Aarhus Universitet

Institut for Fysik og Astronomi

Eksamensprojekt i
Eksperimentelfysik og Statisktisk Dataanalyse

Robuste Fittemetoder: Genetiske Algoritmer

Af:
Mads B. Carlsen
Emil Hansen
Nikolaj Klinkby

13. maj 2020

Resumé

To typer af genetiske algoritmer (GA'er) bliver præsenteret: binære og kontinuerte GA'er, og den teoretiske baggrund og implementeringen af disse bliver gennemgået. Dernæst ses på benchmarking af algoritmerne ift. minimering af Rastrigin-funktionen, og der ses på hvordan algoritmen kan bruges til at fitte til eksperimentel data via minimering af χ^2, på både et simpelt og avanceret problem. Til avanceret fitting foreslåes at den bedste løsning er en hybridløsning: at bruge GA'en til at finde et godt startgæt og derefter bruge SciPy's `curve_fit` til at finde nøjagtigt minimum samt kovariansmatricen.

1 Indledning

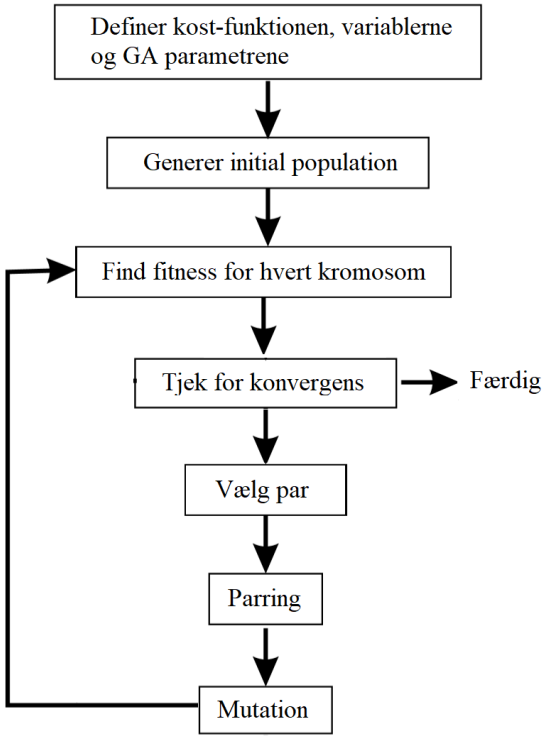
I alle problemstillinger, vi har mødt gennem kurset, har det været tilstrækkeligt at bruge forskellige udgaver af en lokal minimeringsalgoritme (som f.eks. gradient descent eller variationer af Newton's method)[5;4] til at minimere funktioner og fitte til data. En enkelt gang eller to har det dog krævet lidt ekstra tid, fordi metoderne ofte behøver et godt start gæt for at finde et tilfredsstillende minimum. Dette problem kan overkommes ved at anvende en global optimerings algoritme, der i stedet for at skulle bruge et godt gæt (og derfra finde det nærmeste minimum) skal kende det område, den skal søge i og finder det globale minimum i området. Umiddelbart virker det mærkeligt, hvorfor man så ikke altid bruger disse algoritmer, men oftest kræver de meget længere tid om at finde et minimum og skal i mange tilfælde også skrives specifikt til det problem, som de skal behandle.[3]

En type af globale optimerings algoritmer er den såkaldte genetiske algoritme. Idéen bag genetiske algoritmer - og evolutionære algoritmer generelt - kommer fra Charles Darwins princip om "*survival of the fittest*". Grundlæggende går en genetisk algortime ud på, at først danne en tilfældig samling af potentielle løsninger til et problem. De bedste af disse (typisk dårlige) potentielle løsninger 'avles', for at danne en endnu bedre. Dette gentages i flere generationer, til en optimal løsning er nået.[4]

Lidt mere specifikt indeholder en GA en *befolkning* af N_{pop} mulige løsninger til et problem, også kaldet *kromosomer*. Hver kromosom indeholder information af de N_{var} parametre for problemet, såkaldte *gener*. Hvordan selve generne er udtrykt varierer de forskellige typer GA imellem, som vil blive tydeligt i de følgende afsnit. Typisk vil hele populationen dog repræsenteres vha. en matrix $\mathbf{P}$, hvor hver række er et kromosom. Hvor god en løsning de enkelte kromsomer er, evalueres gennem en såkaldt *kostfunktion*. Herved tildeles hvert kromosom en værdi for, hvor gode de er, kaldet *fitness-værdien*. Oftest ønskes at minimere en matematisk funktion $f : \mathcal{S} \to \mathbb{R}$, hvor $\mathcal{S} \subset \mathbb{R}^n$. Målet er således at finde et kromosom repræsenterende et $\mathbf{x}^* \in \mathcal{S}$, hvor

$$f(\mathbf{x}^*) \leq f(\mathbf{x})$$

for alle $\mathbf{x} \in \mathcal{S}$.[3] Dvs. de bedste løsninger i dette tilfælde er de løsninger med lavest fitness-værdi.[4]



Figur 1. *Generelt flowchart for genetiske algoritmer (inspireret af Practical Genetic Algorithms[4]).*

Herefter udvælges den X_{keep} bedste procentdel af den inertielle befolkning, baseret på deres fitness-værdi. De $N_{\text{parents}} = N_{\text{pop}} X_{\text{keep}}$ fitteste individer, rundet op til næste lige tal, får lov til at danne nye indvider, der erstatter de ikke-udvalgte (de dårligste løsninger). Inden denne *parring* skal løsninger sammensættes i par. Der findes forskellige metoder til at udvælge parrene på. Det kan både gøres hvor parrene vælges tilfældigt og uafhængig af fitness-værdien, men også hvor hvert individs fitness-værdi bliver vægtet, så de bedste løsninger har størst sandsynlighed for at føre deres gener videre. De udvalgte par skal nu danne $N_{\text{offspring}} = N_{\text{pop}} - N_{\text{parents}}$ afkom, så den nye population har samme antal som den oprindelige. Igen er der mange måder at gøre dette på, men den simpleste, som er direkte inspireret af genetikken, er at vælge ét eller flere *crossover*-punkter i kromosomerne, hvor

kromosomerne krydser over og bytter gener[4]. Denne metode bliver også illustreret under binære algoritmer.

