A Practical Approach to Evolutionary Algorithm based Automated Mechanism Design

Research to the usability of automated mechanism design

E. Kranendonk

Master of Science Thesis

TU Delft Delft University of Technology

A Practical Approach to Evolutionary Algorithm based Automated Mechanism Design

Research to the usability of automated mechanism design

Master of Science Thesis

For the degree of Master of Science in Mechanical Engineering at Delft University of Technology

E. Kranendonk

August 22, 2016

Faculty of Mechanical, Maritime and Materials Engineering (3mE) · Delft University of Technology

Delft University of Technology
Department of
BioMechanical Engineering, BioRobotics

Dated: August 22, 2016

Supervisor(s):

Ir. W.J. Wolfslag

Prof.dr. M. Wisse

Reader(s):

Dr. J. Zhang

Dr.ir.M. Langelaar

Abstract

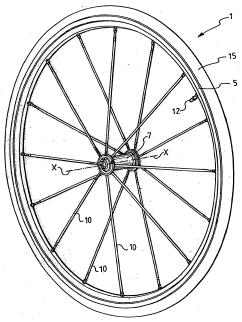
The complexity of current mechanisms continues to increase and their users keep on demanding even more. Current design methods are not sufficient to keep on fulfilling those requirements. Automated design methods have shown to be a viable solution to keep up with the demands and research has started to make this possible for mechanisms design. But there is a big difference in the predicted performance before and the performance after manufacturing. To minimize the effects of production errors while minimizing its computational costs we introduced a novel robust optimization method. In the genetic programming algorithm we add uniform noise to the production dimensions. Every generation the fitness of a mechanism is calculated with this noise and that fitness is added to a memory bank of that mechanism until that memory is full. The average of the fitness values in the memory bank is used as the fitness value to rank the mechanisms of a population. By changing the maximum bounds of the noise and the maximum amount of values possible in the memory-bank we can influence the robustness of the final mechanisms and computational costs of the algorithm. The optimal settings to get the most robust design with the least computation cost was by keeping the noise between $\pm$ 2 times the ISO-2768-m norm and limiting the maximum amount of fitness calculations in the memory-bank to 8. Further decrease of the computational cost was achieved by directly calculating the incidence matrix from the incidence string instead of the iterative method. We also improved the accuracy of the predicted performance by introducing a better weight and moment of inertia calculation based on more accurate descriptions of the shape of the mechanism. The combination of these adjustments resulted in automated mechanism design method that generates robust mechanisms for a wide range of problems within an acceptable time span.

Table of Contents

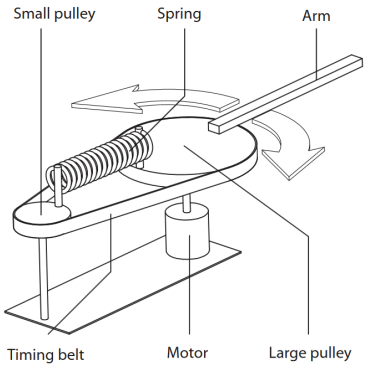
Chapter 1

Introduction

Most people cannot think of a world without the convenience brought by current mechanisms. Simple designs, such as the wheel 1-1a, were once big improvements on the state of the technology and are now taken for granted. We keep on demanding mechanisms with better performance, even for technology most of us will never see, as for example with robotics such as in Figure 1-1b. To keep up with these demands we keep on improving mechanisms and increase their complexity. But with this increase, designing becomes more and more difficult.



(a) A bicycle wheel[1], an example of a mechanism taken for granted but essential in our current society.



(b) A robotic arm with reduced energy consumption by Plooij[1], an example of a mechanism that tries to meet the higher requirements for mechanisms in the future.

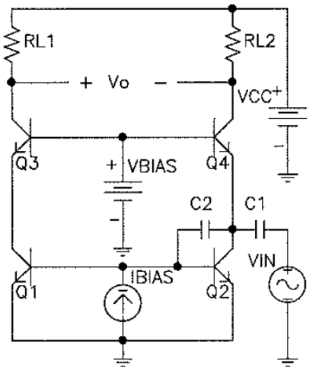
Figure 1-1: A couple of mechanism designs from different centuries.

To aid us in the design process we created tools such as geometry[3][4], technical drawings[5][6] and Computer Aided Design tools[7]. However, at the center of the process we still rely on the human, which introduces uncertainties such as biases through emotions or communication problems. By automating the design process we can reduce these problems.

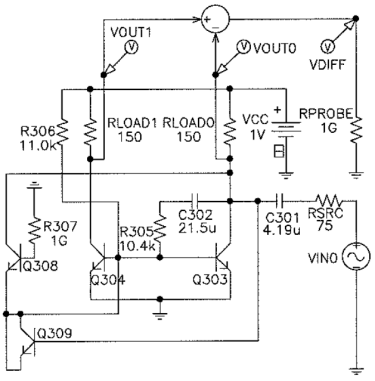
In the field of electric circuit design, this automation step that removes the human, has already started. For example, the evolved balun circuit of Koza in Figure 1-2b from 2004, outperforms Lee's balun circuit patented in 2001 shown in Figure 1-2a with a factor four [8]. Yet these algorithms still need to be adapted to work for mechanical systems.

[1]Meggiolan, Mario. "rim for a bicycle wheel and bicycle wheel comprising such a rim, figure 1" 2011, Espacenet[2]

According to Fireston[9], electronics and mechanics are comparable and the switch to automated mechanical system design should be possible. Several researchers such as Sims[10], Leger[11] and more recently Staal[12] and Kuppens[13] have been working to prepare these tools for mechanism design. But as Leger[11] and Hollinger[14] point out, the designs created with such algorithms will perform differently in the real world than predicted by the automated design algorithms. To make sure these designs can be practically implemented we need to reduce this reality gap.



(a) Balun circuit patented by Lee in 2001.



(b) Balun circuit designed by evolutionary algorithm by Koza in 2004.

Figure 1-2: Example of an evolved balun circuit which outperforms the patented balun circuit by a factor four[8].

Brodback[15] has tried to bridge this reality gap by pulling the whole performance evaluation to the real world. But this reduces the speed advantage current physics simulation tools offer[16], increasing the optimization time significantly and therefore making this a solution that is not usable in practice.

Boeing [17] proposed a method of reducing the reality gap of physical simulations by solving the mechanisms in different simulators. This method relies on the simulation results of several accurate and realistic dynamical engines. As such simulations take time this will also increase the calculation time several folds, making this option also not practical.

A more straightforward way of making sure that the predicted performance is close to the performance in the real world is by using a more accurate physics engine. Even though this will surely reduce the reality gap, it will never completely close it because it does not take into account the manufacturing process itself. As the manufacturing process is never perfect, it will always have an effect on the performance of the mechanism.

Therefore we should incorporate the effect of manufacturing errors in the automated mechanism design algorithm. To investigate these possibilities, we did a literature survey in robust optimization[18]. Applying such an algorithm will make sure that the performance of a manufactured design is as close as possible to the prediction of the automated mechanism design algorithm.

Problem statement

Implementing a method that reduces the effects of production errors, results in the following problem statement:

"How can we create an evolutionary automated mechanism design program, that generates designs within a limited time period, whose predicted performance is little influenced by manufacturing errors?"

With a limited time period we mean that the system should run in the same time-scale as the system on which we build our algorithm. This will be the algorithm of Kuppens [13], which took two to six hours to design a mechanism. If the system would take much longer the automated mechanism design would not be a practical replacement for human design.

With little influence of manufacturing errors we mean that the effect of these errors on the predicted performance of a mechanism from the new algorithm should be smaller than its effects on a mechanism from the original algorithm of Kuppens[13]. How the effect of the manufacturing errors will translate to the performance also depends on the dynamics of the mechanism. As we will verify the created algorithm in a simulation environment, we need to make sure that the effect of the manufacturing errors of the prediction is similar as to how these errors effect a real mechanism. Therefore we should make sure that we use an accurate simulation environment.

In Chapter 2 we will explain the current state of evolutionary based automated mechanism design. A method of reducing the calculation time by improving the incidence string calculation is introduced in Chapter 3. In Chapter 4 we will discuss how we can increase the accuracy of the physics engine. The main topic: reducing the effect of production errors on the performance, is discussed in Chapter 5. These improvements will be tested in Chapter 6 where we will search for the optimal settings for use in the evolutionary algorithm. To see if these adjustments can be used on a wider range of problems we will test it on several design problems in Chapter 7. Finally, we will discuss the results in Chapter 8 and come to a conclusion of this master thesis in Chapter 9.

Chapter 2

Automated mechanism design

In this chapter we will explain how Kuppens[13] implemented automated mechanism design and look into detail in how its evolutionary algorithm functions. Nothing new will be discussed but this way improvements on the algorithm which are made in following chapters will be understood better. According to Kota [19] the automated design method can be divided in two parts: the mathematical representation of the mechanisms and the optimization algorithm. The mechanism representation is described in chapter 2-1 The second part of the automated mechanism design method, the optimization algorithm, will be discussed in chapter 2-2

2-1 Mechanism representation

The mechanism representation creates the blueprints for the possible solutions. It is the translation of the physical world to the mathematical world. As the Oxfords leaner's Dictionary says about a mechanism: "a set of moving parts in a machine that performs a task"[20], we will discuss the individual parts in Chapter 2-1-1, and how they combine to a moving set in Chapter 2-1-2.

2-1-1 Mechanical parts

In this chapter we will present the parts that can be used by the algorithm to create a mechanism. To keep the algorithm simple we will use four elements that will allow us to generate basic mechanisms such as in Figure 2-1. The four used elements are the ground, a mass, a hinge and a spring, as introduced by Kuppens [13].

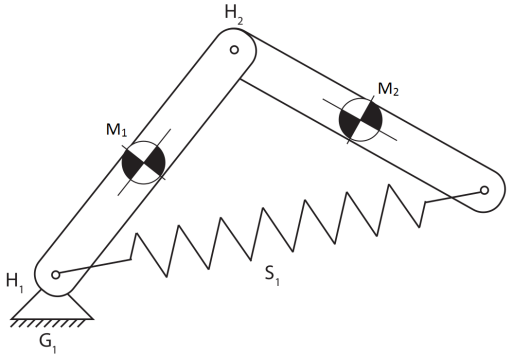


Figure 2-1: Schematic representation of a simple mechanism.

The ground is the representation of the fixed world as G_1 in Figure 2-1. It can be seen as a body with infinite mass and inertia. It will not be a part of the evolution process but will function as a base for connections, so they can be anchored in space.

The mass is the building block for the mechanism and defined as $M_{1\text{-}2}$ in Figure 2-1. To simulate its dynamical response we need its weight, moment of inertia and center of mass. To calculate these we need the shape of the element which is defined by the location of the hinges an springs connected to the mass element.

The hinge connects two bodies together on a fixed point on these bodies with only a rotational freedom as $H_{1\text{-}2}$ in Figure 2-1. The location of the hinge can be optimized over the course of the algorithm.

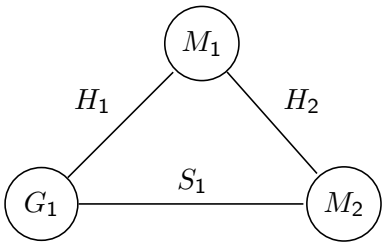
The spring as S_1 in Figure 2-1, is a linear spring as given in Equation 2-1. The spring force (F_{spring}) is defined using the distance between the connection points u, the free length of the spring L_0 and the spring constant k. The free length and the spring constant are parameters that can be optimized alongside the rest of the algorithm to find the most optimal spring. The spring force acts as a force on a point on two bodies limiting their movement.

$$F_{\text{spring}} = k(L_0 - u) \tag{2-1}$$

2-1-2 Mathematical representation

As we now have a collection of parts we need a system that will combine them to function as a mechanism. This combination makes it possible to simulate, calculate and optimize the mechanism so that it can fulfill its task. Ideally this combination should be able to create a mathematical representation of every possible solution that could be designed, while still being easily reconfigurable by the optimization algorithm and being able to simulate the mechanical system it represents.

The method used by Schmidt[21] uses graph theory, which gives total freedom of the internal structure and the components that can be used. In Figure 2-2a we see the graph representation of the simple mechanism of Figure 2-1. This can then be rewritten in a more mathematically easily usable matrix form in Figure 2-2b, also called the incidence matrix. In here G represents the ground (or fixed world), the M a mass, the H a hinge and the S a spring.



(a) Graph representation of a simple mechanism.

	H_1	H_2	S_1
G_1	1	0	1
M_1	1	1	0
M_2	0	1	1

(b) The simple mechanism now described using an incidence matrix.

Figure 2-2: Example of how the simple mechanisms from Figure 2-1 can be described using a graph and an incidence matrix.

A simpler representation, that is more suitable for use in evolutionary algorithms, is the incidence string introduced by Kuppens [13]. He gave every combination of how the ground and mass elements can be connected its own identifier, according to the pattern in Figure 2-3. These values build a string with the same length as the amount of columns in the incidence matrix. To differentiate between hinges and springs a second string is added with labels for identification.

