noexcept -> decltype(Fn(value))
{
    // CALCULATE NEW VALUE FROM THE FORMULA
    // N_A+1 = N_A - F(N_A)/F'(N_A)
    const auto new_value = value - (Fn(value) / thi

    auto has_acceptable_precision = [](T x, T y) ->
    {
        return std::abs(x - y) < PRECISION;
    };

    // RETURN WHEN RESULT IS PRECISE ENOUGH
    if (has_acceptable_precision(value, new_value))
        return new_value;

    // ITERATE
    return this->newton_raphson(new_value);
}

// BISECTION
template <class T>
auto inline bisection(std::pair<T, T> interval)
    const noexcept -> decltype(Fn(interval.first))
{
    // UNPACK INTERVAL USING STRUCTURED BINDINGS
```

STUDIERETNINGSCASE

af Carl Schou

NUMERISKE METODER

Bisektion, Newton-Raphson og numerisk differentiering analyseres, vurderes og sammenlignes. Der bliver sat fokus på køretider, hukommelsesforbrug, tidskompleksitet, operationer og effektivitet. Ligeledes eksamineres, udregnes og opbygges konkret empiri, til brug i de førnævnte numeriske metoder, således at resultatet så godt som mulig afspejler de analytiske realiteter.

Indholdsfortegnelse

Referat

Denne studieretningscase vil, med henblik på teknisk gennemgang, vurdere og bearbejde numeriske metoder, samt sammenligne dem med deres analytiske modpol. De analytiske metoder er i vidt brug til daglig, da de oftest er mere ligetil end deres numeriske modpoler, men de er således også svære at integrere i et computersystem, der ikke i samme omfang har en matematisk kognition. Denne mangel på grundlæggende intelligens lægger grundlag for en hel ny fremgangsmåde indenfor matematik, nemlig numeriske metoder. Denne form for praksis fokuserer på at udforme det samme udbytte som de analytiske metoder, dog udelukkende ved brug af numeriske, matematiske operationer. Det gør det muligt for computerprogrammer at implementere disse algoritmer, og derfor udføre krævende matematisk praksis, uden at udvikle avancerede analytiske systemer. Jeg vil kontinuerligt analysere to fremherskende, analytiske metoder: *Bisektion* og *Newton-Raphson*, der begge bliver brugt til nulpunktsbestemmelse af arbitrære udtryk. Den numeriske metode Newton-Raphson kræver en implementering af en øvrig numerisk metode: *numerisk differentiering*.

Med fokus på teknisk analyse, numerisk præcision og matematisk anvendelse vil jeg hermed eksaminere følgende udtryk (opregnet for fremtidig reference):

$$1: x^2$$

$$2: x^3 - x - 2$$

$$3: \sqrt[3]{x}$$

Disse tre udtryk vil blive gennemkørt i forskellige miljøer såvel som med forskellig metode, for at anslå ikke bare metodernes henholdsvise effektivitet men også de tekniske aspekter og matematiske begrænsninger som medfølger de førnævnte udtryk. For at programmatisk vurdere en algoritme på et lukket computersystem, bruges ofte disse nøglebegreber: *tidskompleksitet*, *hukommelsesforbrug*, *antal operationer* og *køretider på et udvalgt system*. Der vil i nedenstående sider fremvises forskellige iterationer af det henholdsvise computerprogram, der bliver brugt til at udføre disse tekniske analyser, for at fremhæve optimeringer, forbedringer og ændringer.

Formålet med denne studieretningscase er derfor at præsentere forfatterens evner, både matematiske, videnskabelige og tekniske, ved at fokusere på anvendelse af fagligt relevant teori såvel som empiri. Der bliver lagt vægt på, at forfatteren kan producere og anvende en korrekt softwareløsning, der løser den tilstedeværende problemstilling, og derefter kan dokumentere sin faglighed der afspejler sig i projektet.

Indledning

Valg af emne

Jeg har valgt at have mit primære fokus på de relevante numeriske nulpunktbestemmelsesmetoder, Bisektion og Newton-Raphson, da deres formål og relevante udbytte er simpelt at forholde sig til, selv for lægmænd. Udover dette, har jeg valgt dynamisk assimilation af numerisk differentiering, for at udelukke et potentielt inputparameter i den førnævnte, numeriske metode Newton-Raphson.