Det sidste skridt er *mutation*. Her skal vælges endnu en vigtig faktor, *mutationsraten* R_{mut}, som også tager værdier mellem 0 og 1, men normalt ligger på en værdi på omkring 0.2.[3;4] Den mest simple tilgang til mutation er at tilfældigt udvælge indgange i $\mathbf{P}$, og så erstatte indgangen med en tilfældig ny passende værdi (den nye værdi skal stadig være en mulig værdi for parameteren). Antallet af mutationer der skal foretages beregnes ved $N_{\text{mutations}} = \lceil N_{\text{pop}} R_{\text{mut}} \rceil$. En meget vigtig pointe ved mutation er at sørge for ikke at mutere populationens bedste kromosom, for at forhindre at man smider den bedste løsning væk.[4]

Ovenstående serie af skridt gentages nu som illustreret på Fig. 1, indtil en passende løsning er opnået, eller et maximalt antal iterationer nåes.

2 Genetiske algoritmer i praksis

2.1 Binær genetisk algoritme

Den binære GA er den oprindelige udgave af de genetiske algoritmer. Inspireret af hvordan DNA oplagerer information gennem rækkefølgen af baser i DNA'et, lader man hver kromosom i algoritmen bestå af en bit-streng - dvs. en liste af 1'ere og 0'ere.[4] Hvis man fx havde en genetisk algoritme der arbejder med tre parametre, kunne et kromosom have formen:

$$[101\ 010\ 110]$$

Hvert gen i kromosomet består altså af N_{gen} bits. Generne skal omregnes fra binær til reelle tal, for at kunne evalueres i værdi-funktionen. Dette kan gøres vha. følgende ligninger:[4]

$$q = 2^{-(N_{gen}+1)} + \sum_{m=1}^{N_{gen}} \text{gen}[m] \cdot 2^{-m}$$
$$p = q(p_{hi} - p_{lo}) + p_{lo}$$

Hvor p er den reelle værdi af genet, gen$[m]$ er den m'te bit i genet og p_{hi} og p_{lo} er den højeste hhv. laveste værdi for det interval som de binære gener skal repræsentere. Grundet dets binære opbygning, kan hvert gen kun repræsentere $2^{N_{gen}}$ værdier på dette interval. Hvis fx $N_{gen} = 8$ giver dette kun en opløsning på 256 forskellige værdier. Den diskrete natur af den binære algoritme gør dog samtidig, at denne er ideel til at løse diskrete problemstillinger.[4] Bemærk nemlig, at man ikke er begrænset til at lade generne repræsentere reelle tal, hvis man ikke forsøger at optimere en matematisk funktion.

Som udgangspunkt følger den binære genetiske algortime de trin og metoder, der er beskrevet i forrige afsnit. Der findes mange metoder at parre sine individer, men den mest

simple er *single-point crossover*. Her vælges et bit i forældre-kromosomerne og to nye individer dannes ved at ombytte bit-strengene før og efter dette punkt de to forældre imellem. Dette forstås bedst ved et eksempel. Lad følgende være to forældre:

$$[010110101]$$
$$[\mathit{100111001}]$$

Det 4. bit vælges som overkrydsningspunktet og to nye individer dannes:

$$[\mathit{1001}10101]$$
$$[0101\mathit{11001}]$$

På denne måde får man dannet nye indvider til populationen, der er en blanding af genetisk materiale fra forældrene.

Mutation i den binære GA er en ret simpel procedure. Her udvælges blot et antal bits, givet ved $N_{mut} = \lceil N_{pop} X_{mut} N_{gen} \rceil$, i hele populationen, og disse bits inverteres. Hvis fx et udvalg bit har værdien 0 inverteres denne således til 1.[4]

2.2 Kontinuert genetisk algoritme

De kontinuere GA'er tager samme form som de binære GA'er, men kromosomerne er opbygget forskelligt. I det kontinuere tilfælde tildeler vi ikke hver variabel en bit streng, men beholder dem direkte, som en værdi i kromosomet. Dvs. at et en N_{var}-dimensional funktion med $p_1, p_2, ..., p_{N_{\text{var}}}$ variabler har et kromosom givet ved,[4]

$$\mathbf{x} = \begin{pmatrix} p_1 & p_2 & \cdots & p_{N_{\text{var}}} \end{pmatrix}.$$

Fordelen ved dette er at vi ikke længere skal tænke over mængden af bits pr. variable eller i det hele taget konvertere bits.
Herfra løber processen som før og der skal laves en initial population, hvilket gøres ved at lave en $N_{\text{pop}} \times N_{\text{var}}$-matrix $\mathbf{P}$ med tilfældige værdier mellem 0 og 1 i hver indgang, og er på formen

$$\mathbf{P} = \begin{pmatrix} — & \mathbf{x}_1 & — \\ — & \mathbf{x}_2 & — \\ & \vdots & \\ — & \mathbf{x}_{N_{\text{pop}}} & — \end{pmatrix}$$

Dette skaber N_{pop} kromosomer i populationen. Hver gang vi ønsker at finde værdien for hvert kromosom de-normere vi det, ved at tage de øvre og nedre grænser for variablerne, som vi

ønsker at undersøge (vi udvælger et område af funktionen at undersøge). For alle variablerne bruger vi da at[4]

$$p_{\text{denorm}} = (p_{\max} - p_{\min})p_{\text{norm}} + p_{\min}.$$

De de-normerede kromosomer kan vi indsætte direkte i kostfunktionen for at evaluere kromosomerne, men vi beholder de normede kromosomer, som er dem vi arbejder videre med. Hvis vi vælger at parre dem på samme måde, som den binære GA, med et crossover-punkt/punkter, så introducerer vi aldrig nogen ny information, men bytter bare rundt på værdierne for vores variabler. I stedet kan vi bruge en anden metode, nemlig at introducere en tilfældig afvigelse fra enten moder kromosomet p_m eller fader kromosomet p_f, afhængigt af både moderen og faderen. Dette opnås ved at udvælge en tilfældig, positiv, værdi β så,[4]

$$\begin{aligned} p_{\text{ny1}} &= p_{\text{m}} - \beta(p_{\text{m}} - p_{\text{f}}) \\ p_{\text{ny2}} &= p_{\text{f}} + \beta(p_{\text{f}} - p_{\text{m}}) \end{aligned}$$

