KU LEUVEN

FACULTEIT INGENIEURSWETENSCHAPPEN

Evolutionaire optimalisatie van soft robots

Manu De Block

Thesis voorgedragen tot het behalen van de graad van Master of Science in de ingenieurswetenschappen: werktuigkunde

Promotor:
Prof. dr. ir. Dominiek Reynaerts

Assessoren:
Prof. dr. ir. E. Ferraris
Ir. N. Famaey

Begeleider:
Ir. B. Gorissen

Academiejaar 2015 – 2016

Voorwoord

Met deze masterproef komen mijn studies burgerlijk ingenieur tot een einde, een leerzame reis. Het onderwerp soft robots sprak me meteen aan en het onderzoek bood me de kans om mijn beide specialisaties uit de bachelorjaren te combineren: werktuigkunde en computerwetenschappen.

Uiteraard zijn er een aantal personen die ik niet genoeg kan bedanken voor hun inbreng en steun. Dankuwel Dominiek Reynaerts, mijn promotor, om tijd te maken in uw drukke agenda voor advies en begeleiding. Daarnaast wil ik Nele Famaey en Benjamin Gorissen enorm bedanken voor de uitstekende begeleiding en het grote vertrouwen, ook in periodes van weinig vooruitgang, en voor het uitvoeren van de trektesten en experimenten ter ondersteuning van dit onderzoek. Een dikke merci aan mijn familie en vrienden voor hun steun en voor het verdragen van de grillige leefgewoonten van een thesisstudent. Tot slot ben ik Mag Selwa dankbaar, voor het verhelpen van mijn problemen op de VSC, Jonas, omdat we heel de zomer tegen elkaar konden zagen over onze 'beklagenswaardige' situatie, de ramsquad en wijlen Theodoor van der Waeteren, om het overduidelijk te maken dat burgerlijk in Leuven voor mij de toekomst was.

The highest reward for a person's toil is not what they get for it, but what they become by it.

— John Ruskin

Manu De Block

Inhoudsopgave

Samenvatting

Het doel van deze masterproef is het nagaan van de realiseerbaarheid van een optimalisatieprogramma voor het automatiseren van het ontwerp van pneumatische actuatoren. Deze worden veelvuldig gebruikt in soft robots.

Het 3D probleem wordt in dit onderzoek niet behandeld, maar vereenvoudigd tot het 2D probleem van de topologische optimalisatie van de doorsnede van een actuator die zich in een *plane strain* toestand bevindt. Met behulp van een Python-implementatie van NSGA-II als genetisch algoritme wordt de optimalisatie uitgevoerd voor twee actieve objectieven; een minimale massa en een minimale afwijking tussen de gewenste verplaatsing van een eindpunt en de berekende verplaatsing in de vervormde toestand. De vervorming door een aangelegde inwendige druk wordt gesimuleerd met behulp van eindige elementen berekeningen in Abaqus.

Na het aanmaken van een beginpopulatie, doorloopt het genetisch algoritme generaties van nieuwe populaties die door mutatie en kruising uit de vorige populatie samengesteld worden, waarbij de beste oplossingen steeds behouden blijven. Het simuleren in Abaqus en het genereren van figuren gebeurt hierbij automatisch.

Na een parameteroptimalisatie van het genetisch algoritme voor dit specifieke probleem en het toevoegen van mechanismen die de performantie nog verder verbeteren, zoals toernooi selectie en een mechanisme om het ontstaan van duplicaten tegen te gaan, werd de gerealiseerde code op enkele testproblemen toegepast.

Het optimalisatieprogramma bleek in staat de topologieën te ontwerpen met de gewenste verplaatsingen aan een zeer goede convergentiesnelheid. Na enkele tientallen generaties met een populatie van 100 te doorlopen, wat op een server enkele uren in beslag neemt, kan voor kleine topologieën een reeks Pareto-optimale oplossingen bekomen worden met toenemende nauwkeurigheid voor toenemende massa's.

Verder onderzoek is nodig voor de uitbreiding naar 3D problemen, volledige *path tracing* en een parallelle uitvoering van het programma. Ook het implementeren van waarheidsgetrouwe materiaalmodellen, wat in dit onderzoek gestart is, vereist verdere aandacht. Dan pas kan de software voldoende nauwkeurig het werkelijke gedrag simuleren om de productie van bekomen oplossingen te overwegen.

Lijst van figuren

Lijst van tabellen

Lijst van afkortingen en symbolen

Afkortingen

CDA	Crowding Distance Assignment, een onderdeel van NSGA-II dat aan elke vector een waarde toekent op basis van hoe ver hij zich van andere vectoren bevindt in hetzelfde front van de oplossingsruimte.
CSV	Comma Separated Value bestand
DNA	Desoxyribonucleïnezuur, hier gebruikt als aanduiding voor de binaire voorstelling van een topologie, omdat de rol die deze voorstelling speelt in het genetisch algoritme analoog is aan die van DNA in de natuur.
EA	Evolutionair algoritme
FEM	Finite Element Method, meest herkenbare afkorting van de Eindige Elementen Methode.
float	Floating point number, datastructuur die een reëel getal opslaat als een beduidend deel en een exponent.
GA	Genetisch algoritme
Gem (Avg)	Gemiddelde
GUI	Graphical User Interface
int	Integer, geheel getal datatype
Max	Maximum
Min	Minimum
MPI	Message Passing Interface (in parallellisatie).
NSGA-II	Non-dominated Sorting Genetic Algorithm II, een snel genetisch algoritme met elitisme, ontwikkeld door K. Deb et al. [6]
SGA	Simple Genetic Algorithm, het meest primitieve GA.
str	String, tekst datatype
XOR	Exclusive-OR test, True als beide elementen verschillend zijn, False indien ze gelijk zijn.

Symbolen

Δx	Verschil tussen gewenste en werkelijke x-coordinaat.
Δy	Verschil tussen gewenste en werkelijke y-coordinaat.
dx	Gewenste verplaatsing van het eindpunt in de x-richting.
dy	Gewenste verplaatsing van het eindpunt in de y-richting.
G	Glijdings- of schuifmodulus (*shear modulus*).
MxN	Voorstelling van een topologie met M de x-waarde (breedte) en N de y-waarde (hoogte). NxN wordt gebruikt voor een vierkante topologie.
M	Aantal generaties (of objectieven)
N	Populatiegroote
n	Aantal bits in een DNA-string, het aantal cellen van een topologie.
t	Tijd [s]
x	Meestal gebruikt voor de breedte van de topologie.
y	Meestal gebruikt voor de hoogte van de topologie.

De term vector

Oplossingen of individuen in een populatie worden in deze tekst meestal benoemd als vectoren, zoals in de paper van A. Saxena[18]. Hoewel veel genetische algoritmen op een populatie van wiskundige vectoren werkzaam zijn, moet de term vector hier niet in zijn wiskundige maar eerder moleculair-biologische betekenis opgevat worden als de kleinste eenheid of molecule die genetisch materiaal ofwel DNA bevat.

Hoofdstuk 1

Inleiding

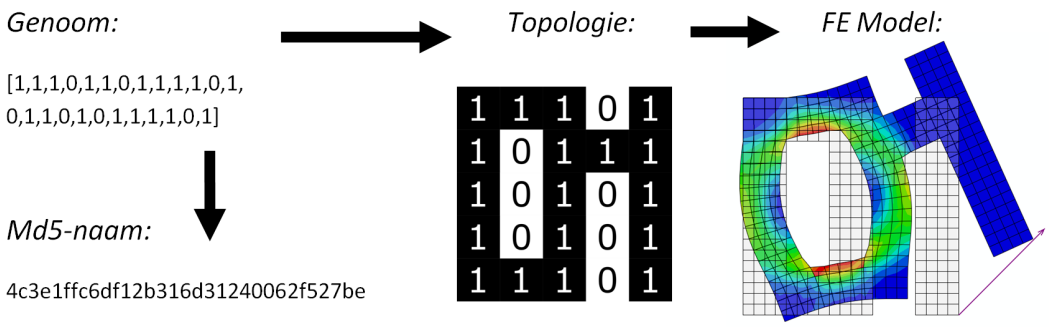
Soft robotics is een tak van de robotica die gebruik maakt van flexibele continue hyper-redundante structuren. Deze winnen aan interesse omwille van de betere prestaties in complexe, delicate en onvoorspelbare omgevingen. Dit opent mogelijkheden voor medische toepassingen en hulpverlening en voor verkenning en reddingswerkzaamheden. Vaak is de werking van deze robots gebaseerd op pneumatische actuatoren, die specifiek ontworpen moeten worden om een gewenste beweging uit te voeren.

Het doel van deze masterproef is de uitvoerbaarheid van een automatisch ontwerpproces dat gebruikt maakt van optimalisatiesoftware achterhalen. Omwille van de complexiteit van een dergelijk optimalisatieproces is een heuristische aanpak gewenst, die in een acceptabele tijd oplossingen levert voor een opgegeven set beperkingen en gewenst gedrag. Een genetisch algoritme voldoet aan deze vereiste en kan op een heuristische manier de optimalisatie in de gewenste richting laten convergeren.

Ondanks de driedimensionale aard van het eigenlijke ontwerpprobleem, beperkt dit onderzoek zich tot de topologische optimalisatie van de doorsnede van een pneumatische actuator, opdat een zijde van deze effector een gewenste beweging zou uitvoeren. Dit behelst dus twee vereenvoudigingen van het algemene probleem; er wordt gewerkt in tweedimensionale in plaats van driedimensionale ruimte en enkel de nauwekeurigheid van de eindpositie van de beweging wordt geëvalueerd. De lengte van deze effector in de niet beschouwde richting wordt groot genoeg verondersteld om de vereenvoudiging naar een *plane strain* toestand te kunnen maken en het gedrag van de tweedimensionale topologie uit te rekenen met behulp van eindige elementen modellering.

Voor de optimalisatie wordt het ontwerpgebied, binnen de maximale dimensies van de doorsnede, gediscretiseerd in vierkante cellen, rekening houdend met de gewenste hoeveelheid detail. Deze cellen vertegenwoordigen ofwel lege ruimte, waarbij ze de binaire waarde 0 toegewezen krijgen, ofwel vertegenwoordigen ze een vierkant blokje materiaal, met binaire waarde 1. Deze topologie kan voorgesteld worden door haar

genoom of lijst van alle opeenvolgende bits, of de *md5-checksum* van dit genoom om grote topologieën met een korte unieke naam te kunnen aanduiden. In figuur 1.1 is een overzicht gegeven van de voorstellingsvormen van de topologie die in deze thesis gebruikt worden.



Figuur 1.1: Verschillende voorstellingen van een vector (of diens DNA): het genoom (lijst bits), de md5-naam, de matrixvoorstelling of het FE model met eventueel een fijner rooster.

Het genetisch algoritme modelleert de 'niet-willekeurige evolutie door middel van willekeurige mutaties' zoals deze bij levende organismen plaatsgrijpt. Het binaire genoom van de topologie ondergaat de simulatie van een genetisch proces, waarin kruisingen en mutaties plaatsgrijpen en de individuen met de beste '*fitness*' overleven.

In hoofdstuk 2 van deze tekst wordt een uiteenzetting gegeven uit de literatuur over de onderwerpen die in deze masterproef aan bod komen. De voordelen en toepassingen van soft robots en genetische algoritmen worden besproken, net als recente ontwikkelingen op deze gebieden. Specifieke concepten die gebruikt worden zoals het Ogden materiaalmodel en het NSGA-II genetisch algoritme worden verklaard en tot slot wordt onderzoek besproken rond vergelijkbare topologische optimalisatieprocessen aan de hand van genetische algoritmen en FEM.

Hoofdstuk 3 gaat in op de specifieke implementatie van de benodigde onderdelen zoals het genetisch algoritme en genetische operatoren, FEM en methodes om vectoren te genereren en controleren.

Het bekomen optimalisatieprogramma wordt in hoofdstuk 4 aan tests en analyses onderworpen om de performantie te verbeteren. Om te beginnen worden parameters zoals de populatiegrootte, kruisings- en mutatiekans geoptimaliseerd door hun onderlinge invloed na te gaan en ook de invloed van andere parameters te evalueren.

Verdere verbeteringen worden bekomen door in hoofdstuk 5 genetische operatoren te verfijnen en toe te voegen, waaronder de *tournament selection operator* en een methode om het ontstaan van duplicaten te reduceren.

De ontwikkelde code kan toegepast worden met een veelheid van materiaalmodellen en FEM instellingen, zolang deze juist naar een invoerbestand vertaald worden. In appendix A wordt er toch even stilgestaan bij het fitten van geschikte parameters voor een specifieke keuze van materiaal en materiaalmodel en de implicaties van een bepaalde keuze van het type en het rooster van eindige elementen. Hoewel dit een volwaardig hoofdstuk is, bevindt het zich in de bijlagen omdat het een zijspoor is in deze reeds lijvige tekst.

De resultaten bekomen met de ontwikkelde code, worden in hoofdstuk 6 voorgesteld voor enkele testproblemen van toenemende complexiteit. De bekomen topologieën en de convergentiesnelheid naar deze oplossingen wordt besproken aan de hand van automatisch gegenereerde grafieken en voorstellingen uit Abaqus.

Tot slot wordt een besluit geformuleerd, met enkele voorstellen tot verder onderzoek. In appendix B wordt nog uitgebreider ingegaan op mogelijke verbeteringen en toevoegingen die zich lenen tot verder onderzoek.

In de appendices zijn, naast het hoofdstuk A over het materiaalmodel en hoofdstuk B over verder onderzoek, extra grafieken te vinden bij de generatiemethodes uit hoofdstuk 3 (appendix C) en parameteroptimalisaties uit hoofdstuk 4 (appendix D). Daarnaast bevatten de bijlagen een voorbeeld van een Abaqus input bestand (appendix E) en uitleg bij de Python modules en functies, evenals de gerealiseerde code zelf (appendix F en G).

Hoofdstuk 2

Literatuuroverzicht

In dit hoofdstuk wordt achtergrondinformatie uit de literatuur gegeven bij de concepten waarvan gebruik wordt gemaakt in deze masterproef, zijnde soft robots, genetische algoritmen en het werken met een niet-lineair constitutief materiaalmodel in eindige elementen berekeningen. Tot slot worden enkele onderzoeken voorgesteld die een vergelijkbare probleemstelling hebben als deze masterproef.

2.1 Soft robots

Klassieke robots bestaan vaak uit rigide delen, hebben een discreet aantal vrijheidsgraden en worden aangestuurd door elektromagnetische actuatoren. Soft robots zijn een recente klasse robots die, in contrast met de klassieke robots, weinig rigide structuurelementen bevatten, ze zijn continu en hyperredundant. Ze zijn vaak geïnspireerd op dieren met niet-rigide lichaamsdelen.[25][27]

2.1.1 Flexibele structuren

Om te beginnen worden de voordelen van een eenvoudig flexibel mechanisme op een stangenmechanisme beschouwd. Flexibele mechanismen, of *compliant mechanisms*, hun Engelse naam waaronder ze beter bekend zijn, zijn elastische structuren die een gewenste beweging uitvoeren of een kracht overbrengen bij het aanbrengen van een inputkracht. [19] Het zijn eigenlijk continue uitbreidingen van stangenmechanismen, gezien de gehele structuur vervormbaar is en een oneindig aantal vrijheidsgraden heeft. Dit vergeleken met het eindig aantal vrijheidsgraden en de opbouw uit scharnieren met daartussen starre elementen bij stangenmechanismen. Flexibele mechanismen hebben sterke voordelen tegenover de klassieke stangenmechanismen. Ze zijn scharnierloos en bestaan uit één geheel, wat leidt tot een verminderde wrijving en bijgevolg ook minder slijtage en lawaai. De productie is eenvoudiger door het gebrek aan een assemblagestap en bovendien zijn ze erg licht. [20] Naast de mechanische kan ook een algoritmische vereenvoudiging bekomen worden in het ontwerpen van robots door het toepassen van flexibele onderdelen. De capaciteiten van soft robots zitten

voor een groot deel vervat in hun materiaalgedrag en morfologie, niet in regel- en controlesystemen.[10] Deze zijn bij klassieke robots uitermate belangrijk omdat ze gemakkelijk hun omgeving en zichzelf kunnen beschadigen als ze onstabiel worden.[26]

In de literatuur zijn er volgens Sharma et al. [21] twee methoden beschreven voor het ontwerpen van flexibele structuren. De eerste is een traditionele kinematische synthese uitgaande van een stangenmechanisme. De tweede methode maakt gebruik van continuümmechanica om de topologie, vorm en grootte van het flexibel mechanisme te bepalen. Bij het bepalen van deze topologie wordt vaak gebruik gemaakt van eindige elementen door de initiële ontwerpruimte te discretiseren. De optimalisatieprocedure zal dan beslissen of een element materiaal bevat of niet, gebruik makend van één van de methoden beschreven in de paper van Sharma et al.[21]. Deze gebruiken ofwel een dichtheidsfunctie met een drempelwaarde die bepaalt of het element materiaal bevat, wat bij bepaalde keuzes tot een niet-optimaal ontwerp kan leiden, ofwel een puur binaire aanpak. Deze laatste garandeert de discrete aard van het probleem, maar heeft een aantal nadelen zoals de aanleg tot het produceren van schaakbordpatronen en losse elementen die geen deel uitmaken van de structuur. Dit zal belangrijk worden verder in deze masterproef, gezien deze methode toegepast zal worden.

2.1.2 Voordelen, nadelen en toepassingen

Soft robotics gaat echter veel verder dan flexibele structuren, vaak betreft het hyperredundante continue actuatoren die pneumatisch aangestuurd worden. Door hun zachte structuur en redundante vrijheidsgraden wordt de interactie met de omgeving minder gelimiteerd dan bij klassieke robots, waardoor ze moeilijke taken kunnen uitvoeren in ongestructureerde en dense omgevingen. Dit opent mogelijkheden in militaire toepassingen en voor reddingswerkzaamheden.[27]

Soft robots zijn meestal niet autonoom of zelfbevattend, omwille van de moeilijkheid om alle nodige onderdelen (batterijen, voor een pneumatische aansturing: compressoren, kleppen, ...) in de robot zelf op te nemen door zijn beperkte grootte en kracht. Met een verbinding zijn ze dan aan een externe (druk)bron vastgemaakt, wat hun mobiliteit belemmert. Deze keten kan echter ook een voordeel zijn, als er samples van stoffen worden genomen en voor het doorsturen van beelden en elektronische communicatie.[25]

Soft robots kunnen de mogelijkheden van natuurlijke systemen beter benaderen dankzij hun flexibele structuur; ze zijn minder gevoelig voor onzekerheid en variatie, ze passen zich aan aan onbekende objecten en omgevingen en kunnen krachten beter afleiden en verdelen.[12] Klassieke robots zouden bijvoorbeeld problemen hebben met gladde en oneven oppervlakken, harde en zachte obstakels, bewegende objecten en wind. Zelfs als deze allemaal bekend zijn, kan de geschikte reactie vaak niet berekend worden of heeft de robot niet genoeg vrijheidsgraden.[26]

Door hun lage stijfheid zijn soft robots daarnaast ook inherent veiliger in taken waar

mensen bij betrokken zijn, waar klassieke robots steeds afgeschermd moeten worden van mensen.[15] Vooropgestelde toepassingen die Onal en Rus [15] zien, zijn systemen om bejaarden of gehandicapten te helpen met moeilijke taken zoals rechtkomen uit bed of dingen van de grond oprapen. Zelfs handschoenen die het effect van een tremor compenseren, worden genoemd als een voorbeeld van wat op lange termijn mogelijk kan worden. Soft robots zijn erg geschikt als medische robots voor het gebruik bij operaties en diagnoses diep in het lichaam.[27][26] In combinatie met *tissue engineering* zijn zelfs hybride systemen voor medische applicaties denkbaar.[10]

Soft robots zijn eveneens veiliger in natuurlijke omgevingen omwille van de hierboven vernoemde redenen. Ze kunnen zo ontworpen worden dat ze de omgeving niet contamineren (bv. smeermiddel) of gecontamineerd worden door de omgeving (bv. stof, vochtinsijpeling); een soft robot die via een verbinding aangestuurd wordt, bestaat enkel uit kunststof en een fluïdum.[10]

Op grote schaal kan de productie van soft robots potentieel veel goedkoper worden dan van klassieke robots. Dit kan ze in grotere getale, zoals in zwermen, inzetbaar maken.[26]

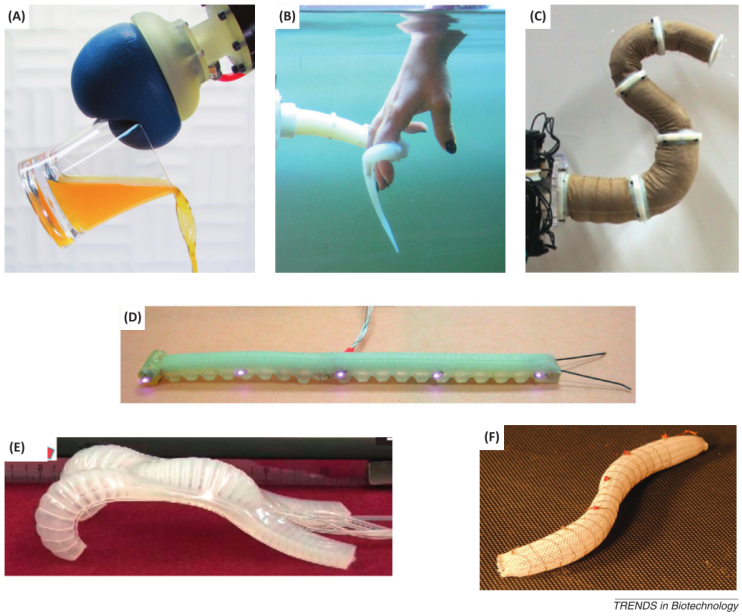
Een nadeel aan soft robots is dat ze net zoals hun natuurlijke tegenhangers gelimiteerd zijn in grootte, omdat ze moeilijk hun eigen lichaamsgewicht kunnen dragen zonder skelet. Ook is de maximale versnelling gelimiteerd door het vervormingsgedrag, de robots kunnen zich maar met een beperkte snelheid verplaatsen. Verschillende delen van de robot kunnen wel snel tegenover elkaar bewegen onder lage belasting. Stijve componenten kunnen dus wenselijk zijn afhankelijk van de schaal van de robot.[10]

2.1.3 Voorbeelden van soft robots

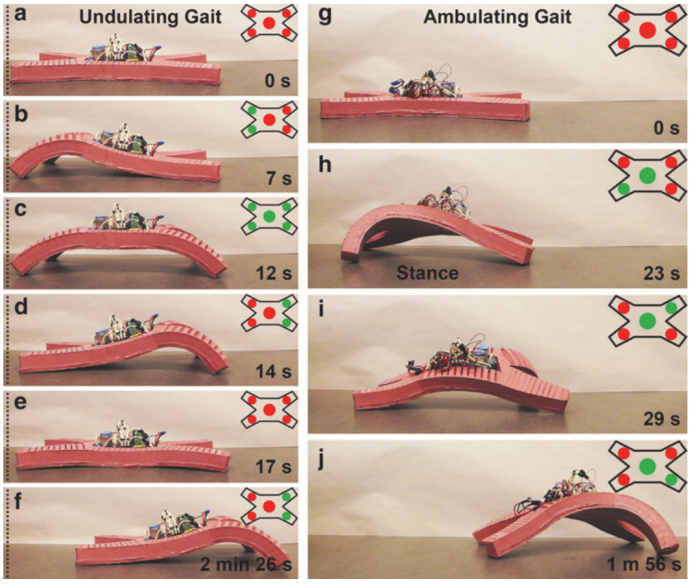
Dieren die als inspiratie dienden voor soft robots zijn onder andere zeesterren[25], inktvissen, wormen, larven,[10], slangen[15], vissen[12], slakken, anemonen en ledematen van zoogdieren, zoals tongen en de slurf van een olifant.[27] Soft robots zonder keten zijn reeds gedemonstreerd onder water[12] en aan land, met rollende en slangachtige bewegingen[15].[25]

Enkele voorbeelden uit de paper van Kim et al.[10] zijn te vinden in figuur 2.1.

In het recente werk van Tolley et al.[25] werd een $0,65\ m$ lange soft robot ontwikkeld die kan rondkruipen. Hij is volledig zelfbevattend, ondanks dat hij op luchtdruk werkt. Miniatuurcompressoren, batterijen, kleppen en een controller bevinden zich allemaal in de robot. Hij heeft een lichaam van silicone elastomeren met polyamide draden en holle glazen microsferen. De robot kan tot 8 kg dragen, kan zich verplaatsen door sneeuw en plassen water en is bestand tegen korte directe blootstelling aan vlammen en het overreden worden door een auto. De robot is te zien in figuur 2.2.



Figuur 2.1: Enkele bio-geïnspireerde soft robots uit [10]. (A) Zachte grijper, bestaande uit een flexibele zak granulair materiaal, werkt bij vacuüm. (B) Zachte manipulator gemodelleerd naar de arm van een octopus. (C) Effector gemodelleerd naar een slurf. (D) GoQBot, kan ballistisch rollen zoals een rups. (E) Wandelende soft robot met meerdere inputs. (F) De 'Meshworm', gemaakt uit een rooster van speciale NiTi-draden dat een peristaltische beweging mogelijk maakt.



Figuur 2.2: De robot uit het werk van Tolley et al. [25]. De groene stippen duiden aan welke delen onder druk staan. Links is een golfbeweging te zien, rechts een wandelbeweging.

2.2 Materiaalmodel voor eindige elementen berekeningen

Om een flexibel mechanisme te ontwerpen, wordt vaak gebruik gemaakt van lineaire eindige elementen. De grote verplaatsingen en vervormingen die zullen optreden bij het functioneren van de soft robots die deze masterproef oogt te ontwerpen, vereisen uiteraard het toepassen van niet-lineaire eindige elementen om een voldoende nauwkeurigheid te bereiken.

Verder moet het gedrag van PDMS-10, het voorgestelde polyelastomeer voor deze robots, in een hyperelastisch materiaalmodel uitgedrukt worden. Opdat de berekeningen in deze thesis overeen zouden komen met de werkelijkheid, is het van belang een constitutief model te kiezen dat het materiaalgedrag goed beschrijft. Aangezien dergelijke constitutieve modellen fenomenologisch zijn, dit wil zeggen dat ze materiaalgedrag beschrijven op basis van observaties en niet op basis van achterliggende fysica, moeten geschikte materiaalparameters zorgvuldig afgeleid worden uit spannings-rek gegevens van relevante testen.

2.2.1 Polydimethylsiloxaan

Voor het realiseren van een soft robot is een elastomeer nodig. Een geschikt elastomeer, dat ook voorhanden was voor testen, is PDMS-10 of polydimethylsiloxaan met een verhouding van tien keer meer basispolymeer dan verhardingsmiddel.

De paper van Kim et al.[11] beschrijft de eigenschappen van dit materiaal en analyseert welk materiaalmodel aangewezen is om het gedrag te beschrijven. PDMS wordt voor gevarieerde micro-apparaten gebruikt vanwege zijn goede eigenschappen. Het is goedkoop, transparant, biocompatibel en flexibel.

De materiaaleigenschappen zijn sterk afhankelijk van de mengverhouding. Data uit unaxiale trektesten op samples van 100 μm bij 4 mm bij 40 mm werden gefit voor PDMS-05, -10 en -15. Als niet-lineaire materiaalmodellen werden het Neo-Hooke, het tweede orde Ogden en het derde orde Mooney-Rivlin model gebruikt. Kim et al. concludeerden dat het Ogden(2) model de voorkeur geniet, enkel dit model is een degelijke benadering in het niet-lineaire gebied met grote rekken. Het Neo-Hooke model is te stijf en het Mooney-Rivlin(3) model onderschat de stijfheid. De voorgestelde parameters voor het Ogden(2) model zijn te vinden in tabel A.2.[11]

Het spannings-rek gedrag kan helaas ook significant verschillen van sample tot sample, zelfs al zijn deze op identieke wijze bereid.[8]

2.2.2 Het Ogden materiaalmodel

Het Ogden model is een frequent gerbuikt model dat steunt op een rek-gebaseerde functie voor de vervormingsenergie:[3]

$$W_O(\lambda_1, \lambda_2) = \sum_{p=1}^{N} \frac{\mu_p}{\alpha_p}(\lambda_1^{\alpha_p} + \lambda_2^{\alpha_p} + \lambda_1^{-\alpha_p}\lambda_2^{-\alpha_p} - 3) \tag{2.1}$$

Met λ_i de normaalspanningen uit de vervormingstensor:[29]

$$F = \begin{bmatrix} \lambda_1 & \kappa_1 & 0 \\ \kappa_2 & \lambda_2 & 0 \\ 0 & 0 & \lambda_3 \end{bmatrix} \tag{2.2}$$

De normaalspanningen λ in de bovenstaande matrix, worden vervangen door de extensie ratio $\lambda = l_t/l_0$ met l_t de huidige lengte van het specimen en l_0 de oorspronkelijke (onbelaste) lengte. De Jacobiaan van F is gelijk aan $\lambda_1 \cdot \lambda_2 \cdot \lambda_3$ en ook aan één, gezien het materiaal als onsamendrukbaar beschouwd mag worden en de afschuifspanningen κ verwaarloosd mogen worden omdat het materiaal in een trektest dun is.

In de Ogden energiefunctie is de voorlaatste term met andere woorden eigenlijk $\lambda_3^{\alpha_p}$. Bovenstaande formulering is echter duidelijker, gezien enkel de λ_1 en λ_2 uit een biaxiale trektest bekomen worden.

In een tweede orde Ogden model is $N = 2$ en moeten dus de parameters μ_1, μ_2, α_1 en α_2 experimenteel bepaald worden uit meetdata om het model op te stellen. Interessant met het oog op de resultaten van Kim et al.[11], is dat uit het Ogden model het Neo-Hooke model bekomen kan worden met $N = 1$ en $\alpha_1 = 2$ en het Mooney-Rivlin(2) model met $N = 2$, $\alpha_1 = 1$ en $\alpha_2 = 2$.[3]

Berselli et al. [3] vermelden ook dat om een model met een realistisch en stabiel materiaalgedrag te bekomen de volgende ongelijkheden moeten gelden (met G de glijdingsmodulus):

$$\mu_p\alpha_p > 0 \qquad\qquad p = 1, ..., N \tag{2.3}$$

$$\sum_{p=1}^{N} \mu_p\alpha_p = 2G \tag{2.4}$$

2.2.3 Parameterfitting uit trektesten

Om de parameters te berekenen uit resultaten van eigen testen, gaat men als volgt te werk. De objectieffunctie van de optimalisatie is hier het verschil tussen de experimentele spanningen en de spanningen die het model voorspelt, uitgaande van de verplaatsingen.

$$\sigma_{ij} = -\overline{p}\delta_{ij} + F_{ik}\frac{\partial W_O}{\partial bf F_{jk}} \tag{2.5}$$

In formule (2.5) is δ_{ij} de Kronecker delta en $\overline{p}$ een onbepaalde Lagrange vermenigvuldiger die nodig is omdat onsamendrukbaarheid verondersteld werd. Hij kan enkel afgeleid worden uit evenwichts- en randvoorwaarden.[3]

De experimentele spanningen moeten ook eerst nog berekend worden, want tijdens het experiment worden de krachten en niet de spanningen gemeten. Een eenvoudige methode die verder in deze thesis aan bod zal komen, is een biaxiale trektest op een dun specimen. Dit type test wordt vaak toegepast bij zachte weefsels die als niet-lineaire elastische materialen gemodelleerd kunnen worden.[29]

2.3 Genetische algoritmen

Evolutionaire algoritmen (EA) zijn heuristische optimalisatie- of zoekprogramma's die analoog werken aan het evolutionaire proces dat in de natuur werkzaam is. Het algoritme opereert in generaties van individuen of oplossingen met een zekere *'fitness'* die samen de populatie vormen. Evolutionaire algoritmen worden meestal gebruikt in problemen met een zeer grote oplossingsruimte.[23]

Genetische algoritmen (GA) horen bij de meest gebruikte EA's en maken gebruik van de analogie met natuurlijke evolutie waarbij ouders uit de ene generatie nakomelingen voor de volgende generatie produceren. De *fitness* van een individu bepaalt hoe groot zijn overlevings- en reproductiekans is. Bij de reproducties wordt het genetisch materiaal van beide ouders gecombineerd (kruising), in dit proces kunnen er fouten optreden (mutatie).[23]

2.3.1 Werking en voorbeelden

De meest primitieve vorm van een GA is het Simple Genetic Algorithm (SGA). Om te beginnen wordt een populatie willekeurige individuen aangemaakt. Vervolgens wordt er geïtereerd over generaties tot een stopcriterium bereikt wordt. Elke iteratiestap verloopt als volgt: de individuen die overleven en naar de volgende generatie gaan, worden gebruikt om nieuwe individuen te produceren door selectie, kruising en mutatie. Selectie gebruikt vaak een schalingsmechanisme dat aan elk individu een bepaalde kans toekent om deel te nemen in de reproductie, gebaseerd op zijn *fitness*. Elitisme is een extra operator die zorgt dat de beste oplossingen van een generatie rechtstreeks doorgaan naar de volgende generatie.[23]

Voor meer informatie over de basis van genetische algoritmen kan men hoofdstuk drie van het boek van Shukla et al.[23] raadplegen.

Wanneer een evolutionair algoritme rekening houdt met meerdere objectieven, spreekt men van een multi-objective evolutionary algorithm (MOEA). Het grote voordeel van deze algoritmen, is dat ze meerdere pareto-optimale oplossingen kunnen vinden in één enkele run. In de periode 1993-1995 werden verschillende MOEA's ontwikkeld, hiervan genoten MOGA, NSGA (Non-dominated Sorting GA) en NPGA (Niched Pareto

GA)[5] het meeste aandacht.[6] De voornaamste innovaties waren *non-dominated sorting* en het behouden van diversiteit. Met het oog op het invoeren van meer operators uit gewone EA's zoals elitisme, ontstonden later algoritmen als SPEA(2) (Strength Pareto Evolutionary Algorithm)[30], PESA(-II) (Pareto Enveloppe-based Selection Algorithm)[5], PAES (Pareto Archived Evolution Strategy), elitist GA en NSGA-II.[6]

Verder zijn er nog algoritmen zoals MOMGA (MO-Messy-GA) en M-PAES (Mimetic-PAES) die het pure evolutionaire model verlaten.[5]

2.3.2 Voordelen

De motivatie voor de keuze van een genetisch algoritme in het onderzoek van A. Saxena[18] en bij uitbreiding ook in deze thesis, is dat een ontwerpprobleem voor een topologie met een vast eindige elementen rooster het best voorgesteld kan worden in zijn oorspronkelijke discrete vorm. Past men een meer traditionele aanpak toe met gradiënt-gebaseerde optimalisatie, treden enkele nadelen op. De ontwerpvariabelen kunnen niet discreet zijn ('1' of '0'), maar hebben een klein positief getal als ondergrens (eerste relaxatie) en verder wordt een continue materiaal interpolatie functie gebruikt om het berekenen van de gradiënt te vereenvoudigen (tweede relaxatie). De ontwerpvariabelen nemen dus intermediaire waarden aan, wat het interpreteren en realiseren van de oplossing bemoeilijkt, gezien er in een regio materiaal moet zijn ('1') of niet ('0'). Verder kan de optimalisatie ook vastlopen op regio's waarin de vervormingsanalyse niet convergeert, gezien elke tussenoplossing in de optimalisatie bepalend is voor de rekenstap en dus de volgende tussenoplossing. Als de gradiënt niet berekend kan worden, stopt het algoritme.[18]

Meer voordelen die aangehaald worden, zijn de mogelijkheid om interferentie tussen elementen tijdens vervormingen aan te pakken en dat modellen onmiddellijk vervaardigd kunnen worden uit de resultaten zonder voorafgaande interpretatie. De resultaten van het genetisch algoritme zijn ook nauwkeuriger, omdat onbestaande elementen gewoon weggelaten worden in plaats van ze te vervangen door een kleine positieve waarde.[18]

2.3.3 Het NSGA-II algoritme

In deze masterproef zal gebruik gemaakt worden van het *Nondominated Sorting Genetic algorithm II* van Deb et al.[6], net als in de papers die bij vergelijkbaar onderzoek in deel 2.4 worden gepresenteerd. De oorspronkelijke paper [6] heeft veruit het grootste aantal referenties voor een paper over GA's.

NSGA-II is gebaseerd op het oorspronkelijke NSGA maar pakt enkele zwakheden in het algoritme aan. Zo wordt de computationele complexiteit van het niet-gedomineerd sorteren herleid van $O(MN^3)$ naar $O(MN^2)$, met M het aantal objectieven en N de populatiegrootte. Ook bevatte het oorspronkelijke NSGA geen elitisme, waar-

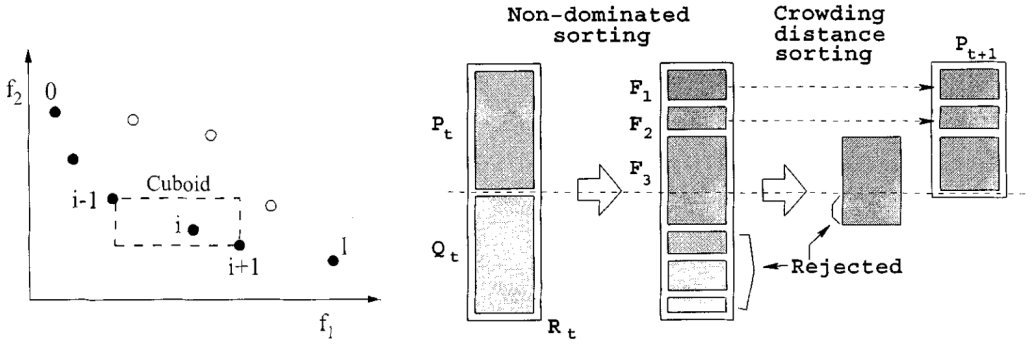
door goede oplossingen verloren konden gaan. Tot slot werd een nieuw mechanisme ingevoerd om diversiteit te bewaren dat niet afhankelijk is van een in te stellen parameter zoals σ_{share} in de oude *sharing function* aanpak en een betere computationele complexiteit heeft.[6]

Dominantie is een belangrijk concept in het algoritme. Een oplossing wordt gedomineerd door een andere oplossing, als deze op alle objectieven beter of even goed scoort als de gegeven oplossing. Dit is de kern van het niet-gedomineerd sorteren, dat de populatie verdeelt in *'fronts'* van vectoren die elkaar onderling niet domineren of gedomineerd worden, maar wel gedomineerd worden door alle voorgaande *fronts* en alle volgende *fronts* domineren. De snelle implementatie die dit in $O(MN^2)$ berekeningen uitvoert, berekent eerst voor elke oplossing hoeveel oplossingen deze vector domineren (dominantie getal n_p) en maakt een set van de oplossingen die door deze oplossing gedomineerd worden (S_p). Oplossingen met $n_p = 0$ behoren bijgevolg tot het eerste front. Voor alle oplossingen in hun set S_p wordt het dominantie getal verlaagd, indien het daardoor nul wordt, wordt de oplossing aan het tweede front toegevoegd. Dit wordt herhaald tot alle fronten gevonden zijn.[6]

Het mechanisme om diversiteit te behouden, is gebaseerd op het principe van *crowding distance assignment*. Deze *crowding distance* die elke oplossing toegewezen krijgt, is gebaseerd op de afstand tot zijn buren in het front. Dit is goed te zien in figuur 2.3 (a). De *crowding distance* is de som van de genormeerde afstanden tussen de buren in alle objectieven. In de figuur zijn dat de zijden van de rechthoek, vergeleken met de afstanden tussen de uiterste oplossingen in het front ter normalisatie. De twee grensoplossingen krijgen een oneindige waarde van de *crowding distance* toegewezen, omdat ze het meest waardevol zijn. De computationele complexiteit is in het slechtste geval (één front) $O(MNlogN)$ indien een goed sorteeralgoritme gebruikt wordt.[6]

Nu kan ook een *crowding comparison operator* gedefinieerd worden. Deze selecteert uit twee oplossingen de oplossing met de laagste rang, of bij gelijke rang de grootste *crowding distance*. Dit garandeert de keuze van de beste en meest gespreide oplossingen en wordt gebruikt bij het opbouwen van een nieuwe populatie en in *tournament selection*.[6]

Het volledige NSGA-algoritme start met het maken van een willekeurige populatie P_0. Deze wordt geëvalueerd en een eerste *offspring* populatie Q_0 wordt gevormd. In elke generatie gaat NSGA-II te werk zoals in figuur 2.3 (b) voorgesteld is. Een gecombineerde populatie $R_t = P_t \cup Q_t$ met grootte $2N$ wordt gevormd om elitisme te verzekeren. Deze wordt gesorteerd in fronts met *fast non-dominated sorting*. De nieuwe generatie bestaat uit de beste fronts en de oplossingen met de hoogste *crowding distance* uit het laatste front dat gedeeltelijk nodig is om de populatie te vervolledigen. Bij het toevoegen van een front aan de nieuwe populatie, wordt de *crowding distance* berekening uitgevoerd, zodat geen tijd verloren gaat met het toekennen van een *crowding distance* aan de verworpen oplossingen. Ook het sorteren kan afgebroken worden nadat de eerste fronts meer dan N vectoren bevatten.[6]



(a) *Crowding Distance* principe (b) Niet-gedomineerd sorteren

Figuur 2.3: Visualisatie van hoe in NSGA-II (a) het toekennen van een *crowding distance* en (b) het selecteren van een nieuwe populatie werkt.

Deb et al.[6] vergeleken de performantie van NSGA-II met PAES en SPEA. In alle negen testproblemen scoorde NSGA-II het beste. PAES leverde de slechtste resultaten maar convergeerde wel het snelst in twee van de testproblemen. NSGA-II vertoonde een kleine variantie voor een gemiddelde van tien runs in alle testproblemen met uitzondering van één. NSGA-II vindt steeds de beste spreiding van mogelijke oplossingen (*real coded* scoort beter dan *binary coded*). Wel wordt er gewaarschuwd dat in sterk epistatische problemen (grote afhankelijkheid tussen de 'genen') MOEA's mogelijk problemen kunnen ondervinden.[6]

SPEA2, de opvolger van SPEA, levert betere resultaten dan PESA en SPEA, maar NSGA-II heeft een gelijkaardige performantie, vaak met een betere spreiding van de resultaten.[30]

2.4 Vergelijkbaar onderzoek

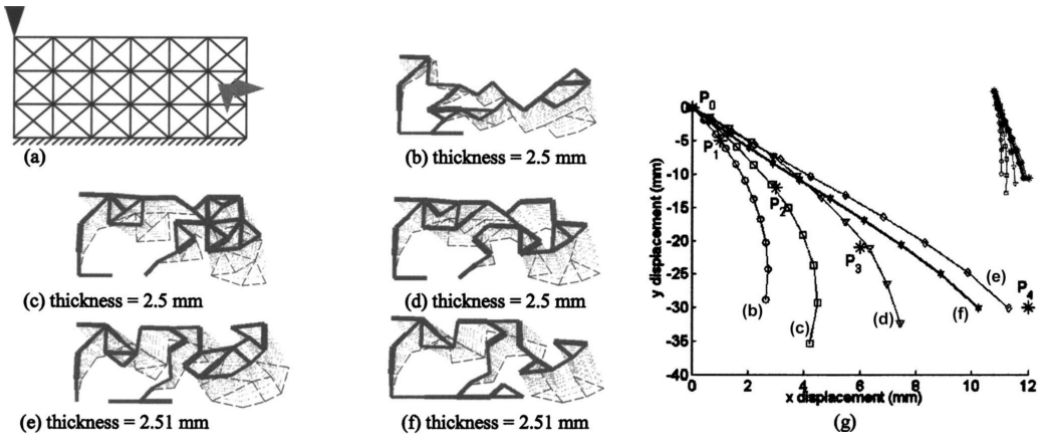
De volgende onderzoeken naar een topologische optimalisatie met behulp van een genetisch algoritme bevatten waardevolle ideeën voor het onderzoek in deze masterproef.

2.4.1 Synthese van flexibele mechanismen met frame elementen

A. Saxena paste in zijn onderzoek naar flexibele mechanismen genetische algoritmen toe, om een mechanisme dat een voorgeschreven pad volgt te optimaliseren.[18]

Het ontwerpdomein bevat een inputkracht, een vaste regio voor bewegingsrandvoorwaarden en precisiepunten die de gewenste verplaatsing opleggen. Het eindige elementen rooster zelf lijkt op een vakwerk, elke 'cel' bestaat uit vier elementen in een rechthoek en vier elementen op de diagonalen. De elasticiteitsmodulus wordt gebruikt als ontwerpvariabele: $E_i = 0$ betekent dat het element afwezig is en $E_i = E_0$

dat het element aanwezig is. Ook de doorsnede wordt geoptimaliseerd, de dikte t uit het vlak is uniform, maar de breedte w_i van een element in het vlak is variabel. Ten derde mogen de posities van de knopen binnen een beperkte ruimte gekozen worden (ontwerpvariabelen x_i en y_i). De afwijking tegenover de precisiepunten wordt geminimaliseerd, zowel elk als afzonderlijk objectief, als gecombineerd met de kleinste kwadraten metriek. Voor n precisiepunten worden dus $n + 1$ pareto optimale oplossingen bepaald. De oplossing van deze optimalisatie wordt berekend met NSGA-II, met de elasticiteit discreet (*'binary coded'*) en de breedtes en coördinaten continu tussen een onder- en bovengrens (*'real coded'*).[18]



Figuur 2.4: Flexibel mechanisme uit de tests van A. Saxena [18]: (a) ontwerpruimte (b)-(e) optimale oplossing voor een specifiek precisiepunt (f) oplossing voor kleinste kwadraten van alle precisiepunten (g) verplaatsing van het eindpunt in alle oplossingen vergeleken met de precisiepunten.

In figuur 2.4 is een resultaat te zien voor een test met een populatie van 100 en een duur van 1000 generaties. De oplossingsruimte bevatte 117 elementen in een rechthoek van 150 mm bij 60 mm. Een prototype uit ABS kunststof bevestigde de resultaten.[18]

In verder onderzoek van Rai, Saxena et al. [17] wordt hetzelfde principe toegepast op elementen die een initiële kromming kunnen hebben, wat de mogelijkheden van de mechanische respons sterk vergroot. Als ontwerpvariabelen worden hiervoor de hoeken gebruikt die het element aan beide eindpunten maakt met hun verbindingslijn.

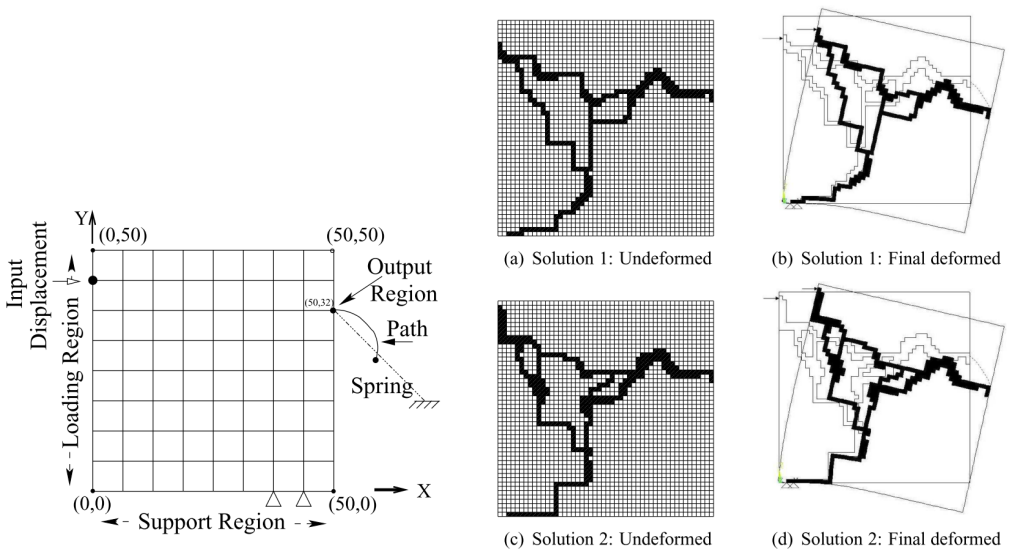
Verder is het kleinste kwadraten criterium vervangen door *Fourier shape descriptors*. Op deze manier worden de vorm, grootte en oriëntatie van het pad niet gelijktijdig gefit, wat tot overbodige beperkingen leidt, maar worden deze afzonderlijk van elkaar beschouwd.[17]

In de implementatie van het GA wordt *roulette selection* toegepast, waarbij ouders gekozen worden met een kans gebaseerd op hun *fitness*. Vijf verschillende kruisings-

operatoren zijn actief. Elitisme wordt toegepast op een beperkte set vectoren, het toegepaste GA wordt niet vermeld maar zal dus zeker geen zuivere NSGA-II zijn. Ook hier werd in prototypes goede overeenkomst met de verkregen resultaten bekomen. Deze prototypes faalden echter aan het einde van de beweging, niet onverwacht bij gebrek aan falingscriterium in de optimalisatie.[17]

2.4.2 Synthese van flexibele mechanismen met rooster elementen

Sharma, Deb en Kishore[21] pasten reeds NSGA-II toe om topologieën te bekomen voor flexibele mechanismen met een minimale massa, die een voorgeschreven traject volgen door het aanleggen van een kracht. Afgezien van het verschil dat de opzet in deze masterproef eruit bestaat om te werken met een aangelegde druk in plaats van een aangelegde verplaatsing als input, kan in grote lijnen een gelijkaardige aanpak gevolgd worden.



