NTNU
Det skapende universitet

Evolusjon av feil-tolerante digitale kretser ved bruk av en beregningsklynge

May Linda Martinsen

Master i informatikk
Oppgaven levert: Juni 2007
Hovedveileder: Morten Hartmann, IDI

Norges teknisk-naturvitenskapelige universitet
Institutt for datateknikk og informasjonsvitenskap

Sammendrag

Elektroniske kretser er meget sårbare for feil. Tradisjonelle løsninger for kjøretids feiltoleranse, innebærer normalt en reprodusering av funksjonelle enheter etter kjente metoder. Ved bruk av evolusjon, med for eksempel genetiske algoritmer, er det mulig å oppnå uvanlige arkitekturer, hvorav noen kan ha fordeler innen feiltoleranse. Denne rapporten ser på evolusjon av kretser, der det til en hver tid eksisterer en inverteringsfeil. Kretser som resulterer etter evolusjonen blir så testet grundig, både med og uten feil. Gjennom et omfattende sett av eksperimenter som er utført, vil resultatene granskes og presenteres her. Resultatene vil vise at i de fleste tilfellene, vil evolusjonen, selv med feil, finne fram til en krets som vil fungere etter spesifikasjonen, når feilen ikke er til stede. Den største faktoren her er antall tilgjengelige porter under evolusjon. Er den for liten, vil ikke evolusjonen oppnå en fungerende krets, er den for stor, vil det oppstå grupper av porter som ikke har en funksjon i kretsen.

Forord

Denne masteroppgaven i Informatikk er skrevet ved Norges Teknisk- Naturvitenskapelige universitet i Trondheim. Oppgaveteksten ble formulert av min veileder, Morten Hartmann ved Institutt for Datateknikk og Informasjonsvitenskap, avdeling for komplekse datasystemer. Oppgaven er skrevet på norsk og inneholder derfor en oversikt over oversatte ord og uttrykk, så det ikke skal være tvil om forståelsen av oppgaven.

Takk til

Takk til min veileder ved instituttet, Morten Hartmann, for sin tålmodighet og fleksibilitet, for faglig veiledning og motivasjon. Takk for din effektivitet og ekstremt raske svartid. Jeg ønsker også å takke Kenneth Østby, Idar Borlaug og Knut Imar Hagen for alle svar på spørsmål angående C++ og Linux. Jeg har spart mye tid på grunn av dere. Til slutt vil jeg takke Silje Sletteng for et langt vennskap gjennom hele studietiden, uten deg hadde jeg nok ikke holdt ut i alle år.

May Linda Martinsen
25. juni 2007

Innhold

Figurer

Tabeller

Kapittel 1

Introduksjon

1.1 Introduksjon

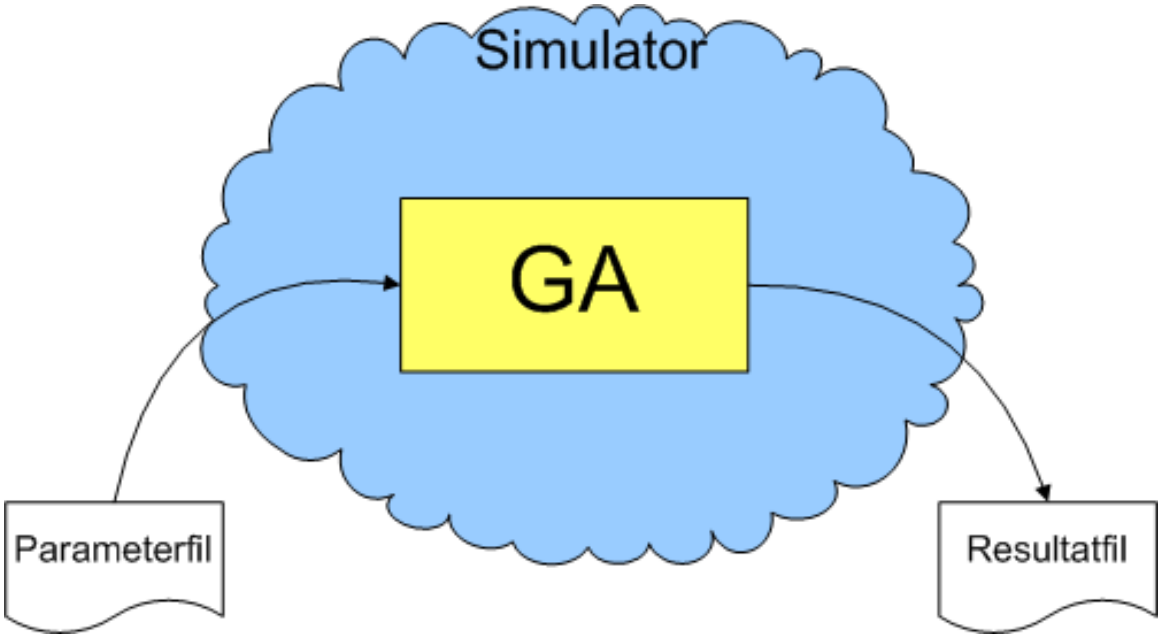
Etterhvert som kretser blir mer og mer komplekse, blir de også mer sårbar for feil, både når det gjeller menneskelige feil og produksjonsfeil. Etterhvert som gapet mellom kravet på løsninger, og evnen utviklere har ved hjelp av den teknologien som finnes blir større, brukes mer og mer kapasitet til å finne nye metoder for å utvikle komplekse kretser. Evolusjon har vist seg å være en bra metode til å utvikle nye typer kretser. Disse kretsene har også vist seg å være tolerante mot både støy og feil.

Denne oppgaven tar for seg evolusjon av kretser, i et miljø som blir simulert med singel inverteringsfeil. Oppgaven har til hensikt å se hvilke typer kretser man kan oppnå, og om man kan oppnå en fungerende krets overhodet. Evolusjonen foregår i en simulator som vises i figur 1.1. Simulatoren vil prøve å utvikle en egnet krets, som tilfredsstiller kravene som stilles i parameterfilen i form av en sannhetsverdi. Den beste kretsen, ved avsluttet simulering, vil bli undersøkt og dens egenskaper identifisert.

1.2 Oppgave

Oppgaveteksten lyder:

Elektroniske kretser er meget sårbare for feil. Tradisjonelle løsninger for kjøretids feiltoleranse innebærer normalt en replikasjon av funksjonelle enheter etter kjente metoder. Ved bruk av evolusjon (f.eks. genetiske algoritmer) er det mulig å oppnå



Figur 1.1: Simulatoren bruker en Genetisk Algoritme til å simulere evolusjon av kretser. Den mottar en parameterfil, der spesifikasjonen til hvordan algoritmen skal oppføre seg finnes, samt spesifikasjonen til individene som algoritmen skal starte med. Etterhvert som evolusjonen foregår, vil algoritmen rapportere den pågående situasjonen til evolusjonen, som legges i en resultatfil.

uvanlige arkitekturer, hvorav noen kan ha fordeler innen feiltoleranse. Et system for å eksperimentere med slike arkitekturer eksisterer, men systemet har behov for å dokumenteres mer inngående, samt at utvidelser med tilhørende nye eksperimenter kreves for å bringe forskningen videre.

Oppgaven består i følgende punkter:

- *Kartlegge og oppsummere lignende arbeider om evolusjon og feiltoleranse*
- *Dokumentere virkemåte og bruk av eksisterende simulator med tilhørende miljø for parallelle kjøringer*
- *Utvide eksisterende system til å støtte flere typer eksperimenter*
- *I den grad tiden tillater det, gjennomføre og dokumentere nye eksperimenter på den utvidede plattformen*

1.3 Simulatoren

Simulatoren ble i utgangspunktet utviklet av Frode Eskelund og Andreas Engh-Halstvedt i deres forprosjekt [1] til diplomen. Den ble da utviklet for å utvikle kretser med MOSFET transistor basert logikk. Simulatoren ble så videreutviklet av Eskelund i hans diplomoppgave [2], der simulatoren ble utvidet til å også kunne utvikle FGMOS kretser. Simulatoren ble deretter videreutviklet av Morten Hartmann i hans doktorgradsavhandling [3], der simulatoren ble brukt til undersøke feiltoleranse til kretser som evolueres.

1.4 Motivasjon

Elektroniske kretser er ekstremt sårbare for feil. I motsetning til biologiske entiteter, kan bare en feil, på en hvilken som helst komponent, gjøre hele kretsen ubrukelig. Innenfor fagfeltet bio-inspirert maskinvare, ønsker man å prøve å overføre biologiske egenskaper til elektroniske kretser. Biologiske organismer har den egenskapen at den kan fortsette å fungere tilfredsstillende, selv om den skulle få en skade. Den har også den egenskapen at den kan tilpasse seg de endringer som oppstår i det miljøet den befinner seg.

Det man ønsker å oppnå ved å bruke kunstig utvikling, er å løse problemet med "Design productivity gap". Dette er et problem som oppstår etterhvert som kretser blir mer og mer kompleks, og Computer Aided Disign (CAD) programmer med menneskelig interaksjon ikke lenger er godt nok. Dette medfører økt fare for at feil kan oppstå.

Er det mulig å lage kretser som kan fungere tilfredsstillende med en feil? Kan en krets hente seg inn etter en feil har oppstått? Dette er spørsmål man prøver å svare på.

1.5 Resultater

Resultatene fra dette prosjektet vil vise at de fleste kretser som evolueres fram med singel inverteringsfeil til stede, vil gi en fungerende multiplikator når det ikke finnes feil i systemet. Det viser videre at antall porter i kretsen er avgjørende for at dette skal være tilfellet. Det magiske tallet ligger på cirka 80 blokker. Blir antall blokker større enn dette, vil det oppstå klynger av blokker som ikke har noe funksjon i kretsen.

Endringen av egnethet gjennom evolusjonen er også sett på, og denne viser at egnetheten synker raskt og vil fort stabilisere seg. Endringen over den siste halvdelen av evolusjonen er minimal.

1.6 Problemer og utfordringer

Et problem jeg slet med i begynnelsen av dette prosjektet, var at oppgaven ikke var klart definert. Det tok ganske lang tid før det ble klart hva dette prosjektet skulle innebære. Jeg var ikke krevende nok, og mye tid gikk med til å lure på hva jeg skulle finne på. Når oppgaven ble definert og jeg viste hva jeg skulle gjøre, gikk det fort et hakk videre. Men med hensyn til oppgavens natur, tok det mellom 3 og 5 dager og kjøre ett eksperiment. Så testing av utvidelsen gjort til programmet, til nesten en måned. Noe som forsinket meg ytterligere.

Etter at programmet var testet grundig, ble det satt i gang eksperimenter, og med en gang et var ferdig ble ett nytt startet. Men tiden strakk ikke til. Det var ønskelig å kjøre flere simuleringer enn det som ble gjort. Mye somling i begynnelsen er mye av årsaken til dette problemet.

Ett annet problem, har vær naturen av den sosiale gjengen jeg har delt lesesal med. De kom på skolen, og satt seg ned til en felles frokost mellom 9.00 og 10.00, og hadde igjen en stående buff rundt kl 14.00. Alt dette skulle da foregå inne på lesesalen. Det var også minst en times diskusjon om dagen, og en og annen som var lei, og ville slå av en prat. Dette har vært et stort forstyrrelsesmoment, og jeg burde egentlig satt ned foten tidlig. Jeg har ikke vært helt uskyldig selv, men det må da finnes grenser. Jeg oppfordret til å gå ut og sette seg i sofaen rett utenfor, med dette ble ledd vekk som en spøk. Dette problemet toppet seg, mot slutten, siden alle leverte en uke før meg. De fleste satt de siste dagene og bare ventet på veileder, uten å ha noe gjøre. Jeg fikk heller ikke gjort så mye den uka.

1.7 Rapportens struktur

I kapittel 2 (Bakgrunn) vil det bli gitt et overblikk over hva evolusjon og genetikk vil si, innenfor datamaskinens verden. Delkapittel 2.1 forklarer om evolusjonsteori, mens i delkapittel 2.2, som omhandler genetisk inspirasjon, blir en del konsepter, ord og uttrykk, som er adoptert fra biologien, forklart. Delkapittel 2.3 presenterer genetisk representasjon, hvordan oppbyggingen av kretser blir lagret som et gen. Delkapittel 2.4 forklarer om feiltoleranse og noen typer feilmodeller, og delkapittel 2.5 gir en kort innføring i Trippelmodulær Redundans. Kapittelet

avsluttes med en kort forklaring om Beregningsklynger i delkapittel 2.6 og OpenPBS i delkapittel 2.7.

Kapittel 3 (Relatert/tidligere arbeider) gir en kort oversikt over relatert arbeid, og hva som tidligere er gjort. I delkapittel 3.1 vil det bli introduser noen institusjoner som driver bio-inspirert forskning. Delkapittel 3.2 presenterer hovedaktørene innen feiltoleranse ved evolusjon, mens delkapittel 3.3 presenterer en aktør som ser på feiltoleranse gjennom redundans. I delkapittel 3.4 introduseres en aktør som har sett på metoder for å effektivisere evolusjonær maskinvare. Delkapittel 3.5 presenterer navn innen representasjon av gener, og hvordan de avbildes til individer, og kapitlet avsluttes med en alternativ måte innen evolusjon i delkapittel 3.6.

Kapittel 4 (Simulatoren) gir et detaljert innblikk i simulatoren og systemet som hører med. Dette kapitlet er delt inn i tre deler. Den første, delkapittel 4.1, forklarer hvordan systemet fungerer praktisk sett, mens den andre delen, delkapittel 4.2, forklarer hvordan systemet fungerer teknisk sett. Disse to delkapitlene beskriver systemet som det var når dette prosjektet ble påbegynt. Delkapittel 4.3 forklarer hvilke endringer som er utført på systemet i løpet av dette prosjektet.

Kapittel 5 (Eksperimenter) beskriver simuleringene som er utført. I delkapittel 5.1 beskrives grunnlaget for hvordan eksperimentene er utført. Delkapitlene 5.2, 5.3, 5.4 og 5.5 beskriver de fire settene med eksperimenter som ble utført.

Kapittel 6 (Resultater) oppsummerer resultatene fra simuleringene som ble utført, med en påfølgende diskusjon om hva som kan ha skapt disse resultatene. Delkapitlene 6.1, 6.2, 6.3 og 6.4 beskriver resultatene fra de fire settene med eksperimenter som ble utført, mens delkapettel 6.5 ser nærmere på resultatene til de individueller kjøringene. Delkapittel 6.6 ser deretter på hvordan egnetheten har endret seg over hele evolusjonen.

Kapittel 7 (Konklusjon og videre arbeid) gir en konklusjon av arbeidet i delkapittel 7.1, og delkapittel 7.2 foreslår hva som kan arbeides med videre.

Vedlegg A inneholder en ordliste med oversetting til engelske ord. Ettersom denne rapporten skrives på norsk og mange ord og uttrykk som brukes innenfor fagfeltet, ikke har en standard oversetning, er dette vedlegget tatt med for at ingen ord skal misforstås.

Vedlegg B inneholder en liste over alle forkortelser som er brukt i rapporten. Hver forkortelse er skrevet ut i sin helhet ved første gangs bruk, deretter er forkortelsen brukt. Dette kapitlet skal være til hjelp for leser, dersom leser ønsker en påminnelse om hva forkortelse var.

Vedlegg C er programkode og skript som er produsert eller oppdatert under dette prosjektet.

Kapittel 2

Bakgrunn

I dette kapitlet vil grunnkonseptene for bio-inspirert evolusjon bli presentert. Delkapittel 2.1 forklarer om evolusjonsteori, mens i delkapittel 2.2, som omhandler genetisk inspirasjon innenfor datamaskinverdenen, blir en del konsepter, ord og uttrykk, som er adoptert fra biologien, forklart. Delkapittel 2.3 presenterer genetisk representasjon, hvordan oppbyggingen av kretser blir lagret som et gen. Delkapittel 2.4 forklarer om feiltoleranse og noen typer feilmodeller, og delkapittel 2.5 gir en kort innføring i Trippelmodulær Redundans. Kapittelet avsluttes med en kort forklaring om beregningsklynger i delkapittel 2.6 og OpenPBS i delkapittel 2.7.

2.1 Evolusjon

Den mest anerkjente av 1800-tallets evolusjonsteoretikere, var Charles Darwin (1809-1882). Han brukte mesteparten av livet sitt til sine teorier, men det var ikke før i 1859 han kom med boken *Artenes opprinnelse* [4], der han introduserte sin teori om naturlig evolusjon. Teorien baserte seg på at planter og dyr hadde stor kontinuerlig variasjon, og produserte mer avkom enn det som overlevde til reproduktiv alder. Avkommet konkurrerte om en begrenset mengde ressurser. Naturlig seleksjon (naturlig utvalg) ga utryddelse av lite tilpassede individer og arter. De best tilpassede individene fikk avkom, og fordelaktige egenskaper fra foreldrene ble overført til avkommet. Drivkraften i evolusjonen er naturlig seleksjon som gjør at de best tilpassede organismene overlever og reproduserer seg.

Darwins evolusjonsteori gir en naturlig forklaring på artsrikdommen på jorda, og har betydd like mye for biologene som Newtons gravitasjonsteori for fysikerne.

Evolusjonsteorien gjorde biologi, studiet av livet og livsformene, til en anerkjent naturvitenskapelig disiplin på linje med fysikk og kjemi.

I Darwins evolusjonsteori om naturlig utvalg, er egnethet et mål på sannsynligheten for at et individ skal overleve og reprodusere seg. Plantene som er best tilpasset miljøet blir valgt ut. De fleste mutanter har redusert egnethet og fjernes fra populasjonen, men det motsatte kan også skje. Disse elementene kan kalles feiltoleranse og tilpasningsdyktigheten til en organisme.

I naturlig sammenheng vil gener føres vider og blandes gjennom reproduksjon. Foreldre viderefører sine gener til sine barn, som sitter igjen med noen gener fra hver forelder.

2.2 Genetisk inspirasjon

Tradisjonelle programmer og algoritmer er lagd for å løse et gitt problem. Hvis grunnlaget for problemet endrer seg, kan det hende at programmet ikke lengre er brukbart til det formålet det var lagd til. Bio-inspirert utvikling er et relativt nytt fagfelt, som prøver og gjøre noe med dette.

En Evolusjonær Algoritme(EA) er en samling teknikker som baserer seg på evolusjonære prinsipper [24]. Hovedklassene innenfor EA er Genetisk Programmering (GP), Genetiske Algoritmer (GA), Evolusjonære Strategier (ES) og Evolusjonær Programmering (EP). GP og GA blir forklart i mer detalj i seksjon 2.2.1 og 2.2.2, men ES og EP faller utenfor omfanget av dette prosjektet.

Et av problemene som prøves å løses av kunstig evolusjon, er "Design productivity gap". Dette er et problem som oppstår etterhvert som kretsene som produseres blir mer og mer kompleks, og Computer Aided Design(CAD) med Menneske-Maskin-Interaksjon, ikke lengre godt nok til å produsere disse kretsene. Kompleksiteten på VLSI(Very Large Scale Integration) og ULSI(Ultra Large Scale Integration) blir for stor for menneskelig forståelse, og det kan i stor grad oppstå feil.

Det er utviklet tre metoder for kunstig evolusjon for design av maskinvaresystemer, intern(eng:intrinsik) evolusjon, ekstern(eng:Extrinsik) evolusjon og Full Maskinvare Evolusjon(CHE, Complete Hardware Evolution) [8]. De to førstnevnte metodene utfører evolusjon i programvare, mens tester blir, for intern evolusjon, utført på maskinvare, og ekstern evolusjon i programvare. CHE utfører all evolusjon og testing i maskinvare.

2.2.1 Genetisk Programmering

GP er en automatisk metode for å lage eksekverbare programmer, eller en annen form for algoritme for å løse et problem [5]. Metoden starter med et gitt sett med tilfeldig genererte programmer, og bruker genetiske operatorer til å generere nye programmer. De to viktigste genetiske operatorene er mutasjon og krysning, som blir forklart i neste seksjon.

2.2.2 Genetiske Algoritmer

GA er basert på prinsippet til naturlig evolusjon. Det finnes ingen enighet i miljøet når det kommer til definisjon av GA, men det finnes et sett elementer som går igjen:

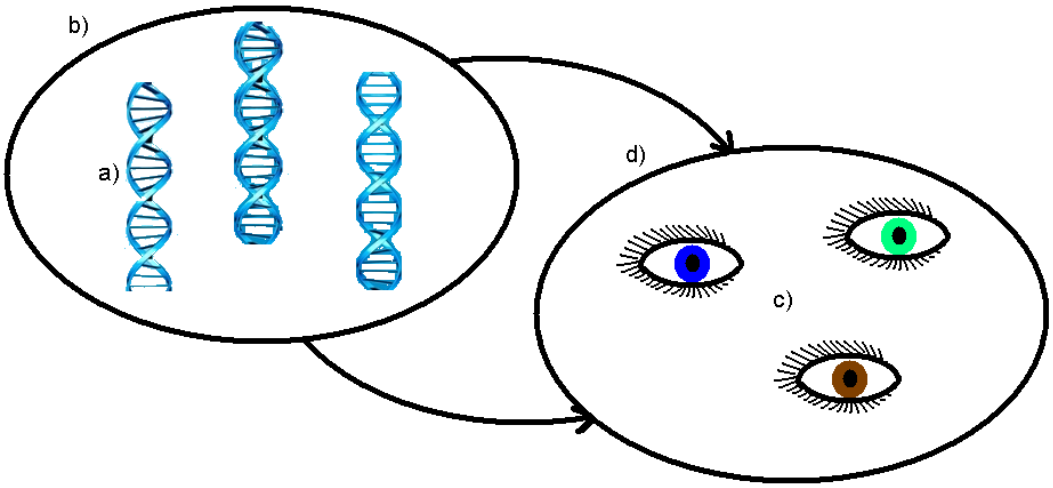
- En populasjon
- Seleksjon basert på egnethet
- Krysning
- Mutasjon

Kunstig evolusjon låner terminologien fra naturlig evolusjon. En populasjon består av et gitt sett med individer, som man ser på innenfor rammen av evolusjonen. Hvert individ består av et sett kromosomer, som kan for eksempel representeres binært. Metoden som velger ut hvilke av individene fra en generasjon som skal bli foreldre i neste, kalles seleksjon.

Uttrykket genom refererer til kolleksjonen av genetisk materiale som finnes tilgjengelig, og genotype refererer til settet av gener i genomet. Figur 2.1 viser relasjonen mellom genom, genotype, fenom og fenotype. Genotypen er byggeklossene til fenotypen, som man kan beskrive som egenskapene til organismen, og alle egenskapene som kommer fra genomet kan kalles fenomet.

Egnethet

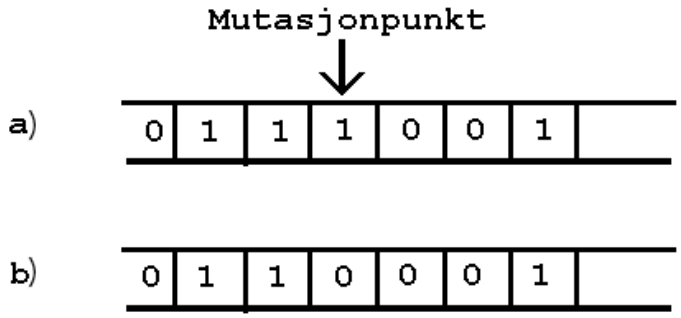
For at den genetiske algoritmen skal kunne skille mellom individer som er godt og dårlig egnet, trenger den en mulighet til å gi individene en poengsum. Denne poengsummen gjenspeiler individets egnethet. Egnetheten indikerer hvor korrekt et individ er, og desto mer egnet individet er, desto større sjanse her det for å bli valgt som forelder til neste generasjon.



Figur 2.1: Genotypen b) består av flere genom a). Egenskapen til hvert genom er fenomet c) og settet av alle fenomene er fenotypen d)

Mutasjon

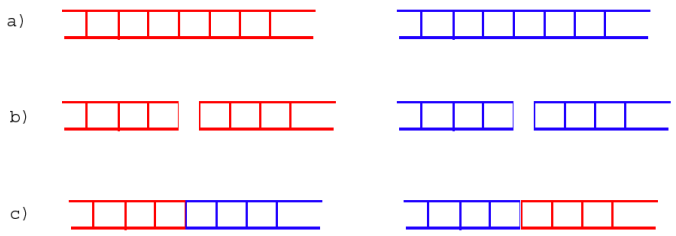
Mutasjon skjer på et individ med en sannsynlighet p_m. Hvis sjansen står mutasjon bi, vil ett av genene i individet forandres. Figur 2.2 viser hvordan mutasjon kan foregå på et individ der kromosomene blir representert binært.



Figur 2.2: Et individ a) før mutasjon, og det samme individet b) etter mutasjon.

Krysning

Krysning er den genetiske operasjonen som ligner mest på komplekse organismers reproduksjon. Krysning skjer ved at to individer blir delt ved et tilfeldig punkt i kromosomrekken, og den ene delen blir byttet om. Dette produserer to helt nye individer. Krysning skjer på to individer med sannsynlighet p_c. I dette prosjektet er punktet for deling det samme for begge individene. Dette blir vist i figur 2.3.



Figur 2.3: a) to utvalgte individer, som b) deles ved et tilfeldig punkt, og c) bytter høyre del av individet, for å skape to nye individer.

Seleksjon

Seleksjonen bestemmer hvilke av individene fra en generasjon, som blir valgt ut til foreldre i neste generasjon. Det finnes mange typer seleksjon, men noe de fleste har felles er at de bruker en kombinasjon av tilfeldighet og sannsynlighet for å velge ut individene til neste generasjon. Mange bruker også seleksjonspress, som gir det beste individet en større sjanse for å bli valgt ut.

I dette prosjektet brukes Tournament Seleksjon(TS). Denne typen seleksjon baserer seg på å velge ut et subsett av n individer, og la disse konkurrere om å bli valgt videre. Basert på en sannsynlighet p_s, for å velge ut det beste individet i gruppen, blir et av individene valgt. Hvor mange individer subsettet skal inneholde, og sannsynlighetsverdien for å velge beste, blir definert i parameterfilen. Parameterfilen blir forklart nærmere i seksjon 4.2.1. Sannsynlighet for krysning og mutasjon er også definert i parameterfilen.

TS fungerer på følgende måte:

1. Velg ut en gruppe på n tilfeldige individer fra denne generasjonen, som skal delta i en turnering om å bli mulige foreldre i neste generasjon.

2. Ved bruk av sannsynlighetsverdien p_s, velges ett av disse på en av to følgende måter.

   - Det individet som er best egnet velges.

   - Ett tilfeldig individ blir valgt.

3. Repeter steg 1 og 2 en gang.

4. Basert på sannsynlighetsverdien p_c, blir det utført krysning på foreldrene.

5. Basert på sannsynlighetsverdien p_m, blir det utført mutasjon på foreldrene.

6. Start på toppen igjen til det er valgt ut like mange individer til neste generasjon som det var i denne generasjonene.

Etter at den nye generasjonen er funnet, må den evalueres og hvert individ i generasjonen får beregnet sin egnethet.

2.2.3 En enkel GA

Med et definert problem, og en bitstreng-representasjon av kromosomene, kan en enkel GA operere på denne måten:

1. Algoritmen starter med en populasjon på n individer, der hvert individ har tilfeldig generterte kromosomer.

2. For hvert individ beregnes egnetheten.

3. Basert på hvilken seleksjonsmetode som brukes, velges det ut to individer fra populasjonen som blir foreldreindivider.

4. Med en sannsynlighet på p_c blir individene krysset, og med en sannsynlighet på p_m muteres individene.

5. De to individene blir så lagt inn i populasjonen til den nye generasjonen.

6. Prosedyren blir gjentatt, fra punkt 3, til den nye generasjonen har like mange individer som den gamle.

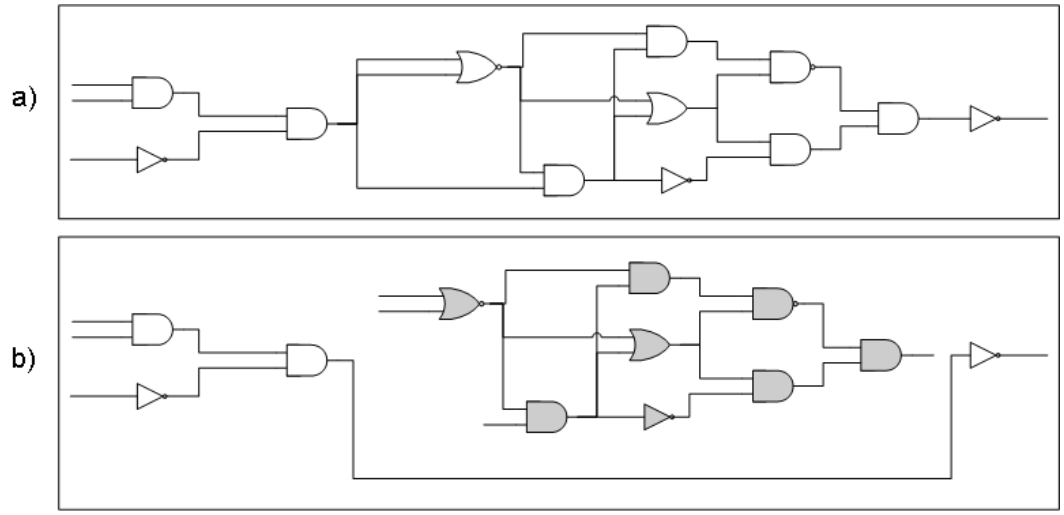
7. Start på nytt fra punkt 2, med å kalkulere egnetheten til individene i den nye generasjonen. Dette gjøres til et gitt antall generasjoner er utført, eller det blir funnet et individ med en egnethet som indikerer at en løsning er funnet.