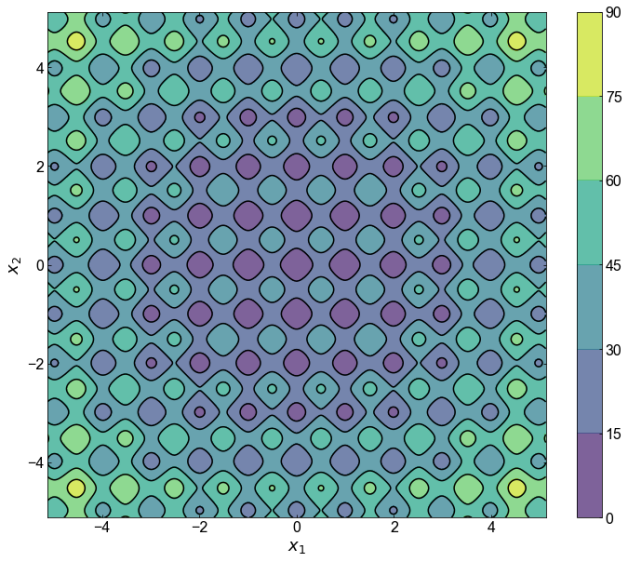
På den måde kan vi introducere variabler udenfor det område, hvor fadere og moderen ligger. Hvis vi vil udvide det meget, så kan vi lade $0 \leq \beta \leq a$, for en værdi $a > 1$. Oftest vil vi dog bare vælge $a = 1$, hvorved vi ender med at undersøge variabler ud fra moderen og faderen i intervaller $[p_m, p_f]$ og $[p_f, 2p_f - p_m]$. Hvis dette gøres for alle variabler og man lader hvert par lave to afkom, så kan afkommet skrives som[4]

$$\begin{aligned} \text{afkom}_1 &= \text{moder} - [\beta_1(p_{\text{m},1} - p_{\text{f},1}), \beta_2(p_{\text{m},2} - p_{\text{f},2}), \cdots, \beta_{N_{\text{var}}}(p_{\text{m},N_{\text{var}}} - p_{\text{f},N_{\text{var}}})] \\ \text{afkom}_2 &= \text{fader} + [\beta_1(p_{\text{f},1} - p_{\text{m},1}), \beta_2(p_{\text{f},2} - p_{\text{m},2}), \cdots, \beta_{N_{\text{var}}}(p_{\text{f},N_{\text{var}}} - p_{\text{m},N_{\text{var}}})] \end{aligned}$$

Den nye population af forældre og afkom, får udvalgt hvilke indgange der skal muteres og da det hele er normeret, så indsættes bare en ny værdi mellem 0 og 1 på den muterede plads. Herefter er det vigtigt at tjekke alle indgange stadig ligger indenfor 0 og 1. Hvis de ikke gør kan man enten sætte dem til den nærmeste værdi 0 eller 1, eller lade dem køre i ring i intervallet.
Herfra har man nu en ny population, der er klar til at blive de-normeret og evalueret.

3 Resultater

3.1 Testkørsler af genetiske algoritmer



Figur 2. *Contour plot af funktion (1) med* $n = 2$*. Farvesøjlen til højre giver sammenhæng mellem farve og funktionsværdier.*

Til at teste genetiske algoritmer findes flere standardfunktioner. Her vælges følgende funktion, kendt som Rastrigin funktionen, der også benyttes til at teste andre typer af minimeringsalgortimer:[6]

$$f(\mathbf{x}) = nA + \sum_{1}^{n} \left[x_i^2 - A\cos(2\pi x_i) \right] \qquad -5.12 \leq x_i \leq 5.12 \tag{1}$$

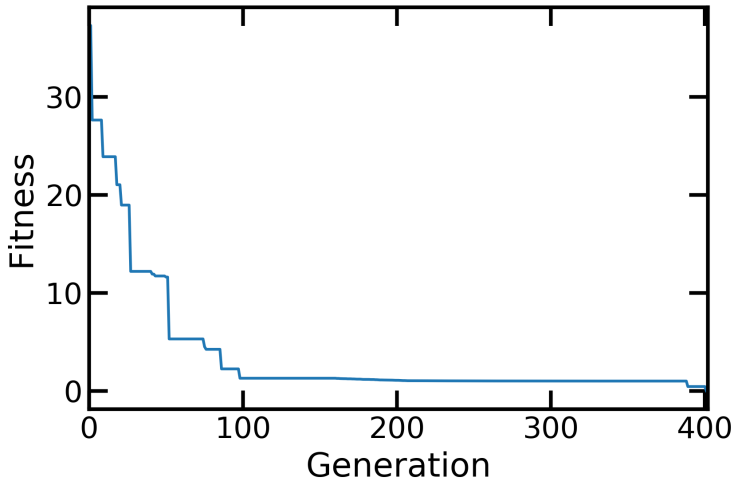
Her sættes $A = 10$ hvorved funktionen vil have globalt minimum i $\mathbf{x} = \mathbf{0}$ med funktionsværdien $f(\mathbf{0}) = 0$. Funktionen er valgt, da det er let at opskalere dimensionen, og dermed øge kompleksiteten af problemet, der skal løses. Derudover har funktionen mange lokale minima (se evt. Fig. 2), som de lokale minimeringsmetoder vil konvergere imod, hvis deres udgangspunkt er i nærheden af disse. Det vil derfor være utroligt svært at finde det globale minimum af (1) med metoder som f.eks. gradient descent, hvorimod det skulle kunne være muligt med en gentisk algoritme.

For at teste dette er skrevet 2 forskellige genetiske algoritmer, fundet i bilag A1 og A2. Algoritmen i A1 er kontinuert, mens den anden er binær. I det følgende vil algortimerne i bilag A1 og A2 testes og sammenlignes. For den binære algoritme benyttes en population

på $N_{pop} = 8$, en mutationsrate på $R_{mut} = 0.03$ og $N_{gen} = 12$ bits til at repræsentere hver parameter. Algortimen i A2 udvælger par til parring, ved at vægte de bedste løsninger højest, (såkladt rank-weighting[4]). Algoritmen søger indtil den bedste løsning har en fitness-værdi på under 0.01 ((1) antager kun denne værdi i brønden ved det globale minimum), da den binære GA ikke vil kunne ramme en funktionsværdi på præcis 0.0 grundet dens endelige opløsning.