Der er blevet taget et valg om hvilke *cases* og miljøer disse algoritmer skal testes i, for at samle videnskabelig empiri, så at gyldig konklusionsudledning er mulig. De tre *cases* er valgt med matematisk baggrund som omtanke. Dermed kan udtrykkene afspejle et bredt matematisk billede, så de numeriske algoritmer bliver afprøvet i forskellige områder af det matematiske spektrum. Jeg har taget dette valg, på baggrund af min nuværende, retrospektive viden om algoritmerne, og jeg føler dermed at den førnævnte afgørelse kan fremkalde relevant teknisk empiri, som kan bruges til at afklare effektiviteten af de henholdsvise metoder.

Problemstilling

Hvordan kan anvendelsen af numeriske metoder bruges til at tilnærme sig nulpunktet for et givent udtryk, og hvor effektive er disse numeriske metoder, både singulært såvel som kontradiktorisk?

Valg af arbejdsmiljø

Jeg har valgt at udarbejde mit konkrete produkt, i form af et softwareprogram, i det omtalte programmeringssprog C++. Denne beslutning er taget på baggrund af min nuværende kompetencer indenfor det tekniske univers, hvor jeg blandt andet har arbejdet med det førnævnte programmeringssprog i et så stort omfang, at jeg sætter stor vægt på min tekniske beføjelser. Sproget har været fremherskende i årtier længere end jeg har levet, og fortsætter med at være en primær drivkraft indenfor tekniske gennembrud. C++ forøger således kontrollen over udfaldet af programmet, hvilket i dette tilfælde åbner op for bundløse optimeringsmuligheder når det kommer til optimering af matematiske såvel som programmatiske softwarealgoritmer. Dette valg gør det muligt for mig at udarbejde polymorfe softwareløsninger der kan blive tilpasseligt optimeret af moderne kompilationsredskaber, således at opretholderen af programmet ikke behøver at lave omfattende restrukturering hvis de tekniske krav om både præcision såvel som optimering ændres. Dette vil i sidste ende øge effektivitet og have store økonomiske gevinster.

Programmet er udarbejdet således, at det kan blive kompileret af enhver C++-standard-kompatibel, der understøtter den nyeste, færdige version: 17. Version 17 af programmeringssproget C++ tilføjer fundamental funktionalitet der grundlæggende ændrer hvordan man kan udarbejde programmer. Jeg har således valgt at færdiggøre de fremadnævnte iterationer i kompileringsværktøjet Clang, der er en front-end for den verdensomtalte softwareløsning LLVM (University of Illinois, 2019). Dette valg er blevet taget på baggrund af faglig erfaring med det førnævnte værktøj. Clang er efter min mening det bedste, offentligt tilgængelige kompileringsværktøj til sproget C++, grundet sin meget nøjeregnende og puritanske eftergivenhed. Værktøjet tillader for moderne optimeringsmuligheder, blandt andet indenfor numeriske algoritmer, og derfor føler jeg at det er det bedst egnede redskab.

Teori

Bisektion

Bisektion er en numerisk metode, der bruges til at finde nulpunktet af et arbitrært, kontinuert udtryk. Funktionalitet består af at dele et givent interval op i to lige store dele, og analysere fortegnet hvor de to dele mødes i forhold til intervalværdiernes henholdsvise fortegn, indtil nulpunktet er fundet. Metoden er fremherskende på grund af dens simple fremgangsmåde, men den har sine begrænsninger. Det arbitrære udtryk skal have mindst en rod, og det givne interval skal være på hver sin side af roden, for at analysen kan tage sted. Hvis udtrykket ikke indeholder en rod, eller roden ikke har et andet fortegn end intervalværdierne, vil bisektionsmetoden vedblive uendeligt; i scenariet med ens fortegn, kan roden udregnes så længe at den er til stede præcist i midten af de to intervalværdier.

Denne metode bruges altså til at løse problemer, hvor man skal finde nulpunktet for en given funktion, som for eksempel:

$$isoler\ x\colon f(x) = 0$$

I følgende gennemgang bruges udtryk 2 fra de førnævnte *test cases*:

$$f(x) = x^3 - x - 2$$

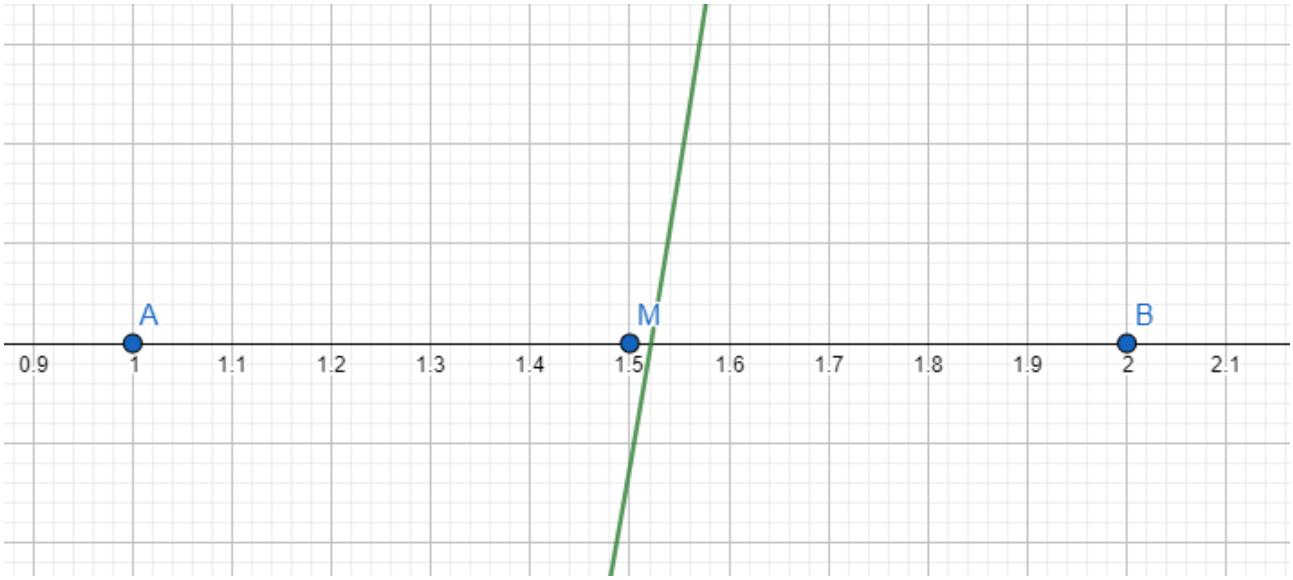