En gjennomgang av denne prosedyren kalles en generasjon, og hele settet med generasjoner kalles en kjøring. Etter endt kjøring er det forhåpentlig vis et individ som innfrir ifølge forventningene.

Denne prosedyren beskriver kun basisen for GA-applikasjonen. Algoritmen er avhengig av noen parametere som må følge med når den kjøres. Disse parametrene kan være blant annet populasjons størrelse, mutasjon- og krysningssannsynlighet og antall generasjoner per kjøring.

2.3 Genetisk representasjon

Det finnes mange måter å representere et genom på, og noen fungerer bedre enn andre. For å nevne noen, så har man for eksempel bitstreng, flerdimensjonel matrise, formler eller nettliste. I dette prosjektet blir genomet representert ved bruk av nettliste, men en stor ulempe med dette er at en enkel mutasjon, kan endre veldig mye på kretsen som en helhet. Figur 2.4 viser hvordan hele kretsen kan omdannes på grunn av dette.



Figur 2.4: a) viser hvordan en nettliste ser ut før mutasjon, og b) hvordan den kan se ut etter mutasjon. Med bare et signalskifte har nesten hele kretsen blitt forbikoblet.

2.3.1 Kartesisk Genetisk Programmering

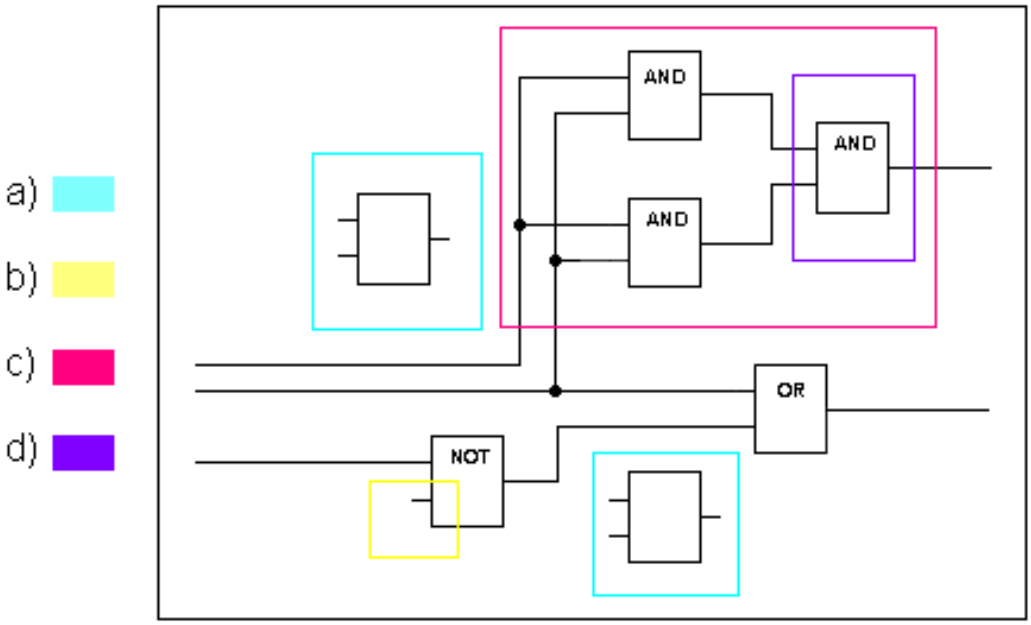
Kartesisk Genetisk Programmering(CGP, Cartesian Genetic Programming) [6] er en metode der genotypen representeres som en liste av integer, og avbildes til en rettet graf. Genotypen er av fast lengde, mens fenotypen er avhengig av hvilke gener som brukes. En fordel med dette, er at det gjør det mulig for flere genotyper til å avbildes til samme fenotype, som gir nøytralitet. Med nøytralitet menes at det eksisterer flere genotyper med samme egnethet, noe som kan forventes på grunn av redundansen som finnes i systemet [7]

Som nevnt tidligere gjør CGP det mulig å avbilde flere genotyper til samme fenotype, og den største grunnen til det, er at det forekommer en hel del redundans, som vistes i figur 2.5. Den kan, ifølge [6], deles inn i tre grupper:

- Redundans av blokker

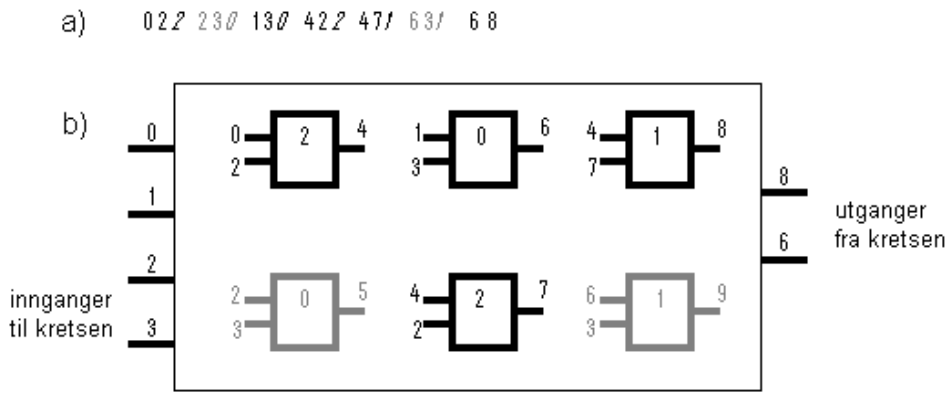
- Funksjonell redundans
- Redundans av innsignaler

I et individ er det tilgjengelig et gitt antall blokker, men det er ikke alle blokker som til en hver tid brukes. De blokkene som da ikke er i bruk gir redundans av blokker. Funksjonell redundans får man når en implementert subkrets kan implementeres med færre blokker. Noen blokker trenger ikke like mange innsignaler som andre, og det er i disse tilfellene, når man får en ekstra signallinje at man har redundans av innsignal.



Figur 2.5: a) viser overflod av blokker og b) viser overflod av innsignaler. c) viser en subkrets som har funksjonell overflødighet, siden subkretsen i d) har en ekvivalent funksjon.

I CGP blir blokkene rangert i et rektangulært grid, der utgangen på blokkene blir nummerert kolonnevis. Inngangene blir nummerert 0 til n-1, der n er antall innganger, og blokkene blir nummerert fra n og oppover. Inngangsportene kan kobles til hvilken som helst blokk, men blokkene han kun koble seg til en annen blokk som er i en kolonne til høyre for seg selv. For hver blokk tas det vare på hvilken funksjon blokken har, og fra hvilken kilde inngangsverdiene kommer fra. Figur 2.6 viser et eksempel med 6 blokker, men to innganger til hver blokk, 4 inngangsporter til kretsen og to utganger. I dette prosjektet brukes CGP for å representere genotyper, men går bort ifra kolonneoppdelingen av fenomet. Det brukes isteden en vanlig "feed forwards"-struktur, der signalet fra en blokk, kan sendes til hvilken som helst blokk som har et høyere nummer.



Figur 2.6: Genotype-Fenotype avbildning. Genonotypen a) avbildes til Fenotypen b). Hvert sett av tre tall, fra genotypen, representerer en blokk i fenotypen. De første to tallene i genotyper representerer inngangene i fenotypen, mens det siste tallet i genotypen er funksjonen til blokken, som representeres inni blokken i fenotypen. Kretsen har 4 innganger og 2 utganger, og den har tre forskjellige funksjoner. De grå tallene, i genotypen, og blokkene, i fenotypen, representerer noder som ikke er tilkoblet kretsen.

I genotypen lagres det for hver blokk, i gitt rekkefølge, hvilke blokker inngangene kommer fra, og hvilken funksjon blokken har. Funksjonen er markert med kursiv i genotypen i figur 2.6a) og er markert inni blokken i Fenotypen i figur 2.6b). Til slutt i genotypen er blokkene som representerer utgangene lagret.

2.4 Feiltoleranse

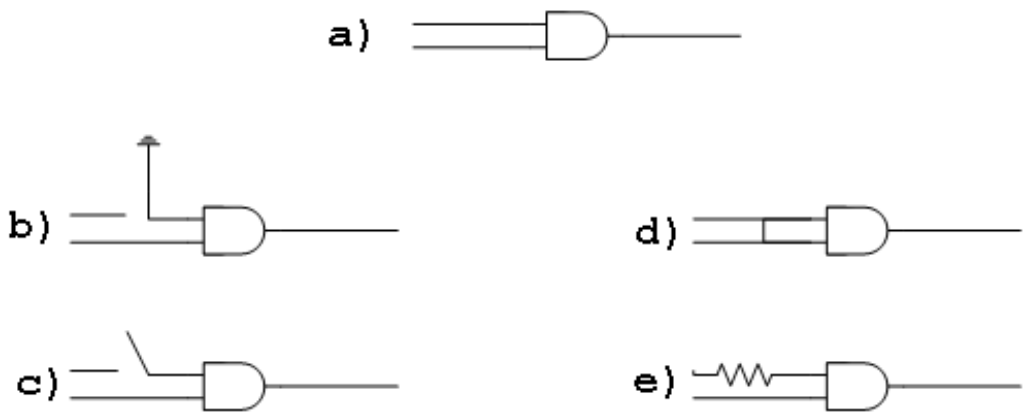
Etter hvert som kompleksiteten til system, og antall transistorer øker, vil behovet for feiltolerante systemer være et stadig økende problem. Feiltoleransen er evnen et system har til å fungere med feil, eller muligheten til å innhente seg hvis en feil oppstår. Dette er viktig for påliteligheten til systemet. Mye av forskingen på dette, går ut på å oppdage og korrigering feil. Teknikkene som brukes er prinsippet om overflødighet. Det kan være overflod av maskinvare, tid, informasjon eller programvare.

Biologiske organismer er, i motsetning til elektroniske kretser, veldig tolerante overfor feil og endringer i omverden. Biologiske systemer vil i de fleste situasjoner fortsette og fungere på helt vanlig måte, selv om det skulle oppstå en feil. Det er denne egenskapen forskere innen området, ønsker å overføre til elektronikken, ved hjelp av evolusjon.

2.4.1 Feilmodeller

Målet med feilmodellering er å finne ut hva som skjer ved feil, og hvor feil kan oppstå. Dette kan gjøres ved å modellere feil tidligst mulig i utviklingen, noe som medfører at man reduserer antall feil man trenger å ta hensyn til senere. Det redusere også kompleksiteten til testingen av enheten senere. Det finnes fire typer feilmodeller [21, 25]:

- Stuck-at feil
- Stuck-open feil
- Foreningsfeil
- Forsinkelsesfeil



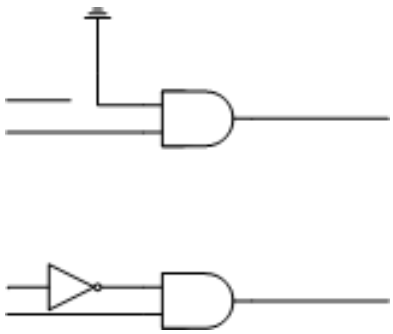
Figur 2.7: a) er kretsen som skal feilmodelleres, etter stuck-at-modellen b), stuck-open-modellen c), foreningsmodellen d) og forsinkelsesmodellen e).

Stuck-at feilmodell

Denne typen feilmodell kan deles opp i to undergrupper; singel og mangfoldig.

Med singel stuck-at-modellen, vil det kun oppstå en permanent feil i kretsen. Feilen oppstår på et signal, så funksjonen av den logiske kretsen vil ikke være påvirket av feilen. Feilen består i at signalet blir enten festet til logisk høy eller logisk lav. Dette er den mest brukte av feilmodellene, og grunnen til det er at det finnes så mange fordeler med akkurat denne feilmodellen. De største fordelene er at man kan avdekke opp til 90% av produksjonsfeil, og algoritmer for test-generering og feilsimuleringer er velutviklet og effektive.

Ved bruk av vanlig stuck-at, vil feilen som oppstår ha en 50% sannsynlighet til å gi en verdi som faktisk er den korrekte. For å teste hva resultatet blir i verste fall, brukes det i denne oppgaven invertering av det eksisterende signalet. Forskjellen mellom tradisjonell stuck-at og inverteringsfeil vises i figur 2.8.



Figur 2.8: a) viser den tradisjonelle måten å modellere stuck-at-feil. b) viser hvordan det blir utført i dette prosjektet.

Mangfoldig stuck-at-modellen er veldig lik den single, men her har man to eller flere feil samtidig. Denne typen feilmodell er litt mer kompleks, og ikke fullt så velutviklet som den single modellen, og den fanger ikke opp like mange feil. Men man vil øke feildeteksjon ved å forene bruken av singel- og mangfoldig stuck-at feilmodellering.

Stuck-open feilmodell

Denne typen feilmodell simulerer hva som skjer hvis en fysisk kobling bryter. Dette medfører at signalet hverken er knyttet til logisk høy eller logisk lav. Man får en hukommelseseffekt, der signalet vil bli husket i perioden det tar å lade ut signalet.

Foreningsfeilmodell

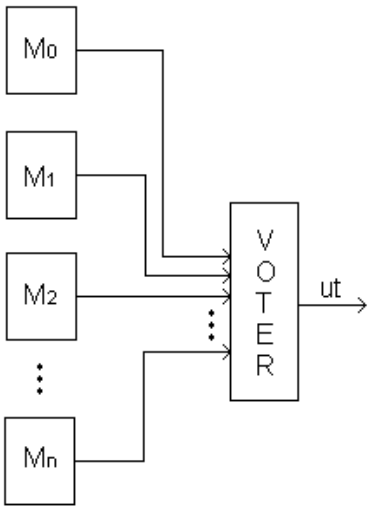
Denne typen feilmodell simulerer hva som skjer hvis det oppstår en forening mellom to elementer i kretsen. Denne feilmodellen har tre klasser; forening innad i et logisk element, forening av signaler men feedback og forening av signaler uten feedback. En stor fordel med denne typen feilmodellering er at den dekker en stor andel feil, ettersom foreningsfeil er beregnet til å utgjøre ca. 30% av alle feil som oppstår.

Forsinkelses-feilmodell

I denne typen feilmodellering ønsker man å finne ut hva som skjer når det oppstår en forsinkelse i kretsen. Man har to typer forsinkelse; blokkforsinkelse og stiforsinkelse. Selv om kretsen ikke har feil i seg selv, kan forsinkelse i et element i kretsen medføre at en funksjon feiler. Signalet kan rett å slett komme for sent.

2.5 Trippel Modulær Redundans

Trippel-Modulær Redundans (TMR) er en klassisk metode innen statisk maskinvareredundans. Konseptet ble først forslått av Neumann [10], og kan utvides til N-modulær redundans (NMR). I TMR har man tre like kretser og en velger, med mulige feil. Den kan håndtere et vilkårlig antall feil, så lenge feilene kun oppstår i delkretsene, og ingen feil forekommer i velgeren. Ut ifra de signalene den har mottatt, velger den ut for hver posisjon i bitstrengen, den verdien som flertallet av individene er representert med. Sannhetstabellene til denne utvelgelsen vises i tabell 2.1. Figur 2.9 viser dette med en NMR-krets.



Figur 2.9: N-modulær redundans. Velgeren tar imot utsignalene fra de n kretsene og velger ut hver bit i forhold til flertallet av innsignalenes verdi.

Ved singel inverteringsfeil vil en krets av type TMR fungere feilfritt i de tilfellene feilen oppstår i en av de tre delkretsene. Dette vil gi kretsen en høy pålitelighet, og det er et godt sammenligningspunkt for de kretsene som evolueres med inverteringsfeil.

Z0	Z1	Z2	Zut
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Tabell 2.1: Sannhetstabell for TMR. For hvert bit i hver av de tre strengene, vil de tre bitene sjekkes og den verdien som har flertallet på sin side, vil bli valgt til å representere løsningen.

2.6 Beregningsklynge, Clustis2

Til utføring av eksperimenter i dette prosjektet, ble det brukt en beregningsklynge med en beowulf konfigurasjon [22]. Klyngen heter clustis2.idi.ntnu.no [23], og består av 23 datamaskiner, hvorav 22 er beregningsnoder og en styrenode (eng:frontend). Styrenoden tar seg av pålogging og fordeling av arbeid på beregningsnodene. Nodene er koblet sammen med en gigabit Ethernet switch, og hver node er av typen DELL 750, med 1 GB minne og 3.4GHz Pentium IV prosessor.

2.7 OpenPBS

OpenPBS er et batchkøsystem for bruk på forskjellige UNIX-plattformer, som er koblet sammen via et nettverk. Hensikten med OpenPBS er å tilføre mer kontroll over utføringen av batchjobber, blant annet initiering og utføring av jobber. Den gir også mulighet for fordeling av jobber på forskjellige noder, og om nødvendig omfordeling.

Systemet gir også veldig mye valgfrihet i forhold til hvordan resursene skal brukes. Blant annet hvilke jobber som skal ha tilgang til hvilke noder, og tidsbegrensninger. Systemet fungerer etter først-inn-først-ut-modellen, og kommunikasjonen er basert på tjener-klient-modellen. Klientene ber om å få utført en jobb, og tjeneren utfører den på vegne av klienten.

Noen av funksjonalitetene til OpenPBS er:

- Jobbprioritering

  - Brukere kan spesifisere hvilken prioritet jobben har, eventuelt kan man bruke standardinnstillinger.

- Jobbavhengighet
  - Brukere kan definere avhengigheter mellom jobbene, som utførelsesrekkefølge og synkronisering.

- En eller flere køer
  - OpenPBS kan konfigureres med så mange køer som ønskes.

For å kjøre en jobb på en beregningsklynge, oppretter man en fil der man, blant annet, spesifiserer hvor mange beregningsnoden man trenger for å utføre jobben, i hvilken kø den skal legges og hvor lang tid jobben trenger å kjøre. For å gjøre det klart at dette er PBS-variabler, legges de etter kommentaren *# PBS*. Sist i filen legges kommandoen for å kjøre et program. En PBS-fil kan se ut som figur 2.10.

```
#PBS -l walltime=00:01:00,mem=10000kb
#PBS -l nodes=1
#PBS -q default@clustis2
#PBS -N ehw_1
./ehw ehw.cfg_1 0 0 > out_1
```

Figur 2.10: De fire første linjene i filen setter PBS-variabler. Den første linjen definerer at jobben trenger en time kjøretid, og maks 10 000Kb minnet. Linje 2 forteller at jobben trenger en beregningsnode, mens linje 3 definerer i hvilken kø jobben skal legges. Linje 4 setter navnet på jobben. Den nederste linjen i filen, setter i gang programmet som skal utføres. I dette tilfellet skal programmet *ehw* kjøres med parametrene ehw.cfg_ 1, 0 og 0, og utskriften fra programmet skal legges i filen out_1.

For å legge jobben i en kø på beregningsklyngen, brukes kommandoen:

- qsub <PBS-fil>

Kapittel 3

Relaterte/tidligere arbeider

I dette kapitlet vil det bli presentert pågående- og tidligere arbeid som er relevant for denne oppgaven. I delkapittel 3.1 vil det bli introduser noen institusjoner som driver bio-inspirert forskning. Delkapittel 3.2 presenterer hovedaktørene innen feiltoleranse ved evolusjon, mens delkapittel 3.3 presenterer en aktør som ser på feiltoleranse gjennom redundans. I delkapittel 3.4 introduseres en aktør som har sett på metoder for å effektivisere evolusjonær maskinvare. Delkapittel 3.5 presenterer navn innen representasjon av gener, og hvordan de avbildes til individer, og kapitlet avsluttes med en alternativ måte innen evolusjon i delkapittel 3.6.

3.1 Biologisk inspirert forskning

NTNU har siden slutten av 90-tallet hatt en kultur innen bio-inspirert forskning, både intern og ekstern evolusjon. Gruppen som holder på med dette kaller seg CRAB Lab (Complex, Reconfigurable, Adaptive, Bio-inspired hardware), og blir ledet av Pauline Haddow. CRAB Lab driver med forskning innen følgende:

- Evolusjon av maskinvare(EHW, Evolutionary Hardware)
- Kunstig utvikling
- Selvorganisering
- Feiltoleranse
- Rekonfigurerbar teknologi
- Fremtidens teknologi

- Cellesignalering

De mest relevante for dette prosjektet er EHW og feiltoleranse, og nøkkelspillerne her er Morten Hartmann og Asbjørn Djupdal. Deres forskning blir presentert nærmere i henholdsvis seksjon 3.2 og 3.3

Intelligent System Group ved universitetet i York har siden slutten av 90-tallet arbeidet med forskning innen bio-inspirert forskning. Med Andrew Tyrrell i spissen, har denne forskningsgruppen sett på blant annet, områdene EHW, feiltoleranse og GP.

Som en del av disse områdene har de forsket både på ekstern- [17] og intern EHW [18]. De forsker også på området feiltoleranse, innenfor forskjellige utviklingsmodeller [19].

3.2 Feiltoleranse og EHW

Morten Hartmann leverte i 2005 sin doktorgradsavhandling på evolusjon av feil- og støytolerante digitale kretser [3]. Dette prosjektet bygger videre på den avhandlingen, som er en samling av syv artikler.

1. "Untidy Evolution: Evolving Messy Gates for Foult-Tolerance"
2. "Evolving Foult Tolerance on an Unreliable Technology Platform"
3. "Evolving Robust Digital Designs"
4. "Evolution of Foult-Tolerant and Noise-Robust Digital Designs"
5. "Development and Complexity-based Fitness Function Modifiers"
6. "Evolved Digital Circuits and Genome Complexity"
7. "The Genotypic Complexity of Evolved Foult-tolerant and Noise-robust Circuits"

De fire første artiklene ser på evolusjon av kretser i miljø med støy og feil. Her blir det sett på komponenter og teknologi som brukes, samt hvordan kretsene evoluere i et miljø med støy og feil. I de tre siste tar for seg skaleringsproblemet innenfor kunstig utvikling. De ser på representasjon av kretsene, og hvordan de avbildes.

I artikkel 1 er spørsmålet som man ønsker å få svar på, er hvilke typer komponenter bør brukes i kunstig utvikling. Det viser seg ut av eksperimenter som er utført,

at kunstig utviklede kretser har en naturlig evne til å fungere med støy og feil. Artikkel 2 går videre på artikkel 1, ved å bruke teknologispesifik utvikling, og da spesifikt Complementary Metal Oxide Semiconductor (CMOS). Etter utførte eksperimenter viste det seg at denne typen teknologi egner seg godt til kunstig utvikling. I artikkel 3 ses det på robustheten til kretser evoluert i et miljø med støy og feil. Disse kretsene ble videre testet i forskjellige andre miljø med støy og feil. Det ble her konkludert med at støy ikke hadde så mye å si ved utvikling av kretser, men at det heller kan hjelpe med litt støy. Artikkel 4 går videre med evolusjon av kretser, men her med mer biologisk inspirert operasjoner. Denne artikkelen viser at evolusjon er et godt egnet redskap for å produsere kretser, som både er robuste overfor støy og tolerant for feil.

Artikkel 5 ser på hvordan man beregner egnetheten til kretser, og hvordan kretsene utvikler seg. Eksperimenter ble utført med en direkte egnethetskoding og en utviklingsavbildning. Det ble konkludert med at avbildningen hadde en fordel ved kunstig utvikling. I artikkel 6 blir det sett på kompleksiteten av genotyper og komprimering av søkerommet. Med den komprimeringen som ble gjort viste det seg at den fungerte som utviklingskartlegging, og det kunne utnyttet i evolusjonsalgoritmen. I artikkel 7 blir det sett på hvordan feiltoleranse av en krets innvirker på kompleksiteten til genotypen. Denne artikkelen viser at desto høyere feiltoleranse en krets har, desto mindre kompleks er kretsen.

Som en overordnet konklusjon til denne avhandlingen, ble det nevn at begrensningene til noen av de observasjonene som er gjort i denne avhandlingen, er at de er knyttet til de spesifikke undersøkelsene som er gjort i denne avhandlingen. Den går videre med å uttrykke at denne typen evolusjon egner seg mest til små problemer med lite kompleksitet og en enkel evolusjonsalgoritme.

En av de som blir betraktet som pionerer innen feiltoleranse og EHW, er Adrian Thompson. Han startet på midten av 90-tallet med å se på evolusjon av feiltolerante systemer [11]. Thompson tilhørte avdelingen for robotteknikk, så hans forskning er strekt knyttet til intern evolusjon. Han har også forsket på hvilke typer kretser som kan evolueres med EHW [12, 13], blant annet noen typer kretser tradisjonelle metoder ikke kan evoluere [14].

3.3 Feiltoleranse gjennom redundans

Asbjørn Djupdal driver for tiden med sin doktoravhandling, der han, blant annet, har sett på hvordan man kan oppnå feiltoleranse gjennom å evoluere fram kretser med redundans [27, 28]. Djupdal har i [27] sett på om det kan skapes redundans av blokker i kretsene, ved å evoluere kretsene med to typer feilmodeller. De to feilmodellene som er brukt er the gate reliability model" og singel feil modellen. Det

viste seg at den førstnevnte modellen ikke ga noen redundans. Singel feil modellen viste seg å skape en redundans av blokker, men ved nærmere undersøkelser viste det seg å være utilgjengelige blokker. Dette ble sett videre på i [28].

3.4 Metoder for å effektivisere EHW

Rundt årtusenskiftet drev Tatiana Kalganova med en del forskning, der hun så på metoder for å forbedre effektiviteten til EHW [29, 30]. I [29] ser hun på om det er mulig å evoluere fram en kompleks krets ved å dele opp kretsen i mindre problemer. Denne metoden kan beskrives som en treløsning, der de oppdelte problemene kan igjen deles inn i mindre deler. Evolusjonen operere på hver del, og setter subkretsene sammen hvis en tilfredsstillende løsning er funnet. Dette viste seg at denne metoden ga en enorm forbedring av evolusjonen.

I [30] ser hun på en alternativ måte for hvordan blokker kan representeres. Ved å representere blokkene i en med en multi-inn- og multi-ut-verdier kunne dette brukes til å syntetisere både binære- og multiverdi logiske funksjoner. Dette viste seg at denne metoden ga en vesentlig forbedring av ytelse.

3.5 Genetisk representasjon og avbildning

En av nøkkelpersonene på dette området er Julian Miller. Han har, i samarbeid med blant annet Vasselin Vassilev og Peter Thomson, utviklet metoden CGP, som ble forklart i detalj i seksjon 2.3.1. Forskningen hans viser at denne typen representasjon av genene med avbildning, er effektiv for boolske læringsfunksjoner [15]. Forskningen viser også at metoden har mye overflødighet, som er bra for feiltoleranse, og nøytralitet i egnetheten for kretsene. Han har utviklet denne metoden videre, til en modulære CGP [16], som viser seg å være 20% raskere en den tradisjonelle CGP.

3.6 En alternativ måte

K. Zhang, R. DeMara og C. Sharma ved universitetet i Central Florida, har sett på en metode som kalles “Consensus Based Evaluation” (CBE) [32]. Denne metoden går ut på at man starter med et sett fungerende individer, som har samme egenskap når det kommer til innverdier og utverdier, men er bygd opp på forskjellige måter. Individene deles inn i to grupper, “L” og “R”. Man laster så

opp et individ fra hver av de to gruppene i tandem på en Field Programmable Gate Array (FPGA). Utsignalene fra hvert av individene, og hvis det skulle oppstå en uenighet, så her det dukket opp en feil i et av individene.

Hvert individ kan ha fire tilstander, *Pristine*, *Suspect*, *Under repair* og *Refurbished*. Individene forblir i tilstand *Prestine* til det oppstår en uenighet. De blir begge individene endret til tilstand *Suspect*. Hvis verdiene til individene i tilstand *Suspect* overgår en gitt verdi vil De gå til tilstand *Under repair*. Da vil FPGA-en omkonfigureres med nye individer.

Kapittel 4

Simulatoren

Dette kapitlet er delt inn i tre deler. Den første, delkapittel 4.1, forklarer hvordan systemet fungerer praktisk sett, mens den andre delen, delkapittel 4.2, forklarer hvordan systemet fungerer teknisk sett. Disse to delkapitlene beskriver systemet som det var når dette prosjektet ble påbegynt. Delkapittel 4.3 forklarer hvilke endringer som er utført på systemet i løpet av dette prosjektet.

4.1 Systemet

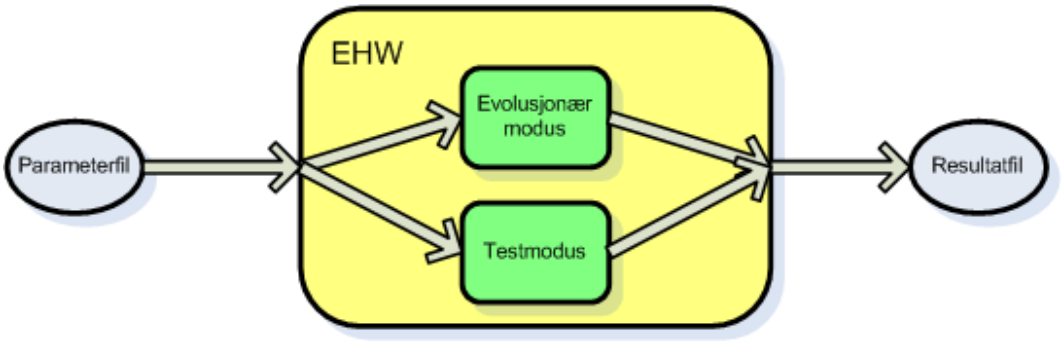
Simulatoren er et uavhengig program som heter *ehw*. I tillegg til dette programmet er det utviklet et sett med hjelpeskript og -program, som skal forenkle massekjøring av simulatoren, i forbindelse med eksperimenter. En noe forenklet framstilling av systemet vises i figur 4.1.