I den kontinuerte GA fundet i A1 bruges at størrelsen af populationen er givet ved $N_{\text{pop}} = 8n$, hvor n er antallet af parametre. Dette gøres grundet en større population gør algortimen mere effektiv når der skal findes minimum i højere dimensioner.[4] Derudover er $X_{\text{keep}} = 0.5$ og $R_{\text{mut}} = 0.2$. At sætte mutationsraten eller X_{keep} meget højere eller lavere resulterer som oftest i markant længere beregningstid. Udvælgelsen er tilfældig blandt de N_{keep} bedste for at bevare nogle mindre gode løsninger, så algoritmen ikke bliver for elitær.

En tilfældig kørsel med GA'en fra bilag A1, på (1) er illustreret på Fig. 3. Her ses, at det lykkedes algoritmen at opnå en løsningen efter ca. 400 generationer. Det bemærkes dog også, at der til start sker et hurtigt fald, mens at udviklingen i den bedste løsning flader ud hen mod de høje generationer.

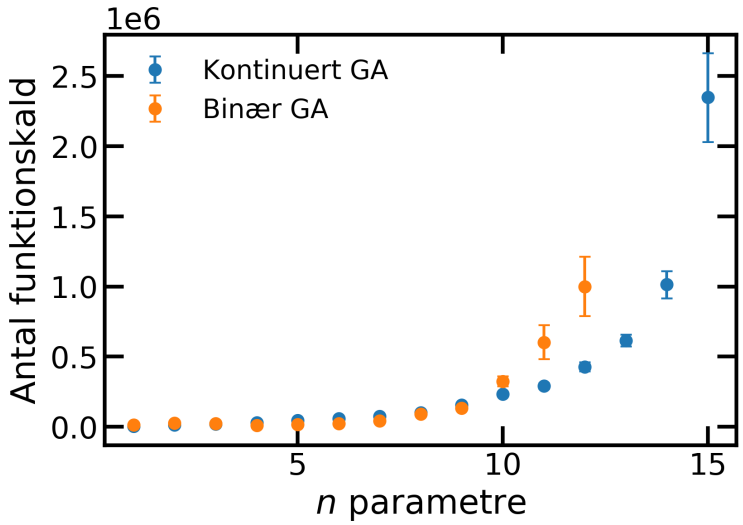


Figur 3. *Eksempel på en kørsel med den kontinuære GA i bilag A1 på (1) med 4 parametre. Fitness-værdien for den bedste løsning er plottet som funktion af antallet af generationer.*

Fordi GA'er er baseret på tilfældighed (både start population, avl og mutation), vil det også variere en del hvor mange generationer, det tager at opnå en løsning. For at undersøge kompleksiteten af beregningerne udføres derfor 15 kørsler for hver dimension. Dette lave antal kørsler skyldes den store mængde tid involveret i beregningerne. Istedet for at måle antallet af generationer, måles istedet hvor mange gange kostfunktionen kaldes (dette gøres ofte,[6] da dette mål er uafhængig af algoritmens opbygning). Fordelingen af antal kald til

kostfunktionen for hver dimension viser sig ikke at være Gauss-fordelt, heller ikke ved højere antal kørsler. Grundet kombinationen af få datapunkter og ikke Gauss-fordelt data vælges usikkerheden på dataet at findes vha. bootstrap metoden.[5] Dvs. for hvert samling af 15 målinger, dannes et stort antal samples, bestående af tilfældige udtrukne værdier fra den oprindelige samling. Hver sample indeholder samme antal elementer som den oprindelige, dvs. 15. For hver sample beregnes nu en middelværdi, og denne samling af middelværdier fra de forskellige sample fordelinger afspejler nu fordelingen af den oprindelige samling af elementer. Ved at sortere middelværdierne kan 2/3 konfidensintervallet herefter findes for fordelingen, og dermed for dataet fundet vha. GA'en.

Ovenstående udføres for både den binære GA fra bilag A2 og den kontinuerte GA fra bilag A1. Resultatet er plottet i Fig. 4.



Figur 4. *Antallet af funktionskald for den kontinuerte og binære GA fra bilag A1 til minimering af (1). Algoritmen stopper og returnerer resultatet, hvis* $f(\mathbf{x}) \leq 0.01$.

Det er altså lykkedes de fremstillede algoritmer, at løse (1) med op til hhv. 12 og 15 parametre. Det ses umiddelbart at den kontinuere algoritme klarer sig bedre end den binære algoritme ved de højere dimensioner, mens de følges ad op til omkring 9 dimensioner. Der er kun 12 datapunkter for den binære GA, da beregningstiden for højere dimensioner var meget stor. I dataet ses desuden en tendens til at antallet af funktionskald stiger eksponentielt. Læg desuden mærke til, at skalaen på aksen er 10^6. Dvs. at når der skal optimeres et stort antal parametre på én gang, benytter GA'erne op mod flere millioner funktionskald.

3.2 Fitting med genetiske algoritmer

Her vil det ses på, hvordan det er muligt at bruge en kontinuert genetisk algoritme til at udføre et fit af en funktion $y(x, \mathbf{a})$, hvor $\mathbf{a}$ er en vektor med funktionens uafhængige parametre, til et datasæt bestående af n datapunkter (x_i, y_i) for $i = 1, 2, \ldots, n$. Der sammenlignes i afsnittet med fitte-rutinen `curve_fit` fra Python's Scipy. Denne benytter som standard Marquardt-Levenberg metoden til minimering, som er en blanding af gradient descent og Gauss-Newton methoden (en videreudvikling af Newton metoden).[5;1] For at udføre et fit med en genetisk algoritme minimeres χ^2-funktionen givet ved,[5]

$$\chi^2(\mathbf{a}) = \sum_i^n \left(\frac{y_i - y(x_i, \mathbf{a})}{\alpha_i} \right)^2, \tag{2}$$

hvor α_i er usikkerheden på det i'te datapunkt. Målet er således at finde den optimale sammensætning af $\mathbf{a}$, nemlig $\mathbf{a}^*$ hvorom det gælder at $\chi^2_{\text{min}} = \chi^2(\mathbf{a}^*)$.