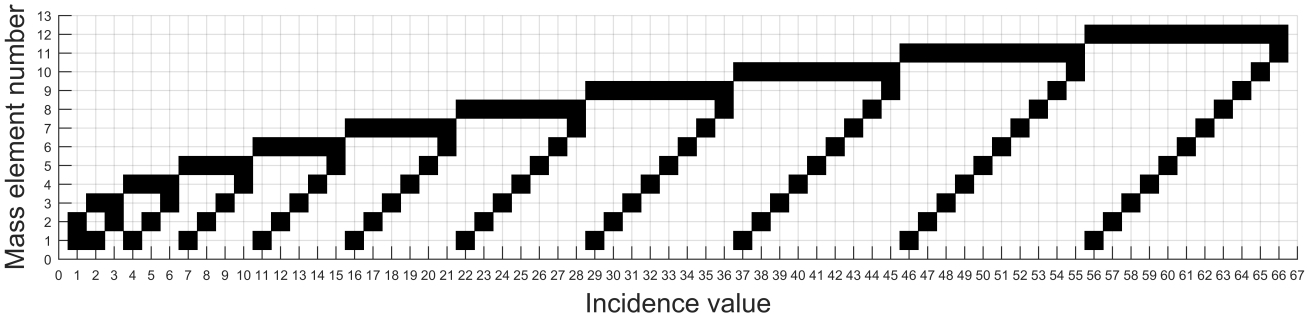


Figure 2-3: Graph of how the incidence value is created from the connections in the incidence matrix.

As an example, the incidence matrix of the simple mechanism from Figure 2-1 transforms to the incidence matrix in the table in Figure 2-2b. The pattern of Figure 2-3 can also be seen in this matrix, as shown in Table 2-1a. The corresponding incidence values create the incidence string in Table 2-1b. The third value in is made red as it represents the spring as can be seen in Table 2-1a and makes it a complete representation of the mechanism of Figure 2-1 into a simple incidence string.

	H_1	H_2	S_1
G_1	1	0	1
M_1	1	1	0
M_2	0	1	1

(a) The simple mechanism described using an incidence matrix.

1	3	2

(b) The simple mechanism described using an incidence string.

Table 2-1: Mathematical representation of the simple mechanism in Figure 2-1.

To get from the incidence matrix to the incidence string values we use Equation 2-2, with I the incidence value and v_1 and v_2 the respective rows in the incidence matrix that are connected, representing which masses are linked, with $v_1 < v_2$. The other way around, to get from the incidence value to the positions in the matrix, Kuppens[13] proposed to calculate all the options that are possible using Equation 2-5 which we know that the system has to obey. This has only one valid solution but to find it, every possible option has to be calculated.

$$I = \sum_{i=1}^{v_2-2} i + v_1 \tag{2-2}$$

$$\text{floor}(\sqrt{2I+3}) \geq v_2 \geq \text{ceil}(\sqrt{2I}) \tag{2-3}$$

$$v_2 = I - \binom{v_1 - 1}{2} \tag{2-4}$$

$$0 < v_1 < v_2 \tag{2-5}$$

Another well known problem with converting a graph to an incidence matrix is isomorphism. This happens when the same mechanisms can be described with multiple different incidence matrices. If in an algorithm it is needed to calculate the fitness of a mechanism (that is represented using an incidence string) it is possible that we are comparing one and the same mechanism, which will be wasted calculation time. It will also decrease the diversity of the algorithm which can lead to other problems. To prevent unnecessary calculation time, it is good to occasionally calculate the system for isomorphism.

2-2 Optimization algorithm

In the previous chapter we showed the mathematical method of representing every mechanism created from combining the ground, masses, hinges and springs. In this chapter we will discuss how we can optimize this mathematical representation of a mechanism to our requirements.

There are many optimization algorithms that can search for the best solution in a particular system. Many of such algorithms take a system that is already fully defined and only optimize its dimensions, this is called parameter optimization. As our representation of a mechanism can take any shape or size and therefore any number of parameters, algorithms that work with parameter optimization will not work for our automated mechanism design method. According to Renner[22] the evolutionary algorithm has proven itself useful in this case and it will also be the choice for our system.

Among Evolutionary Algorithms (EA) we can differentiate four families[23]. Evolutionary Strategy as developed by Rechenberg[24] and Evolutionary Programming developed by Fogel are mainly parameter optimization algorithms and therefore not usable for our application. J. Holland developed Genetic Algorithm which incorporated cross-breeding in which new individuals are created from successful parents. Genetic Programming from Cramer and Koza introduced a tree-structure or graph representation. This last method has the most potential for mechanism design in general[22] and is described in more detail in Chapter 2-2-1.

To differentiate among mechanisms and see which is better, we calculate how well the mechanism is fulfilling the requirements and combine that to a single value: the fitness value. We will use this value as a measure of the performance of a design.

A general problem among optimization algorithms is the danger of getting stuck in a local optimum while this optimum does not fulfill all the requirements. In evolutionary algorithm the danger of getting stuck in a local optimum can be reduced by maintaining a diverse population[25]. When the diversity is low the optimization process is not able to generate a lot of novelty in new mechanisms. To maintain diversity and enlarge the chance of ending in a better optimum or even in the global optimum there are several methods which are discussed in Chapter 2-2-2. As a measure for the diversity we calculate the percentage of non-isomorph mechanisms in a population for every generation, and at the end of the algorithm we take the sum of all those values.

2-2-1 Genetic programming

In this chapter we will explain the working of the evolutionary algorithm: genetic programming. In genetic programming, and evolutionary algorithms in general, not a single mechanism but a population of them is being optimized. The initial population is filled with randomly generated individuals. For every individual of the population its fitness is calculated, after which the population is sorted based on this fitness value. Every generation parents are selected from the population from which a new mechanism, or child, is created. The fitness of this child is calculated and placed back into the population replacing a less fit mechanism. This cycle as seen in figure 2-4 is continued until it is decided to terminate the program and see how the best individual in the final population functions.

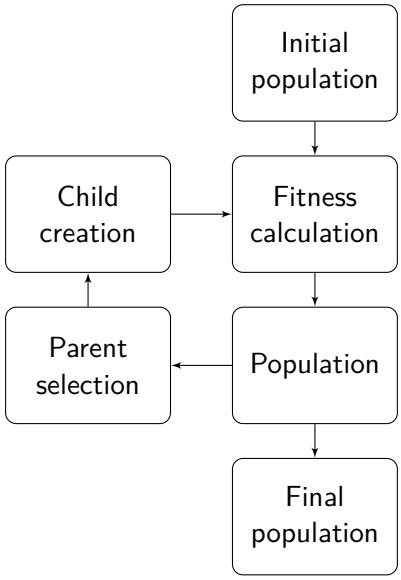


Figure 2-4: Schematic representation of the evolutionary algorithm.

The population is the backbone of the algorithm. It is the collection of all the individual mechanisms created and evaluated by the algorithm. The population knows three phases, the first is the initial population before the optimization is started. The second phase is to hold all the mechanisms while the algorithm is optimizing them and the last population is the resulting list of mechanisms when the algorithm has stopped optimizing.

The initial population is the first state of the population before the optimization algorithm starts. It is filled with mechanisms which are a random arrangement of the mechanical components available. To make sure that the starting diversity is as large as possible every mechanisms in the initial population is unique (not isomorph).

The fitness calculation decides how well the solutions perform compared to the requirements. It calculates the 'fitness' of the individuals in the population. It also sorts the mechanisms in order of fitness for further optimization. The fitness functions used by Kuppens[13] are:

- Degrees of freedom
- Number of components
- Shape of followed path of the components
- Length of the followed path of the components

The parent selection decides which mechanisms are used for making the next generation of new mechanisms. It selects these mechanisms from the sorted population using a Pareto-front, a Roulette wheel, a selection pool or directly using the fitness values.

The child creation uses the previous selected parents to breed a new mechanism. The new mechanism is created using cross-breeding (recombining two survivors to create a new one) and/or mutation. Kuppens [13] used two forms of mutation: removing a random element or changing the dimensions of the mechanism a little. After the new child has been created, its fitness is calculated and it is fed back into the population where it replaces one of the less fit mechanisms.

The final population are the individuals that have survived when the algorithm stops. The algorithm can be set to stop after a fixed amount of generations, when the fitness value stops improving, or when a certain fitness value is achieved.

2-2-2 Maintaining diversity

In this chapter we discuss what we can do to try to reach the global optimum instead of a local optimum. The global optimum is the best choice of all the possible solutions to a problem. A local optimum on the other hand is a solution that is the best choice if we only look to mechanisms similar to it, but if we look further away we can find better solutions. When searching for a solution to a problem the global optimum is preferred above a local solution, but most optimization algorithms can get stuck at a local optima. To decrease the chance of getting stuck in a local optima when using genetic programming, it is best to keep the diversity of the population as high as long as possible[25]. Several methods exist that try to increase the diversity over the generations.

The island model [26] splits the population in several smaller ones. The parent selection is then done by only choosing from this smaller 'island' population and their child will also be fed into this island. After a fixed amount of generations there will be contact between the islands where a selected top few of the mechanisms will be transfered from one island to another, this is called immigration.

By doing so, this method tries to let every population optimize for itself. And if one is stuck in a local optima and the diversity is low it will get new genetic material in the form of a mechanism from the other islands. This will give a small boost to the diversity but more importantly, there is a chance that this new mechanism has new features that will enhance the population and pull it away from the local optimum.

In this model we can define the amount of islands in the algorithm, the amount of generations between immigrations and how many immigrants.

The diffusion model [27] reduces the speed with which some design choices are transferred through the population. When the first parent is selected the second parent is selected in the neighbourhood and their child will also be added within this neighbourhood. This way, if there are several promising designs within a population, they will have more time to develop before mechanisms with higher fitness will force their design options in those mechanisms. We can adjust this model to our preferences by defining the size of this neighbourhood.

The extinction model [28] waits until the diversity is low. When a certain diversity threshold is reached a large amount of the mechanism with the most isomorph sisters within a population is removed. These are replaced by new mechanisms which are generated at random but are not isomorph to any other mechanism in the population. As Kuppens[13] did not find a significant improvement using the method we will not use it in our algorithm.

Chapter 3

Faster incidence string calculation

Kuppens[13] introduced the incidence string as improvement on the incidence matrix. To get from the incidence matrix to the incidence string he used the method in Equation 3-1. In here I is the incidence value and v_1 and v_2 are the rows in the incidence matrix corresponding to which masses are connected, with $v_1 < v_2$. But to get from the incidence string back to the incidence matrix he used the method explained in Chapter 2-1-2. In this chapter we will discuss a faster, analytic solution to the problem, proposed by Kranendonk[18].

$$I = \sum_{i=1}^{v_2-2} i + v_1 = \frac{1}{2}(v_2-2)(v_2-1) + v_1 \tag{3-1}$$

3-1 Incidence matrix calculation

To create an analytic solution to the problem we need to be able to find the inverse of Equation 3-1. We start by splitting the function in two parts: I_{v2} and v_1, as seen in Equation 3-2 with I_{v2} in Equation 3-3. In Figure 3-1 we have visualized v_1 in black, v_2 in red and I_{v2} in green. As we can see, v_1 is not-invertible, hence Kuppens'[13] solution.

$$I = I_{v2} + v_1 \tag{3-2}$$
$$I_{v2} = \frac{1}{2}(v_2-2)(v_2-1) \tag{3-3}$$

In Figure 3-1 we can see that, if we assume v_2 on a continues scale, we can calculate the incidence value directly from v_2 using Equation3-4, with $f(v_2)$ the continues edition of I and $g(v_2)$ the continues edition of I_{v2}. From this function we can calculate the inverse, after which we convert it back to an integer by rounding it to minus infinite resulting in Equation 3-5.

$$f(v_2) = g(v_2) + 1 \tag{3-4}$$
$$v_2 = \text{floor}(g^{-1}(f(v_2) - 1)) \tag{3-5}$$

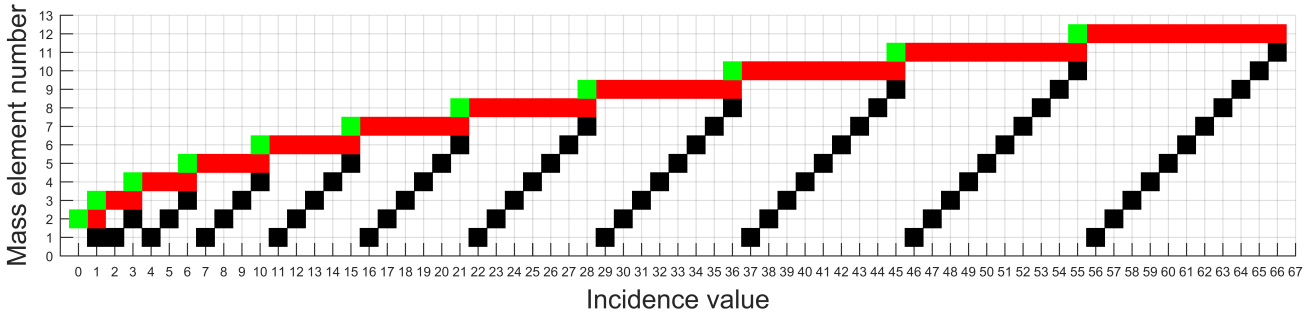


Figure 3-1: Graph of how the incidence values are created from the locations of v_1 and v_2 in the incidence matrix, with v_1 in black, v_2 in red and I_{v2} in green.