Figur 4.1: En parameterfil sendes inn i programmet *ehw*, som skriver resultatet til en fil. Denne filen blir lest inn i programmet *extract*, som ekstraherer relevant statistikkdata og legger det i en ny fil. Denne filen blir lest inn i et R-skript, som produserer en fil med en grafisk framstilling av dataene.

4.1.1 ehw

ehw er et program som evoluerer og simulerer kretser. Man starter med et tilfeldig sett av kretser, og gjennom evolusjon skapes det nye kretser, til men finner en

krets som tilfredsstiller et forhåndsdefinert krav. Gjennom genetiske operasjoner får man nye individer, med nye egenskaper.



Figur 4.2: Simulatoren kan kjøre i to modi, evolusjonær modus og testmodus. Det sendes en parameterfil inn i programmet *ehw*. Innholdet i parameterfilen bestemmer om *ehw* skal kjøre i evolusjonær modus eller testmodus.

Simulatoren har muligheten til å kjøre i to modus, som vist i figur 4.2:

- Evolusjonær modus
- Test modus

Hvilken modus som blir kjørt spesifiseres i parameterfilen, som sendes inn ved programstart. Foruten parameterfilen, sendes det også inn en støy- og feil-parameter. Dette programmet kan kjøres uten noe av det ytre systemet, som beskreves i seksjon 4.1.2. Da vil kjøring av programmet gjøres fra kommandolinjen på følgende måte:

- *./ehw* parameterfil støyvariabel feilvariabel

Evolusjonær modus

I den evolusjonære modusen kan programmet enten motta et spesifikt individ å starte evolusjonen med, eller så kan programmet generere et gitt antall tilfeldige startindivider. Basert på denne dataen vil simulatoren prøve å evoluere en krets som tilfredsstiller kravene som stilles i parameterfilen i form av en sannhetstabell. Programmet vil med jevne mellomrom rapportere framgangen i evolusjonen, som sendes til standard ut, som er skjerm hvis annet ikke er definert. Til eksperimentene som er utført i dette prosjektet, er det ønskelig å legge all data som kommer fra programmet til en fil, da flyttes standard ut til fil. Dette gjøres på følgende måte:

- *./ehw* parameterfil støyvariabel feilvariabel >> filnavn

I denne filen legges også resultatet av den fullførte evolusjonen. Navnet for denne filen vil, for dette prosjektet, være "out_*". Dette er fordi skriptet *extractall.sh*, som forklares i seksjon 4.2.4, er avhengig av dette filnavnet for å eksekvere riktig.

Testmodus

I testmodusen mottar programmet et individ, og utfører tester på dette. Hvilke tester og hvor mange ganger disse testene skal utføres, spesifiseres i parameterfilen. Resultatet av testingen vil sendes til standard ut.

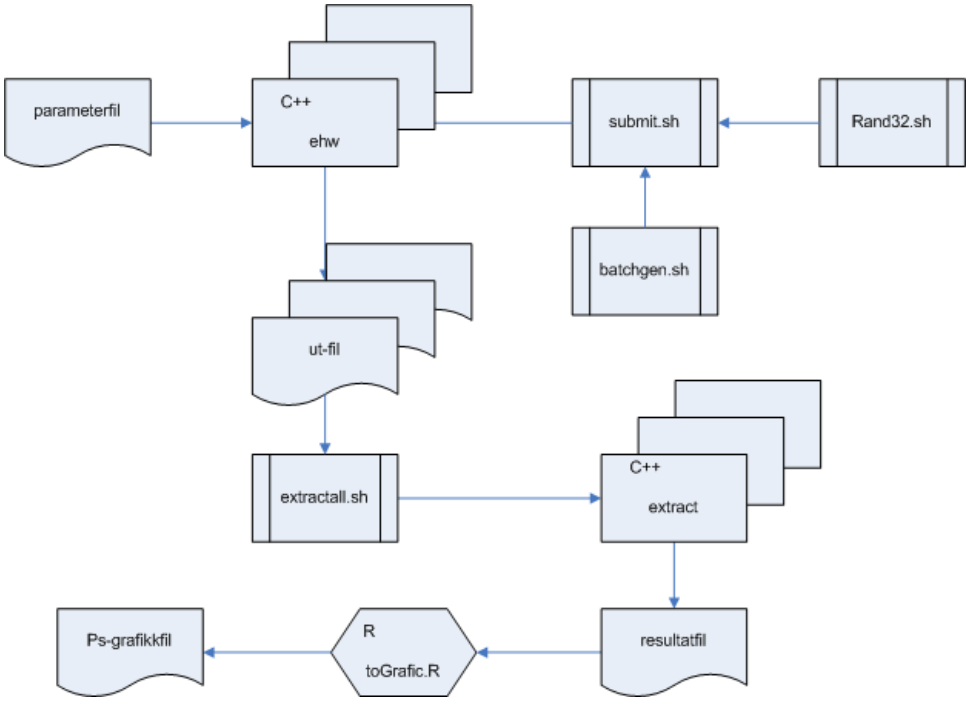
4.1.2 Det ytre systemet

Det ytre systemet består av et sett med batchfiler og et R-skript(se seksjon 4.2.6). Det lager et grunnlag for å kjøre simulatoren flere ganger, og med forskjellige innparametre, uten å gjøre dette manuelt. Det gir også resultatet ut grafisk, for enklere tolkning av dataen.

Man starter systemet med filen *batchgen.sh*, med et sett innparametre som er beskrevet i tabell 4.3. Dette skriptet oppretter et sett med mapper, basert på hvilke verdier av støy og feil som sendes inn som parametere. For hver mappe kopierer den tre filer som også er spesifisert som parametere inn i mappen. Den vil så kalle opp skriptet *submit.sh* for hver mappe.

submit.sh er det skriptet som lager PBS-filen som brukes for å kjøre *ehw* på clustis2, og med denne filen, legger den jobben inn. PBS-filen blir bygd opp så at den først kjører *ehw* i evolusjonær modus. Så blir programmet *extract* kjørt. Dette programmet henter ut individet som ble lagt i resultatfilen etter *ehw*-kjøringen, og legger det i parameterfilen som skal brukes under kjøring i testmodus. Deretter kjøres *ehw* en gang til, denne gangen i testmodus, der det omtalte individet blir testet.

For hver jobb som kjøres, vil det lages en resultatfil. For å tolke informasjonen fra resultatfilene, er det ønskelig å få en grafisk framstilling av dataen. Dette gjøres med et program som er skrevet i R. R er et utviklingsspråk og -miljø for statistiske beregninger og grafisk framvising [9]. For at R-programmet skal kunne benytte dataen fra resultatfilen, må formatet på den gjøres om. Når *ehw* er ferdig med å kjøre, må man starte skriptet *extractall.sh* manuelt. Med dette skriptet vil programmet *extract* bli kjørt på alle resultatfilene. *extract* henter ut dataen den trenger fra resultatfilene, og legger de i en ny fellesfil, med det formatet som kreves av R-programmet. Denne fellesfilen blir lest inn i R-programmet, som vil lage en grafisk framstilling av dataen. Den grafiske framstillingen lagres så i en ny fil.



Figur 4.3: En fullstendig oversikt over simulatorsystemet. *batchgen.sh* startes manuelt med et sett parametere, som automatisk kaller opp *submit.sh* før den avslutter. *submit.sh* bruker *rand*32.*sh* for å generere et frø som legges i parameterfilen. *submit.sh* vil så sette i gang et sett med jobber som ble spesifisert i parameterlisten, som ble sendt inn i *batchgen.sh*. For hver jobb vil *ehw* kjøre og resultatene vil legges i hver sin resultatfil. Etter dette må *extractall.sh* startes manuelt. Den vil kalle opp programmet *extract* for hver jobb som er utført. *extract* henter ut data fra resultatfilene, og legger det i en ny fil med et spesielt format, som skal leses inn i et R-skript for å framstille resultatet av kjøringene grafisk. De grafiske framstillingene blir lagt i en PostScript-fil.

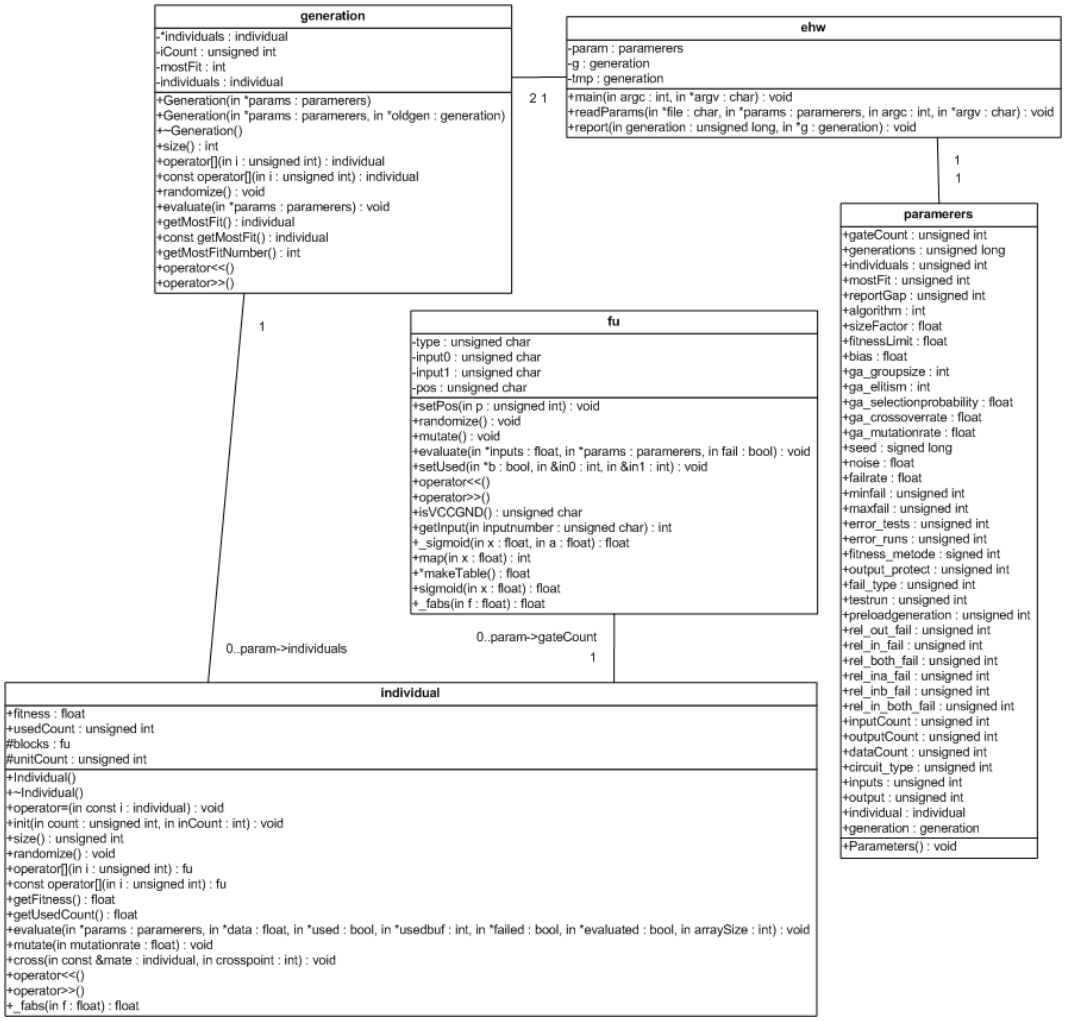
4.2 Implementasjonen

Simulatoren er utviklet i C++, og opererer ved å simulere elektroniske kretser som er bygd opp av blokker som er beskrevet i detalj i [2]. Rammeverket til simulatoren består av et sett batch-skript, et C++program og et R-skript.

4.2.1 ehw

Programmet *ehw* starter med å opprette et parameterobjekt der alle verdier fra parameterfilen blir lagret. Innholdet i parameterfilen finnes i tabell 4.2, og det er disse parametrene som gir grunnlag for evolusjonen. *ehw* vil så lage et generasjonsobjekt, som igjen vil opprette ett gitt antall individobjekter. Hvert individobjekt består videre av et sett fuobjekter, der hvert fuobjekt representerer en blokk i

individet. Klassediagrammet for programmet vises i figur 4.4. Individene i generasjonen blir enten tilfeldig generert, eller duplisert fra et spesifisert individ i parameterfilen. Individene blir evaluert og, basert på denne evalueringen, motta en verdi som representerer hvor egnet kretsene er. Utvelgelsen av individer fra en generasjon til den neste, baserer, i hovedsak, sine valg på denne egnetheten. For hvert individpar som velges ut, utføres krysning og mutasjon.



Figur 4.4: Klassediagram av simulatoren. *ehw*

Evaluering av egnethet og pålitelighet

For å finne ut hvor egnet en krets er, må man ha et kriterium for sammenligning. Sannhetstabellen for kretsen du ønsker å oppnå, må derfor være tilgjengelig for simulatoren ved evalueringen, og blir derfor definert i parameterfilen som sendes inn i systemet. I dette prosjektet er det primært brukt en to bits multiplikator.

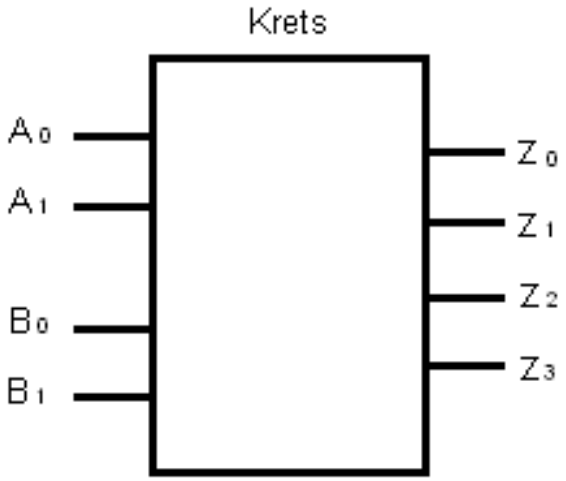
A	B	Z
00	00	0000
00	01	0000
00	10	0000
00	11	0000
01	00	0000
01	01	0001
01	10	0010
01	11	0011
10	00	0000
10	01	0010
10	10	0100
10	11	0110
11	00	0000
11	01	0011
11	10	0110
11	11	1001

Tabell 4.1: Sannhetstabell for en to bits multiplikator. A og B er innverdier som skal ganges sammen, og Y er det forventede resultatet ut ifra kretsen.

Sannhetstabellen er vist i tabell 4.1, med tilhørende kretsen i figur 4.5. Evalueringen tar ingen hensyn til hvordan kretsen er bygd opp, kun resultatet av den.

Det finnes to metoder å måle hvor korrekt en krets er [31], R_ehw og R_trad. R_ehw kan betraktes som hvor egnet en krets er, mens R_trad betraktes som hvor pålitelig en krets er. Når disse to verdiene skal regnes ut, vil simulatoren sjekke kretsen for alle mulige innverdier, og sammenligne resultatet fra kretser med hva den forventer, ut ifra sannhetstabellen. For R_ehw vil hvert signal som ikke stemmer med sannhetstabellen, gi en negativ økning på egnetheten. Egnetheten blir registrert i en variabel som minker i verdi for hvert feilsignal. Egnetheten starter på null og vil for hver feil bli redusert med 1. Egnetheten vil altså få en negativ verdi ved uoverensstemmelser, og den vil forbli null hvis kretsen stemmer med kriteriene. For R_trad er det kun interessant å vite om kretsen fungerer perfekt eller ikke. Ved kun en bit feil vil verdien av påliteligheten bli 0, er den korrekt blir verdien 1.

Disse to målingene vil bli gjort over et sett med tester, der det blir tilført feil. For hver test vil det kun eksistere en feil i kretsen. Det er gjennomsnittet av disse testene som utgjør verdiene til egnetheten og påliteligheten.



Figur 4.5: Signalinnganger og -utganger til en to bits multiplikator.

Filene som sendes inn i systemet

Det er tre filer som er med inn i systemet når det startes opp.

- ehw.cfg (parameterfilen)
- ehw_ t.cfg (testfilen)
- ehwjob.job (PBS-filen)

Parameterfilen og testfilen er filer som *ehw*-programmet leser inn ved starten av kjøringen. Eksempel på hvordan denne filen kan se ut finnes i vedlegg 3. De har et sett med variabler, som bestemmer hvordan programmet skal oppføre seg. En oversikt over filenes variabler finnes i tabell 4.2. Det som bestemmer om filen er en parameterfil eller en testfil, er variabelen testrun. Hvis denne er 0 er det en parameterfil, er den 1 så er det en testfil.

I PBS-filen ligger det, i utgangspunkt, fire linjer med verdier som gjelder for kjøringen på clustis2. Det er verdier som hvor mange noder som skal brukes, hvilken kø som skal brukes, og hvor mye tid programmet trenger for å fullføre. Resten av verdiene denne filen trenger for å fullføre sin funksjon, blir lagt inn når skriptet *submit.sh* kjøres. Dette blir forklart nærmere i seksjon 4.2.3

4.2.2 batchgen.sh

batchgen.sh er et batchskript som oppretter en serie med mapper, og kopierer inn parameterfilen, testfilen og jobbfilen i hver mappe. Antall mapper den lager

Navn	Datatype	Verdi	Beskrivelse
testrun	unsigned int	0-1	Hvilken modus *ehw* skal kjøre
debug	unsigned int	0-1	Programdebuging
gates	unsigned int	1->	Maks antall blokker pr individ
generations	unsigned long	1->	Maks antall generasjoner
individuals	unsigned int	1->	Antall individer i hver generasjon
report_ gap	unsigned int	1->	Hvor ofte fremgangen skrives ut
size_ factor	float	0-1	Økt egnethet for små kretser
fitness_ limit	float	1->	Ønsket egnethet
algorithm	int	1-5	Velger type seleksjonsalgoritme
ga_ groupsize	int	3->	Gruppest. i seleksjonsalg.
ga_ elitism	int	0-1	Beste individ fra en generasjon kopieres direkte til neste
ga_ selectionprobavility	float	0-1	Sannsynlighet for å velge beste individ fra gruppen
ga_ crossoverrate	float	0-1	Sannsynlighet for krysning
ga_ mutationrate	float	0-1	Sannsynlighet for mutasjon
bias	float	0->	Favorisering av beste individ
seed	unsigned long	0->	Frøet individer blir generert ut ifra
individual	individual	0-1	Et individ man starter evolusjonen fra, eller tester
noise	float	0-1	Simulering av støy
failrate	float	0->	Simulering av feil
minfail	unsigned int	0->	Minimum ant. blokker som feiler
maxfail	unsigned int	0->	Maksimum ant. blokker som feiler
error_ tests	unsigned int	0->	Hvilke tester skal utføres
error_ runs	unsigned int	0->	Antall ganger et individ testes
rel_ out_ fail	unsigned int	0-1	Sannsynlighet for at utg. feiler
rel_ in_ fail	unsigned int	0-1	Sannsynlighet for at inng. feiler
rel_ both_ fail	unsigned int	0-1	Sannsynlighet for at begge feiler
rel_ ina_ fail	unsigned int	0-1	Sannsynlighet for at hvis inng. feiler, så feiler inng 1
rel_ inb_ fail	unsigned int	0-1	Sannsynlighet for at hvis inng. feiler, så feiler inng 2
rel_ in_ both_ fail	unsigned int	0-1	Sannsynlighet for at hvis inng. feiler, så feiler begge
fail_ type	unsigned int	0-1	Hvilken type feil som skal brukes
output_ protect	unsigned int	0-1	Beskyttelse av utblokkene, medfører at de ikke feiler
fitness_ method	unsigned int	1-2	Hvordan egnethet skal regnes ut
inputs	unsigned int	1->	Antall innbits i sannhetstab.
outputs	unsigned int	1->	Antall utbits i sannhetstab.
data_ count	unsigned int	1->	Antall linjer i sannhetstabellen
data	float	0-1	Sannhetstabellen
curcuit_ type	unsigned int	0-1	Hvordan type krets som evolueres

Tabell 4.2: Verder som kan leses inn gjennom parameterfilen.

er avhengig av antall iterasjoner av støy og feil som skal kjøres, og er verdier som sendes inn som parametere til *batchgen.sh*. Skriptet kjøres på følgende måte, fra katalogen der *ehw* og parameterfilene ligger:

- *batchgen.sh* støyiters feiliters støystart støyslutt feilstart feilslutt jobber evocfg testcfg jobcfg

En nærmere beskrivelse av innparametrene finnes i tabell 4.3.

Navnene på mappene som opprettes er basert på parametrene støyiters, feiliters, støystart, støyslutt, feilstart og feilslutt. *batchgen.sh* regner ut, basert på disse parametrene, hva endringen i støy- og feil skal være. Ved å ta

$$(støyslutt - støystart)/støyite$$

for å finne delta-støy, og

$$(feilslutt - feilstart)/feiliters$$

for delta-feil. Disse regnes ut med en presisjon på tre desimaler. Mappene får navnet n<støyverdi>_f<feilverdi>. Den begynner med startvariablene, og legger til deltavariablene for hver iterasjon. Sluttvariablene faller derfor utenfor, og vil ikke bli kjørt.

For å ta et eksempel på dette, settes støy- og feiliters til 2, støystart og feilstart til 0.1, støyslutt til 0.2 og feilslutt til 0.3. Deltastøy vil i dette tilfelle bli 0.05 og deltafeil vil bli 0.1. Disse verdiene gir 4 mapper med navnene:

- n0.100_ f0.100
- n0.100_ f0.200
- n0.150_ f0.100
- n0.150_ f0.200

Etter at parameterfilene er kopiert inn i hver mappe, vil den for hver mappe kjøre bacthskriptet *submit.sh*.

4.2.3 submit.sh

submit.sh er et batchskript som har til oppgave å produsere PBS-filen som brukes for å kjøre jobber på clustis2, før den legger jobben i køen med qsub. Filen kjøres på følgende måte:

Variabel	Verdier	Beskrivelse
støyiters	1..->	Antall iterasjoner med støy
feiliters	1..->	Antall iterasjoner med feil
støystart	0..->	Startverdi for støy
støyslutt	0..->	Sluttverdi for støy
feilstart	0..->	Startverdi for feil
feilslutt	0..->	Sluttverdi for feil
jobber	1..->	Antall jobber som skal kjøres med hver verdi av støy og feil
evocfg	filnavn	Parameterfilen for evolusjonær kjøring
testcfg	filnavn	Parameterfilen for testkjøring
jobcfg	filnavn	Filen som inneholder startkommando av hovedprogrammet

Tabell 4.3: Innparametre som sendes inn når batshfilen *batchgen.sh* startes

- *submit.sh* jobber evocfg testcfg jobcfg støy feil

De fire første parametrene er de samme som de siste fire i *batchgen.sh*, og sendes inn der kun for å videresendes til *submit.sh*. Disse fire parametrene er forklart i tabell 4.3. Støy og feil er de verdiene som *batchgen.sh* regnet ut, de samme verdiene som ble brukt til å navngi mappene.

submit.sh vil for hver jobb kopiere de tre filene, og gi dem individuelle navn. For hver av de individuelle parameterfilene, vil den generere et frø og legge det inn nederst i denne filen. For å generere frø brukes skriptet *rand*32*.sh*. Etter det vil den fullføre PBS-filene, ved å leggen inn tre kommandoene for å kjøre på clustis2.

- Kjøring av *ehw* med parameterfilen (evolusjonsmodus)
- Kjøring av programmet *extract*, for å ekstrahere beste individ fra evolusjonskjøringen
- Kjøring av *ehw* med testfilen (testmodus)

Den vil så kjøre qsub med PBS-filen som parameter, for hver jobb.

4.2.4 extractall.sh

extractall.sh er et batshskript som kaller opp programmet *extract*, for et sett med filer i den katalogen, og eventuelt alle subkataloger.

Skriptet kjøres på følgende to måter:

- *extractall.sh* -R

- *extractall.sh*

Hvis striptet kjøres uten -R argumentet, vil skriptet kalle opp programmet *extract* for alle filer i denne katalogen, som har et navn som starter på "out_". Hvis argumentet -R er med vil den vil den starte en rekursiv rutine som kaller seg selv for hver subkatalog, og starter *extract* for hver "out_*"-fil for hver subkatalog. En ulempe med dette skriptet, er at filen som dette skriptet skriver til, må inneholde en overskrift, som man får ved å kjøre programmet *extract* med modus 2. Modiene til *extract* blir forklart i seksjon 4.2.5.

4.2.5 extract

extract er et program som benyttes for å hente ut informasjonen fra resultatfilene etter *ehw*-kjøringen. *extract* tar imot to argumenter, filnavn på filen dataen skal hente fra og en integer som forteller i hvilken modus programmet skal kjøre.

Programmet kjøres på følgende måte:

- *extract* filnavn modus

Programmet kan kjøre i tre forskjellige modi. Modus 0 skriver ut alle variablene etter hverandre på en linje, adskilt med en strek (eng:pipe). Modus 1 skriver kun ut innholdet i individvariablen, og modus 2 skriver ut en tekststreng som inneholder alle variablene i det samme formatet som ble skrevet ut i modus 0. Hvert variabelnavn er delt av med en strek.

4.2.6 R-skriptet

R er et program for statistisk beregning og framstilling, som i dette prosjektet blir brukt til å representere resultatene av kjøringene grafisk. Skriptet leser inn data fra filen som *extract* produserer, og lagrer disse dataene grafisk i en PostSkript-fil.

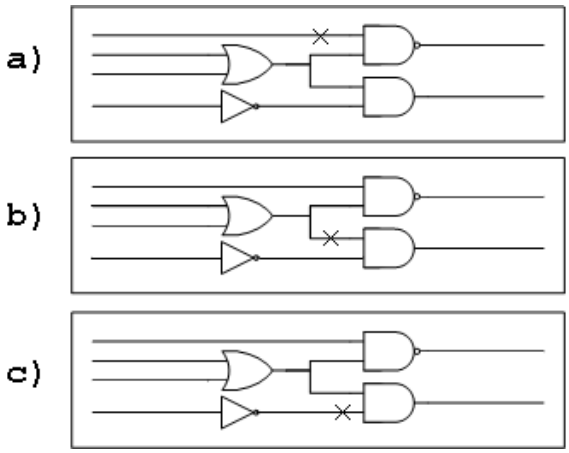
4.3 Utvidelser

Hittil er systemet beskrevet slik det var før dette prosjektet ble påbegynt. Alt som er endret i systemet beskrives i dette delkapitlet.

4.3.1 Simulatoren

Utvidelsen som er lagt til *ehw* i dette prosjektet er singel inverteringsfeilmodellen. Denne feilmodellen er lagt opp til at brukeren selv kan velge om *ehw* skal utføre denne typen feil, og hvor mange feil hver krets skal testes med.

Det er lagt til to variabler i parameterfilen som bestemmer hvordan denne feilmodellen skal utføres. En variabel, A, som bestemmer om feilmodellen skal brukes eller ikke, og den andre, B, bestemmer hvor mange feil hver krets skal testes med. Det vil si at hvis feilmodellen skal brukes og den skal teste med tre feil, vil det genereres tre tilfeldige feil, men kretsen vil testes med kun en feil av gangen. A settes da til 1 og B settes til 3. Altså kretsen vil i dette tilfellet testes tre ganger, som vist i figur 4.6. Ønsker brukeren og teste med feil på alle blokkene i kretsen, settes B til null. Er A satt til 0, vil ikke feilmodellen benyttes, og B blir da ubetydelig.



Figur 4.6: Singel inverteringsfeil. Kretsen i a), b) og c) er samme krets som blir testet med tre singel inverteringsfeil. Først blir a) testet uavhengig av b) og c), før b) blir teste og så c). Egnethet og pålitelighet regnes ut som et gjennomsnitt av disse testene.

Skulle det oppstå en situasjon der en bruker ønsker å utføre flere feil en det er blokker i kretsen, vil simulatoren teste feil på alle blokkene. Det er også tatt hensyn til beskyttelse av blokkene som har en utverdi (eng:outputprotect).

Alle endringer i *ehw* ble gjort i filen individual.cpp, som ligger vedlagt i vedlegg 3.

4.3.2 Det ytre systemet

batchgen.sh og submit.sh

batchgen.sh og *submit.sh* ble vesentlig forenklet for dette prosjektet, og ble da slått sammen til *batchsubmit.sh*. Koden til *batchsubmit.sh* finnes i vedlegg 3. Dette skriptet vil først opprette en mappe for kjøringen som skal utføres, før den kopierer parameterfilen, testfilen og jobbfilen inn i denne mappen. For hver jobb som skal utføres, blir de tre filene kopiert og gitt individuelle navn for hver jobb. For hver jobb vil det også blir generert et frø med *rand*32.*sh*, som legges inn i de individuelle parameterfilene. Den vil så generere ferdig PBS-filene for alle jobbene, før den legger til PBS-filene i kjørekøen på clustis2.