Først ses på et simpelt eksempel; at fitte til en trigonometrisk funktion, i dette tilfælde en sinusfunktion på formen

$$y(x) = A \sin(\omega x + \phi). \tag{3}$$

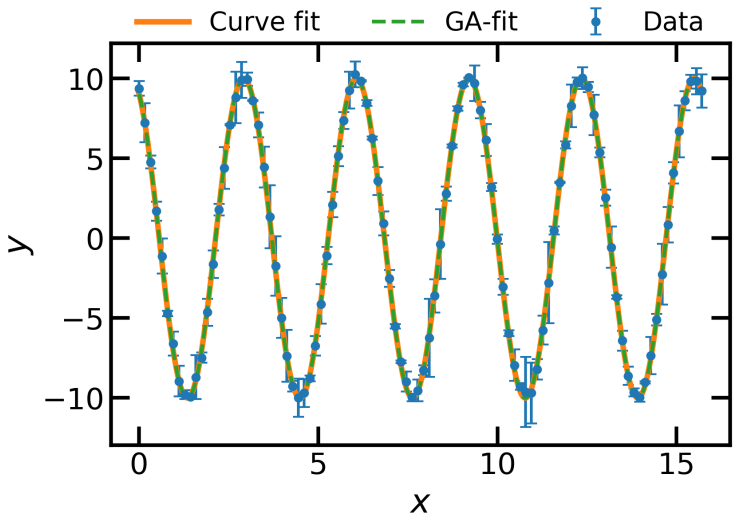
Det, der oftest volder problemer i denne forbindelse at få et ordentligt gæt på vinkelfrekvensen ω, da fitte-rutinen oftest er meget sensitiv overfor denne.

På Fig. 5 ses resultatet af et fit med både `curve_fit` og en kontinuær GA (bilag A1) på noget tilfældigt genereret sinusdata med støj. Visuelt er det tydeligt, at de to metoder finder samme resultat. På Tabel 1 finder vi de returnede parametre fra hhv. `curve_fit` og GA'en, hvor vi ser afvigelsen mellem de to er ganske lille, men grundet i at GA'en ikke nødvendigvis finder minimum lige så nøjagtigt som `curve_fit`. Men mens `curve_fit` skulle have et relativt nøjagtigt gæt for ω (ca. intervallet fra 1.8-2.3) for at konvergere til ovenstående løsning, var GA'en indstillet til blot at undersøge ω i intervallet 0-50. Det lykkedes altså GA'en at søge over et større område, og frasortere de lokale minima, som `curve_fit` ellers godt kan falde i. Til gengæld benyttede GA'en 11.200 funktionskald mens `curve_fit` benyttede omkring 30 (alt efter hvor nøjagtig startgættet var).

Metode	A	ω	ϕ	χ^2	$P(\chi^2; \nu)$
`curve_fit`	10.003	1.998	2.018	87.751	0.738
GA-fit	10.002	1.999	2.017	87.803	0.737

Tabel 1. *Funde parametre til fittene illustreret på Fig. 5.*

Eftersom det lykkedes at udføre et simpelt fit, forsøges i det følgende med et lidt mere ambitiøst. Der forsøges at fitte til data fra Fizeau's eksperiment, hvor der ses på intensiteten



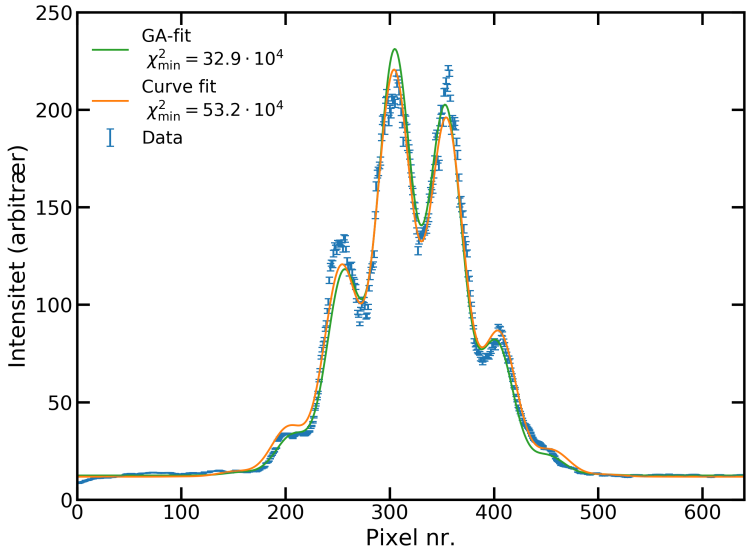
Figur 5. *Generet sinusdata med normaltfordelt støj (blå) hvortil der er lavet to fits: Fit med den genetiske algoritme (grøn) og med SciPy's* `curve_fit` *(orange). Dataet er generetret ud fra (3) med A=10,* $\omega = 2$ *og* $\phi = 2$*. Bemærk desuden, at der ikke regnes med usikkerheder på datapunkterne, da interesseren ikke er hvor godt selve fittet er.*

på tværs af et interferensmønster. Der skal gøres opmærksom på, at dataet ikke er optimalt, og at fittet derfor ikke vil blive det uanset. Interessen ligger i, om GA'en kan finde et evt. bedre bud på en løsning end `curve_fit`, og uden at bruge lang tid på at finde et start-gæt. Funktionen, der skal fittes til dataet er givet ved ligningen[2]

$$f(x) = \sin^2(kx + \delta)\left[b \cdot \mathrm{e}^{-\frac{(x-\mu)^2}{2\sigma^2}} - c \cdot \mathrm{e}^{-\frac{(x-\mu)^2}{2\sigma^2}}\right] + c \cdot \mathrm{e}^{-\frac{(x-\mu)^2}{2\sigma^2}} + a, \tag{4}$$

med parametrene $[a, b, c, k, \delta, \mu, \sigma]$. Her skal (2) altså minimeres for 7 variable på samme tid. At have så mange variable og en så kompliceret funktion gør, at man skal være ret præcis med sit startgæt, hvis man skal opnå et brugbart fit med `curve_fit`.