Jeg har, som nødvendigt, angivet at der ligger en rod mellem intervallet $[1, 2]$ i det førnævnte udtryk. Dette tillader os at finde den præcise rod der finder sted i dette interval. Første trin i en iteration er at udregne midtpunktet i intervallet, altså deler jeg intervallet op i to lige store dele. Dette gøres med den simple matematiske formel:

$$m = \frac{a+b}{2}, \qquad hvor\ a\ og\ b\ er\ intervalværdier$$

I vores eksempel vil det første midtpunkt være:

$$m = \frac{1+2}{2} = \frac{3}{2}$$

Her ses en illustration af intervalværdierne (A, B) og midtpunktet (M):



Figur (1)

Næste trin i iterationen er at sammenligne fortegn for midtpunktet og startværdien i intervallet:

$$start = f(1) = -2$$

$$m_{værdi} = f\left(\frac{3}{2}\right) = -1/8$$

Jeg kan så ud fra observation konkludere at fortegnet ikke har ændret sig, og jeg kan derfor som sidste trin opdatere værdierne i intervallet for at indsnævre vores søgeområde.

Hvis midtpunktets fortegnet er ens med start-værdien, altså den første værdi i intervallet, ændres start-værdien til midtpunktet. I det andet scenarie, hvor fortegnet har ændret sig, ændres slut-værdien, altså den sidste værdi i intervallet. Derfor vil vores nye interval være:

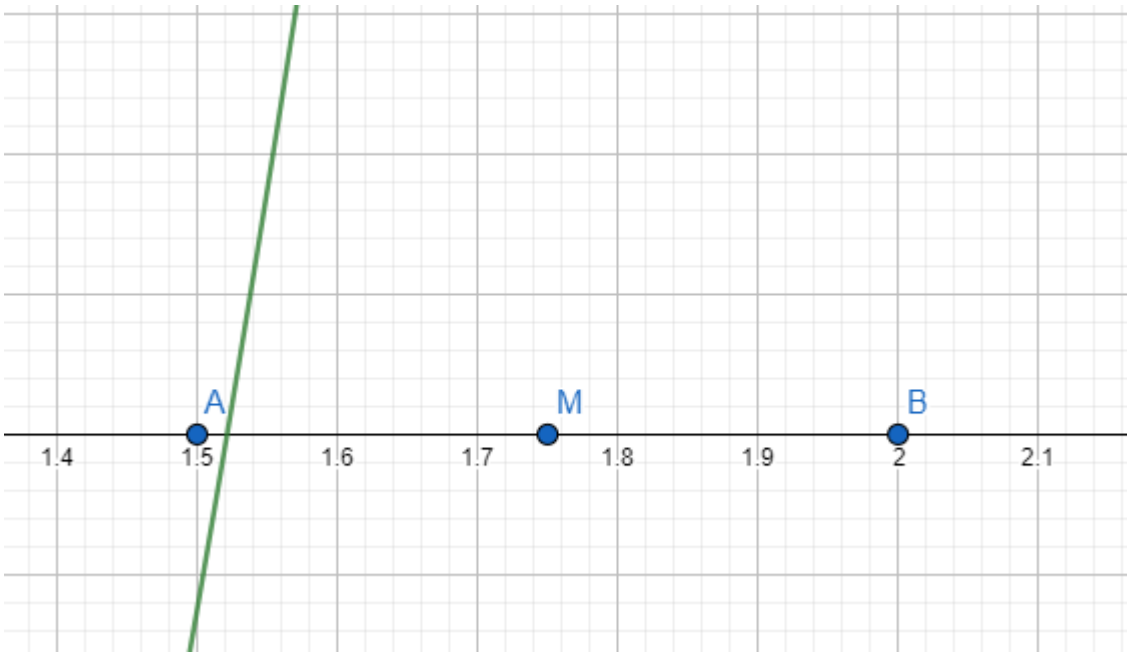
$$\left[\frac{3}{2}, 2\right]$$

Jeg kan derefter begynde en ny iteration af samme fremgang:

Midtpunkt udregnes:

$$m = \frac{\frac{3}{2} + 2}{2} = \frac{7}{4}$$

Her ses en opdateret illustration af det nye interval og dets henholdsvise midtpunkt:



Figur (2)

Udtryk udregnes for midtpunkt og start-værdi:

$$f\left(\frac{7}{4}\right) = 1{,}61$$

$$f(2) = 4$$

Her kan jeg se at fortegnene er ens, hvilket vil sige at jeg har passeret en rod. Derfor opdaterer jeg intervallet til:

$$\left[\frac{3}{2}, \frac{7}{4}\right]$$

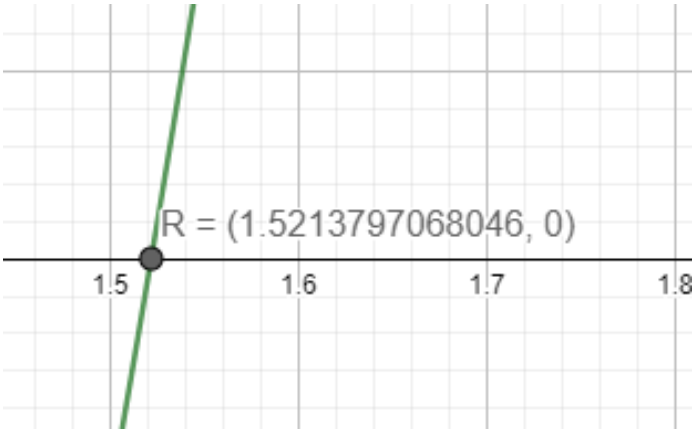
Man kan herefter fortsætte med bisektionsmetoden indtil man finder den præcise rod, hvilket i sidste ende vil være:

$$f(x) = 0$$

⇳ *Ligningen løses for x vha. CAS-værktøjet WordMat.*

$$x = 1{,}52137970680457$$

Som også kan vises i vores førnævnte illustration:



Figur (3)

Bisektion kan i teorien bruges til at udregne fuldstændig præcise roder, men hvis hver iteration skal gøres i hånden, tager dette exceptionelt lang tid at udføre, hvilket gør dem perfekte til automatisering ved hjælp af computerprogrammer. Præcisionen af afkommet kan udregnes ved at tage den absolutte værdi af intervallets delta, og man kan således udregne om ens resultat er præcist nok:

$$|a - b| < p, \qquad hvor\ p\ er\ præcision\ som\ decimaltal$$

Her er forudsætningen, at man ved bisektionsmetoden har fundet frem til den korrekte rod, og vil derfor ikke virke i andre sammenhæng. Hvis jeg for eksempel ønsker et resultat med 15 betydende cifre, bruges formlen således:

$$|a - b| < 0{,}000000000000001$$

Denne metode vil blive fulgt op på i det tekniske afsnit af implementeringen af bisektion i software-løsningen.

Newton-Raphson

Newton-Raphson, også kendt som Newtons-metode, er en numerisk algoritme brugt til at finde akkurate nulpunkter for en given funktion. Newton-Raphson er en iterativ metode der ved hver iteration resulterer i et mere præcist resultat, målt i betydende cifre. Metoden er indrettet således, at den tilnærmer sig nulpunktet ved at tage den nuværende hældnings skæring med x-aksen, hvor første værdi er givet, indtil arbitrær præcision er opnået. Denne algoritme er derfor mere kompliceret end bisektion, hvor man bare halverer og sammenligner fortegn, men opnår eksponentielt højere hastigheder i visse tilfælde, i modsætning til bisektion. Newton-Raphson afhænger derfor også mere på hvilket udtryk der analyseres, i modsætning til bisektion, da hældningen indgår som et variabel, men dette bringer også adskillige skavanker, da udtryk med blandt andet uendelig hældning ikke kan analyseres.

Metodens implementering er simpel:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Hvor $f(x)$ er det givne udtryk, og $f'(x)$ er den henholdsvis afledte funktion. x_n refererer til den nuværende iterations værdi, og vil i første tilfælde være den arbitrære startværdi.

I nedenstående gennemgang bruges udtryk 2 fra de angivne test cases:

$$f(x) = x^3 - x - 2$$

Først er jeg nødt til at vælge en startværdi, og her har jeg konservativt valgt værdien 10. Den første iteration vil derfor se sådan ud:

$$x_1 = 10 - \frac{f(10)}{f'(10)} = 6{,}7$$

I den næste iteration indsætter jeg så den førudregnede værdi ind, med samme fremgangsmåde:

$$x_2 = 6{,}7 - \frac{f(6{,}7)}{f'(6{,}7)} = 4{,}5$$

Og jeg fortsætter så indtil jeg observerer en mangel på ændring af betydende cifre:

$$x_3 = 4{,}5 - \frac{f(4{,}5)}{f'(4{,}5)} = 3$$

$$\Updownarrow$$

$$x_4 = 3 - \frac{f(3)}{f'(3)} = 2{,}15$$

$$\Updownarrow$$

$$x_5 = 2{,}15 - \frac{f(2{,}15)}{f'(2{,}15)} = 1{,}7$$

$$\Updownarrow$$

$$x_6 = 1{,}7 - \frac{f(1{,}7)}{f'(1{,}7)} = 1{,}54$$

$$\Updownarrow$$

$$x_7 = 1{,}54 - \frac{f(1{,}54)}{f'(1{,}54)} = 1{,}52164$$

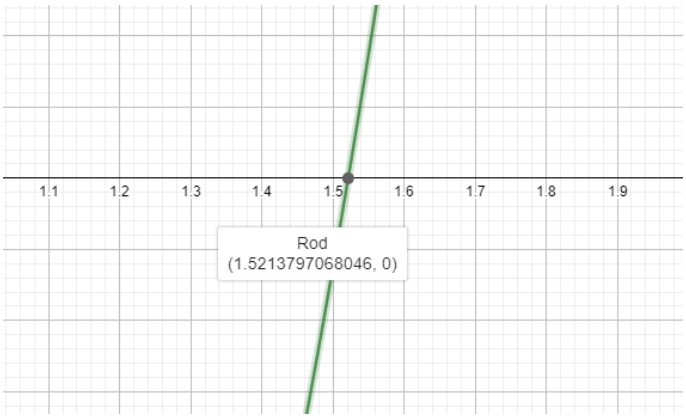
$$\Updownarrow$$

$$x_8 = 1{,}52164 - \frac{f(1{,}52164)}{f'(1{,}52164)} = 1{,}52137$$

Har kan jeg så konkludere at ændringen mellem de to førudregnede iterationer er påpasselig præcis:

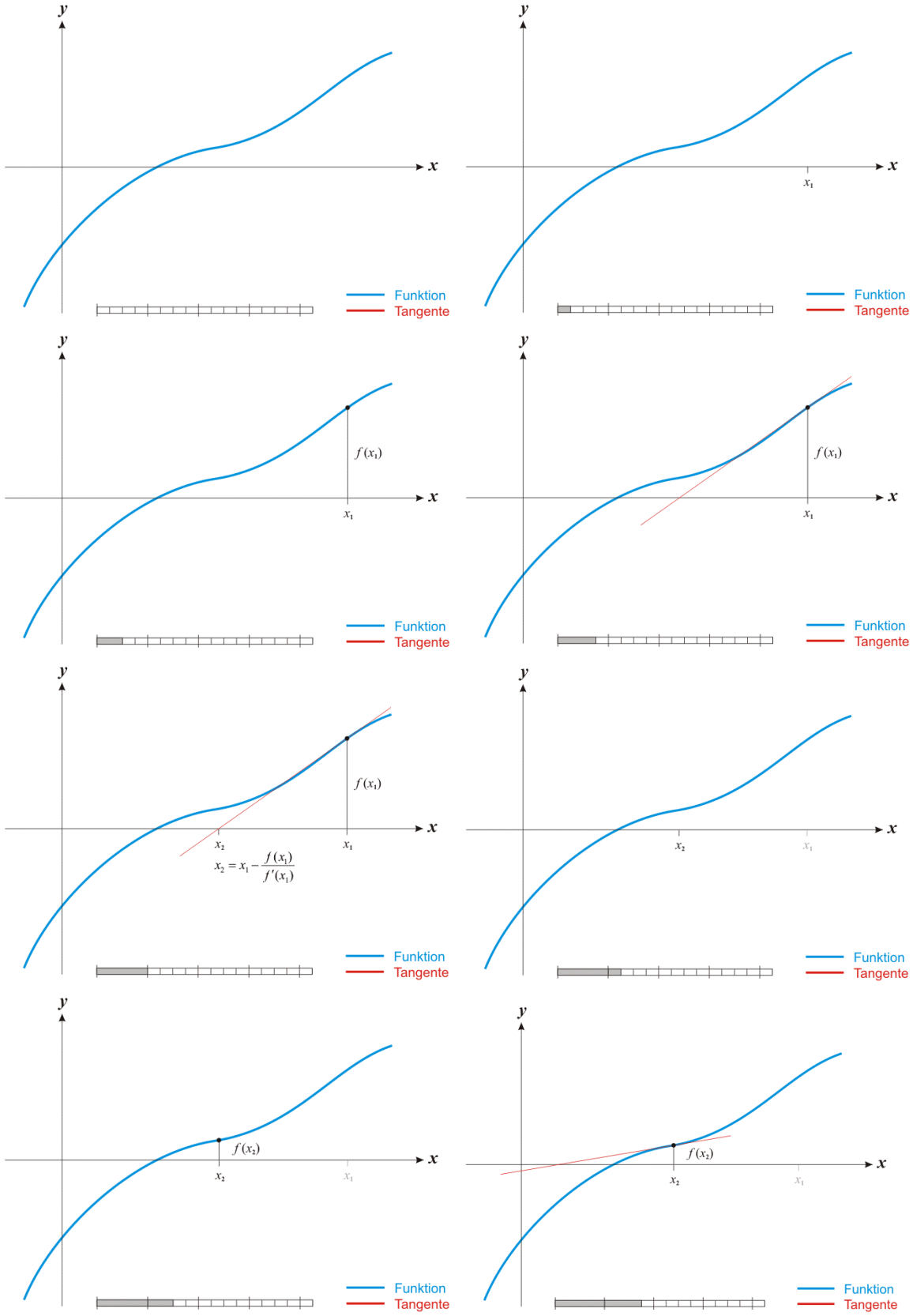
$$\Delta x = |1{,}52137 - 1{,}52164| = \mathbf{0{,}000}27$$

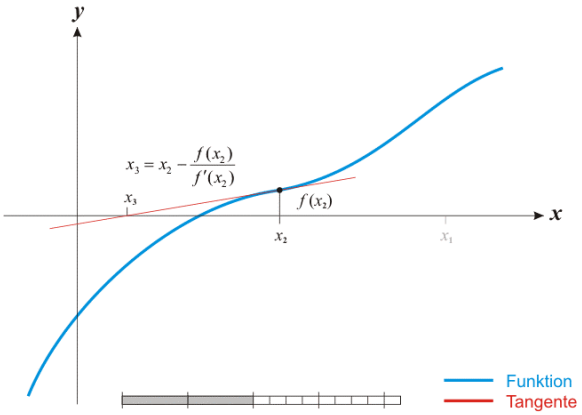
Jeg kan herfra udlede at vores nuværende resultat er udtrykkets henholdsvise nulpunkt med **4** betydende cifres præcision, og dette kan bekræftes analytisk ved hjælp af en visualisering:

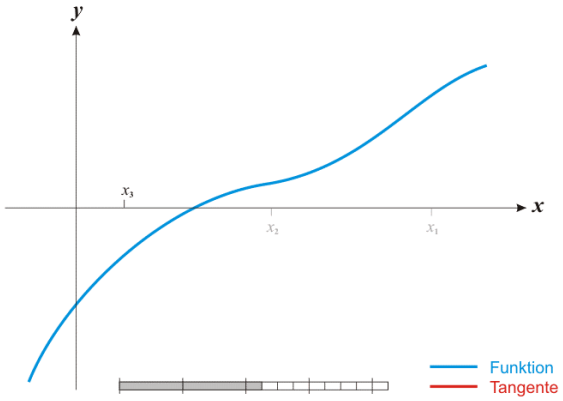


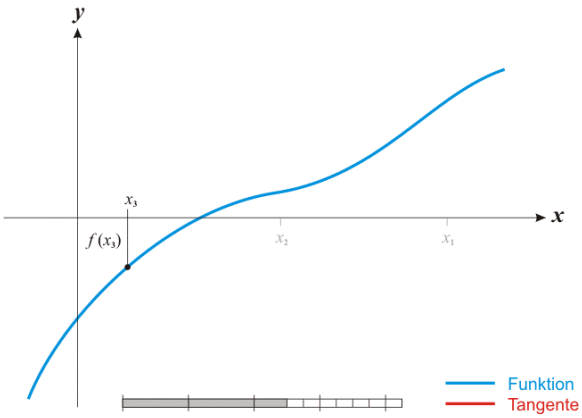
Figur (4)

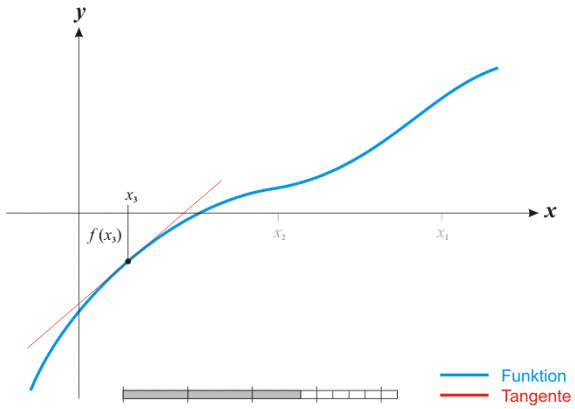
Herunder ses en animation af Newton-Raphson for en given funktion, der viser hvordan roden bliver anslået ved hjælp af hældningen af kurven (Pfeifer, 2005):

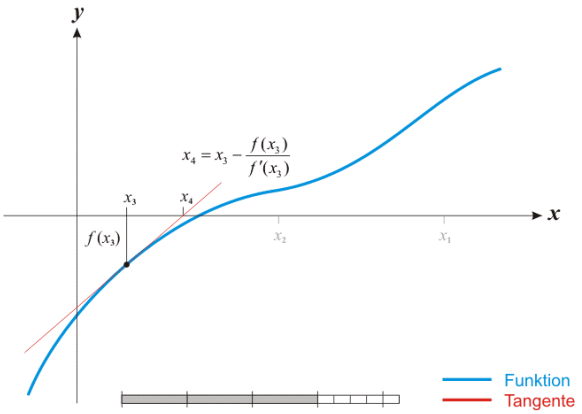


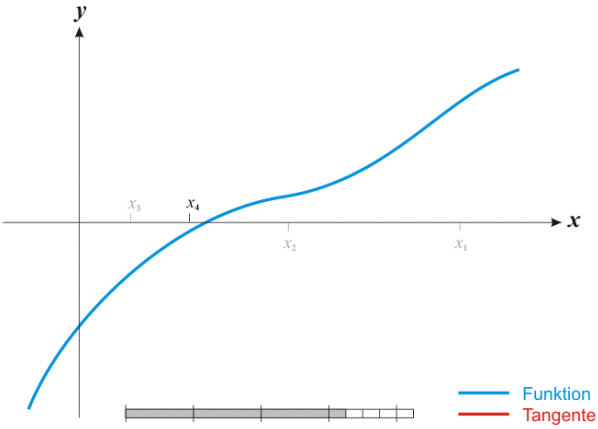

y
x
$x_3 = x_2 - \frac{f(x_2)}{f'(x_2)}$
x_3
$f(x_2)$
x_2
x_1
Funktion
Tangente

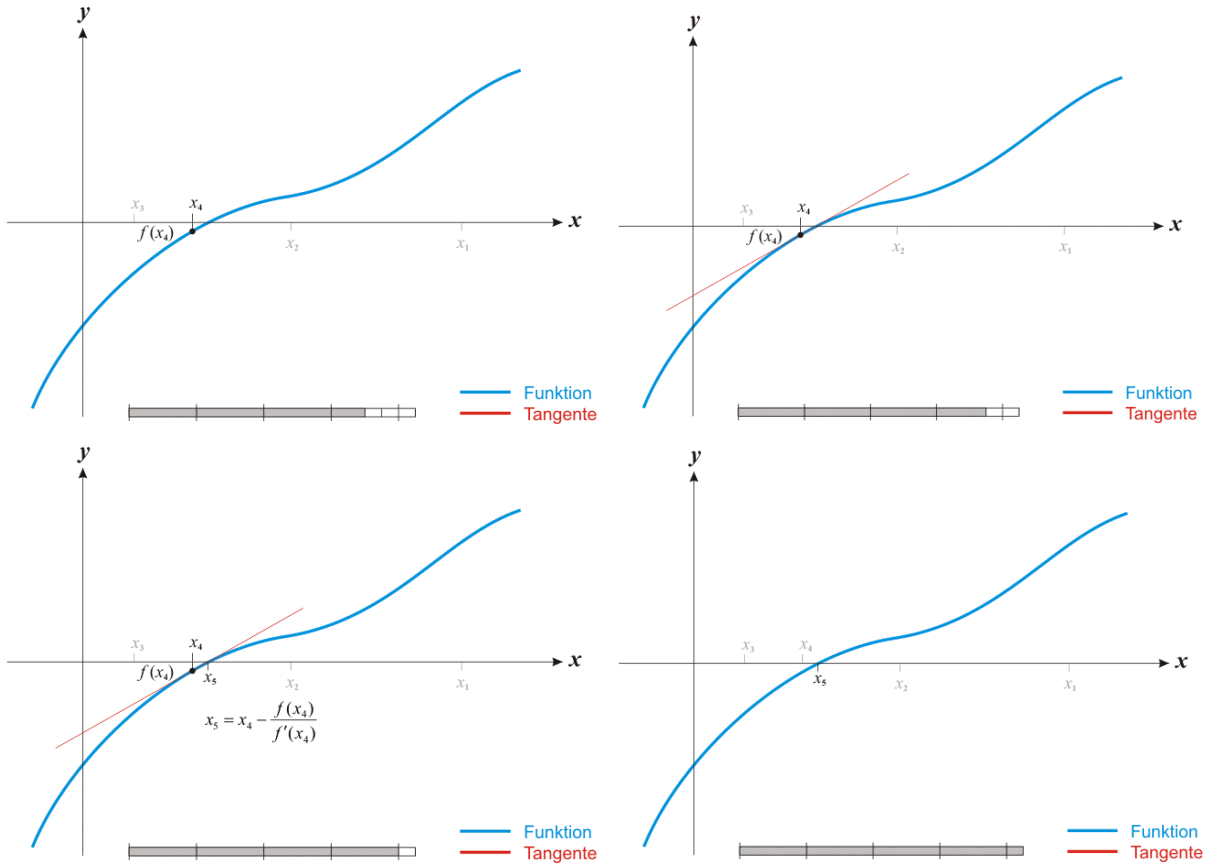

y
x
x_3
x_2
x_1
Funktion
Tangente


y
x
x_3
$f(x_3)$
x_2
x_1
Funktion
Tangente


y
x
x_3
$f(x_3)$
x_2
x_1
Funktion
Tangente


y
x
$x_4 = x_3 - \frac{f(x_3)}{f'(x_3)}$
x_3
x_4
$f(x_3)$
x_2
x_1
Funktion
Tangente


y
x
x_3
x_4
x_2
x_1
Funktion
Tangente



Figur (5)

Numerisk differentiering

At differentiere givne udtryk kræver normalt en analytisk tilgang, hvor den afledte funktion udregnes og senere kan bruges til at udregne hældningen ved et arbitrært punkt. Denne fremgangsmåde er ekstremt upraktisk for computere, og det er så her hvor numerisk differentiering kommer ind i billedet. Computere kan lave meget præcise udregninger, der gør det muligt at bruge den gamle hældningsformel til at udregne den præcise hældning af et udtryk ved et givent punkt. Hældningsformlen har været teori siden 9. klasse, og ser sådan her ud:

$$a = \frac{\Delta y}{\Delta x} = \frac{f(x+k) - f(x)}{k}$$

Man afholder sig fra numerisk differentiering hvis den analytiske fremgangsmåde er mulig, da det kan være meget svært at udregne den præcise hældning uden specifik softwareteori. Hvis jeg tager udgangspunkt i test-case 2, kan jeg sammenligne resultatet, da jeg er i stand til at udføre den analytiske tankegang:

$$Definer: f(x) = x^3 - x - 2$$

Dette udtryk kan differentieres til den afledte funktion:

$$f'(x) = 3x^2 - 1$$

Hvis jeg vælger et fuldstændig arbitrært punkt på linjen, i dette tilfælde $x = 12{,}345678$, kan jeg ved hjælp af den afledte funktion udregne hældningen:

$$f'(12{,}345678) \approx 456{,}247295839052$$

Dette er den de-facto hældning på linjen, og vil bruges som reference-værdi for vores numeriske metode:

$$Definer\colon k = 0{,}000000000001$$

$$a = \frac{f(12{,}345678 + k) - f(12{,}345678)}{k} \approx 456{,}3389666145667$$

Som observeret, kan numerisk differentiering bruges til at udregne hældningen af kurven ved et givent punkt, dog er det ikke lige så præcist som den analytiske modpart. I de næste to kapitler vil jeg forklare hvordan denne algoritme kan optimeres på et computersystem, og derfor resultere i en lige så akkurat hældning som den analytiske metode.

Floating-point