Filen kjøres på følgende måte:

- *batchsubmit.sh* jobber evocfg testcfg jobcfg mappenavn

De fire første parametrene er de samme fire som brukes i både *batchgen.sh* og *submit.sh*. Parameteren mappenavn brukes når mappen skal opprettes.

extractall.sh

Dette skriptet har nå blitt utvidet så man kan produsere hele filen fra en blank start automatisk. Resultatfilen trenger ikke lengre å inneholde overskriften før dette skriptet kjøres. Den har også blitt utvidet til å gi brukeren muligheten til å bestemme navnet på resultatfilen selv. Denne verdien tas med som en parameter. Koden til dette skrptet finnes i vedlegg 3.

Dette skriptet kjøres nå på følgende måte:

- *extractall.sh* filnavn -R
- *extractall.sh* filnavn

extract

Dette programmet er blitt utvidet med fire modi, samt det kan kjøres med andre parametere. Koden for dette programmet er lagt ved i vedlegg 3. Programmet har nå følgende modi:

- Modus 0 skriver ut alle variablene etter hverandre på en linje, adskilt med en strek.

- Modus 1 skriver kun ut innholdet i individvariabelen
- Modus 2 skriver ut en tekststreng, som inneholder alle variablene i det samme formatet som ble skrevet ut i modus 0. Også hver adskilt med en strek.
- Modus 3 skriver ut en tekststreng som inneholder variabel navnene "fitness" og "generation".
- Modus 4 skriver ut alle egnetheter, og hvilken generasjon den tilhører, som er rapportert i løpet av evolusjonen.
- Modus 5 skriver ut hvert tyvende målepunkt av egnethet, og hvilken generasjon den tilhører, som er rapportert i løpet av evolusjonen. Det utgjør hver titusende generasjon
- Modus 6 skriver ut de første trettiførste egnetheter, og hvilken generasjon den tilhører, som er rapportert i løpet av evolusjonen. Det utgjør til og med generasjon 15 000.

Og programmet kan kjøre på følgende måter:

- *extract*
  - Når man kjører på denne måten så vil modusen settes automatisk til 2.
- *extract* modus
  - Når man kjører på denne måten, må modus være 2 eller 4, ettersom disse modiene ikke skriver ut noe data som skal leses inn fra en fil, kun genererte tekststrenger. Kjøres denne metoden med andre modi, vil det ikke gi ut relevant data.
- *extract* filnavn modus
  - Denne måten kan kjøres med alle modi.

extract20.sh og extract31.sh

Dette er to nye skript, som benytter seg av de nye modiene i *extract*. Skriptene fungerer på samme måte som *extractall.sh*, men henter ut et annet datasett. *extract*20.*sh* henter ut hvert tyvende rapporterte egnethetsverdiene fra evolusjonen, og *extract.sh* henter ut de trettiførste rapporterte egnethetsverdiene.

Disse skriptene, som er vedlagt i vedlegg 3, kjøres på følgende måte:

- *extract* ∗ ∗.*sh* filnavn -R
- *extract* ∗ ∗.*sh* filnavn

Kapittel 5

Eksperimenter

I dette kapitlet beskrives simuleringene som er utført. I delkapittel 5.1 beskrives grunnlaget for hvordan eksperimentene er utført. Delkapitlene 5.2, 5.3, 5.4 og 5.5 beskriver de fire settene med eksperimenter som ble utført.

5.1 Introduksjon

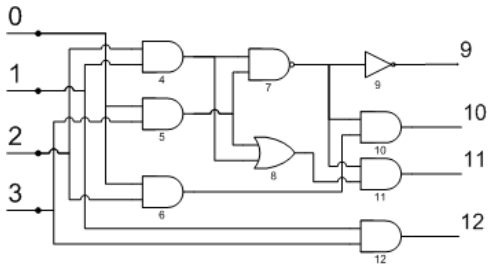
Eksperimentene hadde til mål å prøve å utvikle en 2-bits multiplikator, der det alltid eksisterer en inverteringsfeil. Det påføres inverteringsfeil på alle blokker i den kretsen som evalueres, men kun en blokk av gangen. Det vil si at et individ med 15 blokker, evalueres 15 ganger med en feil for hver evaluering. Feilen på blokken skjer ved at signalet ut i fra blokken inverteres.

Alle eksperimentene ble kjørt på en Beowulf-type beregningsklynge, med bruk av openPBS. Hver eksperiment ble kjørt ti ganger, og det er gjennomsnittet av disse kjøringene som blir sett på. Hver kjøring ble i tillegg testet to ganger, en gang uten feil og en gang med inverteringsfeil. Alle eksperimentene ble kjørt i 500 000 generasjoner, med en ønsket egnethet på 0. Det ble forventet av alle eksperimenter kom til å kjøre i 500 000 generasjoner, og aldri oppnå ønsket egnethet. Grunnen til det høye antallet generasjoner, var ønsket om at egnetheten skulle stabilisere seg mest mulig.

Utvelgelsesalgoritmen som ble brukt var Tournament Selection, med gruppestørrelse på 3 og en seleksjonssannsynlighet på 70%. Det ble også brukt favorisering av beste individ. Favorisering betyr av det beste individet fra en generasjon, blir direkte overført til neste uten at det blir rørt. Antall individer ble i utgangspunktet satt til 20, blokker til 80, mutasjonssannsynlighet til 5% og krysningsraten til 20%. Disse fire parametrene ble satt som utgangspunkt for fire serier med eksperimenter der en og en parameter ble endret.

I forhold til disse parametrene, blir det sett på egnetheten og påliteligheten til resultatindividene som ble utviklet med evolusjon, samt egnethet og pålitelighet til disse individene ved testing uten feil. Egnethet er et mål på hvor mange bits unna en løsning kretsen er, mens pålitelighet ser på i hvor mange tilfeller kretsen vil gi rett resultat.

For å ha et sammenligningsgrunnlag av hva men kan forvente av kretser som er evoluert med inverteringsfeil, er det testet to kretser, en enkel multiplikator og en TMR-krets. Resultatene fra disse kjøringene finnes i vedlegg 3. Den enkle multiplikatoren vises i figur 5.1. TMR-kretsen i figur 5.2 består av tre enkle multiplikato og en velger. Den enkle kretsen gir et resultat som kan betraktes som det dårligste resultatet en multiplikator kan oppnå, ettersom det er kjent at det ikke eksisterer noe redundans i denne kretsen, og den har ingen form for feiltoleranse. TMR-kretsen er den tradisjonelle metoden for å oppnå redundans, så ett mål er å oppnå kretser som er bedre enn denne.

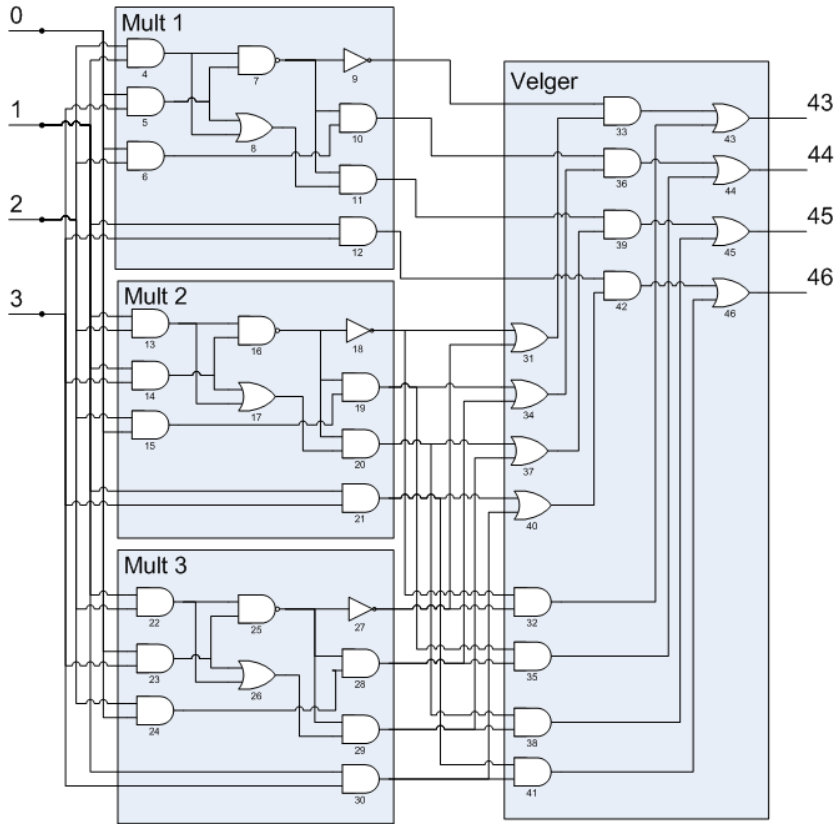


Figur 5.1: Kretstegning av en enkel 2 bits multiplikator. Signal 0 til 3 er innverdier og signal 9 til 12 er utverdier

Det er også sett på egnethet over tid. Hvert resultat blir her sett på i forhold til sitt eget sett med eksperimenter, men det vil også bli sett på gjennomsnittet av alle eksperimenter som er utført i løpet av dette prosjektet.

5.2 Eksperimentsett 1 : Individer

I dette settet med eksperimenter ble antall individer i populasjonen variert. Til å begynne med ble verdien av antall individer satt til 20, etter anbefaling fra veileder, og verdiene rundt dette ble så testet. Verdiene av antall individer ble variert fra 10 til 30, med intervall på 5. Det ble altså testet med 10, 15, 20, 25 og 30 individer.



Figur 5.2: Kretstegning av en TMR-krets. Den består av tre enkle 2 bits multiplikato som er identiske, og en velger.

5.3 Eksperimentsett 2 : Blokker

I dette settet med eksperimenter ble antall blokker i individet variert. Til å begynne med ble verdien av antall blokker satt til 80, og verdiene rundt dette ble så testet. Verdiene til antall blokker skulle i utgangspunktet variere fra 20 til 100, men ble utvidet til 120. Intervallet på antall blokker var 20. Det ble atså testet med 20, 40, 60, 80, 100 og 120 blokker.

5.4 Eksperimentsett 3 : Krysning

I dette settet med eksperimenter ble krysningsraten variert. Til å begynne med ble verdien satt til 30%, og verdiene rundt dette ble så testet. Verdiene til krysningsraten varierte fra 20% til 50%, med et intervallet på antall blokker var 20%. Det ble altså testet med 20%, 30%, 40% og 50% krysningsrate.

5.5 Eksperimentsett 4 : Mutasjon

I dette settet med eksperimenter ble mutasjonsraten variert. Til å begynne med ble verdien satt til 5%, og verdiene rundt dette ble så testet. Verdiene til mutasjonsraten varierte fra 1% til 9%, med et intervallet på antall blokker var 2%. Det ble altså testet med 1%, 3%, 5%, 7% og 9% mutasjonsrate.

Kapittel 6

Resultater

I dette kapitlet blir det presenteres grafer med resultatene fra simuleringene. Delkapitlene 6.1, 6.2, 6.3 og 6.4 beskriver de fire settene med eksperimenter som ble utført, mens 6.5 ser litt nærmere på alle enkeltkjøringene til de forskjellige eksperimentene. I delkapitlene 6.6 blir det sett på hvordan egnetheten har endret seg i løpet av evolusjonene. Hvert delkapittel vil bli avsluttet med en drøfting av resultatene.

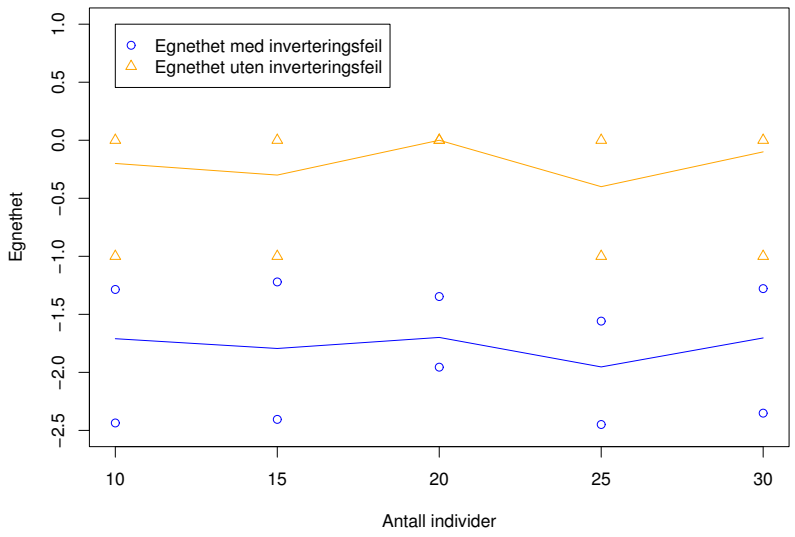
6.1 Resultater fra eksperimentsett 1 : Individer

Det første settet med eksperimenter som ble utført, var simuleringer med endring i antall individer i populasjonen. Det ble kjørt eksperimenter med 10 til 30 individer, med intervall på 5 individer. Diagrammene i figur 6.1 og 6.2 viser resultatet av dette settet med eksperimenter. Diagrammene består av et sett grafer med et tilhørende sett punkter. Grafene er gjennomsnittet av alle kjøringene for hvert eksperiment, der punktene over og under er maksimum og minimum av disse kjøringene. Hvis det eksisterer kun ett punkt vertikalt for hverandre og den ligger på graflinjen, er maksimumspunkt og minimumspunkt sammenfallende.

I figur 6.1 viser X-aksen antall individer som ble testet i settet av eksperimenter og, Y-aksen er egnethetsverdien av disse eksperimentene. Grafen som viser egnethet med inverteringsfeil ligger som en jevn linje, mellom -2.0 og -1.5. Det som er bemerkelsesverdig her, er spredningen mellom maks- og min-verdi, er mindre for eksperimentet med 20 individer, enn for de andre av eksperimentene. Til sammenligning ble egnetheten ved testing av den enkle multiplikatoren med inverteringsfeil -18.33, og egnetheten av TMR-kretsen -3.81. Alle kjøringene i

hele settet med eksperimenter, presterte bedre en TMR-kretsen. Disse verdiene er ikke plottet inn i grafen på grunn av den ekstreme forskjellen i verdier.

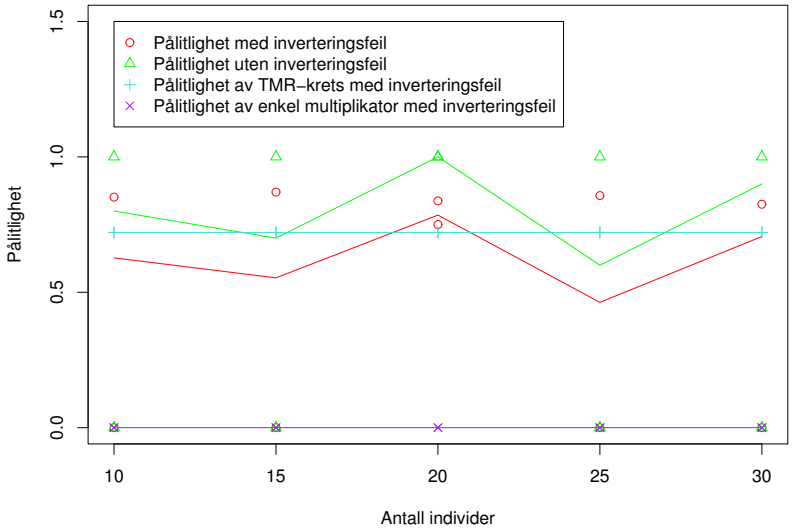
Grafen som viser egnethet uten inverteringsfeil, ligger også som en jevn linje mellom -0.5 og 0. Verdien av egnethet uten feil vil være et heltall mellom 0 og -64, der 0 er en korrekt krets og -64 er en krets som ikke har noen bits rett. Maks-punktet for alle eksperimentene er 0, noe som betyr at kretsen fungerer 100% korrekt når det testes uten feil. Min-punktet for alle eksperimentene, bortsett fra eksperimentet med 20 individer, er -1.0. For 20 individer er minpunktet 0, noe som medfører at alle kjøringene i eksperimentet med 20 individer evoluerte fram en fungerende multiplikator. For de andre eksperimentene, eksisterte det flere kretser med egnethet 0 enn -1. Gjennomsnittlig, av alle kjøringene til alle eksperimentene, var det 80% av kjøringene som resulterte i en fungerende multiplikator når den ble testet uten feil.



Figur 6.1: Egnethet for antall individer. Diagrammet viser to grafer. Den ene viser gjennomsnittet av alle kjøringer, der det eksisterer en inverteringsfeil, og den andre, der det ikke eksisterer en inverteringsfeil.

I figur 6.2 viser X-aksen antall individer som ble testet i settet av eksperimenter og, Y-aksen er pålitelighetsverdien av disse eksperimentene. Grafene som viser pålitelighet med og uten inverteringsfeil, speiler en del av observasjonene fra grafene i figur 6.1. Begge disse grafene ligger mellom 0.5 og 1, og har en lignende stigningstall både med hverandre og grafene fra figur 6.1. Maks- og minpunkter ligger på 1 og 0 for alle bortsett fra eksperimentet med 20 individer. Her ligger påliteligheten uten inverteringsfeil på 1, som igjen viser at alle kjøringene i dette

eksperimentet, produserte en krets som fungerte perfekt uten feil. Med feil ble påliteligheten for alle kjøringene utført i eksperimentet med 20 individer, bedre den testen gjort med en TMR-krets. Eksperimentene med 10 og 30 individer gjorde det bedre enn TMR-kretsen i de fleste tilfellene.



Figur 6.2: Pålitelighet for antall individer. Diagrammet viser fire grafer. Den ene viser gjennomsnittet av alle kjøringer, der det eksisterer en inverteringsfeil, og den andre, der det ikke eksisterer en inverteringsfeil. Den tredje viser påliteligheten av en TMR-krets som er testet med inverteringsfeil. Den siste viser påliteligheten til en enkel multiplikator, som også er testet med inverteringsfeil.

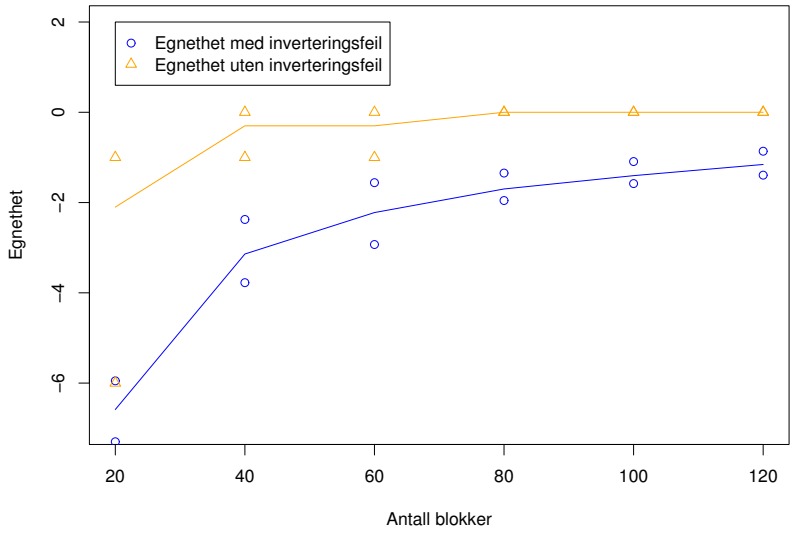
Ved første blikk på dette settet med eksperimenter, kan man tro at 20 individer er ett optimalt antall individer å bruke i populasjonen, men i de fleste andre eksperimentene i dette sette eksisterer det individer med både bedre egnethet og pålitelighet ved test med inverteringsfeil. Eksperimentet med 30 individer gjør det generelt like bra, men det er ett individ som trekker gjennomsnittet ned. Man kan spekulere på om ikke et høyt antall individer er en fordel, i og med at evolusjonen får et større genom og velge av.

Ettersom det er såpass få kjøringer som er gjort til hvert sett, kan man også spekulere på om disse resultatene ikke inneholder en del tilfeldigheter. En fra eller til, gir i dette tilfellet en stor forskjell.

6.2 Resultater fra eksperimentsett 2 : Blokker

Det andre settet med eksperimenter som ble utført, var simuleringer med endring i antall blokker i individet. Det ble kjørt eksperimenter med 20 til 120 blokker, med intervall på 20 blokker. Figur 6.3 og 6.4 viser resultatet av dette settet med eksperimenter.

I figur 6.3 viser X-aksen antall blokker som ble testet i settet av eksperimenter og, Y-aksen er egnethetsverdien av disse eksperimentene. Grafen som viser egnethet med inverteringsfeil viser en logaritmisk formet graf, som stiger sterkt fra 20 til 40, så mindre og mindre etterhvert. Den starter på -5 og stiger etterhvert til cirka -1. Maks- og minpunktene for denne grafen, viser at de tre første eksperimentene har en mye større spredning enn de siste tre. Igjen ser man at den enkle multiplikatoren, med egnethet på -18.33, blir overgått kraftig av alle eksperimenter. Det var kun eksperimentet med 20 blokker, som ikke presterte bedre en TMR-kretsen, med sitt resultat på -3.81. Det skal her gjøres oppmerksom på at TMR-kretsen inneholder 42 blokker.

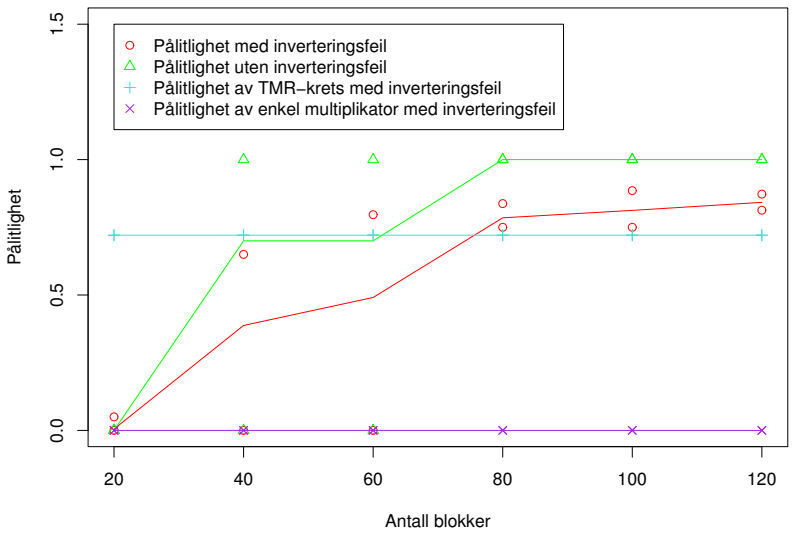


Figur 6.3: Egnethet for antall blokker. Diagrammet viser to grafer. Den ene viser gjennomsnittet av alle kjøringer, der det eksisterer en inverteringsfeil, og den andre, der det ikke eksisterer en inverteringsfeil.

Grafen som viser egnethet uten inverteringsfeil, ligger også som en svak logaritmisk formet graf, som går fra 2.1 ved 20 blokker til 1 ved 120 blokker. Maks- og minpunktene for denne grafen, viser at de starter med en ekstremt stor spredning

for 20 blokker, liten spredning for 40 og 60 blokker og ingen spredning for 80 blokker og oppover. Fra 40 blokker og oppover, vil minst en av kjøringene i hvert eksperiment produsere en krets som fungerer tilfredsstillende uten feil, og fra 80 blokker og oppover vil alle kjøringene produsere korrekte kretser.

I figur 6.4 viser X-aksen antall individer som ble testet i settet av eksperimenter og Y-aksen er pålitelighetsverdien av disse eksperimentene. Grafen som viser pålitelighet med inverteringsfeil, stiger kraftig fra 20 til 40 blokker, jevner seg litt ut mellom 40 og 60 blokker, og har igjen en sterk stigning 60 til 80 blokker. Fra 80 blokker, og oppover, stabiliserer den seg med en svak stigning. Ser man på maks- og minpunkt for denne grafen, kan man se at 20 blokker har en liten spredning fra 0 til 0.05, som er et generelt dårlig resultat. Punktene for 40 og 60 blokker gir stor spredning, mens for 80 blokker, og oppover, har punktene igjen lite spredning, høyt på grafen. I forhold til påliteligheten til den enkle multiplikatoren, er det kun eksperimentet med 20 blokker som kommer i nærheten, med en pålitelighet som er 0.005 bedre enn den enkle multiplikatoren. Når det gjelder TMR-kretsen, gjorde alle eksperimenter med 80 blokker, og oppover, det bedre en den.



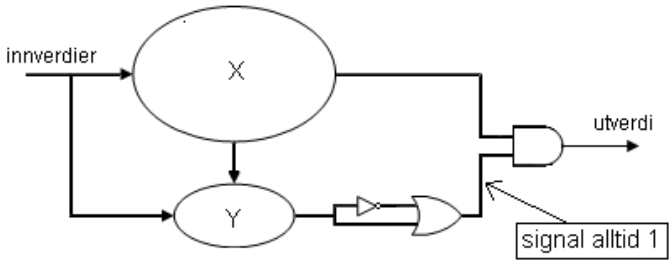
Figur 6.4: Pålitelighet for antall blokker. Diagrammet viser fire grafer. Den ene viser gjennomsnittet av alle kjøringer, der det eksisterer en inverteringsfeil, og den andre, der det ikke eksisterer en inverteringsfeil. Den tredje viser påliteligheten av en TMR-krets som er testet med inverteringsfeil. Den siste viser påliteligheten til en enkel multiplikator, som også er testet med inverteringsfeil.

Grafen som viser pålitelighet uten inverteringsfeil har en lignende form som påliteligheten med inverteringsfeil. Den stiger kraftig fra 20 til 40 blokker, flater

ut mellom 40 og 60 blokker, og har igjen en sterk stigning 60 til 80 blokker. Fra 80 blokker, og oppover, stabiliserer den seg på 1, som er det beste den kan oppnå. Maks- og minpunkt for 20 blokker ligger begge på 0, mens for for 40 og 60 blokker er ligger maks på 1 og min på 0. Maks og min for 80 blokker, og oppover, er 1.

Dette settet med eksperimenter, er det som har gitt størst utslag i resultatene. Resultatet tydet på at hvis man øker antall blokker i individet, så vil man oppnå en grenseverdi på egnetheten med inverteringsfeil. Dette kan antyde at det, med nok antall blokker, er mulig å evoluere fram en krets som fungerer med en hvilken som helst inverteringsfeil.

Forskning som Asbjørn Djupdal har gjort i [27, 28], viser at store kretser skaper klynger av blokker som ikke er tilgjengelig for kretsen. Figur 6.5 viser en av måtene dette konseptet kan oppstå. Etter en klynge blokker, oppstår det en kombinasjon av porter, som alltid resulterer i samme verdi. Dette kan være en av grunnene til at egnethet og pålitelighet øker, ettersom feil som oppstår i disse klyngene ikke vil ha effekt på resultatet.



Figur 6.5: Klynger av utilgjengelige blokker. Signalet fra Y ender opp i en kombinasjon av porter som alltid resulterer i 1. Utverdien vil derfor alltid være det samme som signalet som kommer ut ifra X.

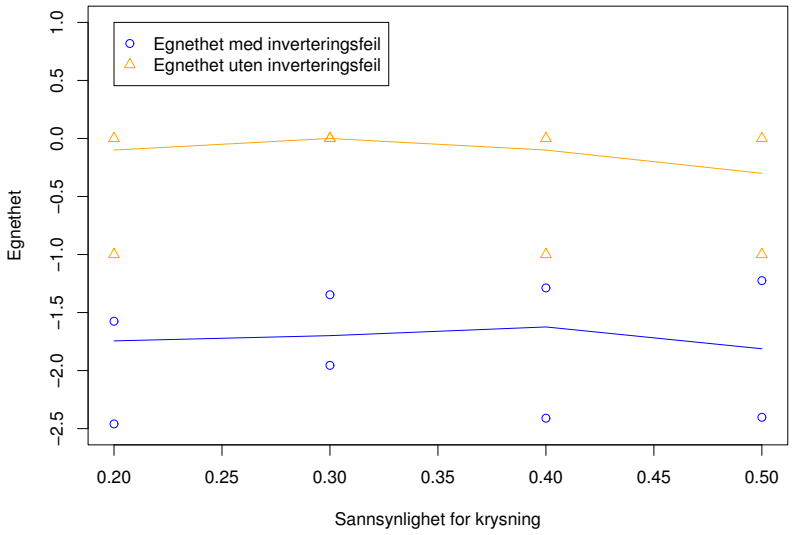
6.3 Resultater fra eksperimentsett 3 : Krysning

Det neste settet med eksperimenter som ble utført, var simuleringer med endring i sannsynligheten for krysning. Det ble kjørt eksperimenter med krysningsrate på 20% til 50%, med intervall på 10%. Diagrammene i figur 6.6 og 6.7 viser resultatet av dette settet med eksperimenter.

I figur 6.6 viser X-aksen sannsynlighetsraten for krysning som ble testet i settet av eksperimenter og, Y-aksen er egnethetsverdien av disse eksperimentene. Grafen som viser egnethet med inverteringsfeil, ligger som en jevn linje mellom -2.5 og -1.4. Maks- og minpunktene for denne grafen viser at de har en moderat spredning, mellom -2.5 og -1.4, for alle eksperimenter, men den er litt mindre

for eksperimentet med 30% krysningsrate. Her ligger de på -2.0 og -1.4. Alle kjøringene, i hele settet med eksperimenter, presterte bedre en den enkle multiplikatoren og TMR-kretsen.