To solve $g^{-1}(v_2)$, knowing Equation 3-3, we rewrite Equation 3-6 to Equation 3-7 which can be solved using the ABC-formula to Equation 3-8. We know that the solutions for v_2 can only be positive therefore we have only to look to the positive side of this function. From this we get to Equation 3-9 using Equation 3-5.

$$g(v_2) = \frac{1}{2}(v_2 - 2)(v_2 - 1) \tag{3-6}$$

$$0 = -v_2^2 + 3v_2 + 2g(v_2) - 2 \tag{3-7}$$

$$g^{-1}(v_2) = 1\frac{1}{2} \pm \frac{1}{2}\sqrt{1 + 8g(v_2)} \tag{3-8}$$

$$v_2 = \text{floor}(1\frac{1}{2} + \frac{1}{2}\sqrt{1 + 8(f(v_2) - 1)}) \tag{3-9}$$

Knowing that in our algorithm f will always be the integer I we can calculate v_2 using Equation 3-10. Using this in Equation 3-1 we can calculate v_1 using Equation 3-11.

$$v_2 = \text{floor}(1\frac{1}{2} + \frac{1}{2}\sqrt{1 + 8(I - 1)}) \tag{3-10}$$

$$v_1 = I - \frac{1}{2}(v_2 - 2)(v_2 - 1) \tag{3-11}$$

3-2 Conclusion

Kuppens'[13] method had to estimate v_1 and v_2 three times after which the valid combination could be picked. With only the two Equations 3-10 and 3-11 we have an analytic solution that will find v_1 and v_2 always in one calculation step. This will make the algorithm faster and reduce the amount of functions needed in the automated mechanism design program.

Chapter 4

Accuracy of the physics engine

The simulation method of Kuppens[13] does make a couple of assumption that reduces its accuracy. The weight of the mass elements is chosen at random which can lead to very unrealistic values. After this he defines the mass moment of inertia of the mass element as $I_{\text{mass}} = \frac{1}{2} m_{\text{mass}}$. Which is unrealistic and carries along the problems of the weight of the mass element. Kranendonk[18] proposed a method based on the shape of the mass element which is explained in Chapter 4-1. With this we can create a better weight estimation in Chapter 4-2. And in Chapter 4-3 we use this knowledge to have a more accurate calculation of the moment of inertia.

4-1 Shape of the mass element

To increase the accuracy of the calculation of the weight and moment of inertia of the mass element, we need an accurate estimation of its surface area which we will discusse in this chapter. In our algorithm the shape of the mass element is defined by the connections that are linked to it, as can be seen in Figure 4-2a. The area this polygon spans is a good basis to calculate the surface area, but it has a problem. As every mass element needs to be connected to an other mass element or to the ground, each mass element will have at least one connection. If we want to manufacture the design, it needs space on the surface of the mass element to mount it. Using the method in Figure 4-2a this space is not available at the locations of the connections. If we enlarge the original polygon with half times the radius needed for the hinge connection, we solve this problem.

To enlarge the polygon we first sort the points on the angle as seen from the origin of the mass elements. This way we ensure that there will be no line crossing, creating a free floating surface area as seen in figure 4-1. After that, we check for every corner if it has an angle higher or lower than 180 degrees. At the angles smaller than 180 degrees as seen at point H_3 in Figure 4-2b we place two extra points on the outside. Both perpendicular to each of the connecting lines and through the original point exactly half the hinge radius from the incoming line, as can be seen at point H_3 in Figure 4-2b. At the angle larger than 180 degrees we place one point further to the outside, so that this new point is at half the hinge radius distance from previous lines. This can be seen in point H_4 in Figure 4-2b. This way the lines enlarging the polygon are parallel to its original lines.

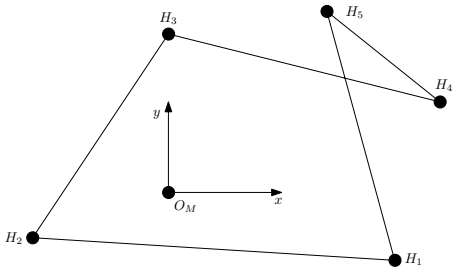
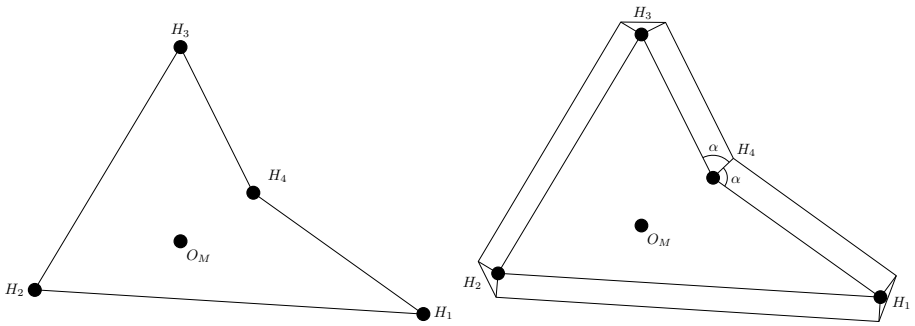


Figure 4-1: When the points of the polygon are not sorted, two lines of the polygon can cross creating a free floating surface.



(a) Polygon representation of a mass element.

(b) Polygon representation of a mass element, enlarged with half times the radius needed to install the hinges and springs.

Figure 4-2: A normal polygon representation of the mass element and how we can enlarge it to create surface for the hinge and spring connection to be installed.

4-2 Weight of the mass elements

Now we have a surface area that is practical after manufacturing, we can use it to give an accurate estimate of the weight. In Kuppens'[13] algorithm, the weight of a mass element was a parameter that was allowed to freely mutate and change over the course of the algorithm. This can result in extremely heavy or light elements which are not realistic to manufacture. We will now define the weight of the mass element as a function of the surface area, the thickness and the density of the mass element. As our algorithm works in two dimensions, it is possible to manufacture the resulting designs of sheets with known thickness and a desired material with uniform density. By predefining this thickness and density at the start of the algorithm we can use Equation 4-1 and 4-2 to calculate the weight of every mass element. In here m_{mass} is the weight, A the surface area, $t_{\text{thickness}}$ the thickness and ρ the density of the mass element. The dimension of the mass element if defined by x_i and y_i with n the amount of corners of the polygon.

$$m_{\text{mass}} = At_{\text{thickness}}\rho \tag{4-1}$$

$$A = \frac{1}{2} \mid \sum_{i=1}^{n} (x_{i-1}y_i - x_i y_{i-1}) \mid \tag{4-2}$$

4-3 Mass moment of inertia of the mass elements

With the shape of the mass element and its weight we can now also give an accurate estimation of its mass moment of inertia. In Kuppens' algorithm the moment of inertia was defined with $I_{\text{mass}} = \frac{1}{2} m_{\text{mass}}$. By using Green's theorem[29] in Equation 4-3, we can get a more accurate value, which will result in a more accurate dynamical response during simulations. In Equation 4-3 I_{mass} is the moment of inertia of the mass perpendicular to the plane and through the center of mass. The i^{th} point on the polygon is given by x and y with n the last point on the polygon and m_{mass} the weight of the mass. The points on the polygon have to be listed in anti-clockwise order.

$$I_{\text{mass}} = \frac{m_{\text{mass}}}{6} \frac{\sum_{i=1}^{n} (x_{i-1}^2 + y_{i-1}^2 + x_{i-1}x_i + y_{i-1}y_i + x_i^2 + y_i^2)(x_{i-1}y_i - x_i y_{i-1})}{\sum_{i=1}^{n} (x_{i-1}y_i - x_i y_{i-1})} \tag{4-3}$$

4-4 Discussion

In the estimation of the surface area in Chapter 4-1 we did not include the complete area needed to mount the connection elements at the sharp angles. This was done to keep the simulation time small and it was assumed that it would not have a large influence on the accuracy of the simulation results. It should be investigated if this assumption is true, and also if there are better, computational efficient ways for solving this problem.

A second point is, that with this improved weight and moment of inertia calculation we still are missing a large part. The weight of the hinges themselves is not included in the calculation. Also the weight of system connecting the hinge or bearings to the mass element will introduce errors in the simulation of the mechanisms. We suggest that further research should be done to get a more accurate simulation, possible in combination with shape optimization as proposed by Kranendonk[18].

With this new calculation method we now have to define the thickness and density of the material that is going to be used for production of the design. This will give the manufacturer the freedom of choice to use the material that is available at the moment, cheap, easy to handle or just aesthetically pleasing. A downside is that optimizing the weight of the mass elements can result in a more favorable dynamical response. Letting the thickness of the mass element freely mutate within a set of bounds, limiting it to realistic values can be an improvement. Another option can be by letting the algorithm select materials from a database with corresponding densities.

4-5 Conclusion

This new method of calculating the weight and mass moment of inertia of the mass elements gives more realistic results as it is now based on its actual dimensions. Independent of how realistic the weight of the mass element is, the current moment of inertia calculation is more accurate which will result in more accurate simulation results.

Chapter 5

Production tolerance influence

In this chapter we will explain how we can reduce the effect of production errors on the performance. These production errors are an effect of the used manufacturing process and will alter the dimensions of the design a little. These alterations will have an influence on the performance of the design. Some designs are more effected by these alterations than others and the designs whose performance is only a little influenced are called robust. With robust optimization we can try to optimize to a design that is robust to these production errors. However, there is always a trade-off between robustness and performance[30], a good balance has to be found. As a measure of performance of a design, we use the fitness value resulting from our evolutionary algorithm.

Another way to cope with production tolerances when you have a design that is not robust, is by choosing a more accurate and precise production method. This will increase the manufacturing cost while it is possible that with another design this was not necessary. Besides that, it is difficult to remove errors that are caused by assembling, wear, creep, etc. A more robust design would also be less influenced by those errors.

In chapter 5-1 we will discuss the robust optimization algorithm. To be able to compare the different settings of the method we need to create a measure of the robustness, this will be discussed in Chapter 5-2.

5-1 Robust optimization

In this chapter we discuss the robust optimization algorithms proposed by Kranendonk [31]. First we will explain some of the robust optimization methods used today in Chapter 5-1-1. After that we will explain in Chapter 5-1-2 how we can combine these methods to get a robust optimization method that functions good in genetic programming. How this method can be used to influence the balance between robustness and computational cost is handled in Chapter 5-1-3. And how we can influence the robustness independent of the computational cost is discussed in Chapter 5-1-4.

5-1-1 Current methods

One of the first to investigate the robustness of a mechanical design was Tagushi[32]. To define the robustness of a mechanism he calculated the fitness of a mechanism while adding noise to every design parameter one by one. From this he calculated the signal-to-noise ratio which he than used as measure of robustness. This method has to recalculate the dynamical response of a mechanism many times per design parameter to see the effect of the noise. This will increase the computational cost of complicated designs with at least a hundred fold due to the curse of dimensionality.

A more refined method is the expectancy measure as proposed by Tsutsui [33][30]. Instead of adding noise to one design parameter at a time, he adds noise to all the parameters at the same time. He repeats this with different noise settings and calculates its effect on the performance. This decreases the effect of the curse of dimensionality and reduces computational costs but also reduces the accuracy of the robustness measure.

However, in evolutionary algorithms, hundreds to tens of thousands of mechanisms need to be simulated over the course of the algorithm to calculate their fitness. By using a robust optimization method that significantly increases the calculation time we dramatically increase the time needed for the evolutionary algorithm. A method specialized for use in evolutionary algorithms to minimize the computational cost was created by Jin [34]. He uses the fitness values of the mechanisms in the population that are already calculated to guess the local robustness of a mechanism. This method only works when the mechanisms have the same construction which limits its use to parameter optimization algorithms. In our genetic programming algorithm we work with graphs whose behavior can change completely even with small changes in the construction of the graph. Therefore this algorithm cannot be used for our algorithm.

Beyer[35] and Thompson[36][37] observed that evolutionary algorithms already have a drive to robust designs. Beyer suggested that this is because if a certain design is robust, designs similar to it will also have a high fitness. Therefore they also have a high chance of being chosen as parent. Ideally we should exploit this and combine it with a method that let us influence the robustness of the final design directly.

5-1-2 Method for genetic programming

As we just explained current robust optimization methods cannot be used in genetic programming or have a high computational cost and are therefore not practical. Now we will explain how we can combine those methods for specific use in genetic programming. Through this combination we can directly influence the calculation time and robustness which is discussed in Chapters 5-1-3 and 5-1-4.

The method we propose works by giving every mechanism in the evolutionary algorithm a memory-bank. The production dimensions then get a light disturbance by adding noise to them before the fitness of the mechanism is calculated. Every generation the fitness of a mechanism is calculated and added to its memory bank as shown in the right loop in Figure5-1. Comparison of the mechanisms for parent selection is done by taking the average of the values that are present in the

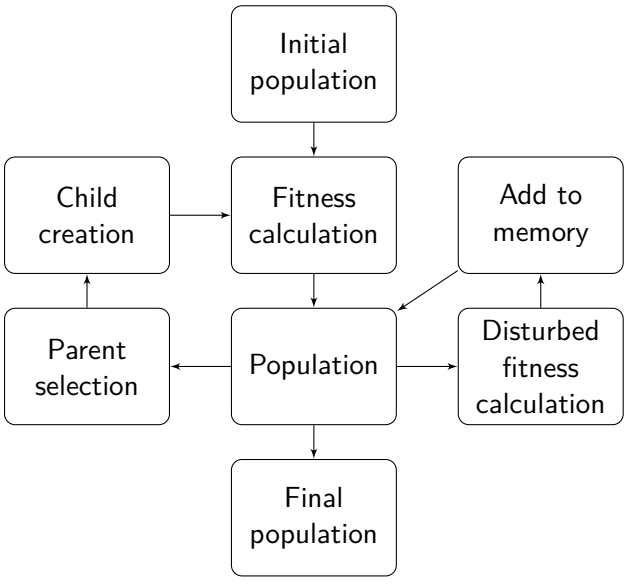


Figure 5-1: Schematic representation of the evolutionary algorithm with robust optimization.

memory-bank. Now if a mechanism has a couple of very bad fitness values it has a high chance of being removed from the population before its memory-bank is full. This will reduce the computational cost of the algorithm.