Hvis vi betragter Fig. 6 kan vi se, at der er blevet fundet en løsning vha. den genetiske algoritme selvom søgerummet er stort. Som forventet har begge fits har en høj χ^2-værdi og er ikke gode fits (faktisk har begge fits en $P(\chi^2_{\text{min}}; \nu)$-værdi på 0), men vi ser dog at χ^2-værdien er lavest for fittet fra den genetske algoritme. Pointen er, at der med GA'en er fundet et sæt af parametre (omend ikke godt) uden at skulle bruge lang tid på at gætte på startværdier til `curve_fit`.



Figur 6. *Dataene fra forsøget (blå) samt fits lavet med hhv. SciPy's* `curve_fit` *(orange) og vores genetiske algoritme (grøn). Vi ser her, at* χ^2_{min}*-værdien er lavest for fittet genereret med den genetiske algoritme. Se koden i bilag B1.*

4 Diskussion

Dataene fra Fig. 4 ligger op til en diskussion om hvilken slags GA er bedst. Med den begrænsede data vi har produceret ligner det at den kontinuere GA klarer sig bedre, men det kan være svært at sammenligne dem, da de bygger på to forskellige måder at lave en GA på. Som det også bliver nævnt i [Haupt & Haupt], så kræver en problemstilling ofte en skredersyet GA, for optimal løsning. Vores data fra Fig. 4 peger på at den kontinuere algoritme i bilag A1 er bedre til at løse en Rastrigin funktion end den binære algoritme i bilag A2, men det betyder ikke at den binære algoritme ikke ville være bedre til at løse et andet problem. En kontinuert GA er, i tilfælde med kontinuere funktioner, bare mest praktisk. Som nævnt i afsnit 3.1, så arbejder den binære algoritme fx meget langsommere end den kontinuerte. En undersøgelse af koden viste, at størstedelen af tiden (og beregningerne) gik til konvertering fra bit til reelle tal. De kontinuerte algoritmer har den klare fordel at de slipper for disse konverteringer.

Parametrene benyttet for GA'erne i afsnit 3.1, var fundet som de umiddelbart bedste. En pointe er, at de overhovedet ikke er ens for de to algoritmer. Generelt kan man ikke fastsætte nogle parametre, der fungerer bedst, det viser sig i stor grad at afhænge af hvordan ens GA er opbygget. Vælges fx en alternativ måde at parre sine kromosomer, vil dette have stor indflydelse på effektivteten af algoritmen og hvilke parametre, der er mest optimale. Da

en GA ikke er en matematisk formuleret algoritme, er det desuden meget svært at forudsige hvilke parametre og metoder, der vil være gode. En advarsel er således, at man hurtigt vil kunne bruge lige så lang tid på at finde de optimale parametre og opsætning for ens GA, som man kan på at finde et godt start gæt for en lokal minimeringsrutine. Endvidere vil nogle problemstillinger kun kunne løses med den helt rette sammensætning af metoder i ens GA.

Hvorvidt det er relevant at bruge en GA til en problemstilling ser ud til at afhænge stærkt af problemet. I tilfældet med den trigonometriske funktion ville man hurtigt kunne komme med et passende gæt til `curve_fit` og fordelen ved den genetiske algoritme kommer nok først i det, man ikke længere ønsker at være begrænset af at skulle gætte. For eksempel i et tilfælde hvor man skal fitte mange funktioner til mange datasæt, hvor alle datasættene ikke kan bruge samme gæt for `curve_fit`. Til gengæld skal man være opmærksom på, at en GA kræver væsentlig mere beregningskraft, som set i afsnit 3.2.

En stor fordel ved `curve_fit` er også, at den giver én covarians-matricen. Man kunne bruge `curve_fit` på sin GA's bedste løsning for at få dette. En kombination af en GA og en lokal minimeringsalgoritme kaldes også for en hybridalgoritme. Ideen er, at man med den genetiske algoritme kan finde den rigtige brønd, hvori det globale minimum ligger, hvorefter en lokal minimeringsalgoritme kan tage over og hurtigt konvergere til en præcis løsning. Man kan også aktivere den lokale minimeringsalgoritme, hvis forbedringen i værdien for kostfunktionen for den GA'en begynder at aftage med en hvis værdi.[4] Et godt eksempel er Fig. 3, hvor det ville have være mere optimalt at bruge en hybrid algoritme, der blev aktiveret efter 100 generationer.

Det skal nævnes, at GA'erne skrevet her, ikke er optimale eller specielt hurtige (jævnfør Fig. 4). Skriver man dem rigtigt (og tilpas avanceret), kan de dog være meget effektive og løse (1) med 400 parametre på ganske kort tid.[6] Det skal også nævnes, at der findes andre globale minimerinsalgortimer, der i mange tilfælde arbejder hurtigere og mere effektivt, fx differentiel-evolution algoritmer.[3] Hvis man programmerer i Python, så har biblioteket `scipy.optimize` allerede en differentiel evolution algoritme til rådighed.[1]

5 Konklusion

Vi har i dette projekt betragtet og udviklet to forskellige genetiske algoritmer. Først en binær genetisk algoritme, som egner sig bedst til diskrete problemer (men som kan bruges til kontinuerte problemer) og dernæst en kontinuert genetisk algoritme, som modsat egner sig bedst til kontinuerte problemer. Algoritmerne fandt succesfuldt det globale minimum på rastrigin-funktionen, op til 15 dimensioner. Illusteret er også hvordan man kan bruge den kontinuerte GA til at minimere χ^2 på både et simpelt og et mere avanceret problem. Konklusionen er, at den bedste måde at udnytte GA'en på, er til at finde et godt startgæt til `curve_fit`, og dermed bruge kombinationen af de to som en hybridalgoritme til fitting.

Litteratur

[1] Optimization and root finding (scipy.optimize), Dec 19, 2019. URL `https://docs.scipy.org/doc/scipy/reference/optimize.html`.

[2] Mads Slot Bertelsen. A reenactment of the fizeau experiment. Master's thesis, Aarhus University, 2015.