Som få mennesker ved, er computere ikke særlig gode til at repræsentere kommatal. Den menneskelige hjerne er eksponentielt mere intelligent og abstrakt, og dette kan blandt andet ses ved vores fortolkning af 'opdeling', så som $\frac{1}{3}$. Computersystemer har ikke dette lag af abstraktion, og kan derfor ikke repræsentere akkurate opdelinger af tal på samme måde som mennesker perfekt forstår at en tredjedel er 'et' delt op i tre lige store dele. For at takle dette problem, har man kommet op med en måde at repræsentere rationelle tal udelukkende ved brug af heltal.

Floating-point-værdier er derfor en tilnærmelse af det henholdsvise rationelle tal, hvor både præcision og rækkevidde, altså højeste og laveste værdi, er faktorer. Et eksempel på sådan en repræsentation er af værdien $1{,}2345$:

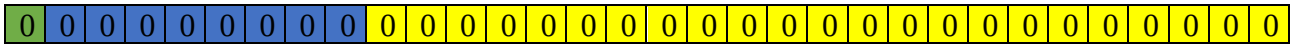
$$1{,}2345 = 12345 \cdot 10^{-4}$$

Som man kan se her, kan man altså repræsentere det førnævnte kommatal ved hjælp af tre forskellige heltal:

1. Mantissen: 12345
2. Basen: 10
3. Eksponenten: -4

Dette kræver dog at man nu gemmer tre forskellige 'tal' i hukommelsen, i modsætning til den menneskelige hjerne der direkte kan gemme $1{,}2345$ uden at skulle dele den op i tre forskellige organer.

Der findes to forskellige fremherskende floating-point implementeringer på moderne computersystemer: single- og double-præcision. Single præcision optager 32 bits i hukommelsen, altså 4 hele bytes. Strukturen i hukommelsen ser sådan her ud:



Repræsentationen her er i en tabel form, og portrætterer de 32 bits som var de i hukommelsen. Det første bit, markeret med grøn, udgør fortegnet af resultatet, også kendt som en *sign-bit*. Hvis sign-bitten ikke er sat (0), er resultatet positivt. De næste 8 bits, markeret med blå, repræsenterer eksponenten. De sidste 23 bits, markeret med gul, repræsenterer mantissen. Den opmærksomme læser havde nok bemærket at basen ikke indgår i strukturen, og dette er der en simpel forklaring på: computersystemer indgår alle i en standard, hvor de er blevet enige om at basen altid skal være to. Tallet to er meget specielt på binære computersystemer, da moderne processorer kan udføre eksponentielt hurtigere matematik med tal der går op i to. Alle hele tal der kan repræsenteres som 2^n, hvor n er et helt tal i mellem -126 og 127, kan omdannes til en single-præcisionsværdi og tilbage til heltallet uden tab af data. Disse to grænseværdier stammer fra eksponentens grænse i den førnævnte formel:

$$v = mantisse \cdot 2^{eksponent}$$

Man har som sagt 8 bits til at repræsentere eksponenten, hvilket giver os $2^8 = 256$ værdier. Grænseværdierne er så antallet af værdier fordelt lige over det negative og positive talsystem: $-126, 0, 127$ - altså 127 positive værdier, 126 negative værdier og en nulværdi; dette giver os 6 til 9 betydende cifre.

En double-præcisionsværdi er stort set ens med en single-præcisionsværdi. Den eneste afvigelse er mængden af data, og derfor mængden af præcision; en double-præcisionsværdi optager 64 bits i hukommelsen, altså 8 hele bytes. Med dette optrap af hukommelsesforbrug kommer et kæmpe gavn: langt mere præcision. Double-præcisionsværdier har 11 bits til at repræsentere eksponenten, og 52 bits til at repræsentere mantissen. Double-præcisionsværdier har således 15 til 17 betydende cifre.

Maskine-epsilon

På computersystemer med floating-pointværdier er der en meget fremherskende konstant: epsilon. Denne konstant er defineret som den øvre grænse for afrundingsfejl, og er altså den mindste værdi der kan lægges til tallet et og stadig påvirke resultatet:

$$1 + \mathrm{E} \neq 1$$

Tallet kan i praksis ikke bruges som en deltaværdi, da mantissen påvirker resultatets præcision; desto højere mantisse desto lavere præcision. Jeg kommer til at arbejde med værdier der er langt højere end 1, og maskine-epsilonen er derfor ikke tilstrækkelig.

Præcision

For at finde den optimale den optimale delta-konstant til numerisk differentiering, udarbejdede jeg en algoritme i min softwareløsning. Algoritmen er simpel:

```
std::printf("Guessing precision..\n");

// SET UP BASE DELTA
const auto expr = expression<test_case::two<math_constant::my_precision_t>>();
constexpr auto value = math_constant::my_precision_t{12.345678};                    // ARBITRARY
constexpr auto real_derivative = math_constant::my_precision_t{456.247295839052};   // CALCULATED BY WOLFRAM-ALPHA

// ESTIMATED GUESS FOR OPTIMAL DELTA VALUE
const auto estimated_delta = std::sqrt(std::numeric_limits<math_constant::my_precision_t>::epsilon());

// VARIABLE FOR OUR BRUTEFORCED, MORE PRECISE DELTA
auto bruteforced_delta = estimated_delta;

// MEASURE PRECISION TO COMPARE GUESSES
auto min_precision = math_constant::my_precision_t{ 1.0 };

// ITERATE FROM ESTIMATION +- DESIRED PRECISION
constexpr auto desired_precision = math_constant::my_precision_t{0.0000001};
for (auto i = -desired_precision;
          i < desired_precision;
          i += std::numeric_limits<math_constant::my_precision_t>::epsilon())
{
    // CALCULATE DERIVATIVE FROM CURRENT, GUESSED DELTA
    const auto derivative = expr.derivative_custom(value, estimated_delta + i);

    // CALCULATE PRECISION FOR COMPARISON
    const auto precision = std::abs(real_derivative - derivative);

    // IF DELTA YIELDS MORE PRECISE RESULTS THAN PREVIOUS, CACHE DELTA
    if (precision < min_precision)
    {
        min_precision = precision;
        bruteforced_delta = estimated_delta + i;
    }
}

// OUTPUT BRUTEFORCE RESULTS
std::printf(" [Real]      %.12lf\n", real_derivative);
std::printf(" [Brute]     %.12lf\n", expr.derivative_custom(value, bruteforced_delta));
std::printf(" [Precision] %.14lf\n", min_precision);
std::printf(" [Delta]     %.14lf\n", bruteforced_delta);