(a) Ontwerpruimte (b) Voorbeeldoplossingen

Figuur 2.5: Figuren uit het werk van Sharma et al.[19]. (a) Voorstelling van de ontwerpruimte en (b) twee voorbeeldoplossingen die een opgegeven pad benaderen.

Het beschouwde optimalisatieprobleem heeft twee objectieven, het minimaliseren van de massa en het maximaliseren van de diversiteit, berekend door bitsgewijze vergelijking met een referentieontwerp. Het ontwerp moet een voorgeschreven traject volgen binnen een bepaalde procentuele afwijking en ook aan de spanning wordt een beperking opgelegd om haalbare ontwerpen te bekomen.[21]

In figuur 2.5 (a) is de ontwerpruimte te zien van het probleem dat met behulp van niet-lineaire FE analyse en NSGA-II opgelost wordt. Er werd gewerkt met een 25x25 rooster waar een 1-bit de aanwezigheid van materiaal voorstelt en een 0-bit

lege ruimte. Het FE rooster bevat 4 elementen per cel uit deze voorstelling. De inklemming en inputverplaatsing kan op verschillende posities van een bepaalde zone aangebracht worden. Ook deze informatie zit in het DNA vervat. Samen met de 625 bits die de structuur voorstellen, coderen 5 bits de inklemmingspositie en 3 bits de inputpositie. In het outputpunt wordt een veer gemodelleerd zodat het bewegend mechanisme weerstand ondervindt.[21]

De structuren worden onderworpen aan een connectiviteitsanalyse die materiaalclusters zoekt in de input-, output-, en inklemmingszone en hun verbondenheid test. Geïsoleerde clusters worden verwijderd. Waar hoeken elkaar raken wordt materiaal toegevoegd en kleine holtes worden opgevuld.[21]

Als eerste stap wordt een optimalisatie uitgevoerd met enkel de minimalisatie van de massa als objectief. Het resultaat hiervan is het referentieontwerp. Dit wordt in een tweede optimalisatie gebruikt om het tweede objectief, de diversiteit, te berekenen. Berekeningen worden steeds zo veel mogelijk parallel uitgevoerd met behulp van een MPI (*Message Passing Interface*) gebaseerde Linux cluster en ANSYS FEA.[21]

Als parameters voor NSGA-II werd een populatie van 240, een maximum van 100 generaties, een kruisingskans van 95% en een mutatiekans van $1/n$ met n het aantal bits in de structuur gekozen. Tweedimensionale *cross-over* wisselt een willekeurig aantal kolommen of rijen, startend op een willekeurige positie.[21]

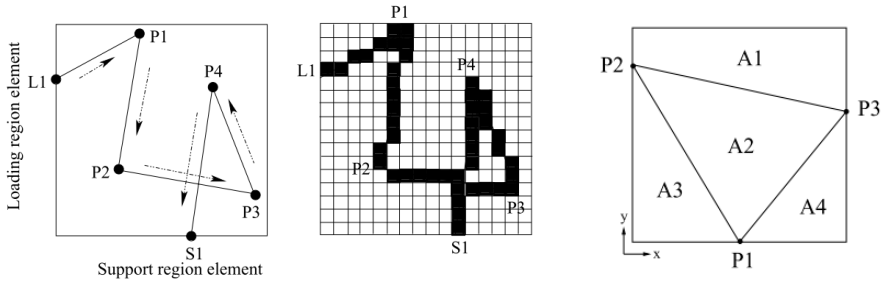
Na de *multi-objective* optimalisatie worden enkele representatieve oplossingen uit het eerste niet-gedomineerde front gekozen en onderworpen aan lokaal zoeken door middel van een variatie op het klassieke *hill climbing*, met als objectieffunctie een gewogen gemiddelde van de eerdere objectieven. Gezien het aantal haalbare Pareto-optimale oplossingen bijna even groot is als de populatie na 100 generaties, worden de oplossingen eerst geclusterd. Tijdens het lokaal zoeken wordt telkens één bit gemuteerd. Als het resultaat de beperkingen niet schendt en een betere *fitness* heeft, wordt de mutatie behouden. Ook de 8 aangrenzende bits worden telkens achtereenvolgens gemuteerd. Dit proces wordt herhaald tot alle bits minstens één keer bezocht zijn. Als de *fitness* niet meer verbetert, wordt het getermineerd.[21]

In verder onderzoek werd de minimalisatie van de inputenergie opgenomen als objectief en werd de maximalisatie van de diversiteit weggelaten. Verder werden aan het DNA 4 bits toegevoegd die de grootte van de inputverplaatsing coderen. Er werd een optimalisatie uitgevoerd voor verschillende waarden van de toegelaten afwijking van het gewenste pad. Waar er resultaten bekomen werden, waren deze vergelijkbaar en vulde de populatie zich langzaam met duplicaten van deze pareto-optimale oplossingen.[19]

In figuur 2.5 (b) zijn twee voorbeeldoplossingen te zien van een optimalisatie uit het onderzoek van Sharma et al.[19].

In een derde paper van Sharma et al.[22] wordt het genereren van een beginpopulatie verbeterd door een 'initiële populatie strategie'. In plaats van willekeurige binaire strings te produceren, worden de drie belangrijke regio's verbonden met een variabel aantal lijnsegmenten, die op het rooster gediscretiseerd worden. Figuur 2.6 (a) illustreert het proces. Deze strategie leverde resultaten die de oplossingen met willekeurige initialisatie domineerden en de tweedimensionale resultatenruimte beter benutten.[22]

Ook een specifieke kruisingsoperator werd onderzocht door Sharma et al.[20]. In figuur 2.6 (b) is te zien hoe de operator in zijn werk gaat. Op drie zijden van de topologie wordt een willekeurig punt gekozen, zodat de verbindingslijnen de ontwerpruimte in vier domeinen indelen. Bij kruising wordt voor elk van deze domeinen met 50% kans besloten of ze uitgewisseld worden.[20]



(a) Initiële populatie strategie

(b) Domein-specifieke kruising

Figuur 2.6: Figuren uit het werk van Sharma et al. (a) Werking van de initiële populatie strategie[22] en (b) illustratie bij de specifieke kruisingsoperator[20].

Hoofdstuk 3

Implementatie van de onderdelen

Dit hoofdstuk heeft als doel de implementatie uit te werken van de onderdelen die nodig zijn om een programma te ontwikkelen voor het beoogde onderzoek, gebruik makend van NSGA-II en FE software.

Het programma zal starten met het genereren van een beginpopulatie en in elke 'generatie' of iteratie van het genetisch algoritme een aantal vaste stappen doorlopen. Vectoren worden geselecteerd als ouders, hun genetisch materiaal wordt gekruist en ondergaat mutatie. Dit nieuw genetisch materiaal stelt een nakomeling voor en wordt gecontroleerd op fouten. Als het DNA geldig is, wordt voor elk objectief een *fitness* toegekend aan deze nieuwe vector, voor de verplaatsing gebeurt dit aan de hand van het resultaat van een FEM simulatie. Tot slot wordt een nieuwe populatie opgebouwd uit de beste vectoren van de vorige populatie en het nageslacht.

Dit hoofdstuk bespreekt eerst enkele ontwerpbeslissingen voor de eigen code, behandelt de nodige testen om DNA te controleren op fouten en ontwikkelt een aanpak om een beginpopulatie te initialiseren door generatie en herstelmethoden. Verder komt aan bod hoe de communicatie met de FEM software verloopt en wordt de implementatie van NSGA-II besproken.

3.1 Architectuur van het programma

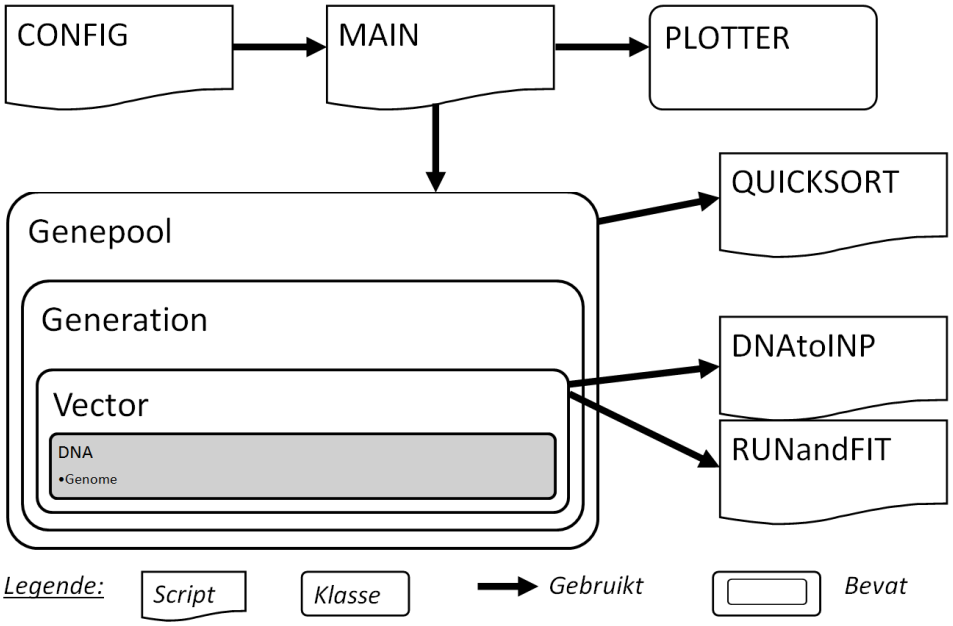
Als eindige elementen software werd gekozen voor Abaqus, een standaard op het vlak van FEM in de industrie. Abaqus is een software omgeving voor *Computer Aided Engineering* en eindige elementen berekeningen, ontwikkeld door *Dassault Systèmes.*[1]

Als programmeertaal voor de eigen modules werd gekozen voor Python[2], een populaire *high level* programmeertaal, vanwege de goede compatibiliteit met Abaqus.

Python scripts kunnen uitgevoerd worden in de Abaqus softwareomgeving. De Python code is bijgevoegd in appendix G

In Python worden de modules uitgewerkt die NSGA-II implementeren als genetisch algoritme, die een structuur van klasses voorzien om in de optimalisatie te gebruiken, die controles implementeren, generatiemethodes en een parser om DNA naar Abaqus-compatibele invoerbestanden te vertalen.

In figuur 3.1 is een schematische voorstelling van alle Python bestanden te zien. Een config bestand bevat alle parameters en start het hoofdscript. Dit maakt een *Genepool*-klasse aan die instaat voor het genetisch evolutieproces en de ouder- en nageslacht generaties bevat. Deze generatieklasses bevatten vector objecten, die zelf een instantie van de DNA-klasse met alle genetische informatie bevatten. De klasse vector zelf bevat de *fitness* gegevens en maakt gebuik van twee scripts om Abaqus bestanden te schrijven (DNAtoINP) en simulaties uit te voeren (RUNandFIT).



Figuur 3.1: Klassestructuur van de ontwikkelde Pythoncode.

3.2 DNA controletests

Niet elke willekeurige reeks bits kan een adequate voorstelling van een doorsnede van een pneumatische actuator zijn. Er zijn enkele voorwaarden waar aan voldaan moet zijn, opdat een reeks bits of DNA in een bruikbare structuur vertaald zou kunnen worden. Zo moet de dimensie overeenkomen met de lengte van het genoom, moet het materiaal één aaneengesloten deel vormen met een afgesloten holte voor het aanleggen van druk en mogen er geen puntcontacten zijn die spannngsconcentraties veroorzaken en zouden scheuren.

Topologieën die niet aan de voorwaarden voldoen, worden nooit opgenomen in de genetische optimalisatie omdat ze nooit aan de objectieven kunnen voldoen. Ze zouden fouten kunnen veroorzaken in de FE simulaties en zijn in vele gevallen zelfs niet realiseerbaar in werkelijkheid.

Afgezien van de controle van de dimensie, die met een assert-statement wordt afgedwongen (de code geeft een foutmelding en stopt indien niet aan de voorwaarde voldaan is), zorgt het falen van een controle op het genoom dat de variabele `self.valid` van het DNA naar `False` wordt veranderd. Voor meer informatie over de implementatie van deze tests, zie de `DNA` klasse in appendix F en G.

De volgende controles moeten steeds uitgevoerd worden om te garanderen dat een gegeven DNA aan de voorwaarden voldoet.

3.2.1 Controle van de dimensies

De grootte van de tweedimensionale vorm moet uiteraard overeenkomen met het aantal bits in het DNA. Deze test is vrij triviaal, maar beschermt de software tegen foutieve invoer.

3.2.2 Controle op puntcontacten

In het geval dat twee blokjes materiaal in de topologie elkaar enkel met hun hoeken raken, is de structuur niet bruikbaar. Zulke puntcontacten zijn niet mogelijk omdat ze in berekeningen enorme spanningsconcentraties zouden geven en in werkelijkheid meteen zouden falen.

De structuur moet dus doorlopen worden om dergelijke puntcontacten op te sporen. Eerder dan een genoom met puntcontacten te verwerpen, wordt in de huidige implementatie de fout hersteld zoals verder beschreven in deel 3.3.4 en figuur 3.5. (Zie `DNA.rupture_test()` in de appendices F en G.)

3.2.3 Controle van de samenhang

Indien er verschillende clusters materiaal gedetecteerd worden in de voorstelling die niet met elkaar verbonden zijn, kan het DNA niet gebruikt worden om als ontwerp voor de doorsnede van een actuator te dienen.

In een kopie van het genoom worden, startende vanaf de eerste 1-bit, recursief de vier aangrenzende cellen gecontroleerd en worden alle 1-bits veranderd in 0-bits. Als dit het genoom niet verandert in een nulmatrix, moet het DNA verworpen worden. (Zie `DNA.consistency_test()` in de appendices F en G.)

3.2.4 Controle van de opblaasbaarheid

Indien zich nergens in de structuur een afgesloten holte bevindt, is het uiteraard geen voorstelling van een actuator.

Op een analoge recursieve manier als voor de samenhang wordt het genoom doorlopen en worden alle aansluitende gebieden van 0-bits aangeduid met een specifiek nummer of label en wordt bijgehouden hoe groot ze zijn en of ze een echte holte of een inham in de topologie zijn. Als er in de lijst met holtes geen enkele holte opblaasbaar is, is het genoom niet bruikbaar en moet het verworpen worden. De code is te vinden in appendix G als `DNA.map_holes()` en `DNA.inflatability_test()`.

3.3 Initialisatie van een beginpopulatie

Om het genetisch algoritme te kunnen starten, moet er allereerst een startpopulatie van vectoren aangemaakt worden met voldoende genetische variatie. Hier volgen enkele mogelijke pistes die gevolgd kunnen worden om DNA-strings te genereren, die in de beginpopulatie opgenomen kunnen worden indien ze de controles doorstaan. Ook herstelmethodes zullen besproken worden, die pogen om het nieuw gegenereerde DNA dat niet aan de controles voldeed alsnog aan de controles te laten voldoen.

Alvorens tot de implementatie over te gaan, worden eerst vier mogelijke generatie- en herstelmethodes uiteengezet.

3.3.1 Mogelijke generatiemethoden voor DNA

Hier volgt een lijst met denkbare generatiemethodes, deze is allesbehalve exhaustief. Zowel willekeurige als gerichte manieren van aanpak worden besproken.

Puur willekeurig

Willekeurige bits genereren is de meest voor de hand liggende manier om de benodigde variatie te garanderen. Deze methode zal dus het gewenste aantal bits aanmaken met een random-functie met een gelijke kans op een 0 of een 1.

Gewogen willekeurig

Een tweede mogelijkheid is een meer verfijnde generatiemethode die er rekening mee kan houden dat men niet noodzakelijk structuren met 50% materiaal en 50% open ruimte wil genereren. Er worden nog steeds random bits gegenereerd, maar nu hebben 0 en 1 geen gelijk gewicht in de random-functie.

Deelblokken

Grotere structuren genereren die aan alle voorwaarden voldoen, is moeilijker dan kleinere door onder andere de grotere kans op puntcontacten. Hierop kan ingespeeld

worden door grote topologieën samen te stellen uit kleinere, die sneller gegenereerd kunnen worden en bij combinatie enkel op de scheidingslijnen fouten kunnen veroorzaken.

Of deze methode in zijn geheel sneller is dan de random methode moet uiteraard experimenteel nagegaan worden. Om een voorbeeld te geven: een 16x16-matrix kan gemaakt worden uit vier 8x8-matrices of zestien 4x4-matrices. Als men zestien van deze 4x4-matrices heeft gevonden, kan men een 16x16-matrix samenstellen. Het is echter ook mogelijk om meteen zestien grote te genereren door de kleine onderling te permuteren. Dit doet geen afbreuk aan de variatie, want kruisingen wisselen enkel overeenkomstige regio's uit.

Paden discretiseren

In contrast met de willekeurige generatie, kan er ook doelbewust een structuur gegenereerd worden die met een grote kans aan de voorwaarden voldoet. In het onderzoek van Sharma et al. [22] dat in sectie 2.4.2 aan bod kwam, wordt een dergelijke methode beschreven om een flexibele structuur te genereren die een aangelegde kracht vertaalt in een gewenste beweging. Dit gebeurt door de inputzone van de kracht te verbinden met de eindeffector en het steunpunt met een aantal lijnsegmenten (een willekeurig aantal van één tot vijf, ook de coördinaten van de knopen worden willekeurig bepaald). Deze lijnen worden dan gediscretiseerd: de cellen die doorkruist worden door een lijn krijgen materiaal toegewezen, de cellen waar geen lijn door loopt blijven leeg.

Dit onderzoek zou een gelijkaardige techniek kunnen toepassen die garandeert dat de structuur samenhangend en opblaasbaar is. Een mogelijke implementatie hiervan is tussen het steunpunt en de eindeffector een aantal van deze paden te maken, met een minimum van twee. In dat geval is de structuur samenhangend en is er in de grote meerderheid van de gevallen een opblaasbare holte gecreëerd tussen deze paden. Variatie kan verzekerd worden door ook hier het aantal paden en lijnsegmenten per pad willekeurig te kiezen tussen bepaalde grenzen. Puntcontacten worden hiermee echter niet voorkomen.

3.3.2 Herstelmethoden om het genereren van DNA te versnellen

Met eenvoudige correcties kunnen ongeldige genomen aangepast worden zodat ze alsnog aan de voorwaarden voldoen, dit zijn enkele mogelijke strategieën.

Regio's verbinden

De eerste methode wordt ook toegepast in het onderzoek van Sharma et al. [19] en garandeert de samenhang van de structuur. Wanneer de cluster materiaal in de drie regio's niet samenhangen, wordt een verbindingslijn tussen hun zwaartepunten getrokken en gediscretiseerd op het rooster.

In het geval van dit onderzoek zouden op die wijze de steun- en outputregio verbonden kunnen worden, maar de aanpak met een rechte lijn is vermoedelijk te eenvoudig omdat deze geen opblaasbare holte kan introduceren, deze moet dan reeds aanwezig zijn in één van de clusters om een bruikbare structuur te genereren. Beter zou zijn om meerdere paden te tekenen, maar als herstelmethode is dit omslachtig, gezien het meteen als generatiemethode kan toegepast worden zoals eerder beschreven. Hier zal dus ook niet verder op in gegaan worden.

Puntcontacten versterken

Deze snelle en efficiënte herstelmethode wordt in dezelfde paper [19] beschreven als de vorige methode, beiden worden in volgorde van vermelding toegepast. Als in een 2x2 regio van de structuur twee cellen gevuld zijn met materiaal die diagonaal ten opzichte van elkaar staan, is het voldoende om een derde cel te vullen om het puntcontact weg te werken.

Materiaal wegnemen

Als er clusters materiaal voorkomen die niet verbonden zijn met het steunpunt, kunnen deze verwijderd worden. Dit is enkel toepasbaar als de outputregio niet vooraf vast bepaald is en zal bovendien veel kleine structuren opleveren die maar een deel van het ontwerpdomein benutten. Erg veelbelovend is deze methode dus niet.

Gaten introduceren of holtes sluiten

Volledig tegengesteld aan de herstelmethode van Sharma et al. [19], waar de gaten die omgeven zijn met acht gevulde cellen opgevuld worden voor de stevigheid, zou men in regio's met voldoende materiaal gaten kunnen introduceren. Alternatief kunnen inhammen in het materiaal afgesloten worden tot volwaardige holtes.

3.3.3 Implementatie van een generatiemethode

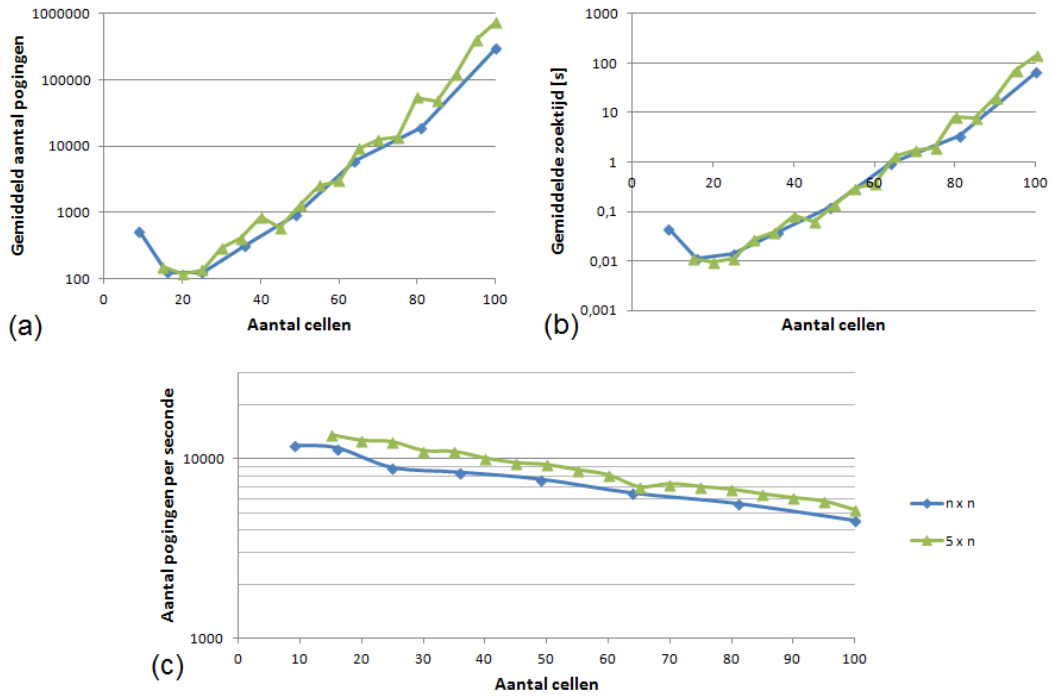
Om een werkende solver te creëren, is het noodzakelijk één van de eerder genoemde generatiemethodes te implementeren, eventueel vergezeld van een herstelmethode.

Puur willekeurig

Gezien de puur willekeurige methode het eenvoudigst te implementeren is, wordt er met deze methode gestart. Veel meer dan de Pythonfunctie `random.getrandbits()` is hiervoor niet nodig.

Om na te gaan aan welke snelheid deze methode DNA kan genereren, werd voor toenemende groottes van NxN-matrices en NxM-matrices telkens een populatie van tien DNA objecten gemaakt. Het resultaat is te zien in figuur 3.2, uitgezet in x-log(y) diagrammen met basis 10. De gemiddelde waarden van de tijd nodig om een geldig DNA te maken, of het gemiddeld aantal pogingen dat nodig was, wordt telkens

weergegeven. In bijlage C is een uitgebreidere verzameling grafieken terug te vinden die de prestaties van deze methode aangeven, voor elke meting is er in een aparte grafiek te vinden, ook zijn telkens het minimum, het maximum en de gemiddelde waarde opgelijst voor alle groottes en metingen.



Figuur 3.2: Vergelijking van het NxN en NxM geval voor het genereren van geldige vectoren. (a) Gemiddeld aantal pogingen, (b) Gemiddelde zoektijd, (c) Gemiddeld aantal pogingen per seconde, telkens in functie van aantal cellen of DNA-bits van de vector.

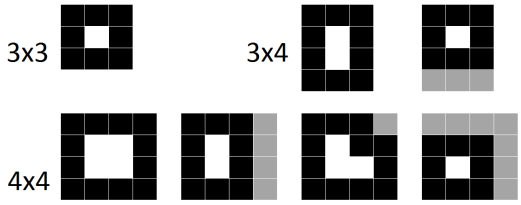
Uit figuur 3.2 kunnen enkele belangrijke conclusies getrokken worden. Om te beginnen is het aantal pogingen en de rekentijd die nodig is vooral afhankelijk van het aantal cellen in een matrix, dat is te zien in deelfiguur (a) en (b) waar de resultaten voor een NxN- en NxM-matrix uitgezet zijn in functie van het aantal cellen. Voor NxM-matrices lijken de waarden net iets hoger te liggen op het einde. Dit kan een toevalligheid zijn, maar kan ook het gevolg zijn van de minder compacte vorm, waardoor aan de voorwaarde van samenhang van het materiaal moeilijker voldaan wordt.

De grafieken (a) en (b) tonen een vrij rechte lijn, wat duidt op een exponentiële stijging. Een opmerkelijk effect dat wat toelichting verdient, is het feit dat de grafieken aanvankelijk dalen, om dan terug te stijgen. Dit is eenvoudig verklaarbaar omwille van de voorwaarde van opblaasbaarheid. Voor zeer kleine structuren is het aantal configuraties die een holte bevatten zeer klein, waardoor het percentage van alle mogelijke combinaties dat aan alle voorwaarden voldoet zeer laag wordt voor

kleine matrices. Deze aantallen en percentages worden gegeven in tabel 3.1. Ter illustratie zijn in figuur 3.3 ook de toegelaten combinaties weergegeven. De zwarte en witte cellen zijn vast bepaald, enkel voor de grijze cellen is de keuze van de bitwaarde vrij. 3x4-Matrices kunnen verder ook om één as gespiegeld worden, 4x4-matrices om twee assen of geroteerd (dit komt beide op hetzelfde neer en verviervoudigt het aantal mogelijkheden).

Afmetingen	# Geaccepteerd	# Totaal	Procentueel
3 x 3	1	512	0,20%
3 x 4	17	4096	0,42%
4 x 4	585	65536	0,89%

Tabel 3.1: Aantal mogelijke combinaties, aantal geaccepteerde combinaties en de onderlinge verhouding voor zeer kleine topologieën.



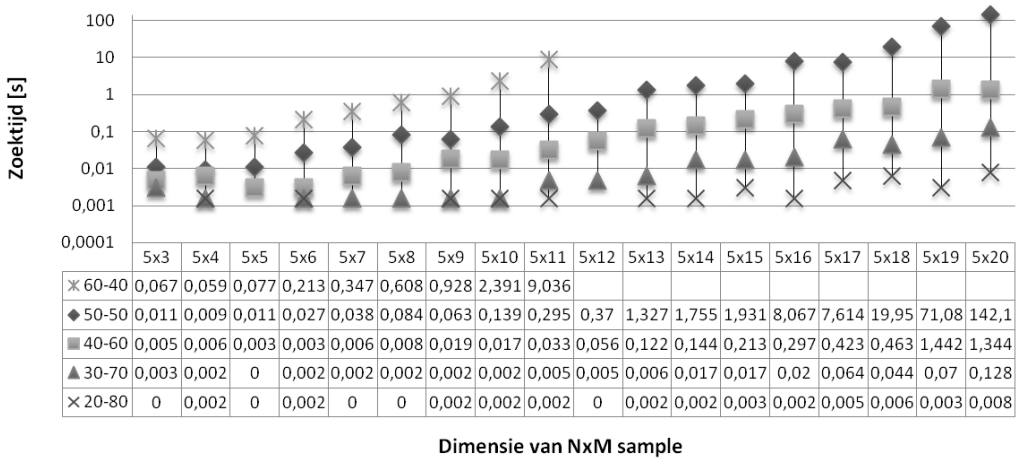
Figuur 3.3: Overzicht van de geaccepteerde combinaties voor kleine topologieën. Zwarte bits zijn 1, witte 0, grijze vrij te kiezen.

Zoals eerder gezegd, neemt de rekentijd exponentieel toe, dit is omdat het nodige aantal pogingen exponentieel toeneemt. Uit de uitgebreide data blijkt dat het aantal pogingen per seconde voor toenemende grootte nauwelijks afneemt, dit is te zien in figuur 3.2 (c). Het aantal pogingen per seconde neemt voor deze data maar af met een factor twee, terwijl de laagste en hoogste waarden voor rekentijd en benodigde pogingen met een factor tienduizend verschillen! Dit zou geen probleem hoeven te vormen als de waarden laag genoeg bleven. Voor 10x10-matrices worden rekentijden opgemeten die gemiddeld meer dan een minuut bedragen voor één enkel DNA-object. Gezien er een grote populatie vereist zal zijn en 10x10 nog steeds een lage resolutie is, is dit een onacceptabel resultaat. Deze methode voldoet niet aan de vereisten.

Tot slot kan uit deze grafiek ook geconcludeerd worden dat het NxM geval en het NxN geval goed overeenkomen, maar dat bij het NxM geval in dezelfde hoeveelheid tijd resultaten voor meer meetpunten kunnen bekomen worden. Omwille van de hogere nauwkeurigheid door het beschikken over meer meetpunten, zal verder enkel het NxM geval behandeld worden. Ook zullen enkel nog rekentijden weergegeven worden, omdat het deze tijden zijn die tot een redelijke waarde moeten beperkt worden. Data voor het aantal pogingen en voor het NxN geval zijn wel in bijlage C te vinden.

Gewogen willekeurig

De eenvoudigste manier om dit probleem te verhelpen, is de methode van gewogen willekeurige generatie. Als meer dan 50% van de ruimte door materiaal wordt ingenomen, wordt de kans op puntcontacten kleiner en de kans dat alles één samenhangend geheel vormt groter. In figuur 3.4 en C.5 is dit effect te zien. De figuur is logaritmisch met basis e en de labels # # - # # moeten gelezen worden als %open-%materiaal. Voor vijf scenario's worden de gemiddelde zoektijden gegeven voor het aanmaken van tien strings DNA, variërend van 20% vrije ruimte tot 60% vrije ruimte. Zoals verwacht, versnelt het algoritme behoorlijk, zelfs al wanneer men de vrije ruimte met 10% doet toenemen. De implementatie van deze methode gebeurde met twee regels code: eerst wordt een verzameling aangemaakt die de percentages vertegenwoordigt, bijvoorbeeld `probability = [0]*4 + [1]*6` voor 40% open ruimte en 60% materiaal. Vervolgens wordt hieruit willekeurig gekozen met `random.choice(probability)`. Het aanpassen van de materiaalverhouding heeft helaas invloed op de variatie binnen de populatie en zal het nodige aantal generaties om naar een oplossing te convergeren, die veel meer vrije ruimte bevat, doen toenemen. Een neutrale materiaal/open ruimte-verhouding blijft gewenst.



	5x3	5x4	5x5	5x6	5x7	5x8	5x9	5x10	5x11	5x12	5x13	5x14	5x15	5x16	5x17	5x18	5x19	5x20
60-40	0,067	0,059	0,077	0,213	0,347	0,608	0,928	2,391	9,036									
50-50	0,011	0,009	0,011	0,027	0,038	0,084	0,063	0,139	0,295	0,37	1,327	1,755	1,931	8,067	7,614	19,95	71,08	142,1
40-60	0,005	0,006	0,003	0,003	0,006	0,008	0,019	0,017	0,033	0,056	0,122	0,144	0,213	0,297	0,423	0,463	1,442	1,344
30-70	0,003	0,002	0	0,002	0,002	0,002	0,002	0,002	0,005	0,005	0,006	0,017	0,017	0,02	0,064	0,044	0,07	0,128
20-80	0	0,002	0	0,002	0	0	0,002	0,002	0,002	0	0,002	0,002	0,003	0,002	0,005	0,006	0,003	0,008

Figuur 3.4: De zoektijd om een geldige vector te bekomen, in seconden, voor verschillende materiaalverdelingen en groottes.

Het is aangewezen om eerst herstelmethodes uit te proberen alvorens over te gaan naar het implementeren van ingewikkelde gerichte generatiemethodes.

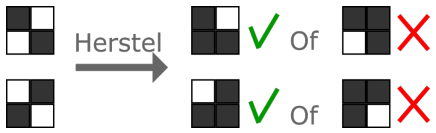
3.3.4 Implementatie van een herstelmethode

In dit deel wordt nagegaan of één van de voorgestelde herstelmethodes in combinatie met een eenvoudige generatiemethode aan de vereisten voldoet.

Puntcontacten versterken

Uit alle mogelijke problemen die er met een genoom kunnen optreden waardoor het niet aan de voorwaarden voldoet, is de kans dat er ergens een puntcontact optreedt waarschijnlijk de grootste. De kans op losse clusters materiaal is klein in niet al te grote structuren. De kans dat er geen holte aanwezig is, is enkel in zeer kleine structuren aanzienlijk en neemt af met de grootte. Maar elk 2x2 blokje dat in de structuur beschouwd wordt, kan onafhankelijk van de geldigheid en grootte van de rest van de structuur een puntcontact bevatten.

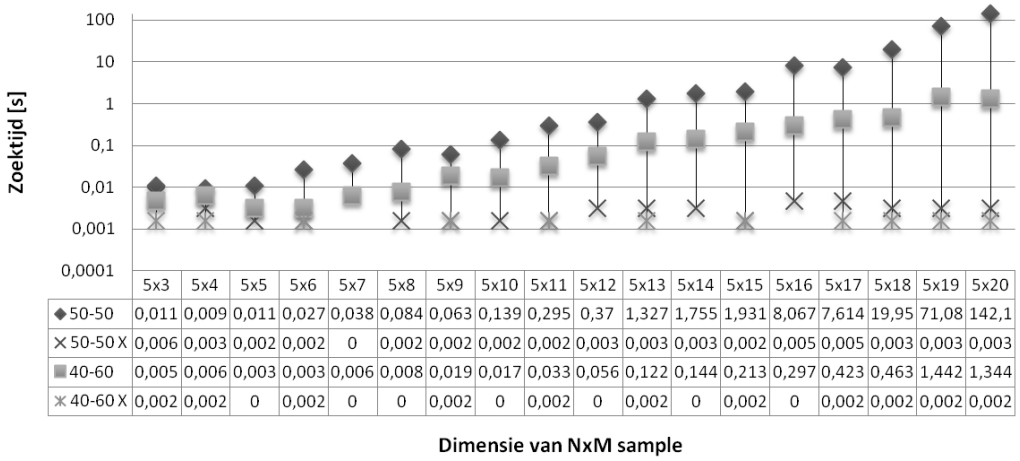
Dit probleem verhelpen zou dus al een aanzienlijke versnelling kunnen betekenen en bovendien is het niet moeilijk om deze methode te implementeren. Daarvoor moeten er enkele aanpassingen in de controle op puntcontacten gebeuren. Wanneer een puntcontact gedetecteerd wordt en de methode het DNA zou afkeuren, moet een aanliggende cel met materiaal gevuld worden en moet het algoritme verder zoeken naar het volgende puntcontact en dit tot de volledige structuur overlopen is. Voor de duidelijkheid is deze eenvoudige methode voor beide gevallen van een puntcontact weergegeven in figuur 3.5. Van de twee denkbare manieren om de zone te versterken, is er telkens één aangewezen, namelijk degene die materiaal toevoegt op de onderste rij van de 2x2-matrix. Dit is zo omdat een eventueel nieuw puntcontact dat hiermee geïntroduceerd zou kunnen worden ook door het algoritme gedetecteerd zou worden. Past men de bovenste rij aan, kan er een puntcontact optreden met de rij erboven, een 2x2-matrix die reeds gecontroleerd was. Het algoritme zal dus steeds de onderste rij aanpassen.



Figuur 3.5: Grafische voorstelling van het herstellen van puntcontactfouten.

In figuur 3.6 en C.6 is een vergelijking te zien tussen generatie met en zonder herstel van puntcontacten, voor het geval van 50% open ruimte en van 60% open ruimte. Het resultaat is duidelijk zeer goed, voor beide gevallen met een herstelmethode blijven de gemiddelde zoektijden beperkt tot de precisie van de tijdsmeting, duizendsten van een seconde.

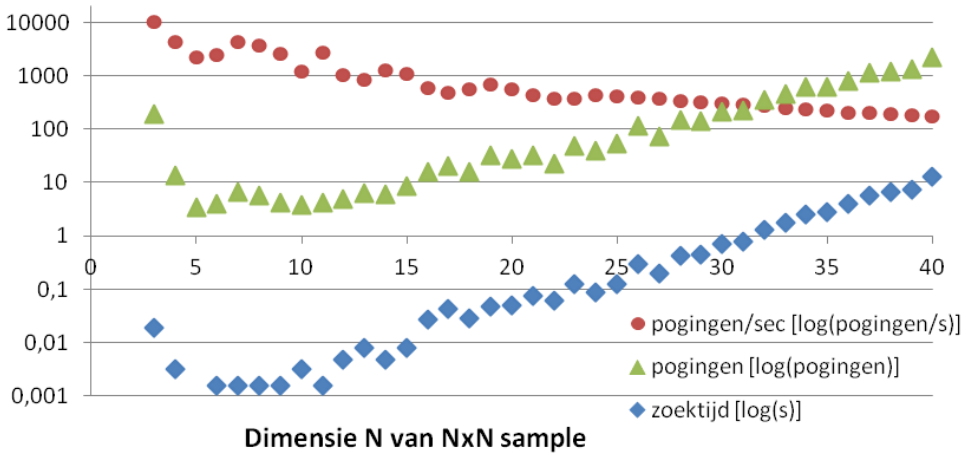
Gezien de opvallend hogere versnelling, is het aangewezen ook voor grotere structuren uit te rekenen hoe de zoektijd evolueert. Omdat het aantal meetpunten in dat geval hoger wordt, kan terug overgeschakeld worden naar NxN-matrices zonder het nadeel van het beperkte aantal meetpunten. In figuur 3.7 zijn alle resultaten van deze simulatie gegeven, zoals in de vorige simulaties zijn dit gemiddeldes voor tien pogingen. (Afzonderlijke resultaten zijn te vinden in figuur C.7 en C.8.) De benodigde zoektijd, het gemiddeld aantal pogingen en het gepresteerd aantal pogingen per seconde zijn weergegeven.



	5x3	5x4	5x5	5x6	5x7	5x8	5x9	5x10	5x11	5x12	5x13	5x14	5x15	5x16	5x17	5x18	5x19	5x20
◆50-50	0,011	0,009	0,011	0,027	0,038	0,084	0,063	0,139	0,295	0,37	1,327	1,755	1,931	8,067	7,614	19,95	71,08	142,1
×50-50 X	0,006	0,003	0,002	0,002	0	0,002	0,002	0,002	0,002	0,003	0,003	0,003	0,002	0,005	0,005	0,003	0,003	0,003
■40-60	0,005	0,006	0,003	0,003	0,006	0,008	0,019	0,017	0,033	0,056	0,122	0,144	0,213	0,297	0,423	0,463	1,442	1,344
✳40-60 X	0,002	0,002	0	0,002	0	0	0,002	0	0,002	0	0,002	0	0,002	0	0,002	0,002	0,002	0,002

Figuur 3.6: Gemiddelde zoektijd bij generatie met en zonder herstel van puntcontacten, voor het geval van 50% open ruimte en 60% open ruimte.

Het aantal pogingen per seconde (rood), dat vroeger van tienduizend voor 3x3 naar vijfduizend voor 10x10 evolueerde, daalt nu van zo'n tweeduizend voor 10x10 naar tweehonderd voor 40x40. Dit is lager dan voorheen omdat er meer operaties moeten uitgevoerd worden voor het herstel. Ook de daling is meer uitgesproken, maar op zich niet alarmerend gezien de absolute tijden (blauw) beperkt blijven tot een gemiddelde van twaalf seconden. Dit is een zeer goede prestatie en betekent dat deze methode verder gebruikt kan worden. Een ontwerpdomein van 40x40 kan al meer dan voldoende gedetailleerde structuren beschrijven, terwijl het genereren van een populatie voor deze grootte duidelijk nog steeds een klein deel van de totale rekentijd zal vergen.



Figuur 3.7: Overzicht van het gemiddeld aantal pogingen, de gemiddelde zoektijd, en hun onderlinge verhouding voor willekeurige generatie met herstel.

Bemerk dat door het implementeren van de herstelmethode op het model met 50% materiaal in de willekeurige generatie, de werkelijke hoeveelheid materiaal in de doorsnede toeneemt. Indien deze toename rond de 10% bedraagt, kan het interessant zijn om een gewogen willekeurige generatie met gemiddeld 40% materiaal te gebruiken, zodat er een doorsnede met 50% materiaal bekomen wordt.

3.4 Vertalen van de DNA voorstelling naar een Abaqus input file

In deze sectie wordt uitvoeriger toegelicht hoe de Abaqus input file wordt opgebouwd, vertrekkende van een DNA-object en een reeks parameters. Het voorbeeld in appendix E kan hier nuttig bij zijn.

Abaqus is FEM software waarin de gebruiker met een GUI (*Graphical User Interface*) het beschouwde probleem tekent, een FE rooster definieert en randvoorwaarden aanlegt om het gedrag in deze omstandigheden te simuleren. Abaqus vertaalt zelf deze getekende voorstelling naar een bondig inputbestand met een vaste structuur dat het gestelde probleem ondubbelzinnig beschrijft. Van deze aanpak maakt de gerealiseerde *parser* gebruik door de topologie en de gekozen parameters rechtstreeks te vertalen naar een dergelijk bestand, zonder dat er een gebruiker of GUI aan te pas moet komen.

De input file bestaat uit de volgende delen:

Heading: Hier moet de job name in enkele standaard lijnen ingevoegd worden.

PARTS: Hier komen de knopen, de elementen en sets met alle elementen, alle knopen en de outputknoop.

ASSEMBLY: Hier moeten de oppervlakken gedefiniëerd worden waarop de druk werkzaam zal zijn, evenals de set van ingeklemde knopen.

MATERIALS: Hier komen de materiaalparameters.

STEP: Standaard waarden voor berekeningsstappen.

BC: Specificeert een randvoorwaarde op een knopenset.

LOADS: Legt de opgegeven druk aan op de juiste oppervlakken.

OUTPUT: Standaard code.

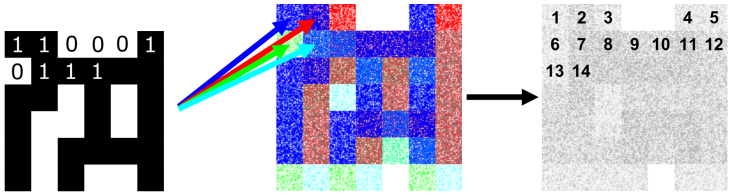
Een groot deel van het input bestand bestaat dus uit standaard lijnen waar de waarden van parameters zoals de druk, het soort elementen en het materiaalmodel moeten ingevoegd worden.

Voor het aanleggen van de belasting is inklemming aangewezen. Deze wordt aangelegd op de knopen aan de linkerzijde en bovenzijde van het eerste element (bovenaan, links). De verplaatsing wordt uitgelezen in de laatste knoop. Door de oplossing te roteren, kunnen ook eenvoudig andere types problemen verkregen worden. Indien gewenst zou ook het inklemmen van een volledige zijde eenvoudig te implementeren zijn.

De grootste moeilijkheid bestaat erin om de lijst binaire waarden van het genoom om te zetten in een lijst knopen, elementen en de oppervlakken te definiëren waarop de druk werkzaam is. Om te beginnen wordt in de parser klasse het genoom omgezet van een lijst in een tweedimensionale matrix-voorstelling. Om rekening te houden met de opgegeven fijnheid van het FEM-rooster (`grid_refinement`), worden deze matrices ook herschaald tot grotere kopieën, zodat één positie in de matrix nu overeenkomt met één FEM-element. Ook de 2D-map van de holtes, die reeds gegenereerd is bij het testen van de opblaasbaarheid, wordt herschaald voor gebruik in de parser.

3.4.1 Definiëren van knopen

Allereerst moeten alle knopen gedefiniëerd worden in het input bestand. Een knoop bestaat als minstens één van de vier omliggende elementen bestaat. De eenvoudigste manier om na te gaan of een knoop bestaat, is dus een (X+1)x(Y+1)-matrix aan te maken. Daarin wordt de oorspronkelijke voorstelling vier maal gekopieerd: zonder verschuiving, één kolom verschoven naar rechts, één rij verschoven naar beneden en diagonaal verschoven naar rechtsonder. Zo wordt een kaart bekomen van alle elementen. Enkel wanneer een cel de waarde nul bevat, is de knoop niet actief in de FEM-voorstelling. Vervolgens worden alle actieve knopen genummerd van links naar rechts, rij per rij. Figuur 3.8 geeft een voorstelling van dit proces. De code is terug te vinden in appendix G.

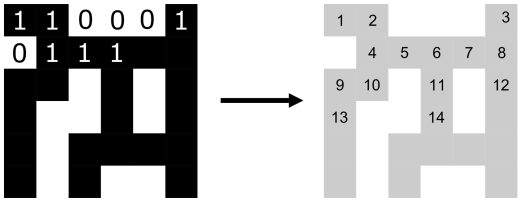


Figuur 3.8: Vier deels overlappende topologieën creëren een voorstelling van de knopen voor het Abaqus model.

In het input bestand moet een lijst komen van alle actieve knopen en hun coördinaten. Het relatief assenstel ligt hiervoor in de linkerbovenhoek: de x-as valt samen met de bovenrand, de y-as met de linkerzijde. Elke knoop krijgt dus een positieve x-coördinaat en een negatieve y-coördinaat, met als grootte de positie van de knoop in de voorstelling, vermenigvuldigd met $\frac{cell\_edge}{grid\_refinement}$ om de gewenste schaal te verkrijgen.

3.4.2 Definiëren van elementen

Het maken van een lijst met elementen is een stuk eenvoudiger dan het geval was voor de knopen. Uit de 2D-voorstelling kunnen ze meteen genummerd worden, zie figuur 3.9.



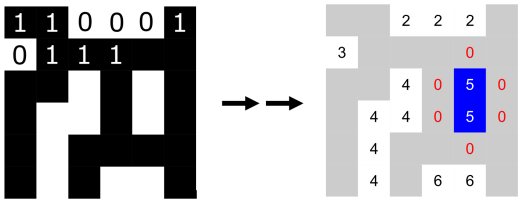
Figuur 3.9: De elementen worden genummerd door elke 1-waarde te vervangen door een oplopend cijfer.

Het input bestand vereist een lijst met elementen, met telkens een opsomming van de vier knopen die een hoekpunt van het element vormen. Hiervoor gaat de parser te werk zoals bij het bepalen van de knopen. Voor element $[i, j]$ zijn de nummers van de knopen te vinden in de 2D-map van de knopen op posities $[i, j]$, $[i+1, j]$, $[i+1, j+1]$ en $[i, j+1]$. Dit is de volgorde waarin ze moeten opgelijst worden; kloksgewijs rond het element.

3.4.3 De belaste oppervlakken opsommen

Met de gegevens die in de DNA-klasse aangemaakt zijn tijdens het testen van de opblaasbaarheid, kan het nummer bekomen worden van de grootste opblaasbare holte, waarop druk aangebracht zal worden. De oppervlakken die aan druk blootgesteld worden, moeten in het input bestand worden genoteerd als vier lijsten van elementen die een gedeelde belasting dragen op een specifieke zijde.

In de 2D-map van de holtes worden de cellen, die tot de gekozen holte behoren, opgezocht en wordt de waarde van alle aanliggende cellen gelijk gesteld aan nul, zoals voorgesteld in figuur 3.10. Alle elementen die met een nulcel overeenkomen, behoren tot de rand van de holte. Afhankelijk van de zijde waar het cijfer van de holte met druk voorkomt (dit kunnen meerdere zijdes zijn), wordt het overeenkomstige element bij de juiste set(s) gevoegd. Met deze vier sets wordt een SURFACE-element aangemaakt om de belasting op aan te leggen.



Figuur 3.10: De elementen worden genummerd.

3.5 Implementatie van het Genetisch algoritme

In deze sectie wordt beschreven hoe NSGA-II en de genetische operatoren in deze specifieke solver geïmplementeerd zijn. In appendix G is de eigenlijke python code te vinden. In dit hoofdstuk wordt de implementatie voorgesteld in pseudocode.

3.5.1 Onderdelen van NSGA-II

In deel 2.3.3 is beschreven hoe NSGA-II in zijn werk gaat. In dit deel wordt er verder ingegaan op hoe de code van de cruciale onderdelen opgebouwd is.