[3] Andries P. Engelbrecht. *Computational Intelligence An Introduction*. John Wiley & Sons, Inc, second edition edition, 2007.

[4] Randy L. Haupt and Sue Ellen Haupt. *Practical Genetic Algorithms*. John Wiley & Sons, Inc, second edition edition, 2004.

[5] Ifan G. Huges and Thomas P.A. Hase. *Measurements and their Uncertainties*. Oxford University Press, 2010.

[6] H. Mühlenbein, M. Schomisch, and J. Born. The parallel genetic algorithm as function optimizer. *Parallel Computing*, 17(6):619 – 632, 1991. ISSN 0167-8191. doi: https://doi.org/10.1016/S0167-8191(05)80052-3. URL `http://www.sciencedirect.com/science/article/pii/S0167819105800523`.

A Genetiske algoritmer

A.1 Kontinuert algoritme 1

Her ses koden til den ene af de to kontinuerte genetiske algoritmer, der er vedhæftet. Under selve algoritmen findes et eksempel på at finde minimum på Rastrigin-funktionen i tre dimensioner, da udpakningen af værdierne fra funktionen ikke er triviel.

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3
4  ### ALGORITMEN ###
5  def ga(cost_function, constraints, mutrate=0.2, X_keep=0.5,  N_pop=100,
       generations=1000, terminate=False, criteria=0.1):
6
7      # Konstanter
8      N_var = len(constraints)
9      N_keep = int(2*np.ceil(N_pop*X_keep/2))
10     N_offspring = int(N_pop - N_keep)
```

```
    N_mutations = int(N_pop*mutrate)

    # Counter til antal funktionskald
    func_call_count = 0

    # Befolkninig med N_pop kromosomer
    pop = np.random.rand(N_pop, N_var)

    # Constraints
    min_c, max_c = np.asarray(constraints).T
    diff = np.fabs(min_c - max_c)

    # Beregn forste fitness
    pop_denorm = min_c + np.multiply(pop, diff)
    fitness = np.asarray([cost_function(chromosome) for chromosome in
pop_denorm])
    func_call_count += len(fitness)

    # Udvaelgelse
    best_indx = np.array(fitness).argsort()[:N_keep]
    parents = pop[best_indx]
    pop = parents

    for i in range(generations):

        # Parring
        offspring = np.zeros((N_offspring, N_var))
        for _ in range(N_offspring)[0::2]:
            parent1 = parents[np.random.randint(N_keep)]
            parent2 = parents[np.random.randint(N_keep)]
            alpha = np.random.rand(N_var)
            beta = np.random.rand(N_var)
            offspring1 = np.clip(parent1 + alpha*(parent1 - parent2), 0, 1
)
            offspring2 = np.clip(parent2 - beta*(parent1 - parent2), 0, 1)
            offspring = np.vstack([offspring, offspring1, offspring2])

        pop = np.vstack([parents, offspring])

        # Mutationer
        rows = np.random.randint(1, N_pop, N_mutations)
        cols = np.random.randint(0, N_var, N_mutations)
        for i, j in zip(rows, cols):
            pop[i, j] = np.random.rand()

        # Beregn fitness og find bedste kromosom
```

```
        pop_denorm = min_c + np.multiply(pop, diff)
        fitness = np.asarray([cost_function(chromosome) for chromosome in
    pop_denorm])
        func_call_count += len(fitness)
        best_idx = np.argmin(fitness)
        best_individual = pop_denorm[best_idx]

        # Udvaelgelse
        best_idxs = np.array(fitness).argsort()[:N_keep]
        parents = pop[best_idxs]
        pop = parents

        # Stop algoritme, hvis kriteriet er naaet
        if terminate:
            if fitness[best_idx] <= criteria:
                yield best_individual, fitness[best_idx], func_call_count
                break

        yield best_individual, fitness[best_idx], func_call_count

### EKSEMPEL - minimum paa rastrigin ###

def rastrigin(x):
    A = 10
    dim = len(x)
    return (A*dim + sum(x**2 - A*np.cos(2*np.pi*x)))

# Antal dimensioner
N_dim = 3

# Kor algoritmen og udpak vaerdier
# Terminer algoritmen, hvis der faaes en fitness paa 0.0
params, fitness, func_calls = zip(*list(ga(rastrigin,\
                                           [[-5.12, 5.12]]*N_dim,\
                                           N_pop=25*N_dim,\
                                           generations=10000,\
                                           terminate=True, criteria=0.0)))

# Print parametre og fitness
print(f'Parametre: {params[-1]}')
print(f'Fitness: {fitness[-1]}')
print(f'Funktionskald: {func_calls[-1]}')
```

A.2 Binær algoritme

Er man interesseret i at downloade og selv køre koden, findes den på følgende link: `https://drive.google.com/open?id=1lljTx7TwWO7A9aUP_V5wHpb4mLWVeA14`

```
import numpy as np

def cost_func(p):
    """Funktionen der forsoges at minimeres med GA'en. Her i from af Rastrigin"""
    n = len(p)
    A = 10
    return A*n + sum(x**2-A*np.cos(2*np.pi*x) for x in p)

class GA:
    def __init__(self, cost_func, N_params, p_lo, p_hi, N_gene=8, N_pop=12, mutation_rate = 0.2, selection_rate=0.5):
        self.N_gene = N_gene
        self.N_pop = N_pop if N_pop%4 == 0 else N_pop+N_pop%4
        self.p_lo = p_lo              #Nedre graense for bit repraesentation
        self.p_hi = p_hi              #Ovre graense
        self.N_params = N_params      #Antal parametre
        self.N_keep = int(2*np.ceil(selection_rate*self.N_pop/2)) #Antal kromosomer beholdt, skal vaere lige
        self.mutation_rate = mutation_rate
        self.len_chromosomes = int(self.N_gene*self.N_params)
        self.cost_func = cost_func

        self.population = self.make_random_pop()
        self.costs = self.eval_cost(self.population)
        self.rank_prop = self.find_rank_prop()
        self.N_news = self.N_pop - self.N_keep

    def make_random_pop(self):
        '''Danner den tilfaeldige inertielle population'''
        population = np.random.randint(2,size=(self.N_pop,self.N_params,self.N_gene))
        return np.array([chromosome.flatten() for chromosome in population])

    def eval_cost(self,eval_population):
        '''Evaluerer cost-funktionen. Dette indebaerer at konvertere bitrepraesentationen til reelle tal.'''
        eval_population = np.array([np.split(chromosome,self.N_params) for chromosome in eval_population])
```