Adding noise to the production dimensions introduces a chance that the first fitness value is bad while it could be possible that the mechanism as a whole is a good choice. If this mechanism, with only one bad fitness value in its memory-bank, is compared to the rest of the population whose memory-banks are already filled, it will have a lower chance of surviving that generation while it is not sure if it is a worse choice than the others. To reduce this chance of skewed comparison, we let the first value in the memory bank be the fitness value of the not-disturbed production dimensions.

This robust optimization method has two variables that can be altered to influence its functionality. First we can alter the size of the memory-bank explained in Chapter 5-1-3. Secondly we can alter the noise that is added to the production dimensions as discussed in Chapter 5-1-4

5-1-3 Fitness memory

The first parameter that we can alter to influence the robust optimization algorithm is the size of the memory-bank. By increasing its size we will increase the computational cost of the algorithm. This increase in computational cost will have a limit. This is because we can increase the memory-bank size but the chance that a mechanism survives for a long time in the population is small. The maximum computational cost is when every generation the fitness of every mechanism in the population is calculated.

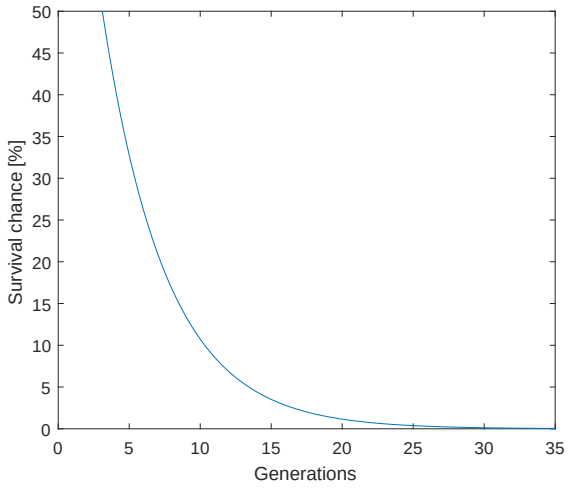


Figure 5-2: Survival chance of a mechanism in an algorithm with 20% replacement per generation and equal chance of being replaced every generation.

The positive side of a bigger memory-bank is that the average of those fitness values gives a more accurate measure for the robustness of the mechanism. The root mean square error compared to the real average decreases with $n^{-\frac{1}{2}}$, with n the amount of values in the memory[38]. With a more accurate robust fitness value the final mechanism will probably be more robust against production errors. But there is a limit to the effectiveness of a large memory size. If two mechanisms are compared and one has a large completely filled memory bank and the other only a hand full of values, the accuracy of the less accurate mechanism will have the most influence. With the a renewal rate of 20% as proposed by Kuppens[13] we can plot the survival of a mechanism over several generations in Figure 5-2. Thus with a very large memory-bank size the effect of a higher accuracy is nearly gone. This means that by choosing the correct memory-bank size we can directly influence the balance between the robustness of the final design and the computational efficiency of the algorithm.

5-1-4 Noise magnitude

The second parameter with which we can influence the robust optimization method, is the shape and size of the noise that is added to the production parameters. This will not increase the amount or difficulty of the calculations needed in the algorithm and therefore it will not influence the computational efficiency.

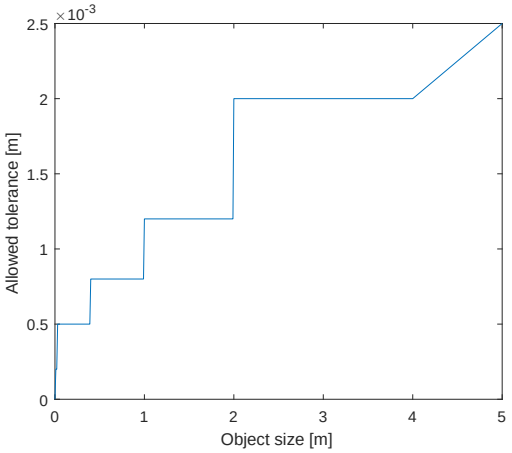


Figure 5-3: Maximum allowed production tolerances according to ISO-2768-m[39].

In the manufacturing process the maximum allowed size of the errors are defined in the ISO-2768[39] norm. We will focus on only the medium tolerance class (ISO-2768-m) as given in Figure 5-3. We will use this as the basis of the maximum allowed disturbance in the robust optimization algorithm. The maximum dimensions of every mass element in a mechanism are determined and used as input to calculate the maximum allowed deviation using Figure 5-3. A new location of every hinge and spring is then chosen uniformly distributed within the maximum allowed tolerances. The origin of the mass elements and the radius of the hinge, which defines the enlargement of the surface area of the element, are also disturbed within these limits.

By multiplying the maximum allowed tolerances from ISO-2768-m we influence the noise added to the production dimensions of the mechanism. If we increase this so called noise magnitude we increase the chances that this has a bigger influence on the fitness of a mechanism. With this larger noise magnitude we thus increase the chance that not-robust mechanisms will show this in the fitness calculations. But if we keep on increasing it, it is to be expected that the average of the fitness values in the memory-bank are a less accurate representation of the real robustness of the mechanism. Also robustness is always a trade-off between performance and robustness of a mechanism[30]. Therefore if we enlarge the noise-magnitude too much we can expect this will result in less fit final mechanisms.

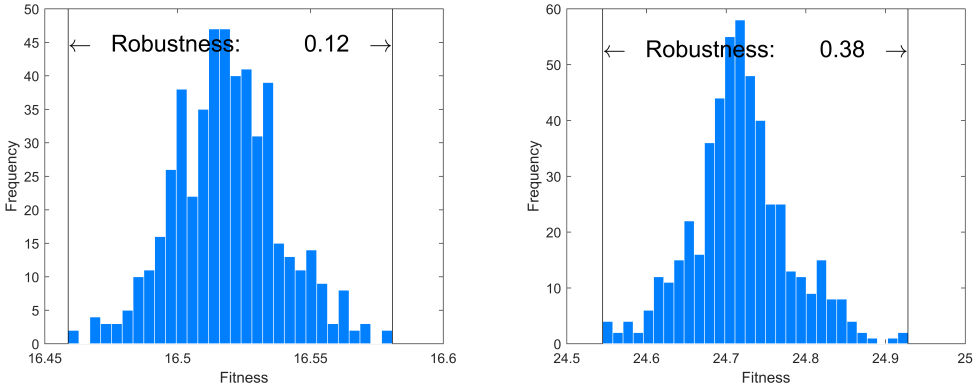
5-2 Robustness measure

In Chapter 5-1 we proposed a method for robust optimization for genetic programming. To verify the effect of this method we need to compare the real robustness of the final mechanisms. As robustness is the effect of variation of the dimensions on the performance, we can define the robustness using the worst-case scenario measure in Equation 5-1. In here R is the robustness measure, (x, y) are the production dimensions of the mechanism and δ is the noise on these dimensions. If true Gaussian noise is used in this function the arms of the robustness distribution would go to infinite when n goes to infinite. But we are only interested in the effect of the production tolerances which are limited to the ISO-2768-m[39] norm as in Figure 5-3. Therefore we can assume that a robust mechanism will have limits on the minimum and maximum fitness values if this noise is added. Also, as every value within the production tolerance is allowed and we assume that they are as likely, we introduce a uniform distribution within the bounds as was done in the robust optimization method.

$$R = \max((F((x, y) + \delta_n))_{n=1}^{\infty}) - \min((F((x, y) + \delta_n))_{n=1}^{\infty}) \tag{5-1}$$

We now have a measure for defining the real robustness of a mechanism. But this requires an infinite series to be calculated and will therefore not be practical. To get a good estimate of the real robustness we will use a Monte Carlo simulation. As the accuracy of the robustness calculated using the Monte Carlo method is defined with $n^{-\frac{1}{2}}$[38] we need minimum of $n = 400$ simulations to get a 95% accuracy. With $n = 500$ simulations we are on the safe side with an accuracy of 95.5%.

In Figure 5-4 we see the histograms of the fitness of two mechanisms using the Monte Carlo simulation. These results are the final mechanisms from two non-robust evolutionary algorithm runs using the settings in Table 6-1 working on the straight line problem explained in Chapter 6-2. As we can see in Figure 5-4 the robustness of the left mechanism is 0.12 and from the right mechanism 0.38. Also, the results from the Monte Carlo method are normally distributed around the fitness value of the not-disturbed mechanism. These are fairly good robustness values in respect to other results using the same settings. In Figure 5-5 and 5-6 we see such mechanisms with with higher robustness measures that resulted from the automated mechanism design method with the same settings.

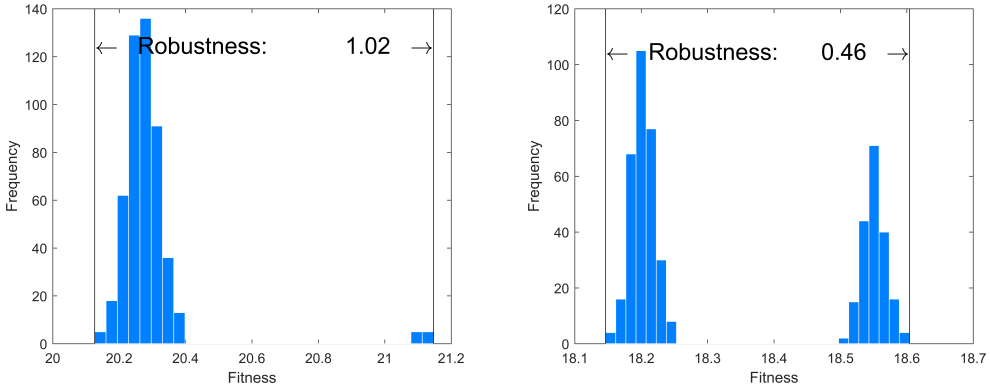


(a) Histogram of mechanism with original fitness value of 16.52.

(b) Histogram of mechanism with original fitness value of 24.71.

Figure 5-4: Histograms of two mechanisms from a not-robust optimized algorithm, each with 500 noisy fitness calculations. These mechanisms are robust as they have little deviation around the original fitness value.

In Figure 5-5a we see a normal distribution around the original fitness value of 20.26. Besides that we see a couple of outliers around a fitness value of 21.15 which gives a big increase of the robustness measure. This happens more often with the mechanism in Figure 5-5b. In this figure we see two similar sized peaks at different fitness values. This probably means that if we build the mechanism with the dimensions within the production tolerances we have two possible setups with each their own performance. But as those performances are pretty similar this will probably not matter.

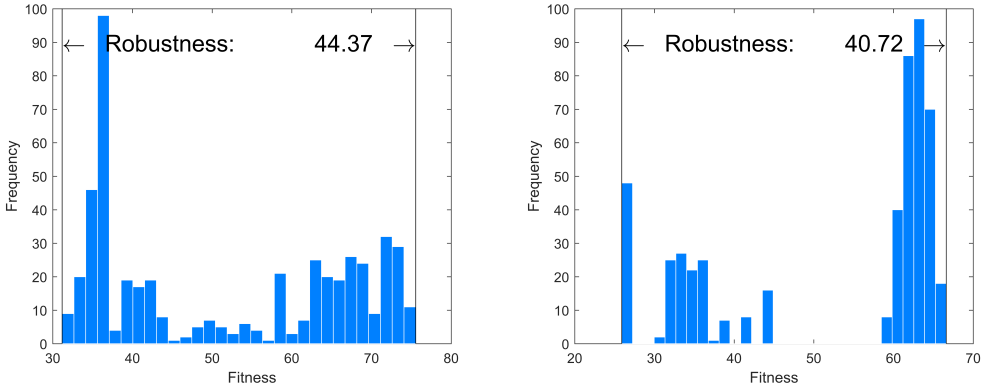


(a) Histogram of mechanism with original fitness value of 20.26 with a couple of outliers away from the original value.

(b) Histogram of mechanism with original fitness value of 18.20 with a second peak around a lower fitness value.

Figure 5-5: Histograms of two mechanisms from a not-robust optimized algorithm, each with 500 noisy fitness calculations. In these histograms we see that the noise introduces a second fitness peak with different performance but still fairly robust.

As we can see in Figure 5-6, the robustness of the previous four mechanisms where still good compared to these. The results from Monte Carlo method of the first four mechanisms were mostly spread around the original fitness value and always quite close. But the fitness resulting from the simulations of the mechanisms in Figure 5-6a and 5-6b start at the original fitness value and only increase from there. The mechanism in Figure 5-6a with a robustness of 44.37 spreads from a value 30 to 75 with random peaks and a large peak around 35.



(a) Histogram of mechanism with original fitness value of 31.27.

(b) Histogram of mechanism with original fitness value of 25.94.

Figure 5-6: Histograms of two mechanisms from a not-robust optimized algorithm, each with 500 noisy fitness calculations. These mechanisms are heavily influenced by the noise with random peaks far from the original fitness value resulting in a not robust and thus unreliable mechanism.

The mechanism in Figure 5-6b with a robustness of 40.72 has more than half of the simulation results around the fitness value of 63 while the fitness value of the original mechanism was at 25.9. This looks like it was designed on a peak in the solution space where most of the valley around it had a fitness value of 63.

5-3 Conclusion

With the robust-optimization method explained in Chapter 5-1 we have a method that will function well with genetic programming. In this method we can set the size of the memory-bank to get the optimal balance between robustness of the final mechanism and the calculation efficiency of the algorithm. To further influence the robustness of the final mechanism we can also change the size of the maximum allowed noise magnitude.