Hoofdfunctie

De hoofdlus van NSGA-II is vrij eenvoudig, ze combineert een ouderpopulatie met haar nakomelingen, sorteert ze, selecteert de beste helft als nieuwe populatie en maakt nieuwe nakomelingen. In pseudocode beschreven, gebeurt er het volgende:[6]

$R_t = P_t \cup Q_t$	Combineer populatie en nakomelingen
F = fast-nondominated-sort(R_t)	Sorteer in fronts $F = (F_1, F_2, ...)$
$P_{t+1} = \emptyset$	
tot $\lvert P_{t+1}\rvert + \lvert F_i\rvert \leq$ N	Vul de nieuwe populatie
crowding-distance-assignment(F_i)	Berekent de crowding afstand in front F_i
$P_{t+1} = P_{t+1} \cup F_i$	Voeg front F_i toe aan de populatie
$i = i + 1$	Ga naar het volgende front
Sorteer(F_i, $\prec_n$)	Sorteer aflopend op crowding afstand
$P_{t+1} = P_{t+1} \cup F_i[1 : (N - \lvert P_{t+1}\rvert)]$	Vul aan met eerste $(N - \lvert P_{t+1}\rvert)$ uit F_i
Q_{t+1} = maak-nieuwe-populatie(P_{t+1})	Maak nakomelingen door selectie, kruising en mutatie
$t = t + 1$	Ga naar de volgende generatie

Dit spreekt voor zich en in de implementatie moeten er nog heel wat dingen toegevoegd worden, zoals voor het evalueren van de oplossingen en het verwerken van gegevens. Deel F.3 bevat meer informatie over de delen van de hoofdfunctie, terug te vinden als `evolve()` en hulpmethoden. De functie `reproduce()` is de implementatie van bovenstaande 'maak-nieuwe-populatie(P)'.

Fast non-dominated sorting

Een essentieel deel van NSGA-II is het snel niet-gedomineerd sorteren. In pseudocode ziet dit er als volgt uit, gebaseerd op Deb et al.[6]. In deze code is $p.n$ de dominantie teller en $p.S$ de set oplossingen gedomineerd door p. De betekenis van $p \prec q$ is 'p domineert q'.

$\forall p \in P$
    $p.S = \emptyset$

$p.n = 0$
$\forall q \in P$
als $p \prec q$ dan
$p.S = p.S \cup q$
anders als $q \prec p$ dan
$p.n = p.n + 1$
als $p.n = 0$ dan
p.rank = 1
$F_1 = F_1 \cup p$
$i = 1$
terwijl $F_i \neq \emptyset$
$Q = \emptyset$
$\forall p \in F_i$
$\forall q \in p.S$
$q.n = q.n + 1$
als $q.n = 0$ dan
$q.rank = i + 1$
$Q = Q \cup q$
$i = i + 1$
$F_i = Q$
$\forall p \in P$
$p.S = \emptyset$

In `Genepool.fast_nondominated_sort(P)` in deel F.3 is de implementatie in python te vinden. De laatste twee lijnen code hierboven maken de set gedomineerde oplossingen in het `Vector` object leeg om problemen in de volgende generatie te vermijden.

Crowding distance assignment

Een tweede essentieel deel van NSGA-II is het toekennen van een crowding distance. In `Genepool.crowding_distance_assignment(L)` in deel F.3 is de implementatie in python te vinden. Hier volgt het werkingsprincipe in pseudocode.[6]. Om te normaliseren is voor elk objectief gewoon de grootste en kleinste voorkomende waarde in de set vectoren als maximum en minimum gebruikt.

$l = |L|$
$\forall i \in L$
$i.distance = 0$
voor elk objectief m
$L = \text{sorteer}(L, m)$
$L[0] = \infty$
$L[l - 1] = \infty$
voor $i = 1$ tot $(l - 2)$

$$L[i].distance = L[i].distance + (L[i+1].m - L[i-1].m)/(L[l-1] - L[0])$$

3.5.2 Genetische operatoren

De volgende stap naar een werkend genetisch algoritme is het implementeren van de genetische operatoren selectie, kruising en mutatie. De implementatie van de genetische operatoren komt overeen met de aanpak van Sharma et al.[21].

Selectie

Uit de ouderpopulatie worden twee vectoren gekozen die een nakomeling zullen voortbrengen. Hoe deze vectoren gekozen worden, wordt bepaald door de selectie operator. In deel 5.3 zal nagegaan worden of binaire *tournament selection* (beste van twee) er een betere keuze is voor deze operator maar voorlopig worden twee willekeurige vectoren uit de volledige populatie gekozen als ouders.

Kruising

Saxena et al. vermelden het gebruik van vijf verschillende mutatie operatoren voor een betere prestatie. Deze zijn éénpunts kruising, tweepunts kruising, uniforme kruising, aritmetische kruising en heuristische kruising. In de eerste twee vormen wordt gekruist volgens één of twee partitielijnen. Uniforme kruising is het wisselen van stukken met een kans van 50% tussen een groot aantal partitielijnen. Dit zal hier niet verder uitgewerkt worden. Aritmetische kruising neemt het gemiddelde van de genen van de ouders, dit is in deze toepassing niet mogelijk omwille van de binaire vorm van de genetische informatie. In heuristische kruising tot slot wordt een nakomeling geproduceerd die identiek is aan de beste ouder. Het introduceren van duplicaten is echter nadelig zoals later zal blijken in deel 5.1.

Voor de implementatie werden de volgende ontwerpbeslissingen genomen. Ten eerste moet kruising zowel op kolommen als rijen kunnen werken in de 2D voorstelling van de topologie. Verder kan een operator ontworpen worden die zowel één- als tweepunts kruisingen kan produceren met een variabele grootte van het gekruist gebied. Dit leidt tot de volgende implementatie:

- Kies twee ouders.
- Beslis met een kans gelijk aan de kruisingskans om een kruising uit te voeren.
- Met 50% kans wordt de operatie op rijen uitgevoerd i.p.v. kolommen. De matrix wordt gewoon getransponeerd en later teruggetransponeerd.
- Een willekeurige breedte voor het te wisselen gebied wordt gekozen. Deze is steeds lager dan de helft van het aantal rijen (kolommen).
- Rekening houdend met deze breedte wordt een willekeurig startpunt gekozen.

- De delen worden gewisseld en twee nieuwe vectoren worden gemaakt met het geproduceerde genetisch materiaal.
- Deze vectoren ondergaan verder nog mutatie en worden bij de *offspring* gevoegd indien ze geldig zijn.

De functie `Genepool.crossover(one, two)` bevat de python code en is te raadplegen in appendix G.

Mutatie

Het implementeren van mutaties is eenvoudig. Met een bepaalde kans moet een bit van toestand veranderd worden. In het ideale geval wordt gemiddeld een bepaald aantal bits in de structuur veranderd en niet een percentueel aantal omwille van de variabele grootte.

Dit wordt mogelijk door een mutatie-interval n te specifiëren. Dit interval bepaalt om de hoeveel bits er gemiddeld één bit veranderd wordt. Dat gebeurt door met een kans gelijk aan $1/n$ de bitwaarde met 1 te verhogen en de 2-modulus te nemen. De functie `Genepool.mutate(genome)` bevat dit zeer kort stukje python code en is te raadplegen in appendix G.

3.5.3 Implementatie van een objectieffunctie

Om de *'fitness'* van een oplossing te bepalen worden één of meerdere objectieffuncties gekozen waarmee de vectoren vergeleken worden. Minimalisatie van de massa is een eenvoudig voorbeeld, dit is gewoon de som van het volledige genoom.

Om na te gaan hoe goed voor een opgegeven gewenste verplaatsing aan dit objectief voldaan is, wordt voor de gemakkelijkste aanpak gekozen voorlopig. De totale verplaatsing in de x- en y-richting van een bepaalde knoop wordt vergeleken met de gewenste verplaatsing en geëvalueerd met het kleinste kwadraten criterium. De gebruikte objectieffunctie is dus simpelweg:

$$Afwijking = \Delta x^2 + \Delta y^2 \tag{3.1}$$

3.6 Besluiten

In dit hoofdstuk werden de belanrgijkste routines in NSGA-II verder besproken voor hun implementatie in een Python code. Routines werden uitgewerkt om vectoren te genereren en deze te controleren, om zo een populatie op te bouwen. FE berekeningen worden uitbesteed aan Abaqus door het DNA te vertalen naar een Abaqus invoerbestand, vergezeld van de juiste parameters, en na simulatie het bekomen resultaat te laden. Van dit resultaat wordt de eindpositie van het uitvoerpunt vergeleken met een gewenste positie door middel van het kleinste kwadraten criterium om zo het objectief van minimale afwijking te kunnen valideren.

Hoofdstuk 4

Sensitiviteitsanalyse van de parameters

De performantie van het genetisch algoritme is afhankelijk van de gekozen waarden voor enkele bepalende parameters. In een genetisch algoritme zijn de bepalende parameters:

- De populatiegrootte.
- Het aantal iteraties of generaties dat het algoritme berekent.
- De kans op kruisen bij het maken van een *offspring* populatie.
- De kans op mutaties in het DNA van de topologie.

Het algoritme zal beter presteren bij bepaalde waarden van deze parameters dan bij andere. Optimale parameters liggen niet eenduidig vast, maar zullen afhankelijk zijn van de gekozen objectieven, topologiegrootte en andere beïnvloedende factoren. Ook beïnvloeden ze zich onderling, er zal bijvoorbeeld een afweging tussen de populatiegrootte en het aantal generaties gemaakt moeten worden.

Om uit te zoeken welke invloeden er zijn en welke waarden van de parameters wenselijk zijn, worden enkele veronderstellingen geformuleerd en vervolgens een aantal tests uitgevoerd.

- De benodigde rekenkracht hangt af van het aantal beschouwde vectoren, wat het product is van het aantal generaties en de populatiegrootte.
- Er zal bij een vastgelegde rekenkracht een *trade-off* gemaakt moeten worden tussen de populatiegrootte en het aantal generaties, voor een bepaald product van beide is er vermoedelijk een optimale verhouding.

- Het gemiddeld aantal generaties tot de beste oplossing bereikt wordt, kan bij een voldoende grote steekproef een metriek zijn om de performantie van het algoritme te meten. Deze aanpak zal in de tests gebruikt worden (enkel de populatiegrootte ligt vast). Bij normaal gebruik van het algoritme kan een vaste voorspelde rekenkracht wenselijk zijn, in dat geval zal zowel de populatiegrootte en het aantal generaties moeten vastgelegd worden.

- kruising en mutatie kunnen vrij onafhankelijk geoptimaliseerd worden. (De testen moeten hier verifiëren of ze eventueel toch een *trade-off* zijn.)

- Bij een bepaalde populatiegrootte bestaat er een optimale mutatie- en kruisingskans. Deze optima zijn afhankelijk van het specifieke probleem. (Hier moeten tests nagaan of de populatie inderdaad een invloed heeft).

Er is een overvloed aan methoden om de parameters te optimaliseren, zowel vooraf als tijdens het uitvoeren van het algoritme. [4] De eenvoudigste methode om parameters vooraf te bepalen, die ook in de volgende secties gebruikt zal worden, is heel de oplossingsruimte afgaan en dingen uitproberen. Met behulp van de bovenstaande veronderstellingen wordt deze ruimte al enigszins beperkt. Andere methoden die vooraf toegepast moeten worden, gebruiken een meta-genetisch algoritme dat zelf versies van een genetisch algoritme als vectoren heeft, of passen een *racing strategy* toe [13]. De andere optie, die het algoritme een stuk ingewikkelder maakt, is de kans op mutatie en kruising *realtime* aanpassen [24], de populatiegrootte aanpassen of werken met genetische operatoren die tot op een gewenst niveau deterministisch zijn in plaats van probabilistisch [16].

4.1 Invloed van kruisen en muteren

Om te beginnen wordt voor een specifieke populatie nagegaan of er een optimale combinatie van kruisings- en mutatiekans bestaat en of deze afhankelijk zijn van elkaar.

Als achtergrond is het belangrijk te vermelden dat in het werk van Sharma et al. een kruisingskans van 95% en een mutatie-interval van lengte n wordt gebruikt (mutatiekans $1/stringlengte$, voor een populatie van 240 en een DNA van 637 bits waarvan er 625 de topologie voorstellen, 25x25 dus).[21] [19] Deb et al. gebruiken in de oorspronkelijke NSGA-II paper dezelfde mutatieparameters en een kruisingskans van 90% [6], maar vermelden dat deze nog niet geoptimaliseerd zijn.

De meerderheid van de figuren bij deze sectie zijn te vinden in appendix D omdat ze een groot aantal pagina's beslaan.

4.1.1 Het concept mutatie-interval

In de hierop volgende besprekingen wordt gebruik gemaakt van het 'mutatie-interval' en niet van de mutatiekans, omdat dit een eenvoudiger te hanteren begrip is. Het mutatie-interval is het aantal bits waarin gemiddeld één mutatie verwacht wordt. De mutatiekans is dus $1/n$ met n het mutatie-interval. Dit is ook exact de manier waarop de mutatiekans van een afzonderlijke bit bepaald wordt in de code, na het opgeven van een mutatie-interval. De voornaamste reden hiervoor is dat de mutatiekans automatisch moet kunnen variëren met de grootte van de topologie. Men geeft dus niet op hoe groot de kans is dat een bit muteert, maar hoeveel mutaties er gemiddeld per vector moeten optreden.

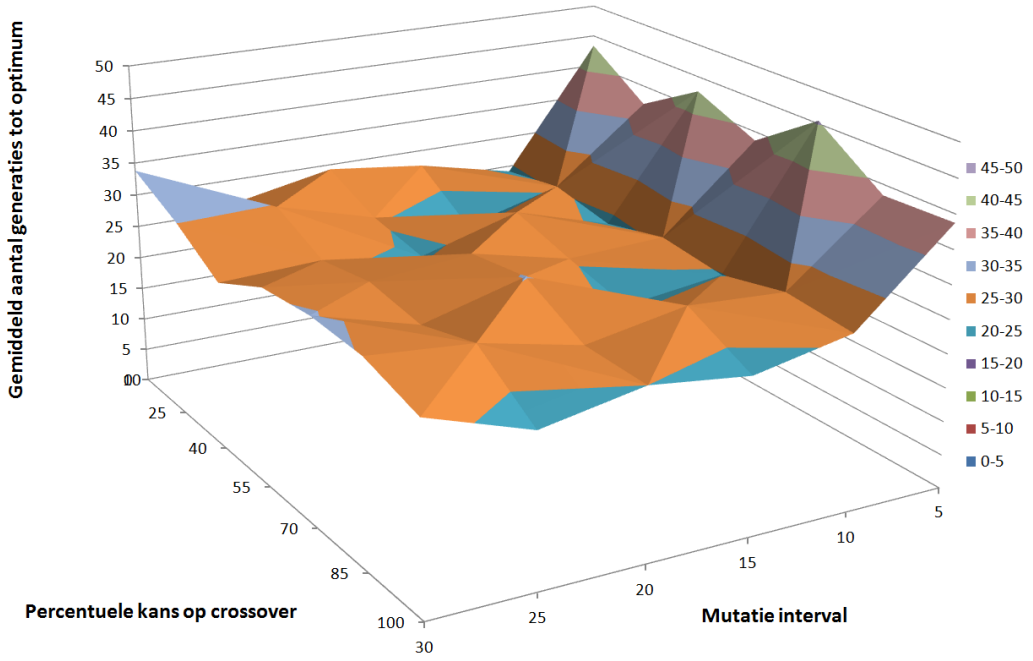
4.1.2 Het objectief determinant

De objectieven waar het genetisch algoritme gebruik van zal maken, zijn gebaseerd op de verplaatsing van een eindpunt en op de massa van de structuur. Om verplaatsingen na te gaan zijn berekeningen in Abaqus nodig die te veel rekentijd in beslag nemen om voor deze tests gebruikt te kunnen worden. Daarom wordt bij alle tests in deze sectie gebruik gemaakt van metrieken die eenvoudig te berekenen zijn voor een kleine matrix: de som (minimale massa) en de determinant van het DNA in matrixvoorstelling. De determinant minimaliseren zou echter een objectief geven dat ook de massa minimaliseert en dus samenwerkt met het eerste objectief. De optimale oplossing, het 3x3 blokje uit figuur 3.3 heeft determinant 0, wat ook geldt voor de eenvoudigste mutaties daarvan. Dit is niet wenselijk gezien de bekomen resultaten geen front zullen vormen zoals onder normale omstandigheden, maar er convergentie zal zijn naar een bepaald resultaat voor de hele populatie. Maximalisatie van de determinant zal de normale werking van het algoritme beter nabootsen door de concurrentie tussen beide objectieven, wat het convergeren naar een oplossing met minimale massa uitdagender maakt. Het aantal generaties alvorens een oplossing met de minimummassa van 8 ontstaat, zal in onderstaande test ook telkens als maat van de performantie gebruikt worden.

4.1.3 Testcase 1: 5x5 topologie

Voor een populatie van 20 werd bij een hele reeks combinaties van mutatie- en kruisingskans voor telkens 50 pogingen het gemiddeld aantal generaties berekend tot de minimummassa van 8 bereikt werd. In de oplossingsruimte van 5x5-topologieën bestaan er 9 optimale oplossingen tussen in totaal $2^{25} = 33554432$ verschillende DNA-codes. Daarvan zullen er, verder bouwend op de berekeningen in tabel 3.1, zeker 1% overeenkomen met een geldige vector. Om op brute kracht een oplossing te vinden, zouden er dus conservatief geschat gemiddeld 37283 vectoren doorlopen moeten worden.

In figuur 4.1 is duidelijk te zien dat een mutatie-interval van 5 overeenkomt met een veel te hoge mutatiekans en een beduidend langere rekentijd geeft. Voor de andere combinaties is er echter geen duidelijk verloop naar een optimum. In figuur D.2 en



Figuur 4.1: Oppervlak van het gemiddeld aantal generaties voor verschillende combinaties van mutatie- en crossoverkans.

D.3 in bijlage zijn doorsneden te zien van figuur 4.1. Het gemiddeld aantal generaties lijkt te dalen van een mutatie-interval van 30 tot 10 voor een gemiddelde van alle kruisingskansen, om daarna scherp te stijgen. Het effect van de kruisingskans is niet duidelijk, gemiddeld ligt het minimum bij 100% kans. De variantie is tot slot nog weergegeven in figuur D.4, om te wijzen op erg variabele rekentijden of uitbijters. Enkel bij mutatie-interval 5 is de variantie opvallend hoog.

De enige besluiten die met zekerheid genomen kunnen worden, is dat een mutatie-interval best kleiner is dan 30, maar zeker groter moet zijn dan 5, gezien een dergelijk klein interval slechte performantie oplevert. Een waarde tussen 25 en 10 is acceptabel. De kruisingskans lijkt vrij gekozen te kunnen worden, maar bij het onderzoeken van de invloed van de populatiegrootte zal verder blijken dat dit zo is omdat in een topologie van 5x5 mutatie veel belangrijker is en ook zonder kruisingen in een bevredigende tijdsduur een optimum gevonden kan worden.

4.1.4 Testcase 2: grotere populatie

Alvorens tot het bovenstaande besluit te komen dat de topologiegrootte de boosdoener is, werd toch een simulatie uitgevoerd onder dezelfde omstandigheden als testcase 1, maar met een populatie van 40. Met deze voorkennis zou men hetzelfde resultaat verwachten, met eventueel een kleiner aantal generaties tot het optimum gevonden wordt. Het gemiddeld aantal generaties ligt inderdaad tussen 15 en 20 in figuren D.6

en D.7, vergeleken met 25 voor een populatie van 20. In figuur D.7 is duidelijker zichtbaar dan in figuur D.3 dat met een grotere kruisingskans sneller het optimum bereikt wordt. Figuur D.6 beperkt het mutatie-interval op haar beurt van 10 tot 20.

De besluiten uit testcase 1 worden dus aangescherpt. Deze resultaten moeten wel met voorzichtigheid gebruikt worden, gezien figuur D.8 een grilliger verloop van de variantie aantoont en daarmee ook een lagere betrouwbaarheid van de resultaten.

4.1.5 Testcase 3: grotere topologie

Testcase 3 gebruikt een grotere topologie van 6x6 met een populatiegrootte van 100 (gekozen in de buurt van het optimum zoals zal blijken in de volgende sectie). Zowel de populatie en de topologie zijn groter, dus kruising zou tegenover mutatie in belang moeten winnen. Helaas zijn deze berekeningen zwaarder door een grote toename van het nodige aantal generaties, met voor elk datapunt gemiddeldes over 10 pogingen, duurde deze simulatie al ongeveer 37 uur. Het feit dat de minimaal mogelijke massa nog steeds 8 is, zorgt op zichzelf al voor een toename in het aantal generaties, de beginpopulatie heeft namelijk een gemiddelde massa van de helft van het aantal bits in zijn DNA voorstelling en zelfs iets meer door herstel. De gemiddeld startmassa is gestegen van $13 + h$ tot $18 + h$ met h het gemiddeld aantal herstelbits dat wordt toegevoegd om puntcontacten te vermijden.

Het enige van de voorgaande besluiten dat bevestigd wordt, is de sterke toename van het aantal generaties bij een mutatie-interval van 5. Zo opvallend zelfs dat deze waarde van de figuren verwijderd diende te worden om voldoende detail te kunnen tonen. De gemiddeldes in figuur D.10 en D.10 zijn niet betrouwbaar omwille van het algemeen grillig verloop van de verkregen resultaten. De resultaten geven de indruk dat het optimale mutatie-interval groter is dan bij een kleinere populatie of topologie, maar de resultaten zijn te onbeslist om dit met zekerheid te besluiten. Dit resultaat verwacht men echter wel, gezien het DNA nu 36 bits groot is, waardoor het effect van een mutatie-interval van 36 hier vergelijkbaar is met een interval van 25 voor een 5x5 topologie.

4.2 Invloed van de populatiegrootte

Het doel van deze sectie is om een test uit te voeren om de afweging te maken tussen de populatiegrootte en het aantal generaties bij een bepaalde hoeveelheid rekenkracht. Het is helaas moeilijk om hiervoor een test te bedenken, omwille van de moeilijkheid om objectief te oordelen over de prestaties van het algoritme. Bekijkt men de oplossing met de laagste massa, is de beoordeling erg ruw, de uitkomst is één uit een beperkt aantal waarden en kan al generaties lang op hetzelfde minimum blijven steken zijn. Wanneer het optimum bereikt blijkt, is de situatie nog erger, want dit is het plafond voor het meten van de performantie. Er is bij een dergelijk resultaat enkel geweten dat het optimum bereikt is, maar niet hoe snel.

De uitgevoerde test die dit omzeilt, laat het algoritme wederom itereren tot een oplossing met minimummassa gevonden is. Dit wordt voor verschillende populatiegroottes een significant aantal keren geprobeerd. Uit deze test worden een aantal gegevens bekomen; van belang zijn het gemiddeld aantal generaties en beschouwde vectoren (rekenkracht), de gemiddelde rekentijd en de variantie. De optimale populatiegrootte is deze waarbij de rekenkracht (en evt. rekentijd) een lokaal minimum vertonen. Het gemiddeld aantal benodigde generaties zou een dalende functie moeten zijn van de populatiegrootte. Ten slotte kan de variantie uitschieters in de test onthullen en is ze een algemene maat voor de consistentie van het benodigd aantal generaties. De variantie is dus bij voorkeur laag voor een betrouwbaar testresultaat en een voorspelbaar algoritme .

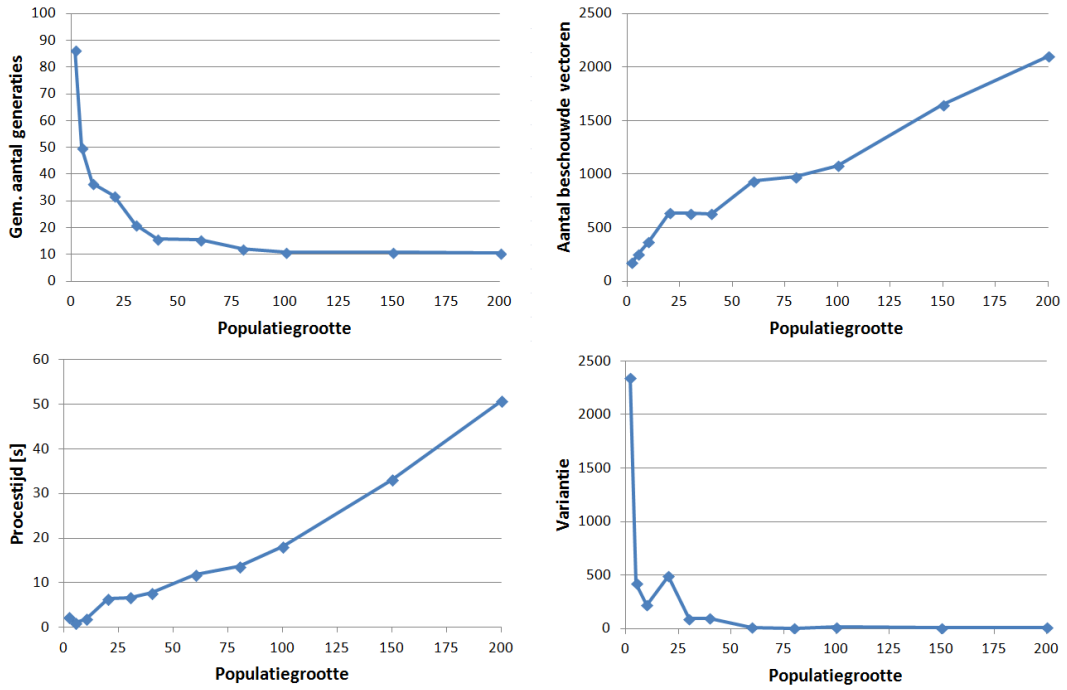
Ter verificatie kan nog nagegaan worden of de rekentijd en het aantal beschouwde vectoren wel een vaste verhouding geven zoals eerder verondersteld is.

Interessante case: bij een populatiegrootte van 1 kan enkel mutatie plaatsgrijpen en begint de code op een willekeurige manier en op brute kracht een oplossing te zoeken. Als beide objectieven elkaar helpen zoals wanneer massa en determinent geminimaliseerd worden, werkt dit degelijk, doch trager dan bij grotere populaties. Wordt de determinant echter gemaximaliseerd als tweede objectief, is een minimale populatie van 2 nodig, gezien de doelen concurrentieel zijn. Met een populatie van 1 bereikt het algoritme de optimale massa niet (getest tot meer dan 5000 generaties). Een eenvoudige verklaring is dat de populatie in volgorde van de objectieven gesorteerd wordt en het determinant-objectief volgt op het massa-objectief.

4.2.1 Testcase 1: 5x5

In figuur 4.2 is het resultaat zichtbaar van een test met 5x5-topologieën, geoptimaliseerd op minimale massa en maximale determinant, met kruisingskans 95% en mutatie-interval $n/2$, ofwel gemiddeld één mutatie in elke 13 bits. De data bestaat uit gemiddeldes, genomen voor tien pogingen, in de beschouwde populaties van 2, 5, 10, 20, 30, 40, 60, 80, 100, 150 en 200 vectoren. Het minimum is 2 omwille van de reden beschreven in de vorige paragraaf.

In de eerste grafiek is te zien dat het gemiddeld aantal generaties inderdaad een dalende functie is van de populatiegrootte P. Er is echter geen optimum zichtbaar in rekengrootte en rekentijd, deze zijn een stijgende functie van P. Een mogelijke verklaring is dat een 5x5-topologie vrij klein is, waardoor mutaties op zich reeds in een aanvaardbaar aantal generaties een oplossing genereren. Als kruisingen relatief onbelangrijk zijn, wordt de populatiegrootte eveneens onbelangrijk. De variantie wijst er wel op dat het aantal generaties wisselvallig is en dergelijk kleine populaties te vermijden zijn. Bemerk wel dat het onderzochte aantal vectoren veel lager is dan de 37283 die nodig zouden zijn bij ongericht zoeken, een verbluffend resultaat. Omwille van de invloed van mutaties is een test met een grotere topologie vereist.



Figuur 4.2: Resultaten van testcase 1: uitgezet tegenover de populatiegrootte het gemiddeld aantal generaties (linksboven) en beschouwde vectoren (rekenkracht)(rechtsboven), de gemiddelde rekentijd (linksonder) en de variantie (rechtsonder).

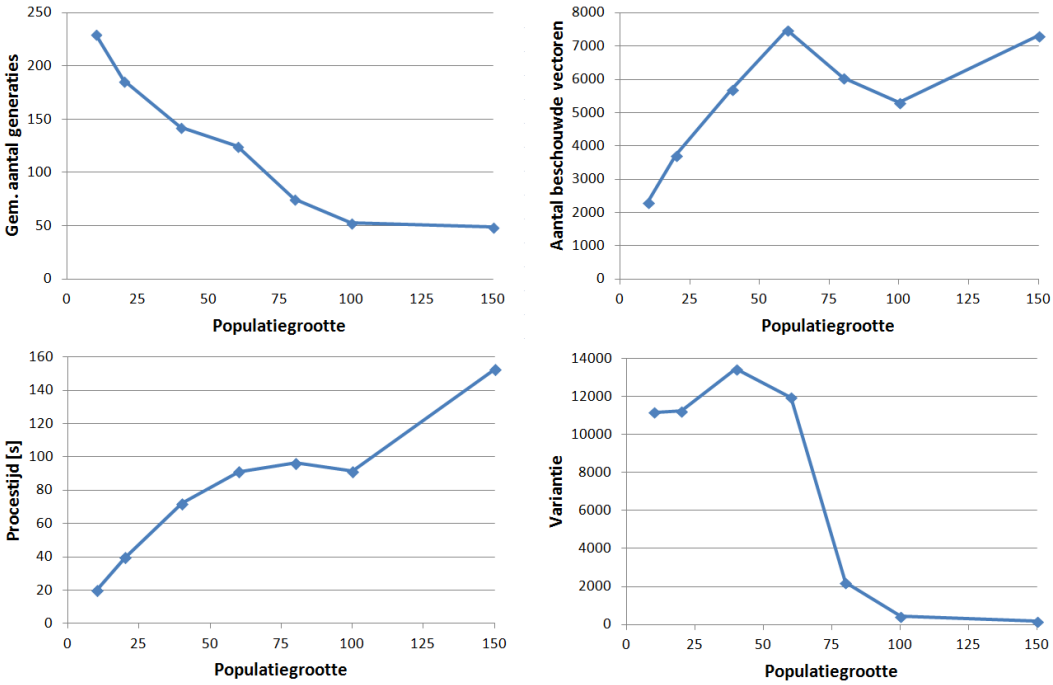
4.2.2 Testcase 2: 6x6

In figuur 4.3 is het resultaat zichtbaar van een test met 6x6-topologieën, met dezelfde objectieven en mutatie- en kruisingskans als testcase 1. Gemiddeldes zijn ook hier genomen voor tien pogingen, de beschouwde datapunten zijn populaties van 10, 20, 40, 60, 80, 100 en 150 vectoren. Het minimum is 10 omwille van enorm trage convergentie, mutatie op zichzelf kan duidelijk geen oplossing forceren in een bevredigende tijd, wat goed nieuws is.

In de eerste grafiek is het vertrouwde dalende verloop te zien. De variantie is zeer hoog tot een populatie van 60 en daalt drastisch vanaf 80. De procestijd en het aantal berekeningen vertonen bij deze probleemgrootte wel een duidelijk minimum dat zich tussen 80 en 150 moet bevinden. Zoals werd verondersteld, is bij een grotere topologie wel een minimum aanwezig, wat te verklaren is door een verhoogd belang van de kruisingen tegenover de mutaties.

4.2.3 Algemene effecten van de Populatiegrootte

De testcases hierboven toonden aan dat er enkele effecten optreden die gerelateerd zijn aan de grootte van de populatie. Om te beginnen geven kleine populaties aanleiding



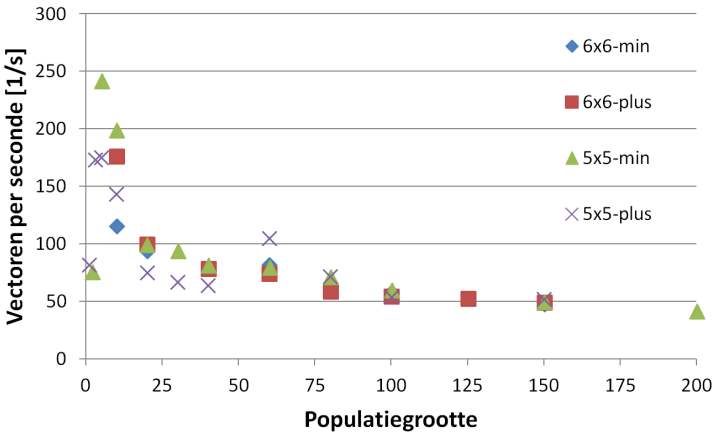
Figuur 4.3: Resultaten van testcase 2: uitgezet tegenover de populatiegrootte het gemiddeld aantal generaties (linksboven) en beschouwde vectoren (rekenkracht)(rechtsboven), de gemiddelde rekentijd (linksonder) en de variantie (rechtsonder).

tot sterk variërende rekentijden. Naarmate de populatie groter wordt, stijgt de rekentijd of het aantal berekeningen, deze stijging gaat door een lokaal minimum voor een voldoende grote topologie en een voldoende uitdagende set objectieven, zoals in de volgende sectie te zien is in figuur 4.5.

Dat een dergelijk minimum bestaat, is niet bijzonder. Een te kleine populatie zorgt er immers voor dat men te lokaal blijft zoeken en er te weinig genetisch materiaal in de pool voorhanden is voor kruisingen, wat het vinden van een globaal optimum bemoeilijkt. Bij en grote populatie nemen de voordelen van het genetisch algoritme af omdat de aanpak meer op het zoeken met brute kracht begint te lijken, waarbij alle mogelijke vectoren één voor één worden overlopen.

Nog een interessante bedenking is de volgende: kolommen en rijen wisselen niet van onderlinge positie, ze worden wel gekruist tussen vectoren. In een topologie van 5x5 zijn er dus $2^5 = 32$ unieke rijen of kolommen mogelijk, bij 6x6 zijn dat er $2^6 = 64$. Als de respectievelijke populaties kleiner zijn dan dat aantal, ligt het vast dat niet al het genetisch materiaal voorhanden is om met rij- of kolomkruisingen alleen alle mogelijke oplossingen te kunnen bekomen. In de figuren 4.2 en 4.3 valt het ook op dat het gemiddeld aantal generaties beduidend hoger ligt bij populatiegroottes

beneden deze grenzen. Er kan dus besloten worden dat het een goed idee is, met het oog op voldoende genetische diversiteit, om de populatie niet kleiner te nemen dan 2^L, met L de gemiddelde dimensie.



Figuur 4.4: Vergelijking van het beschouwde aantal vectoren per seconde voor de testcases en hun tegenhanger met tegengestelde eisen aan de determinant als objectief.

In figuur 4.4 is te zien wat het aantal beschouwde vectoren per seconde bedroeg voor beide testcases, evenals voor de objectieffuntie die de determinant minimaliseerde. In de legende staat plus voor minimalisatie en min voor maximalisatie, het bijschrift duidt dus op het teken van de determinant in het objectief.

Het is duidelijk dat het aantal beschouwde vectoren per seconde in alle gevallen gelijkaardig is, behalve voor zeer kleine populaties. Daar treden grote schommelingen op, voor zeer kleine populaties (1 of 2) is het aantal vectoren per seconde zelfs lager dan grotere populaties, dit kan verklaard worden omdat het behandelen van de vector relatief minder tijd inneemt in vergelijking met het uitvoeren van de code in zijn geheel. Een belangrijk deel van de tijd die niet aan een afzonderlijke vector kan worden toegeschreven is het sorteren en samenvoegen van een generatie en haar *offspring* met NSGA-II. De complexiteit van het algoritme is $O(M \cdot N^2)$ met M het aantal generaties en N de populatiegrootte, terwijl het aantal beschouwde vectoren gelijk is aan $N \cdot M$. [6]

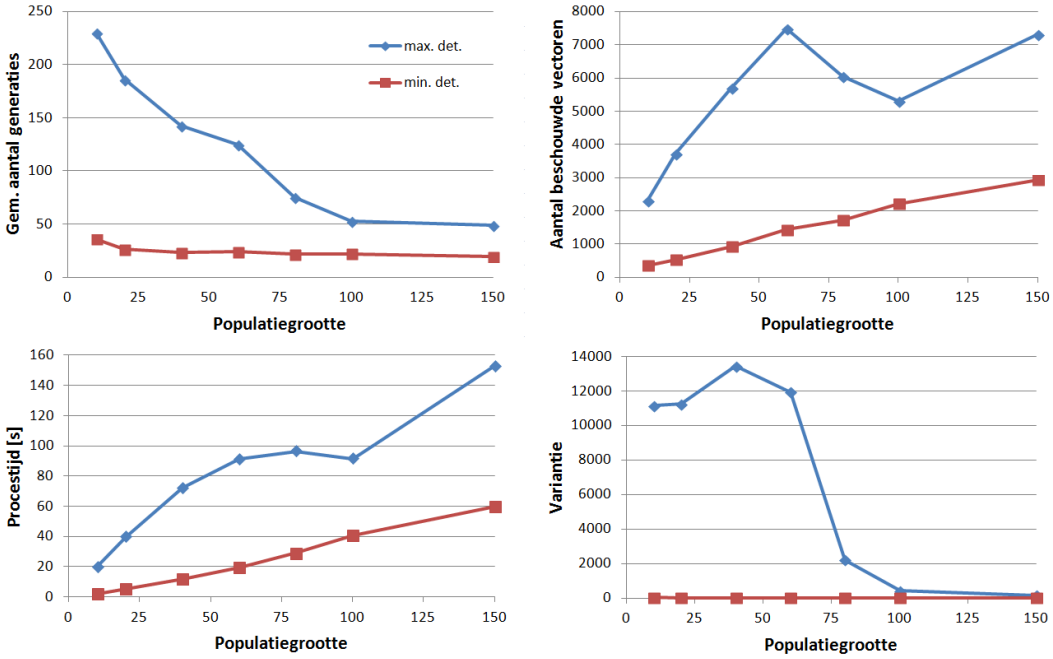
De berekeningen gerelateerd aan het algoritme nemen dus toe in belang voor grotere populaties, wat de dalende trend in figuur 4.4 kan verklaren. Afgezien van dit effect kan toch besloten worden dat de veronderstelling dat de rekenkracht afhangt van het beschouwde aantal vectoren een goede benadering is, zeker wanneer met Abaqus gewerkt zal worden en de eindige elementen simulaties dominant worden wat rekentijd betreft.

4.3 Invloed van andere factoren

Tal van andere factoren spelen ook een rol, zoals in de eerste plaats de grootte van de topologie, de gebruikte objectieffuncties en de machine waarop gewerkt wordt. Zoals eerder aangetoond, heeft de grootte van de topologie en dus van het oplossingsdomein een ingrijpende invloed op de vier eerder besproken factoren, in die mate dat deze bepalender is dan de vier vorige. Een aparte optimalisatie van de parameters voor elke topologiegrootte is aangewezen.

In figuur 4.4 waren de rekentijden een licht dalende trend in functie van populatiegrootte, maar voor toenemende topologiegroottes mag veilig aangenomen worden dat de curve van beschouwde vectoren per seconde verticaal naar beneden zal verschuiven, omdat het langer zal duren om de overeenstemming te berekenen voor ongewijzigde objectieffuncties. Deze gebruikte objectieffuncties hebben uiteraard een nog meer ingrijpende invloed en zijn bepalend voor het bereikt aantal vectoren per seconde. Met de verplaatsing als objectief, zijn de Abaqus berekeningen dominerend en wordt het aantal beschouwde vectoren per seconde enkele machten van tien kleiner.

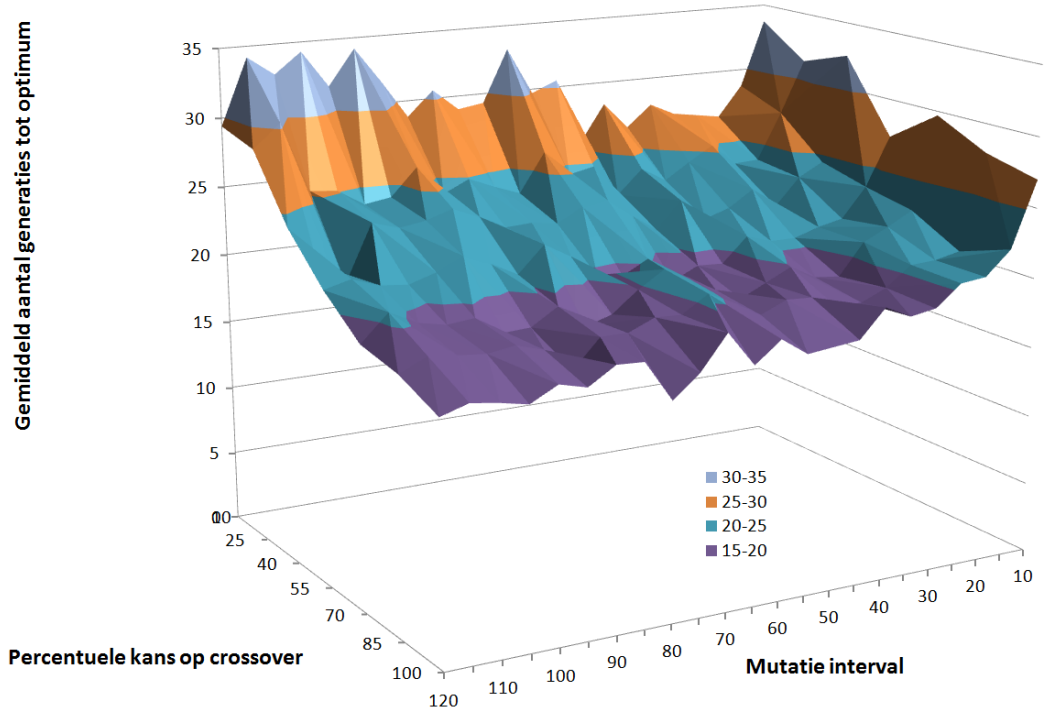
In figuur 4.5 zijn de resultaten te zien voor testcase 2 van de populatiegrootte, vergeleken met de resultaten voor een minimalisatie van de determinant onder dezelfde condities. In deze resultaten, voorgesteld door de rode grafieken, is geen optimum zichtbaar, maar de convergentie is dan ook veel sneller, met een zeer kleine variantie.



Figuur 4.5: Resultaten van testcase 2, vergeleken met de resultaten als de determinant geminimaliseerd i.p.v. gemaximaliseerd werd onder dezelfde randvoorwaarden.

Figuur 4.6 en 4.7 geven tot slot nog het equivalent van D.9 en D.9 (invloed mutatie en kruising testcase 3) voor minimalisatie van de determinant. Ze zijn hier opgenomen in plaats van in appendix D om de besluiten te illustreren, gezien ze de beste resultaten geven. Omdat de berekeningen bij benadering vijf maal zo snel verliepen, is er gekozen om een groter gebied te beschouwen.

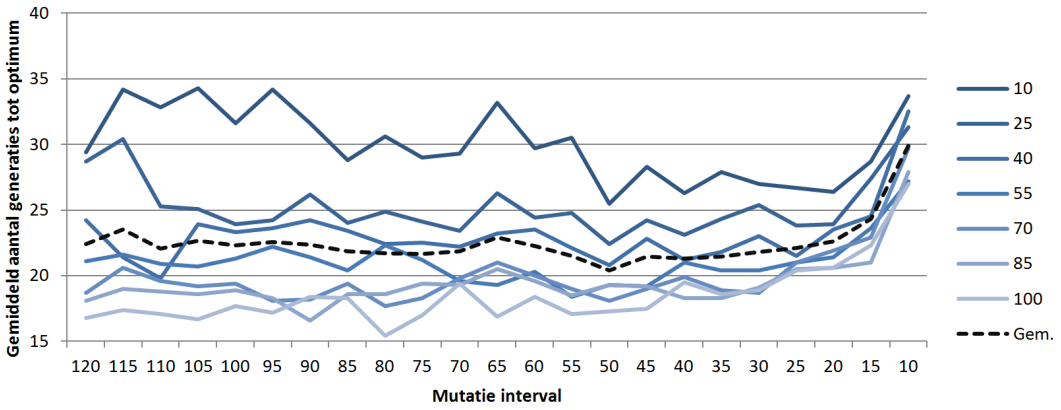
Dezelfde conclusies blijven geldig: de kruisingskans wordt best hoog genomen, het mutatie-interval best niet te klein. Zoals in figuur 4.7 daalt de performantie bij een mutatie-interval kleiner dan 30. Vooral bij lagere kruisingskansen is te zien dat een mutatie-interval groter dan 50 eveneens de performantie doet dalen.



Figuur 4.6: Oppervlak van het gemiddeld aantal generaties bij maximalisatie van de determinant, voor verschillende combinaties van mutatie- en kruisingskans.

4.4 Besluiten

Sharma et al. vermelden een gebruikte kruisingskans van 95% en een mutatiekans van $1/n$ voor DNA van n bits, oftewel een mutatie-interval van n. [21] [19] Hoewel niet beweerd wordt dat dit optimaal is in de betreffende papers, mag aangenomen worden dat deze waarden zorgvuldig gekozen zijn en in overeenkomst zijn met bovenstaande resultaten. De kruisingskans kan best zo hoog mogelijk gekozen worden, maar niet gelijk aan 100%. Dit om de eenvoudige reden dat pure mutaties dan niet meer voorkomen, welke belangrijk zijn om lokaal te zoeken.



Figuur 4.7: Gemiddeld aantal generaties uitgezet tegen de mutatiekans voor verschillende kruisingskansen, bij maximalisatie van de determinant.

Ook een hoge mutatiekans maakt erg lokaal zoeken onmogelijk. Uit de bekomen resultaten blijkt dat het mutatie-interval wel ruimer gekozen mag worden rond de waarde n, zolang het groot genoeg blijft. Ook hier moet echter opgemerkt worden dat het een doordachte keuze is om het interval gelijk te stellen aan het aantal bits. Dit is de ideale waarde om het snelst het optimum te bereiken als men er maar één bit van verwijderd is, omdat er bij pure mutaties gemiddeld één bit gemuteerd wordt.

Voor een bepaald probleem kan er een bepaalde populatiegrootte bestaan die een optimale convergentie oplevert, dit is echter een lokaal optimum en komt enkel voor bij voldoende grote topologieën en uitdagende objectieven. Dit optimum kan best voor het specifieke probleem berekend worden, indien er met een beperkte beschikbare rekentijd gewerkt moet worden.

Hoe snel een goede oplossing gevonden wordt, is sterk afhankelijk van toeval en het beschouwde probleem, maar neemt gemiddeld toe met een optimale keuze van populatiegrootte, mutatie- en kruisingskans.

Op het uiteindelijke aantal beschouwde vectoren per seconde hebben veel factoren een invloed. De topologiegrootte, kruisings- en mutatiekans en gebruikte objectieven vertegenwoordigen een vaste rekentijd per vector. De populatiegrootte voegt, los van de andere factoren, een variabel deel toe aan de rekentijd, want een grotere populatie sorteren duurt langer per vector.

Hoofdstuk 5

Verdere analyse van genetische operatoren

Dit hoofdstuk sluit aan op het vorige hoofdstuk en gaat in op effecten die het gevolg zijn van de specifieke manier waarop de genetische operatoren geïmplementeerd zijn. Door middel van aanpassingen en extra operatoren wordt gepoogd om het algoritme verder te verbeteren en de convergentie te versnellen.

5.1 Ontstaan van duplicaten

De vectoren in de populatie zijn niet noodzakelijk uniek. Als er geen kruising of kruising met zichzelf of een duplicaat optreedt, terwijl er geen bits gewijzigd worden, ontstaat een duplicaat. In bepaalde gevallen, bijvoorbeeld een populatie van 10 en een 6x6-topologie, is de kans groot dat heel de populatie na enkele tientallen generaties volledig uit dezelfde vector bestaat. Kruising wordt dan triviaal, enkel mutatie is nog aan het werk. De genoemde populatiegrootte is uiteraard ongezond klein. Toch dient onderzocht te worden of duplicaten onder normale omstandigheden de convergentie niet vertragen doordat ze de genetische variatie doen afnemen. Duplicaten kunnen anderzijds ook een positief effect hebben, door het algoritme meer lokaal te laten zoeken. Experimenten moeten uitwijzen welk effect de bovenhand heeft.