```
34          p_quant = np.array([[sum([gene[m-1]*2**(-m) for m in range(1,self.
    N_gene+1)])+2**(-(self.N_gene+1)) \
35                                  for gene in chromosome] for chromosome in
    eval_population])
36          cost_func_params = p_quant*(self.p_hi-self.p_lo)+self.p_lo
37          return np.array([self.cost_func(params) for params in
    cost_func_params])
38
39      def find_rank_prop(self):
40          '''Finder sandsynligheder til rank-selection. Kores kun en gang.
    Ligning 2.11 i "Haupt & Haupt".'''
41          n_sum = sum(list(range(1,self.N_keep+1)))
42          rand_prop = [(self.N_keep-n+1)/n_sum for n in range(1,self.N_keep+
    1)]
43          return np.cumsum(rand_prop)
44
45      def natural_selection(self):
46          '''Udforer en generation, dvs. samler alle funktionerne herunder.
    '''
47          population_to_keep,costs_of_old = self.find_survivors() #costs
    retuneres, da den evt. kan bruges i mate
48          mates = self.rank_select(population_to_keep)
49          new_population_members = self.mate_one_cross(population_to_keep,
    mates)
50          self.population = np.vstack((population_to_keep,
    new_population_members))
51          self.mutate()
52          new_costs = self.eval_cost(self.population[1:])
53          self.costs = np.hstack((costs_of_old[0],new_costs))
54
55      def find_survivors(self):
56          '''Finder de bedste loesninger og sorterer saa bedste er overst'''
57          selection_index = np.argsort(self.costs)
58          sorted_costs = self.costs[selection_index]
59          sorted_population = self.population[selection_index]
60          return sorted_population[:self.N_keep],sorted_costs[:self.N_keep]
61
62      def rank_select(self,pop_to_mate):
63          '''Udvaelger hvilke par, der skal parrre ved rank-selection.
64          Bedste losninger har storst sandsynlighed. Side 39 i "Haput &
    Haupt'''
65          rands = np.random.random((self.N_news//2,2))
66          mates = np.searchsorted(self.rank_prop,rands)
67          dupes_where = mates[:,0] == mates[:,1]
68          not_to_dupe = mates[dupes_where,0]
```

```
69          new_mate = [np.random.choice([x for x in range(self.N_news//2) if
    x != dupe]) for dupe in not_to_dupe]
70          mates[dupes_where,0] = new_mate
71          return mates
72
73      def mate_one_cross(self,population,mates):
74          '''Avler nye individer. Sker vha. single-point-crossover'''
75          crossover_points = np.random.randint(1,self.N_gene*self.N_params+1
    ,size=len(mates))
76          new_population_members = np.array([[np.hstack((population[parent[0
    ]][:point],population[parent[1]][point:])),
77                                             np.hstack((population[parent[1]][:point
    ],population[parent[0]][point:]))]
78                                            for parent,point in zip(mates,
    crossover_points)])
79          new_population_members = np.vstack((new_population_members[:,0],
    new_population_members[:,1]))
80          return new_population_members
81
82      def mutate(self):
83          '''Mutation. Sker ved at udvaelge tilfaeldige bits og invertere
    dem.'''
84          N_mutations = int(np.ceil(self.mutation_rate*(self.N_pop-1)*self.
    len_chromosomes))
85          i_chromosome = np.random.randint(1,self.N_pop,size=N_mutations)
86          j_base = np.random.randint(self.len_chromosomes,size=N_mutations)
87          self.population[i_chromosome,j_base] = 1-self.population[
    i_chromosome,j_base]
88
89  #EKSEMPEL:
90  max_generation = 1000
91  test = GA(cost_func=cost_func, N_params=2, p_lo=-5.12, p_hi=5.12, N_gene=1
    2, N_pop=8, mutation_rate=0.03)
92  for gen in range(max_generation):
93      test.natural_selection()
94  print(min(test.costs))
```

B Fitting til χ^2

B.1 Fit til Fizeau's eksperiment data

Herunder er koden der er brugt til at udføre minimering af χ^2 til at fitte ligning 4 til dataene. Funktionen ga er funktionen fra appendix A1.

```
### DATA ###
I = np.loadtxt('Dat_sren.txt')
I_err = np.loadtxt('Dat_fejl.txt')
pixels = np.arange(len(I))

### FIT-FUNKTION ###
def fit(x, params):
    a, b, c, k, delta, sigma, mu = params
    return np.sin(k*x+delta)**2*(b*np.exp(-(x-mu)**2/(2*sigma**2))\
           -c*np.exp(-(x-mu)**2/(2*sigma**2)))\
           +c*np.exp(-(x-mu)**2/(2*sigma**2))+a

## CHI^2 ###
def chi_square(params):
    y_model = fit(pixels, params)
    return sum(((I - y_model)/I_err)**2)

## UDFoR FITTING ###
p_opt, fitness, function_calls = zip(*list(ga(chi_square, [[0, 15],\
                                                           [150, 230],\
                                                           [130, 250],\
                                                           [0, 1],\
                                                           [0, 30],\
                                                           [30, 70],\
                                                           [250, 330]],\
                                                           generations=100
    00)))

### PLOT ###
x = np.linspace(0, len(pixels), 600)

# Plot data
plt.errorbar(pixels, I, I_err, color='tab:blue', capsize=3, label='Data')

# Plot curve_fit-resultatet
cf_params = [1.18144971e+01, 1.22075435e+02, 2.18451351e+02,\
             5.72908613e-02, -1.61860351e+00, 5.93860975e+01, 3.21019680e+
    02]
plt.plot(x, fit(x, cf_params), color='tab:orange', label='Curve fit')

# Plot GA-resultat
plt.plot(x, fit(x, p_opt[-1]), color='tab:green', label='GA-fit')

plt.xlim(0, len(pixels))
plt.xlabel('Pixel nr.')
plt.ylabel('Intensitet')
```

```
45 plt.show()
```