Chapter 6

Verification of robust optimization

In previous chapter we proposed a new way for robust optimization for genetic programming. In this chapter we will try to verify this method and look for its optimal settings of memory size and noise magnitude. This will be done by designing a straight line mechanism further explained in Chapter 6-2 with the evolutionary algorithm settings from Chapter 6-1. The optimal size for the memory-bank is determined in Chapter 6-3. And the ideal noise magnitude is determined in Chapter 6-4. We will test the algorithm under the combination of these settings in Chapter 6-5. As our goal was to create a more computationally efficient method we will try to verify this in Chapter 6-6. Finally we will take a more detailed look at the working of the mechanisms designed by the algorithm in Chapter 6-7.

6-1 Evolutionary algorithm settings

The settings of the evolutionary algorithm will determine its functioning and therefore capability of getting to a good optimum. In this chapter we will choose the settings for the evolutionary algorithm. And at the end we will determine the optimal settings under which we will test the robust optimization method.

Kuppens[13] compared several functions and settings for solving the straight-line problem which we will use for verifying our algorithm. To maintain the diversity of the population he came to the conclusion that the island model with 12 islands gave better results than 2 islands and the use of a diffusion model also significantly improved the results. For this reason we will use the the diffusion model and island model in our algorithm.

For the population we use seven times seven mechanisms for each island. Kuppens proposed the use of twelve islands but he showed that with two islands the algorithm would still function. As our focus is to compare the different settings and not to achieve the best solution, four islands will be the best trade-off between calculation-time and functionality of the algorithm. Kuppens also showed that one migrant per migration wave and 10 generations between the migrations gave good results which we will also use in our algorithm. The diffusion model will use a neighbourhood raster of three by three mechanisms. For the breeding of new mechanisms we also use the same settings as Kuppens: a crossover rate of 1, the mutation rate of 0.05 which defines the chance if the production dimensions are changed and a loss rate of 0.03 if an element of the mechanism is deleted. We will use the same amount of 10 mechanisms that are replaced every generation (20% of the total population). A schematic overview of the settings can be found in Table 6-1. All the experiments will be run 30 times to obtain a fair sample of the performance.

Generations	600
Migrations	60
Islands	4
Migrants	1
Population size	[7,7]
Neigbourhood size	[3,3]
New mechanisms	10
Crossover rate	100 %
Mutation rate	5 %
Loss rate	3 %
Mass element thickness	0.01 [m]
Hinge radius	0.01 [m]
Material density	1180 [kg/m^3]

Table 6-1: Settings for the evolutionary algorithm based on experiments done by Kuppens[13].

We now have the settings of the evolutionary algorithm but we also need the settings for the robust optimization method. The relative error of the estimated mean compared to the real mean of the memory-bank decreases with: $n^{-1/2}$ [38], with n the amount of values in the memory. If we want a accuracy of 95% we need 400 values in the memory-bank. As this is two third of the total amount of generations, the chance that the algorithm will fill up a memory is very small. This will also remove the whole purpose of this method of robust-optimization which was reduced computational costs. The survival chance of a mechanism through one generation is around 80% if we assume that every mechanism has equal chance to be replaced by a new mechanism. The chance of surviving multiple generations is given by Equation 6-1 which is plotted in Figure 5-2. In here g_n is the amount of generations and r_survival is the chance of a mechanism surviving that many generations.

$$r_\text{survival} = 0.8^{g_\text{n}} \tag{6-1}$$

Comparing an accurate calculation of the robustness (meaning many values in the memory-bank) with a inaccurate calculation (just a few values in the memory-bank) will make the inaccurate calculation dominant in the comparison of robustness. Therefore we can expect that that a large memory bank will have little effect on the robustness of the final mechanism while it will have a large effect on the computational efficiency. If we want the comparison to be meaningful we need to calculate the chance that there are still mechanisms in the population with similar filled memory banks. With a survival rate of 80%, using Equation 6-1 we can expect that of the 196 mechanisms (4 islands and 49 mechanisms per island) in the algorithm only one will still be present after 24 generations. This is assuming equal chance of survival of all the mechanisms which is not true as we know that mechanisms with a higher fitness value have a higher survival chance. To see if these high survivors have an effect on the robust optimization we want to measure a memory-bank size higher than 24 values. We can also expect that with small memory-bank sizes, little changes will have a big effect on the robustness of the final mechanism. Therefore it is interesting to do more experiments with small memory sizes. As we want to keep the needed calculation time reasonable we will limit the amount of experiments to six memory-bank sizes. The different settings for the memory sizes to be tested will be: 1, 4, 8, 16, 24 and 32. In here we have no robust optimization when the memory-bank size is 1 as the first fill will be the fitness value of a not-disturbed mechanism. The memory-bank size walks with steps of 8 to a size of 32 giving us the extra large memory size. An extra low size of 4 is also introduced to see the effects in that region more precisely.

The second setting of the robust optimization method we need to test is the noise magnitude. As we want to create the robust solutions to ISO-2768-m tolerances[39], we should test the system if it works with noise of that size, thus a magnitude of 1. The ISO-2768-m norm has a higher relative effect at lower dimensions. Therefore to see the effects of the robust optimization more clearly we need to look to small mechanisms. For this purpose we will give an optimization drive for mechanisms of around 1 meter which we will discuss in more detail in Chapter 6-2. As the final mechanisms will try to follow this requirement, we can expect that the elements which make up the mechanism, are smaller but in the order of magnitude of this 1 meter window. According to the ISO-2768-m norm [39] dimensions between 0.4 and 1 meter are allowed to have a deviation within $\pm$0.8mm of the base dimension. This results in an allowed deviation of 0.16% to 0.4%. To maintain reasonable calculation times for the experiments we also limit ourselves here to 6 settings for the noise magnitude. A linear increase of the noise magnitude from 0 to 5 with steps of 1 gives us a maximum noise magnitude of 5 which means an allowable error of $\pm$4mm. This results in an allowed deviation of 0.8% to 2%. This is the same size as the very coarse tolerance class in the ISO-2768-v norm [39], which is the highest tolerance class in the norm. Therefore noise magnitude settings for the robust optimization will be: 0, 1, 2, 3, 4 and 5 times the ISO-2768-m norm which ranges from no deviation to the maximum norm which gives a good representation of a real mechanism.

Running the algorithm for the six chosen memory sizes and noise magnitudes would mean running 36 experiments. We can reduce this amount to 26 experiments as the a memory size of 1, or noise magnitude of 0, all mean a non-robust evolutionary algorithm. The algorithm will be run on a machine with two E5-2665 cpu's (8 cores, 2.40GHz, L3 20Mb cache) and 32Gb ECC PC3-12800 memory. This means that the system needs 32.5 to 97.5 days of computing if we run the algorithm in parallel (8 cores are available for our use and we need 4 cores per experiment). This is too long given the amount of time available for this project. For that reason we will run all the memory-bank settings with one noise magnitude, and all the noise magnitude settings with one memory-bank size. The fixed noise magnitude for the memory-bank settings will be the center of the noise magnitude settings if we exclude the zero (which is the non-robust optimization). This is a noise magnitude of 3. In Chapter 6-3 we will discuss the experimental results of the different memory-bank sizes. For the fixed memory-bank size for the noise magnitude settings we use the same reasoning which results in a memory-bank size of 16, whose results will be discussed in Chapter 6-4. If it appears that the setting for optimal working combination has not yet been tested we will also experiment with this setting in Chapter 6-5. To see if our proposed robust optimization method has a higher calculation efficiency than the expectancy measure of Tsutsui [33] we will compare this in Chapter 6-6. The 10 different robust optimization experiments, 1 optimal robust optimization experiment and 1 computational cost experiment will result in 15 to 45 days of non-stop calculating on two machines.

6-2 Straight line fitness functions

In this chapter we will discuss the optimization problem with which we will try to verify the robust optimization algorithm. In his work, Kuppens[13] used the straight-line problem to compare design choices for his algorithm. Because our algorithm is based on his, we have many elements in common. This makes the straight-line problem a good choice for optimizing our algorithm as it has already proven itself to be compatible with a similar algorithm. Kuppens used four measures to translate the requirements of a straight line mechanism to mathematical functions. The curvature of the line, the length of the line, the complexity of the mechanism and the degrees of freedom of the mechanism. We will add one extra fitness function to the straight-line problem: the size of the mechanism. At the end of this chapter we will show how we combine these fitness functions to one fitness value for the complete mechanism. The plots of the fitness functions can be seen in Figure B-1 in Appendix B. The mechanisms are simulated for 5 seconds with a step size of 0.05 seconds using a Runge-Kutta 4^{th} order solver.

Line curvature

To find the point on a mechanism that best approximates a straight line, Kuppens'[13] calculated the curvature of the trajectory of every point on the mechanism. He defined the curvature of line by standing on a data point and look to the angle between the two connected lines. Using Equation 6-2 he calculated the measure for the curvature of the complete line κ which, if you want a straight line should be zero. Here t is the n^{th} point on the trajectory. The trajectory of the point on the mechanism with the lowest curvature is than chosen as the input trajectory for further calculations.

$$\kappa(t) = \frac{x'(t)y''(t) - y'(t)x''(t)}{(x'(t)^2 + y'(t)^2)^{2/3}} \tag{6-2}$$

$$r_{\text{correlation}} = \left|\frac{\sum_{t=1}^{n}(x_t - \bar{x})(y_t - \bar{y})}{\sqrt{\sum_{t=1}^{n}(x_t - \bar{x})^2 \sum_{t=1}^{n}(y_t - \bar{y})^2}}\right| \tag{6-3}$$

$$E_{\text{Straight}} = |r_{\text{correlation}} - r_{\text{Desired}}| \tag{6-4}$$

This method has its limits, the moment a line travels back over its own trajectory it will be registered as a 180 degree curvature at the end-point. In normal observation you only count the curvature between the turning points and not the turning itself. We propose using the absolute value of the correlation coefficient of the data points $r_{\text{correlation}}$ in Equation 6-3 as a measure of straightness of the line as done by Stojmenović [40]. In here n is the total amount of points in the trajectory. If the absolute value of the correlation is 1 the line will be perfectly straight while the trajectory could have traveled several times over the same path. This results in the error function for the straightness of a trajectory E_{Straight} in Equation 6-4 with r_{Desired} the desired correlation.

Line length

We have now defined the curvature of a line, but if the trajectory travels no distance the mechanism will not be practical. Therefore we need a method for measuring the length of the trajectory. Kuppens introduced a fitness function that created a measure of the length of the trajectory. This measure calculates the distance between all the data points and sums them using Equation 6-5.

$$L_{\text{trajectory}} = \sum_{t=1}^{n} \sqrt{x'(t)^2 + y'(t)^2} \tag{6-5}$$

$$E_{\text{Length}} = |L_{\text{Trajectory}} - L_{\text{Desired}}| \tag{6-6}$$

This length measure has the same limitation as the curvature of Kuppens discussed before. If it travels back over its own path it will keep on adding this to the total distance. But we want the maximum length of the path, without the extra length if it travels back over itself. By cutting the line in segments wherever the derivatives of the horizontal and vertical trajectory crosses zero, we have the lengths between the change of direction. This does not take in account that not every time the trajectory changes direction it will travel back over exactly the same path. But if it does not travel back exactly the same its straightness will also be low resulting in a low total fitness, making this inaccuracy less important. The error function for the length E_{Length} is given in Equation 6-6 where the desired length L_{Desired} is 1 meter.

Complexity

A low complexity, or in other words few elements, is wanted as it will make the manufactured mechanism less prone to errors and easier to produce. The complexity of a mechanism will be defined by the sum of the amount of mass elements, amount of hinge elements and amount of spring elements. This is the same method as used by Kuppens [13] and described in Equation 6-7. In here C is the measure for complexity, n_M the number of mass elements, n_H the number of hinge elements and n_S the number of spring elements. This results in the error function in Equation 6-8 with E_{Complex} the error and C_{Desired} the desired complexity which is 0 to always have a drive to a less complexity.

$$C = n_M + n_H + n_S \tag{6-7}$$

$$E_{\text{Complex}} = |C - C_{\text{Desired}}| \tag{6-8}$$

Degrees of freedom

With a low degree of freedom we will reduce the possibility that the optimal trajectory is a fluke. If the degree of freedom of the mechanism is 1 we know the trajectory will only be able to follow its prescribed path resulting in a more practical mechanism. A low degree of freedom will also drive the algorithm to design a simpler mechanism. The degrees of freedom (DOF) of a mechanism is calculated by means of the dimension of the nullspace of the Jacobian of the constraint vector D_k as done by Kuppens[13]. In Equation 6-9 we give the error function E_{DOF} of the degrees of freedom with DOF_{Desired} the desired value. Ideally the desired value should be 1, but due to some errors during the experiments we used a DOF_{Desired} of 2. As we are not trying to get the best straight line mechanism but are only interested in the comparison between different settings this will not influence our results.

$$E_{\text{DOF}} = |DOF - DOF_{\text{Desired}}| \tag{6-9}$$

Mechanism size

In Chapter 5-2 we calculate the measure of robustness by adding uniform noise to the production dimensions. This noise is shaped according to the ISO-2768-m norm[39] which has a higher relative effect when the dimensions of the mechanism are smaller. This means that the measure of robustness is better visible when a mechanism is small. Therefore it is preferable that we add an extra requirement that tries to force the dimensions of the mechanism down. This will be done by looking to the distance between the lowest and highest point in the mechanism at the start of the simulation, horizontal and vertical, as seen in Equation 6-10 and 6-11. We combine these in the error function E_{Size} in Equation 6-12. In here the preferred horizontal and vertical dimensions for the mechanism ($L_{\text{DesiredHor}}$ and $L_{\text{DesiredVer}}$) will both be 1 meter.