5.1.1 Grootte van het effect

Duplicaten kunnen op twee manieren gevormd worden. Enerzijds door het uitblijven van mutaties als er ook geen kruising plaatsvond. Voor elke geproduceerde vector is deze kans:

$$P_{dupl,1} = (1 - P_m) \cdot (1 - P_k) = \left(\frac{i_m - 1}{i_m}\right)^n \cdot (1 - P_k) \tag{5.1}$$

Met P_k de kruisingskans, P_m de mutatiekans, i_m het mutatie interval en $n = x \cdot y$ het aantal bits. Stelt men $i_m = \frac{n}{a}$ en neemt men de limiet voor n naar positief oneindig

geeft dit:

$$(1 - P_m) = \left(\frac{\frac{n}{a} - 1}{\frac{n}{a}} \right)^n = \left(n - \frac{a}{n} \right)^n = \frac{1}{e^a} \tag{5.2}$$

Zelfs voor een kleine topologie zoals 5x5 benadert dit de limiet al snel, dus kan men steeds gebruik maken van de waarde in de limiet. Voor $a = 1$ bijvoorbeeld wordt $(1 - P_m) = 0,368$ en voor $a = 2$ slechts $(1 - P_m) = 0,135$ en heeft het effect minder invloed. In figuren zoals 5.1 wordt $P_{dupl,1}$ met de letter M aangeduid. Het constant, onafhankelijk van de bestaande hoeveelheid duplicaten

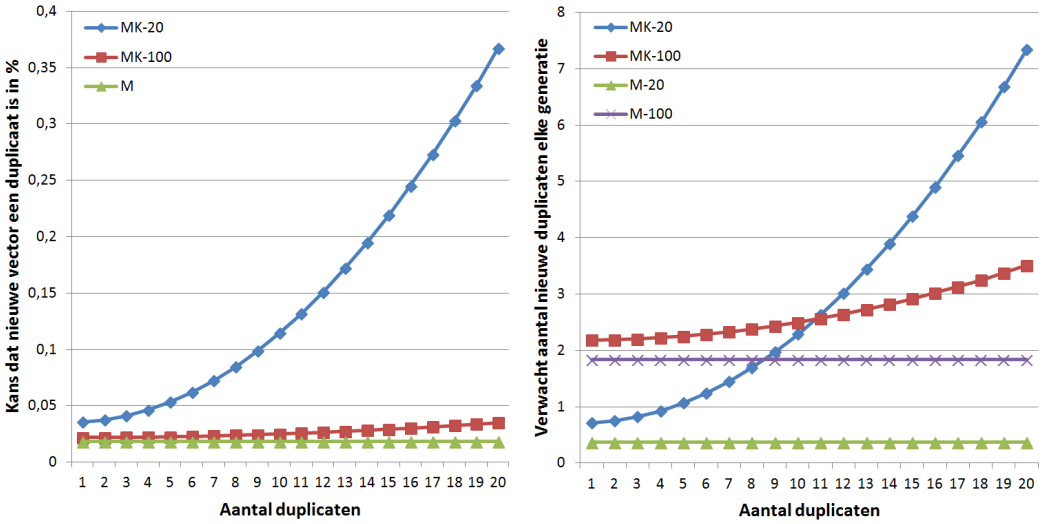
De tweede manier waarop duplicaten tot stand komen, is door zelfkruising of kruising met een duplicaat, gevolgd door het uitblijven van mutatie. Deze kans is dus aanvankelijk lager en hangt af van de populatiegrootte. Hoe meer duplicaten er ontstaan, hoe groter deze kans bovendien wordt. In formule uitgedrukt is de kans (met N de populatiegrootte en D het aantal duplicaten van de vector die meermaals voorkomt, in dit model komen dus van slechts één vector duplicaten voor):

$$P_{dupl,2} = P_k \cdot \left(\frac{N - D}{N} \cdot \frac{1}{N} + \frac{D}{N} \cdot \frac{D}{N} \right) = \frac{P_k}{N^2} \cdot \left(N - D + D^2 \right) \tag{5.3}$$

In figuren zoals 5.1 wordt dit aangeduid met de letter K. MK is dus $P_{dupl,1} + P_{dupl,2}$. In figuur 5.1 (a) zijn de kansen M en MK te zien, door deze te vermenigvuldigen met de populatiegrootte N wordt figuur (b) bekomen. Vanaf er duplicaten ontstaan, neemt de kans op het ontstaan van nog meer duplicaten duidelijk sterk toe. Figuur (b) geeft alweer een reden om geen te kleine populaties te gebruiken, de kans dat heel de populatie opvult met duplicaten van dezelfde vector is veel groter. De kans op zelfkruising is immers groter.

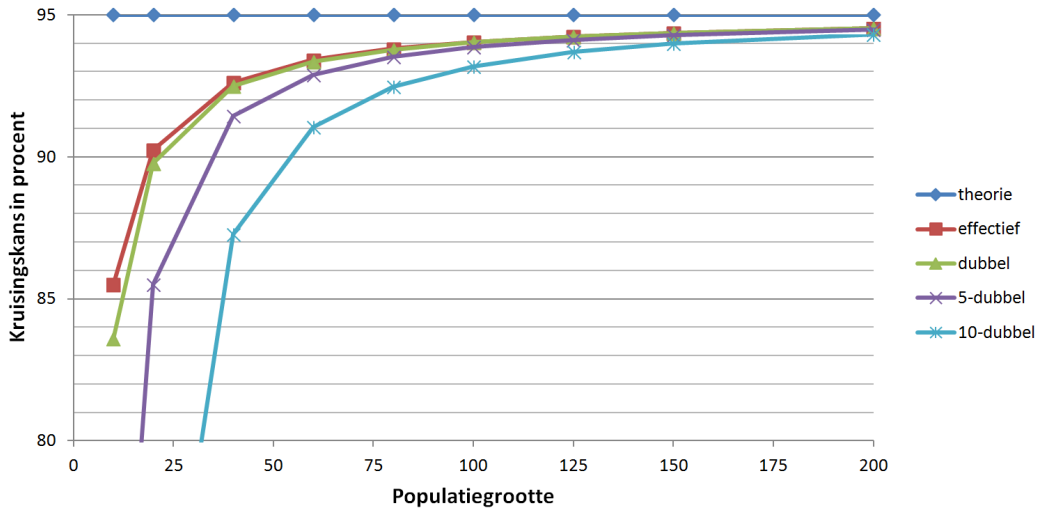
De implicaties van het duplicatie-effect zijn te zien in figuur 5.2: De kruisingskans is lager dan theoretisch opgelegd omwille van de zelfkruisingen en kruisingen met duplicaten. Hoe meer duplicaten er voorkomen, hoe lager de werkelijke mutatiekans. Ook hier wordt in de scenario's aangenomen dat er 2, 5, of 10 duplicaten bestaan van een bepaalde vector en de rest uniek is.

Dit betekent ook dat de assen van alle voorgaande figuren die de kruisingskans aangeven een vertekend beeld geven, gezien ze de theoretische kruisingskans aanduiden. De besluiten blijven echter correct, de figuren in appendix D stellen nog steeds hetzelfde oppervlak voor als een as herschaald wordt. De figuren uit subsectie 4.2 zijn echter niet uitgerekend bij constante kruisingskans, maar ook dit effect is enkel van invloed bij kleine populatiegroottes, die niet de voorkeur genieten.



(a) Kans op duplicaten (b) Verwacht aantal duplicaten per *offspring*

Figuur 5.1: Kans op duplicaten voor populaties van 20 en 100, puur uit mutatie (M) of uit mutatiegedrag en zelfkruising (MK).



Figuur 5.2: Invloed van zelfkruisingen en kruising met duplicaten op de effectieve kruisingskans, in functie van de populatiegrootte voor een aantal scenario's.

5.1.2 Experimenten en besluiten

Een vraag die zich om te beginnen stelt, naar aanleiding van het verschil tussen effectieve en theoretische kruisingskans, is of 100% kruisingskans niet aangewezen is. De waarde 95% werd gekozen om pure mutaties toe te laten, welke sowieso reeds plaatsvinden door zelfkruising en kruising met duplicaten. In de figuur is echter te zien dat voor een degelijk grote populatie deze daling aanvankelijk maar één procentpunt bedraagt, wat ook bij een mutatiekans van 100% zo is. De kruisingskans komt dus op 99% te liggen, wat waarschijnlijk nog te hoog is.

Er zijn heel wat aantal aanpassingen mogelijk om het duplicatiegedrag te veranderen. In figuur 5.3 worden deze vergeleken. Volgende gevallen worden er onderscheiden en vergeleken voor minimalisatie van de determinant bij verschillende populatiegroottes en een 6x6 topologie:

MK Zonder aanpassingen, duplicatie gebeurt op beide manieren. Vergelijkbaar met de rode lijn in figuur 4.5 maar met mutatie-interval n, de kruisingskans nog steeds 95%.

M Zelfkruising en kruising met duplicaten wordt onmogelijk gemaakt door een vergelijking van de md5-naam, tenzij de populatie al gevuld is met één en dezelfde vector door duplicatie dankzij $P_{dupl,1}$ (M). Voor een populatie van 10 gebeurt dit bijna even vaak wel als niet.

NoDup Beide vormen van dupliceren worden uitgeschakeld door nieuwe vectoren te vergelijken met hun ouders. Er wordt dus nooit een vector gedupliceerd, wat niet betekent dat er niet toevallig identieke vectoren gevormd kunnen worden tijdens het genereren van de beginpopulatie of uit afzonderlijke kruisingen. Deze kans is echter zeer laag voor een voldoende grote populatie.

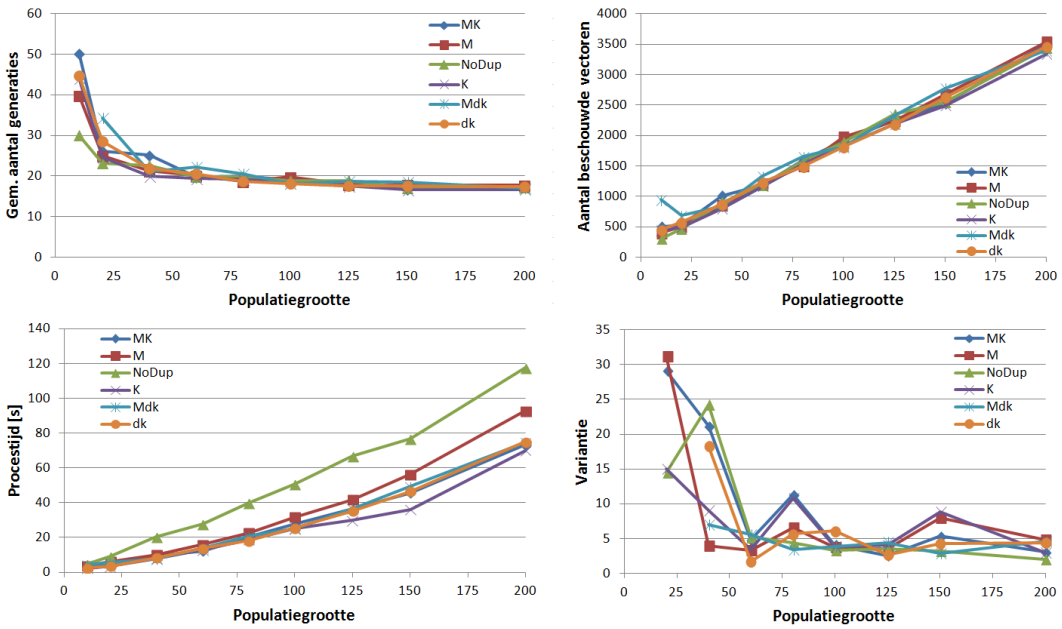
K Door de kruisingskans op 100% te zetten, wordt het M-effect uitgeschakeld.

Mdk Zelfkruising wordt vermeden, maar kruising met duplicaten niet. (zk duidt op dit effect, wat een deel is van het K-effect.)

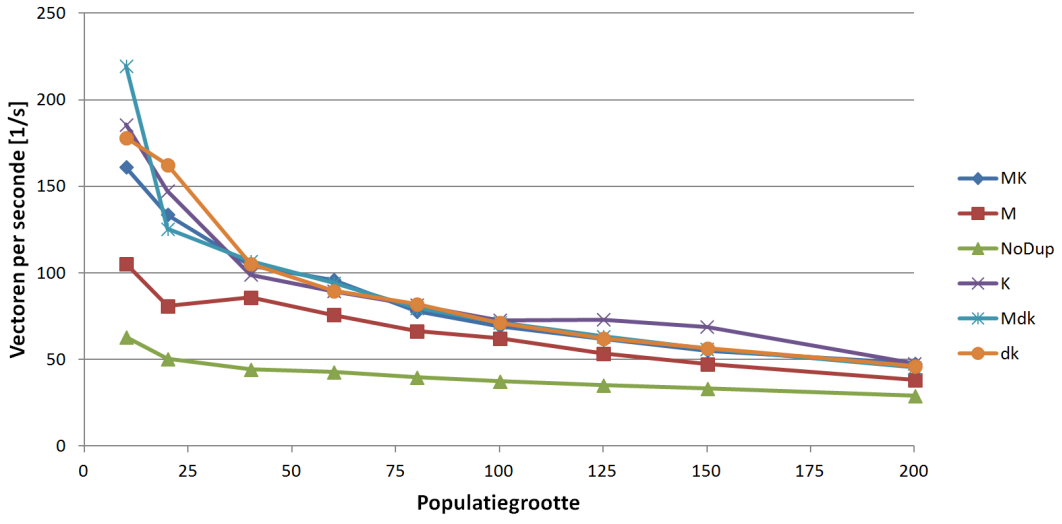
dk De combinatie van de twee bovenstaande correcties, wat eigenlijk geen enkele duplicatie meer toelaat, maar met deze aanpak worden ze voorkomen in plaats van ze achteraf niet te aanvaarden.

Er is gekozen om eerst met minimalisatie van de determinant te werken, omdat dit toelaat sneller een grote hoeveelheid data te produceren en voor het bestuderen van de optimale mutatie-kruising verhouding even bruikbaar bleek als het meer representatieve maximaliseren van de determinant.

In figuren 5.3 en 5.4 zijn de statistieken te zien voor alle bovenstaande cases als gemiddeldes van 10 pogingen. Er is voor minimalisatie van de determinant duidelijk



Figuur 5.3: Voor elk duplicatiegedrag is tegenover de populatiegrootte uitgezet: het gemiddeld aantal generaties (linksboven) en beschouwde vectoren (rekenkracht)(rechtsboven), de gemiddelde rekentijd (linksonder) en de variantie (rechtsonder).



Figuur 5.4: Vergelijking van het beschouwde aantal vectoren per seconde voor de testcases, aanvullend bij figuur 5.3.

geen relevante invloed op het aantal generaties en het beschouwd aantal vectoren. In de variantie zijn geen opvallende uitschieters te zien (enkele datapunten voor kleine populaties zijn weggelaten voor de duidelijkheid). Merk wel op dat ze het laagst is in de zone die eerder werd aangewezen als optimale populatiegrootte (rond de 100) voor deze topologiegrootte bij maximalisatie van de determinant in plaats van minimalisatie.

De rekentijd vertoont echter duidelijke verschillen die niet aan louter toeval toegeschreven kunnen worden. Het limiteren van het duplicatiegedrag door het verwerpen van geproduceerde duplicaten, vertraagt het optimalisatieproces, gezien de ligging van de M- en NoDup-lijn. Niet geheel onverwacht omdat er geen tijd gewonnen is door het reduceren van het aantal generaties. Zelfkruising uitsluiten heeft geen noemenswaardig effect; MK, Mdk en dk liggen erg dicht bij elkaar. Net zoals NoDup produceert dk geen duplicaten, maar dk is toch snel omdat het duplicaten vermijdt in plaats van verwerpt. De kruisingskans op 100% instellen lijkt de laagste rekentijd op te leveren (K).

Met de verplaatsing als objectief zal het verschil in rekentijd maar enkele seconden bedragen van de meerdere uren dat het algoritme in uitvoering zal zijn, gezien FE berekeningen het grootste deel van de tijd in beslag zullen nemen. Ten tweede worden vectoren die reeds berekend zijn, niet opnieuw ter berekening voorgelegd aan Abaqus. Als alle cases een gelijk aantal vectoren doorlopen, zoals blijkt uit de figuren, is de oplossing die dit doet met veel duplicaten dus te verkiezen boven de andere. Dit is onder standaardomstandigheden MK of K.

Zoals eerder aangehaald, is het voordeel van het minimaliseren van de determinant de lagere rekentijd. De resultaten zijn echter minder representatief omdat beide objectieven hetzelfde resultaat nastreven en er daardoor lokaal gezocht wordt, in plaats van op een breed front. Om een beter besluit te maken, geeft tabel 5.1 de gemiddeldes voor vijftig pogingen met een populatie van honderd en bij maximalisatie van de determinant.

Dit resultaat is anders. Het vermijden van duplicaten heeft een positieve invloed op de convergentie, en dk scoort slechter dan NoDup, niet andersom. De oorzaak van de slechte prestatie van dk ligt waarschijnlijk bij het feit dat er geen pure mutaties zijn, i.t.t. bij NoDup, dit lijkt niet wenselijk. Naar de rekentijd kijken is niet interessant gezien er nu wel verschillen tussen het gemiddeld aantal generaties zijn. NoDup scoort in dit opzicht duidelijk het best en daarom is deze aanpak voor een aantal mutatiekansen bekeken. De waarde van 95% blijft een goede keuze. Door na te gaan wanneer een duplicatie optrad, kan ook vastgesteld worden dat er onder normale omstandigheden (MK) gemiddeld 1958 duplicaties waren. Dit is erg veel, maar maakt het aantal unieke vectoren in deze situatie nog steeds groter dan met NoDup. De NoDup code zal dus behouden worden.

Een kanttekening hierbij is het verschil in variantie van het aantal generaties. Waarden

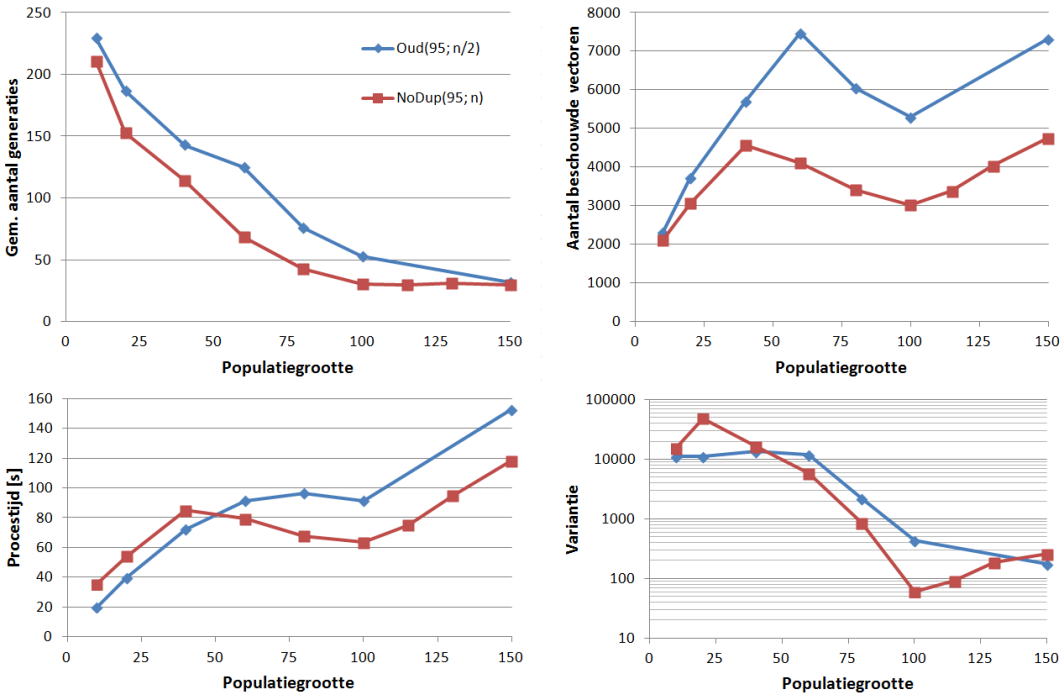
Case	Kruisingskans	Generaties	Vectoren	Rekentijd [s]	Variantie
MK	95	74	7400	127,30	7152
M	95	47,14	4714	92,66	857
K	100	63,62	6362	93,30	8115
Mdk	95	66,66	6666	101,59	5927
dk	100	73,46	7346	106,51	13571
NoDup100	100	37,44	3744	110,61	3088
NoDup95	95	34,68	3468	96,31	401
NoDup90	90	37,88	3788	106,46	478
NoDup85	85	34,86	3486	98,46	345

Tabel 5.1: Vergelijking van de prestaties voor verschillende cases bij maximalisatie van de determinant.

met een erg uiteenlopende variantie zijn niet goed vergelijkbaar, gezien een hoge variantie wijst op uitbijters die de cijfers negatief beïnvloeden. Deze uitbijters zijn anderzijds vrij zeker een gevolg van de toename van duplicaten die de convergentie ernstig kan storen. De lage variantie kan in het geval van NoDup een extra argument voor deze aanpak zijn. Enkele extra testruns hebben aangetoond dat deze lage variantie niet gewoon toeval is. Tests met een vergelijkbare variantie als in de andere cases bekomen om een meer objectieve vergelijking te maken, zou wel eens erg moeilijk kunnen zijn en niet erg correct.

Figuur 5.5 bewijst dat ook in dit geval een populatiegrootte van om en bij de honderd aangewezen is. De trends zijn dit keer zelfs duidelijker zichtbaar. Ter vergelijking zijn ook de vorige resultaten uit figuur 4.3 zichtbaar. Bemerk wel dat, naast het implementeren van de anti-duplicatiemaatregelen, ook het mutatie-interval dubbel zo groot genomen is (n in plaats van n/2), al zal dit minder invloed gehad hebben. Het gemiddelde is telkens genomen voor dertig pogingen (tien in de blauwe data). De performantie is duidelijk gestegen.

Tot slot nog een verduidelijking waarom duplicaten niet volledig worden vermeden door bijvoorbeeld alle vectoren die een duplicaat zijn op te sporen en te verwijderen. De reden waarom dit niet wenselijk is, is omdat het objectief van minimale massa discrete waarden aanneemt tussen 8 en $n-1$ met n het aantal bits in het genoom. Vaak is dit aantal waarden dus lager dan de populatiegrootte. De optimalisatie convergeert echter naar een eindoplossing met een volledige populatie van pareto-optimale oplossingen in één front. Er kunnen maar even veel verschillende pareto-optimale oplossingen zijn als er massawaarden zijn. Het voorkomen van duplicaten in het eindresultaat is dus allerminst onverwacht. Het is net een teken dat een goede convergentie bereikt is.



Figuur 5.5: Resultaten van testcase 2 uit deel 4.2 met en zonder NoDup. Uitgezet tegenover de populatiegrootte het gemiddeld aantal generaties (linksboven) en beschouwde vectoren (rekenkracht)(rechtsboven), de gemiddelde rekentijd (linksonder) en de variantie (rechtsonder).

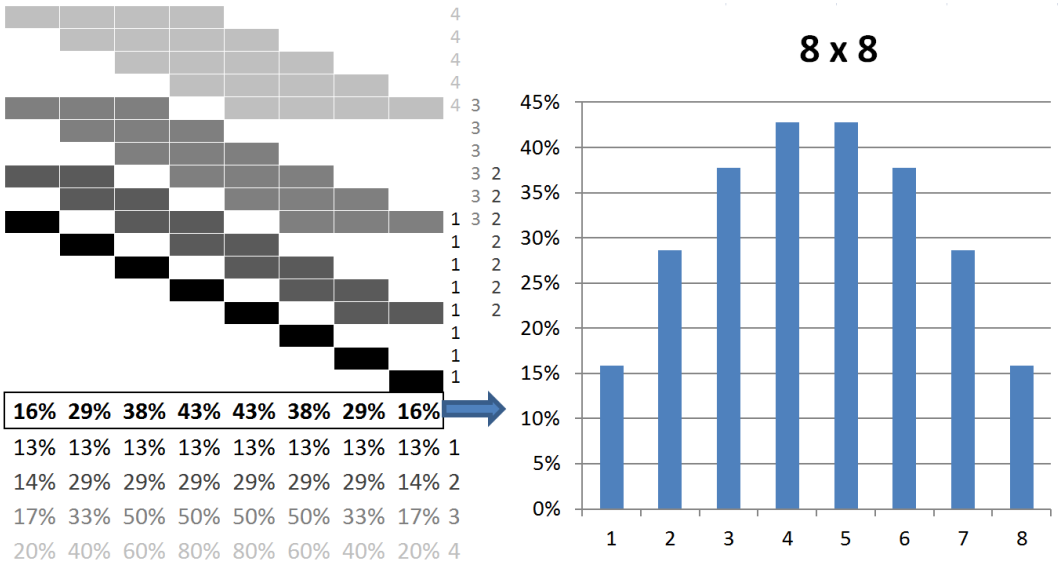
5.2 Ongelijke verdeling van de kruisingskans

Om een kruising uit te voeren, wordt eerst willekeurig gekozen hoeveel rijen of kolommen gewisseld moeten worden. Dit zijn er nooit meer dan de helft. Daarna wordt willekeurig gekozen wat de eerste kolom of rij van dit blok zal zijn, rekening houdend met de breedte. Dit heeft tot gevolg niet elke rij of kolom een even grote kans heeft om gewisseld te worden.

5.2.1 Grootte van het effect

Figuur 5.6 geeft een voorbeeld voor een topologie die acht rijen hoog is of acht kolommen breed. De maximale breedte van het gekruiste gebied zal dus vier zijn. Er wordt eerst met gelijke kans willekeurig gekozen tussen de vier mogelijke breedtes, aangegeven in verschillende tinten grijs. Voor een kruising van één kolom is de kans gelijk verdeeld, voor de kruising van vier kolommen komen de buitenste kolommen (1 en 8) maar in één van de vijf mogelijke kruisingen voor, terwijl de centrale kolommen (4 en 5) in vier van de vijf mogelijkheden voorkomen. De kans dat bij een kruising een bepaalde kolom gewisseld wordt, is weergegeven in de grafiek rechts in figuur 5.6. De centrale kolommen worden in bijna de helft van de gevallen gewisseld, bijna drie maal zo frequent als de kolommen aan de buitenkant. Voor grotere topologieën

neemt dit effect verder toe.



Figuur 5.6: Visualisatie van alle kruisingsmogelijkheden (links) en de resulterende kansverdeling over de kolommen of rijen (rechts).

5.2.2 Besluit

De manier waarop kruisingen plaats vinden, zorgt ervoor dat het centrale deel van de topologie veel dynamischer is in het genetisch proces. Het is niet duidelijk of dit echt een probleem is. Gezien het om soft robots gaat met een holte waar druk op gezet zal worden, is het centrum van de topologie ook daadwerkelijk het belangrijkst, het heeft het meeste invloed op het resultaat.

Dit fenomeen is niet verholpen, maar is interessant voor verder onderzoek. Een eenvoudige oplossing is willekeurig twee partitielijnen te kiezen, in plaats van eerst een breedte en dan een startpunt te kiezen. Dit kan echter neveneffecten hebben, omdat de uitgewisselde gebieden groter zullen zijn en het uitwisselen van één enkele kolom of rij zelden zal voorkomen, wat de convergentie kan vertragen.

5.3 Selectie operator

Tot nog toe werd bij het selecteren van twee ouders om een kruising uit te voeren nog geen selectie operator toegepast. De ouders worden met gelijke kans uit de *genepool* geselecteerd. Deb et al. vermelden echter dat NSGA-II gebruikt maakt van toernooiselectie (*tournament selection*). Daarbij wordt telkens de beste uit twee mogelijkheden als ouder gekozen.[6]

Er zijn enkele redenen om dit pas als laatste te implementeren. Ten eerste is het probleem van deze masterproef epistatisch, dit betekent dat de genen onderling

afhankelijk zijn. De genen van twee goede oplossingen kunnen een kruising produceren die slechter is dan beide ouders. Het is dus niet zeker of het selecteren van een ouder op basis van hun fit een sterke verbetering is. Toernooiselectie heeft namelijk ook een nadeel, door sommige vectoren bij voorkeur te selecteren worden vaker dezelfde vectoren gekruist en neemt het duplicatiegedrag toe. In deel 5.1 is echter net aangetoond dat dit dupliceergedrag nadelig is. Verder leverde het optimaliseren van de kruisings- en mutatiekans geen duidelijk optimum, dit zal ook niet het geval zijn met toernooiselectie. Er is dus geen bezwaar om dit pas achteraf te implementeren, dan kan het effect vergeleken worden met het geval zonder toernooiselectie.

5.3.1 Effect van toernooi selectie

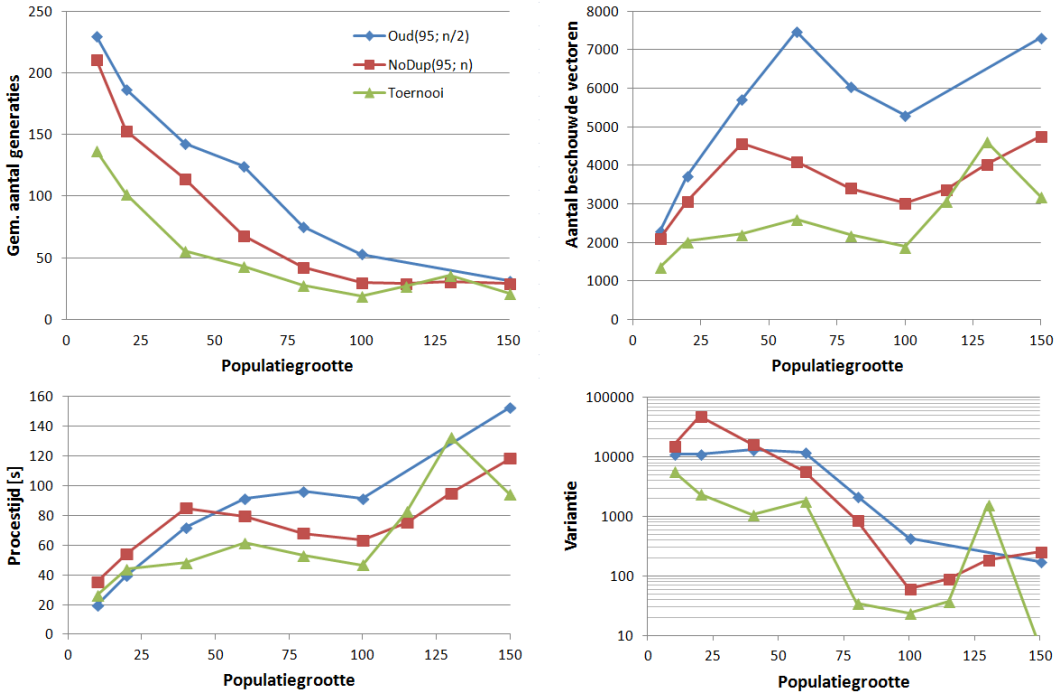
Toernooiselectie werd geïmplementeerd in `Genepool.tournament()` (zie appendix F en G) en aan tests onderworpen.

In tabel 5.2 zijn de resultaten te zien uit tabel 5.1, aangevuld met resultaten verkregen met toernooiselectie in dezelfde tests. Toernooiselectie biedt duidelijk een aanzienlijke verbetering, het gemiddeld aantal generaties daalt met 20-25%. Het toegenomen duplicatiegedrag doet de variantie niet stijgen. De optimale kruisingskans blijft 95%.

Case	Kruisingskans	Generaties	Vectoren	Rekentijd [s]	Variantie
NoDup	100	37,44	3744	110,62	3088
NoDup	95	34,68	3468	96,32	401
NoDup	90	37,88	3788	106,47	478
NoDup	85	34,86	3486	98,46	345
Toernooi	100	27,7	2770	74,58	310
Toernooi	95	27,64	2764	71,48	485
Toernooi	90	28,68	2868	79,76	103
Toernooi	85	29,9	2990	82,53	508

Tabel 5.2: Vergelijking van de resultaten met toernooiselectie met de resultaten uit tabel 5.1.

Om te bewijzen dat ook de optimale populatiegrootte niet verandert, zijn in figuur 5.7 de resultaten herhaald uit figuur 5.5. De groene lijn illustreert dat met toernooiselectie steeds een verbetering bekomen wordt, zowel voor het aantal bezochte vectoren, de rekentijd en de variantie, met als lokaal optimum een populatie van 100. De datapunten 40 en 130 zijn niet helemaal representatief zoals uit de variantie duidelijk blijkt (respectievelijk onverwacht laag en hoog), maar dit verhindert niet om op basis van de figuur tot een duidelijk conclusie te komen.



Figuur 5.7: Resultaten van testcase 2 uit deel 4.2 zonder NoDup (blauw) met NoDup (rood) en met NoDup en toernooi selectie (groen). Uitgezet tegenover de populatiegrootte het gemiddeld aantal generaties (linksboven) en beschouwde vectoren (rekenkracht)(rechtsboven), de gemiddelde rekentijd (linksonder) en de variantie (rechtsonder).

5.4 Versnellen van het sorteeralgoritme

Deb et al. vermelden dat het snel *non-dominated* sorteren verder kan versneld worden door de tweede lus af te breken nadat er genoeg vectoren in fronts gesorteerd zijn om de volledige populatie te vullen. Dit kan geïmplementeerd worden met een teller en een toevoeging aan de while conditie, in appendix G is dit in de `Genepool.fast_nondominated_sort(P)` aangeduid met `# SPEEDUP`.

Ondanks dat dit duidelijk een verbetering inhoudt, blijkt uit tests niet dat de code sneller is. In tabel 5.3 zijn de gemiddelde resultaten van 50 runs vergeleken voor twee tests met een vergelijkbare variantie. Er is geen merkbare versnelling. Door het sorteren op een dergelijke manier af te breken, kunnen de fronts die volledig verworpen worden voor de volgende populatie achteraf ook niet meer gevisualiseerd worden.

	Generaties	Vectoren	Rekentijd [s]	Variantie	Tijd/Generatie [s]
Voor	27,04	2704	69,166	176,08	2,558
Na	27,38	2738	68,841	171,20	2,514

Tabel 5.3: Effect van het vroegtijdig afbreken van de tweede lus van het sorteeralgoritme op de rekentijd.

5.5 Besluiten

In dit hoofdstuk werden mogelijke verbeteringen van het genetisch optimalisatieprogramma onderzocht.

De oorspronkelijke implementatie van kruising en mutatie stond toe dat vectoren met zichzelf gekruist werden en dat bij het uitblijven van een kruising of zelfkruising ook mutaties konden uitblijven. Het vermijden van zelfkruising levert voornamelijk een constantere kans op 'echte' kruisingen en het verwerpen van nakomelingen, identiek aan een ouder, leverde een betere en meer betrouwbare convergentie op. Dit sluit het ontstaan van duplicaten uiteraard niet volledig uit, maar het is dan ook niet zeker of dat wel wenselijk is.

De implementatie van toernooiselectie, waarbij ouders gekozen worden als de beste van twee willekeurig geselecteerde alternatieven, is een aanzienlijke verbetering. De toevoeging aan de code kan behouden worden met de overige parameters ongewijzigd.

De tweede lus van het algoritme voor snel niet-gedomineerd sorteren kan vroegtijdig afgebroken worden wanneer voldoende vectoren gesorteerd zijn om een nieuwe populatie op te bouwen. Dit biedt echter geen waarneembare snelheidswinst.

Met de manier waarop kruisingen gebeuren, bestaat mogelijk nog een probleem dat nog niet verholpen is. De kans op uitwisseling van genetische informatie wordt kleiner van het centrum naar de randen van de topologie toe. Er is echter niet genoeg informatie voorhanden om te besluiten of dit goed of slecht is. Het implementeren en vergelijken van alternatieve kruisingsmethoden is aangewezen.

Hoofdstuk 6

Resultaten voor testproblemen

6.1 Hardware en parameters

Dit hoofdstuk presenteert enkele resultaten voor testproblemen. Ze hebben elk een andere topologiegrootte, een eenvoudige verplaatsing van het eindpunt rechtsonder als objectief en een inklemming linksboven. De convergentiesnelheid en de kwaliteit van de oplossingen wordt voor deze testproblemen beschouwd. Er waren telkens twee objectieven actief, minimalisatie van de afwijking van het eindpunt en van de massa. Voor elke simulatie werden de volgende parameters gebruikt:

population_size: 20 (eerste, 5x5) of 100 (alle overige)

cell_edge: $0,1\ mm$

element_type: CPE4R

materiaalmodel: 2de Orde Ogden, parameters uit [9]

pressure: $0,5\ MPa$

De eerste optimalisatie werd uitgevoerd op een *Medion The Touch 300 S6615T* laptop met *Intel Core i7-4500U*. De softwareversies zijn Python 3.4.3 en Abaqus 6.13-4.

De overige optimalisaties werden uitgevoerd op een processor van de ThinKing cluster van het Vlaams Supercomputer Centrum (https://www.vscentrum.be). De toegewezen rekenkern verschilt van test tot test; in sommige tests is het een kern met twee 10-core *"Ivy Bridge"Xeon E5-2680v2* CPU's (2.8 GHz, 25 MB level 3 cache, 64-128GB RAM), in andere een kern met twee 12-core *"Haswell"Xeon E5-2680v3* CPU's (2.5 GHz, 30 MB level 3 cache, 64GB RAM), waarvan 20 cores gebruikt werden.[7] Softwareversies Python 2.7.6 en Abaqus 6.13-3.

De NoDup verbetering, zie deel 5.1, was nog niet actief in de eerste optimalisatie (5x5) en toernooiselectie, zie deel 5.3, niet bij de eerste twee optimalisaties (5x5 en 6x6).

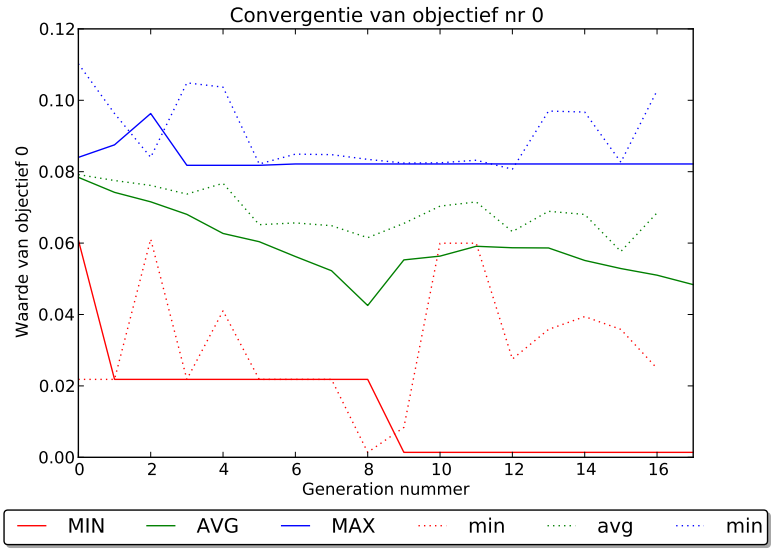
6.2 Beweging naar rechtsboven, 5x5

Voor de verplaatsing werd volgend eindpunt opgegeven:

dx $0,2\ mm$

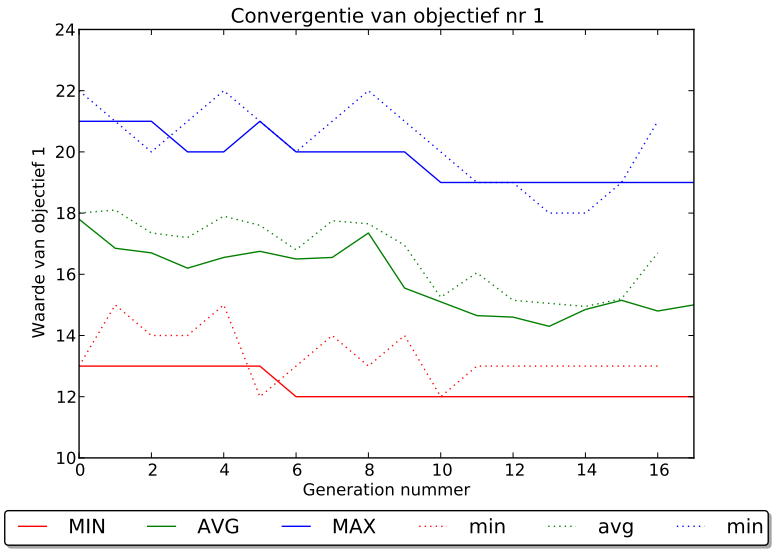
dy $0,2\ mm$

In figuur 6.1 en 6.2 en is de convergentie van beide objectieven te zien. Het is duidelijk dat er een afweging wordt gemaakt tussen beide objectieven die concurreren. Hoewel in generatie 8 toevallig een zeer goede oplossing ontdekt wordt wat de afwijking betreft, tonen de grafieken dat de gemiddelde afwijking slechter wordt, ten voordele van de gemiddelde massa, die daalt.



Figuur 6.1: Evolutie van de maximale, minimale en gemiddelde waarde van de afwijking in de populatie en offspring.

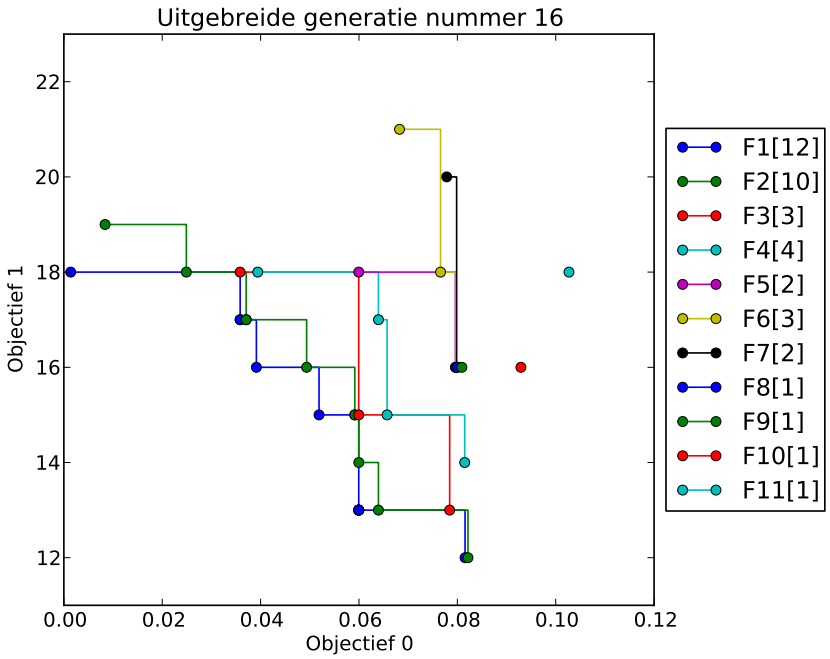
In figuur 6.3 zijn alle niet-gedomineerde fronts voor de twee objectieven te zien voor de laatste generatie. In het eerste front zijn 12 vectoren aanwezig, die over heel het front verspreid zijn. Voor elke massa tussen 12 en 18 bestaat er een Pareto-optimale oplossing met lokaal optimale afwijking in het eerste front. In het geval er geen vervorming zou zijn, zou objectief 0 de waarde $0,08$ aannemen. De oplossingen met de laagste massa zijn dus verre van ideaal wat de verplaatsing betreft. Daarnaast



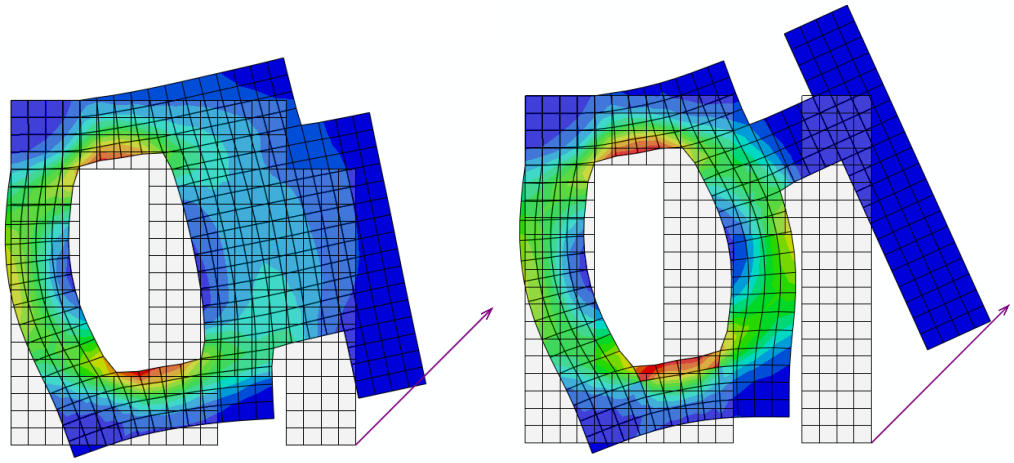
Figuur 6.2: Evolutie van de maximale, minimale en gemiddelde waarde van de massa in de populatie en offspring.

leveren de kruisingen en mutaties ook oplossingen op die zwaarder zijn of de verkeerde kant uit bewegen, maar deze worden uiteraard niet geselecteerd. Figuren zoals figuur 6.3 worden automatisch gegenereerd voor elke generatie en kunnen tot een animatie verwerkt worden.

In figuur 6.4 (a) is de vector te zien die de beste benadering voor de verplaatsing voorstelt die in generatie één is gevonden, (b) is de vector verantwoordelijk voor de verbetering in generatie negen. Deze topologie is erg interessant omwille van de zeer goede benadering en doet als het ware aan een zuiger denken.



Figuur 6.3: Fronten voor generatie 16. De waarde van objectief 0 is de som der kwadraten $\Delta x^2 + \Delta y^2$.



(a) Vector b58c514f7832bde1b06cb19723727f63 (b) Vector 4c3e1ffc6df12b316d31240062f527be

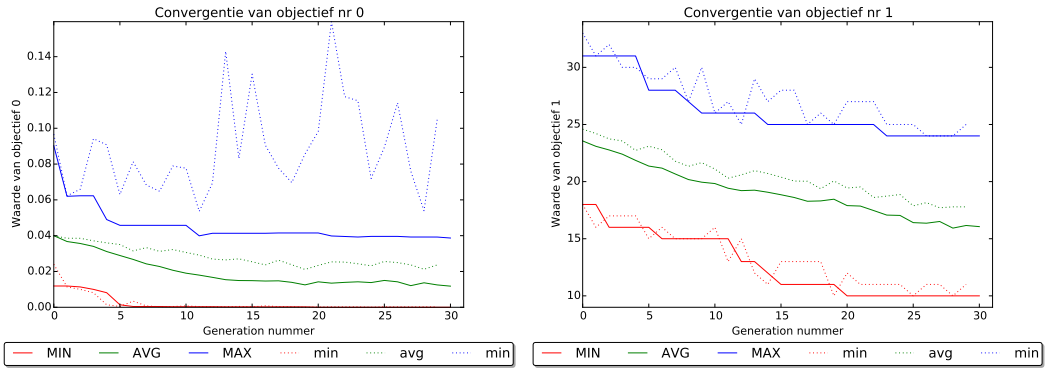
Figuur 6.4: FEA resultaten van twee oplossingen, met vervormingen, Von Mises spanningen in kleur en de gewenste verplaatsing als paarse pijl.

6.3 Beweging naar rechts, 6x6

Voor de verplaatsing werd volgend eindpunt opgegeven:

dx $0,2\ mm$

dy $0\ mm$



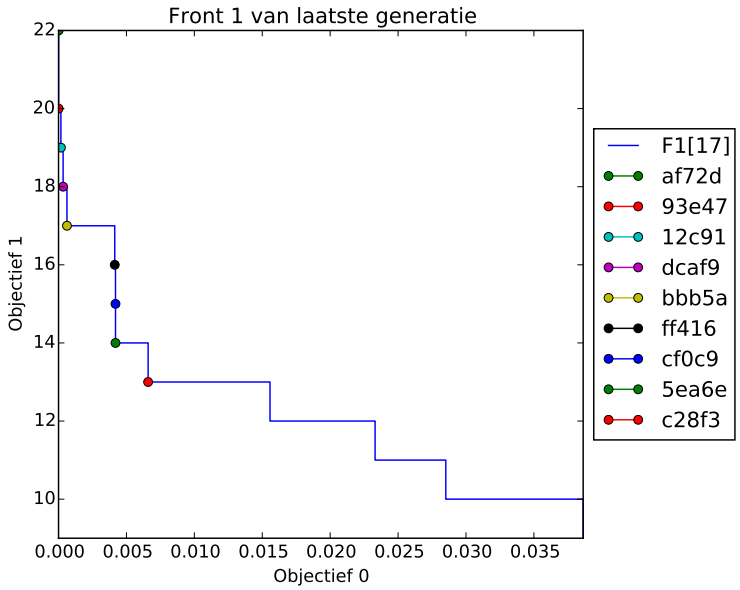
(a) Afwijking (b) Massa

Figuur 6.5: Evolutie van de maximale, minimale en gemiddelde waarde van de objectieven in de populatie en offspring.

In figuur 6.5 is te zien dat de minimale afwijking al snel zeer kleine waarden aanneemt. Dit wil zeggen dat er al een oplossing bestaat die de afwijking sterk minimaliseert en zeer geschikt is. Het betekent echter niet dat er niet verder geoptimaliseerd kan worden. Zoals in (b) duidelijk is, daalt de minimale en gemiddelde massa nog gestaag, en kunnen er dus oplossingen gevonden worden met een vergelijkbare lage afwijking maar een lagere massa.

In figuur 6.6 is het eerste front te zien. Zoals men kan verwachten, bevat het voor bijna elke waarde van de massa een vector die de afwijking minimaliseert. Voor een massa van 17 tot 24 is deze afwijking zeer klein (10^{-4} tot 10^{-6}). Vector af72d is dus niet noodzakelijk de beste keuze, omdat er lichtere vectoren bestaan die nauwelijks inboeten aan nauwkeurigheid, de verschillen in nauwkeurigheid vallen met vertrouwen binnen de onzekerheid op het model.

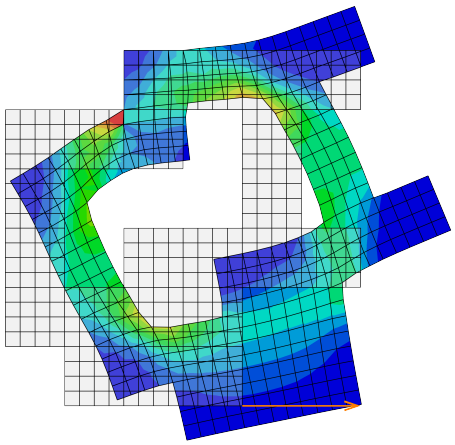
In tabel 6.1 zijn voor elke oplossing de scores op de objectieven gegeven, in figuur 6.7 en 6.8 zijn de topologieën weergegeven. De vervormde en onvervormde voorstellingen uit Abaqus zijn er te zien met een oranje pijl die de gewenste verplaatsing van het eindpunt aanduidt.