return bruteforced_delta;
```

Figur (7)

Delta-værdien starter som udgangspunkt ved $\sqrt{E}$, og tilnærmer os indenfor en grænse af $-0{,}000001$ og $0{,}000001$ med intervallet E. Hver iteration udregner så hældningen og sammenligner mængden af korrekte betydende cifre med det givne facit, udregnet ved hjælp af CAS-værktøjet WolframAlpha. (wolframalpha.com). Når alle iterationer er færdiggjort, har man udregnet den optimale delta-værdi, der giver det mest præcise resultat, uden at blive afrundet, for vores givne udtryk.

I ovenstående tilfælde, udregnes vores deltaværdi til $-0{,}00000007270030$, og med denne værdi kan jeg udføre Newton-Raphson analyser med hele 15 betydende cifre.

Køretid

Køretid af applikationer defineres som den tid det tager for processoren at udføre et givent stykke arbejde. At måle køretid har sine komplikationer, da et moderne computersystem ikke altid kører et stykke kode monokront eller sekventielt. Et andet problem der tit opstår er implementeringen af uret der bruges til at tage tid på, som nødvendigvis heller ikke er monoton, altså en given tick har den samme arbitrære mængde af tid. På Windows, som softwareprogrammet køres på, er præcisionen af systemets ur ikke defineret, og det frarådes derfor at bruge relaterede systemfunktioner til at måle tiden hvis præcision er vigtig. C++ standarden har defineret en klasse ved navn *steady_clock* (Eel Is, 2019), der er specifikt opbygget med formål at måle tidsintervaller i softwareprogrammer. Den er således monoton, så to sekventielle forespørgsler om tid ikke resulterer i et negativt tidsinterval, hvilket gør den egnet til at vurdere køretid af algoritmer. På moderne computersystemer, så som Windows, vil processorens arbejde blive delt op af det man kalder en *thread-scheduler* (John Kennedy, 2018). Denne komponent vil fordele arbejdet virtuelt over et givent system, da moderne processorer ikke har hundredvis af tråde, som det ville kræve at køre et fuldstændigt operativsystem. Denne abstraktionslag kan skabe problemer, da man ikke kan kontrollere at sin algoritme bliver kørt fra start til slut uden at blive forsinket af *thread-scheduleren*. Heldigvis er det muligt på Windows at ønske, at sin algoritme bliver kørt fra start til slut uden ophørelse af eksekvering, ved at sætte sin tråds prioritet til *time-critical*. Dette gøres ved at lave et systemkald til funktionen *SetThreadPriority*. Dog skal det bemærkes at det blot er et ønske, og man ikke kan regulere om sit program reelt kørte sekventielt, så akkurat måling af køretider kræver en statistisk analyse af et større dataset end bare en enkelt prøve.

Hukommelsesforbrug

Hukommelse bruges til at behandle kortvarigt data, der ikke skal bevares efter computeren er blevet slukket. Dette gør det meget egnet til at opbevare forskelligt information ved kørsel af matematiske algoritmer og operationer, da hukommelse er eksponentielt hurtigere end harddiske. Jeg vil i analysen af algoritmerne tale om to forskellige former for hukommelsesforbrug: formelt og reelt forbrug. Formelt hukommelsesforbrug er det forbrug der ’på papir’ kan observeres som brugt i kildekoden. Det vil sige, at hvis jeg har et program der bruger to forskellige variabler, til to forskellige ting, kan man antage at programmet har et hukommelsesforbrug på summen af de to variablers størrelse. Man vil dog hurtigt indse at hukommelsesforbrug på moderne computersystemer er yderst optimeret, og lokationer i hukommelsen bliver genbrugt til en så grad, at det reelle forbrug kan være langt mere end tidligere anslået. I nogle tilfælde kan pendulet også svinge i den anden retning: programmer kan bruge mere hukommelse end anslået. Interne implementeringer i programmeringssprog kan resultere i et højere, reelt hukommelsesforbrug end den formelle modpart. Moderne software har adskillige forbrugsnoder i hukommelsen, der er fuldstændig implicitte for udvikleren af programmet. Dette skjulte forbrug indebærer for eksempel *return-adresser* (Doeppner, 2018), der efterlades når assemblyinstruktionen *call* (Doeppner, 2018) bruges til at lave et funktionskald, så funktionen kan returnere flowet uden at kende lokationen, ved brug af instruktionen *ret* (Doeppner, 2018).

Tidskompleksitet

Tidskompleksitet (Kurt Nørmark, 1994) er begrebet for den komplekse tid det tager for en given algoritme at køre. Algoritmer på computersystemer kan køre i forskellig tid alt efter størrelsen af input, og derfor bruges O-notation til at formidle den generelle hastighed af en algoritme ud fra inputdataen. Det kan være svært at anslå tidskompleksiteten for en given algoritme, da mange funktioner og metoder slet ikke tager den samme hastighed med samme størrelse input. I situationer med førnævnte problem, bruges begreberne: best-case-, worst-case- og gennemsnitstidskompleksitet. Disse begreber er henholdsvis det hurtigst mulige udkom, det langsomst mulige udkom og det gennemsnitlige udkom. Den gennemsnitlige tidskompleksitet er sjældent brugt, og jeg har derfor valgt kun at forholde mig til best-case og worst-case i min analyse. Tidskompleksitet er som sagt portrætteret ved hjælp af O-notation, der er en funktion af størrelsen af input n. Et eksempel ville være en gennemgang af en liste: $O(n)$. Da tiden det tager at kigge listen i gennem lineært stiger med længden af listen, er tidskompleksiteten lineær. Hvis man nu skulle gå i gennem hvert element i listen x gange, hvor x er antallet af elementer, ville tidskompleksiteten være: $O(n^2)$, da gennemgangen tager eksponentielt længere tid ved tilføjelse af nye elementer til listen. I situationer hvor algoritmens køretid er uafhængig af input, angives tidskompleksiteten som: $O(1)$, altså konstant. I tilfælde af at algoritmen aldrig slutter, angives tidskompleksiteten som: $O(\infty)$.

Antal operationer

Antallet af operationer en given computeralgoritme bruger, afhænger af definitionen af operationer. Operationer er et vagt begreb der kan fortolkes som blandt andet: iterationer, instruktioner eller matematiske operationer. Dette skulle have været klart defineret, eftersom det var et krav i projektoplægget. Antallet af operationer i den senere analyse er defineret som det antal af maskininstruktioner der bliver kørt per iteration af en given algoritme og antallet af iterationer.

Optimeringer

Iterationer

Programmets første iteration blev udarbejdet uden nogle former for optimeringer fra kompileringsværktøjets side af. Jeg skriver helt automatisk velfungerende kode, og første iteration var derfor allerede semantisk perfekt, så de eneste optimeringer jeg kunne foretage mig var at udnytte kompileringsværktøjets indstillinger til at presse alt juice ud af programmet. For at opnå den maksimale hastighed, har jeg ændret disse indstillinger:

- Optimization: Maximum Optimization (Favor Speed) (*/O2*)
- Inline Function Expansion: Any Suitable (*/Ob2*)
- Enable Intrinsic Functions: Yes (*/Oi*)
- Favor Size or Speed: Favor Fast Code (*/Ot*)
- Omit Frame Pointers: Yes (*/Oy*)
- Whole Program Optimization: Yes (*/GL*)
- Enable Enhanced Instruction Set: Advanced Vector Extensions (*/arch:AVX*)
- Floating Point Model: Fast (*/fp:fast*)

Vurdering af algoritme

Jeg vil i følgende afsnit sammenligne de to algoritmer så videnskabeligt som muligt. Jeg har taget fokus på effektivitet, køretid, hukommelsesforbrug. Jeg har i programmet taget højde for

Sammenligning af effektivitet

Effektiviteten af de to algoritmer er observeret som svingende, alt efter hvilken test-case der er tale om. Iteration 1 og 2 ses på venstre side af tabellen, og de henholdsvise køretider er fremvist i nano-sekunder.

x^2	Bisektion	Newton-Raphson
Iteration 1	2046 ns	1699362 ns
Iteration 2	1462 ns	63123 ns

$x^3 - x - 2$	Bisektion	Newton-Raphson
Iteration 1	17534 ns	3169 ns
Iteration 2	584 ns	0 ns

$\sqrt[3]{x}$	Bisektion	Newton-Raphson
Iteration 1	2396641 ns	N/A
Iteration 2	19872 ns	N/A

Som observeret, kan Newton-Raphson ikke udføres på den tredje test case, da en uendelig hældning optræder i udtrykkets kurve. Bemærk at hastigheden for Newton-Raphson i test-case to, iteration to ikke er en fejl, resultatet kan forklares ved, at algoritmens køretid var under de 200 nanosekunder, som er det minimale tidsinterval urkomponenten, brugt til at måle tidsintervallerne, kan opfange. Vi kan altså udlede en delkonklusion, der læner sig op ad, at algoritmerne hver i sær er effektive under forskellige forhold. Der er derfor ikke én algoritme der er bedre end den anden.

Der var heller ikke forskel i de forskellige facitter, så vi kan derfor også konkludere, at algoritmerne begge kan bruges til at udføre numeriske operationer uden at præcision er et problem.

Tidskompleksitet

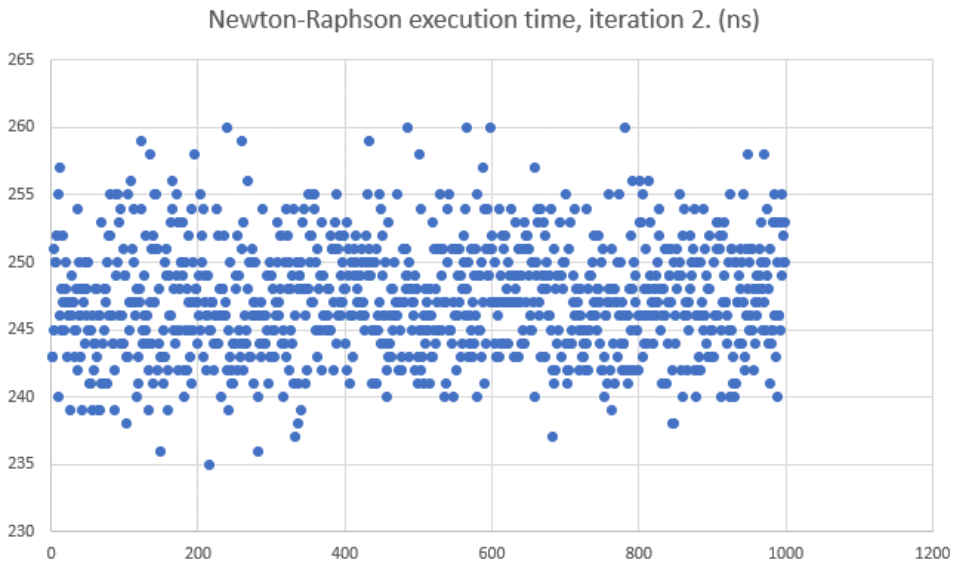
Tidskompleksiteten for algoritmerne afhænger af input, hvis input anses som udtrykket der analyseres, men kan generaliseres således:

	Bisektion	Newton-Raphson
Worst-case	$O(\infty)$	$O(\infty)$
Best-case	$O(1)$	$O(1)$

Bisektion kan i tilfælde ikke finde nulpunktet, som i udtrykket x^2, hvor at der ikke er forskellige fortegn på begge sider af roden. I sådan et tilfælde, er bisektionsalgoritmen enten $O(1)$ eller $O(\infty)$, alt efter om startintervallet er perfekt balanceret, således at bisektionsmetoden finder nulpunktet i første forsøg.

Newton-Raphson vil i tilfælde af et lineært udtryk være $O(1)$, da algoritmen arbejder ud fra startværdiens hældning og henholdsvise skæring med x-aksen. I tilfælde af udtryk med uendelig hældning vil Newton-Raphson ikke kunne udregne et punkt, og dette resulterer i en tidskompleksitet der, alt efter implementeringen, er enten $O(\infty)$ eller $O(1)$.

Hvis input anses som at være startværdien eller -intervallet, vil tidskompleksiteterne ikke ændre sig. Dette kan påvises ved at opsætte en algoritme der iterativt forværrer gættet og udregner køretiden. I tilfælde af test-case to, har jeg således udregnet den gennemsnitlige køretid, fra i alt 10000 prøver, og kan konkludere at der ikke er en statistisk ændring af køretiden. Dette kan derfor bruges som grundlag for min tidligere påstand: startværdi påvirker ikke tidskompleksitet. Herunder ses en visualisering af min videnskabelige undersøgelses resultater, med en graf af køretid i nanosekunder og startværdi hen ad x-aksen.



Figur (8)

Hukommelsesforbrug

Jeg har valgt at fokusere på det formelle hukommelsesforbrug for de to algoritmers implementeringer. Det totale hukommelsesforbrug for funktionerne kan ses i nedenstående tabel:

Newton-Raphson	Bisektion
32 bytes	50 bytes

De individuelle variabler der opbygger den samlede mængde af brugt hukommelse er for bisektion:

- Interval, 16 bytes
- Konstant ”1”, 8 bytes
- Mid_point 8 bytes
- Mid 8 bytes
- Mid_absolute 8 bytes
- Signbit begin 1 byte
- Signbit mid 1 byte

Og for Newton-Raphson:

- Fn(value) 8 bytes
- Derivative 8 bytes
- Value 8 bytes
- New_value 8 bytes

Dette vil sige at Newton-Raphson er mere effektiv når det kommer til pladsforbrug i hukommelsen, hvor der altså bliver sparet hele 12 bytes, altså en kæmpe forskel.

Det aktuelle hukommelsesforbrug har jeg ikke udregnet, dog kan jeg konkludere at begge algoritmer er korrekt implementeret som rekursive funktioner, således ingen instruktionspointere bliver gemt på *stacken*, hvilket sparrer en del hukommelse. Dette er gjort ved at funktionerne er afsluttet med et funktionskald til sig selv, og dette bliver ændret til en *jmp* instruktion takket være de førnævnte optimeringsindstillinger.

Antal operationer

Jeg har valgt at tage udgangspunkt i iteration 2, når det kommer til optællingen af instruktioner og iterationer.

Antallet af iterationer er talt ved at øge en talværdi ved hver eksekvering af de givne funktioner, og så nedskrive resultatet når analysen er færdig. Her ses en tabel med antallet af iterationer de forskellige algoritmer brugte i de forskellige cases:

Case	Antal af iterationer for bisektion	Antal af iterationer for Newton-Raphson
x^2	2	8567
$x^3 - x - 2$	51	7
$\sqrt[3]{x}$	1	N/A

Antallet af iterationer for Newton-Raphson er ikke angivet i tilfælde af test-case tre, da algoritmen ikke understøtter udtrykket. Ud fra ovenstående data jeg kan jeg således udlede, at Newton-Raphson i visse tilfælde fejler meget stort, hvor bisektion klarer sig godt. I test-case to klarer Newton-Raphson sig fantastisk, og er langt hurtigere end bisektion. I tredje test-case udregner bisektion nulpunktet i første huk, på grund af, at startintervallet er lige fordelt, og midtpunktet er derfor præcist ved nulpunktet.

For at udregne instruktioner per iteration, optælles hver instruktion i det kompilerede program. Dette er muligt, fordi at funktionerne er rekursive, og en hel iteration resulterer så derfor i at hele funktionen bliver kørt.

Bisektion	Newton-Raphson
Cirka 48 instruktioner	10 instruktioner

Imponerende nok, har kompileringsværktøjet formodet at koge HELE implementeringen af Newton-Raphson ned til 10 instruktioner. Instruktionerne består altså af udelukkende af multiplikation, subtraktion, addition og division, og en enkelt sammenligningsinstruktion for at tjekke resultatet:

```
loc_140011B0:
vmulsd  xmm5, xmm1, xmm1
vmulsd  xmm5, xmm5, xmm1
vsubsd  xmm6, xmm9, xmm1
vaddsd  xmm6, xmm6, xmm5
vaddsd  xmm7, xmm1, xmm0
vmulsd  xmm3, xmm7, xmm7
vmulsd  xmm3, xmm3, xmm7
vsubsd  xmm5, xmm2, xmm5
vaddsd  xmm3, xmm5, xmm3
vmulsd  xmm5, xmm6, xmm0
vdivsd  xmm3, xmm5, xmm3
vsubsd  xmm1, xmm1, xmm3
vandpd  xmm3, xmm3, xmm8
vucomisd xmm3, xmm4
jnb     short loc_140011B0
```

Figur (9)

Konklusion

Jeg kan konkludere, at det er muligt at bruge numeriske algoritmer til at udregne præcise nulpunkter på lukkede computersystemer. Jeg har påvist, at algoritmernes effektivitet afhænger drastisk af udtrykkets kurve, og ikke startværdierne. Jeg har blandt andet også observeret et kompileringsværktøjs muligheder for at optimere en algoritme på en computer, og redegjort for hvordan jeg har optimeret programmet for at opnå det bedst mulige resultat. For at optimere programmet, ændrede jeg altså C++ kompileringsindstillinger med fokus på hastighed, og gav grønt lys for at kompileringsværktøjet brugte et specielt instruktionssæt der er hurtigere end normale instruktionssæt.

De to algoritmer har samme O-notation i worst-case og best-case, henholdsvis $O(\infty)$ og $O(1)$. De endelige resultater for hastigheden af algoritmerne er:

x^2	Bisektion	Newton-Raphson
Iteration 1	2046 ns	1699362 ns
Iteration 2	1462 ns	63123 ns

$x^3 - x - 2$	Bisektion	Newton-Raphson
Iteration 1	17534 ns	3169 ns
Iteration 2	584 ns	0 ns

$\sqrt[3]{x}$	Bisektion	Newton-Raphson
Iteration 1	2396641 ns	N/A
Iteration 2	19872 ns	N/A

Jeg kan hermed konkludere at de to algoritmers effektivitet afhænger meget af det analyserede udtryk, og man kan derfor ikke lave en absolut konklusion om at den ene er bedre end den anden.

Litteraturliste

Referencemateriale

- Doeppner (2018). *x64 Cheat Sheet*. Besøgt 12/04/2019 på https://cs.brown.edu/courses/cs033/docs/guides/x64_cheatsheet.pdf
- Eel Is (12. april, 2019). *Working Draft, Standard for Programming Language C++*. Besøgt 12/04/2019 på: http://eel.is/c++draft/
- IEEE (12. oktober, 1985). *IEEE Standard for Binary Floating-Point Arithmetic.* Besøgt 12/04/2019 på: https://ieeexplore.ieee.org/document/30711
- John Kennedy (31. maj, 2018). *Scheduling - Windows applications.* Besøgt 12/04/2019 på: https://docs.microsoft.com/en-us/windows/desktop/procthread/scheduling
- Kurt Nørmark (1994). *Algoritmeanalyse.* Besøgt 12/04/2019 på: http://people.cs.aau.dk/~normark/ps1-94-notes/pdf/analyse.pdf
- Martin Zachariasen. *Grundlæggende køretidsanalyse af algoritmer.* Besøgt 12/04/2019 på: https://docplayer.dk/34968160-Grundlaeggende-koeretidsanalyse-af-algoritmer.html
- Microsoft (12. maj, 2018). *SetThreadPriority function*. Besøgt 12/04/2019 på https://docs.microsoft.com/en-us/windows/desktop/api/processthreadsapi/nf-processthreadsapi-setthreadpriority
- University of Illinois (20. marts, 2019). *The LLVM Compiler Infrastructure*. Besøgt 12/04/2019 på: https://llvm.org/
- Weisstein, Eric W. *Bisection*. Besøgt 12/04/2019 på: http://mathworld.wolfram.com/Bisection.html
- Weisstein, Eric W. *Newton's Method*. Besøgt 12/04/2019 på: http://mathworld.wolfram.com/NewtonsMethod.html
- Weisstein, Eric W. *Numerical Differentiation.* Besøgt 12/04/2019 på: http://mathworld.wolfram.com/NumericalDifferentiation.html

Figurer

1. Carl Schou. bisektion1.ggb. screenshot
2. Carl Schou. bisektion2.ggb. screenshot
3. Carl Schou. bisektion3.ggb. screenshot
4. Carl Schou, newton_raphson.ggb, screenshot
5. Ralf Pfeifer (9. april, 2005). *Die Animation zeigt mehrere Iterationsschritte des Newton-Verfahrens.* Fra: https://de.wikipedia.org/wiki/Datei:NewtonIteration_Ani.gif
6. Carl Schou, expression.hpp, screenshot (bruges på forside)
7. Carl Schou, analysis.hpp, screenshot
8. Carl Schou, newton.xlsx, screenshot
9. Carl Schou, src.exe, screenshot

Filer

- bisektion1.ggb
- bisektion2.ggb
- bisektion3.ggb
- code_1.png
- front.png
- iteration_1.zip
- iteration_2.zip
- newton.xlsx
- newton_raphson.ggb
- src.exe