$$L_{\text{Hor}} = \max(x_i) - \min(x_i) \tag{6-10}$$
$$L_{\text{Ver}} = \max(y_i) - \min(y_i) \tag{6-11}$$
$$E_{\text{Size}} = \frac{1}{2}(|L_{\text{Hor}} - L_{\text{DesiredHor}}| + |L_{\text{Ver}} - L_{\text{DesiredVer}}|) \tag{6-12}$$

Combining the fitness functions

Our algorithm works with one value that represents the fitness of the whole mechanism which is then used in the parent selection. Therefore we need to combine the previously found error functions to one value. By assigning each error function a weight in the weight vector $\bar{w}$ in Equation 6-13 and multiplying it with the vector with the error functions we get this single value F. The weights vector for our straight line problem will be $\bar{w} = [80, 40, 30, 20, 10]^T$.

$$F = [\bar{w}]^T \begin{bmatrix} E_{\text{Straight}} \\ E_{\text{Length}} \\ E_{\text{Complex}} \\ E_{\text{DOF}} \\ E_{\text{Size}} \end{bmatrix} \tag{6-13}$$

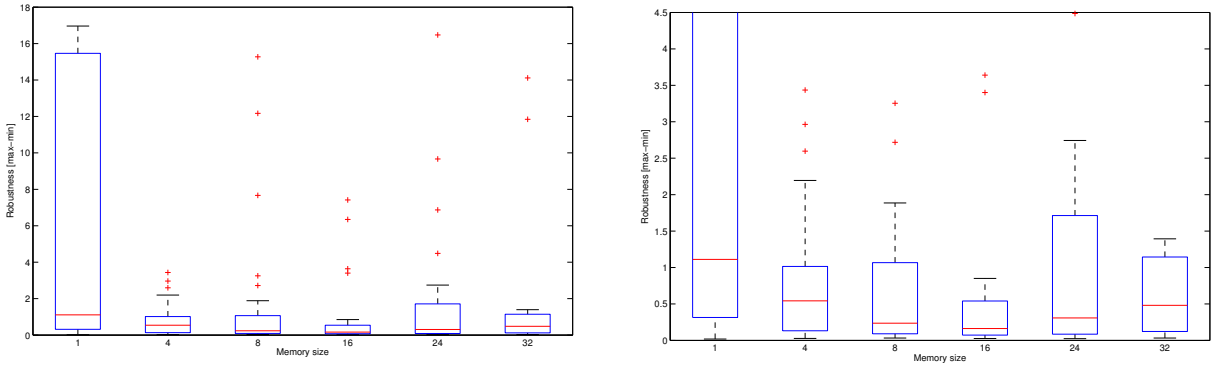
6-3 Optimal memory size

In this chapter we will determine the optimal size of the memory bank for our robust optimization method. The optimal memory size is where it has the best robustness while having the lowest computational cost. If this is a memory size larger than 1 we will also have verified that this method works as a robust optimization method in automated mechanism design.

As we explained in Chapter 6-1 we will test the robust optimization method using the settings in Table 6-1, a noise magnitude of 3 and over the six memory sizes: 1, 4, 8, 16, 24 and 32. In Chapter 6-3-1 we will present the results of the experiments. These results will be discussed in Chapter 6-3-2 and the conclusion is in Chapter 6-3-3

6-3-1 Experimental results

In Figure 6-1a we see the distribution of the robustness of each of the sizes of the memory-bank. To improve the visibility at higher memory sizes, a zoomed in version is given in Figure 6-1b. The robustness at a memory size of 1 has two outliers of more than 9.900. These outliers do not originate from the normal calculation but are caused by an error in the simulation when the fitness was calculated. Possible causes for such errors can be extreme accelerations or if singularities have appeared while solving the differential equations. In both cases the algorithm is ordered to stop the simulation and give these high fitness values. These errors do have real world counterparts. If the accelerations are too large this means some forces are extremely high which will be able to damage the mechanism. And a singularity happens when the mechanisms too gets into a singular position, which can give problems to its real functioning. We should also try to avoid mechanisms that give these errors as they will create problems in a real mechanism. Therefore we cannot simply remove these points from the data as they still hold valid information. For the statistical calculation to work we will replace these values with the max allowed value from the z-scores[41]. This is a value of 3.29 times the standard deviation which is 72.21.

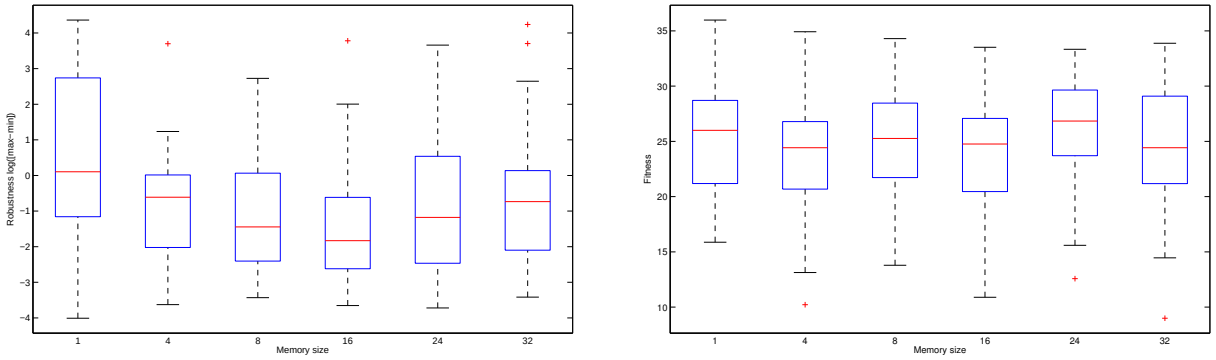


(a) Robustness of the best mechanisms from the experiments with different memory sizes. In here a lower robustness value is better.

(b) Zoomed in robustness of the best mechanisms from the experiments with different memory sizes.

Figure 6-1: Box-plots of the robustness of the six experiments with different sizes memory-bank. We can see here that the larger memories often show better robustness.

Using a Shapiro-Wilks test for normality we concluded that the robustness data from the experiments is not a normal distribution (all memory sizes $p<0.001$). A logarithmic transformation was used which results are shown in Figure 6-2. The transformation did get the data normal or close to normal. A one-way ANOVA test shows that with $F(5,174)=2.83$ $p=0.0174$ there is a significant difference between robustness of the different memory sizes. A Tukey post-hoc test showed that there is significant difference between the means from memory size of 1 compared to 8 ($p=0.0355$) and also between 1 compared to 16 ($p=0.0073$). This is a good sigh as this means that our algorithm works and does create more robust mechanisms. No other differences were significant.



(a) Robustness of the best mechanisms for different sizes memory-banks after data transformation using the logarithmic function.

(b) Fitness of the best mechanisms from the experiments with different sizes memory-banks. We do not see any change in the fitness of these mechanisms.

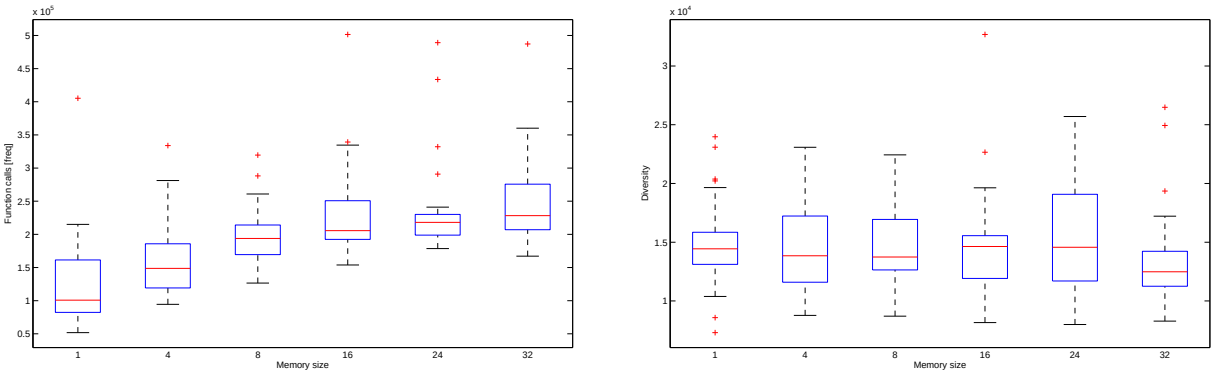
Figure 6-2: Box-plots of the robustness measures and fitness values from the experiments with different sizes memory-banks.

As stated by Beyer[30], to gain a more robust design you often have to sacrifice fitness. To verify this for our algorithm the results of the fitness values are given in Figure 6-3b. An one-way ANOVA test was conducted to examine the effect of memory size on the fitness of the best mechanisms from the robust optimization algorithm. With a $F(5,174)=1.17$ $p=0.3247$ the test did not find any statistical difference between the groups with different memory sizes. This is positive as apparently our algorithm does create more robust mechanisms without reducing their fitness.

With the increase of the memory size we expect that the algorithm has to simulate more mechanisms. This will result in larger calculation times. The algorithm was tested on a system where also other processes were running. As the run-time of the algorithm does not only depend on its own settings but also on the hardware and software of the system and the other processes running on it, we cannot use the logged run-time as a reliable basis for comparison. In the algorithm, the simulation is the most time-consuming part. Therefore we can use the amount of times a simulation is done as a measure of how calculation heavy the algorithm is. In here we have to take into account that a simulation can be stopped before finished if it generates errors. From here on we will call the amount of times the simulation is called in the algorithm 'function calls'. In Figure 6-3a we see the distribution of the amount of function calls per algorithm run for every memory size tested.

The data was tested for normality using a Shapiro-Wilks test which showed that all but one were not normal with p<=0.001. Transforming the data did not result in an approximately normal distribution. Therefore we applied the Kruskal-Wallis H-test which showed a significant difference between the groups with $\chi^2(5)=74.96$ p<0.0001. Dunn's post-hoc test was performed to show which groups differed. Compared to the memory size of 1, the groups with memory size 8, 16, 24 and 32 differ significantly with p<0.001. And compared to the memory size of 4, the groups with memory size 16 (p=0.0066), 24 (p=0.0002) and 32 (p<0.0001) differed significantly. Among the other memory sizes no significant differences were found. This means that compared to the memory sizes of 16 and higher, a memory size of 1 and 4 does decrease the computational cost, while among them no significant difference is visible. However a memory size of 8 is only significantly more heavy in computation than a memory size of 1. We have to take into account that a memory size of 8 and 16 are both more robust than mechanisms created with a memory size of 1 and between them no difference can be seen. This means that mechanisms from an automated mechanisms design method with a memory size of 8 does not create less robust mechanisms than with a memory size of 16, while it shows more favorable computational efficiency. Therefore a memory size of 8 is the best choice for robust optimization according to this data.

In Chapter 2-2-2 we showed that a evolutionary algorithm with a higher diversity is preferred to prevent getting stuck in unwanted local optima. To see if the robust optimization method has any effect on the diversity of the algorithm we plotted that data in Figure 6-3b. An one-way ANOVA test showed that there was no significant difference between the groups with F(5,174)=0.74 p=0.5731. With this we do not have to fear that the robust optimization algorithm does decrease the chances of the algorithm to search for the optimal point.



(a) Distribution of the amount of function calls per algorithm run for each size of memory

(b) Distribution of the diversity over the course of the algorithms for each size of memory

Figure 6-3: Box-plots of the robustness measures and fitness values for different sizes of memory banks.

6-3-2 Discussion

In the data of the function calls in Figure 6-3a we see that the increase in the smaller memory sizes smooths out at higher memory sizes. This was to be expected as we explained in Chapter 5-1-3. There we showed that only a few mechanisms will survive for longer than 24 generations. Which means that a memory size of more than 24 will probably just mean that every generation every mechanism will fill one space in the memory-bank. Therefore we did not expect any increase in computational cost from a memory size of 24.

During the statistical tests we transformed the robustness measure to get normal distributed data. But not all data had a normal distribution, therefore we should keep in mind that this can influence the accuracy of the used test.

6-3-3 Conclusion

We showed that with a memory size of 8 or 16 the algorithm will create significant more robust mechanisms. With this we have shown that our robust optimization algorithm works. Besides the better robustness we did not see any change in diversity or fitness, which is good as this means that our algorithm does not decrease the chance of finding the optimal mechanism or decrease the performance of that mechanism. A downside that we expected was that the algorithm did show an increase in computational costs from a memory size of 8 or higher. As we said earlier we are looking for the highest robustness for the lowest computational cost. Both a memory size of 8 and 16 fulfill this task of better robustness but a memory size of 16 is more computational heavy than a memory size of 4 and 1 while a memory size of 8 is only more computational heavy than a memory size of 1. This makes an automated mechanism design system with a memory size of 8 the best choice.