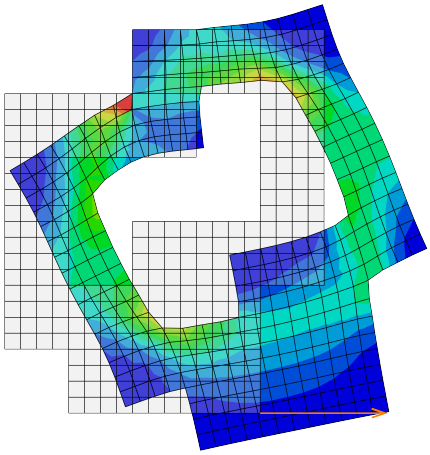
Figuur 6.6: Front 1 van de laatste generatie, met aanduiding van enkele vectoren (eerste 5 tekens van hun md5-naam).

Md5-naam	Afwijking ($\Delta x^2 + \Delta y^2$)	Massa
af72d90acb3f87eb880f7bd0de6d1072	$7,6166 \cdot 10^{-6}$	22
93e47472e4056013caa2ad981875e36d	$7,6686 \cdot 10^{-6}$	20
12c919eb7b2617aa9abae5bf1e352a79	0,00017434	19
dcaf9c79414e885e8dd375453d8814f2	0,00033665	18
bbb5aed0c9c87f912e4eacfe674db287	0,00062138	17
ff416a5997b6496b05bccbe137debef3	0,0041394	16
cf0c962081b6cd0306e496f39988401d	0,0041939	15
5ea6e14ba913d9b4ec1b354d12c44be3	0,0041958	14
c28f3b1582849de1245893c20df366c0	0,0065967	13

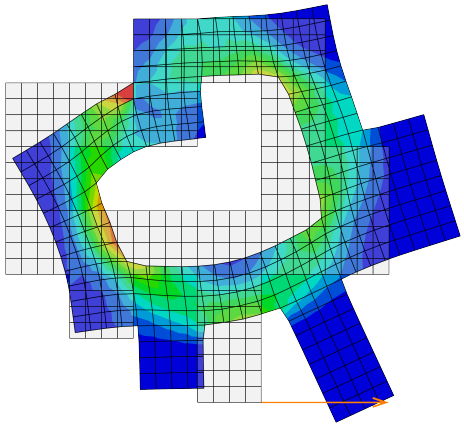
Tabel 6.1: Tabel met md5-namen, afwijkingen en massa's van enkele oplossingen voor het 6x6 testprobleem met beweging naar rechts.



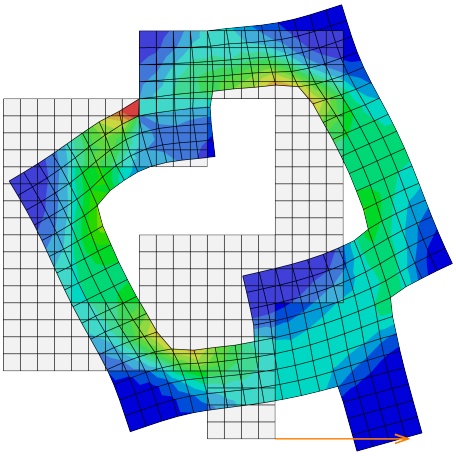
(a) af72d90acb3f87eb880f7bd0de6d1072



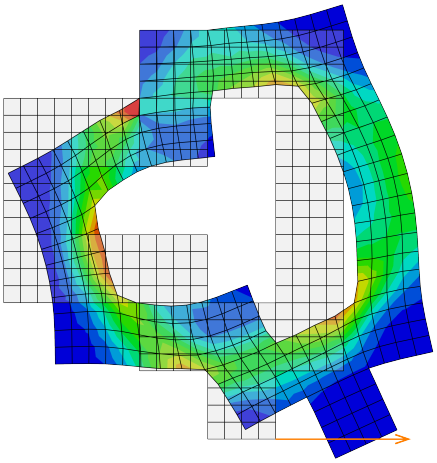
(b) 93e47472e4056013caa2ad981875e36d



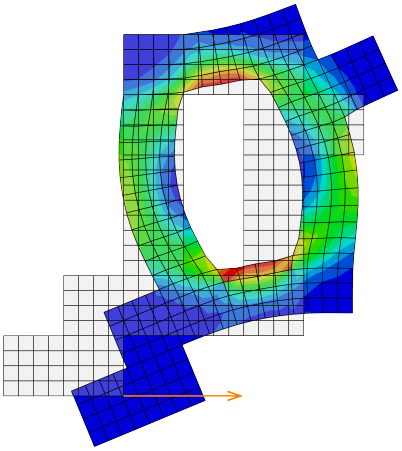
(c) 12c919eb7b2617aa9abae5bf1e352a79



(d) dcaf9c79414e885e8dd375453d8814f2

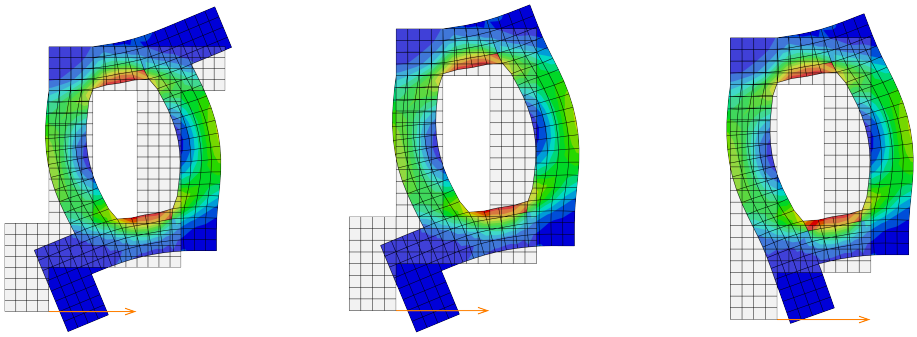


(e) bbb5aed0c9c87f912e4eacfe674db287



(f) ff416a5997b6496b05bccbe137debef3

Figuur 6.7: De zes eerste oplossingen uit tabel 6.1. De oranje pijl is de gewenste verplaatsing van het eindpunt.



(a) cf0c9 (b) 5ea6e (c) c28f3

Figuur 6.8: De drie laatste oplossingen uit tabel 6.1.

In figuur 6.7 worden enkele veronderstellingen bevestigd. Figuur (a) heeft weliswaar de laagste verplaatsing, maar (c) en (d) zijn duidelijk variaties op (a) met een lagere massa (respectievelijk 2 en 4 minder) die amper inboeten aan nauwkeurigheid. Figuur (c) toont aan dat er nog ruimte is voor verbetering, want in deze topologie kunnen gemakkelijk drie cellen verwijderd worden. Met een massa van 16 zou deze nieuw bekomen topologie waarschijnlijk (d), (e) en (f) uit het eerste front domineren.

In figuur 6.8 zijn de lichtere oplossingen te zien die al een behoorlijke afwijking van de verplaatsing vertonen. Net als Figuur 6.7 (f), zijn ze variaties op dezelfde structuur. Ook hier is te zien dat niet-functionele uitstekende cellen verwijderd kunnen worden met weinig verlies in nauwkeurigheid, maar bij oplossing (c) is de daling in nauwkeurigheid wel duidelijk. In figuur 6.6 is dit ook duidelijk te zien, het zwarte, blauwe en groene punt liggen zo goed als op een verticale lijn. Topologie 5ea6e zal dus steeds de voorkeur genieten over ff416 en cf0c9, indien diens afwijking nog aanvaardbaar is voor de beoogde toepassing uiteraard.

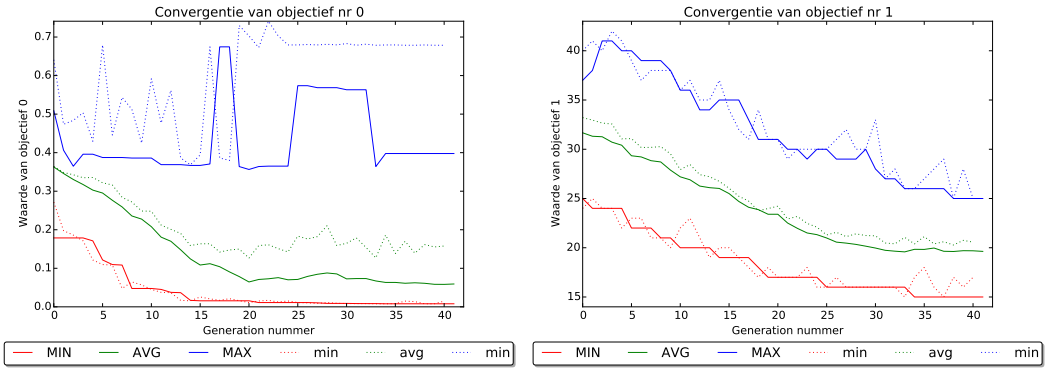
Interessant om te vermelden is dat in de 3100 berekende vectoren uit de 30 generaties reeds 61 duplicaten voorkwamen, die toevallig geproduceerd werden zonder een andere vector rechtstreeks te dupliceren.

6.4 Veegbeweging naar links, 4x12

Voor dit resultaat werd een langwerpige 4x12 oplossingsruimte gekozen en een sterke beweging naar links, met als doel een oplossing te bekomen die een veegbeweging uitvoert. Voor de verplaatsing werd volgend eindpunt opgegeven:

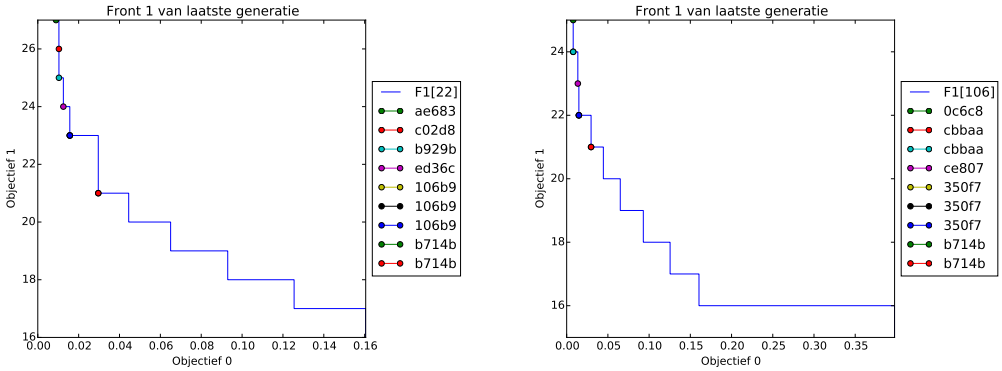
dx $-0,6\ mm$

dy $0\ mm$



(a) Convergentie van afwijking (b) convergentie van massa

Figuur 6.9: Evolutie van de maximale, minimale en gemiddelde waarde van de objectieven in de populatie en offspring voor het 4x12 testprobleem.



(a) Generatie 28 (b) Generatie 40

Figuur 6.10: Analyse van front 1 met aanduiding van enkele vectoren (eerste 5 tekens van hun md5-naam) voor het 4x12 testprobleem.

Dit zijn de eerste resultaten waarbij *tournament selection* toegepast wordt, wat het sneller ontstaan van duplicaten verklaart.

In figuur 6.9 (a) is voor de afwijking wederom de snelle convergentie te zien naar zeer lage waarden, waarna het evenwicht meer verschuift van globaal naar lokaal zoeken, rond generatie twintig. Minieme verbeteringen in de afwijking worden nog bekomen en de gemiddelde massa van de bereikte oplossingen blijft dalen zoals figuur (b) aangeeft, maar aan een trager tempo dan aanvankelijk.

In figuur 6.10 is de analyse van twee fronts te zien. Na een initiële 28 generaties, zijn er nog 12 bijkomende uitgevoerd, omdat uit de resultaten bleek dat deze nog

verbeterd konden worden omwille van de aanwezigheid van niet-functionele cellen. Het rode punt in beide fronten is een gemeenschappelijke oplossing. Alle oplossingen van lagere afwijking zijn verbeterd tussen beide generaties, zoals ook te zien is in tabel 6.2.

Ook is te zien dat in generatie 40 de oplossing met massa 25 beter scoort dan die met massa 25, 26 en 27 in generatie 28. Optima voor een massa van 26 en 27 komen bijgevolg niet meer voor in het eerste front; het front is dus gekrompen aan de linkerkant. Aan de rechterkant is het front gegroeid: er is een oplossing met massa 15 gevonden (de rechterbenedenhoek is steeds een punt van het front in de figuren). Dit is de lichtste oplossing, maar met een enorme afwijking: ongeveer $0,4$ terwijl de afwijking zonder vervorming maar $0,36$ zou bedragen. De structuur beweegt dus weg van het objectief.

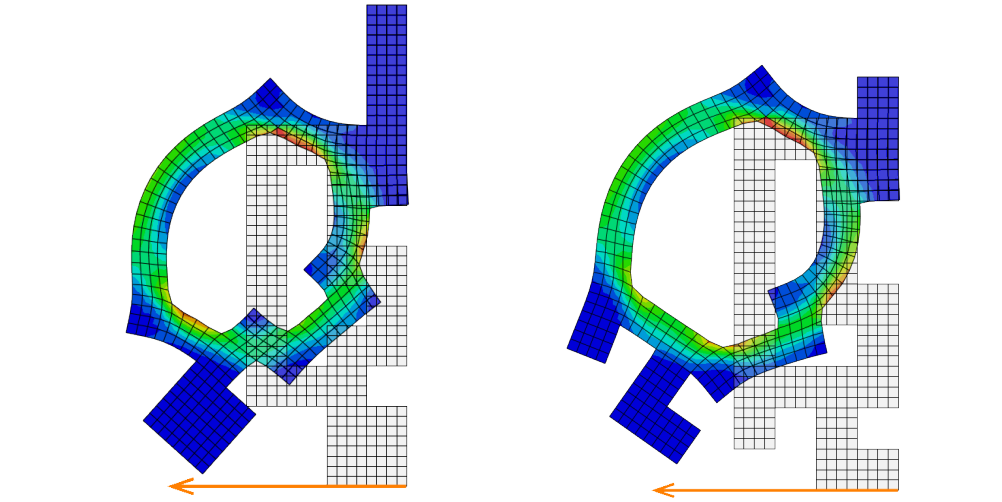
Tot slot geeft 6.10 (b) aan dat er 106 vectoren in het eerste front zitten, meer dan de populatie. Gezien het front uit elf punten bestaat, zijn dat zeer veel duplicaten, maar zoals in deel 5.1 besproken is dat allerminst onverwacht als eindtoestand na een voldoende aantal generaties.

		Generatie 28		Generatie 40			
Naam	Fig.	Afwijking	Massa	Naam	Fig.	Afwijking	Massa
				0c6c8	ii-a	0,007908	25
				cbbaa	ii-b	0,007911	24
ae683	i-a	0,008922	27				
c02d8	i-b	0,010345	26				
b929b	i-c	0,010354	25				
ed36c	i-d	0,012516	24				
				ce807	ii-c	0,013547	23
				350f7	ii-d	0,014810	22
106b9	i-e	0,015674	23				
b714b	i-f	0,029610	21	b714b	i-f	0,029610	21

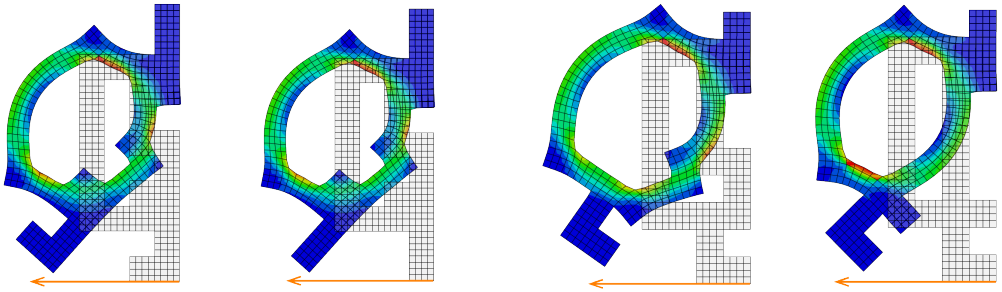
Tabel 6.2: Overzicht van de oplossingen van het 4x12 testprobleem uit twee generaties, gerangschikt volgens minimale afwijking.

In figuur 6.11 tot slot zijn alle oplossingen weergegeven in vervormde en onvervormde Abaqus voorstelling. Alle oplossingen aangegeven in figuur 6.10 en tabel 6.2 zijn weergegeven.

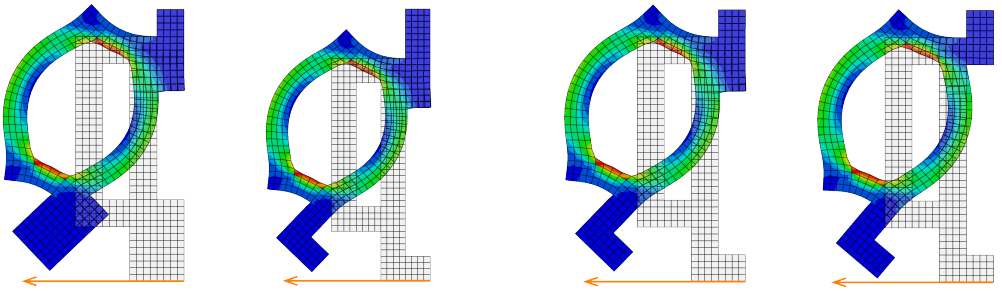
Enkel generatie 28 beschouwend (linkerzijde, figuur 6.11 (i-*)) kan men zien dat de beste oplossing nog drie niet-functionele cellen heeft: twee bovenaan aan de inklemming en één onderaan. Figuren (i-b) en (i-c) zijn variaties met andere niet-functionele cellen. Hier is (i-b) niet interessant omdat (i-c) lichter is en het verschil in afwijking triviaal is. In (i-d,e,f) zijn variaties te zien van een andere structuur die



(i-a) ae683 (ii-a) 0c6c8



(i-b) c02d8 (i-c) b929b (ii-b) cbbaa (ii-c) ce807



(i-d) ed36c (i-e) 106b9 (ii-d) 350f7 (i-f) b714b

Figuur 6.11: De beste oplossingen uit generatie 28 (links, i) en 40 (rechts, ii) voor het 4x12 testprobleem.

lichter maar minder nauwkeurig is door de andere vorm van de holte. Enkel in (i-f) van deze variaties komen geen niet-functionele cellen voor.

In generatie 40, rechts in 6.11, is te zien dat de beste oplossingen gebruik maken van nog een andere vorm van holte. In geen enkele oplossing komen nog niet functionele cellen voor aan de inklemming. De tweede oplossing geniet bovendien de voorkeur boven de eerste. Ze is lichter door het verdwijnen van de niet-functionele cel en het verschil in afwijking is triviaal zoals in tabel 6.2 duidelijk wordt. Figuren 6.11 (ii-c,d) zijn variaties op de eenvoudige structuur van de gemeenschappelijke oplossing (i-f). De toegevoegde cel om (ii-d) uit (i-f) te bekomen, biedt een aanzienlijke verbetering van de afwijking, al kon deze verkeerdelijk aanzien worden als niet-functioneel. Ze biedt wel degelijk steun aan de opgeblazen holte. De verdere toevoeging naar (ii-c) is echter niet-functioneel, de verbetering van de afwijking is erg beperkt.

Globaal is het resultaat cbbaa uit figuur 6.11 (ii-b) de beste keuze, tenzij de beweging een lagere nauwkeurigheid vereist en een oplossing met lagere massa gekozen kan worden. Voor bepaalde toepassingen kunnen de afwijkingen Δx en Δy een andere nauwkeurigheid vereisen. In dat geval kunnen beide verplaatsingen best afzonderlijk als objectief gekozen worden. Het kiezen van een gewenste oplossing uit een dergelijk 3-dimensionaal front kan moeilijk zijn, tenzij een duidelijke grens wordt gesteld aan Δx en Δy. Omwille van de moeilijkheid om dergelijke resultaten bevattelijk voor te stellen, wordt deze optie in dit hoofdstuk niet uitgewerkt.

Opvallend is dat een oplossingsruimte van 4x10 in plaats van 4x12 volstond om deze optima te vinden. Het resultaat is een actuator die een horizontale veegbeweging van $0,6\ mm$ uitvoert en maar $1\ mm$ hoog en $0,4\ mm$ breed is.

6.5 Beweging naar rechtsbeneden, 12x12

Deze test heeft als doel na te gaan hoe goed de optimalisatie presteert bij een opvallend grotere topologie zoals 12x12. Voor de verplaatsing werd een uitdagend objectief gesteld dat misschien niet haalbaar zou blijken, gezien het vergelijkbaar is met de diagonaal van de structuur.

dx $1\ mm$

dy $1\ mm$

Ook hier werd een populatie van 100 gebruikt om voldoende generaties te kunnen doorlopen, hoewel deze kleiner dan optimaal is. Na 37 generaties, 18 uur rekenen, een gemiddelde van bijna een half uur per generatie, en het genereren van 27GB aan bestanden, brak de simulatie af door een Abaqus Error. Dit was ongetwijfeld te wijten aan te grote vervormingen met faling tot gevolg. Zoals in figuur 6.12

(d)-(f) blijkt, geeft een druk van $0,5\ MPa$ op de beduidend grotere holtes zeer grote vervormingen die mogelijk onrealistisch zijn. (Na 10 uur was er reeds een herstart nodig omwille van het overschrijden van de gereserveerde 10 uur rekentijd).

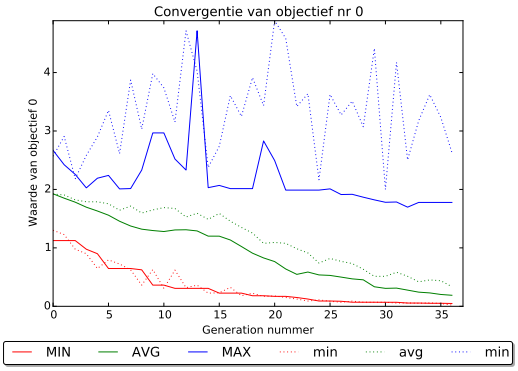
Desondanks werden in 37 generaties reeds goede oplossingen bekomen en was er een sterke convergentie zoals figuren 6.5 (a) en (b) illustreren. De laagste afwijking is $0,021$, terwijl een structuur die niet vervormt in dit geval een afwijking van 2 zou vertonen. Het werken met kleinste kwadraten doet de oplossing beter lijken dan ze is uiteraard, omdat de waarden kleiner zijn dan één. In werkelijkheid duidt deze waarde op een afwijking op beide objectieven van rond de $0,1\ mm$ ofwel 10%. Voor de veeleisende verplaatsing is dit een goed resultaat. Bovendien is de optimalisatie zeker niet ten einde, zoals afgeleid kan worden uit de aanwezigheid van niet-functionele cellen, het ontbreken van een optimum voor bepaalde massawaarden en het nauwelijks dalen van de convergentiesnelheid in figuur 6.5 (b).

In tabel 6.3 zijn alle oplossingen uit het pareto-optimale eerste front van generatie 37 gegeven, ze zijn ook te vinden in figuur 6.5 (c). Merk op dat de oplossingen erg licht zijn, de maximale massa zou 143 kunnen bedragen en een opvulling van 50% materiaal heeft een massa van 72.

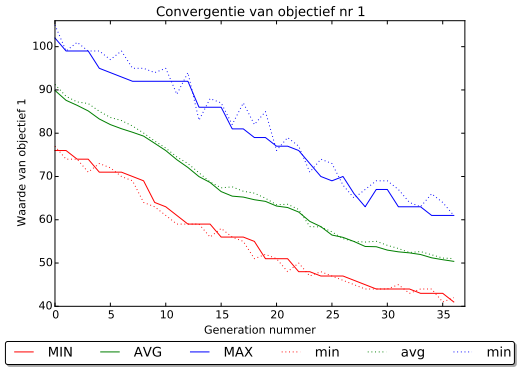
De oplossingen c9882, 7323b en 5a5d2 zijn gekozen ter illustratie van de gevonden topologieën en zijn te zien in figuur 6.5 (d), (e) en (f). Deze eerste twee hebben een erg ingewikkelde vorm van holte, net als c72be, die niet weergegeven is vanwege de overeenkomst met 7323b. De overige lichtere maar minder nauwkeurige oplossingen zijn varianten van 5a5d2, met een eenvoudigere holte.

Naam	Fig.	Afwijking	Massa
c9882	(d)	0,0214	54
c72be		0,0558	51
7323b	(e)	0,0566	50
00d39		0,0944	48
03e96		0,1162	47
77cfb		0,1208	46
5a5d2	(f)	0,1307	44
19641		0,1837	43
651d3		0,5741	42

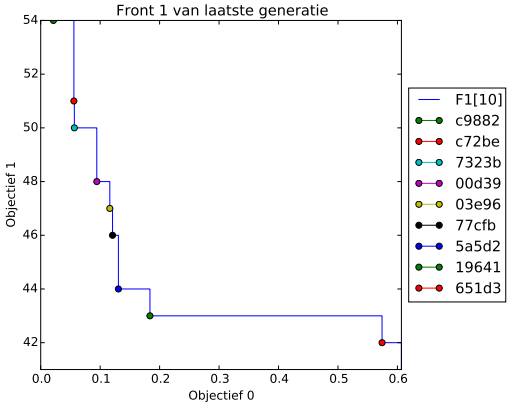
Tabel 6.3: Overzicht van de oplossingen uit generatie 37 met de kleinste afwijking in het 12x12 testprobleem.



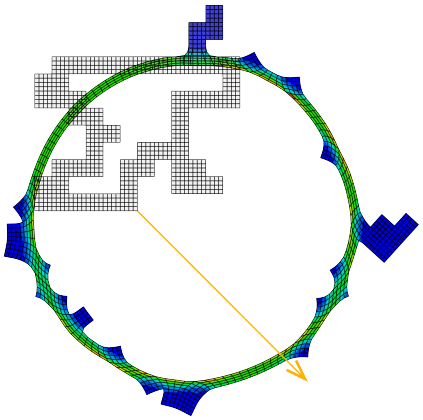
(a) Afwijking



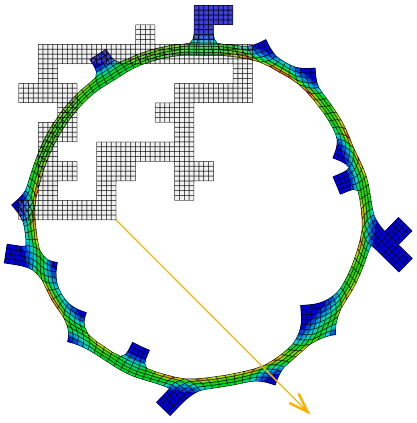
(b) Massa



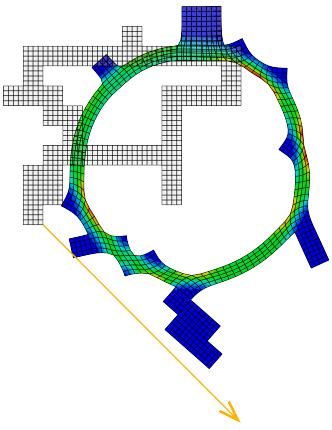
(c) Front 1



(d) c9882



(e) 7323b



(f) 5a5d2

Figuur 6.12: Figuren ter illustratie van het 12x12 testprobleem. (a) en (b) geven de convergentie, (c) de analyse van front 1 uit generatie 37 en (d), (e), en (f) zijn geselecteerde oplossingen.

6.6 Snelheid van de simulaties

In tabel 6.4 zijn enkele gegevens te zien uit bovenstaande testproblemen om een inschatting te geven van de rekentijd. De kolommen *Vec_Th* en *Vec* geven het verschil aan in verwacht aantal vectoren (populatie x generaties) en het werkelijk aantal vectoren dat lager ligt door duplicaten en nodig is om nauwkeurigere rekentijden te verkrijgen. De letter F duidt aan dat de simulatie afgebroken is door de supercomputer omwille van het overschrijden van de gereserveerde tijd of een fout in Abaqus. In die situatie is enkel het werkelijk berekend aantal vectoren relevant. De letter R staat voor een herstarte simulatie.

Deze tijden die uit de output van de supercomputer komen, zijn *walltimes* en geen *processor times*. Gezien de kernen van de VSC toegewijd zijn aan hun toegewezen taak, zorgt dit niet voor onnauwkeurige rekentijden zoals op een PC het geval zou zijn. Wat er wel gebeurt, is dat de tijd dat de processor wacht op invoer/uitvoer wordt opgenomen in het totaal, maar dit is wenselijk om een correcte inschatting te maken van de totale optimalisatietijd.

De derde rij in tabel 6.4 hoort bij het resultaat uit deel 6.3, rij 4 en 5 bij resultaat 6.4 en de voorlaatste rij is de eerste 10 uur van het resultaat uit 6.5. De laatste rij is een afzonderlijke 12x12 simulatie.

De berekeningen op de server nemen per vector slechts enkele seconden in beslag. Dit is beduidend beter dan op een laptop en opvallend snel voor eindige elementen berekeningen. Een grotere topologie geeft een beduidend grotere rekentijd, maar de stijging is niet proportioneel met het aantal cellen of elementen.

Tussen de eerste twee resultaten en het derde, en ook bij de 12x12 resultaten, is een verschil in tijd per vector van 20% te zien. Dit verschil is te groot voor toeval, mogelijk is het een gevolg van een verschil in de gebruikte processor (*Ivy Bridge* tegenover *Haswell*), maar het gebruikte type kon niet uit de uitvoer afgeleid worden.

In het geval van een 4x12 topologie bedraagt het verschil maar 10% en zijn er in de tweede run maar half zoveel berekeningen dan verwacht, omwille van een vergevorderde convergentie met veel duplicaten. Dit betreft dus waarschijnlijk toch hetzelfde processortype en biedt de mogelijkheid om een inschatting te maken hoeveel tijd ingenomen worden door de Abaqus berekeningen. Met a de rekentijd per vector in Abaqus en p de totale uitvoertijd per vector in Python kan het volgende stelsel opgesteld worden:

$$2826(a+b) = 21629 \tag{6.1}$$
$$513a + 1200p = 4365 \tag{6.2}$$

$$a = 7,01 \tag{6.3}$$
$$p = 0,64 \tag{6.4}$$

De oplossing levert een schatting van de verhouding. Abaqus simulaties nemen bijna 92% van de rekentijd in beslag. Dit is de ruime meerderheid zoals altijd aangenomen, maar de 8% voor Python en daarme gerelateerde invoer/uitvoer is toch niet te verwaarlozen.

Top.	Pop.	Gen.	Vec_Th.	Vec.	Tijd [s]	t/Vec. [s]
6x6	100	5	600	568	2937	5,17
6x6	100	5	600	587	2989	5,09
6x6	100	30	3100	3039	19445	6,40
4x12	100	F		2826	21629	7,65
4x12	100	12, R	1200	513	4365	8,51
12x12	100	F		2491	36022	14,46
12x12	100	F		990	11344	11,46

Tabel 6.4: Tabel met voor enkele tests de gegevens (grootte, populatie, generaties, vectoren) en performantie (tijd en tijd per vector).

6.7 Besluiten

Gebruik makend van de servers van het Vlaams Supercomputer Centrum zijn enkele testproblemen opgelost. Het optimalisatieprogramma vertoont goede convergentie en bereikt voor de gestelde problemen bevredigende oplossingen. Per vector is maar enkele seconden rekentijd nodig en voor een kleine topologie kan in slechts 30 (voor 6x6) of 40 (voor 4x12) generaties al een sterke convergentie waargenomen worden. Aanvankelijk wordt vooral geconvergeerd naar een optimale topologie, waarna de optimalisatie meer op lokaal zoeken begint te steunen en niet-functionele cellen verwijdert. De set resultaten bevat steeds een reeks alternatieven die toelaten de optimale keuze te maken, afhankelijk van het relatieve belang van een nauwkeurige beweging of lage massa. Het aandeel van de Abaqus berekeningen kan geschat worden op 92%. De grootte van de geproduceerde data kan al snel tientallen Gigabytes bedragen.

Hoofdstuk 7

Besluit

Bespreking en aanbevelingen

In dit onderzoek werd een programma ontwikkeld in Python voor de topologische optimalisatie van doorsnedes van pneumatische actuatoren. Het genetisch algoritme NSGA-II en een parser om Abaqus input files te genereren werden onder andere geïmplementeerd, samen met routines om geldige vectoren te genereren en deze te controleren.

Een parameteroptimalisatie werd uitgevoerd om dit algoritme aan te passen aan het specifiek probleem. De mutatie- en kruisingskans bleken ruim instelbaar voor optimale performantie, met een voorkeur voor hoge kruisingskans en zolang een zeer hoge mutatiekans vermeden wordt. Optimale parameters uit de literatuur kunnen dus gebruikt worden.

Voor sommige problemen bestaat er een optimale populatiegrootte die de beste performantie biedt op een betrouwbare basis. Grote populaties berusten meer op brute kracht en te kleine populaties bieden een onvoorspelbare performantie. Het zoeken van een dergelijk optimum is vaak rekenintensief en is enkel aangeraden voor specifieke problemen wanneer rekentijd kostbaar is. De performantie is afhankelijk van veel extra factoren, zoals de topologiegrootte, de gebruikte objectieven en de gebruikte hardware.

Verdere verbeteringen worden bekomen door te verhinderen dat *offspring* vectoren identiek zijn aan een ouder om duplicaten te vermijden en door het implementeren van *tournament selection*.

PDMS-10 werd als geschikt materiaal bevonden voor het realiseren van pneumatische actuatoren, maar het modelleren van het materiaalgedrag met een tweede orde Ogden model leverde geen nauwkeurige resultaten. Ook werd het FEM model beschouwd en werden geschikte elementen gekozen.

Een variatie van testproblemen bewees de goede werking van de code en toonde een snelle convergentie. Na een tiental generaties werden steeds vectoren gevonden die voldoende aan het gestelde objectief beantwoordden voor kleine topologieën. Voor grotere topologieën is een grotere populatie dan de gebruikte aangewezen en is betere hardware aanbevolen.

Beperkingen

Het gebruik van deze software is beperkt door een aantal factoren. Het oorspronkelijk driedimensionaal ontwerpprobleem werd gereduceerd tot een tweedimensionale optimalisatie en de keuze van de randvoorwaarden is nog vrij beperkt. De eindpositie van een punt wordt opgegeven maar volledige *path tracing* is niet mogelijk. Omwille van de tijdsduur van de Abaqus simulaties is de grootte van de beschouwde topologieën beperkt als men in de beperkte tijdsperiode van bijvoorbeeld een dag tot een bevredigende oplossing komen. De onnauwkeurigheid van het materiaalmodel tot slot zorgt voor een afwijking tegenover de werkelijke vervorming van het gebruikte materiaal en verhindert voorlopig het realiseren van nauwkeurige actuatoren met behulp van dit ontwerpproces.

Verder onderzoek

Verder onderzoek kan de huidige code verbeteren en aanvullen of uitbreiden. Mogelijkheden tot verbetering zijn bijvoorbeeld het ontwikkelen van betere generatie- en herstelmethoden, volledige *path tracing*, extra soorten kruising en mutatie en het vinden van een betere selectie operator. Parameters zoals de kruisings- en mutatiekans en de populatiegrootte kunnen dynamisch gemaakt worden, zodat ze tijdens de optimalisatie veranderen op basis van de waargenomen convergentie. Het proces kan ook intelligent bijgestuurd worden door bijvoorbeeld nutteloze uitsteeksels in een topologie op te sporen en te verwijderen.

Grote toevoegingen, die cruciaal zijn om dit onderzoek dichter bij een industrieel bruikbaar optimalisatieprogramma te brengen, zijn het toevoegen van een *postprocessing* stap door middel van lokaal zoeken, de uitbreiding naar 3D structuren, en een volledige parallellisatie van de code. Het gebruik van een andere programmeertaal en manier om eindige elementen berekeningen uit te voeren kan een aanzienlijke tijdswinst opleveren. Ook moeten er opties toegevoegd worden om meer vrijheid te geven in de keuze van inklemmingszones en welk punt de gewenste verplaatsing ondergaat, of zelfs toelaten om in meerdere punten een verplaatsing op te leggen.

Uiteraard is in ieder geval ook verder onderzoek naar materiaalmodellen nodig om het werkelijk gedrag van actuatoren nauwkeurig te simuleren.

Een meer uitgebreide behandeling van de mogelijkheden tot verder onderzoek is te vinden in appendix B.

Conclusie

Het gevoerde onderzoek bewijst dat het ontwikkelen van een ontwerpprogramma voor pneumatische actuatoren, gebruik makend van genetisch algoritmen en FEM, zeker mogelijk is. De gerealiseerde code kan een vereenvoudigd ontwerpprobleem, dat nog steeds representatief is voor de moeilijkheden die overwonnen moeten worden, met een goede convergentie oplossen. Knelpunten voor het uitbreiden van dit onderzoek naar een volwaardige automatische optwerptool zijn de lange rekentijd, de beperkte topologiegrootte die een erg ruwe resolutie oplegt aan het ontwerpdomein en de onnauwkeurige simulaties van het werkelijke materiaalgedrag. Het combineren van een betere resolutie met de uitbreiding naar 3D kan deze ontwerpmethode tot een zeer computationeel veeleisend proces maken, maar er zijn geen aanwijzingen dat industriële toepassingen van deze methode ondanks verder onderzoek niet haalbaar zouden zijn.

Besluitend kan worden gesteld dat dit onderzoek een *proof of concept* is voor het evolutionair ontwerp van pneumatische actuatoren voor soft robots.

Bijlagen

Bijlage A

Materiaalmodel en eindige elementen

Dit hoofdstuk bespreekt modellering met eindige elementen en een materiaalmodel om het werkelijke niet-lineaire materiaalgedrag te beschrijven. Dit moet eerder gezien worden als een zijspoor in het onderzoek dat deze masterproef beoogt. De gerealiseerde solver kan werken met een vrije keuze van materiaalmodel en instellingen voor Abaqus.

Om goede eindige elementen berekeningen uit te voeren, moet er gebruik gemaakt worden van een materiaalmodel dat de werkelijkheid goed benadert. Bovendien moet er een juist aantal van een geschikte type elementen gekozen worden. Zowel het materiaalmodel en de FEM discretisatie houden een vereenvoudiging van de werkelijkheid in die de nauwkeurigheid sterk kan beïnvloeden.

Zoals in hoofdstuk 2 vermeld, viel de keuze voor een materiaal, dat geschikt zou zijn voor het vervaardigen van de pneumatische actuatoren, op PDMS-10 en kan dit gemodelleerd worden met een tweede orde Ogden model. Dit komt verder in dit hoofdstuk aan bod, maar het zal nog verder onderzoek vereisen vanwege tegenvallende resultaten.

De focus in dit deel ligt naast het bekomen van een materiaalmodel op de instellingen van de FEM software die algemeen geldig zijn als *best practices* op dit gebied.

A.1 Keuze van geschikte elementen

Zoals in de inleiding verklaard werd, is er een vereenvoudiging gemaakt naar een 2D *plane strain* toestand en moeten er overeenkomstige elementen gekozen worden. Abaqus biedt een grote verscheidenheid aan elementen voor verschillende toepassingen. Voor een 2D *plane strain* toestand zijn dit CPE-elementen. Deze afkorting wordt gevolgd door een nummer dat het aantal knopen per element voorstelt:[28]

3-knoops lineair, driehoek

4-knoops bilineair, vierhoek

6-knoops kwadratisch, driehoek

8-knoops bikwadratisch, vierhoek

Gezien de vorm van de topologie zijn vierhoekige elementen aangewezen. Kwadratisch elementen bieden in sommige gevallen een wat hogere nauwkeurigheid, maar omwille van het grote aantal FEM berekeningen is het aangewezen om de eenvoudigere bilineaire elementen te kiezen.

Verder zijn er speciale varianten van deze elementen die één of meerdere letters dragen na het cijfer:[28]

H Hybride met constante druk. Hybride elementen zijn aangewezen bij onsamendrukbare materialen.[14]

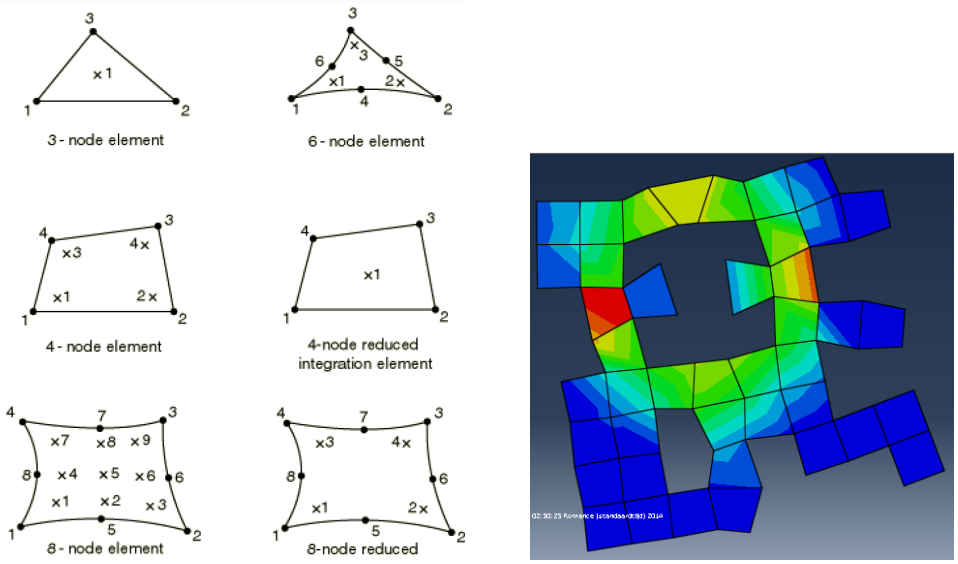
I Incompatibele modes, geeft grotere nauwkeurigheid door het toevoegen van extra vervormingsmodes om het overschatten van de stijfheid tegen te gaan.

R Gereduceerde integratie met *hourglass control*. De integratieregel die toegepast wordt is één orde lager.[14]

M Aangepast, enkel voor 6-knoops elementen.

In figuur A.1 (a) zijn de verschillende elementen te zien die hierboven besproken zijn (H, I en M veranderen niets aan het uitzicht). De kruisjes zijn de integratiepunten.

Om te beginnen is een verklaring van het fenomeen *hourglassing* aangewezen. Wanneer er maar één integratiepunt gebruikt wordt zoals in CPE4R elementen en er maar één element in de doorsnede van een aan buiging onderworpen structuur voorkomt, gedragen de elementen zich niet correct. Sommige elementen ondergaan een vervorming tot een trapeziumvorm, die zich gemakkelijk voortplant door heel het rooster. De naam *hourglassing* komt voort uit de zandlopervorm van paren van dergelijke trapeziumvormige elementen. Figuur A.1 (b) geeft een voorbeeld van een simulatie waar dit probleem optreedt.



(a) Soorten elementen (b) *Hourglassing*

Figuur A.1: (a) Vorm van de verschillende soorten CPE elementen: CPE3, CPE6, CPE4(R) en CPE8(R).[28] en (b) Onnauwkeurige simulatie als gevolg van het *hourglassing*-fenomeen.

R-elementen bevatten een controlemechanisme om dit tegen te gaan, maar het is aangewezen om minstens vier elementen te nemen in een doorsnede.[14]. Gezien de keuze voor vier of vijf elementen in de dikte van een doorsnede ook voor de nauwkeurigheid een *best practice* is die in alle handboeken voorkomt, zullen modellen voorzien worden van minstens vier rijen van de meest eenvoudige elementen, CPE4R, door het rooster van cellen in de topologie vier keer fijner te maken in FEM met behulp van de parameter `grid_refinement=4`.

Uit deze bespreking van de variaties valt af te leiden dat hybride elementen aangewezen zijn omwille van de onsamendrukbaarheid, voor realistische berekeningen mag dit dus niet ontbreken, maar voor de verdere berekeningen in deze masterproef is hier helaas geen gebruik van gemaakt. De berekeningen zouden echter ook met hybride elementen niet waarheidsgetrouw zijn, gezien een te stijf materiaalmodel gecombineerd wordt met een te hoge druk ter compensatie.

Incompatibele modes zij interessant om verder te onderzoeken, gezien zal blijken dat het materiaalmodel te stijf is en de experimenteel waargenomen vervormingen onderschat. Dit kan echter niet gecombineerd worden met het gebruik van gereduceerde integratie en zal de rekentijd ongetwijfeld verhogen gezien minsten vier elementen in de doorsnede aangewezen blijven.

A.2 Invloed van de roosterverfijning

Om het hoofdstuk over het modelleren af te sluiten, wordt nog even stilgestaan bij de invloed van de fijnheid van het FE rooster op de nauwkeurigheid en rekentijd. Hiervoor werden voor verschillende roosters simulaties uitgevoerd met het FE model van de experimentele opstelling uit de vorige sectie op een *Medion The Touch 300 S6615T* laptop met *Intel Core i7-4500U*.

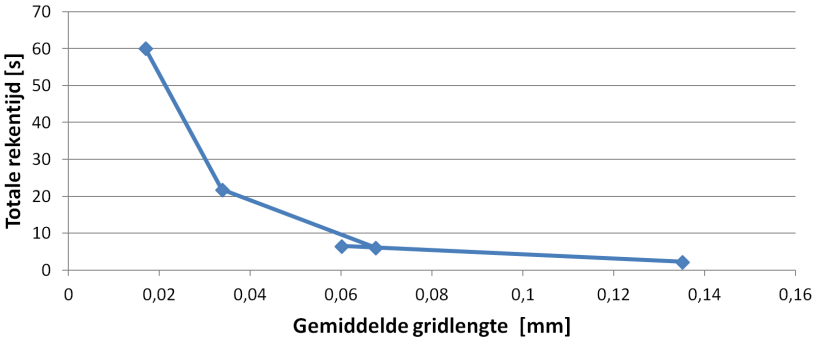
De resoluties van de beschouwde roosters zijn terug te vinden in tabel A.1 samen met het aantal cellen die in de doorsnede van het dunne vlies van de ballon voorkomen. Er zijn twee datapunten voor het geval met twee lagen cellen, omdat die elders in de structuur (aan weerszijden de holte van 100 μm), een verschillend rooster opleveren (60 μm levert twee rijen cellen, $67,5$ μm levert er één).

In figuur A.2 zijn de resultaten te zien van Abaqus berekeningen. De bekomen nauwkeurigheid blijkt lineair toe te nemen met het verlagen van de roosterafstand en de rekentijd neemt exponentieel toe met het fijner maken van het rooster (ze verdrievoudigt ongeveer met elke verfijning). De overgang van één naar twee lagen cellen biedt een aanzienlijke winst in nauwkeurigheid bij een rekentijd die nog steeds onder de tien seconden blijft. Volgens de *best practices* van FEM is een verdere verfijning naar vier cellen in de doorsnede aangewezen. Dit doet de nauwkeurigheid nog verder stijgen en de rekentijd loopt op tot ongeveer 20 s. Van een verdere verfijning naar acht cellen in de doorsnede kan beter afgezien worden. De verandering in de uitwijking bedraagt nog minder dan 1% voor een toename van de rekentijd tot ongeveer 60 s.

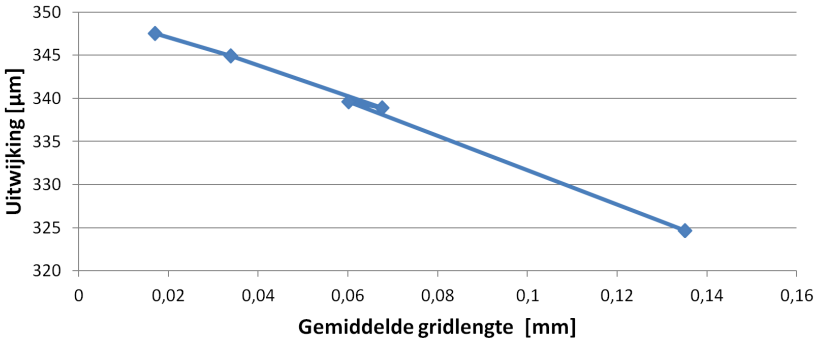
Resolutie	135 μm	60 μm	$67,5$ μm	$33,75$ μm	$16,875$ μm
Cellen in doorsnede	1	2	2	4	8

Tabel A.1: Overzicht van de datapunten in figuur A.2

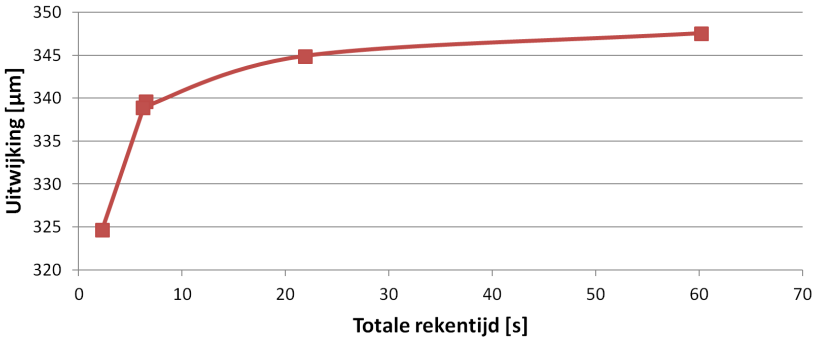
De keuze voor een verfijning naar minimum vier cellen in de doorsnede van het materiaal mag behouden blijven en zal, naar schatting en afhankelijk van de grootte van de topologie, op een laptop processortijden geven van enkele tientallen seconden. Het gebruik van een meer performante computer dan een laptop dient voor lange simulaties onderzocht te worden.