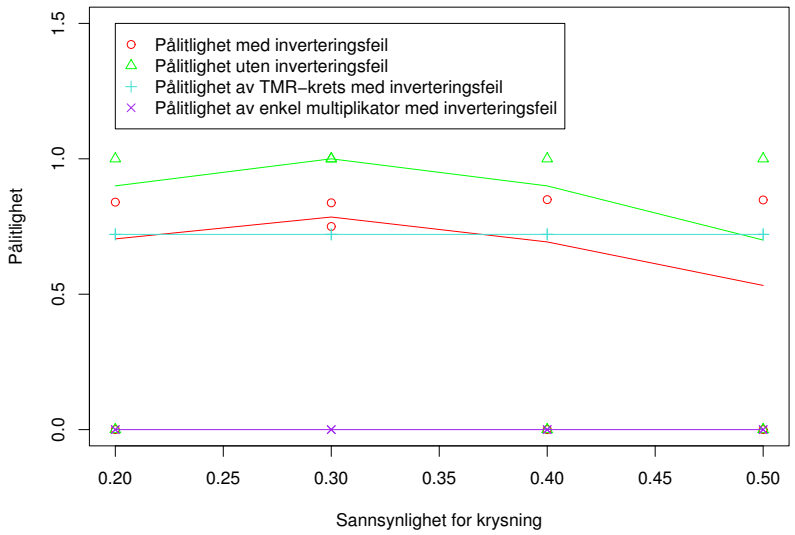
Grafen som viser egnethet uten inverteringsfeil, ligger også som en jevn linje, rett oppunder 0. Makspunktet for alle eksperimentene i settet er 0, og minpunkt er 1 for alle, bortsett fra når krysningsraten er 30%. Minpunktet for eksperimentet med 30% krysningsraten, er 0.



Figur 6.6: Egnethet for krysningssannsynlighet. Diagrammet viser to grafer. Den ene viser gjennomsnittet av alle kjøringer, der det eksisterer en inverteringsfeil, og den andre, der det ikke eksisterer en inverteringsfeil.

Diagrammet i figur 6.7 viser X-aksen sannsynlighetsraten for krysning som ble testet i settet av eksperimenter og, Y-aksen er pålitelighetsverdien av disse eksperimentene. Grafene som viser pålitelighet med og uten inverteringsfeil ligger med parallelle kurver , der pålitelighet uten inverteringsfeil ligger ca 0.2 over pålitelighet med inverteringsfeil. Spredningen av maks- og minpunktene, er for de fleste eksperimentene store. Det eneste eksperimentet med liten spredning av pålitelighet med inverteringsfeil, og ingen spredning på pålitelighet uten inverteringsfeil, var med 30% krysningsrate. Dette er også det eneste eksperimentet som gjorde det bedre enn TMR-kretsen. Alle eksperimentene gjorde det vesentlig mye bedre enn den enkle multiplikatoren.

Resultatene her viser en klar fordel ved 30% krysningsrate. Dette er den ideelle verdien, som skaper nye individer, og opprettholder genotypen som eksisterer



Figur 6.7: Pålitelighet for krysningssannsynlighet. Diagrammet viser fire grafer. Den ene viser gjennomsnittet av alle kjøringer, der det eksisterer en inverteringsfeil, og den andre, der det ikke eksisterer en inverteringsfeil. Den tredje viser påliteligheten av en TMR-krets som er testet med inverteringsfeil. Den siste viser påliteligheten til en enkel multiplikator, som også er testet med inverteringsfeil.

i populasjonen. Selv om alle eksperimentene produserte kretser som ga gode resultat, ga alle kjøringene for eksperimentet med 30% krysningsrate et godt resultat med mindre spredning i resultatene.

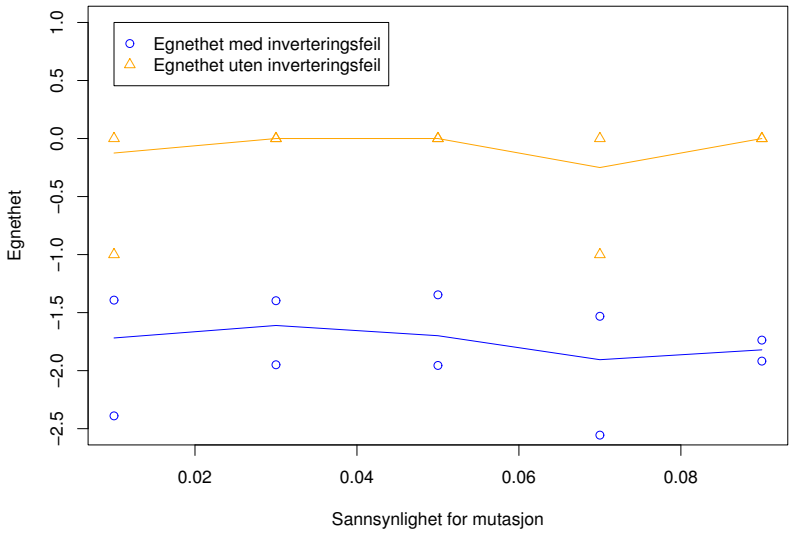
6.4 Resultater fra eksperimentsett 4 : Mutasjon

Det siste settet med eksperimenter som ble utført, var simuleringer med endring i sannsynligheten for mutasjon. Det ble kjørt eksperimenter med mutasjonsrate på 1% til 9%, med intervall på 2%. Diagrammene i figur 6.8 og 6.9 viser resultatet av dette settet med eksperimenter.

I figur 6.8 viser X-aksen sannsynlighetsraten for mutasjon som ble testet i settet av eksperimenter og, Y-aksen er egnethetsverdien av disse eksperimentene. Grafen som viser egnethet med inverteringsfeil, ligger som en jevn linje mellom -2.0 og -1.5. Maks- og minpunktene for denne grafen, viser at de har en moderat spredning for de 4 første eksperimenter, men den er litt mindre for 3% og 5% mutasjonsrate, enn 1% og 7%. For eksperimentet med 9% mutasjonsrate, er spredningen minimal.

Alle kjøringene i hele settet med eksperimenter, presterte bedre en TMR-kretsen, og vesentlig mye bedre en den enkle multiplikatoren.

Grafen som viser egnethet uten inverteringsfeil, ligger også som en jevn linje, rett oppunder 0. For eksperimentene med 1% og 7% er maks og minpunktet 0 og 1, men for resten er begge punktene 0.

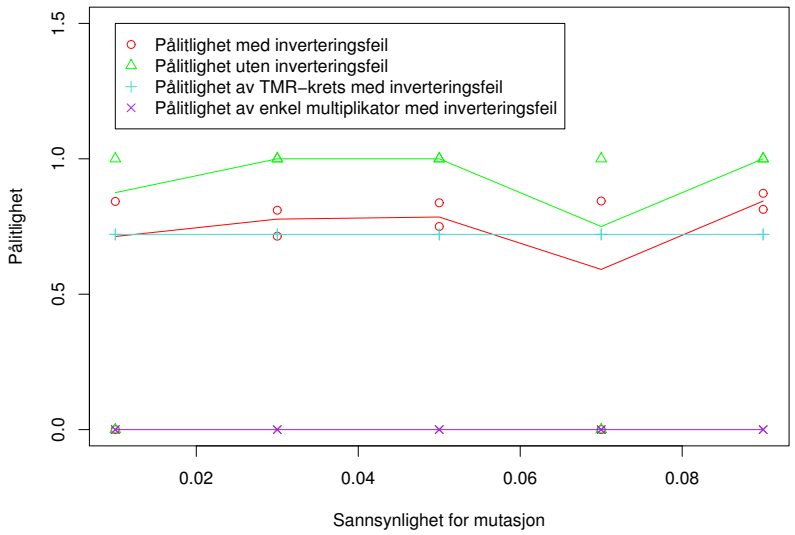


Figur 6.8: Egnethet for mutasjonssannsynlighet. Diagrammet viser to grafer. Den ene viser gjennomsnittet av alle kjøringer, der det eksisterer en inverteringsfeil, og den andre, der det ikke eksisterer en inverteringsfeil.

I figur 6.9 viser X-aksen sannsynlighetsraten for mutasjon som ble testet i settet av eksperimenter, og Y-aksen er pålitelighetsverdien av disse eksperimentene. Grafene som viser pålitelighet med og uten inverteringsfeil har, også her, parallelle grafer, der pålitelighet uten inverteringsfeil ligger ca 0.2 over pålitelighet med inverteringsfeil.

Spredningen på maks- og minpunktene, for eksperimentene med 1% og 7% mutasjonsrate, er store for begge grafene. Eksperimentene med 3%, 5% og 9% mutasjonsrate har, for grafen som viser pålitelighet med inverteringsfeil, liten spredning, og ingen spredning for pålitelighet uten feil. De tre sistnevnte eksperimentene gjorde det bedre enn TMR-kretsen, og alle gjorde det vesentlig mye bedre enn den enkle multiplikatoren.

For dette settet med eksperimenter er det mye usikkerhet. Generelt gjør alle kjøringene det bra, med unntak av et par kjøringer som trekker resultatet ned for



Figur 6.9: Pålitelighet for mutasjonssannsynlighet. Diagrammet viser fire grafer. Den ene viser gjennomsnittet av alle kjøringer, der det eksisterer en inverteringsfeil, og den andre, der det ikke eksisterer en inverteringsfeil. Den tredje viser påliteligheten av en TMR-krets som er testet med inverteringsfeil. Den siste viser påliteligheten til en enkel multiplikator, som også er testet med inverteringsfeil.

noen. Noe som er bemerkelsesverdig er at, selv om eksperimentet med 3% og 5% mutasjonsrate generelt gjør det bedre enn eksperimentet med 9%, når det gjelder egnethet, ble påliteligheten til 9% bedre enn både 3% og 5%. Det som kan ha betydning er at spredningen på egnethet til eksperimentet med 9%, er mindre enn de to andre.

Eksperimentet med 7% mutasjonsrate gjør det generelt like godt som de tre beste eksperimentene, men her er det to individer som trekker gjennomsnittet grundig ned. Noe som mest sannsynlig er tilfeldigheter.

6.5 En nærmere titt på de individuelle kjøringene

Dette delkapitlet tar for seg alle settene av eksperimenter, og går grundigere inn på resultatene av hver kjøring. Det blir her sett på egnethet, pålitelighet og antall blokker som brukes.

6.5.1 Egnethet og pålitelighet

Går man nærmere inn på verdiene fra hver og en kjøring, kan man se for eksperimentsett 1, 3 og 4, at alle egnethetsverdier som er lavere enn -2 ved endt evolusjon, gir -1 når kretsen testes uten feil, og verdier som er høyere en -2 gir egnethet 0. Når det gjelder pålitelighet for disse tre settene, kan man se at alle pålitelighetsverdiene ligger mellom 0.7 og 0.85, eller på 0. Det er de samme kjøringene som har mindre en -2 i egnethet som har 0 i pålitelighet. Resultatet av de individuelle kjøringene finnes i vedlegg 3.

Det er kun det andre settet med eksperimenter, som ser på blokker, som bryter med dette mønsteret. Når det gjelder eksperimentene med lavest antall blokker, 20, 40 og 60, som i utgangspunktet gjør det dårlig, er ikke resultatene like konsekvente. For eksperimentet med 20 blokker, gir en egnethet på -7.00 under test med inverteringsfeil, en egnethet på -4 i test uten inverteringsfeil, mens en annen kjøring gir -7.05 under test med inverteringsfeil, og -1 i test uten inverteringsfeil. Det finnes ingen mønster når man ser på resultatene til dette eksperimentet.

I eksperimentet med 40 blokker, ser man at to kjøringer som har resultert i en egnethet på -3.55 under test med inverteringsfeil, har resultert i -1 og 0 i test uten inverteringsfeil. Det kan i dette tilfellet betraktes som en grenseverdi, ettersom alle kjøringer med bedre resultat under test med inverteringsfeil, fikk 0 som egnethet i test uten inverteringsfeil, og alle kjøringer med dårligere resultat under test med inverteringsfeil fikk -1 som egnethet i test uten inverteringsfeil. I eksperimentet med 60 blokker ligger denne grenseverdien på cirka -2.5.

De tre siste eksperimentene i det andre settet, følger mønsteret til de tre andre settene, med unntak av at påliteligheten ligger mellom 0.75 og 0.89 for eksperimentet med 100 blokker, og 0.8 og 0.88 for eksperimentet med 120 blokker.

Ved å se på alle kjøringene, kan man se at i 79% av tilfellene få en fungerende multiplikator når kretsen testes uten feil. Hvis man fjerner det eksperimentet med dårligst resultat, eksperimentet med 20 blokker, øker det til 84%.

Her ser man igjen den effekten klyngene med utilgjengelige blokker kan ha på påliteligheten, som ble diskutert i seksjon 6.2. Desto større klynge, desto bedre resultat.

6.5.2 Antall brukte blokker

Ser man nærmere på hvor mange blokker som brukes i hvert eksperiment, i forhold til hvor mange blokker som er tilgjengelig, ser man at desto flere blokker som

er tilgjengelig desto lavere prosentandel av dem brukes. En oversikt over hvor mange blokker som brukes i hvert eksperiment finnes i vedlegg 3. Tabell 6.1 viser hvor mange prosent av de tilgjengelige blokkene som brukes i hvert eksperiment. Tabellen viser hver at desto større antall blokker et individ har, desto større redundans av blokker får man. Denne typen redundans ble sett på i seksjon 2.3.1, der det ble nevnt av redundans skaper nøytralitet.

	Prosentandel av brukte blokker	
20 blokker	**Lavest antall**	**Gjennomsnitt**
20 blokker	95.00%	99.00%
40 blokker	95.00%	97.50%
60 blokker	91.00%	95.00%
80 blokker	82.50%	92.25%
100 blokker	82.00%	90.00%
120 blokker	73.00%	86.25%

Tabell 6.1: Prosentandel av brukte blokker. Kolonnen *Lavest antall* viser prosentandel av brukte blokker i det individet som brukte færrest blokker.

6.6 Egnethet over tid

Figurene 6.10 til 6.17 ser på hvordan egnetheten har bedret seg i løpet av evolusjonen. Figurene 6.10, 6.11, 6.12 og 6.13 ser på de første 15 000 generasjonene, og figurene 6.14, 6.15, 6.16 og 6.17 ser på hvordan egnetheten har utviklet seg over hele evolusjonen.

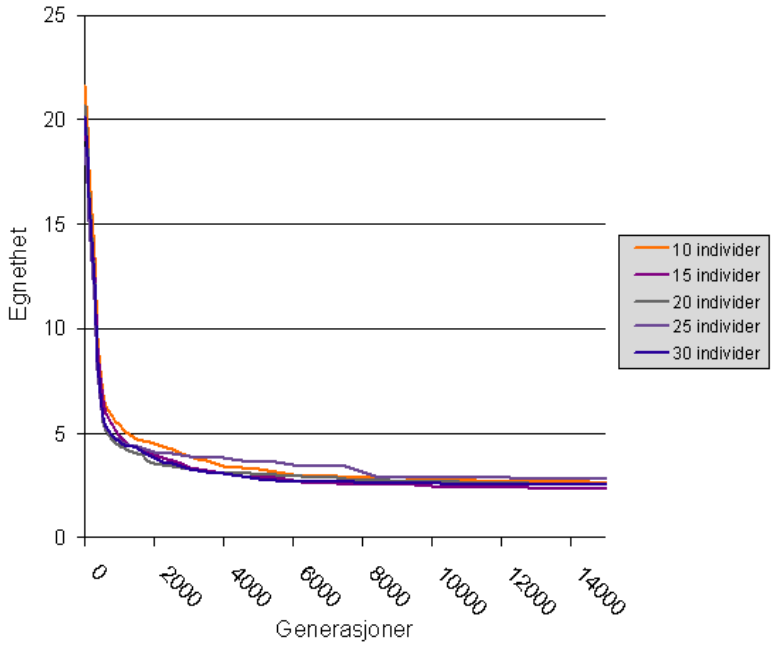
Ved å se på figurene 6.10, 6.11, 6.12 og 6.13 kan man se deres likhet. Alle synker drastisk innen de første 1 000 generasjonene, og ved generasjon 10 000 stabiliserer grafene seg. Endringen på de første 1 000 generasjonene er gjennomsnittlig 16.54 for alle eksperimentene, der det settet som synker kraftigst, er settet med eksperimenter der individene ble sett på, med en nedgang på 17.08. Når det gjelder stabiliseringen fra generasjon 10 000 til 15 000, er det eksperimentsettet med individer som har minst endring. Resten av resultatene her kan leses i tabell 6.2.

Ser man på grafene i figurene 6.14, 6.15, 6.16 og 6.17, kan man se hvor stabil egnetheten egentlig blir. I løpet av de siste 250 000 generasjonene, som utgjør halve evolusjonen, forbedrer egnetheten seg gjennomsnittlig med 0.13, og i løpet av de siste 400 000 forbedrer egnetheten seg med 0.32. Det settet med eksperimenter som viste seg å ha størst stabilitet er den med individer, mens den med størst endring er settet med krysningsraten. Resultatene for restene av eksperimentsettene kan leses i tabell 6.2

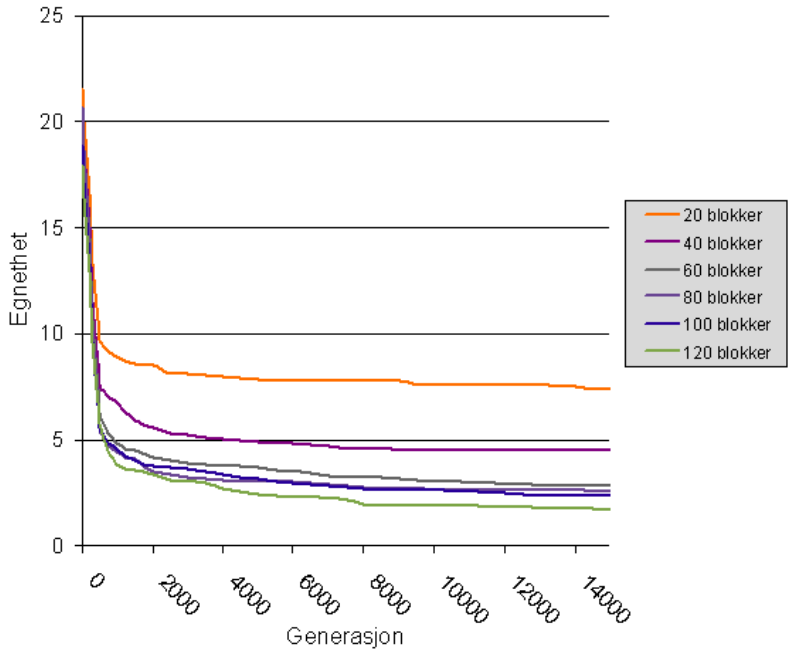
Endring av egnethet for generasjonene(per tusen):				
Ekperimentsett	**0-5**	**10-15**	**100-500**	**250-500**
Individer	17.08	0.09	0.26	0.09
Blokker	15.78	0.16	0.35	0.19
Krysningsrate	16.75	0.17	0.36	0.13
Mutasjonsrate	16.56	0.14	0.29	0.10
Gjennomsnitt	16.56	0.14	0.32	0.13

Tabell 6.2: Endring av egnethet over et sett generasjoner. Generasjoner er vist i per tusen.

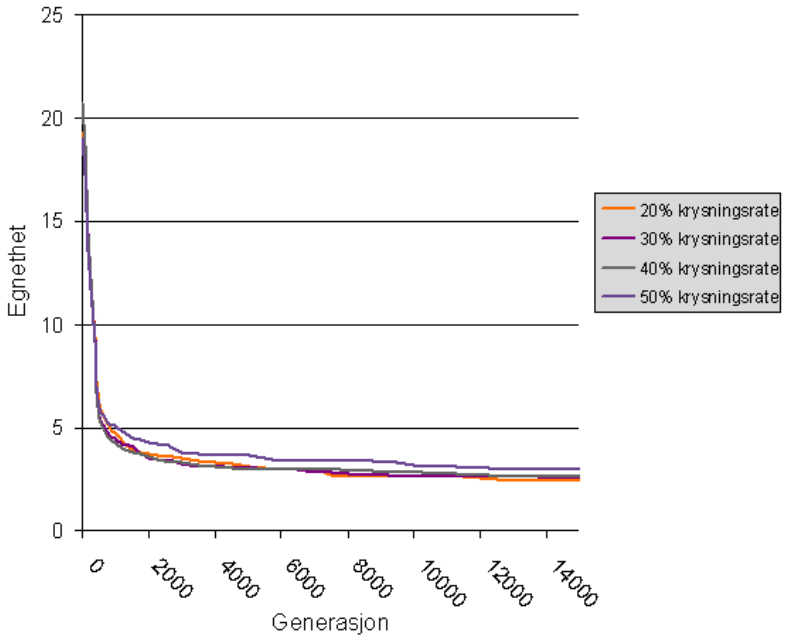
Med hensyn til til den lave endringen av egnethet på de siste 250 000 generasjonene, hadde det vært interessant å se hva slags resultater man får ved å halvere evolusjonen. I forhold til de resultatene som er gitt, er det grunnlag til å tro at forskjellen blir minimal, om ikke tilfeldig. Ettersom kjøretiden til eksperimentene var opptil 100 timer, hadde det vært en stor fordel og kunne kutte ned dette. Ved å halvere antall generasjoner, vil man også halvere kjøretiden.



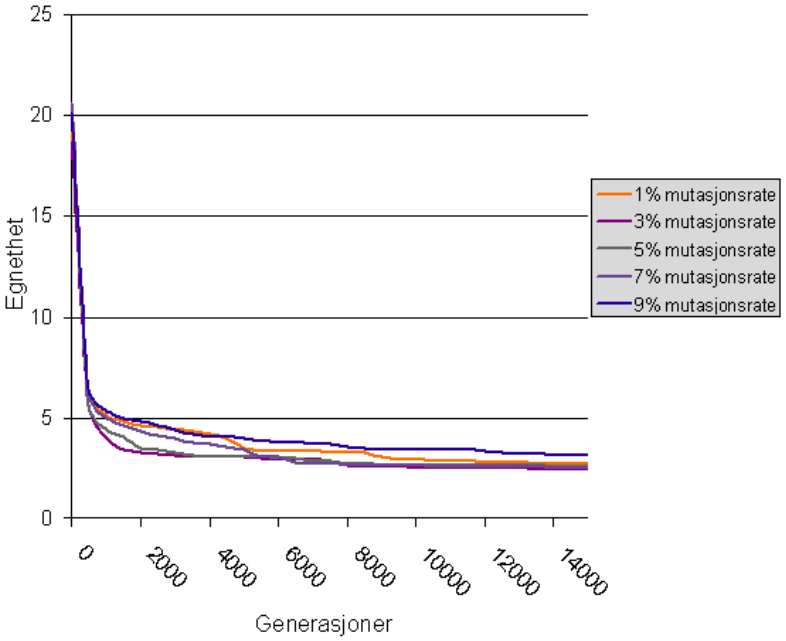
Figur 6.10: Grafen viser utviklingen av egnethet gjennom de 15 000 første generasjonene av evolusjonen, for settet av eksperimenter der individene var i fokus.



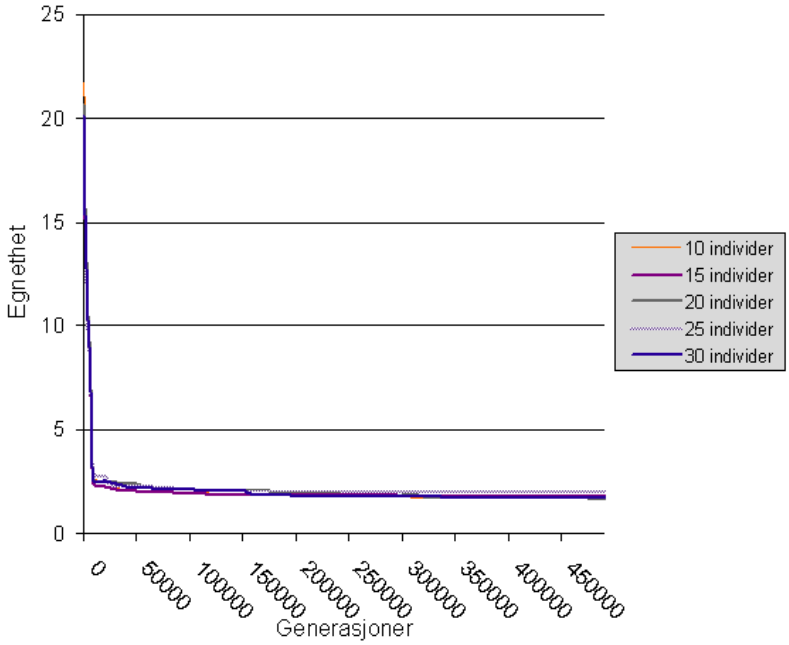
Figur 6.11: Grafen viser utviklingen av egnethet gjennom de 15 000 første generasjonene av evolusjonen, for settet av eksperimenter der antall blokker var i fokus.



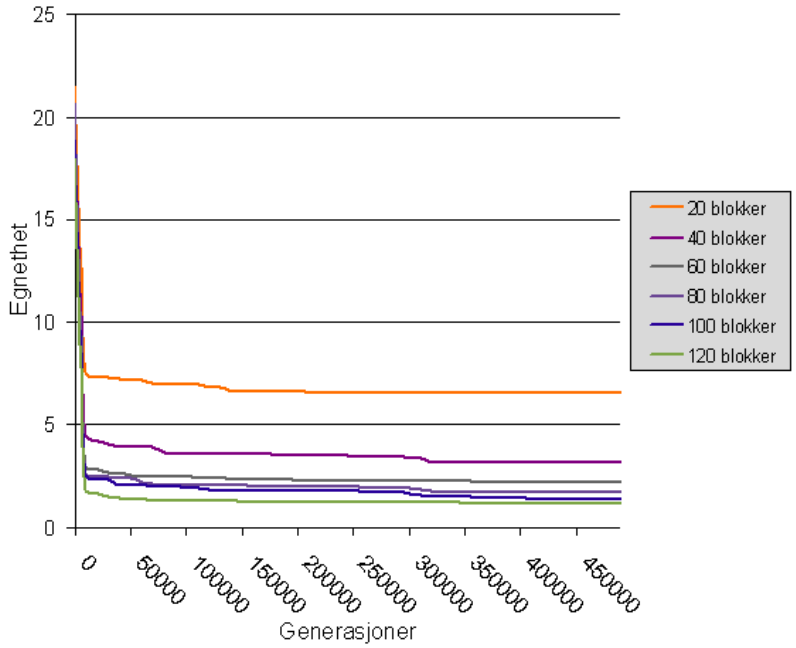
Figur 6.12: Grafen viser utviklingen av egnethet gjennom de 15 000 første generasjonene av evolusjonen, for settet av eksperimenter der krysningsraten var i fokus.



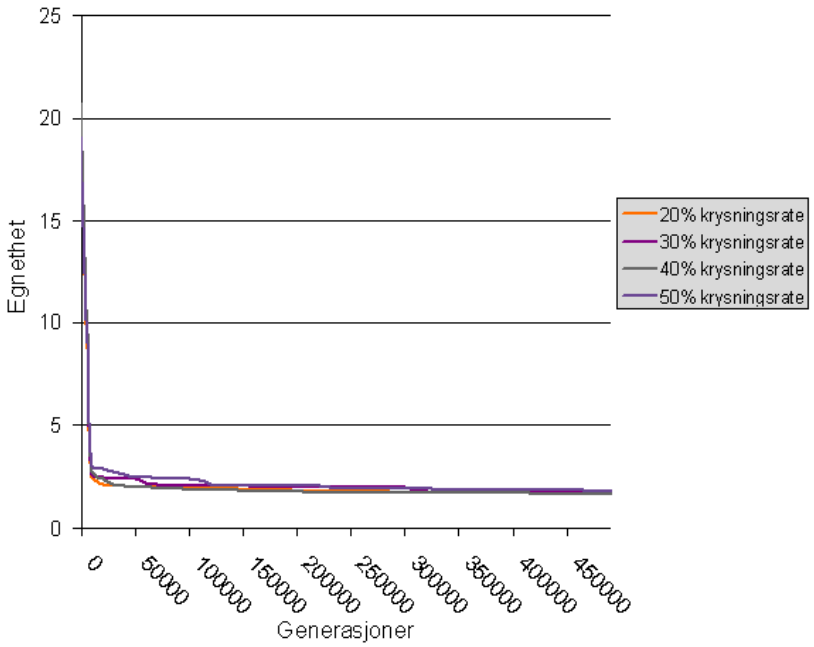
Figur 6.13: Grafen viser utviklingen av egnethet gjennom de 15 000 første generasjonene av evolusjonen, for settet av eksperimenter der mutasjonsraten var i fokus.