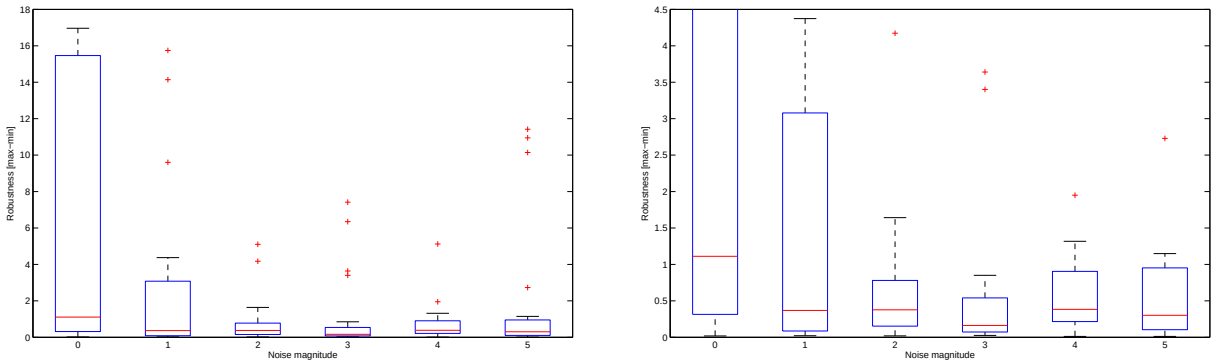
6-4 Optimal noise magnitude

In this chapter we will determine the optimal bounds for the noise on the production dimensions. Here the optimal value for the noise magnitude is also the one with the best robustness while having the lowest computational cost.

As we explained in Chapter 6-1 we will test the robust optimization method using the settings in Table 6-1, a memory size of 16 and for the six noise magnitudes: 0, 1, 2, 3, 4 and 5 times the ISO-2768-m norm [39]. In Chapter 6-4-1 we will present the results of these experiments. They will be discussed in Chapter 6-4-2 and the conclusion is in Chapter 6-4-3

6-4-1 Experimental results

The results of the experiments with different noise magnitudes are given in Figure6-4a. To increase the visibility between the experiments with higher noise magnitudes we included a zoomed in version of the data in Figure 6-4b. In the data with no noise (which is the same as no robust optimization) we again found a couple of outliers higher than 9.900. We replaced them with values 3.29 times the standard deviation as explained in Chapter 6-3-1. We tested the robustness distribution for normality using Shapiro-Wilks and found the data to be not normal ($p<0.001$ for all noise magnitudes). Transforming the data with a logarithmic function made it a normal or approximately normal distribution and is shown in Figure 6-5a. A one-way ANOVA test was done showing that there is significant difference between the groups with $F(5,175)=3.25$ and $p=0.0079$. A Tukey post-hoc test showed significant difference between a noise magnitude of 0 (no robust optimization) compared to a noise magnitude of 2 with $p=0.0216$, a noise magnitude of 3 with $p=0.0049$ and a noise magnitude of 5 with $p=0.0271$.



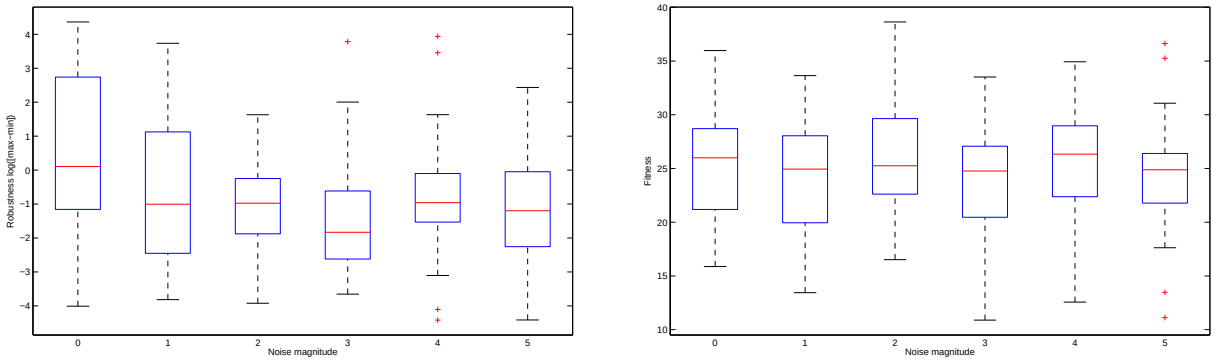
(a) Robustness of the best mechanisms over different noise magnitudes. In here a lower robustness value is better.

(b) Zoomed in robustness of the best mechanisms over different noise magnitudes.

Figure 6-4: Box-plots of the robustness of the six experiments with different noise magnitudes. We see a trend that with higher noise magnitudes the mechanisms become more robust.

As we explained in Chapter 5-1-4 we here also want to know if we need to sacrifice robustness for fitness. The data in Figure 6-5b shows the fitness values of the mechanisms created with different noise magnitudes. Using a one-way ANOVA test we did not find significant difference between the groups with $F(5,174)=1.13$ and $p=0.3477$.

To see if different noise magnitudes influence the diversity during the evolutionary algorithm we plotted the data in Figure 6-6b. Using a one-way ANOVA test we did not find significant difference between the groups with $F(5,174)=0.54$ $p=0.7441$).



(a) Robustness of the best mechanisms for different noise magnitudes after data transformation using the logarithmic function.

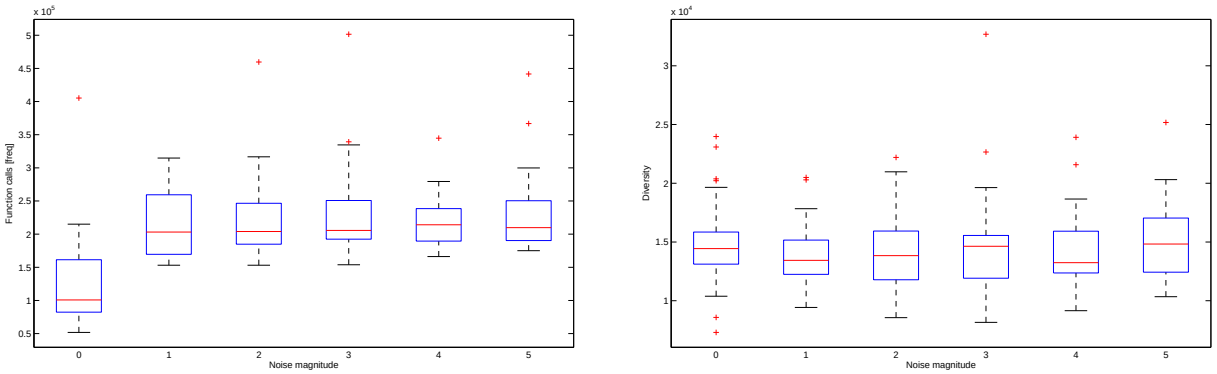
(b) Fitness of the best mechanisms for different noise magnitudes. In here we do not see any difference in fitness between the groups.

Figure 6-5: Box-plots of the robustness measures and fitness values for different noise magnitudes.

We expected in Chapter 5 that changing the noise magnitude would not influence the computational efficiency. To verify this we will use the function calls of the simulation function in our algorithm, which is given in Figure 6-6a, as a measure for computational efficiency. The noise magnitude of 0 is the same as no robust optimization which means that it worked with a memory size of 1. For this reason a noise magnitude of 0 is not included in the comparison between the amount of functions calls over the different noise magnitudes. We tested the data for normality using Shapiro-Wilks, which showed that all noise magnitudes were significantly not normal as shown in Table 6-2. Transforming the data did not give better results. Therefore a Kruskal-Wallis test was performed which showed no difference between the groups with $\chi^2(5)=1.48$ $p=0.8296$. This is as expected and a good sign, as it means that we can influence the robustness of the designed mechanisms without sacrificing computational efficiency. With this the only result that is influenced by the noise magnitude is the robustness making this an ideal method of influencing the robust optimization method. The noise magnitudes of 2, 3 and 5 have better robustness than a noise magnitude of 0, but between them we found no difference. It is possible that changing the noise magnitude did influence the algorithm on points we did not see or measure, therefore we should choose the noise magnitude that has the best robustness while having the least changes compared to the normal automated mechanism design. The result of this is that a noise magnitude of 2 is the most optimal setting according to our data.

Noise magnitude	p-value
1	0.03
2	<0.001
3	<0.001
4	0.035
5	<0.001

Table 6-2: Shapiro-Wilks test results of the distribution of the function calls for the different noise magnitudes which shows that the data is not normal distributed.



(a) Distribution of the amount of function calls per algorithm run for each noise magnitude. If we exclude the noise magnitude of 0, we do not see difference between the groups.

(b) Distribution of the diversity over the course of the algorithms for each noise magnitude, which shows no difference between the different noise magnitude settings.

Figure 6-6: Box-plots of the function calls and diversity of the evolutionary algorithm for different noise magnitudes.

6-4-2 Discussion

If the noise magnitude is chosen even larger than we did now it is possible that other problems will emerge. We can expect that with an extremely large noise magnitude the fitness will drop because the construction of the mechanism would completely change due to the noise. Further tests can be done but if this has any meaning is doubtful because the decrease in the robustness measure already smooths out. The reason for this flattening of the robustness at higher noise magnitudes is not clear. A possible cause can be that with the higher noise magnitudes do not give more information about the robustness of a mechanism.

In the data analysis we should take into account that during the transformation of the robustness data not all data was completely normal distributed. Therefore it is possible that the test results from the one-way ANOVA are less accurate.

6-4-3 Conclusion

Mechanisms from a robust optimization algorithm with a memory size of 16 and noise magnitude of 2, 3 or 5 have significantly better robustness than a from a robust optimization algorithm with a noise magnitude of 0 (meaning no robust optimization). No significant change in fitness, function calls or diversity has been found. As the noise magnitudes of 2, 3 and 5 have the same results we will choose the one with the least adjustment to the original algorithm as optimal. With the noise magnitude of 2 having the lowest bounds for the noise that will be the best choice.

6-5 Robust optimization with optimal settings

In previous chapters we have shown that for the experiments we ran, a memory size of 8 and a noise magnitude of 2 were the optimal settings. However it is possible that the combination of these two do not hold the same results. Therefore we will test the robust optimization method with these settings against the normal evolutionary algorithm.

6-5-1 Experimental results

The test results of the robust optimization method with a memory size of 8 and noise magnitude of 2 are compared to the automated mechanism design method with no robust optimization. The robustness was tested for normality using Shapiro-Wilks test and were found to be not normal with $p<0.001$. Transforming the data did result in an approximately normal distribution and is shown in Figure 6-7. Using Student's t-test we saw that the robustness of the robust optimization was significant different with $p=0.0019$. This is good as it shows that the found optimal settings do result in mechanisms with a better robustness as was expected. The fitnesses of these mechanisms are given in Figure 6-8a. We tested them using Student's t-test and found no difference between them with $p=0.9038$. Before we found that the better robustness did come at a price of computational efficiency. We plotted the function calls in Figure 6-8b to see if this is still the case with these settings. The data was found to be not normally distributed and transforming did not work. Therefore we used a Mann-Whitney U-test which showed a significant difference with $p<0.0001$. This shows that the robust optimization method with these settings does give more robust solutions but as expected that we do need to sacrifice computational efficiency. The upside is that this method does not reduce the fitness of the resulting mechanisms.

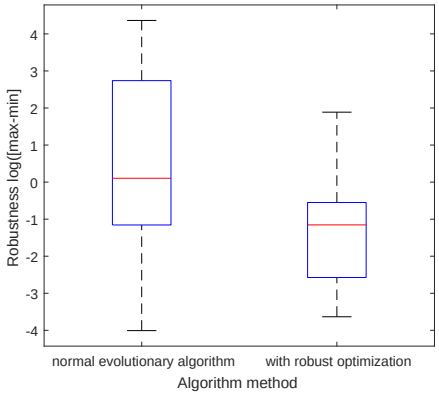
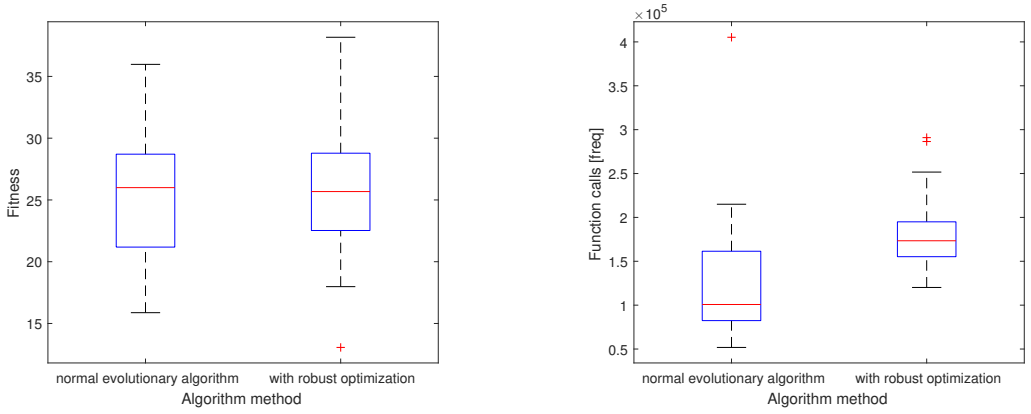


Figure 6-7: Distribution of the transformed robustness of the best mechanisms of the normal algorithm and the robust optimization with optimal settings.



(a) Fitness of best mechanisms of the normal algorithm and the robust optimization with optimal settings

(b) Function calls of best mechanisms of the normal algorithm and the robust optimization with optimal settings

Figure 6-8: Distribution of the robustness of the six experiments with different sizes memory-banks.

6-5-2 Discussion

It is possible that there is interaction between the two variables. But we know from previous tests that a two-way ANOVA with higher settings will not result in significant differences. And a test with lower settings will not have any meaning is we will than compare three times a non-robust optimization against one robust optimization. It is possible that a larger sample size can increase the significance of the results with which such a test becomes possible. However this will take at least a 30 days computing which is not available for us at this moment.