(a) Procestijd in functie van resolutie.



(b) Nauwkeurigheid (uitwijking) in functie van resolutie.



(c) Nauwkeurigheid (uitwijking) in functie van procestijd.

Figuur A.2: Onderlinge invloed van de resolutie van het FE rooster, de rekentijden en de nauwkeurigheid van de simulatie.

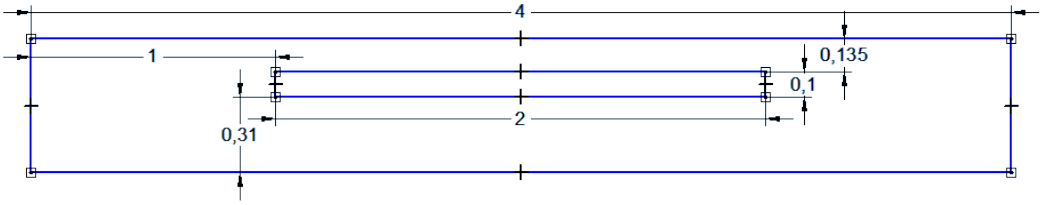
A.3 Materiaalmodel en betrouwbaarheidsanalyse

In dit deel wordt de aanpak besproken om tot een materiaalmodel te komen en om dit model te verifiëren. Om de overeenkomst met de werkelijkheid na te gaan, is eerst en vooral een experiment uitgevoerd. Vervolgens worden modellen uit de literatuur en een eigen parameterfitting hierop getoetst.

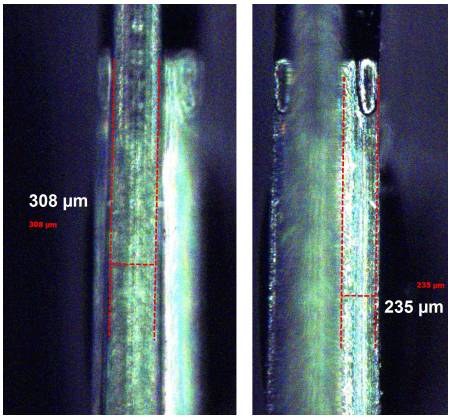
Wat volgt is een beschrijving van de werkwijze van de verschillende test alvorens alle resultaten tegelijk te bespreken en tot een besluit te komen.

Experimentele opstelling

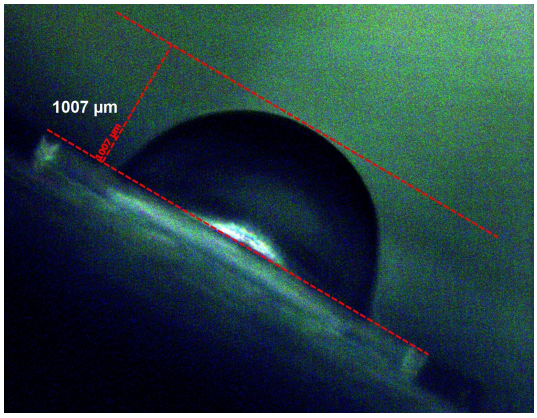
Benjamin Gorissen vervaardigde hiervoor een microactuator en mat de uitwijkingen ervan op in functie van de druk. In figuur A.3 is de doorsnede van deze actuator te zien en een foto die het experiment illustreert. De actuator bevat een langwerpige ruimte waar druk in aangebracht wordt en is over de volledige lengte van de onderkant vastgehecht op een wafer. Voor elk increment van $0,1$ *bar* is de verplaatsing van de ballon opgemeten, deze reeks zal als verificatie dienen voor de nauwkeurigheid van de verschillende sets parameters voor het materiaalmodel.



(a) Technische tekening van de doorsnede van de gerealiseerde actuator (eenheden in *mm*).



(b) Wanddikte van ballon en wafer.



(c) Foto van de vervorming bij een druk van 1 *bar*.

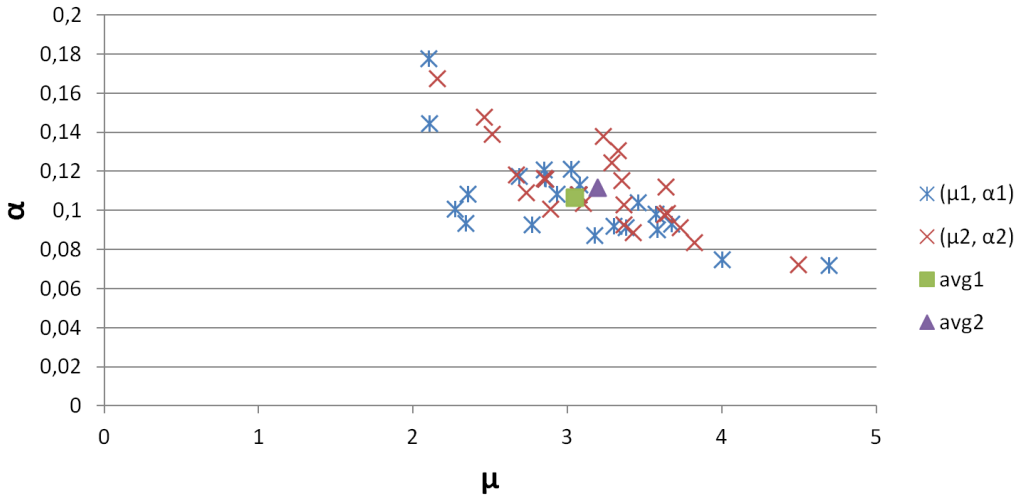
Figuur A.3: De actuator van Benjamin Gorissen: (a) dimensies, (b) meting van de wanddiktes en (c) Foto van het experiment op het ogenblik dat de druk 1 bar bedraagt en de uitwijking 1007 μm

A.3.1 Verificatie van parameters uit de literatuur

De literatuur is een eerste bron voor het vinden van geschikte parameters voor het Ogden(2) model die het gedrag van PDMS-10 modelleren. In tabel A.2 zijn er twee reeksen uit de literatuur te vinden: de waarden beschreven in de paper van Kim et al.[11] en in de masterproef van K. Jordens[9].

A.3.2 Parameterfitting van het materiaalmodel

Alternatief kan een model gefit worden op de data van een trektest. Met dank aan Nele Famaey konden biaxiale trektestdata bekomen worden, evenals MatLab-code om uit deze data een set parameters van het Ogden model te bekomen. In tabel A.2 naast 'Trektest' wordt een gemiddelde van deze parameters voor een hele reeks fittingen uit verschillende reeksen testdata gegeven. De dataset waaruit deze waarden verkregen zijn, is te zien in figuur A.4. Gezien deze trektesten maar data tot een rek van 7% bevatten, wordt het gedrag in het inelastisch gebied onvoldoende in rekening gebracht, wat invloed kan hebben op de nauwkeurigheid.

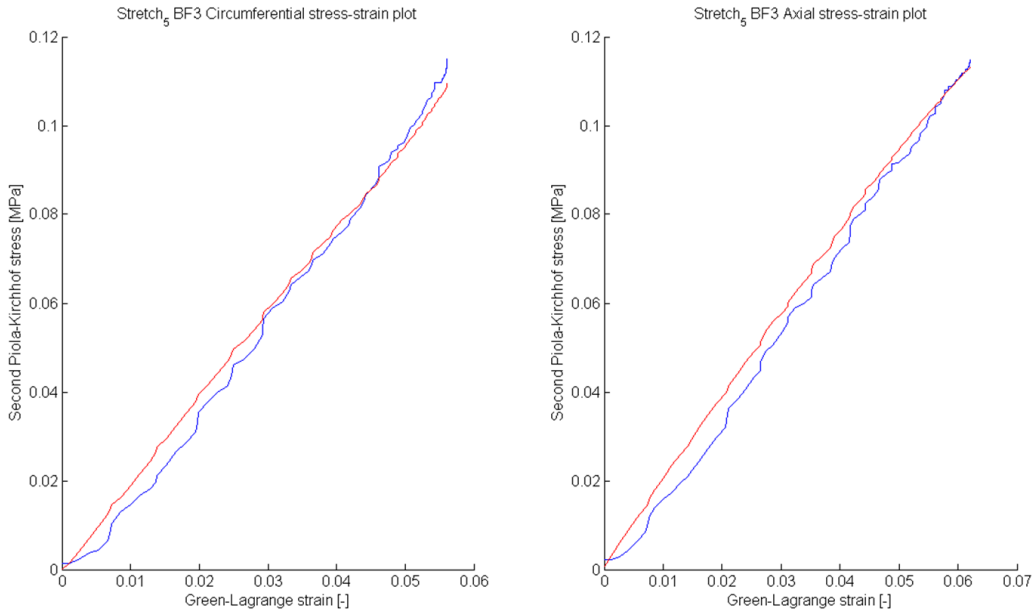


Figuur A.4: Koppels parameters uitgezet in een μ-α diagram om de spreiding na te gaan, gemiddeldes (groen en paars) zijn gebruikt als parameters.

Figuur A.5 geeft een figuur gegenereerd door de MatLab-code van de onderzoeksgroep van Nele Famaey. In dit spannings-rek diagram is de blauwe curve de experimentele data en de rode curve het gefitte model.

A.3.3 Verificatie van gefitte parameters

De verschillende keuzes voor de Ogden-parameters zijn verzameld in tabel A.2. Het valt meteen op dat de parameters van Kim et al.[11] opmerkelijk zijn door de zeer hoge μ_1 en enorm lage α_1 waarde. In dezelfde paper hebben de waarden voor PDMS-5 en PDMS-15 nochtans een normale ordegrootte. De PDMS-10 parameters vertalen



Figuur A.5: Spannings-rek curve uit de MatLab-parameterfitting met een vergelijking tussen de data (blauw) en de bekomen fit (rood).

zich ook in vreemd materiaalgedrag, vanaf een luttele $0,1$ *bar* leverden modellen met deze parameters reeds fouten in Abaqus. Bovendien was zelfs de vorm van de uitbuiging niet helemaal realistisch. Deze parameters zijn daardoor niet bruikbaar.

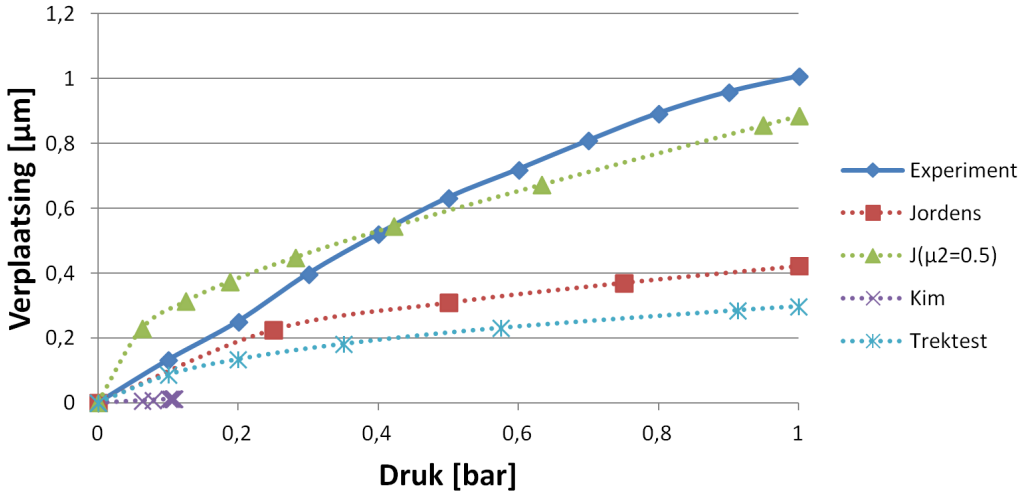
Met de parameters uit de masterproef van K. Jordens[9] werden betere resultaten bekomen, ondanks dat deze parameters duidelijk de eerste voorwaarde uit Berselli et al.[3] schenden, die vermeld werd in deel 2.2.2: het product van μ_2 en α_2 is negatief.

Dat de μ en α waarden uit de trektest sterk op elkaar lijken is eenvoudig te verklaren. Door de lage rek gedroeg het materiaal zich nog lineair, wat een rechte spannings-rek curve oplevert. Deze kan voldoende nauwkeurig benaderd worden met een eerste orde Ogden model. Het tweede orde Ogden model is daardor een samenstelling van twee eerste orde modellen die elk reeds dezelfde curve modelleren.

Bron	μ_1	μ_2	α_1	α_2
Kim et al.	63,4885	0,041103	6,37E-10	3,81166
Jordens	0,192509	2,67748	5,26851	-0,0505378
Trektest	3,04649	3,1924	0,106509	0,111909
J($\mu 2 = 0,5$)	0,192509	0,5	5,26851	-0,0505378

Tabel A.2: Tabel met enkele parameterwaarden voor het tweede orde Ogden model voor PDMS-10.

In figuur A.6 zijn curves te zien die de verplaatsing van het midden van de actuator of 'ballon' aangeven in het experiment en de simulaties. De simulaties zijn in Abaqus uitgevoerd met een heel hoge nauwkeurigheid. Er werd gebruik gemaakt van bikwadratische CPE8H elementen met acht elementen in de doorsnede van het dunne vlies van de ballon.

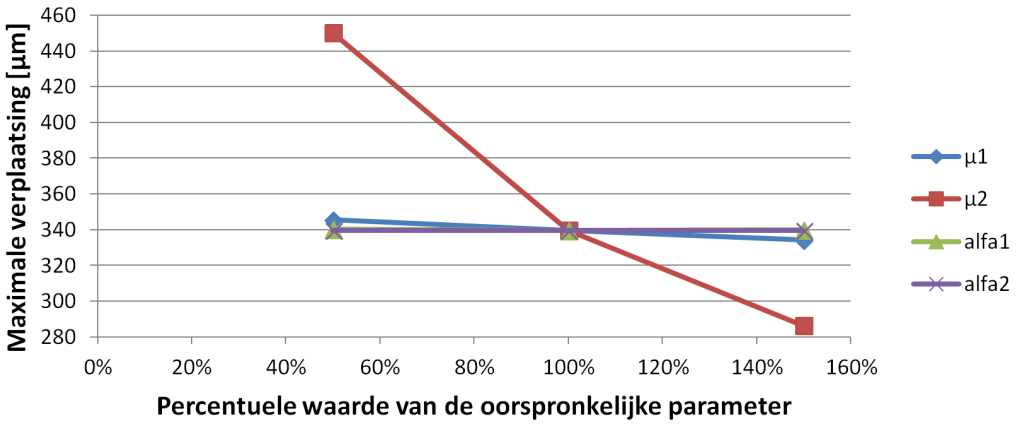


Figuur A.6: Verplaatsings-Druk curves voor van het experiment en enkele simulaties.

De waarden J($\mu 2 = 0,5$) die te vinden zijn in tabel A.2 en figuur A.6 zijn een poging om aan de hand van een eenvoudige sensitiviteitsanalyse het model te verbeteren. Hierbij onderging elke parameter uit het model van Jordens achtereenvolgens een percentuële verhoging of verlaging. De resultaten van simulaties bij $0,5$ *bar* zijn te zien in figuur A.7. De parameter μ_2 heeft duidelijk de grootste invloed op de uitwijking, het verlagen van de waarde verhoogt de verplaatsing. Door middel van *trial and error* werd de parameter verlaagd tot $\mu_2 = 0,5$ en werd de J($\mu 2 = 0,5$) curve bekomen, die te zien is in figuur A.6. Ze benadert de verplaatsing erg nauwkeurig in het punt $p = 0,5$ *bar*.

Er zijn twee belangrijke redenen waarom deze parameters echter niet bruikbaar zijn. Ten eerste is de vorm van de curve anders, er wordt maar in één punt overeenstemming bekomen met deze simplistische aanpak. Men kan niet verwachten dat een materiaalmodel met vier parameters af te stellen is door maar één parameter te veranderen. Ten tweede werd het model afgeleid uit de data die als verificatie voor het gebruikte model hadden moeten dienen.

Geen enkele besproken methode levert een bruikbare set parameters voor het materiaalmodel. Nieuwe trektesten met een veel grotere rek zouden een oplossing kunnen bieden. Het materiaalgedrag van PDMS is afhankelijk van het belastingsgeval, wat voor een groot deel het verschil tussen experiment en model kan verklaren. De simulatie volgens het model uit de trektesten en dat van Jordens verschillen immers



Figuur A.7: Sensitiviteitsanalyse van de materiaalparameters van Jordens.

veel minder onderling dan met het experiment.

Als besluit worden de parameters van Jordens verder gebruikt in deze masterproef, omdat ze de beste optie zijn die voorhanden is. De J($\mu 2 = 0,5$)-parameters zijn om eerder genoemde redenen uitgesloten. In hoofdstuk 6, waar het ontwikkelde programma toegepast wordt op enkele testproblemen, zal een hogere druk opgegeven worden dan in werkelijkheid aangewezen is, om de overschatting van de stijfheid te compenseren.

A.3.4 Besluiten

Omwille van teleurstellende resultaten op het vlak van het modelleren van het gedrag van PDMS-10, zal in deze masterproef verder gebruik gemaakt moeten worden van niet waarheidsgetrouwe parameters. Voor het beoogde doel van dit onderzoek is dat echter geen probleem. De ontwikkelde solver kan met een materiaalmodel naar keuze werken.

De parameters die gekozen worden voor verder gebruik zijn de materiaalparameters uit de masterproef van K. Jordens[9], CPE4R(H) eindige elementen en een roosterverfijning die zorgt dat er voor voldoende nauwkeurigheid minstens vier elementen in een doorsnede voorkomen.

Als men de optimalisaties wil aanwenden voor het ontwerpen van actuatoren die ook werkelijk gerealiseerd zullen worden, zijn verdere inspanningen in het modelleren van het materiaalgedrag noodzakelijk.

Bijlage B

Verder onderzoek

Dit hoofdstuk bevat een niet exhaustieve lijst met suggesties om de verwezenlijkte solver uit deze masterproef te verbeteren of uit te breiden.

B.1 Verbeteringen

B.1.1 Extra DNA generatiemethoden en herstelmethoden

In deel 3.3 zijn vier generatiemethodes en vier hersteltechnieken voor het opbouwen van een initiële populatie voorgesteld. Deze zijn niet allemaal uitgewerkt omdat reeds bevredigende resultaten bekomen werden met willekeurige generatie en herstel van puntcontacten. Het is echter mogelijk dat het toepassen van andere technieken het genereren kan versnellen en er zelfs een geschiktere beginpopulatie bekomen kan worden door vectoren te genereren, rekening houdend met het beoogde doel of algemeen geldende eigenschappen (bijvoorbeeld uitsteeksels dragen niet bij tot een lagere afwijking en verhogen de massa).

B.1.2 Meer vormen van inklemming en outputzone

In dit onderzoek is de inklemming en output beperkt tot een vast punt, namelijk respectievelijk de eerste en laatste 1-bit van het genoom als string. Om meer vrijheid te geven in het modelleren van problemen moet het mogelijk zijn een andere input en outputzone te kiezen. Ook kan het wenselijk zijn de beweging van meerdere eindpunten uit te lezen, bijvoorbeeld om een grijper te ontwerpen. Hiervoor moeten vooral in de parser `DNAtoINP` aanpassingen gemaakt worden die de verschillende vereisten vertalen naar het input bestand.

B.1.3 Path tracing

De huidige code leest in de `RUNandFIT` module het volledige pad uit, maar enkel het eindpunt wordt gebruikt als objectief. Als uitbreiding zou een pad gevolgd kunnen worden als objectief, door een reeks precisiepunten te vergelijken met het pad, evenals

een kleinste kwadraten operator toe te passen op het hele pad zoals in het werk van A.Saxena[18]. Ook hier zal het gebruik van *Fourier shape descriptors* voordelen bieden in vergelijking met kleinste kwadraten. Een belangrijk aandachtspunt is dat het verkregen pad bestaat uit tussenoplossingen. In de huidige Abaqus berekeningen wordt enkel de eindtoestand met een bepaalde druk gesimuleerd. Deze waarden zijn dus mogelijk niet bruikbaar.

B.1.4 Beter materiaalmodel

Het beomen materiaalmodel is niet accuraat en dient vervangen te worden door een betere set parameters voor het tweede Ogden model, of een volledig ander materiaalmodel. Meer experimenten en simulaties zijn nodig om tot een bruikbaar materiaalmodel te komen.

B.1.5 Meerdere soorten mutatie en kruising

De mogelijkheid bestaat steeds om naast de huidige implementatie van kruising en mutatie extra manieren toe te voegen, om te proberen de kansen op een snellere convergentie en betere resultaten te verhogen.

Een idee dat specifiek toegespitst is op het probleem van opblaasbare structuren is het invoeren van een soort 'macro-mutaties', die niet één cel maar een regio van NxN cellen binnen de topologie vervangen door een holte met een rand. Die rand kan eventueel doorbroken zijn waar de oorspronkelijke cellen bij een interne holte horen, om aan te sluiten op deze holte. Dit soort mutaties die met zeer lage kans zouden plaatsvinden, introduceren dus grote nieuwe holtes en veranderen de topologie ingrijpend, maar op een manier die compatibel is met het beoogde resultaat van een opblaasbare structuur.

B.1.6 Andere vormen van selectie

Deb et al. gebruiken voor de selectie van twee ouders binaire toernooiselectie. Dat dit ook voor deze specifieke toepassing de beste keuze is, staat niet vast. In deel 5.3 werd reeds aangetoond dat binaire toernooiselectie beter is dan willekeurige selectie. Dit zijn echter niet de enige mogelijkheden, andere opties zijn het selecteren uit enkel de beste n vectoren of fronts, roulette selectie waarbij de kans van elke vector evenredig is aan zijn *fitness*, het toepassen van meerdere rondes van toernooiselectie of het toepassen van een andere kansverdeling, bijvoorbeeld op rang of positie in de populatie.[23]

B.1.7 Dynamische parameters

Alle instellingen van het GA blijven gelijk tijdens heel de optimalisatie, terwijl in verschillende fases van de optimalisatie andere waarden optimaal kunnen zijn. In het begin van de optimalisatie wordt er bijvoorbeeld sterk globaal gezocht, maar na

een voldoende groot aantal generaties verschuift de focus naar lokaal zoeken om de laatste kleine verbeteringen te bekomen.

Veranderlijke populatiegrootte

Een grote beginpopulatie is wenselijk om een grote genetische diversiteit te verkrijgen. Gezien de impact van de populatiegrootte op de rekentijd en het geheugengebruik, kan het echter wenselijk zijn om de populatiegrootte te doen afnemen.

Een voorbeeld waarin dit zeker geldt, is met 6x6-topologieën en een populatiegrootte van 100. De minimale massa van een geldige structuur is hier 8 en de maximale 35. Er kunnen dus maximaal 28 Pareto-optimale oplossingen gevonden worden, dus de populatie zal bij convergentie bestaan uit één front van minder dan 28 unieke vectoren en minstens 72 duplicaten. Een populatiegrootte van 28 zou dus volstaan na een voldoende aantal generaties.

Dynamische waarden voor cross-over en mutatie

Als alternatief op het vooraf instellen van optimale kansen voor mutatie en kruising, kunnen deze aangepast worden naarmate de optimalisatie vordert. Zoals eerder vermeld, wordt lokaal zoeken belangrijker op het einde. Kruisingen tussen oplossingen die goed scoren op een verschillend objectief zijn waarschijnlijk al vaak uitgevoerd en leveren vaak een oplossing die matig scoort op beide objectieven en verworpen wordt. Men zou de kruisingskans dus kunnen verlagen, of de toernooiselectie kunnen baseren op een bepaald objectief en niet op de rang van een oplossing. Onderzoek moet uitwijzen of dit aantoonbare verbeteringen in de convergentie oplevert.

Verder kunnen de kansen op mutatie en kruising ook aangepast worden uitgaande van informatie van de tussenresultaten zelf. In de gemiddelde leeftijd van een vector, de minimale en maximale waarde van een objectief dat waargenomen wordt in de populatie, en de evolutie van de gemiddeldes van deze objectieven zit informatie vervat over in welke mate lokaal en globaal voortgang geboekt wordt en kunnen ook aanwijzingen te vinden zijn of een bepaalde kans verhoogd of verlaagd mag worden.

B.1.8 Intelligent bijsturen

In hoofdstuk 6 werd duidelijk dat de convergentie van het algoritme bevorderd kan worden door bewuste mutaties, gezien veel optimale tussenoplossingen duidelijk verbeterd kunnen worden door het verwijderen van niet-functionele cellen.

Menselijke interventie in het proces is een optie, waarbij een onderzoeker elke x generaties de beste oplossingen bekijkt en aanpassingen daarvan in de populatie introduceert die naar zijn of haar expertise verbeteringen op bestaande oplossingen zijn. Dit proces kan mogelijk geautomatiseerd worden door de niet-functionele cellen die geen deel uitmaken van inklemming of output zone op te sporen, bijvoorbeeld

aan de hand van een algoritme, patroonherkenning, of uitgaande van de spanningen in het materiaal.

B.2 Toevoegingen

De volgende uitbreidingen zijn interessante opties bij het verder ontwikkelen van de gebruikte code.

B.2.1 Postprocessing door lokaal zoeken

Zoals eerder aangehaald kan op een bepaald ogenblik overgegaan worden op lokaal zoeken om de optimale oplossing nog te verbeteren. Hiervoor kan gebruik gemaakt worden van uitsluitend mutaties op deze oplossing. Sharma et al. passen dit proces standaard toe.[21] Dit laat ook toe om het rooster fijner te maken voor deze lokale optimalisatie, zoals in de Abaqus voorstelling gebeurt dankzij de parameter `grid_refinement`.

Het kan dus interessant zijn om dit standaard toe te passen, maar met de huidige code is dit ook reeds mogelijk. Met dient enkel de populatie in te stellen op één vector die men zelf specificeert en de kruisingskans op nul.

B.2.2 Grafische interface voor CONFIG.py

De manier waarop alle parameters die de instellingen van het algoritme en Abaqus bepalen in `CONFIG.py` zijn verzameld, maakt het eenvoudig om een grafische interface te maken bij de code. Voor het onderzoek zelf heeft dit helaas geen meerwaarde.

B.2.3 Uitbreiding naar 3D

De huidige code optimaliseert de doorsnede van actuatoren met een derde dimensie die veel groter is dan de x en y dimensies. Het zou zeer interessant zijn om de code uit te breiden naar 3D-FEM modellen om het automatisch ontwerpen van willekeurige 3D structuren mogelijk te maken. Het zou in combinatie met 3D printen eender wie in staat stellen tot het ontwikkelen van een geschikte pneumatische actuator voor eender welk probleem.

Om de huidige code uit te breiden naar 3D moeten vooral in de `DNAtoINP` parser veel wijzigingen aangebracht worden om input bestanden te kunnen genereren voor 3D topologieën in Abaqus. In de klasse `DNA` moeten ook alle tests en methodes die op het DNA toegepast worden rekening houden met driedimensionale input. In de overige delen van de code zullen de aanpassingen minder ingrijpend zijn.

Inschatten hoe moeilijk dat is en waar dit veel invloed op de code zou hebben.

B.2.4 Parallellisatie

Gebruik makend van rekeneenheden van het Vlaams Supercomputer Centrum kan een tijdswinst bekomen worden door de code parallel uit te voeren. Hiervoor zijn echter aanpassingen van de code nodig, niet elke code kan zomaar parallel uitgevoerd worden. Standaard programma's zoals Abaqus zijn echter wel al geoptimaliseerd voor het rekenen met veel rekenkernen. Het loont waarschijnlijk niet om het grootste deel van de code te parallelliseren, maar het script uitvoeren op één kern en alle andere beschikbare kernen een afzonderlijke FE berekening laten uitvoeren kan een enorme tijdwinst opleveren. In het optimale geval worden evenveel kernen ingezet als er vectoren in de populatie zijn, plus één voor het uitvoeren van het python script.

B.2.5 Inbouwen van een niet-lineaire FE solver

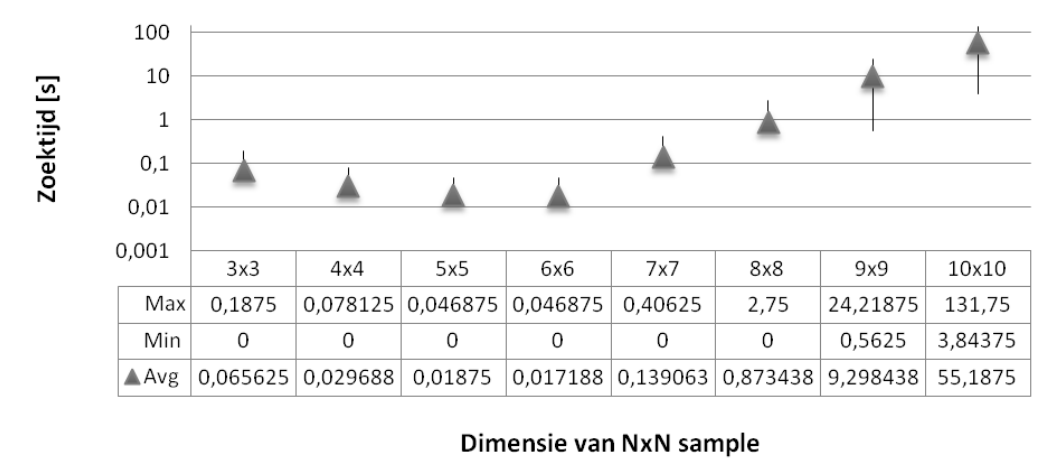
Abaqus is een veelzijdige robuuste solver, maar voor het specifieke probleem van (2D) Niet lineaire FE berekeningen met een Ogden(2) materiaalmodel kan een veel compactere en snellere solver opgesteld worden. Dit kan het proces aanzienlijk versnellen gezien Abaqus berekeningen het grootste deel van de tijd in beslag nemen. Bovendien is er dan geen voorkeur meer voor python als programmeertaal omwille van de compatibiliteit met Abaqus en kan een snellere code geschreven worden in bijvoorbeeld C++.

Bijlage C

DNA generatie: extra grafieken van aantal pogingen en rekentijd

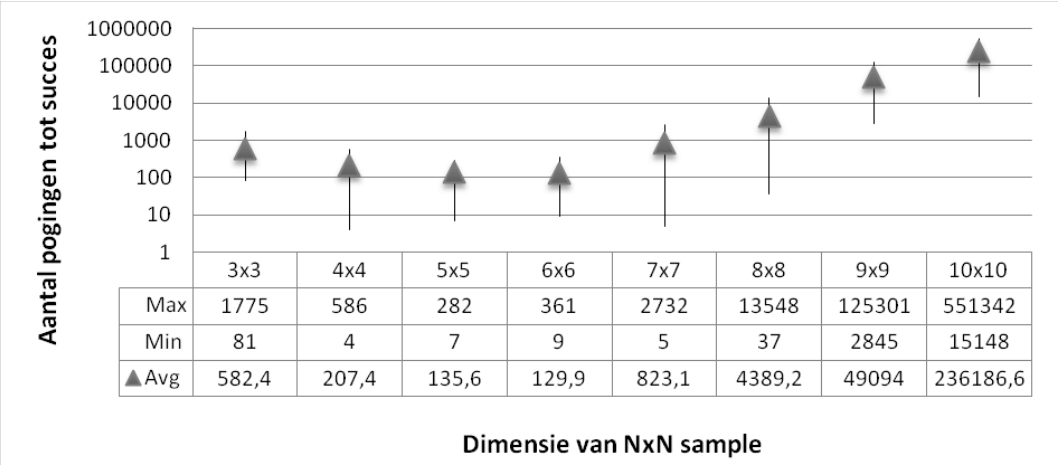
Deze appendix bevat een reeks aanvullende grafieken van resultaten die niet in deel 3.3 opgenomen zijn. Logaritmische grafieken telkens met basis e.

Puur willekeurig



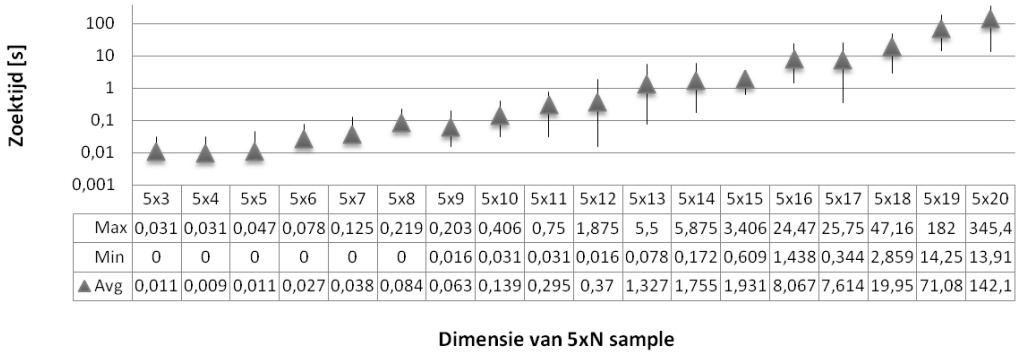
	3x3	4x4	5x5	6x6	7x7	8x8	9x9	10x10
Max	0,1875	0,078125	0,046875	0,046875	0,40625	2,75	24,21875	131,75
Min	0	0	0	0	0	0	0,5625	3,84375
Avg	0,065625	0,029688	0,01875	0,017188	0,139063	0,873438	9,298438	55,1875

Figuur C.1: Zoektijden i.f.v. grootte voor het NxN geval van willekeurige DNA generatie.



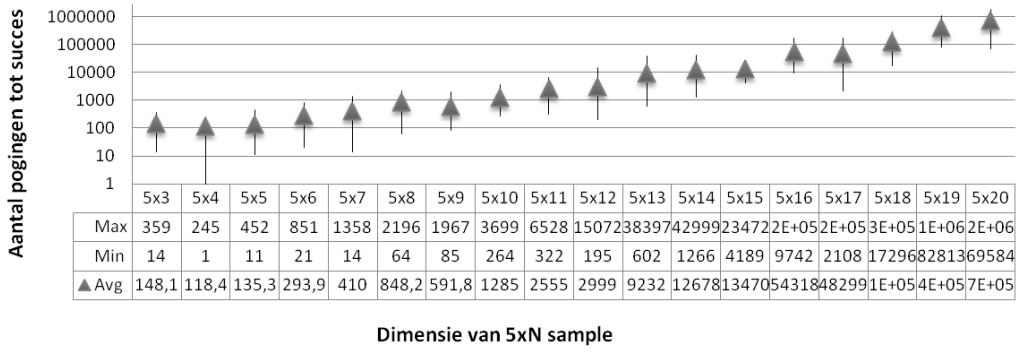
	3x3	4x4	5x5	6x6	7x7	8x8	9x9	10x10
Max	1775	586	282	361	2732	13548	125301	551342
Min	81	4	7	9	5	37	2845	15148
▲Avg	582,4	207,4	135,6	129,9	823,1	4389,2	49094	236186,6

Figuur C.2: Aantal pogingen i.f.v. grootte voor het NxN geval van willekeurige DNA generatie.



	5x3	5x4	5x5	5x6	5x7	5x8	5x9	5x10	5x11	5x12	5x13	5x14	5x15	5x16	5x17	5x18	5x19	5x20
Max	0,031	0,031	0,047	0,078	0,125	0,219	0,203	0,406	0,75	1,875	5,5	5,875	3,406	24,47	25,75	47,16	182	345,4
Min	0	0	0	0	0	0	0,016	0,031	0,031	0,016	0,078	0,172	0,609	1,438	0,344	2,859	14,25	13,91
▲Avg	0,011	0,009	0,011	0,027	0,038	0,084	0,063	0,139	0,295	0,37	1,327	1,755	1,931	8,067	7,614	19,95	71,08	142,1

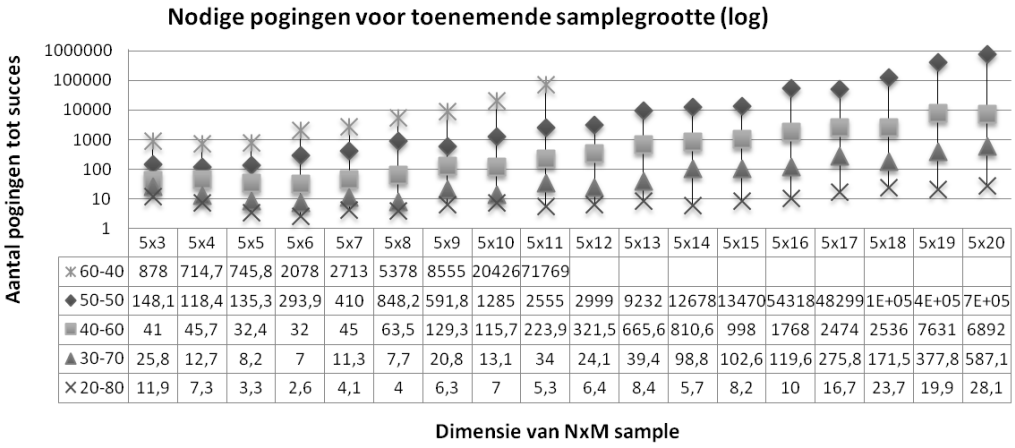
Figuur C.3: Zoektijden i.f.v. grootte voor het 5xN geval van willekeurige DNA generatie.



	5x3	5x4	5x5	5x6	5x7	5x8	5x9	5x10	5x11	5x12	5x13	5x14	5x15	5x16	5x17	5x18	5x19	5x20
Max	359	245	452	851	1358	2196	1967	3699	6528	15072	38397	42999	23472	2E+05	2E+05	3E+05	1E+06	2E+06
Min	14	1	11	21	14	64	85	264	322	195	602	1266	4189	9742	2108	17296	82813	69584
▲Avg	148,1	118,4	135,3	293,9	410	848,2	591,8	1285	2555	2999	9232	12678	13470	54318	48299	1E+05	4E+05	7E+05

Figuur C.4: Aantal pogingen i.f.v. grootte voor het 5xN geval van willekeurige DNA generatie.

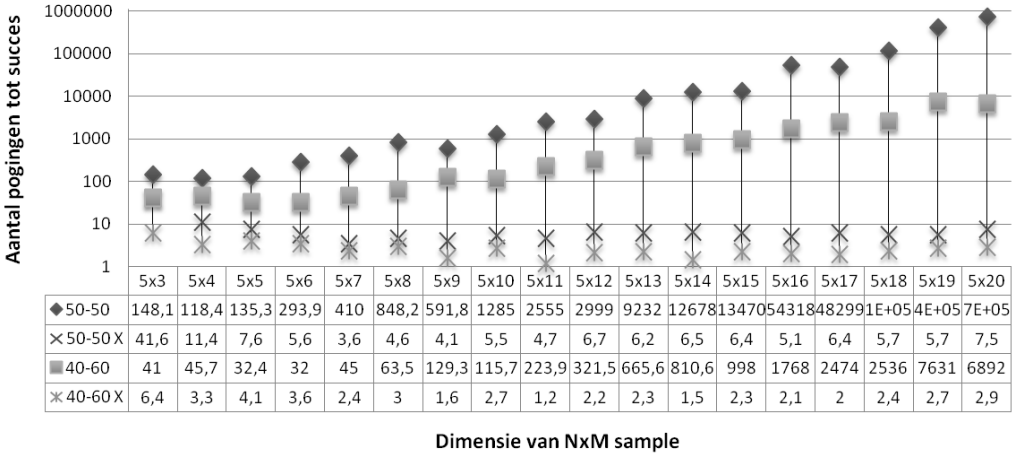
Gewogen willekeurig



	5x3	5x4	5x5	5x6	5x7	5x8	5x9	5x10	5x11	5x12	5x13	5x14	5x15	5x16	5x17	5x18	5x19	5x20
60-40	878	714,7	745,8	2078	2713	5378	8555	20426	71769									
50-50	148,1	118,4	135,3	293,9	410	848,2	591,8	1285	2555	2999	9232	12678	13470	54318	48299	1E+05	4E+05	7E+05
40-60	41	45,7	32,4	32	45	63,5	129,3	115,7	223,9	321,5	665,6	810,6	998	1768	2474	2536	7631	6892
30-70	25,8	12,7	8,2	7	11,3	7,7	20,8	13,1	34	24,1	39,4	98,8	102,6	119,6	275,8	171,5	377,8	587,1
20-80	11,9	7,3	3,3	2,6	4,1	4	6,3	7	5,3	6,4	8,4	5,7	8,2	10	16,7	23,7	19,9	28,1

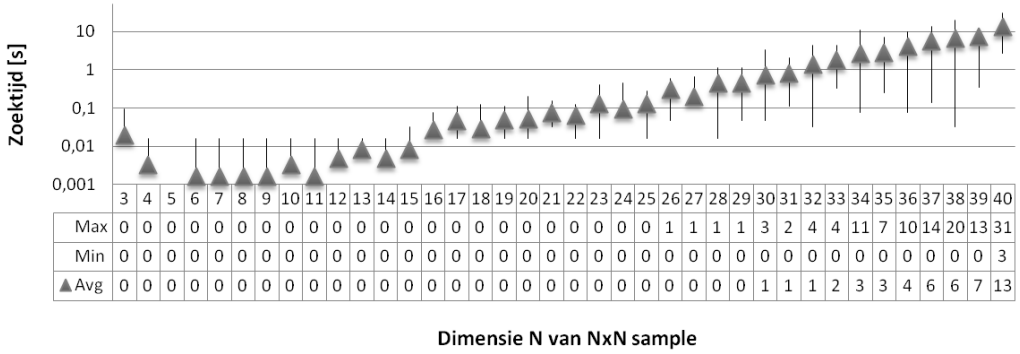
Figuur C.5: Gemiddeld aantal pogingen i.f.v. grootte om een geldig DNA te genereren voor verschillende materiaalverhoudingen.

Herstel van puntcontacten



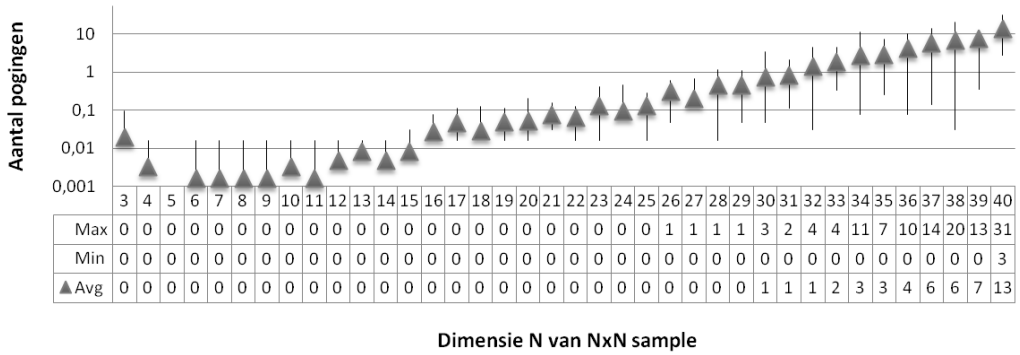
	5x3	5x4	5x5	5x6	5x7	5x8	5x9	5x10	5x11	5x12	5x13	5x14	5x15	5x16	5x17	5x18	5x19	5x20
50-50	148,1	118,4	135,3	293,9	410	848,2	591,8	1285	2555	2999	9232	12678	13470	54318	48299	1E+05	4E+05	7E+05
50-50 X	41,6	11,4	7,6	5,6	3,6	4,6	4,1	5,5	4,7	6,7	6,2	6,5	6,4	5,1	6,4	5,7	5,7	7,5
40-60	41	45,7	32,4	32	45	63,5	129,3	115,7	223,9	321,5	665,6	810,6	998	1768	2474	2536	7631	6892
40-60 X	6,4	3,3	4,1	3,6	2,4	3	1,6	2,7	1,2	2,2	2,3	1,5	2,3	2,1	2	2,4	2,7	2,9

Figuur C.6: Gemiddeld aantal pogingen i.f.v. grootte bij DNA generatie met en zonder herstel van puntcontacten, voor het geval van 50% en 60% open ruimte.



	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40
Max	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	3	2	4	4	11	7	10	14	20	13	31
Min	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	3
Avg	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	1	1	1	2	3	3	4	6	6	7	13

Figuur C.7: Zoektijden i.f.v. grootte voor het NxN geval bij DNA generatie met herstel van puntcontacten.



	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40
Max	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	3	2	4	4	11	7	10	14	20	13	31
Min	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	3
Avg	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	1	1	1	2	3	3	4	6	6	7	13

Figuur C.8: Aantal pogingen i.f.v. grootte voor het NxN geval bij DNA generatie met herstel van puntcontacten.

Bijlage D

Parameter optimalisatie van het GA: volledige resultaten

Dit zijn alle resultaten van de simulaties om uit te zoeken welke mutatie- en crossoverkans het beste is, voor verschillende populaties.

D.1 Testcase 1: Populatiegrootte 20, 5x5

Resultaten weergegeven in figuren D.1, D.2, D.3 en D.4.

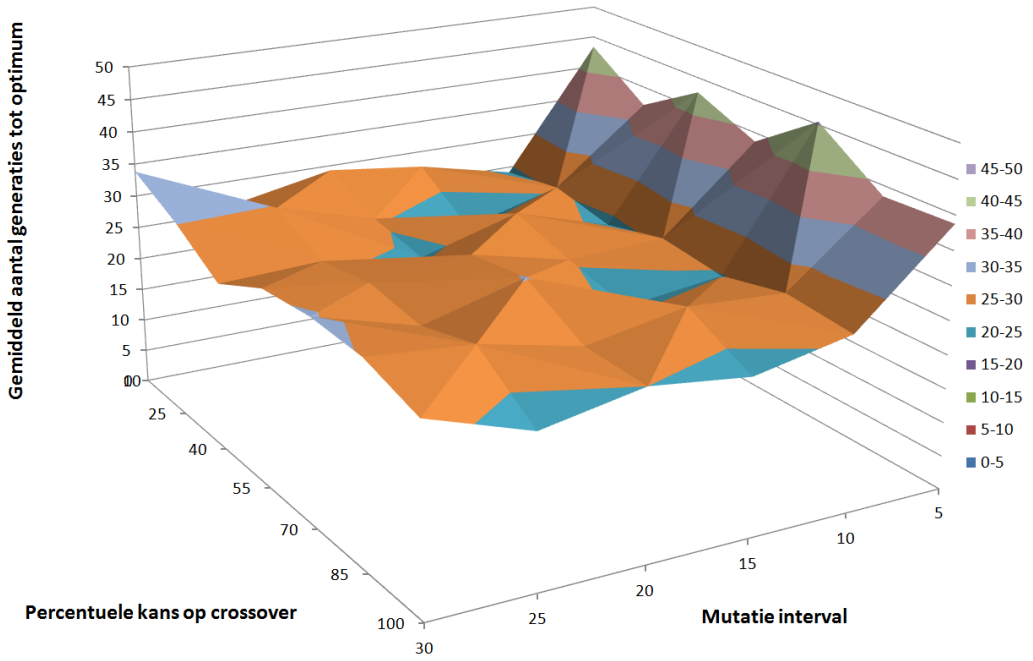
- Gemiddeldes voor: 50 runs
- Populatie: 20
- Topologie: 5x5

De asterisk bij het gemiddelde in figuur D.3 duidt aan dat dit niet zomaar een gemiddelde is, de data voor mutatie interval 30 en 5 is niet in dit gemiddelde opgenomen omwille van het duidelijk verschillend verloop en de grotere variantie.

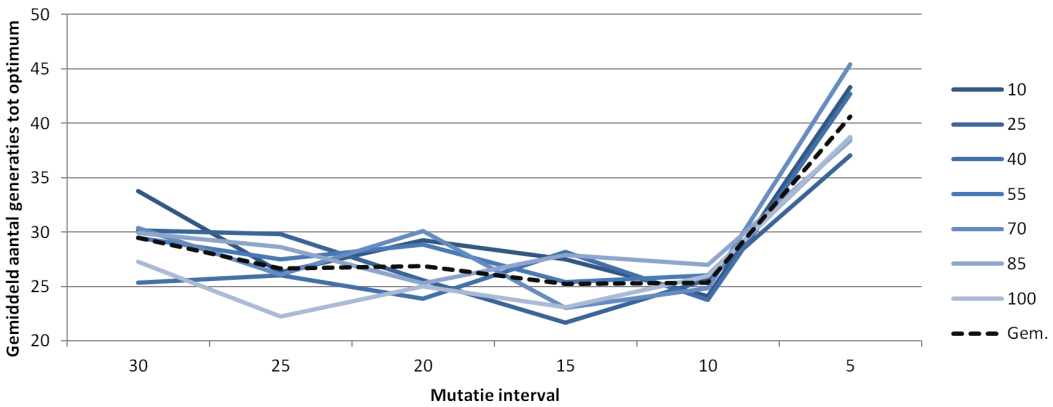
D.2 Testcase 2: Populatiegrootte 40, 5x5

Resultaten weergegeven in figuren D.5, D.6, D.7 en D.8.