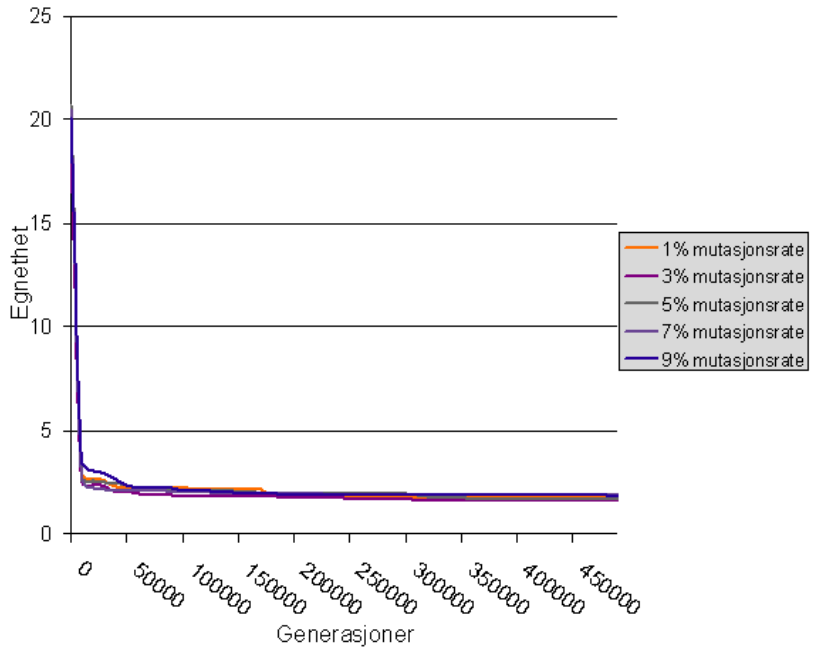
Figur 6.14: Grafen viser utviklingen av egnethet gjennom hele evolusjonen, for settet av eksperimenter der antall individer er i fokus.



Figur 6.15: Grafen viser utviklingen av egnethet gjennom hele evolusjonen, for settet av eksperimenter der antall blokker er i fokus.



Figur 6.16: Grafen viser utviklingen av egnethet gjennom hele evolusjonen, for settet av eksperimenter der krysningsraten er i fokus.



Figur 6.17: Grafen viser utviklingen av egnethet gjennom hele evolusjonen, for settet av eksperimenter der mutasjonsraten er i fokus.

Kapittel 7

Konklusjon og videre arbeid

7.1 Konklusjon

Bio-inspirert utvikling har vist seg å være veldig skalerbart, noe forskere innen EHW finner meget attraktivt. De ønsker å utvikle en kunstig metode for å etterligne den evnen biologiske organismer har for feiltoleranse og tilpasningsdyktighet. Sammen med genetiske algoritmer, har denne typen kunstig utvikling blitt brukt til å produsere enkle kretser med vellykket resultat.

Denne rapporten har sett på evolusjon av kretser der det alltid eksisterer en inverteringsfeil, og resultatene er positive. Det viser seg at i de fleste tilfeller vil evolusjon med singel inverteringsfeil, produsere en fungerende krets, når feilen ikke er til stede.

Det har også vist seg at kretser, som utvikles med singel inverteringsfeil, har en høy pålitelighet, selv med feilen til stede. Hvis en feil skulle dukke opp er det cirka 80% sannsynlighet for at kretsen fungerer feilfritt. Og skulle feilen oppstå i en i en kritisk blokk, vil den regne riktig i de fleste tilfellene.

7.2 Videre arbeid

7.2.1 Grundigere forskning på hvilke typer kretser som resulterte av disse resultatene

Resultatene viste at mange av individene hadde en pålitelighet på cirka 0.8, som sier at hvis det skulle dukke opp en feil i kretsen, så er det 80% sannsynlighet

at kretsen vil fungere 100% korrekt. Men hvor stor blir regnefeilen, hvis feilen i kretsen gir feil resultat. Dette kan ha betydning for feiltoleransen av en krets. Er regnefeilen liten, kan det ha lite å si i det lange løpet.

7.2.2 Utvidelse av funksjonaliteter til simulatoren

Det er mulig å utvide valgmulighetene man har ved kjøring av simulatoren betraktelig. Seleksjonsalgoritmen som ble brukt i dette prosjektet, er foreløpig den eneste som er implementert, men det er lagt opp til flere muligheter. Det er også mulig å implementere og teste andre genetiske algoritmer, en den som er brukt her.

Det hadde også vært en mulighet til å legge inn flere feilmodeller, blant annet mangfoldig stuck-at og open stuck-at, som ble forklart i seksjon 2.4.

7.2.3 Parallell kjøring

Det er her to muligheter som kan utnyttes ved og parallellisere simulatoren. Den første er at tiden på en kjøring kan reduseres, og den andre muligheten er migrasjon av individer.

Ettersom mengden tid det tok for å kjøre disse eksperimentene var så lang, hadde det vært en stor fordel om programmet kunne kjørt parallelt. Man må være oppmerksom på at man da trenger flere noder å kjøre på enn ved sekvensiell kjøring. Det er inne vits i å parallellisere om man likevel må kjøre simuleringene etter hverandre.

Hvis hver simulering kjøres sekvensielt, med er koblet sammen parallelt, kan simuleringene bytte individer med hverandre. Dette kan skape en mer variert genom.

Bibliografi

[1] Anders Engh-Halstvedt, Frode Eskelund *Evolution of fault-tolerant digital systems*, Prosjektoppgave NTNU, November 2001

[2] Frode Eskelund *Evolution of Robust Circuits In a Simulated Enviornment*, Masteroppgave NTNU, Juni 2002

[3] Morten Hartmann *Evolution of Fault and Noise Tolerant Digital Circuits*, Doktorgradsavhandling NTNU, April 2005

[4] Charles Darwin *The Origin of Species by Means of Natural Selection* Random House inc., New York, USA 1859

[5] J.Koza *Genetic Programming*, The MIT Press, 1993

[6] Julian F. Miller, Peter Thomson *Cartesian Genetic Programming*, Third European Conference om Genetic Programming, Edinburg 2000.

[7] Vesselin K. Vassilev, Julian F. Miller *The Advantages of Landscape Neutrality in Digital Circuit Evolution*. Third International Conference on Evolvable Systems: From Biology to Hardware, Edinburgh, 2000, sider 252-263

[8] Gunnar Tufte, Pauline Haddow *Prototyping a GA Pipeline for Complete Hardware Evolution*, Evolvable Hardware, 1999. Proceedings of the First NASA/DoD, sider 18-25.

[9] The R Project for Statistical Computing http://www.r-project.org/, aksessert mars 2007

[10] J. von Neumann *Probabalistic logics and synthesis of reliable organisms from unreliable components*, Automata Studies of; Annals of Mathematical Studies, 1956, sider 43-98.

[11] A. Thompson *Evolving fault tollerant systems* Proc. 1st IEE/IEEE Int. Conf. on Genetic Algorithms in Engineering Systems: Innovations and Applications (GALESIA'95), sider 524-529, 1995

[12] A. Thompson and P. Layzell *Analysis of Unconventional Evolved Electronics* Communications of the ACM, sider 71-79, 1999

[13] M. Garvie and A. Thompson*Evolution of Self-Diagnosing Hardware*, Proc. 5th Int. Conf. on Evolvable Systems (ICES2003): From biology to hardware, sider 238-248, 2003

[14] A. Thompson and P. Layzell *Evolution of Robustness in an Electronics Design* Proc. 3rd Int. Conf. on Evolvable Systems (ICES2000): From biology to hardware,sider 218-228, 2000.

[15] J.F. Miller *An empirical study of the efficiency of learning boolean functions using a Cartesian Genetic Programming Approach* Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'99), Orlando, Florida, USA, 13-17 July 1999, pp. 1135-1142.

[16] James A. Walker and Julian F. Miller *Evolution and Acquisition of Modules in Cartesian Genetic Programming* Proceedings of EuroGp 2004.

[17] Yang Zhang, Smith, S.L. Tyrrell, A.M. *Digital circuit design using intrinsic evolvable hardware*, Evolvable Hardware, 2004. Proceedings. 2004 NASA/DoD Conference, sider 55 - 62

[18] Heng Liu, Miller, J.F., Tyrrell, A.M. *Intrinsic evolvable hardware implementation of a robust biological development model for digital systems*, Evolvable Hardware, 2005. Proceedings. 2005 NASA/DoD, sider: 87 - 92.

[19] Tyrrell, A.M., Sun, H. *A Honeycomb Development Architecture for Robust Fault-Tolerant Design*, 1st NASA/ESA IEEE Conference on Adaptive Hardware Systems, IEEE Computer Society Press, 2006, sider 281-287.

[20] Portable Batch System http://www.openpbs.org/, aksessert juni 2007.

[21] Feilmodeller http://en.wikipedia.org/wiki/Fault_model, aksessert mai 2007

[22] Beowulf beregnigsklynge http://en.wikipedia.org/wiki/Beowulf_(computing), aksessert juni 2007

[23] Clustis2 http://norgrid.ntnu.no/norgrid/Clustis2Introduction, aksessert juni 2007

[24] P.K. Lehre *Complexity Issues in Artificial Development*, Hovedoppgave NTNU, 2004

[25] Foiler fra Indian Institute of Tecnology Kharagpur http://sit.iitkgp.ernet.in/archive/teaching/ee497f/rassp_43/sld018.htm, slide 18-26, aksessert mai 2007

[26] The Bio-Inspired Architectures lab ved universitetet i York http://www.elec.york.ac.uk/intsys/projects/inspired.htm, aksessert mai 2007

[27] A. Djupdal, P. C. Haddow *Evolving Redundant Structures for Reliable Circuits–Lessons Learned*, Adaptive Hardware and Systems 2007, Kommer i AHS i 2007.

[28] A. Djupdal, P. C. Haddow *Evolving and Analysing Useful Redundant Logic*, ICES 2007, Kommer i ICES i 2007.

[29] Kalganova T. *Bidirectional Incremental Evolution in Evolvable Hardware.* Proc. of The Second NASA/DoD Workshop on Evolvable Hardware (EH'2000).

[30] Kalganova T. *An Extrinsic Function-Level Evolvable Hardware Approach.* Proc. of the Third European Conference on Genetic Programming, EuroGP2000.

[31] M.Hartmann, P. Haddow, A.Djupdal *Addressing the Metric Challenge: Evolved versus Traditional Fault Tolerant Circuits*

[32] K. Zhang, R. DeMara og C. Sharma *Consensus-Based Evaluation for Fault Isolation and On-line Evolutionary Regeneration* ICES 2005, LNCS 3637, Springer-Verlag Berlin Heidelberg 2005, sider 12-24.

Vedlegg 1

Ordliste

Avbildning	Mapping
Beregningsklynge	Cluster
Beregningsnode	Computational node
Egnethet	Fitness
Ekstern evolusjon	Extinsik evolution
Foreningsfeil	Bridging foult
Forsinkelsesfeil	Delay foult
Intern evolusjon	Intrinsik evolution
Krysning	Crossover
Mangfoldig	Multiple
Matrise	Array
Pålitelighet	Reliablilty
Strek	Pipe
Styrenode	Frontend machine
Velger	Voter

Vedlegg 2

Forkortelser

CAD	Computer Aided Design
CGP	Cartesian Genetic Programming
CHE	Complete Hardware Evolution
CMOS	Complementary Metal Oxide Semiconductor
CRAB	Complex, Reconfigurable, Adaptiv, Bio-inspired Hardware
EA	Evolusjonær Algoritme
EHW	Evolutionary Hardware
EP	Evolusjonær Programmering
ES	Evolusjonær Strategi
FPGA	Field Programmable Gate Array
GA	Genetisk Algoritme
GP	Genetisk Programmering
NMR	N-Modulær Redundans
TMR	Trippel Modulær Redundans
TS	Tournament Seleksjon
ULSI	Ultra Large Scale Integration
VLSI	Very Large Scale Integration

Vedlegg 3

Programkode

"batchsubmit.sh"

```
1  #!/bin/bash
2
3  #
4  # batchsubmit.sh <jobs> <evocfg> <testcfg> <jobcfg> <dirname>
5  #
6
7  jobs=$1
8  evocfg=$2
9  testcfg=$3
10 jobcfg=$4
11 dirname=$5
12
13 #Oppretter em mappe og kopierer inn filene i denne mappen
14 mkdir ${dirname}
15 cp $evocfg ${dirname}/
16 cp $testcfg ${dirname}/
17 cp $jobcfg ${dirname}/
18 cd ${dirname}/
19
20 # for antall jobber som skal kjores
21 for ((job=1;job <= jobs;job++))
22 do
23   #kopier parameterfilen, saa den faar et eget navn for hver jobb
24   #og generer froet som med rand32.sh, og legg det i parameterfilen
25   cp $evocfg ${evocfg}_$job
26   echo -e "\n#Seed generated by submit.sh" >> ${evocfg}_$job
27   echo -e "seed \t `rand32.sh`" >>  ${evocfg}_$job
28
29   #kopier testfilen og jobfilen, saa den faar et eget navn for hver
         jobb
30   cp $testcfg ${testcfg}_$job
31   cp $jobcfg ${jobcfg}_$job
32
33   #Hent inn stien til denne mappen og legg den i en variabel
```

```
34    pwd_temp=`pwd`
35
36    #Legg til navnet man onsker og kalle kjoringen og goto mappe, inn
          i jobbfilen
37    echo \#PBS -N ehw_${job} >> ${jobcfg}_$job
38    echo cd `pwd` >> ${jobcfg}_$job
39
40    #Legg inn kommandoene for aa kjore ehw paa clustis med param-fil,
41    #exstract for aa hente ut resultatindividet og kjoring av ehw
42    #med testfil
43    echo "~maylinda/Master/ehw/ehw ${evocfg}_$job 0 0 > out_$job" >> $
          {jobcfg}_$job
44    echo "~maylinda/bin/extract out_$job 1 >> ${testcfg}_$job" >> ${
          jobcfg}_$job
45    echo "~maylinda/Master/ehw/ehw ${testcfg}_$job 0 0 >> out_$job" >>
           ${jobcfg}_$job
46
47    #Legg til jobben i ko paa clustis
48    qsub ${jobcfg}_$job
49
50  done
```

"extractall.sh"

```
1  #!/bin/bash
2
3  #
4  # extractall.sh [filnavn] [options]
5  #
6
7  filnavn=$1
8  options=$2
9
10 #Hvis -R er satt, startes en rekursiv prosedyre som starter opp
11 #dette programmet for hver subkatalog
12 if [[ $options = "-R" ]]
13 then
14     for dir in *
15     do
16       if [ -d $dir ]
17       then
18     cd $dir
19     extractall.sh $filnavn -R
20     cd ..
21       fi
22     done
23 fi
24
25 #leser fra alle out_* filene og henter ut data og legger det
26 #i samme fil
27 if [ -e out_1 ]
28 then
29     #Skriver ut overskriftene
30     extract out_1 2 >> $filnavn.results
31     for file in out_*
32     do
33       #Skriver ut data
34       extract $file 0 >> $filnavn.results
35     done
36 fi
```

"extract31.sh"

```
1  #!/bin/bash
2
3  #
4  # extract31.sh [filnavn] [options]
5  #
6
7  filnavn=$1
8  options=$2
9
10 #Hvis -R er satt, startes en rekursiv prosedyre som starter opp
11 #dette programmet for hver subkatalog
12 if [[ $options = "-R" ]]
13 then
14     for dir in *
15     do
16       if [ -d $dir ]
17       then
18     cd $dir
19     extract31.sh $filnavn -R
20     cd ..
21       fi
22     done
23 fi
24
25 #leser fra alle out_* filene og henter ut data og legger det
26 #i samme fil
27 if [ -e out_1 ]
28 then
29     #Skriver ut overskriftene
30     extract 3 > $filnavn.txt
31     for file in out_*
32     do
33       #Skriver ut data
34       extract $file 6 >> $filnavn.txt
35       extract 3 >> $filnavn.txt
36     done
37 fi
```

"extract20.sh"

```
1  #!/bin/bash
2
3  #
4  # extract20.sh [filnavn] [options]
5  #
6
7  filnavn=$1
8  options=$2
9
10 #Hvis -R er satt, startes en rekursiv prosedyre som starter opp
11 #dette programmet for hver subkatalog
12 if [[ $options = "-R" ]]
13 then
14     for dir in *
15     do
16       if [ -d $dir ]
17       then
18     cd $dir
19     extract20.sh $filnavn -R
20     cd ..
21       fi
22     done
23 fi
24
25 #leser fra alle out_* filene og henter ut data og legger det
26 #i samme fil
27 if [ -e out_1 ]
28 then
29     #Skriver ut overskriftene
30     extract 3 > $filnavn.txt
31     for file in out_*
32     do
33       #Skriver ut data
34       extract $file 5 >> $filnavn.txt
35       extract 3 >> $filnavn.txt
36     done
37 fi
```

"Resultat av enkel multiplikatortest med inverteringsfeil"

```
1
2  Using specified individual
3
4
5  gate_count:           80
6  generations:          500000
7  individuals:          20
8  report_gap:           500
9  groupsize:            5
10 elitism:              1
11 sel.prob:       0.7
12 crossoverrate:        0.3
13 mutationrate:         0.05
14 size_factor:          0
15 fitness_limit:        0
16 algorithm:            5
17 bias:                 3
18 seed:                 1182238420
19 inputs:               4
20 outputs:              4
21 data:                 16 specified, 16 found
22 circuit_type:         1
23 fail_type:            1
24 fitness_method:       0
25 output_protect:       0
26
27 single stuck at run: 0
28 failrate:             0
29 noise:                0
30 error_tests:          1
31 error_runs:           10
32
33 This is an evolutionary run!
34
35 Generation: 0
36  Best fitness: 0
37 Evolution complete (used: 0 seconds)
38
39
40 best_individual:  Individual{ size: 9 used: 9 fitness: 0 reliability
       : 1 blocks: (4: 1 AND 2)(5: 0 AND 3)(6: 0 AND 2)(7: 4 NAND 5)(8:
       4 OR 5)(9: NOT 7)(10: 6 AND 7)(11: 7 AND 8)(12: 1 AND 3) }
41 best_fitness: 0
42 reliability:  1
43 total_gens: 0
44 used_gates: 9
45
46 Using specified individual
47
48
```

```
49 t_seed:              1182238420
50 t_inputs:            4
51 t_outputs:           4
52 t_data:              16 specified, 16 found
53 t_circuit_type:      1
54
55 t_fail_type:         1
56 t_fitness_method:    0
57 t_output_protect:    0
58
59 single stuck at run: 1
60 t_failrate:          0
61 t_noise:             0
62 t_error_tests:       1
63 t_error_runs:        10
64
65 This is a testing run!
66
67 Testing complete (used: 0 seconds)
68
69
70 t_best_individual:  Individual{ size: 9 used: 9 fitness: -18.3333
      reliability: 0 blocks: (4: 1 AND 2)(5: 0 AND 3)(6: 0 AND 2)(7: 4
      NAND 5)(8: 4 OR 5)(9: NOT 7)(10: 6 AND 7)(11: 7 AND 8)(12: 1 AND
      3) }
71 t_best_fitness: -18.3333
72 t_reliability:  0
73 t_used_gates: 9
```

"Resultat av tmrkrets-test med inverteringsfeil"

```
1
2  Using specified individual
3
4
5  gate_count:           80
6  generations:          500000
7  individuals:          20
8  report_gap:           500
9  groupsize:            5
10 elitism:              1
11 sel.prob:             0.7
12 crossoverrate:        0.3
13 mutationrate:         0.05
14 size_factor:          0
15 fitness_limit:        0
16 algorithm:            5
17 bias:                 3
18 seed:                 1182238420
19 inputs:               4
20 outputs:              4
21 data:                 16 specified, 16 found
22 circuit_type:         1
23 fail_type:            1
24 fitness_method:       0
25 output_protect:       0
26
27 single stuck at run: 1
28 failrate:             0
29 noise:                0
30 error_tests:          1
31 error_runs:           10
32
33 This is an evolutionary run!
34
35 Generation: 0
36  Best fitness: 0
37 Evolution complete (used: 0 seconds)
38
39
40 best_individual:  Individual{ size: 43 used: 43 fitness: 0
       reliability: 1 blocks: (4: 1 AND 2)(5: 0 AND 3)(6: 0 AND 2)(7: 4
      NAND 5)(8: 4 OR 5)(9: NOT 7)(10: 6 AND 7)(11: 7 AND 8)(12: 1 AND
      3)(13: 1 AND 2)(14: 0 AND 3)(15: 0 AND 2)(16: 13 NAND 14)(17: 13
      OR 14)(18: NOT 16)(19: 15 AND 16)(20: 16 AND 17)(21: 1 AND 3)(22:
       1 AND 2)(23: 0 AND 3)(24: 0 AND 2)(25: 22 NAND 23)(26: 22 OR 23)
      (27: NOT 25)(28: 24 AND 25)(29: 25 AND 26)(30: 1 AND 3)(31: 18 OR
       27)(32: 18 AND 27)(33: 9 AND 31)(34: 19 OR 28)(35: 19 AND 28)
      (36: 10 AND 34)(37: 20 OR 29)(38: 20 AND 29)(39: 11 AND 37)(40:
      21 OR 30)(41: 21 AND 30)(42: 12 AND 40)(43: 32 OR 33)(44: 35 OR
      36)(45: 38 OR 39)(46: 41 OR 42) }
```

```
41 best_fitness: 0
42 reliability:  1
43 total_gens: 0
44 used_gates: 43
45
46 Using specified individual
47
48
49 t_seed:              1182238420
50 t_inputs:            4
51 t_outputs:           4
52 t_data:              16 specified, 16 found
53 t_circuit_type:      1
54
55 t_fail_type:         1
56 t_fitness_method:    0
57 t_output_protect:    0
58
59 single stuck at run: 1
60 t_failrate:          0
61 t_noise:             0
62 t_error_tests:       1
63 t_error_runs:        10
64
65 This is a testing run!
66
67 Testing complete (used: 0 seconds)
68
69
70 t_best_individual:  Individual{ size: 43 used: 43 fitness: -3.81395
       reliability: 0.72093 blocks: (4: 1 AND 2)(5: 0 AND 3)(6: 0 AND 2)
       (7: 4 NAND 5)(8: 4 OR 5)(9: NOT 7)(10: 6 AND 7)(11: 7 AND 8)(12:
       1 AND 3)(13: 1 AND 2)(14: 0 AND 3)(15: 0 AND 2)(16: 13 NAND 14)
       (17: 13 OR 14)(18: NOT 16)(19: 15 AND 16)(20: 16 AND 17)(21: 1
       AND 3)(22: 1 AND 2)(23: 0 AND 3)(24: 0 AND 2)(25: 22 NAND 23)(26:
        22 OR 23)(27: NOT 25)(28: 24 AND 25)(29: 25 AND 26)(30: 1 AND 3)
       (31: 18 OR 27)(32: 18 AND 27)(33: 9 AND 31)(34: 19 OR 28)(35: 19
       AND 28)(36: 10 AND 34)(37: 20 OR 29)(38: 20 AND 29)(39: 11 AND
       37)(40: 21 OR 30)(41: 21 AND 30)(42: 12 AND 40)(43: 32 OR 33)(44:
        35 OR 36)(45: 38 OR 39)(46: 41 OR 42) }
71 t_best_fitness: -3.81395
72 t_reliability:  0.72093
73 t_used_gates: 43
```

"Antall blokker brukt i resultatindividene"

```
1  Eksperimentsett 1 : Individer med 80 blokker
2
3   10 individer : 75 71 72 74 74 75 72 72 74 77
4   15 individer : 69 71 74 73 73 74 75 78 74 69
5   20 individer : 74 73 71 74 78 76 73 73 66 80
6   25 individer : 64 71 75 77 69 75 76 70 77 65
7   30 individer : 76 77 76 70 76 74 79 73 71 78
8
9
10 Eksperimentsett 2 : Blokker
11
12  20 blikker  : 19  20  20 20  19  20 20  20  20  20
13  40 blokker  : 40  40  38 39  39  39 38  40  39  38
14  60 blokker  : 56  58  57 55  58  59 57  59  55  56
15  80 blikker  : 74  73  71 74  78  76 73  73  66  80
16  100 blokker : 82  93  97 90  90  92 84  85  94  93
17  120 blokker : 102 112 88 105 109 96 106 107 106 104
18
19
20 Eksperimentsett 3 : Krysning med 80 blokker
21
22  20% krysningsrate : 76 71 77 76 73 77 74 74 76 70
23  30% krysningsrate : 74 73 71 74 78 76 73 73 66 80
24  40% krysningsrate : 75 72 76 76 77 74 79 72 73 73
25  50% krysningsrate : 73 70 75 74 75 76 69 75 80 72
26
27
28 Eksperimentsett 4 : Mutasjon med 80 blokker
29
30  1% mutasjonsrate : 73 72 76 75 75 70 76 77
31  3% mutasjonsrate : 73 77 75 76 70 76 75 75
32  5% mutasjonsrate : 74 73 71 74 78 76 73 73 66 80
33  7% mutasjonsrate : 74 72 67 74 74 76 68 78
34  9% mutasjonsrate : 73 70 68 74 71 71 66 71
```

"Egnethet og pålitelighet"

```
1  Eksperimentsett 1 : Individer
2
3  Egnethet ved test med inverteringsfeil
4   10 individer : -1.31169 -1.46053 -1.57333 -1.49333 -2.43590
       -2.42105 -1.78667 -1.70270 -1.28571 -1.62821
5   15 individer : -1.51351 -1.58108 -1.22078 -1.34667 -1.76000
       -2.37500 -2.40506 -2.37975 -1.54430 -1.81579
6   20 individer : -1.77922 -1.73684 -1.89041 -1.63158 -1.53165
       -1.67500 -1.85526 -1.34667 -1.95522 -1.58750
7   25 individer : -2.44928 -1.81944 -1.58974 -1.55844 -2.37838
       -1.69737 -1.58974 -2.36842 -1.65385 -2.42647
8   30 individer : -1.67949 -1.37975 -1.80769 -2.35135 -1.68354
       -1.78667 -1.27848 -1.64000 -1.72727 -1.70000
9
10 Egnethet ved test uten inverteringsfeil
11  10 individer :  0  0  0  0 -1 -1  0  0  0  0
12  15 individer :  0  0  0  0  0 -1 -1 -1  0  0
13  20 individer :  0  0  0  0  0  0  0  0  0  0
14  25 individer : -1  0  0  0 -1  0  0 -1  0 -1
15  30 individer :  0  0  0 -1  0  0  0  0  0  0
16
17 Plitelighet ved test med inverteringsfeil
18  10 individer : 0.805195 0.815789 0.733333 0.720000 0.000000
       0.000000 0.733333 0.851351 0.818182 0.794872
19  15 individer : 0.783784 0.743243 0.870130 0.813333 0.773333
       0.000000 0.000000 0.000000 0.784810 0.763158
20  20 individer : 0.779221 0.763158 0.808219 0.815789 0.772152
       0.750000 0.776316 0.786667 0.761194 0.837500
21  25 individer : 0.000000 0.777778 0.782051 0.857143 0.000000
       0.736842 0.756410 0.000000 0.717949 0.000000
22  30 individer : 0.820513 0.797468 0.794872 0.000000 0.721519
       0.800000 0.772152 0.773333 0.753247 0.825000
23
24 Plitelighet ved test uten inverteringsfeil
25  10 individer : 1 1 1 1 0 0 1 1 1 1
26  15 individer : 1 1 1 1 1 0 0 0 1 1
27  20 individer : 1 1 1 1 1 1 1 1 1 1
28  25 individer : 0 1 1 1 0 1 1 0 1 0
29  30 individer : 1 1 1 0 1 1 1 1 1 1
30
31
32 Eksperimentsett 2 : Blokker
33
34 Egnethet ved test med inverteringsfeil
35  20 blokker  : -7.00 -6.20 -6.55 -5.95 -7.05 -6.20 -6.55 -6.75 -6.30
       -7.30
36  40 blokker  : -3.600 -2.675 -3.550 -3.150 -3.775 -2.375 -3.325
       -2.625 -2.775 -3.550
37  60 blokker  : -1.68333 -2.36667 -2.83333 -2.92982 -1.55932 -2.80000
       -2.00000 -1.81667 -2.14286 -2.08621
```

```
38  80 blokker  : -1.77922 -1.73684 -1.89041 -1.63158 -1.53165 -1.67500
        -1.85526 -1.34667 -1.95522 -1.58750
39  100 blokker : -1.49438 -1.31250 -1.34021 -1.48421 -1.32979 -1.45833
        -1.51765 -1.57955 -1.42857 -1.09184
40  120 blokker : -1.247620 -1.000000 -1.274730 -1.220180 -1.392860
       -0.862745 -1.189190 -1.266050 -1.109090 -1.000000
41
42 Egnethet ved test uten inverteringsfeil
43  20 blokker  : -4 -1 -2 -1 -1 -1 -1 -6 -1 -3
44  40 blokker  : -1  0 -1  0 -1  0  0  0  0  0
45  60 blokker  :  0  0 -1 -1  0 -1  0  0  0  0
46  80 blokker  :  0  0  0  0  0  0  0  0  0  0
47  100 blokker :  0  0  0  0  0  0  0  0  0  0
48  120 blokker :  0  0  0  0  0  0  0  0  0  0
49
50 Plitelighet ved test med inverteringsfeil
51  20 blokker  : 0.00 0.00 0.00 0.05 0.00 0.00 0.00 0.00 0.00 0.00
52  40 blokker  : 0.000 0.550 0.000 0.475 0.000 0.650 0.500 0.600 0.475
        0.625
53  60 blokker  : 0.700000 0.766667 0.000000 0.000000 0.796610 0.000000
        0.666667 0.733333 0.625000 0.620690
54  80 blokker  : 0.779221 0.763158 0.808219 0.815789 0.772152 0.750000
        0.776316 0.786667 0.761194 0.837500
55  100 blokker : 0.775281 0.885417 0.835052 0.800000 0.776596 0.843750
        0.764706 0.750000 0.846939 0.846939
56  120 blokker : 0.857143 0.826087 0.813187 0.825688 0.848214 0.872549
        0.864865 0.844037 0.827273 0.842593
57
58 Plitelighet ved test uten inverteringsfeil
59  20 blokker  : 0 0 0 0 0 0 0 0 0 0
60  40 blokker  : 0 1 0 1 0 1 1 1 1 1
61  60 blokker  : 1 1 0 0 1 0 1 1 1 1
62  80 blokker  : 1 1 1 1 1 1 1 1 1 1
63  100 blokker : 1 1 1 1 1 1 1 1 1 1
64  120 blokker : 1 1 1 1 1 1 1 1 1 1
65
66
67 Eksperimentsett 3 : Krysning
68
69 Egnethet ved test med inverteringsfeil
70  20% krysningsrate : -1.63636 -1.64865 -1.57500 -1.70886 -1.57895
        -1.67089 -1.66234 -1.70667 -1.79487 -2.45946
71  30% krysningsrate : -1.77922 -1.73684 -1.89041 -1.63158 -1.53165
        -1.67500 -1.85526 -1.34667 -1.95522 -1.58750
72  40% krysningsrate : -1.37975 -1.56164 -1.79221 -2.41026 -1.67500
        -1.50000 -1.31250 -1.28767 -1.71622 -1.60526
73  50% krysningsrate : -1.63291 -1.83333 -1.74026 -1.55696 -2.35897
        -1.46154 -1.58333 -2.40260 -1.22500 -2.32468
74
75 Egnethet ved test uten inverteringsfeil
76  20% krysningsrate : 0  0  0  0  0  0  0  0  0 -1
77  30% krysningsrate : 0  0  0  0  0  0  0  0  0  0
```

```
 40% krysningsrate : 0  0  0 -1  0  0  0  0  0  0
 50% krysningsrate : 0  0  0  0 -1  0  0 -1  0 -1

Plitelighet ved test med inverteringsfeil
 20% krysningsrate : 0.766234 0.743243 0.837500 0.721519 0.710526
     0.797468 0.831169 0.840000 0.794872 0.000000
 30% krysningsrate : 0.779221 0.763158 0.808219 0.815789 0.772152
     0.750000 0.776316 0.786667 0.761194 0.837500
 40% krysningsrate : 0.759494 0.767123 0.766234 0.000000 0.725000
     0.756410 0.800000 0.849315 0.756757 0.750000
 50% krysningsrate : 0.759494 0.694444 0.688312 0.848101 0.000000
     0.756410 0.763889 0.000000 0.812500 0.000000

Plitelighet ved test uten inverteringsfeil
 20% krysningsrate : 1 1 1 1 1 1 1 1 1 0
 30% krysningsrate : 1 1 1 1 1 1 1 1 1 1
 40% krysningsrate : 1 1 1 0 1 1 1 1 1 1
 50% krysningsrate : 1 1 1 1 0 1 1 0 1 0


Eksperimentsett 4 : Mutasjon

Egnethet ved test med inverteringsfeil
 1% mutasjonsrate : -1.86667 -1.56410 -1.60759 -1.65823 -1.87179
     -2.38961 -1.39744 -1.39241
 3% mutasjonsrate : -1.52632 -1.39744 -1.40789 -1.73418 -1.75949
     -1.94937 -1.63158 -1.48052
 5% mutasjonsrate : -1.77922 -1.73684 -1.89041 -1.63158 -1.53165
     -1.67500 -1.85526 -1.34667 -1.95522 -1.58750
 7% mutasjonsrate : -1.82051 -2.38667 -1.68657 -1.82051 -1.57333
     -1.87013 -2.55556 -1.53165
 9% mutasjonsrate : -1.79221 -1.75676 -1.89706 -1.73684 -1.76623
     -1.91781 -1.85915 -1.84000

Egnethet ved test uten inverteringsfeil
 1% mutasjonsrate : 0  0  0  0  0 -1  0  0
 3% mutasjonsrate : 0  0  0  0  0  0  0  0
 5% mutasjonsrate : 0  0  0  0  0  0  0  0  0  0
 7% mutasjonsrate : 0 -1  0  0  0  0 -1  0
 9% mutasjonsrate : 0  0  0  0  0  0  0  0

Plitelighet ved test med inverteringsfeil
 1% mutasjonsrate : 0.800000 0.743590 0.797468 0.772152 0.743590
     0.000000 0.782051 0.822785
 3% mutasjonsrate : 0.802632 0.769231 0.802632 0.784810 0.784810
     0.810127 0.750000 0.714286
 5% mutasjonsrate : 0.779221 0.763158 0.808219 0.815789 0.772152
     0.750000 0.776316 0.786667 0.761194 0.837500
 7% mutasjonsrate : 0.769231 0.000000 0.805970 0.807692 0.733333
     0.844156 0.000000 0.772152
 9% mutasjonsrate : 0.805195 0.770270 0.705882 0.776316 0.818182
     0.726027 0.718310 0.813333
```

```
116
117 Plitelighet ved test uten inverteringsfeil
118  1% mutasjonsrate : 1 1 1 1 1 0 1 1
119  3% mutasjonsrate : 1 1 1 1 1 1 1 1
120  5% mutasjonsrate : 1 1 1 1 1 1 1 1 1 1
121  7% mutasjonsrate : 1 0 1 1 1 1 0 1
122  9% mutasjonsrate : 1 1 1 1 1 1 1 1
```

"individual.cpp"

```
1  #include "stdafx.h"
2  #include <string.h>
3
4  inline static float _fabs(float f){ return f<0?-f:f; }
5
6  using namespace std;
7
8  extern unsigned int debug;
9
10 // The created individual is not initialized, so
11 // you must call either init() or operator=() before it is used.
12
13 Individual::Individual():
14 blocks(0),
15 unitCount(0)
16 {}
17
18 Individual::~Individual()
19 {
20   delete[] blocks;
21 }
22
23 void Individual::operator=(const Individual& i)
24 // Make a copy of an individual
25 {
26   if(blocks)
27   // slette eventuelle FU
28     delete[] blocks;
29
30   // kopier antall FU, fitness og alle FUer
31   unitCount=i.unitCount;
32   fitness=i.fitness;
33   reliability=i.reliability;
34   usedCount=i.usedCount;
35   blocks=new FU[unitCount];
36   for(unsigned int a=0;a<unitCount;++a)
37     blocks[a]=i.blocks[a];
38 }
39
40 void Individual::cross(const Individual& mate, unsigned int
       crosspoint)
41 {
42   for(unsigned int a=crosspoint;a<unitCount;a++)
43     blocks[a]=mate.blocks[a];
44 }
45
46 void Individual::init(unsigned int count, int inCount)  //
       Initialize individual, allocate memory
47 {
48   unitCount=count;
```

```
49    blocks=new FU[count];
50
51    for(unsigned int a=0;a<count;++a)
52      blocks[a].setPos(a+inCount);
53  }
54
55  void Individual::randomize()
56  // Randomize every FU
57  {
58    for(unsigned int a=0;a<unitCount;++a)
59      blocks[a].randomize();
60  }
61
62
63  void Individual::evaluate(Parameters *params, float* data, bool*
        used, int* usedbuf, bool* failed, bool* evaluated, int arraySize)
64  {
65    //real gatecount does not count vcc/gnd as used
66    unsigned int a, b, run, test, real_gatecount;
67    int in0, in1, top, top_short;
68
69    float fitness_tmp;
70
71    // Data: Holds all block output values, starting with the inputs.
          In this array
72    //    blockdata points to the first gate after the inputs and
73    //    output points to the first of the output gates
74
75    // Used: An array of bools corresponding to the data array, with
          the value T/F whether that
76    //    gate is used. Again, blockused is a shortcut to the first
          gate after the inputs.
77
78    // Usedbuf: An array of INT, where each specifies the number of a
          used gate, thus indexing
79    //    the used array. E.g. usedbuf[0]==19 means blockused[19] is a
           used gate.
80
81    // Failed: An array of bools, corresponding to the numbering in
          usedbuf. E.g. if failed[18]==true
82    //    means gate 18 failed, e.g. blockused[19] or blockdata[19]
83
84    // Number of units+inputs (length of all arrays)
85    unsigned int dataLength=params->inputCount+unitCount;
86    // Shortcut to the output of the FUs
87    float* blockdata=&data[params->inputCount];
88    // Shortcut to the output of the individual
89    float* output=&blockdata[unitCount-params->outputCount];
90    // Shortcut to the used-state of the FUs
91    bool* blockused=&used[params->inputCount];
92    stack<int> Stack;
93
```

```
94    // Reset fitness value
95    fitness=0;
96    fitness_tmp=0;
97    reliability=0;
98
99    // Set all FUs as not used
100   for(a=0;a<dataLength;++a)
101     used[a]=false;
102
103   // Set output FUs as used - they are always used
104   for(a=1;a<=params->outputCount;++a) {
105     blockused[unitCount-a]=true;
106     //cout << "blockused[" << unitCount-a << "] is true\n";
107     Stack.push(unitCount-a);
108   }
109
110   while( !Stack.empty() ) {    // new stack-loop for detecting used
        gates in non-CGP networks
111     blocks[Stack.top()].setUsed(used,in0,in1);
112     in0-=params->inputCount;
113     in1-=params->inputCount;
114     Stack.pop();
115     if (in0>=0)
116       Stack.push(in0);
117     if (in1>=0)
118       Stack.push(in1);
119   }
120
121   usedCount=0;
122   real_gatecount=0;
123
124   // For each FU, update usedbuf and usedCount after stack-loop
125   for(a=unitCount-1;a<unitCount;--a){
126     if(blockused[a]){
127       usedbuf[usedCount]=a;
128       usedCount++;
129       if(!blocks[a].isVCCGND())
130         real_gatecount++;
131     }
132   }
133
134
135   // OLD used-setting loop (only stricly feedforward CGP networks)
136   /*  usedCount=0;
137
138   // For each FU: if it is used set its inputs as used and increase
        usecount
139   for(a=unitCount-1;a<unitCount;--a){
140     // Continue backwards through the network
141     if(blockused[a]){
142       // The unsigned value of a wraps around to end the FOR loop
143       blocks[a].setUsed(used,in0,in1);
```

```
144         cout << endl << "block " << a << " in0:" << in0 << " in1:" <<
                in1;
145         usedbuf[usedCount]=a;
146         ++usedCount;
147       }
148       }*/
149
150
151
152   //*****************************************************
153   //  Start of Stuck-at kode
154   //*****************************************************
155     unsigned int temp = params->number_of_ssa;
156
157
158
159     if (params->single_stuck_at){
160       // if the number of stuck at point is bigger den the
161       // avaleble blocks used, sett it to stuck at all (=0)
162       if (params->number_of_ssa >= usedCount || (params->
              output_protect && params->number_of_ssa >= usedCount-params->
              outputCount))
163         temp = 0;
164
165       if (temp)
166         // For a given number of stuck at points
167         params->error_runs = params->number_of_ssa;
168       else if (params->output_protect)
169         // Stuck at all, with output protection
170         params->error_runs = usedCount - params->outputCount;
171       else
172         // Stuck at all, without output protection
173         params->error_runs = usedCount;
174     }//end if
175
176     int* stuck_at = new int[params->error_runs];
177
178     for (int t = 0 ; t < params->error_runs; t++)
179       stuck_at[t] = -1;
180
181     // Generate whitch gates to stick at
182     if (temp){
183       unsigned int i = 0;
184       unsigned int j = 0;
185       unsigned int number;
186
187       // while the array is not full
188       while(i < temp){
189         if (params->output_protect)
190           // generates a number between outputCount and usedCount
191           number = ((rand()%(usedCount-params->outputCount))+params->
                  outputCount);
```

```
192         else
193           // generates a number between 0 and usedCount
194           number = rand()%usedCount;
195
196         j = 0;
197
198         //checks the array to see if this block is added already
199         while(stuck_at[j] != -1){
200           if (stuck_at[j] == usedbuf[number])
201             break;
202           else
203             j++;
204         }//end while
205         // if it is'nt in the array
206         if (stuck_at[j] == -1){
207           stuck_at[j] = usedbuf[number];// add it
208           i++;
209         }//end if
210       }//end while
211     } else
212       for (int w = 0; w < params->error_runs; w++)
213         stuck_at[w] = usedbuf[w];
214
215   //*****************************************************
216   //  End of Stuck-at preparation
217   //*****************************************************
218
219
220     // For each error configuration
221     for(run=0;run<params->error_runs;++run){
222
223       for(a=0;a<unitCount;a++)
224         // initially none fail
225         failed[a] = false;
226
227       // start checking for fail at first gate in usedbuf
228       unsigned int failstart=0;
229       if (params->output_protect)
230         // start after output count if output gates are protected
231         failstart=params->outputCount;
232
233       if (params->failrate) {
234         // check for failures
235         for(a=failstart;a<usedCount;a++)
236           if ( (rand()*(1.0f/RAND_MAX)) > params->failrate)
237             failed[usedbuf[a]]=false;
238           else
239             failed[usedbuf[a]]=true;
240       } else if (params->single_stuck_at){
241         // removes the faults for last run
242         for(int t = 0; t < arraySize;t++)
243           failed[t] = false;
```

```
244
245         failed[stuck_at[run]]=true;
246
247       }//end if
248
249       for(test=0; test < params->error_tests; ++test){  // For each
              test run
250         for(a=0;a<params->dataCount;++a){      // For each test vector
251           for(b=0;b<params->inputCount;++b)      // Copy inputs into
                  data array
252             data[b]=params->inputs[a][b];
253
254           // The inputs values are allready set / evaluated
255           for(b=0;b<params->inputCount;b++)
256             evaluated[b]=true;
257
258           // Others are not yet evaluated
259           for(b=params->inputCount;b<dataLength;b++)
260             evaluated[b]=false;
261
262           // Prepare outputs for evaluation and parse network
263           for(b=1;b<=params->outputCount;++b)
264             // Use full index (not shortcut) on stack
265             Stack.push(unitCount-b+params->inputCount);
266
267           // LOOP FOR EVALUATION (supports non-strict feed forward)
268
269           while( !Stack.empty() ) {
270
271             top = Stack.top();
272             top_short = Stack.top()-params->inputCount;
273
274             // if this FU has allready been eval'ed, pop it
275             if( evaluated[top] )
276         Stack.pop();
277
278             else {
279         in0=blocks[top_short].getInput(0);
280         in1=blocks[top_short].getInput(1);
281
282               if(in0 == -1) { // *** zero inputs ***
283           blockdata[top_short]=blocks[top_short].evaluate(data, params
                  , failed[top_short]); // eval'
284           evaluated[top]=true;
285             Stack.pop();
286         } else if(in1 == -1) // *** single input ***
287           if(evaluated[in0]){ // if that input is eval'ed
288             blockdata[top_short]=blocks[top_short].evaluate(data,
                    params, failed[top_short]); // eval'
289             evaluated[top]=true;
290             Stack.pop();
291           } else
```

```
292             Stack.push(in0); // else push it
293
294         // *** two inputs ***
295         else {
296
297           // if first input not eval'ed
298           if(!evaluated[in0])
299             Stack.push(in0);   // push it
300
301           // if second input not eval'ed
302           if(!evaluated[in1])
303             Stack.push(in1);   // push it
304
305           else if((evaluated[in0])&&(evaluated[in1])) {
306             blockdata[top_short]=blocks[top_short].evaluate(data,
                    params, failed[top_short]); // eval'
307             evaluated[top]=true;
308             Stack.pop();
309
310           }//end if
311         }//end if
312             }//end if
313           }//end while
314
315           // Compute fitness as negative sum of deviation from target
316           for(b=0;b<params->outputCount;++b) {
317             fitness_tmp-=_fabs(params->outputs[a][b]- (output[b]>=0.5f
                    ?1.0f:0.0f));
318
319             if (debug & DBG_EVALUATION){
320               cout << "output [" << b << "] is " << output[b];
321               cout << ", correct is " << params->outputs[a][b] << endl
                        ;
322             }//end if
323           }//end for
324         } // end of test vector
325
326         if (fitness_tmp==0)
327           // if all vectors correct (correct circuit), increase
                realiability
328           reliability++;
329
330         fitness+=fitness_tmp;
331         fitness_tmp=0;
332       } //end of test run
333
334     } // end of error configuration
335
336     // Average fitness
337     fitness/=(params->error_runs*params->error_tests);
338
339     // Average reliability
```

```
340     reliability/=(params->error_runs*params->error_tests);
341
342     // fitness_method==1 calculates fitness as realiability (1->good,
            0->faulty)
343     if (params->fitness_method == 1)
344       fitness = reliability;
345
346     // START OF OPTIONAL NONFAULTED TEST
347     if (params->fitness_method == 2) {  // combine truthtable fitness
            and reliability
348     // give dominating score to those whose unfaulted function is 100%
            correct
349       fitness_tmp=0;
350
351       for(a=0;a<unitCount;a++)
352       // initially none fail
353       failed[a] = false;
354
355       // For each test vector
356       for(a=0;a<params->dataCount;++a){
357         // Copy inputs into data array
358         for(b=0;b<params->inputCount;++b)
359           data[b]=params->inputs[a][b];
360
361         // The inputs values are allready set / evaluated
362         for(b=0;b<params->inputCount;b++)
363           evaluated[b]=true;
364
365         // Others are not yet evaluated
366         for(b=params->inputCount;b<dataLength;b++)
367           evaluated[b]=false;
368
369         // Prepare outputs for evaluation and parse network
370         for(b=1;b<=params->outputCount;++b)
371           // Use full index (not shortcut) on stack
372           Stack.push(unitCount-b+params->inputCount);
373
374         while( !Stack.empty() ) {
375           top = Stack.top();
376           top_short = Stack.top()-params->inputCount;
377
378           // if this FU has allready been eval'ed, pop it
379           if( evaluated[top] )
380             Stack.pop();
381           else {
382             in0=blocks[top_short].getInput(0);
383             in1=blocks[top_short].getInput(1);
384                   if(in0 == -1) { // *** zero inputs ***
385               blockdata[top_short]=blocks[top_short].evaluate(data,
                      params, failed[top_short]); // eval'
386         evaluated[top]=true;
387         Stack.pop();
```

```
388             } else if(in1 == -1) // *** single input ***
389
390         // if that input is eval'ed
391         if(evaluated[in0]){
392           blockdata[top_short]=blocks[top_short].evaluate(data, params
                , failed[top_short]); // eval'
393           evaluated[top]=true;
394           Stack.pop();
395         } else
396           Stack.push(in0); // else push it
397             else {                // *** two inputs **
398
399         // if first input not eval'e
400         if(!evaluated[in0])
401           Stack.push(in0);    // push it
402
403         // if second input not eval'ed
404         if(!evaluated[in1])
405           Stack.push(in1);    // push it
406
407         else if((evaluated[in0])&&(evaluated[in1])) {
408           blockdata[top_short]=blocks[top_short].evaluate(data, params
                , failed[top_short]); // eval'
409           evaluated[top]=true;
410           Stack.pop();
411         }//end if
412             }//end if
413           }//end if
414         }//end while
415
416         // Compute fitness as negative sum of deviation from target
417         for(b=0;b<params->outputCount;++b) {
418           fitness_tmp-=fabs(params->outputs[a][b]- (output[b]>=0.5f
                ?1.0f:0.0f));
419           if (debug & DBG_EVALUATION)
420             cout << "output [" << b << "] is " << output[b] << ",
                  correct is " << params->outputs[a][b] << endl;
421         }
422       } // end of test vector
423
424       // if unfaulted fitness is MAX
425       if (fitness_tmp==(params->dataCount*params->outputCount))
426         fitness+=fitness_tmp;  // add MAX to faulted fitness
427
428     } // END OF EXTRA OPTIONAL NONFAULTED TEST
429
430
431
432     //if(fitness>-1)
433     //  fitness-=params->sizeFactor*usedCount;  // Add size value
434
435     if (!params->count_vcc_gnd)
```

```
436       usedCount=real_gatecount;
437
438     delete[] stuck_at;
439   }
440
441
442   // Mutates FUs according to the given mutation rate
443   void Individual::mutate(float mutationrate)
444   {
445     // new method, check prob. with every gate (slower)
446     for (int a=0;a<unitCount;a++)
447       if ((rand()*(1.0f/RAND_MAX))<mutationrate)
448         blocks[a].mutate();
449   }
450
451
452   std::ostream& operator<<(std::ostream &os, const Individual& i)
453   {
454     os << "Individual{ size: " << i.unitCount;
455     os << " used: " << i.usedCount;
456     os << " fitness: " << i.fitness;
457     os << " reliability: " << i.reliability << " blocks: ";
458
459     for(unsigned int a=0;a<i.unitCount;++a){
460       os << i.blocks[a];
461     }
462
463     os << " }";
464
465     return os;
466   }
467
468   std::istream& operator>>(std::istream &is, Individual& i)
469   {
470     std::string stmp;
471     int itmp;
472     char ctmp;
473
474     is >> stmp;
475     if(stmp!="Individual{")
476       FAULT_END("Individual Read error (Individual{)");
477
478     is >> stmp;
479     if(stmp!="size:")
480       FAULT_END("Individual Read error (size:)");
481
482
483     is >> itmp;
484     i.init(itmp, 0);
485
486     is >> stmp;
487     if(stmp!="used:")
```

```
488       FAULT_END("Individual Read error (used:)");
489
490     is >> i.usedCount;
491
492     is >> stmp;
493     if(stmp!="fitness:")
494       FAULT_END("Individual Read error (fitness:)");
495
496     is >> i.fitness;
497
498     is >> stmp;
499     if(stmp!="reliability:")
500       FAULT_END("Individual Read error (reliability:)");
501
502     is >> i.reliability;
503
504     is >> stmp;
505     if(stmp!="blocks:")
506       FAULT_END("Individual Read error (blocks:)");
507
508     for(unsigned int a=0;a<i.unitCount;++a)
509       is >> i.blocks[a];
510
511     is >> ctmp;
512     if(ctmp!='}')
513       FAULT_END("Individual Read error (})");
514
515     return is;
516   }
```

"extract.cpp"

```
1
2  #include <fstream>
3  #include <string>
4  #include <memory>
5  #include <iostream>
6
7  #include <stdio.h>
8  #include <time.h>
9  #include <stdlib.h>
10 #include <search.h>
11 #include <math.h>
12 #include <unistd.h>
13
14 using namespace std;
15
16 float noise, fail, fitness, reliability, t_fitness, t_reliability,
       t_noise, t_fail, crossoverrate, mutationrate, algorithm, selprob;
17 unsigned int used, t_used, groupsize, gate_count, individuals,
       elitism, fitness_method, fail_type, output_protect,
       t_fitness_method, t_fail_type, t_output_protect, report_gap;
18 unsigned long totgen, maxgen, seed, t_seed,error_runs,error_tests,
       t_error_runs,t_error_tests;
19 unsigned int mode = 0;
20 unsigned int i = 0;
21 string batch, type, individual, id;
22
23 void readFile(char* file)
24 {
25   ifstream in(file);
26
27   string tmp;
28   in >> tmp;
29   while(!in.eof()){
30     if(tmp[0]=='#'){//a comment
31       in.ignore(10000, '\n');//ignore the rest of the line
32     }else if(tmp=="failrate:"){
33       in >> fail;
34     }else if(tmp=="best_fitness:"){
35       in >> fitness;
36     }else if(tmp=="reliability:"){
37       in >> reliability;
38     }else if(tmp=="fitness_method:"){
39       in >> fitness_method;
40     }else if(tmp=="fail_type:"){
41       in >> fail_type;
42     }else if(tmp=="output_protect:"){
43       in >> output_protect;
44     }else if(tmp=="noise:"){
45       in >> noise;
46     }else if(tmp=="used_gates:"){
```

```
47        in >> used;
48      }else if(tmp=="groupsize:"){
49        in >> groupsize;
50      }else if(tmp=="elitism:"){
51        in >> elitism;
52      }else if(tmp=="error_runs:"){
53        in >> error_runs;
54      }else if(tmp=="error_tests:"){
55        in >> error_tests;
56      }else if(tmp=="t_error_tests:"){
57        in >> t_error_tests;
58      }else if(tmp=="t_error_runs:"){
59        in >> t_error_runs;
60      }else if(tmp=="individuals:"){
61        in >> individuals;
62      }else if(tmp=="crossoverrate:"){
63        in >> crossoverrate;
64      }else if(tmp=="mutationrate:"){
65        in >> mutationrate;
66      }else if(tmp=="algorithm:"){
67        in >> algorithm;
68      }else if(tmp=="sel.prob:"){
69        in >> selprob;
70      }else if(tmp=="gate_count:"){
71        in >> gate_count;
72      }else if(tmp=="seed:"){
73        in >> seed;
74      }else if(tmp=="t_seed:"){
75        in >> t_seed;
76      }else if(tmp=="t_noise:"){
77        in >> t_noise;
78      }else if(tmp=="t_failrate:"){
79        in >> t_fail;
80      }else if(tmp=="t_used_gates:"){
81        in >> t_used;
82      }else if(tmp=="generations:"){
83        in >> maxgen;
84      }else if(tmp=="total_gens:"){
85        in >> totgen;
86      }else if(tmp=="circuit_type:"){
87        in >> type;
88      }else if(tmp=="t_best_fitness:"){
89        in >> t_fitness;
90      }else if(tmp=="t_reliability:"){
91        in >> t_reliability;
92      }else if(tmp=="t_fitness_method:"){
93        in >> fitness_method;
94      }else if(tmp=="t_fail_type:"){
95        in >> fail_type;
96      }else if(tmp=="t_output_protect:"){
97        in >> output_protect;
98      }else if(tmp=="report_gap:"){
```

```
99        in >> report_gap;
100     }else if(tmp=="best_individual:"){
101       in >> tmp;
102       while (tmp != "}") {
103         individual = individual + " " +tmp;
104         in >> tmp;
105       }
106       individual = individual + " }";
107     }else{
108       //cerr << "Error while extracting: unknown value '" << tmp <<
              "'!\n";
109       in.ignore(10000, '\n');//ignore the rest of the line
110     }
111     in >> tmp;;
112   }
113
114 }
115
116 void readFitness(char* file, float* fitnessMat){
117
118   ifstream in(file);
119
120   string tmp;
121   in >> tmp;
122   while(!in.eof()){
123     if(tmp[0]=='#'){//a comment
124       in.ignore(10000, '\n');//ignore the rest of the line
125     }else if(tmp=="fitness:"){
126       in >> fitnessMat[i];
127       i++;
128     }else if(tmp=="Best"){
129
130     }else{
131       //cerr << "Error while extracting: unknown value '" << tmp <<
              "'!\n";
132       in.ignore(10000, '\n');//ignore the rest of the line
133     }
134     in >> tmp;;
135   }
136
137 }
138
139 int main(int argc, char* argv[])
140 {
141
142   if((argc!=1)&&(argc<2)){
143     cout << argc <<"\n";
144     cout << "Usage 1: " << argv[0] << " [file_to_extract] [mode]\n";
145     cout << "       2: " << "OR\n";
146     cout << "       3: " << argv[0] << " [mode]\n\n";
147     cout << "          " << "OR\n";
148     cout << "          " << argv[0] << "\n\n";
```

```
149     cout << "(mode)___" << "_0_extract_all_info(only_usage_1)_\n";
150     cout << "__________" << "_1_extract_only_individual(only_usage_1)
            \n";
151     cout << "__________" << "_2_show_datafields(default)\n\n";
152                 cout << "__________" << "_3_show_datafields_
                        generation|fitness\n\n";
153     cout << "__________" << "_4_extract_all_fitness_values_with_
            generation,_from_the";
154     cout <<           "_evolution(only_usage_1)\n\n";
155                 cout << "__________" << "_5_extract_every_20eth_
                        fitness_values_with_generation_from_the";
156     cout <<           "_evolution(only_usage_1)\n\n";
157                 cout << "__________" << "_6_extract_first_31_fitness_
                        values_with_generation_from_the";
158     cout <<           "_evolution(only_usage_1)\n\n";
159     return 1;
160   }//end if
161
162   mode = 2;
163
164   float fitnessMat[1000];
165
166   if(argc>2){
167     readFile(argv[1]);     // les inn fra fil (argv[1])
168     mode = atoi(argv[2]);
169     readFitness(argv[1], fitnessMat);
170   }//end if
171
172   if(argc == 2)
173     mode = atoi(argv[1]);
174
175   char buffer[250];
176
177   /* Get the current working directory: */
178   if( getcwd( buffer, 250) != NULL )
179     batch = buffer;
180
181
182   switch (mode) {
183     case 1: cout << "\nindividual"<< individual << endl; break;
184
185     case 2: cout << "type|noise|fail|error_tests|error_runs|totgen|
            fitness|reliability|";
186       cout << "t_fitness|t_reliability|gate_count|used|batch|id|seed
              |individuals|";
187       cout << "max_gen|algorithm|fitness_method|fail_type|
              output_protect|elitism|";
188       cout << "group_size|mutationrate|crossoverrate|t_noise|t_fail|
              t_used|t_seed|";
189       cout << "t_error_tests|t_error_runs|t_fitness_method|
              t_fail_type|";
190       cout << "t_output_protect|individual\n"; break;
```

```
191
192     case 3: cout << "generation\tfitness\n"; break;
193
194     // Skriver ut alle aalepunktenn(1000 stk)
195     case 4: for(i=0;i<1000;i += 1)
196         cout << (i*report_gap) << "\t" << fitnessMat[i] << endl;
197       break;
198
199     // Skriver ut hvert 20 aalepunkt(50 stk, hver 10000. generasjon
200     case 5: for(i=0;i<1000;i += 20)
201                                 cout << (i*report_gap) << "\t" <<
                                        fitnessMat[i] << endl;
202                         break;
203
204     // Skriver ut de orste 31 maalepunktene(tom gen 15000)
205     case 6: for(i=0;i<31;i += 1)
206                                 cout << (i*report_gap) << "\t" <<
                                        fitnessMat[i] << endl;
207                         break;
208
209
210     case 0:
211     default: cout << type << "|" << noise << "|" << fail << "|" <<
            error_tests << "|";
212         cout << error_runs << "|" << totgen << "|" << fitness << "|"
            << reliability;
213         cout << "|" << t_fitness << "|" << t_reliability << "|" <<
            gate_count << "|";
214         cout << used << "|" << batch << "|" << id << "|" << seed << "
            |" << individuals;
215         cout << "|" << maxgen << "|" << algorithm << "|" <<
            fitness_method << "|";
216         cout << fail_type << "|" << output_protect << "|" << elitism
            << "|" << groupsize;
217         cout << "|" << mutationrate << "|" << crossoverrate << "|" <<
             t_noise << "|";
218         cout << t_fail << "|" << t_used << "|" << t_seed << "|" <<
            t_error_tests << "|";
219         cout << t_error_runs << "|" << t_fitness_method << "|" <<
            t_fail_type << "|";
220         cout << t_output_protect << "|" << individual << endl; break;
221   }//end switch
222
223   return 0;
224 }//end main
```