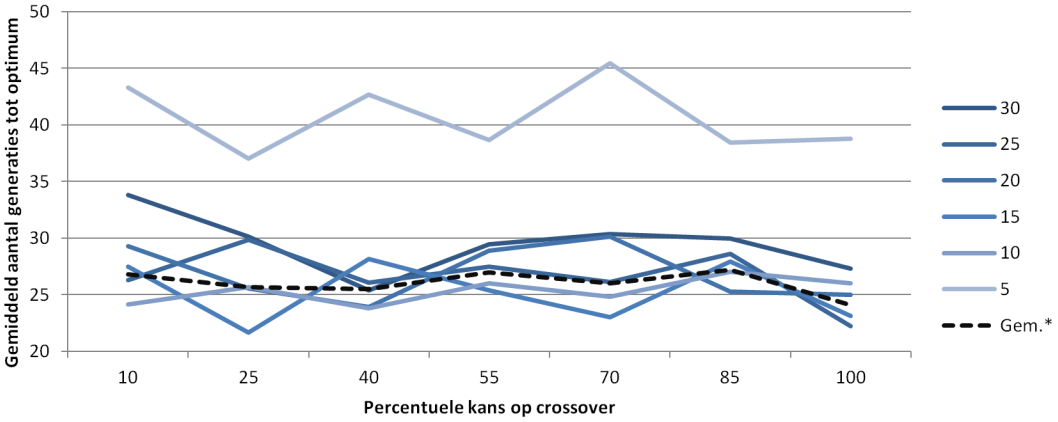
- Gemiddeldes voor: 50 runs
- Populatie: 40
- Topologie: 5x5



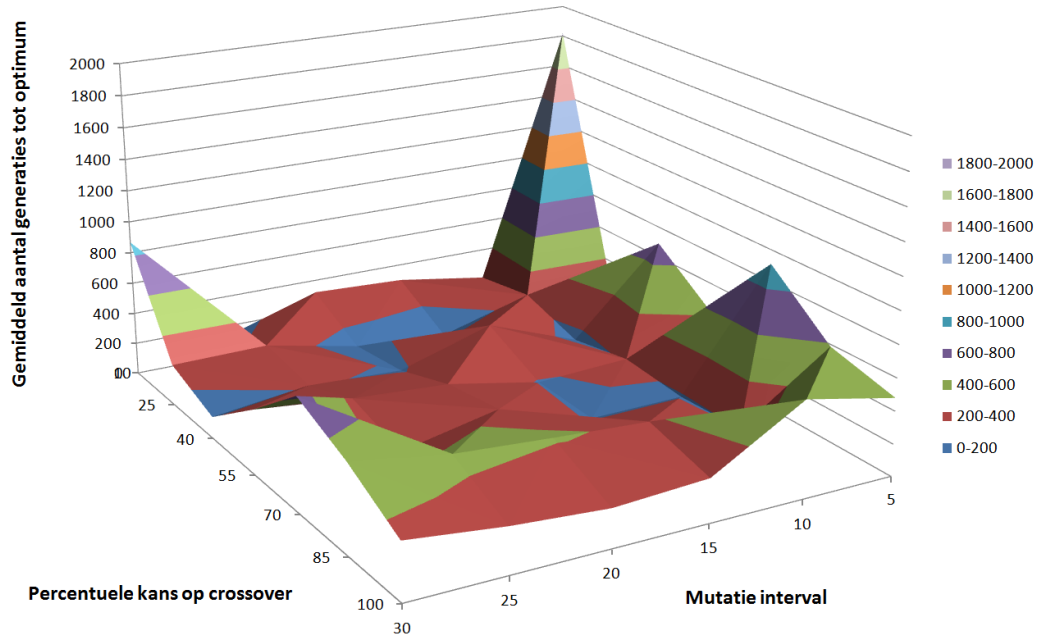
Figuur D.1: Testcase 1: Oppervlak van het gemiddeld aantal generaties voor verschillende combinaties van mutatie- en crossoverkans.



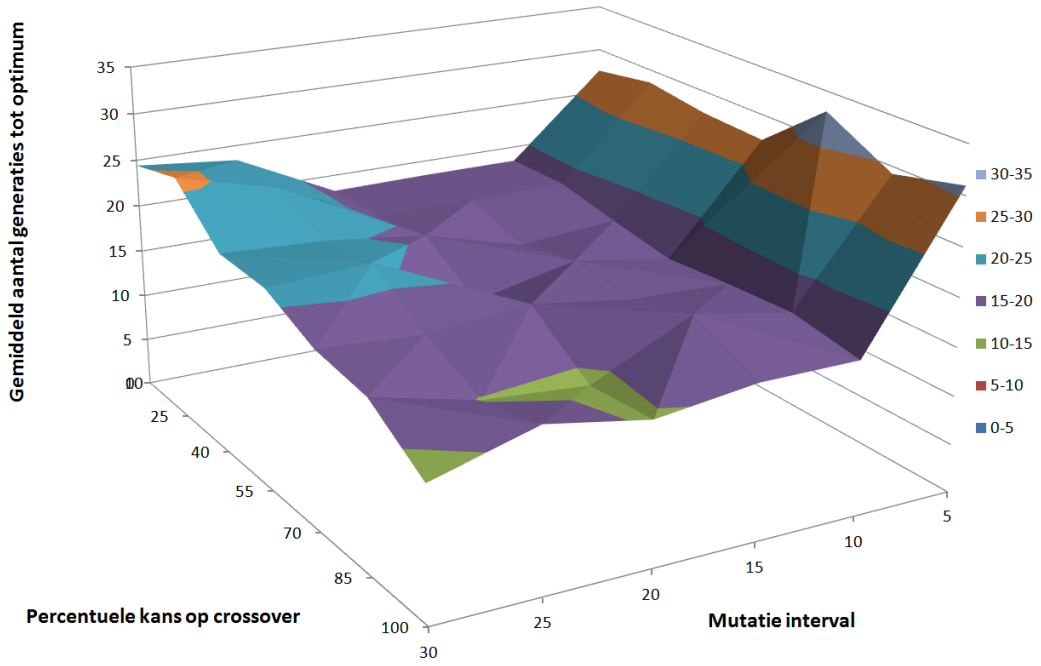
Figuur D.2: Testcase 1: Gemiddeld aantal generaties uitgezet tegen de mutatiekans voor verschillende crossoverkansen.



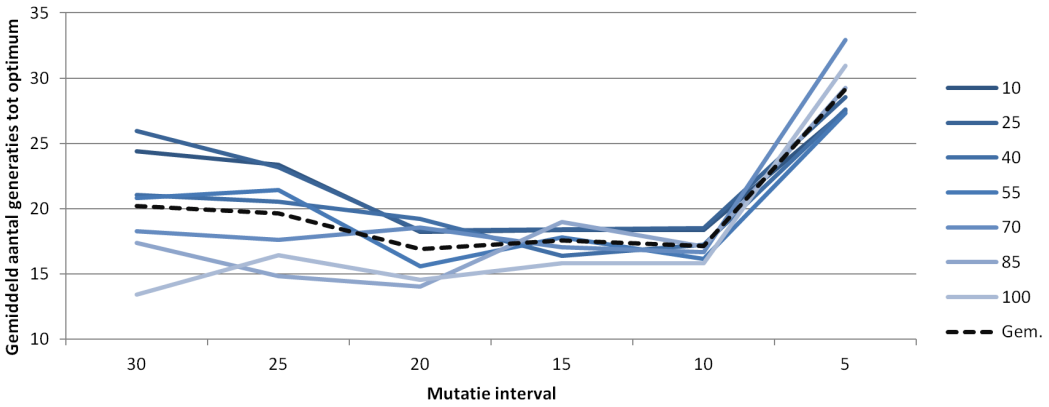
Figuur D.3: Testcase 1: Gemiddeld aantal generaties uitgezet tegen de crossoverkans voor verschillende mutatiekansen.



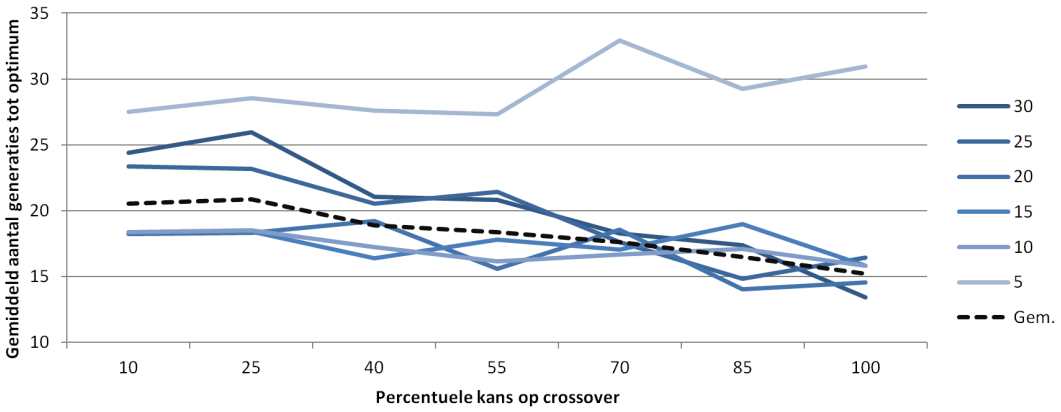
Figuur D.4: Testcase 1: Oppervlak van de variantie voor verschillende combinaties van mutatie- en crossoverkans.



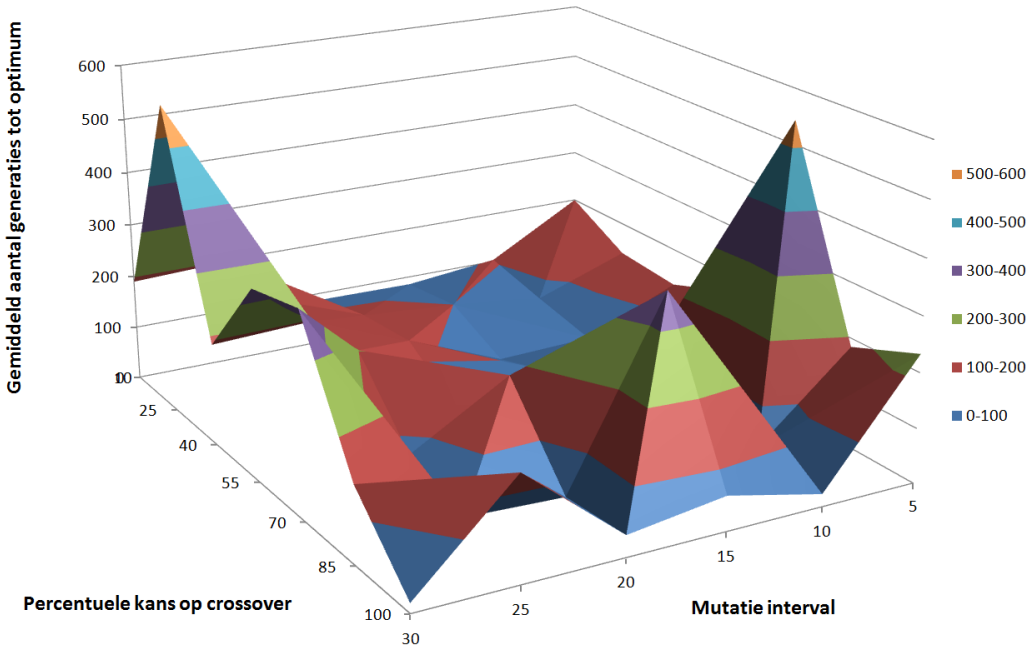
Figuur D.5: Testcase 2: Oppervlak van het gemiddeld aantal generaties voor verschillende combinaties van mutatie- en crossoverkans.



Figuur D.6: Testcase 2: Gemiddeld aantal generaties uitgezet tegen de mutatiekans voor verschillende crossoverkansen.



Figuur D.7: Testcase 2: Gemiddeld aantal generaties uitgezet tegen de crossoverkans voor verschillende mutatiekansen.



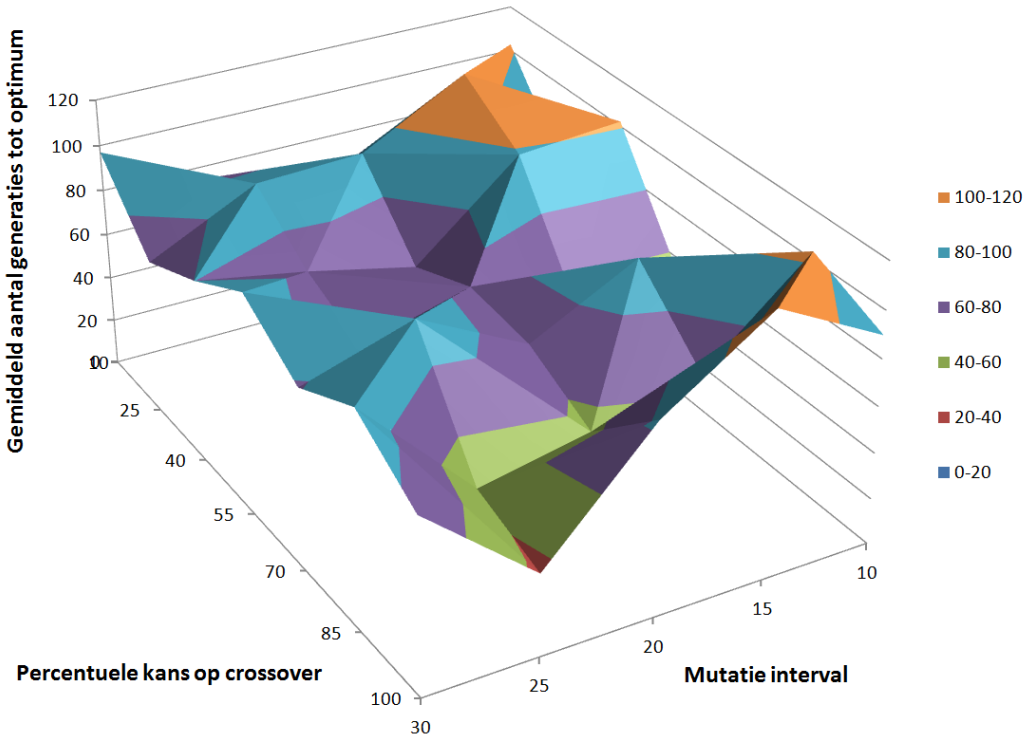
Figuur D.8: Testcase 2: Oppervlak van de variantie voor verschillende combinaties van mutatie- en crossoverkans.

Voor het gemiddelde in figuur D.7 is ditmaal enkel de data voor mutatie interval 5 niet in dit gemiddelde opgenomen omwille van het duidelijk verschillend verloop en de grote variantie.

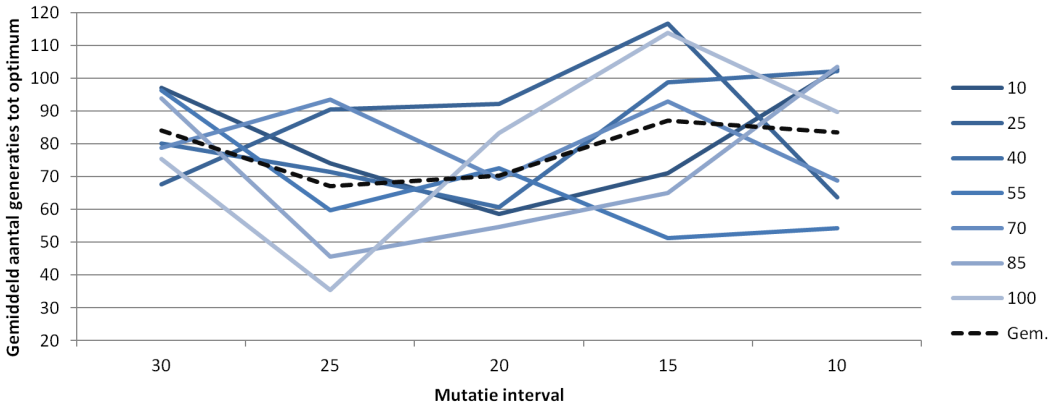
D.3 Testcase 3: Populatiegrootte 100, 6x6

In figuren D.9, D.10 en D.11 is de data voor de gevallen met een mutatie interval van 5 weggelaten, omdat deze nog extremer was dan in de voorgaande gevallen, en de kleinere variaties in de overige punten nauwelijks zichtbaar waren. Het uitvoeren van deze simulatie op een Medion The Touch 300 S6615T laptop met Intel Core i7-4500U nam ongeveer 37 uur in beslag, daarom is voor het gemiddelde van een lager aantal runs gekozen in deze test.

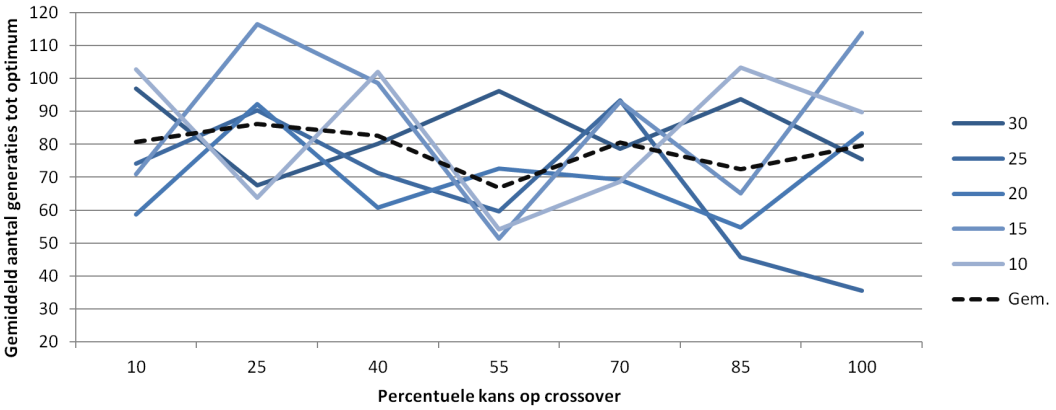
- Gemiddeldes voor: 10 runs
- Populatie: 100
- Topologie: 6x6



Figuur D.9: Testcase 3: Oppervlak van het gemiddeld aantal generaties voor verschillende combinaties van mutatie- en crossoverkans.



Figuur D.10: Testcase 3: Gemiddeld aantal generaties uitgezet tegen de mutatiekans voor verschillende crossoverkansen.



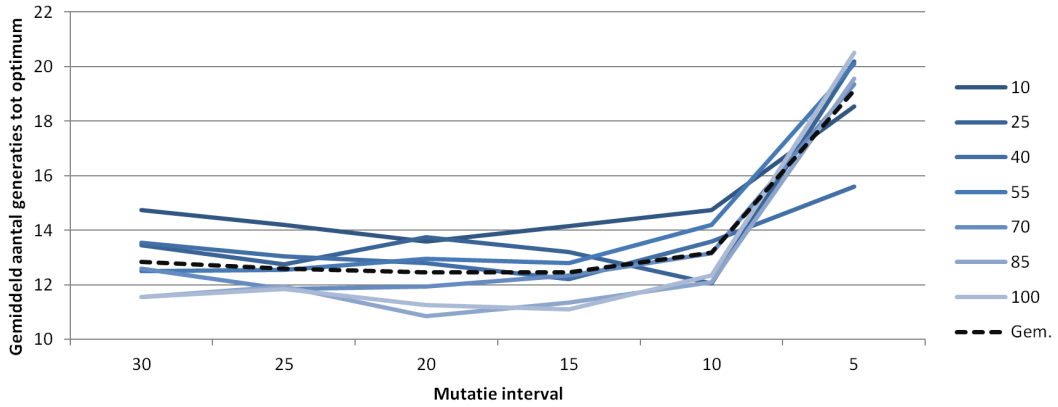
Figuur D.11: Testcase 3: Gemiddeld aantal generaties uitgezet tegen de crossoverkans voor verschillende mutatiekansen.

D.4 Populatiegrootte 40, minimalisatie determinant

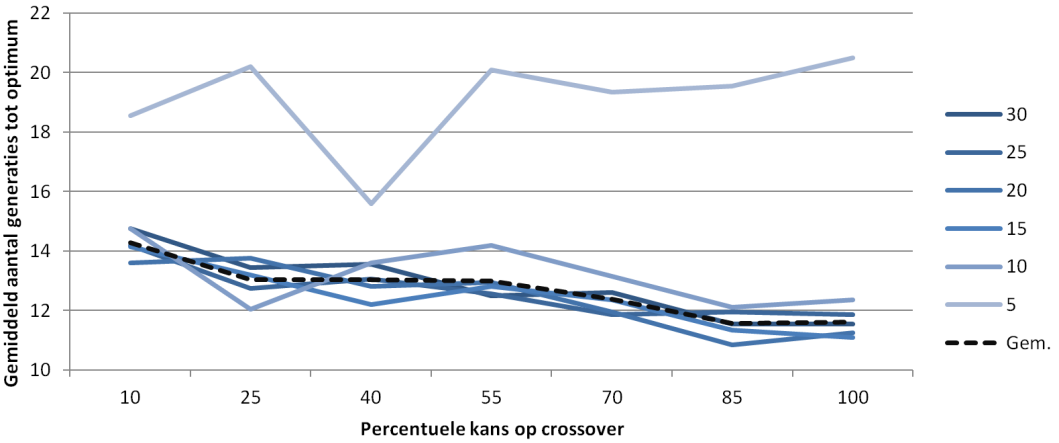
In figuren D.12 en D.13 zijn resultaten te zien met als tweede objectief de minimalisatie in plaats van de maximalisatie van de determinant. De 3D-voorstelling van het aantal generaties en de variantie zijn weggelaten, deze waren zeer vlak behalve voor een mutatie interval gelijk aan 5. Dezelfde resultaten als bij het andere objectief worden bekomen, maar beduidend sneller.

- Gemiddeldes voor: 10 runs
- Populatie: 40
- Topologie: 5x5

Voor het gemiddelde in figuur D.13 is de data voor mutatie interval 5 niet opgenomen omwille van het duidelijk verschillend verloop en de grotere varianties.



Figuur D.12: Gemiddeld aantal generaties bij maximalisatie van de determinant, uitgezet tegen de mutatiekans voor verschillende crossoverkansen.

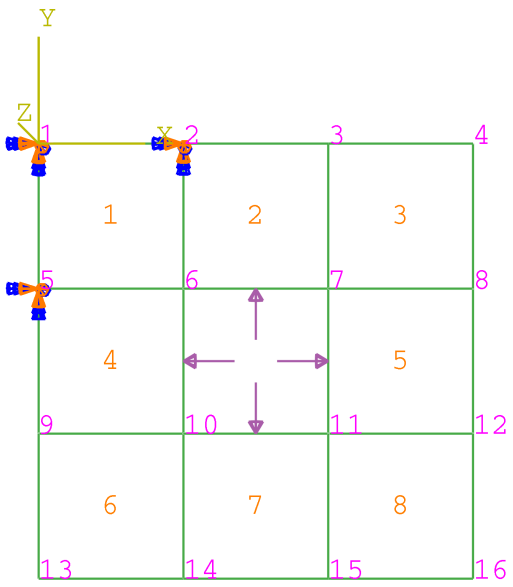


Figuur D.13: Gemiddeld aantal generaties bij maximalisatie van de determinant, uitgezet tegen de crossoverkans voor verschillende mutatiekansen.

Bijlage E

Voorbeeld van een Abaqus input bestand

Ter informatie is hier een input bestand te zien voor de kleinst mogelijke geldige topologie, die als verduidelijking kan dienen bij de uitleg over de parser. De topologie is de enige geldige 3x3-topologie met `grid_refinement=1`. De voorstelling in Abaqus is te zien in figuur E.1, waarop de elementen en knopen met hun nummer aangeduid zijn, evenals de randvoorwaarden. De md5-naam van dit genoom (1, 1, 1, 1, 0, 1, 1, 1, 1) is c7809e7118efa1801bd780cb441b8b98.



Figuur E.1: Structuur van de topologie die beschreven is in input file I-c7809e7118efa1801bd780cb441b8b98.inp.

Input file I-c7809e7118efa1801bd780cb441b8b98.inp

```
*Heading
** Job name: I-c7809e7118efa1801bd780cb441b8b98.inp Model name: Model-1
** Generated by: Abaqus/CAE Student Edition 6.13-2
*Preprint, echo=NO, model=NO, history=NO, contact=NO
**
** PARTS
**
*Part, name=Part-1
*Node
        1,              0.000000,              -0.000000
        2,              0.100000,              -0.000000
        3,              0.200000,              -0.000000
        4,              0.300000,              -0.000000
        5,              0.000000,              -0.100000
        6,              0.100000,              -0.100000
        7,              0.200000,              -0.100000
        8,              0.300000,              -0.100000
        9,              0.000000,              -0.200000
        10,              0.100000,              -0.200000
        11,              0.200000,              -0.200000
        12,              0.300000,              -0.200000
        13,              0.000000,              -0.300000
        14,              0.100000,              -0.300000
        15,              0.200000,              -0.300000
        16,              0.300000,              -0.300000
*Element, type=CPE4R
1,  1,  5,  6,  2
2,  2,  6,  7,  3
3,  3,  7,  8,  4
4,  5,  9,  10,  6
5,  7,  11,  12,  8
6,  9,  13,  14,  10
7,  10,  14,  15,  11
8,  11,  15,  16,  12
*Nset, nset=Set-1, generate
 1,  16,  1
*Elset, elset=Set-1, generate
 1,  8,  1
*Nset, nset=Res-1
 16,
** Section: Section-1
*Solid Section, elset=Set-1, material=PDMS-10
100.,
*End Part
**
**
** ASSEMBLY
**
*Assembly, name=Assembly
**
*Instance, name=Part-1-1, part=Part-1
*End Instance
**
*Nset, nset=Set-1, instance=Part-1-1
 1,  5,  2
*Elset, elset=Set-1, instance=Part-1-1, generate
 1, 1, 1
*Elset, elset=_Surf-1_S1, internal, instance=Part-1-1
5,
*Elset, elset=_Surf-1_S2, internal, instance=Part-1-1
2,
*Elset, elset=_Surf-1_S3, internal, instance=Part-1-1
4,
```

```
*Elset, elset=_Surf-1_S4, internal, instance=Part-1-1
7,
*Surface, type=ELEMENT, name=Surf-1
_Surf-1_S1, S1
_Surf-1_S2, S2
_Surf-1_S3, S3
_Surf-1_S4, S4
*End Assembly
**
** MATERIALS
**
*Material, name=PDMS-10
*Density
 0.000000000920,
*Hyperelastic, n=2, ogden
 0.192509, 5.26851,    2.67748, -0.0505378,    0.0007,    0.0007
** ----------------------------------------------------------------
**
** STEP: Step-1
**
*Step, name=Step-1, nlgeom=YES
*Static
0.1, 1., 1e-05, 0.1
**
** BOUNDARY CONDITIONS
**
** Name: BC-1 Type: Symmetry/Antisymmetry/Encastre
*Boundary
Set-1, ENCASTRE
**
** LOADS
**
** Name: Load-1   Type: Pressure
*Dsload
Surf-1, P, 0.500000
**
**OUTPUT REQUESTS
**
*Restart, write, frequency=0
**
** FIELD OUTPUT: F-Output-1
**
*Output, field, variable=PRESELECT
**
** HISTORY OUTPUT: H-Output-1
**
*Output, history, variable=PRESELECT
*End Step
```

Bijlage F

Structuur van de gerealiseerde code

Dit hoofdstuk geeft een overzicht van alle klasses, variabelen en functies van de gerealiseerde solver, zodat de globale structuur duidelijk en reproduceerbaarder wordt. De volledige code is te vinden in appendix G, het doel van dit hoofdstuk is louter het bespreken van de functie van de verschillende klasses, methodes en belangrijkste variabelen.

F.1 Configuratiefile: CONFIG.py

De CONFIG.py file bevat alle parameters die aangepast kunnen worden. Alle opties die er bestaan zijn hierin verzameld. Dankzij deze ontwerpbeslissing kunnen alle inputs voor het genetisch algoritme en voor Abaqus centraal aangepast worden, zonder in de code te gaan zoeken. De code kan op deze manier veelzijdig gebruikt worden door anderen die minder vertrouwd zijn met de eigenlijke structuur en werking. Zelfs het toevoegen van een extra objectieffunctie is vrij eenvoudig, er moet maar op enkele plaatsen iets aangepast worden want de code werkt per definitie met een lijst van een onbepaald aantal objectieven. Deze file start ook de hoofdklasse, die instaat voor het uitvoeren van het volledige algoritme.

Variabelen

- I/O variabelen:

  abaqus_run (Boolean) Geeft aan of Abaqus gebruikt zal worden, op basis hiervan worden extra modules ingeladen.

  restart (Boolean) Laat de optimalisatie starten met een oude generatie als populatie, die opgeslagen was in een pickle file.

save_cvs (Boolean) Geeft aan of voor elke generatie een aantal gegevens van de vectoren in een CSV bestand opgeslagen moeten worden. Voor elke populatie gebeurt dit in generation.csv, voor elke offspring in offspring.csv.

save_pickle (Boolean) Bepaalt of de datastructuur RES die alle data over generaties, offspring en fronts bijhoudt, moet opgeslagen worden als pickle bestand

save_graphs (Boolean) Bepaalt of PLOTTER een grafiek moet genereren en opslaan van de convergentie van de verschillende objectieven.

save_fronts (Boolean) Bepaalt of PLOTTER voor elke generatie een figuur moet maken van de verschillende niet-gedomineerde fronts van NSGA-II voor twee objectieven.

- Systeemvariabelen:

  path (str) Bestandslocatie van de pythonfiles van de solver, de files beschreven in dit hoofdstuk dus.

  result_path (str) Bestandslocatie waar alle resultaten (CSV, pickle, PDF's van figuren) opgeslagen moeten worden.

  calc_path (str) Bestandslocatie voor tijdelijke files zoals .inp en .odb bestanden, de locatie waar Abaqus output zal plaatsen.

  pickle_file (str) Naam voor de pickle file waarin Genepool.RES opgeslagen wordt.

  pickle_egg (str) Naam voor de pickle file om het generation-object van de laatste genepool op te slaan.

- DNA parameters:

  x (int) X-afmeting van de topologie (breedte van de doorsnede).

  y (int) Y-afmeting van de topologie (hoogte van de doorsnede).

- Parameters van het algoritme:

  population_size (int) Aantal vectoren in een generatie.

  generations (int) Aantal generaties dat het algoritme moet doorlopen.

  crossover_chance (int) Percentuele kans dat Vectoren in de offspring populatie uit een kruising zullen ontstaan.

  mutation_interval (int) Bepaalt om de hoeveel bits er gemiddeld een bit gemuteerd zal zijn in vectoren van de offspring populatie.

- Actieve objectieven:

  - Bevat voor elk objectief een boolean die aanduidt of het actief is.
  - dx en dy zijn getallen die aangeven wat de gewenste verplaatsingen zijn van het eindpunt in het verplaatsings objectief.

- Abaqus parser parameters:

  grid_refinement (int) In hoeveel elementen de hoogte en breedte van een cel in de topologie (DNA of binarie voorstelling) ingedeeld moet worden voor FEM berekeningen.

  cell_edge (float) Afmeting van een cel in de DNA voorstelling in millimeter.

  element_type (str) Type eindige elementen die gebruikt dienen te worden.

  pressure (float) Aan te leggen druk in MPa.

  density (float) Densiteit van het materiaal in ton per kubieke millimeter.

  model_info (str) Type materiaalmodel om te gebruiken.

  model_param (str) Reeks bepalende parameters voor het materiaalmodel.

Methodes

Deze file bevat éń methode Main(), die de hoofdfile of een testscript aanroept door ze te importeren als CONFIG.py uitgevoerd wordt. Door een if-test gebeurt dit enkel wanneer CONFIG.py zelf gestart wordt en niet gewoon geïmporteerd wordt. Dit is belangrijk, aangezien alle klasses CONFIG.py moeten importeren om de parameters te kunnen lezen, en python standaard alle code uitvoert bij het importeren.

F.2 Hoofdfile

Dit script zonder aparte methoden of variabelen staat enkel in voor het aanmaken van een genepool klasse en diens methodes een gewenst aantal maal aanroepen. Stopcondities kunnen in deze klasse toegevoegd worden. Ook het aanroepen van de PLOTTER module om figuren te genereren indien gewenst gebeurt in dit script.

F.3 Klasse Genepool

De langste en meest complexe klasse van de solver. Alle onderdelen van het genetisch algoritme NSGA-II zijn hierin onder gebracht, evenals alle methodes met betrekking tot genetische operatoren en het opslaan van data.

Methodes en variabelen

___init___(self) De constructor maakt een object van de klasse aan en initialiseert dus ook de belangrijke variabelen:

pool (Generation) Het generatie object dat de huidige generatie voorstelt.

age (int) De leeftijd van de genepool, ofwel het aantal generaties dat momenteel doorlopen is.

RES Deze variabele is een datastructuur die alle belangrijke gegevens bijhoudt doorheen heel de uitvoering van het algoritme. Voor zowel elke pool als offpring houdt hij voor elk objectief de beste, slechtste en gemiddelde score bij, de md5 naam van de beste vector en een lijst met vectoren. Ook worden alle fronts en de vectoren die ze bevatten opgeslagen voor elke generatie. Dit gebeurt in lijsten en dictionaries. Een voorstelling van de structuur wordt gegeven in figuur F.1. Winners, bests, averages en worsts zijn eerst ingedeeld volgens objectief, dan volgens generatie. Members en fronts zijn ingedeeld volgens generatie, fronts is verder nog opgedeeld volgens niet-dominerend front. Vierkante haken symboliseren een lijst, accolades een dictionary.

$$\left\{\begin{array}{c}\textit{'lineage'}: \left\{\begin{array}{c}\textit{'winners'}: \left[\left[\begin{bmatrix}'md5',\\ 'md5',\\ \vdots\end{bmatrix}, \begin{bmatrix}'md5',\\ 'md5',\\ \vdots\end{bmatrix}, \begin{bmatrix}'md5',\\ 'md5',\\ \vdots\end{bmatrix},\cdots\right]\right.\\ \textit{'bests'}: [[\#,\ \#,\ \#,\cdots],\ [\#,\ \#,\ \#,\cdots],\ [\#,\ \#,\ \#,\cdots],\cdots]\\ \textit{'averages'}: [[\#,\ \#,\ \#,\cdots],\ [\#,\ \#,\ \#,\cdots],\ [\#,\ \#,\ \#,\cdots],\cdots]\\ \textit{'worsts'}: [[\#,\ \#,\ \#,\cdots],\ [\#,\ \#,\ \#,\cdots],\ [\#,\ \#,\ \#,\cdots],\cdots]\\ \textit{'members'}: \left[\left[\begin{bmatrix}VECTOR,\\ VECTOR,\\ \vdots\end{bmatrix}, \begin{bmatrix}VECTOR,\\ VECTOR,\\ \vdots\end{bmatrix}, \begin{bmatrix}VECTOR,\\ VECTOR,\\ \vdots\end{bmatrix},\cdots\right]\right.\end{array}\right\}\\ \textit{'offspring'}: \{Heeft\ dezelfde\ structuur\ als\ lineage\}\\ \textit{'fronts'}: \left[\left[\begin{bmatrix}[\,]\\ [VECTOR]\\ [VECTOR,\ VECTOR]\\ [VECTOR,\ \cdots]\\ \vdots\end{bmatrix}, \begin{bmatrix}[\,]\\ [VECTOR,\ \cdots]\\ [VECTOR,\ VECTOR,\ VECTOR]\\ [VECTOR]\\ \vdots\end{bmatrix}, \cdots\right]\right.\end{array}\right\}$$

Figuur F.1: Voorstelling van hoe de data in de structuur RES opgeslagen wordt in een Genepool object.

genesis() Maakt de genepool klaar voor evolutie door een populatie willekeurige vectoren aan te maken.

evolve() De belangrijkste methode, die instaat voor één stap in de optimalisatie. Ze maakt een offspring populatie, evalueert de vectoren (deze functie staat in Generation), sorteert ze, slaat data op en voegt de pool en de offspring samen tot een nieuwe populatie. Ze heeft een hele reeks hulpmethoden:

tick() Verhoogt de leeftijd van de pool en alle vectoren daarin met één.

rank(generation, type) Berekent statistieken zoals het gemiddelde voor de objectieven en voegt informatie toe in RES (bests, averages, worsts en winners). Type is dus 'lineage' of 'offspring'.

reproduce() Maakt een offspring-generatie door kruising en mutatie, met behulp van volgende methoden:

crossover(one, two) Kruist de opgegeven vectoren, rekening houdend met de opgegeven kans in CONFIG.py.

mutate(genome) Brengt mutaties aan in een kopie van het aangereikte genoom met de in CONFIG.py opgegeven kans.

fast_nondominated_sort(P) Dit is een deel van het NSGA-II algoritme, dat de vectoren organiseert in fronts afhankelijk van hun onderlinge dominantie.

tournament() Selecteert twee willekeurige vectoren uit de genepool en retourneert de beste (dominant of hoogste crowding distance).

crowding_distance_assignment(L) Eveneens een deel van NSGA-II, berekent voor de vectoren in een front de oppervlakte die ze voorstellen om hun belang in te schatten als niet het volledige front aan de volgende pool zal toegevoegd worden.

terminate() Na een gewenst aantal generaties zal terminate aangeroepen worden, om de laatste generatie te ranken en data in RES te schrijven, en indien gewenst ook naar een CSV-bestand en pickle bestand.

restart() Alternatief voor genesis() om de instantie klaar te maken voor optimalisatie, door oude resultaten en een oude generatie in te laden als self.pool en deze optimalisatie verder te zetten.

csvwrite(name, generation) Slaat data op over alle vectoren in een generatie-object op één lijn van het met 'naam' gespecifieerde CSV-bestand.

pickle(self) Slaat indien gewenst self.RES en self.pool op in pickle bestanden, voor het genereren van figuren achteraf en om restart mogelijk te maken.

F.4 Klasse Generation

Voorstelling van een generatie, een verzameling vectoren die een populatie vormen. Ze bevat maar twee methoden:

___init___(members) De constructor om een instantie aan te maken, deze vereist een lijst met vectoren als argument.

evaluate(best) De hulpmethode van Genepool.evolve() die ervoor zorgt dat voor alle vectoren in deze generatie wordt berekend hoe de vector scoort op alle actieve objectieven. Deze methode geeft dus ook opdracht om input files aan te maken en FEM berekeningen te starten indien `CONFIG.deviation_objective = True`.

F.5 Klasse Vector

De vector klasse is de voorstelling van een levend specimen met een bepaald DNA, ze bevat een groot aantal variabelen:

- Een DNA en een leeftijd
- Variabelen om resultaten voor de verschillende objectieven op te slaan.
- Variabelen die nodig zijn in de methoden `fast_nondominated_sort(P)` en `crowding_distance_assignment(L)` van NSGA-II

Daarnaast ook nog enkele methoden:

get_md5name() Retourneert `DNA.get_md5name()`.

get_objectives() Retourneert een lijst met de resultaten van alle objectieven.

generate_file() Maakt een Abaqus input file van het DNA met behulp van DNAtoINP.

simulate_result() Roept RUNandFIT aan om de geassocieerde input file door Abaqus te laten uitrekenen en het resultaat te fitten op het gewenste eindpunt.

dominates(q) Gaat na of deze Vector vector q domineert. (Dit is zo als q op geen enkel objectief beter scoort.) Wordt gebruik in `fast_nondominated_sort(P)`.

De constructor verhindert het aanmaken van een vector met een niet geldig DNA doormiddel van een assert-statement.

F.6 Klasse DNA

De DNA klasse is de laagste niveau in de klassestructuur. Het bevat alle informatie over de topologie en de testen en herstelmethoden die bepalen of het een geldig DNA is of niet. De contructor voert tijdens het aanmaken van het object deze testen achtereenvolgens uit. De variabelen van deze klasse zijn:

genome Het genoom, dit is een lijst binaire tekens die de structuur van het DNA voorstelt, zonder de dimensie te specifiëren.

x, y, n (int) De dimensies en het aantal bits.

valid (Boolean) Geeft aan of dit DNA de controles doorstaan heeft en dus bruikbaar is.

hole_map 2D-kaart van de holtes, die elk een eigen label (nummer) dragen.

hole_type Lijst volgens holtelabel die aangeeft of deze opblaasbaar (1) is of niet (0).

hole_area Lijst volgens holtelabel met de groottes van de holtes.

hole_nr Label/nummer van de grootste holte.

De volgende methodes zijn in DNA te vinden, de implementatie van de controles is beschreven in deel 3.2.

get_md5name() Berekent de md5-naam, de MD5-checksum van het genoom.

is_valid() Retourneert `self.valid`.

consistency_test() Controle van de samenhang.

rupture_test() Controleert puntcontacten. Het herstel, beschreven in 3.3.4, is er in opgenomen en gebeurt gelijktijdig met de controle.

inflatability_test() Controleert of er een opblaasbare holte (1) is in `hole_type` nadat `hole_map()` reeds uitgevoerd is.

recursive_edit_neighbours(genome_copy, n, old=1, new=0, area=1) Veelzijdige methode om vanaf een bepaalde cel recursief de aanliggende cellen met een bepaalde waarde te veranderen. Houdt de behandelde oppervlakte bij en voor holtes ook of ze gesloten zijn.

get_certain_neighbours(genome_copy, n, old=0, x=0) Wordt gebruikt door bovenstaande methode en door `DNAtoINP` om de buren van een cel op te halen die een bepaalde waarde hebben.

get_biggest_cavity() Retourneert het label van de grootste opblaasbare holte. Eénmalig gebruikt om `hole_nr` in te stellen.

map_holes() Wordt gebruikt om de holtes in kaart te brengen en retourneert `hole_map`, `hole_type` en `hole_area`.

fenotype() Print een 2D-grafische voorstelling van de structuur in de terminal voor debugging.

F.7 De Abaqus parser DNAtoINP

Dit bestand is één lang script dat de methode `parse(dna)` bevat die gebruikt wordt om een Abaqus input bestand aan te maken voor een DNA-object in `DNA.generate_file()`. In appendix E is een voorbeeld van een dergelijk bestand te zien en in het deel 3.4 is de implementatie van de belangrijkste onderdelen van deze methode besproken.

F.8 De RUNandFIT file

Dit script dat de methode `simulate(name)` omvat, zorgt ervoor dat een Abaqus job aangemaakt wordt om het input bestand `name.inp` te simuleren. `DNA.simulate_result()` start dit script. Nadat de job uitgevoerd is, worden de verplaatsingen van de gewenste knoop uitgelezen uit de output database en wordt hieruit berekend hoe goed aan het verplaatsings objectief voldaan is.

F.9 QUICKSORT

Deze hulpmodule is een eigen implementatie van het Quicksort-algoritme voor het sorteren van lijsten vectoren aan de hand van een objectief. Sorteren gebeurt door `sort(vectors, func, k=0)` uit te voeren. Func is een dynamische functie die op de vectoren aanroepen wordt als criterium waarop de vectoren gesorteerd moeten worden. Dit kan `get_objective()` zijn waarbij k het objectief aanduidt, of `get_distance` om te sorteren op de crowding distance.

F.10 PLOTTER

Deze laatste hulpklasse gebruikt het pakket matplotlib om de gewenste figuren te genereren van de convergentie van de actieve objectieven en van de optimalisatie-fronts. Dit gebeurt door de data opnieuw in te laden uit de pickle bestanden en de gegenereerde figuren op te slaan als PDF files. De grafieken in hoofdstuk 6 zijn met deze klasse gemaakt. De module bevat een functie om de convergentie van een objectief weer te geven (`plot_evolution(obj)`), om de fronten van generaties te visualiseren (`plot_front(generation, obj1, obj2)`)en om front 1 van een generatie te analyseren en de vectoren met de laagste afwijking aan te duiden (`analyse(obj1, obj2)`).

Bijlage G

Python code

In appendix F werd voor elke module besproken wat haar specifieke taak was in de solver en welke klasses, parameters en methodes er in voorkwamen. In deze appendix is de volledige code te vinden, voor de duidelijkheid en reproduceerbaarheid van het werk in deze masterproef.

Voor een korte toelichting bij de onderdelen van deze code kan appendix F gebruikt worden. De code zelf is ook van commentaren voorzien voor de leesbaarheid. De code en ingevoegde commentaar zijn echter volledig in het Engels.