"Parameterfil"

```
1  #example parameter file for 2-bit multiplier
2
3  #If testrun is 1, there is no evolution
4  #Instead, the one input individual is tested many times
5  testrun     0
6
7  #Debug level (default 0) (WARNING: can produce massive amounts of
       output data)
8  debug     0
9
10 #how many logic gates to use
11 gates     80
12
13 #max number of generations to run
14 generations    500000
15
16 #number of individuals per generation
17 individuals    20
18
19 #how many generations between each report
20 report_gap     500
21
22 #how the size of the circuit modifes the fitness score
23 #fitness_score = fitness_score + size_factor * gates_used
24 size_factor    0.00
25
26 #Stop if the fitness score reaches this value
27 fitness_limit 0.0
28
29 #Defines which genetic algorithm to use
30 #Currently defined algorithms are
31 # 1 MUTATE_FITTEST_1     - don't mutate the best idividual
32 # 2 MUTATE_FITTEST_2     - mutate all
33 # 3 KILL_HALF            - Clone best half, mutate to make second
       half
34 # 4 KOOZA                - 1% mutation, 9% clone, 90% crossover
35 # 5 TOURNAMENT
36 algorithm    5
37
38 #Parameters for tournament selection
39 #ga_groupsize      - GA Groupsize in Tournament Selection
40 #ga_elitism      - Number of elite individuals to clone to next
       generation (0=off) CURRENTLY ONLY 0 or 1
41 #ga_selectionprobability  - GA Tournament - Probability of selecting
        best in group
42 #ga_crossoverrate    - Probability of crossover
43 #ga_mutationrate     - Probability of mutation
44
45 ga_groupsize      5
46 ga_elitism       1
```

```
47 ga_selectionprobability    0.7
48 ga_crossoverrate     0.3
49 ga_mutationrate      0.05
50
51 #defines how much to bias towards the best individuals when
       selecting
52 # (currently only for algorithm 4)
53 #  bias > 1 favours the best individuals
54 #  bias = 1 does not favour any
55 #  0 < bias < 1 favours the worst individual - not very useful
56 #  Don't use bias <= 0 !
57 bias       3.0
58
59 #Defines the seed value of the random number generator
60 #If not specified, the system clock is used.
61 #seed      12345678
62
63 #Defines an individual to start from
64 #If not specified, a random generation is used
65 #individual Individual{ size: 10 fitness: -42 blocks: (4: 0 AND 1)
       (5: 2 AND 3)(6: 4 AND 4)(7: 4 AND 5)(8: 6 AND 6)(9: 6 AND 7)(10:
       NOT 8)(11: NOT 8)(12: GND )(13: VCC ) }
66
67 #to use single stuck at or not (0 = false, 1 = true)
68 single_stuck_at    1
69
70 #how many stuck at points to use (0 = all)
71 number_of_ssa    0
72
73 #how much noise to use
74 noise      0.00
75
76 #chance of logic gate failing
77 failrate     0.00
78
79 #minimum number of failed gates (not in use)
80 minfail      0
81
82 #maximum number of failed gates (not in use)
83 maxfail      1
84
85 #how many error configurations to test each individual on (inner
       loop, recalc only fault type) [SHOULD BE 1]
86 error_tests    1
87
88 #how many times to test each error configuration (outer loop, recalc
        gates to fail)
89 error_runs     10
90
91 #relative chance of input/output failing in a failed gate
92 rel_out_fail     1
93 rel_in_fail      0
```

```
 94 rel_both_fail   0
 95
 96 #relative chance of input A/B failing if inputs fail
 97 rel_ina_fail     1
 98 rel_inb_fail     1
 99 rel_in_both_fail 1
100
101 #type of failure in gates (0=original, 1=invert only)
102 fail_type  1
103
104 #whether to protect output gates from failures
105 output_protect 0
106
107 #method to use for fitness calculation (0=ordinary, 1=realiability
       measure)
108 fitness_method 0
109
110 #defines the goal
111 inputs      4
112 outputs     4
113 data_count    16
114 #data inputs  outputs
115 data  0000  0000
116 data  0001  0000
117 data  0010  0000
118 data  0011  0000
119 data  0100  0000
120 data  0101  0001
121 data  0110  0010
122 data  0111  0011
123 data  1000  0000
124 data  1001  0010
125 data  1010  0100
126 data  1011  0110
127 data  1100  0000
128 data  1101  0011
129 data  1110  0110
130 data  1111  1001
131
132 #Just a literary oneword string to identify the target circuit
133 #0=2b ADD, 1=2b MUL
134 circuit_type 1
```

"R-skript"

```
1
2  # Read the data from file
3  run10ind <- read.delim("ind10.results",sep="|",colClasses=c("NULL",
       NA,NA,"NULL","NULL",NA,NA,NA,NA,NA,"NULL",NA,"NULL","NULL","NULL"
       ,"NULL","NULL","NULL","NULL","NULL","NULL","NULL","NULL","
       NULL","NULL","NULL","NULL","NULL","NULL","NULL","NULL","
       NULL","NULL"))
4  run15ind <- read.delim("ind15.results",sep="|",colClasses=c("NULL",
       NA,NA,"NULL","NULL",NA,NA,NA,NA,NA,"NULL",NA,"NULL","NULL","NULL"
       ,"NULL","NULL","NULL","NULL","NULL","NULL","NULL","NULL","
       NULL","NULL","NULL","NULL","NULL","NULL","NULL","NULL","
       NULL","NULL"))
5  run20ind <- read.delim("ind20.results",sep="|",colClasses=c("NULL",
       NA,NA,"NULL","NULL",NA,NA,NA,NA,NA,"NULL",NA,"NULL","NULL","NULL"
       ,"NULL","NULL","NULL","NULL","NULL","NULL","NULL","NULL","
       NULL","NULL","NULL","NULL","NULL","NULL","NULL","NULL","
       NULL","NULL"))
6  run25ind <- read.delim("ind25.results",sep="|",colClasses=c("NULL",
       NA,NA,"NULL","NULL",NA,NA,NA,NA,NA,"NULL",NA,"NULL","NULL","NULL"
       ,"NULL","NULL","NULL","NULL","NULL","NULL","NULL","NULL","
       NULL","NULL","NULL","NULL","NULL","NULL","NULL","NULL","
       NULL","NULL"))
7  run30ind <- read.delim("ind30.results",sep="|",colClasses=c("NULL",
       NA,NA,"NULL","NULL",NA,NA,NA,NA,NA,"NULL",NA,"NULL","NULL","NULL"
       ,"NULL","NULL","NULL","NULL","NULL","NULL","NULL","NULL","
       NULL","NULL",NA,"NULL","NULL","NULL","NULL","NULL","NULL","NULL",
       "NULL"))
8
9
10 run2cross <- read.delim("cross2.results",sep="|",colClasses=c("NULL"
       ,NA,NA,"NULL","NULL",NA,NA,NA,NA,NA,"NULL",NA,"NULL","NULL","NULL
       ","NULL","NULL","NULL","NULL","NULL","NULL","NULL","NULL",
       "NULL","NULL",NA,"NULL","NULL","NULL","NULL","NULL","NULL","NULL"
       ,"NULL"))
11 run4cross <- read.delim("cross4.results",sep="|",colClasses=c("NULL"
       ,NA,NA,"NULL","NULL",NA,NA,NA,NA,NA,"NULL",NA,"NULL","NULL","NULL
       ","NULL","NULL","NULL","NULL","NULL","NULL","NULL","NULL",
       "NULL","NULL",NA,"NULL","NULL","NULL","NULL","NULL","NULL","NULL"
       ,"NULL"))
12 run5cross <- read.delim("cross5.results",sep="|",colClasses=c("NULL"
       ,NA,NA,"NULL","NULL",NA,NA,NA,NA,NA,"NULL",NA,"NULL","NULL","NULL
       ","NULL","NULL","NULL","NULL","NULL","NULL","NULL","NULL",
       "NULL","NULL",NA,"NULL","NULL","NULL","NULL","NULL","NULL","NULL"
       ,"NULL"))
13
14
15 run20blokk <- read.delim("bl20.results",sep="|",colClasses=c("NULL",
       NA,NA,"NULL","NULL",NA,NA,NA,NA,NA,"NULL",NA,"NULL","NULL","NULL"
       ,"NULL","NULL","NULL","NULL","NULL","NULL","NULL","NULL","
       NULL","NULL",NA,"NULL","NULL","NULL","NULL","NULL","NULL","NULL",
```

```
    "NULL"))
16 run40blokk <- read.delim("bl40.results",sep="|",colClasses=c("NULL",
    NA,NA,"NULL","NULL",NA,NA,NA,NA,NA,"NULL",NA,"NULL","NULL","NULL"
    ,"NULL","NULL","NULL","NULL","NULL","NULL","NULL","NULL","
    NULL","NULL",NA,"NULL","NULL","NULL","NULL","NULL","NULL","NULL",
    "NULL"))
17 run60blokk <- read.delim("bl60.results",sep="|",colClasses=c("NULL",
    NA,NA,"NULL","NULL",NA,NA,NA,NA,NA,"NULL",NA,"NULL","NULL","NULL"
    ,"NULL","NULL","NULL","NULL","NULL","NULL","NULL","NULL","
    NULL","NULL",NA,"NULL","NULL","NULL","NULL","NULL","NULL","NULL",
    "NULL"))
18 run100blokk <- read.delim("bl100.results",sep="|",colClasses=c("NULL
    ",NA,NA,"NULL","NULL",NA,NA,NA,NA,NA,"NULL",NA,"NULL","NULL","
    NULL","NULL","NULL","NULL","NULL","NULL","NULL","NULL","NULL","
    NULL","NULL","NULL",NA,"NULL","NULL","NULL","NULL","NULL","NULL",
    "NULL","NULL"))
19 run120blokk <- read.delim("bl120.results",sep="|",colClasses=c("NULL
    ",NA,NA,"NULL","NULL",NA,NA,NA,NA,NA,"NULL",NA,"NULL","NULL","
    NULL","NULL","NULL","NULL","NULL","NULL","NULL","NULL","NULL","
    NULL","NULL","NULL",NA,"NULL","NULL","NULL","NULL","NULL","NULL",
    "NULL","NULL"))
20
21
22 run01mut <- read.delim("mut01.results",sep="|",colClasses=c("NULL",
    NA,NA,"NULL","NULL",NA,NA,NA,NA,NA,"NULL",NA,"NULL","NULL","NULL"
    ,"NULL","NULL","NULL","NULL","NULL","NULL","NULL","NULL","
    NULL","NULL",NA,"NULL","NULL","NULL","NULL","NULL","NULL","NULL",
    "NULL"))
23 run03mut <- read.delim("mut03.results",sep="|",colClasses=c("NULL",
    NA,NA,"NULL","NULL",NA,NA,NA,NA,NA,"NULL",NA,"NULL","NULL","NULL"
    ,"NULL","NULL","NULL","NULL","NULL","NULL","NULL","NULL","
    NULL","NULL",NA,"NULL","NULL","NULL","NULL","NULL","NULL","NULL",
    "NULL"))
24 run07mut <- read.delim("mut07.results",sep="|",colClasses=c("NULL",
    NA,NA,"NULL","NULL",NA,NA,NA,NA,NA,"NULL",NA,"NULL","NULL","NULL"
    ,"NULL","NULL","NULL","NULL","NULL","NULL","NULL","NULL","
    NULL","NULL",NA,"NULL","NULL","NULL","NULL","NULL","NULL","NULL",
    "NULL"))
25 run09mut <- read.delim("mut09.results",sep="|",colClasses=c("NULL",
    NA,NA,"NULL","NULL",NA,NA,NA,NA,NA,"NULL",NA,"NULL","NULL","NULL"
    ,"NULL","NULL","NULL","NULL","NULL","NULL","NULL","NULL","
    NULL","NULL",NA,"NULL","NULL","NULL","NULL","NULL","NULL","NULL",
    "NULL"))
26
27 #Load functions and generate plots
28 source("my_functions.R")
29
30 ps.options(pointsize=16)
31
32 indFitness <- c(run10ind$fitness, run15ind$fitness,
33                 run20ind$fitness, run25ind$fitness,
34                 run30ind$fitness)
```

```
35 indTFitness <- c(run10ind$t_fitness, run15ind$t_fitness,
36                  run20ind$t_fitness, run25ind$t_fitness,
37                  run30ind$t_fitness)
38 indRel <- c(run10ind$reliability, run15ind$reliability,
39             run20ind$reliability, run25ind$reliability,
40             run30ind$reliability)
41
42 indTRel <- c(run10ind$t_reliability, run15ind$t_reliability,
43              run20ind$t_reliability, run25ind$t_reliability,
44              run30ind$t_reliability)
45
46 tmrR <-
47   c(0.72093, 0.72093, 0.72093, 0.72093, 0.72093, 0.72093, 0.72093,
       0.72093, 0.72093, 0.72093,
48     0.72093, 0.72093, 0.72093, 0.72093, 0.72093, 0.72093, 0.72093,
         0.72093, 0.72093, 0.72093,
49     0.72093, 0.72093, 0.72093, 0.72093, 0.72093, 0.72093, 0.72093,
         0.72093, 0.72093, 0.72093,
50     0.72093, 0.72093, 0.72093, 0.72093, 0.72093, 0.72093, 0.72093,
         0.72093, 0.72093, 0.72093,
51     0.72093, 0.72093, 0.72093, 0.72093, 0.72093, 0.72093, 0.72093,
         0.72093, 0.72093, 0.72093)
52
53 enkelR <- c(0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
54             0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
55             0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
56             0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
57             0, 0, 0, 0, 0, 0, 0, 0, 0, 0)
58
59
60 ind <- c(10, 10, 10, 10, 10, 10, 10, 10, 10, 10,
61          15, 15, 15, 15, 15, 15, 15, 15, 15, 15,
62          20, 20, 20, 20, 20, 20, 20, 20, 20, 20,
63          25, 25, 25, 25, 25, 25, 25, 25, 25, 25,
64          30, 30, 30, 30, 30, 30, 30, 30, 30, 30)
65
66 postscript(file="indFit.ps",onefile=F)
67 plot(indFitness, ind, xlab="Antall individer",ylab="Egnethet",col="
       blue",pch=1,xlim =c(10, 30), ylim=c(-2.5, 1),type="n")
68 avgline2(ind, indFitness,"blue",1)
69 maxpoints(ind, indFitness,"blue",1)
70 minpoints(ind, indFitness,"blue",1)
71 avgline2(ind, indTFitness,"orange",2)
72 maxpoints(ind, indTFitness,"orange",2)
73 minpoints(ind, indTFitness,"orange",2)
74 legend(10,1,c("Egnethet med inverteringsfeil","Egnethet uten
       inverteringsfeil"),pch=c(1,2),col=c("blue","orange"))
75 dev.off()
76
77 postscript(file="indRel.ps",onefile=F)
78 plot(indFitness, ind, xlab="Antall individer",ylab="Plitlighet",col=
       "blue",pch=1,xlim =c(10, 30), ylim=c(0, 1.5),type="n")
```

```
79  avgline2(ind, indRel,"red",1)
80  maxpoints(ind, indRel,"red",1)
81  minpoints(ind, indRel,"red",1)
82  avgline2(ind, indTRel,"green",2)
83  maxpoints(ind, indTRel,"green",2)
84  minpoints(ind, indTRel,"green",2)
85  avgline2(ind, tmrR,"turquoise",3)
86  maxpoints(ind, tmrR,"turquoise",3)
87  minpoints(ind, tmrR,"turquoise",3)
88  avgline2(ind, enkelR,"purple",4)
89  maxpoints(ind, enkelR,"purple",4)
90  minpoints(ind, enkelR,"purple",4)
91  legend(10,1.5,c("Plitlighet med inverteringsfeil","Plitlighet uten
        inverteringsfeil","Plitlighet av TMR-krets med inverteringsfeil",
        "Plitlighet av enkel multiplikator med inverteringsfeil"),pch=c
        (1,2,3,4),col=c("red","green","turquoise","purple"))
92  dev.off()
93
94  crossFit <- c(run2cross$fitness, run20ind$fitness,
95                run4cross$fitness, run5cross$fitness)
96  crossTFit <- c(run2cross$t_fitness, run20ind$t_fitness,
97                 run4cross$t_fitness, run5cross$t_fitness)
98  crossRel <- c(run2cross$reliability, run20ind$reliability,
99                run4cross$reliability, run5cross$reliability)
100 crossTRel <- c(run2cross$t_reliability, run20ind$t_reliability,
101                run4cross$t_reliability, run5cross$t_reliability)
102
103 tmrR <-
104   c(0.72093, 0.72093, 0.72093, 0.72093, 0.72093, 0.72093, 0.72093,
        0.72093, 0.72093, 0.72093,
105     0.72093, 0.72093, 0.72093, 0.72093, 0.72093, 0.72093, 0.72093,
          0.72093, 0.72093, 0.72093,
106     0.72093, 0.72093, 0.72093, 0.72093, 0.72093, 0.72093, 0.72093,
          0.72093, 0.72093, 0.72093,
107     0.72093, 0.72093, 0.72093, 0.72093, 0.72093, 0.72093, 0.72093,
          0.72093, 0.72093, 0.72093)
108
109 enkelR <- c(0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
110              0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
111              0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
112              0, 0, 0, 0, 0, 0, 0, 0, 0, 0)
113
114 cross <- c(0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2,
115            0.3, 0.3, 0.3, 0.3, 0.3, 0.3, 0.3, 0.3, 0.3, 0.3,
116            0.4, 0.4, 0.4, 0.4, 0.4, 0.4, 0.4, 0.4, 0.4, 0.4,
117            0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5)
118
119 postscript(file="crossFit.ps",onefile=F)
120 plot(crossFit,cross,xlab="Sannsynlighet for krysning",ylab="Egnethet
        ",col="blue",pch=1,xlim=c(0.2,0.5), ylim=c(-2.5,1),type="n")
121 avgline2(cross,crossFit,"blue",1)
122 maxpoints(cross,crossFit,"blue",1)
```

```
123 minpoints(cross,crossFit,"blue",1)
124 avgline2(cross,crossTFit,"orange",2)
125 maxpoints(cross,crossTFit,"orange",2)
126 minpoints(cross,crossTFit,"orange",2)
127 legend(0.2,1,c("Egnethet med inverteringsfeil","Egnethet uten
    inverteringsfeil"),pch=c(1,2),col=c("blue","orange"))
128 dev.off()
129
130
131 postscript(file="crossRel.ps",onefile=F)
132 plot(crossRel,cross,xlab="Sannsynlighet for krysning",ylab="
    Plitlighet",col="blue",pch=1,xlim=c(0.2,0.5), ylim=c(0,1.5),type=
    "n")
133 avgline2(cross, crossRel,"red",1)
134 maxpoints(cross, crossRel,"red",1)
135 minpoints(cross, crossRel,"red",1)
136 avgline2(cross, crossTRel,"green",2)
137 maxpoints(cross, crossTRel,"green",2)
138 minpoints(cross, crossTRel,"green",2)
139 avgline2(cross, tmrR,"turquoise",3)
140 maxpoints(cross, tmrR,"turquoise",3)
141 minpoints(cross, tmrR,"turquoise",3)
142 avgline2(cross, enkelR,"purple",4)
143 maxpoints(cross, enkelR,"purple",4)
144 minpoints(cross, enkelR,"purple",4)
145 legend(0.2,1.5,c("Plitlighet med inverteringsfeil","Plitlighet uten
    inverteringsfeil","Plitlighet av TMR-krets med inverteringsfeil",
    "Plitlighet av enkel multiplikator med inverteringsfeil"),pch=c
    (1,2,3,4),col=c("red","green","turquoise","purple"))
146 dev.off()
147
148
149 blokkFitness <- c(run20blokk$fitness, run40blokk$fitness,
150                   run60blokk$fitness, run20ind$fitness,
151                   run100blokk$fitness, run120blokk$fitness)
152 blokkTFitness <- c(run20blokk$t_fitness, run40blokk$t_fitness,
153                    run60blokk$t_fitness, run20ind$t_fitness,
154                    run100blokk$t_fitness, run120blokk$t_fitness)
155 blokkRel <- c(run20blokk$reliability, run40blokk$reliability,
156               run60blokk$reliability, run20ind$reliability,
157               run100blokk$reliability, run120blokk$reliability)
158 blokkTRel <- c(run20blokk$t_reliability, run40blokk$t_reliability,
159                run60blokk$t_reliability, run20ind$t_reliability,
160                run100blokk$t_reliability, run120blokk$t_reliability)
161
162 blokk <- c(20, 20, 20, 20, 20, 20, 20, 20, 20, 20,
163            40, 40, 40, 40, 40, 40, 40, 40, 40, 40,
164            60, 60, 60, 60, 60, 60, 60, 60, 60, 60,
165            80, 80, 80, 80, 80, 80, 80, 80, 80, 80,
166            100, 100, 100, 100, 100, 100, 100, 100, 100, 100,
167            120, 120, 120, 120, 120, 120, 120, 120, 120, 120)
168
```

```
169 tmrR <-
170   c(0.72093, 0.72093, 0.72093, 0.72093, 0.72093, 0.72093, 0.72093,
        0.72093, 0.72093, 0.72093,
171     0.72093, 0.72093, 0.72093, 0.72093, 0.72093, 0.72093, 0.72093,
        0.72093, 0.72093, 0.72093,
172     0.72093, 0.72093, 0.72093, 0.72093, 0.72093, 0.72093, 0.72093,
        0.72093, 0.72093, 0.72093,
173     0.72093, 0.72093, 0.72093, 0.72093, 0.72093, 0.72093, 0.72093,
        0.72093, 0.72093, 0.72093,
174     0.72093, 0.72093, 0.72093, 0.72093, 0.72093, 0.72093, 0.72093,
        0.72093, 0.72093, 0.72093,
175     0.72093, 0.72093, 0.72093, 0.72093, 0.72093, 0.72093, 0.72093,
        0.72093, 0.72093, 0.72093)
176
177 enkelR <- c(0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
178              0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
179              0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
180              0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
181              0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
182              0, 0, 0, 0, 0, 0, 0, 0, 0, 0,)
183
184 postscript(file="blokkFit.ps",onefile=F)
185 plot(blokkFitness,blokk,xlab="Antall␣blokker",ylab="Egnethet",col="
      blue",pch=1,xlim=c(20,120), ylim=c(-7,2),type="n")
186 avgline2(blokk, blokkFitness,"blue",1)
187 maxpoints(blokk, blokkFitness,"blue",1)
188 minpoints(blokk, blokkFitness,"blue",1)
189 avgline2(blokk, blokkTFitness,"orange",2)
190 maxpoints(blokk, blokkTFitness,"orange",2)
191 minpoints(blokk, blokkTFitness,"orange",2)
192 legend(20,2,c("Egnethet␣med␣inverteringsfeil","Egnethet␣uten␣
      inverteringsfeil"),pch=c(1,2),col=c("blue","orange"))
193 dev.off()
194
195
196 postscript(file="blokkRel.ps",onefile=F)
197 plot(blokkRel,blokk,xlab="Antall␣blokker",ylab="Plitlighet",col="
      blue",pch=1,xlim=c(20,120), ylim=c(0,1.5),type="n")
198 avgline2(blokk, blokkRel,"red",1)
199 maxpoints(blokk, blokkRel,"red",1)
200 minpoints(blokk, blokkRel,"red",1)
201 avgline2(blokk, blokkTRel,"green",2)
202 maxpoints(blokk, blokkTRel,"green",2)
203 minpoints(blokk, blokkTRel,"green",2)
204 avgline2(blokk, tmrR,"turquoise",3)
205 maxpoints(blokk, tmrR,"turquoise",3)
206 minpoints(blokk, tmrR,"turquoise",3)
207 avgline2(blokk, enkelR,"purple",4)
208 maxpoints(blokk, enkelR,"purple",4)
209 minpoints(blokk, enkelR,"purple",4)
210 legend(20,1.5,c("Plitlighet␣med␣inverteringsfeil","Plitlighet␣uten␣
      inverteringsfeil","Plitlighet␣av␣TMR-krets␣med␣inverteringsfeil",
```

```
    "Plitlighet av enkel multiplikator med inverteringsfeil"),pch=c
    (1,2,3,4),col=c("red","green","turquoise","purple"))
dev.off()


mutFitness <- c(run01mut$fitness, run03mut$fitness,
                run20ind$fitness, run07mut$fitness, run09mut$fitness
                  )
mutTFitness <- c(run01mut$t_fitness, run03mut$t_fitness,
                 run20ind$t_fitness, run07mut$t_fitness, run09mut$t_
                    fitness)
mutRel <- c(run01mut$reliability, run03mut$reliability,
            run20ind$reliability, run07mut$reliability, run09mut$
                reliability)
mutTRel <- c(run01mut$t_reliability, run03mut$t_reliability,
             run20ind$t_reliability, run07mut$t_reliability,
                run09mut$t_reliability)

tmrR <-
  c(0.72093, 0.72093, 0.72093, 0.72093, 0.72093, 0.72093, 0.72093,
      0.72093,
    0.72093, 0.72093, 0.72093, 0.72093, 0.72093, 0.72093, 0.72093,
       0.72093,
    0.72093, 0.72093, 0.72093, 0.72093, 0.72093, 0.72093, 0.72093,
       0.72093, 0.72093, 0.72093,
    0.72093, 0.72093, 0.72093, 0.72093, 0.72093, 0.72093, 0.72093,
       0.72093,
    0.72093, 0.72093, 0.72093, 0.72093, 0.72093, 0.72093, 0.72093,
       0.72093)

enkelR <- c(0, 0, 0, 0, 0, 0, 0, 0,
            0, 0, 0, 0, 0, 0, 0, 0,
            0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
            0, 0, 0, 0, 0, 0, 0, 0,
            0, 0, 0, 0, 0, 0, 0, 0)

mutasjon <- c(0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,
              0.03, 0.03, 0.03, 0.03, 0.03, 0.03, 0.03, 0.03,
              0.05, 0.05, 0.05, 0.05, 0.05, 0.05, 0.05, 0.05, 0.05,
                 0.05,
              0.07, 0.07, 0.07, 0.07, 0.07, 0.07, 0.07, 0.07,
              0.09, 0.09, 0.09, 0.09, 0.09, 0.09, 0.09, 0.09)


postscript(file="mutFit.ps",onefile=F)
plot(mutFitness,mutasjon, xlab="Sannsynlighet for mutasjon",ylab="
    Egnethet",col="blue",pch=1,xlim=c(0.01,0.09), ylim=c(-2.5,1),type
    ="n")
avgline2(mutasjon, mutFitness,"blue",1)
maxpoints(mutasjon, mutFitness,"blue",1)
minpoints(mutasjon, mutFitness,"blue",1)
avgline2(mutasjon, mutTFitness,"orange",2)
```

```
249 maxpoints(mutasjon, mutTFitness,"orange",2)
250 minpoints(mutasjon, mutTFitness,"orange",2)
251 legend(0.01,1,c("Egnethet med inverteringsfeil","Egnethet uten
        inverteringsfeil"),pch=c(1,2),col=c("blue","orange"))
252 dev.off()
253
254
255 postscript(file="mutRel.ps",onefile=F)
256 plot(mutFitness,mutasjon,xlab="Sannsynlighet for mutasjon",ylab="
        Plitlighet",col="blue",pch=1,xlim=c(0.01,0.09), ylim=c(0,1.5),
        type="n")
257 avgline2(mutasjon, mutRel,"red",1)
258 maxpoints(mutasjon, mutRel,"red",1)
259 minpoints(mutasjon, mutRel,"red",1)
260 avgline2(mutasjon, mutTRel,"green",2)
261 maxpoints(mutasjon, mutTRel,"green",2)
262 minpoints(mutasjon, mutTRel,"green",2)
263 avgline2(mutasjon, tmrR,"turquoise",3)
264 maxpoints(mutasjon, tmrR,"turquoise",3)
265 minpoints(mutasjon, tmrR,"turquoise",3)
266 avgline2(mutasjon, enkelR,"purple",4)
267 maxpoints(mutasjon, enkelR,"purple",4)
268 minpoints(mutasjon, enkelR,"purple",4)
269 legend(0.01,1.5,c("Plitlighet med inverteringsfeil","Plitlighet uten
        inverteringsfeil","Plitlighet av TMR-krets med inverteringsfeil"
        ,"Plitlighet av enkel multiplikator med inverteringsfeil"),pch=c
        (1,2,3,4),col=c("red","green","turquoise","purple"))
270 dev.off()
```