G.1 CONFIG.py

```
__author__ = 'Manu'
# This is the file containing all variables
# If a user wants to change any parameter, only the values listed here have to be changed
##############################################################################################
# WARNING:  to save time, earlier results are used if found for the same geometry
#           if parameters were changed here, these results are wrong however!
#           clear your C:\Temp folder in that case before launching!

# Defining parameters (I/O)
abaqus_run = False
restart = False
save_csv = False
save_pickle = False
save_graphs = False
save_fronts = False

# SYSTEM PARAMETERS
# ---------------------
# ##### PC #####
path = "C:\\Users\\Manu\\Dropbox\\LEUVEN\\!!THESIS\\Python\\Parser\\"
result_path = "C:\\Users\\Manu\\Dropbox\\LEUVEN\\!!THESIS\\Python\\Output\\"
calc_path = "C:\\Temp\\"
# ##### VSC #####
# path = "/vsc-hard-mounts/leuven-user/312/vsc31246/Parser/"
# result_path = "/vsc-hard-mounts/leuven-user/312/vsc31246/Output/"
# calc_path = "/data/leuven/312/vsc31246/"
pickle_file = "pickledump"
pickle_egg = "EGG"
```

```
# DNA PARAMETERS
# ------------------
x = 6  # x size
y = 6  # y size
n = int(x*y)  # DO NOT CHANGE, amount of bits/cells/genes

# ALGORITHM PARAMETERS
# ------------------
population_size = None  # number of Vectors in a generation
generations = None  # number of generations
crossover_chance = 95  # chance of crossover in percentages
mutation_interval = int(n)  # mutate one bit every mutation_interval bits
upper_bounded = False  # rejects offspring vectors with
                       # worse deviation_score than worst of pool

# ACTIVE OBJECTIVES
# ------------------
# Change boolean to choose objective
# To add an objective:
# put the associated calculation in Generation.evaluate() and a variable in Vector
deviation_objective = False
dx = 0.2
dy = -0.2
weight_objective = True
similarity_objective = False
determinant_objective = True
objectives = [deviation_objective, weight_objective,
              similarity_objective, determinant_objective]

# PARSER PARAMETERS
# ------------------
grid_refinement = 4  # How many FEM cells in the width and height of a topology cell
cell_edge = 0.1  # DNA structure cell edge length  [mm]
# FEM element type
element_type = "CPE4R"
# Load
pressure = 0.5  # Applied pressure [MPa]
# BC
#   fixed
# Material model
density = 9.2e-10  # for PDMS-10 [tonne/mm3]
model_info = "Hyperelastic, n=2, ogden"  # 2 full lines of input file
#               mu 1       alpha 1    mu 2      alpha 2     D1          D2
# analysis aggregate:
# model_param = "3.04649, 0.106509,   3.1924, 0.111909,   0.0007,   0.0007"
# thesis Jordens 2011
model_param = "0.192509, 5.26851,   2.67748, -0.0505378,   0.0007,   0.0007"
# Output zone
#    fixed

##############################################################################

# TO START THE ALGORITHM FROM THIS FILE
# -------------------------------------

def main():
    import sys
    sys.path.append(path)
    import GATESTER # 'unused' but does run the code, importing = running in python

if __name__ == "__main__":
    main()
```

G.2 Hoofdfile

```
__author__ = 'Manu'
import CONFIG
if CONFIG.abaqus_run:  # import the abaqus specific libraries
    from odbAccess import *
from Genepool import PoolTESTER

# Initialise the gene pool
print("# MAIN STARTED")
POOL = PoolTESTER()
if CONFIG.restart:
    POOL.restart()
else:
    POOL.genesis()

# Run the optimisation for a desired number of generations
for i in range(CONFIG.generations):
    POOL.evolve()

# Terminate the evolution, save data
print("# MAIN FINISHED, RETURNING RESULTS")
POOL.terminate()

# Code to save .pdf's of all figures if desired
if CONFIG.save_graphs or CONFIG.save_fronts:
    print("# GENERATING FIGURES")
    from PLOTTER import PLOTTER
    plotter = PLOTTER()
    if CONFIG.save_graphs:
        for i in range(len(CONFIG.objectives)):
            if CONFIG.objectives[i]:
                plotter.plot_evolution(i)
        plotter.analyse(0, 1)
    if CONFIG.save_fronts:
        plotter.plot_fronts(0, 1)

print("# END OF TRANSMISSION ////////////////////////////////////////////////")
```

G.3 Klasse Genepool

```
import random
import os
import copy
import pickle
import time as timer
import numpy as np
import DNA
import Generation
import Vector
import CONFIG
import QUICKSORT
__author__ = 'Manu'


# This class contains all generations of vectors and methods to make the evolutionary
# process run. Running the process requires calling genesis(), calling evolve() as
# often as desired, and calling terminate()
class PoolTESTER():

    # Constructor to initiate objects of this class
    def __init__(self):
        self.pool = None
```

```
        self.age = 0
        self.result_path = os.path.dirname(CONFIG.result_path)
        # RESULT DATA STRUCTURE
        l = [list([]) for i in range(len(CONFIG.objectives))]
        lineage = {'winners': copy.deepcopy(l), 'bests': copy.deepcopy(l),
                   'averages': copy.deepcopy(l), 'worsts': copy.deepcopy(l),
                   'members': []}
        offspring = {'winners': copy.deepcopy(l), 'bests': copy.deepcopy(l),
                     'averages': copy.deepcopy(l), 'worsts': copy.deepcopy(l),
                     'members': []}
        self.RES = {'lineage': lineage, 'offspring': offspring, 'fronts': []}
        # First and last list in [fronts] will be empty (first is empty to match front
        # names and indices).

    # Creates the first generation of random vectors to start the algorithm
    def genesis(self):
        # print("# GENEPOOL AND FIRST GENERATION EMERGE")
        members = []
        while len(members) < CONFIG.population_size:
            genome = []
            for x in range(CONFIG.n):  # Create random genomes
                genome.append(random.getrandbits(1))
            genome = np.array(genome)
            dna = DNA.DNA(genome, CONFIG.x, CONFIG.y)  # Make a DNA object
            if dna.valid:  # Test if the DNA is valid
                chosen_one = Vector.Vector(dna)
                members.append(chosen_one)
        # Calculate, sort and rank new pool
        pool = Generation.Generation(members)
        pool.evaluate(pool.members[0])
        F = self.fast_nondominated_sort(pool.members)
        P = []
        for i in F:
            if len(i) > 0:
                self.crowding_distance_assignment(i)
                P.extend(i)
        self.pool = Generation.Generation(P)
        self.rank(self.pool, 'lineage')

    # Advances to the next generation. Offspring is created, validated and merged
    # with the genepool through selection.
    def evolve(self):
        self.tick()  # Age all
        # print("# NEW GENERATION FROM MERGE, GENERATION NR: %d" % self.age)
        # Make an offspring generation #'age'
        offspring = self.reproduce()
        # Evaluate offspring 'age'
        offspring.evaluate(self.pool.members[0])
        # Evaluate pool
        # self.pool.evaluate(self.pool.members[0])
        # STORE DATA IN 'RES'
        ###################################
        # Rank and generate statistics
        self.rank(offspring, 'offspring')  # STORE winner, best, average, worst DATA
        # self.rank(self.pool, 'lineage')  # STORE winner, best, average, worst DATA
        # STORE MEMBERS DATA: Add both generations to an archive
        # self.RES['lineage']['members'].append(copy.copy(self.pool.members))
        # self.RES['offspring']['members'].append(copy.copy(offspring.members))
        if CONFIG.save_csv:
            # WRITE DATA TO FILE
            ###################################
            # CSV Generation #'age'
            name = "generation"
```

```
            self.csvwrite(name, self.pool)
            # CSV Offspring #'age'
            name = "offspring"
            self.csvwrite(name, offspring)
        # Real NSGA-II main code
        ################################
        R = []
        R.extend(offspring.members)
        R.extend(self.pool.members)
        # Code to reject results with larger deviation than latest worst one
        if CONFIG.upper_bounded:
            L = QUICKSORT.sort(self.pool.members, QUICKSORT.get_objective, 1)
            pool_max = L[-1].deviation
            for vector in R:
                if pool_max < vector.deviation:
                    R.remove(vector)
        F = self.fast_nondominated_sort(R)
        # STORE the fronts
        fronts = copy.deepcopy(F)
        for i in fronts:
            QUICKSORT.sort(i, QUICKSORT.get_objective, 1)
            for j in range(len(i)):
                # For filesize, only store objectives and first 10 chars of md5name
                temp = i[j].get_objectives()[:2]
                temp.append(i[j].get_md5name()[:5])
                i[j] = temp
        self.RES['fronts'].append(fronts)
        # Build a new population from merge
        P = []
        i = 1
        while (len(P) + len(F[i])) < CONFIG.population_size:
            self.crowding_distance_assignment(F[i])
            P.extend(F[i])
            i += 1
        self.crowding_distance_assignment(F[i])
        F_i = QUICKSORT.sort(F[i], QUICKSORT.get_distance)
        F_i.reverse()
        P.extend(F_i[0:(CONFIG.population_size - len(P))])
        self.pool = Generation.Generation(P)
        self.rank(self.pool, 'lineage')  # STORE winner, best, average, worst DATA
        # pickle the data (as a crash checkpoint)
        ################################
        if CONFIG.save_pickle:
            self.pickle()

    # Help method of evolve
    # Make all vectors in the generation 1 time unit older
    def tick(self):
        self.age += 1
        for m in self.pool.members:
            m.age += 1

    # Help method of evolve
    # Sorts vectors on objective fit and generates statistics
    def rank(self, gen, type):
        for i in range(len(CONFIG.objectives)):
            if CONFIG.objectives[i]:
                QUICKSORT.sort(gen.members, QUICKSORT.get_objective, i)
                self.RES[type]['bests'][i].append(gen.members[0].get_objectives()[i])
                self.RES[type]['winners'][i].append(gen.members[0].get_md5name())
                self.RES[type]['worsts'][i].append(gen.members[-1].get_objectives()[i])
                sum = float(0)
                for vec in gen.members:
```

```
                sum += float(vec.get_objectives()[i])
            avg = sum/len(gen.members)
            self.RES[type]['averages'][i].append(avg)

    # Help method of evolve
    # generates a new offspring generation by mutating and crossing current pool
    def reproduce(self):
        youth = []
        while True:
            if len(youth) > CONFIG.population_size-1:  # Termination condition
                break
            # Take random genes, (just pass them, crossover breaks pointers)
            # NEWWWW
            one = self.binary_tournament()
            two = self.binary_tournament()
            i = 0  # NODUP
            while one.get_md5name() == two.get_md5name() \
                    and i < CONFIG.population_size:
                two = self.binary_tournament()
                i += 1
            [a, b] = self.crossover(one.dna.genome, two.dna.genome)
            a = self.mutate(a)
            # Process 'a' genome
            a = DNA.DNA(a, CONFIG.x, CONFIG.y)
            if a.is_valid() and not a.get_md5name() == one.get_md5name() and not \
                    a.get_md5name() == two.get_md5name():  # NODUP"""
                a = Vector.Vector(a)
                youth.append(a)
            # only if necessary process the 'b' genome
            if len(youth) > CONFIG.population_size-1:  # termination condition
                break
            b = self.mutate(b)
            b = DNA.DNA(b, CONFIG.x, CONFIG.y)
            if b.is_valid() and not b.get_md5name() == one.get_md5name() and not \
                    b.get_md5name() == two.get_md5name():  # NODUP
                b = Vector.Vector(b)
                youth.append(b)
        # Make a generation object to return
        children = Generation.Generation(youth)
        return children

    # Help method of reproduce, for binary tournament selection
    def binary_tournament(self):
        one = random.choice(self.pool.members)
        two = random.choice(self.pool.members)
        if one.rank < two.rank:
            winner = one
        elif one.rank > two.rank:
            winner = two
        elif one.distance > two.distance:
            winner = one
        else:
            winner = two
        return winner

    # Help method of reproduce
    # Returns two dna genome lists
    def crossover(self, one, two):
        # Break pointers
        one = copy.deepcopy(one)
        two = copy.deepcopy(two)
        # Decide on crossover or not
        probability = [False]*(100 - CONFIG.crossover_chance) \
```

```
                    + [True]*CONFIG.crossover_chance
        if not random.choice(probability):
            return one, two
        # Put in matrix representation
        A = np.array(one).reshape((CONFIG.y, CONFIG.x))
        B = np.array(two).reshape((CONFIG.y, CONFIG.x))
        # If crossover, rows or columns to switch?
        flip = random.choice([True, False])
        if flip:
            # If rows, transpose and use same method as columns
            A = np.transpose(A)
            B = np.transpose(B)
            bound = CONFIG.y
        else:
            bound = CONFIG.x
        # Randomly select how many columns to switch and starting from where
        width = random.randint(1, int(bound/2))
        start = random.randint(0, bound - width)
        # Change the chosen regions
        temp = copy.copy(A[:, start:(start + width)])
        A[:, start:(start + width)] = copy.copy(B[:, start:(start + width)])
        B[:, start:(start + width)] = temp
        # If rows, transpose again
        if flip:
            A = np.transpose(A)
            B = np.transpose(B)
        # Restore array shape
        A = np.array(A).reshape((1, CONFIG.n))[0]
        B = np.array(B).reshape((1, CONFIG.n))[0]
        return A, B

    # Help method of reproduce
    # Returns a genome
    def mutate(self, genome):
        genome = copy.deepcopy(genome)  # Immediately break pointer
        probability = [True] + [False]*(CONFIG.mutation_interval-1)
        for k in range(len(genome)):
            if random.choice(probability):
                genome[k] = int((genome[k] + 1) % 2)
        return genome

    # Help method of evolve
    # Part of NSGA-II
    # Sorts all vectors in P in fronts of mutually non-dominating vectors
    def fast_nondominated_sort(self, P):
        F = [[], []]  # Match the indexes so rank 1 is in F[1]
        for p in P:
            p.S = []
            p.n = 0
            # Build list of dominated vectors and domination counter
            for q in P:
                if p.dominates(q):
                    p.S.append(q)
                elif q.dominates(p):
                    p.n += 1
            # Extract first front
            if p.n == 0:
                p.rank = 1
                F[1].append(p)
        i = 1
        n = 0  # SPEEDUP
        while not len(F[i]) == 0 and not n >= CONFIG.population_size:  # SPEEDUP
            Q = []
```

```
            for p in F[i]:
                # Decrease domination score of vectors dominated by last front
                for q in p.S:
                    q.n -= 1
                    # Build next front
                    if q.n == 0:
                        q.rank = i + 1
                        Q.append(q)
            i += 1
            n += len(Q)  # SPEEDUP
            F.append(Q)
            # Empty all S lists when sorting is done
        for p in P:
            p.S = None
        return F

    # Help method of evolve
    # Part of NSGA-II
    # Calculates the area represented by a solution to judge its importance
    def crowding_distance_assignment(self, L):
        l = len(L)
        for i in L:
            i.distance = 0
        for k in range(len(CONFIG.objectives)):
            if CONFIG.objectives[k]:
                L = QUICKSORT.sort(L, QUICKSORT.get_objective, k)
                # Ends of interval are most important
                L[0].distance = float("inf")
                L[-1].distance = float("inf")
                # Distance in each objective is the normalised distance between its
                # neighbours
                for j in range(1, l-1):
                    L[j].distance += \
                        ((L[j+1].get_objectives()[k] - L[j-1].get_objectives()[k])
                         / (max(L[-1].get_objectives()[k]-L[0].get_objectives()[k], 1)))

    # End the evolutionary process
    def terminate(self):
        self.tick()  # Age
        print("# Terminating with final generation: %d" % self.age)
        # STORE DATA
        ####################################
        # Rank and generate statistics: STORE winner, best, average, worst DATA
        self.rank(self.pool, 'lineage')
        # Add pool to an archive: STORE members data
        # self.RES['lineage']['members'].append(copy.copy(self.pool.members))
        # WRITE DATA TO FILE
        ####################################
        if CONFIG.save_csv:
            # CSV Generation x
            name = "generation"
            self.csvwrite(name, self.pool)
        # pickle the data for image generation scripts and restart
        ####################################
        if CONFIG.save_pickle:
            self.pickle()

    # Alternative for genesis(), restart from a saved generation.
    # Requires save_pickle = True
    def restart(self):
        assert CONFIG.save_pickle
        pickle_file_name = os.path.join(self.result_path, CONFIG.pickle_file)
        with open(pickle_file_name + "-L", 'rb') as f:
```

```
            self.RES['lineage'] = pickle.load(f)
            f.close()
        with open(pickle_file_name + "-O", 'rb') as f:
            self.RES['offspring'] = pickle.load(f)
            f.close()
        # with open(pickle_file_name + "-F", 'rb') as f:
            # self.RES['fronts'] = pickle.load(f)
            # f.close()
        # Load EGG as pool, set age
        filename = os.path.join(CONFIG.result_path, CONFIG.pickle_egg)
        with open(filename, 'rb') as egg:
            self.pool = pickle.load(egg)
            egg.close()
        self.age = len(self.RES['lineage']['bests'][1])

    # Help method to store results in csv file per generation
    # Places all md5 names and active objective values on one line
    def csvwrite(self, name, generation):
        file_name = os.path.join(self.result_path, "csv-" + name + ".csv")
        file = open(file_name, 'a')
        BOL = str(timer.clock()) + ", "
        file.write(BOL)
        for i in generation.members:
            l = i.get_objectives()
            p = i.get_md5name() + ", "
            for k in range(len(l)):
                if CONFIG.objectives[k]:
                    p += "%s, " % l[k]
            file.write(p)
        file.write("\n")
        file.close()

    # Help method to store RES and last pool in pickle file
    # Needed to make optimisation restartable
    def pickle(self):
        pickle_file_name = os.path.join(self.result_path, CONFIG.pickle_file)
        with open(pickle_file_name + "-L", 'w') as f:
            pickler = pickle.Pickler(f)
            pickler.dump(self.RES['lineage'])
            f.close()
        with open(pickle_file_name + "-O", 'w') as f:
            pickler = pickle.Pickler(f)
            pickler.dump(self.RES['offspring'])
            f.close()
        with open(pickle_file_name + "-F%d" % self.age, 'w') as f:
            pickler = pickle.Pickler(f)
            pickler.dump(self.RES['fronts'][-1])
            f.close()
        # Create an EGG to start from
        pickle_file_name = os.path.join(self.result_path, CONFIG.pickle_egg)
        with open(pickle_file_name, 'wb') as f:
            pickler = pickle.Pickler(f)
            pickler.dump(self.pool)
            f.close()
```

G.4 Klasse Generation

```
__author__ = 'Manu'
import numpy as np
import CONFIG
# This class contains all vectors of a specific generation
# and after running evaluation() their evolutionary scores as well
```

```
class Generation():
    # Initialise with a list of Vector objects
    def __init__(self, members):
        self.members = members  # holds all vectors

    # Calculate scores for active objectives
    def evaluate(self, best):
        # Calculate movement objective
        if CONFIG.objectives[0]:
            for i in self.members:
                # Parse .inp file
                i.generate_file()
            for i in self.members:
                # Launch Abaqus calculation
                i.simulate_result()
        # Calculate other get_objectives
        for i in self.members:
            if CONFIG.objectives[1]:
                i.weight = sum(i.dna.genome)
            if CONFIG.objectives[2]:
                i.similarity = np.sum(i.dna.genome == best.dna.genome)
            if CONFIG.objectives[3]:
                [s, l] = np.linalg.slogdet(i.dna.genome.reshape([CONFIG.y, CONFIG.x]))
                i.determinant = CONFIG.x + CONFIG.y - np.exp(l)
            # Add new objectives here
```

G.5 Klasse Vector

```
import DNA
import DNAtoINP
import RUNandFIT
import CONFIG
import os
# This class represents a single vector in some generation in some genepool
# It has specific DNA, a map of it's cavities and their types (enclosed or not)
# and a record of it's dimension and validity


class Vector():
    def __init__(self, dna):
        description = "To generate a vector, DNA has to be provided"
        # Vectors can only live with valid DNA
        assert dna.is_valid()
        self.dna = dna
        # Log how many generations this DNA/vector has survived
        self.age = 0
        # LIST OF OBJECTIVES
        # ------------------
        # Also add new objectives to get_objectives() below!
        self.deviation = float("inf")
        self.path = None
        self.weight = CONFIG.n
        self.similarity = CONFIG.n
        self.determinant = 0
        # NSGA-II STUFF
        # -------------
        #self.crowding_distance = 0  # Crowding distance for NSGA-II
        self.rank = 0  # Nondomination rank
        self.distance = 0  # Crowding distance rank
        self.n = None  # Domination counter
        self.S = None  # Solutions dominated by this Vector
```

```
    def get_md5name(self):
        return self.dna.get_md5name()

    def get_objectives(self):
        return [self.deviation, self.weight, self.similarity, self.determinant]

    def generate_file(self):
        if not os.path.isfile("inp-" + self.get_md5name() + ".inp"):
            DNAtoINP.parse(self.dna)

    def simulate_result(self):
        if self.path is None:
            [self.path, self.deviation] = RUNandFIT.simulate(self.get_md5name())

    def dominates(self, q):
        domination = True
        for i in range(len(CONFIG.objectives)):
            if CONFIG.objectives[i] and self.get_objectives()[i] > q.get_objectives()[i]:
                domination = False
        return domination

    def __str__(self):
        return self.get_md5name()
```

G.6 Klasse DNA

```
__author__ = 'Manu'
import random, os
from hashlib import md5
import numpy as np
from array import array
# This class just represents a string of DNA:
# a representation in a N^2 long array and a  N x N matrix


class DNA():
    # Initialise with a genome (list of bits) and the dimensions of the topology
    # Executes validity tests and associated calculations
    def __init__(self, genome, x, y):
        # DNA attributes
        self.genome = np.array(genome.copy())
        self.x = int(x)
        self.y = int(y)
        self.n = x * y
        # self.md5name = self.get_md5name()
        assert self.n == len(self.genome)
        # Sequence that controls the DNA's validity
        # If invalidity is detected, stop execution to speed things up
        self.valid = True
        self.rupture_test()
        if self.valid:
            self.consistency_test()
        # Make variables to store further information if useful (valid=True)
        if self.valid:
            [self.hole_map, self.hole_type, self.hole_area] = self.map_holes()
            # Continue mapping if inflatable
            self.inflatable = self.inflatability_test()
            if self.inflatable:
                self.hole_nr = self.get_biggest_cavity()

    # Use hash to get filename that does not get longer with DNA size
    # but is unique with a high certainty
    def get_md5name(self):
```

```
        return str(md5(str(self.genome).encode()).hexdigest())

    # Returns true of false based on whether the DNA can generate a valid specimen
    def is_valid(self):
        return self.valid

    # ########################## TESTS #############################

    # Tests whether the DNA describes a specimen without any separated parts
    def consistency_test(self):
        genome_copy = self.genome.copy()
        if sum(genome_copy) == 0:
            self.valid = False
            return
        n = 0
        # Loop over the genes to find the first '1' (polymer present)
        for x in genome_copy:
            if x == 0:
                n += 1
            else:
                break
        # Make this 1-bit a 0-bit and
        # recursively do the same with its neighbours and their neighbours
        genome_copy[n] = 0
        [result, l, a] = self.recursive_edit_neighbours(genome_copy, n)
        # Remaining '0's indicate the described structure is not one part
        # So the evaluate of all should be zero
        if sum(result) == 0:
            return
        else:
            self.valid = False

    # Test for corner contacts (and immediately correct them if wanted)
    # Tests if there are corners of material connected,
    # this will most likely cause errors in Abaqus
    def rupture_test(self):
        # Test 2x2 squares starting with upper left corner
        for i in range(self.x * (self.y - 1) - 1):
            if not ((i % self.x) == (self.x - 1)):  # Last column is not tested
                if (self.genome[i] ^ self.genome[i + self.x]) \
                        and (self.genome[i] ^ self.genome[i + 1]) \
                        and (self.genome[i + 1] ^ self.genome[i + self.x + 1]):
                    # IMPORTANT: the first two lines line make it a test
                    # The second and third line a correction
                    # ------ (choose between both by commenting the other)
                    # self.valid = False
                    # break
                    self.genome[i + self.x] = 1
                    self.genome[i + self.x + 1] = 1

    # Tests if there is a closed cavity present
    def inflatability_test(self):
        inflatable = (sum(self.hole_type) > 4)
        if not inflatable:
            self.valid = False
        return inflatable

    # ###################### HELP METHODS ##########################

    # General function to replace old by new in recursive neighbours (used in Vector)
    def recursive_edit_neighbours(self, genome_copy, n, old=1, new=0, area=1):
        # Get the positions in the array of the real neighbours with value 'old'
        [neighbours, cavity_flag] = self.get_certain_neighbours(genome_copy, n, old)
```

```
        # Update the area to figure out how big the cavity is
        new_area = len(neighbours)
        area += new_area
        # Change value from old to new
        for x in neighbours:
            genome_copy[x] = new
        # Recurse!
        for x in neighbours:
            [junk, flag, area] = self.recursive_edit_neighbours(genome_copy, x, old, new, area)
            cavity_flag = cavity_flag * flag
        return genome_copy, cavity_flag, area

    # Get the real matrix-neighbours of element 'n' in the genome list with value 'old'
    def get_certain_neighbours(self, genome_copy, n, old=0, x=0):
        if x == 0:
            x = self.x
        # Get the positions in the array of the four neighbours
        possible_neighbours = [n - 1, n + 1, n - x, n + x]
        neighbours = []
        cavity_flag = 1
        # Loop over them
        for i in possible_neighbours:
            # Ignore indexes out of bounds, not all elements have 4 neighbours
            if i < 0 or i > len(genome_copy) - 1:
                cavity_flag = 0  # Near top or bottom edge, so not a real cavity
            # Ignore 'neighbours' that wrap over a left or right edge
            # with a modulo of the indices of with the row length
            # Memo: XOR is no option, 0 and 1 are not the only possibilities here
            elif (n % x == x - 1 and i % x == 0) or (n % x == 0 and i % x == x - 1):
                cavity_flag = 0  # Near left or right edge, so not a real cavity
            # Check whether this neighbour has the desired value
            elif genome_copy[i] == old:
                neighbours.append(i)
        return neighbours, cavity_flag

    # Look for the biggest inflatable cavity
    def get_biggest_cavity(self):
        biggest = 0
        for p in range(len(self.hole_type)):
            if self.hole_type[p] == 1 and (self.hole_area[p] > self.hole_area[biggest]):
                biggest = p
        return biggest

    # ######################## CORE METHODS ##########################

    # Locates all cavities and cutouts at the edges and makes a distinction between them
    def map_holes(self):
        hole_map = self.genome.copy()  # Map of the holes
        hole_type = [2, 2]  # Open or closed hole parameter for each hole
        hole_area = [0, 0]  # Amount of cells a hole spans
        # Position being analysed
        n = 0
        # Cavity label
        label = 2
        while n < len(hole_map):
            if hole_map[n] != 0:
                n += 1
            elif hole_map[n] == 0:
                # Give this '0' another value 'label' and recursively do the same
                # with its neighbours and their neighbours
                hole_map[n] = label
                [map, cavity_flag, area] = self.recursive_edit_neighbours(hole_map, n, 0, label)
                hole_map = map.copy()
```

```
                label += 1
                hole_type.append(cavity_flag)  # Inflatable or not
                hole_area.append(area)  # To be able to pick the biggest one
        return hole_map, hole_type, hole_area

    # ######################## VARIA ######################

    # Just delivers an understandable textual form when printing the DNA object
    def __repr__(self):
        text = ''.join(map(str, self.genome)) + "␣Size:%dx%d" % (self.x, self.y)
        return text

    # Returns a string with a 2D representation of the topology
    def fenotype(self):
        output = ""
        for i in range(self.y):
            for j in range(self.x):
                if self.genome[j + self.x * i] == 1:
                    output += "1"
                elif self.genome[j + self.x * i] == 0:
                    output += "0"
            output += "\n"
        return output
```

G.7 DNAtoINP.py

```
__author__ = 'Manu'
import numpy as np
from array import array
import os
import CONFIG
# This parser will write an abacus input file for a given DNA, based on


def parse(dna):
    pressure = CONFIG.pressure
    # How many grid elements for one side of a square of the topology?
    grid_refinement = CONFIG.grid_refinement
    # Topology cell side length
    cell_edge = CONFIG.cell_edge
    # Length of the side of an element
    edge = cell_edge / grid_refinement

    ##################################################
    ############## DATA                ###############
    ##################################################
    # transfer data
    n = dna.n
    x = dna.x
    y = dna.y
    hole_nr = dna.hole_nr
    # refined dimensions
    X = x * grid_refinement
    Y = y * grid_refinement
    N = X * Y

    # transfer and convert to numpy
    genome = np.array(dna.genome)
    hole_map = np.array(dna.hole_map).reshape([y, x])

    # GEN: matrix version of genome, enlarged
    gen = np.array(genome).reshape([y, x])  # reshape dna array to matrix
    gen = np.repeat(np.repeat(gen, grid_refinement, axis=0), grid_refinement, axis=1)
```

```
    # GENOME: array version, reshaped from gen to get enlarged version
    genome = gen.reshape([1, N])[0]

    # HOLE_MAP: matrix that maps holes, only needed to compute edge_map here
    hole_map = np.repeat(np.repeat(hole_map, grid_refinement, axis=0), grid_refinement, axis=1)

    # EDGE_MAP: Make an edge map
    # Use the hole map, make edge cells 0 (as there are no 0's at start)
    edge_map = np.copy(hole_map)
    edge_map = edge_map.reshape([1, N])[0]
    for h in range(len(edge_map)):
        if edge_map[h] == hole_nr:
            [neighbours, junk] = dna.get_certain_neighbours(edge_map, h, 1, X)
            for g in neighbours:
                edge_map[g] = 0

    # NODES: initialize empty node map (1 larger than element map (gen) in both dimensions)
    nodes = np.array([[0] * (X + 1)] * (Y + 1))

    ##################################################
    ############### FILE INIT          ###############
    ##################################################
    inp_dir = os.path.dirname(CONFIG.calc_path)
    filename = "I-" + dna.get_md5name() + ".inp"
    full_path = os.path.join(inp_dir, filename)
    file = open(full_path, 'w')

    ##################################################
    ############### HEADER             ###############
    ##################################################
    file.write("*Heading\n\
** Job name: %s Model name: Model-1\n\
** Generated by: Abaqus/CAE Student Edition 6.13-2\n\
*Preprint, echo=NO, model=NO, history=NO, contact=NO\n" % filename)

    ##################################################
    ############### PARTS              ###############
    ##################################################
    file.write('**\n\
** PARTS\n\
**\n\
*Part, name=Part-1\n\
*Node\n')
    # generate nodes
    # Copy the cell map 4 times for the four nodes of each cell
    # zero's in the resulting map correspond to nodes that aren't needed in any element
    nodes[0:Y, 0:X] = nodes[0:Y, 0:X] + gen
    nodes[0:Y, 1:X + 1] = nodes[0:Y, 1:X + 1] + gen
    nodes[1:Y + 1, 0:X] = nodes[1:Y + 1, 0:X] + gen
    nodes[1:Y + 1, 1:X + 1] = nodes[1:Y + 1, 1:X + 1] + gen
    # node map found, name and write them
    nn = 0  # node number
    for i in range(Y + 1):
        for j in range(X + 1):
            if nodes[i, j] > 0:  # node exists
                nn += 1  # raise before use so highest value is saved
                nodes[i, j] = nn
                file.write('      %d,         %f,         %f\n' % (nn, j * edge, -edge * i))
    # type of element (can be an option later
    file.write('*Element, type=%s\n' % CONFIG.element_type)

    # list of elements
```

```
    e = 0  # element number
    for k in range(Y):
        for l in range(X):
            if gen[k, l] == 1:  # element exists
                e += 1  # raise before use so highest value is saved
                gen[k, l] = e  # save the node number at its location in the matrix
                genome[k * X + l] = e  # save the node number at its location in the list
                # write the number of the node
                file.write('%d, %d, %d, %d, %d\n' % (e, nodes[k, l],
                           nodes[k + 1, l], nodes[k + 1, l + 1], nodes[k, l + 1]))

    # /!\ At this point, nodes and gen contain the specific numbers,
    # not just a flag whether they exist
    # easy for later use in BC an Loads

    # Generate nodeset and element set
    file.write('*Nset, nset=Set-1, generate\n\
 1,  %d,  1\n\
*Elset, elset=Set-1, generate\n\
 1,  %d,  1\n' % (nn, e))

    #Generate result node set
    #TODO option which node to take
    #TODO multiple sets for multiple get_objectives
    file.write('*Nset, nset=Res-1\n\
 %d,\n' % (nn))

    #Section (options later)
    file.write('** Section: Section-1\n\
*Solid Section, elset=Set-1, material=PDMS-10\n\
100.,\n\
*End Part\n')

    ##################################################
    ############### ASSEMBLY          ################
    ##################################################
    file.write('**\n\
**\n\
** ASSEMBLY\n\
**\n\
*Assembly, name=Assembly\n\
**\n\
*Instance, name=Part-1-1, part=Part-1\n\
*End Instance\n\
**\n')

    # Nodes and elements of BC
    file.write('*Nset, nset=Set-1, instance=Part-1-1\n')  #TODO option which node to take
    for k in range(Y):
        for l in range(X):
            if gen[k, l] == 1:
                for m in range(grid_refinement - 1):
                    file.write(' %d,  %d, ' % (nodes[k + m + 2, l], nodes[k, l + m + 2]))
                file.write(' %d,  %d,  %d\n' % (
                    nodes[k, l], nodes[k + 1, l], nodes[k, l + 1]))
                break
    file.write('*Elset, elset=Set-1, instance=Part-1-1, generate\n\
 1, 1, 1\n')

    # Surface elements of cavity
    # write the elements around the cavity after looking for them
    # specify which surfaces of the element face the cavity
    # 4 surfaces: #4# -> cells are grouped under right surface
```

```
    #              1 3
    #             #2#
    if dna.inflatable:
        S1 = array('I')
        S2 = array('I')
        S3 = array('I')
        S4 = array('I')
        for k in range(len(edge_map)):
            if edge_map[k] == 0:
                if edge_map[k - 1] == hole_nr:
                    S1.append(k)  # left side
                if k + X < N:  # Not out of bounds
                    if edge_map[k + X] == hole_nr:
                        S2.append(k)  # bottom
                if edge_map[k + 1] == hole_nr:
                    S3.append(k)  # right side
                if k - X > 0:  # Not out of bounds
                    if edge_map[k - X] == hole_nr:
                        S4.append(k)  # top
        # Now write all this information
        # if tests will put the line breaks
        file.write('*Elset, elset=_Surf-1_S1, internal, instance=Part-1-1')
        for k in range(len(S1)):
            if k % 15 == 0:
                # prevent putting more than 16 elements on a row, they will be deleted
                file.write('\n')
            file.write('%d, ' % genome[S1[k]])
        file.write('\n*Elset, elset=_Surf-1_S2, internal, instance=Part-1-1')
        for k in range(len(S2)):
            if k % 15 == 0:
                # prevent putting more than 16 elements on a row, they will be deleted
                file.write('\n')
            file.write('%d, ' % genome[S2[k]])
        file.write('\n*Elset, elset=_Surf-1_S3, internal, instance=Part-1-1')
        for k in range(len(S3)):
            if k % 15 == 0:
                # prevent putting more than 16 elements on a row, they will be deleted
                file.write('\n')
            file.write('%d, ' % genome[S3[k]])
        file.write('\n*Elset, elset=_Surf-1_S4, internal, instance=Part-1-1')
        for k in range(len(S4)):
            if k % 15 == 0:
                # prevent putting more than 16 elements on a row, they will be deleted
                file.write('\n')
            file.write('%d, ' % genome[S4[k]])
        # line break and standard input lines
        file.write('\n')
        file.write('*Surface, type=ELEMENT, name=Surf-1\n')
        file.write('_Surf-1_S1, S1\n')
        file.write('_Surf-1_S2, S2\n')
        file.write('_Surf-1_S3, S3\n')
        file.write('_Surf-1_S4, S4\n')

    file.write('*End Assembly\n')

    ##################################################
    ############## MATERIALS          ################
    ##################################################
    file.write('**\n\
** MATERIALS\n\
**\n\
*Material, name=PDMS-10\n\
*Density\n\
```

```
␣%.12f,\n' % CONFIG.density)
    file.write('*%s\n\
␣%s\n' % (
        CONFIG.model_info, CONFIG.model_param))  # Properties of material
    file.write("**␣————————————————————————————————————————————————————————————\n")

    ##########################################################
    ############## STEP                       ################
    ##########################################################
    file.write('**\n\
**␣STEP:␣Step-1\n\
**\n\
*Step,␣name=Step-1,␣nlgeom=YES\n\
*Static\n\
0.1,␣1.,␣1e-05,␣0.1\n')  # Could move to config

    ##########################################################
    ############## BOUNDARY CONDITIONS  ################
    ##########################################################
    file.write('**\n\
**␣BOUNDARY␣CONDITIONS\n\
**␣\n\
**␣Name:␣BC-1␣Type:␣Symmetry/Antisymmetry/Encastre\n\
*Boundary\n\
Set-1,␣ENCASTRE\n')

    ##########################################################
    ############## LOADS                      ################
    ##########################################################
    file.write('**\n\
**␣LOADS\n\
**␣\n')
    if dna.inflatable:
        file.write('**␣Name:␣Load-1␣␣␣Type:␣Pressure\n\
*Dsload\n\
Surf-1,␣P,␣%f\n' % pressure)  # Pressure value as option [MPa]

    ##########################################################
    ############## OUTPUT                     ################
    ##########################################################
    file.write('**\n\
**OUTPUT␣REQUESTS\n\
**␣\n\
*Restart,␣write,␣frequency=0\n\
**␣\n\
**␣FIELD␣OUTPUT:␣F-Output-1\n\
**␣\n\
*Output,␣field,␣variable=PRESELECT\n\
**␣\n\
**␣HISTORY␣OUTPUT:␣H-Output-1\n\
**␣\n\
*Output,␣history,␣variable=PRESELECT\n\
*End␣Step\n')

    ##########################################################
    ############## RETURN                     ################
    ##########################################################
    file.close()
```

G.8 RUNandFIT.py

```
__author__ = 'Manu'
import sys, os
```

```
import CONFIG
if CONFIG.abaqus_run:  # import the abaqus specific libraries
    from odbAccess import *
    from abaqus import *
    from abaqusConstants import *
# This module consists of one method
# It submits an .imp file, reads the .odb and fits the resulting path
# Returns the displacement path and the fit

def simulate(name):
    odb_path = os.path.dirname(CONFIG.calc_path)
    odb_file = os.path.join(odb_path, "O-" + name + '.odb')

    # Run model if it is new
    if not os.path.isfile(odb_file):
        inp = os.path.join(odb_path, "I-" + name + ".inp")
        id = "O-" + name
        job = mdb.JobFromInputFile(name=id, inputFileName=inp, numDomains=3, numCpus=3)
        job.submit()
        job.waitForCompletion()

    # DATA COLLECTION
    # Open output database
    odb = openOdb(path=odb_file)

    # Loop over all frames
    step_name = odb.steps.keys()[-1]
    step = odb.steps[step_name]
    nr_of_frames = len(step.frames)

    # Extract the displacement data
    PATH = []
    for f in range(nr_of_frames):
        # Get displacements
        field_values = odb.steps[step_name].frames[f].fieldOutputs['U']
        load_set = odb.rootAssembly.instances['PART-1-1'].nodeSets['RES-1']
        load_set_u = field_values.getSubset(region=load_set).values
        Us = [load_set_u[0].data[0], load_set_u[0].data[1]]
        PATH.append(Us)

    # close output database
    odb.close()

    # CALCULATE FIT
    ##############################
    # overly easy first implementation: move end to certain position
    deviation = (PATH[-1][1]-CONFIG.dy)**2 + (PATH[-1][0]-CONFIG.dx)**2

    return PATH, deviation
```

G.9 QUICKSORT.py

```
__author__ = 'Manu'


# Help function for sorting along objectives
def get_objective(item, k):
    return item.get_objectives()[k]


# Help function for sorting along crowding distance
def get_distance(item, k):
    return item.distance
```

```
# MAIN FUNCTION, quicksort list 'vectors' based on function(vectors, k)
def sort(vectors, func, k=0):
    L = vectors
    recurse(L, 0, len(L)-1, func, k)
    return L


# Recursive help function
def recurse(L, first, last, func, k):
    if first < last:
        split = partition(L, first, last, func, k)
        recurse(L, first, split-1, func, k)
        recurse(L, split+1, last, func, k)


# Partition help function
def partition(L, first, last, func, k):
    pivot = func(L[first], k)

    leftmark = first+1
    rightmark = last

    done = False
    while not done:
        while leftmark <= rightmark and func(L[leftmark], k) <= pivot:
            leftmark += 1
        while rightmark >= leftmark and func(L[rightmark], k) >= pivot:
            rightmark -= 1
        if rightmark < leftmark:
            done = True
        else:
            tmp = L[leftmark]
            L[leftmark] = L[rightmark]
            L[rightmark] = tmp

    tmp = L[first]
    L[first] = L[rightmark]
    L[rightmark] = tmp
    return rightmark
```

G.10 PLOTTER.py

```
__author__ = 'Manu'
import CONFIG
import os, io
import pickle
import matplotlib.pyplot as plt

class PLOTTER():
    def __init__(self):
        self.RES = {'lineage': None, 'offspring': None, 'fronts': [0]}
        # Load the data
        filename = os.path.join(CONFIG.result_path, CONFIG.pickle_file)
        with open(filename + "-L", 'rb') as f:
            unpickler = pickle.Unpickler(f, encoding="unicode_escape")
            self.RES['lineage'] = unpickler.load()
            f.close()
        with open(filename + "-O", 'rb') as f:
            unpickler = pickle.Unpickler(f, encoding="unicode_escape")
            self.RES['offspring'] = unpickler.load()
            f.close()
```

```
        with open(filename + "-F" + str(len(self.RES['lineage']['bests'][1])),
                  'rb') as f:
            self.RES['fronts'][0] = pickle.load(f, encoding="unicode_escape")
            f.close()
        print(self.RES['lineage']['bests'])
        print(self.RES['lineage']['winners'])
        print(self.RES['fronts'])

    # Plot the convergence
    def plot_evolution(self, obj):
        fig = plt.figure()
        ax = plt.subplot(111)  # New figure
        # plot all the different stats on the figure
        ax.plot(self.RES['lineage']['bests'][obj], 'r', label="MIN")
        ax.plot(self.RES['lineage']['averages'][obj], 'g', label="AVG")
        ax.plot(self.RES['lineage']['worsts'][obj], 'b', label="MAX")
        ax.plot(self.RES['offspring']['bests'][obj], 'r:', label="min")
        ax.plot(self.RES['offspring']['averages'][obj], 'g:', label="avg")
        ax.plot(self.RES['offspring']['worsts'][obj], 'b:', label="min")
        # Edit axes and text of figure
        ax.autoscale(tight=False)
        #ax.margins(0, 0.01, tight=False)
        plt.title('Convergentie van objectief nr %s' % obj)
        plt.ylabel('Waarde van objectief %d' % obj)
        plt.xlabel('Generation nummer')
        box = ax.get_position()
        ax.set_position([box.x0, box.y0 + box.height * 0.1,
                         box.width, box.height * 0.9])
        ax.legend(loc='upper center', bbox_to_anchor=(0.5, -0.1),
                  fancybox=True, shadow=True, ncol=6)
        plt.xlim([0, len(self.RES['lineage']['bests'][obj])])
        if obj == 0:
            plt.ylim([0, max(self.RES['offspring']['worsts'][obj])])
        else:
            plt.ylim([self.RES['lineage']['bests'][obj][-1]-1,
                      max(self.RES['offspring']['worsts'][obj])+1])
        # Save the figure as .pdf in the output folder
        name = "Graph-objective-%s" % obj
        filename = os.path.join(CONFIG.result_path, name)
        plt.savefig("%s.pdf" % filename)

    # Plot the fronts of a certain generation
    def plot_front(self, generation, obj1, obj2):
        # New variables and figure
        fig = plt.figure()
        ax = plt.subplot(111)
        n = generation  # generation to show
        x = []
        y = []
        # Make front data more accessible
        for i in range(min(7, len(self.RES['fronts'][n])-2)):
            x.append([])
            y.append([])
            for obj in self.RES['fronts'][n][i+1]:
                x[i].append(obj[obj1])
                y[i].append(obj[obj2])
            # Print the current front as step function
            # post in stead of pre might be necessary sometimes
            ax.step(x[i], y[i], 'o', where='pre', label="F%d[%d]" % (i+1, len(x[i])))
        # Make it aesthetically more pleasing
        ax.autoscale(tight=False)
        ax.margins(0.01, 0.01, tight=False)
        # Text of figure
```

```
        plt.title('Uitgebreide generatie nummer %d' % n)
        plt.ylabel('Objectief %d' % obj2)
        plt.xlabel('Objectief %d' % obj1)
        box = ax.get_position()
        ax.set_position([box.x0, box.y0, box.width * 0.8, box.height])
        ax.legend(loc='center left', bbox_to_anchor=(1, 0.5))
        ax.axis([0, max(x[0]), min(y[0]), max(y[0])])
        # Save figure as .pdf file
        name = "Fronts-obj(%d,%d)-gen%d" % (obj1, obj2, n)
        filename = os.path.join(CONFIG.result_path, name)
        plt.savefig("%s.pdf" % filename)

    # Save figure of all fronts at once
    def plot_fronts(self, a, b):
        for i in range(len(self.RES['fronts'])):
            self.plot_front(i, a, b)

    # Plot the fronts of a certain generation
    def analyse(self, obj1, obj2):
        # New variables and figure
        fig = plt.figure()
        ax = plt.subplot(111)
        x = []
        y = []
        label = []
        for obj in self.RES['fronts'][-1][1]:
            x.append(obj[obj1])
            y.append(obj[obj2])
            label.append(obj[-1])
        # Print the current front as step function
        # post in stead of pre might be necessary sometimes
        ax.step(x, y, where='pre', label="F%d[%d]" % (1, len(x)))
        for i in range(min(9, len(x))):
            ax.step(x[-i-1], y[-i-1], 'o', where='pre', label=label[-i-1])
            print( label[-i-1], x[-i-1], y[-i-1])
        # Make it aesthetically more pleasing
        ax.autoscale(tight=False)
        # Text of figure
        plt.title('Front 1 van laatste generatie')
        plt.ylabel('Objectief %d' % obj2)
        plt.xlabel('Objectief %d' % obj1)
        box = ax.get_position()
        ax.set_position([box.x0, box.y0, box.width * 0.8, box.height])
        ax.legend(loc='center left', bbox_to_anchor=(1, 0.5))
        ax.axis([0, max(x), min(y), max(y)])
        ax.margins(0.01, 0.01, tight=False)
        # Save figure as .pdf file
        name = "Fronts-analysis"
        filename = os.path.join(CONFIG.result_path, name)
        plt.savefig("%s.pdf" % filename)
```

Bibliografie

[1] ABAQUS UNIFIED FEA - complete solutions fot realistic simulation. *Dassault Systèmes* [Online]. Aug. 2015. URL: http://www.3ds.com/products-services/simulia/products/abaqus/.

[2] Python. *Python Software Foundation* [Online]. Aug. 2015. URL: https://www.python.org.

[3] G. Berselli, R. Vertechy, M. Pellicciari, and G. Vassura. Hyperelastic Modeling of Rubber-Like Photopolymers for Additive Manufacturing Processes. *Rapid Prototyping Technology - Principles and Functional Requirements*, pages 135–153, 2011.

[4] Chutsu. Antwoord op: How to find the best parameters for a Genetic Algorithm? [Online]. Jul. 2015. URL: https://stackoverflow.com/questions/1075628/how-to-find-the-best-parameters-for-a-genetic-algorithm.

[5] D. Corne, N. Jerram, J. Knowles, M. Oates, and J. Martin. PESA-II: Region-based Selection in Evolutionary Multiobjective Optimization. *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2001)*, pages 283–290, 2001.

[6] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, 2002.

[7] Herculesstichting. The KU Leuven/UHasselt cluster (ThinKing and Cerebro). [Online]. Aug. 2015. URL: https://www.vscentrum.be/infrastructure/hardware/hardware-kul.

[8] R. C. Huang and L. Anand. Non-linear mechanical behavior of the elastomer polydimethylsiloxane (PDMS) used in the manufacture of microfluidic devices. *Innovation in Manufacturing Systems and Technology*, 2005.

[9] K. Jordens. Modelleren en karakteriseren van flexibele hydraulische microactuatoren, 2011.

[10] S. Kim, C. Laschi, and B. Trimmer. Soft robotics: A bioinspired evolution in robotics. *Trends in Biotechnology*, 31(5):287–294, 2013.

[11] T. K. Kim, J. K. Kim, and O. C. Jeong. Measurement of nonlinear mechanical properties of PDMS elastomer. *Microelectronic Engineering*, 88(8):1982–1985, 2011.

[12] A. D. Marchese, C. D. Onal, and D. Rus. Autonomous Soft Robotic Fish Capable of Escape Maneuvers Using Fluidic Elastomer Actuators. *Soft Robotics*, 1(1):75–87, 2014.

[13] O. Maron and A. W. Moore. The Racing Algorithm: Model Selection for Lazy Learners. *Artificial Intelligence Review*, 11(1-5):193–225, 1997.

[14] M. Mashayekhi. Element Selection Criteria. [Online]. Aug. 2015. URL: http://mashayekhi.iut.ac.ir/sites/mashayekhi.iut.ac.ir/files/u32/presentation10.pdf.

[15] C. D. Onal and D. Rus. Autonomous undulatory serpentine locomotion utilizing body dynamics of a fluidic soft robot. *Bioinspiration & biomimetics*, 8(2):026003, 2013.

[16] A. Postula. Genetic engineering versus natural evolution Genetic algorithms with deterministic operators. *Journal of Systems Architecture*, 48:99–112, 2002.

[17] A. K. Rai, A. Saxena, and N. D. Mankame. Synthesis of Path Generating Compliant Mechanisms Using Initially Curved Frame Elements. *Journal of Mechanical Design*, 129(October 2007):1056, 2007.

[18] A. Saxena. Synthesis of Compliant Mechanisms for Path Generation using Genetic Algorithm. *Journal of Mechanical Design*, 127(May 2010):745, 2005.

[19] D. Sharma, K. Deb, and N. Kishore. Developing multiple topologies of path generating compliant mechanism (PGCM) using evolutionary optimization. (2009002), 2009.

[20] D. Sharma, K. Deb, and N. N. Kishore. A domain-specific crossover and a helper objective for generating minimum weight compliant mechanisms. *Proceedings of the 10th annual conference on Genetic and evolutionary computation*, page 1723, 2008.

[21] D. Sharma, K. Deb, and N. N. Kishore. Towards generating diverse topologies of path tracing compliant mechanisms using a local search based multi-objective genetic algorithm procedure. *2008 IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence)*, pages 2004–2011, 2008.

[22] D. Sharma, K. Deb, and N. N. Kishore. Advancement of Path Generating Compliant Mechanisms (PGCM) Topologies by Initial Population Strategy of Customized Evolutionary Algorithm. (2009003):1–14, 2009.

[23] A. Shukla, R. Tiwari, and R. Kala. Towards hybrid and adaptive computing. *Sci*, pages 59–82, 2010.

[24] M. Srinivas and L. M. Patnaik. Adaptive probabilities of crossover and mutation in genetic algorithms. *IEEE Transactions on Systems, Man and Cybernetics*, 24(4):656–667, 1994.

[25] M. T. Tolley, R. F. Shepherd, B. Mosadegh, K. C. Galloway, M. Wehner, M. Karpelson, R. J. Wood, and G. M. Whitesides. Original article. 1(3):213–223, 2014.

[26] B. Trimmer. Soft robots. *Current Biology*, 23(15):R639–R641, 2013.

[27] D. Trivedi, C. D. Rahn, W. M. Kier, and I. D. Walker. Soft robotics: Biological inspiration, state of the art, and future research. *Applied Bionics and Biomechanics*, 5(3):99–117, 2008.

[28] UFlorida. Plane strain elements, 2010.

[29] UMICH BME456. Fitting elastic model constants. [Online]. Dec. 2014. URL: http://www.umich.edu/ bme456/ch6fitelasticmodelconstant/bme456fitmodel.htm.

[30] E. Zitzler, M. Laumanns, and L. Thiele. SPEA2: Improving the Strength Pareto Evolutionary Algorithm. *Evolutionary Methods for Design Optimization and Control with Applications to Industrial Problems*, pages 95–100, 2001.

Fiche masterproef

Student: Manu De Block

Titel: Evolutionaire optimalisatie van soft robots

Engelse titel: Evolutionary Optimisation of Soft Robots

UDC: 621.0

Korte inhoud:

In dit onderzoek werd de realiseerbaarheid nagegaan van een optimalisatieprogramma voor het automatiseren van het ontwerp van pneumatische actuatoren voor soft robots. Het 3D probleem wordt niet behandeld, maar vereenvoudigd tot het 2D probleem van de topologische optimalisatie van de doorsnede van een actuator die zich in een *plane strain* toestand bevindt. Met behulp van een Python-implementatie van NSGA-II als genetisch algoritme, wordt de optimalisatie uitgevoerd voor twee actieve objectieven; een minimale massa en een minimale afwijking op de gewenste verplaatsing van een eindpunt. De vervorming door een aangelegde inwendige druk wordt gesimuleerd met behulp van eindige elementen berekeningen in Abaqus. Na een parameteroptimalisatie van het genetisch algoritme voor dit specifieke probleem en het toevoegen van mechanismen die de performantie nog verder verbeteren, werd de gerealiseerde code op enkele testproblemen toegepast. Het optimalisatieprogramma bleek in staat topologieën te ontwerpen met de gewenste verplaatsing en aan een zeer goede convergentiesnelheid. Na enkele tientallen generaties met een populatie van 100 te doorlopen, kan voor kleine topologieën een reeks Pareto-optimale oplossingen bekomen worden met toenemende nauwkeurigheid voor toenemende massa's. Besluitend kan worden gesteld dat dit onderzoek een *proof of concept* is voor het evolutionair ontwerp van pneumatische actuatoren voor soft robots.

Thesis voorgedragen tot het behalen van de graad van Master of Science in de ingenieurswetenschappen: werktuigkunde

Promotor: Prof. dr. ir. Dominiek Reynaerts

Assessoren: Prof. dr. ir. E. Ferraris
Ir. N. Famaey

Begeleider: Ir. B. Gorissen