6-5-3 Conclusion

We have shown that the robust optimization method with the optimal settings that determined in Chapter 6-3 and Chapter 6-4 does create more robust mechanisms. This without sacrificing fitness but it does increase the computational efficiency.

6-6 Direct memory filling

We now have verified that our method does give more robust mechanisms. But the reason for how we designed it, was that it should reduce the computational cost compared to the conventional methods such as the expectancy measure while resulting in mechanisms with the same robustness. In this chapter we will test the expectancy measure and analyze the results.

6-6-1 Experimental results

To compare our robust optimization to the the expectancy measure we will use the data from Chapter 6-5 which had a memory size of 8 and noise magnitude of 2. The expectancy measure of a mechanism will be calculating with the same amount of data points as the memory size of our method. This way they will have the same accuracy, as explained in Chapter 6-1. In practice this will mean that we compare our algorithm, where the memory is filled with one sample every generation, against the expectancy measure, where the whole memory is filled directly in the first generation.

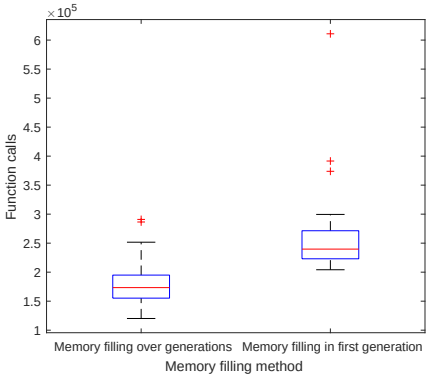
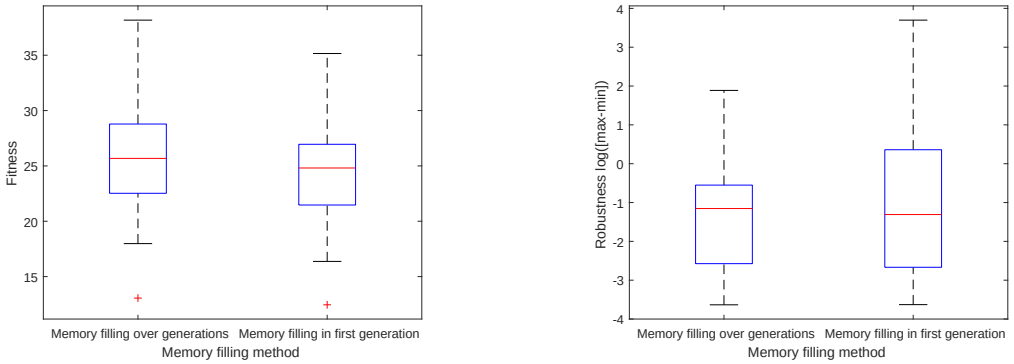


Figure 6-9: Distribution of the amount of function calls of our robust optimization and the robust optimization where the memory is filled in the first generation. It shows that our method is computational more efficient.

Our main objective for using a different robust optimization method was a lower computational cost. Therefore we will first look to the difference between the amount of function calls as seen in Figure 6-9. The distribution of the function calls was found to be not normal with $p=0.024$ for our method and $p<0.001$ for the direct memory filling method using Shapiro-Wilks tests. Transformation did not work, therefore we used Mann-Whitney U-test which showed that the means were significant different with $p<0.0001$. This clearly shows that our method is computationally more efficient than the conventional expectancy measure. The same comparison with a memory of 4 and noise magnitude of 2 resulted in a $p=0.7506$ using a Mann-Whitney U-test. And a comparison with a memory of 16 and noise magnitude of 3 resulted in a $p<0.0001$. Both of them did not show any significant difference between the fitnesses or robustnesses of the final mechanisms. This showed that with a smaller memory size the computational efficiency gain decreases while there is no effect seen on robustness or performance.

However, a different method for robust optimization can also result in different mechanisms. Therefore we we also test the fitness of the mechanisms in Figure 6-10a. Using Student's t-test wit $p=0.4187$ we did not see a significant difference in the fitness values of the mechanisms created with the different methods. Another difference can be that the conventional method will result in more robust mechanisms. This robustness in Figure 6-10 found not normal distributed using the Shapiro-Wilks test with $p<0.001$ for both methods. Transforming using the logarithmic function did get it approximately normal. Using the Student's t-test with $p=0.3775$ we did not see a significant difference in robustness between the two methods. With this we can say that we have created a computational more efficient method that generates mechanisms with similar robustness and fitness.



(a) Fitnesses of the final mechanisms for our robust optimization and the conventional robust optimization.

(b) Robustness of the final mechanisms for our robust optimization and the conventional robust optimization.

Figure 6-10: Box-plots of the fitness and robustness of our robust optimization and the robust optimization where the memory is filled in the first generation. As they are both not significant different we do not need to sacrifice robustness or fitness for our increase in computational efficiency.

In the problem statement we defined the requirement that we wanted to create a method that would produce more robust mechanisms within the time of the same order of magnitude as Kuppens did. As we explained earlier, it will not be able to compare the real run-times of our algorithm as this is influenced not only by the algorithm, but also the computational system, the software and the other processes running on the system. Still we will show the real calculation times in Figure 6-11. In here we see that the algorithm without robust optimization but with our improvements from previous chapters did take 0.4 and 1.4 hours to compute. The robust optimization method with a memory size of 8 and noise magnitude of 2 did take between the 1.4 and 2.4 hours. This is higher than without robust optimization which we already expected but still lower than the 2 to 6 hours seen by Kuppens. The robust optimization where the memory is filled in the first generation did take between the 2.3 and 3.7 hours to compute. This increase in computational time is what we expect if we looked to the amount of function calls of the different automated mechanism design methods.

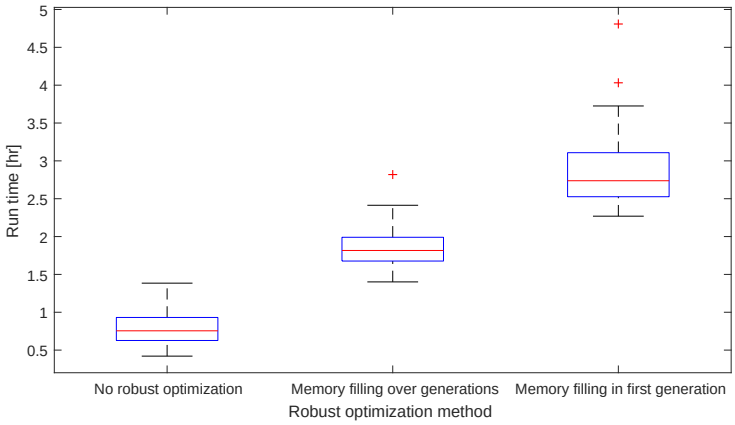


Figure 6-11: Box-plot of the time needed for the automated mechanism design method to come with a mechanisms. This is measured for the algorithm without robust optimization, our robust optimization and the optimization method where the memory is filled in the first generation. We can clearly see the increase of calculation-time needed for each method.

6-6-2 Discussion

With the smaller memory size we saw a reduction in the computational gain we achieved with our robust optimization. This is probably because with a smaller memory, a larger percentage of the total population will reach a completely filled memory-bank. This means that those mechanisms with a completely filled memory bank did have the same amount of simulations as when their memory was filled in their first generations. This reduces the difference between the two methods.

We still need to take into account that our transformed robustness is only approximately normal distributed. This can mean that the results from our statistical tests are less accurate.

6-6-3 Conclusion

With the comparison between our method and the expectancy measure we have shown that our method is computationally more efficient. These improved results on the conventional method do not come at a cost for fitness or robustness making this a good method for designing robust mechanisms.

6-7 Resulting mechanisms

The strength of the algorithm has been discussed in previous chapters. In this chapter we will discuss the resulting mechanisms themselves. The problem that the algorithm had to solve was to find a mechanism that could draw a straight line. Known solutions for this problem can roughly be divided in four categories: 4-bar mechanisms, true straight line mechanisms, pendula and other. Straight line approximations using 4-bars are the Roberts Mechanism, Watt's linkage, Chebyshev linkage, Chebyshev's Lambda Mechanism and Hoeckens linkage (not relevant for us as it uses a slider which is not available in our algorithm). More complex systems such as Peaucellier-Lipkin linkage (8-bar), Hart's Inversor and Hart's A-frame (both 6-bars) can replicate true straight lines. If our algorithm would replicate such a mechanism it would show that it can be used as a design-tool alongside a human. When a pendulum has a long enough arm is will approximate a straight line, therefore we include this as the third category. The final category contains all other solutions that can reproduce approximations of a straight line, for example a falling mass. To compare the systems we classified the final mechanisms from each experiment in Table 6-3 and 6-4.

Memory size	1	4	8	16	24	32
Pendulum	18	15	11	14	13	16
4 bar mechanism	10	9	15	13	15	11
Other	2	6	4	3	2	3

Table 6-3: Classification of best mechanism of each of the 30 runs per memory size.

Noise magnitude	0	1	2	3	4	5
Pendulum	18	17	13	14	12	15
4 bar mechanism	10	9	13	13	15	11
Other	2	4	4	3	3	4

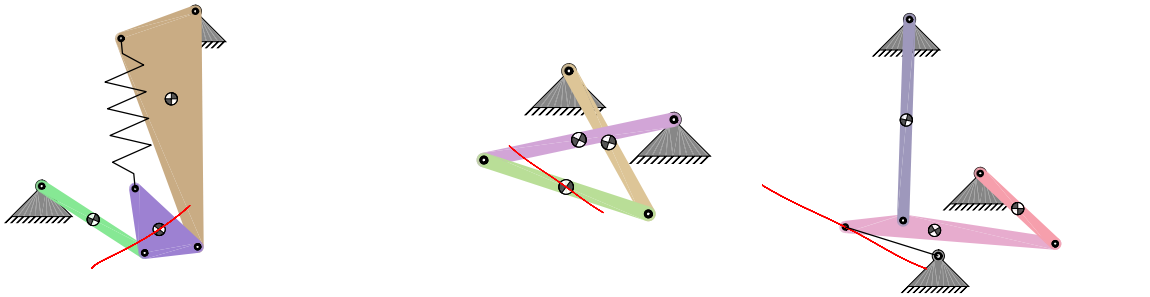
Table 6-4: Classification of best mechanism of each of the 30 runs per noise magnitude.

We see that every design option except a true straight line mechanism is among the final mechanisms of the algorithm. We do not see a direct preference for a specific solution to the problem. This makes the results from our algorithm less reproducible. A good sign is that the distribution over the categories is roughly the same for every setting in robust optimization. This means that our algorithm does not have much influence on the resulting mechanisms.

Examples of the resulting mechanisms are given in Figure 6-12 and 6-13. This includes the Roberts Mechanism in Figure 6-12a, Chebyshev linkage in Figure 6-12b, Chebyshev's Lambda Mechanism in Figure 6-12c, Watt's linkage 6-13a and the pendulum in Figure 6-13b In Figure 6-13c we see even an approximation of Harts Inversor.

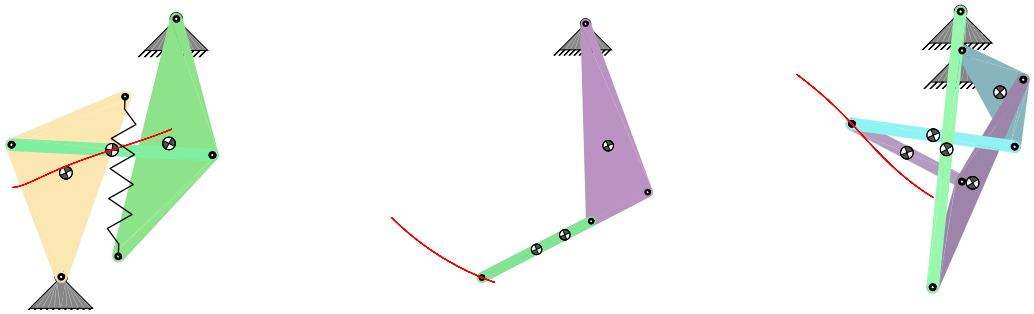
Among the solutions one mechanism stood out. As we said before, there are a couple of true straight line mechanisms. The mechanism in Figure 6-13c infringes upon Hart's Inversor. Its line is not yet perfectly straight but it is possible that if the algorithm was allowed to continue optimizing it would result in a true straight line mechanism.

Most of the mechanisms belonging to the category other were double pendula with a specific own frequency that made it always fall back along the same straight path as shown in Figure 6-14a. These double pundula often used a large amount of springs to ensure this own frequency. This often let to straighter lines than a normal pendula but have an increased complexity.



(a) Robberts mechanism **(b)** Chebyshev linkage **(c)** Chebyshev Lambda mechanism

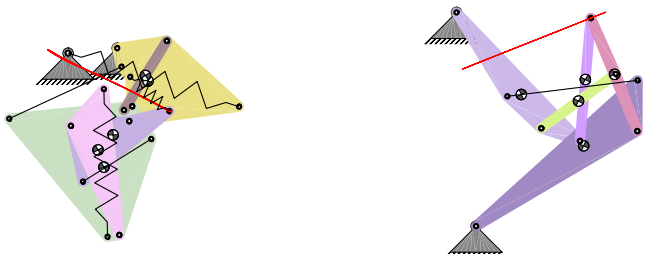
Figure 6-12: Several very good approximation of well known straight line mechanisms.



(a) Watt's linkage **(b)** Pendulum **(c)** Approximation of Hart's Inversor

Figure 6-13: Several approximation of well known straight line mechanisms.

The algorithm even came up with some more sophisticated mechanisms that fulfill the requirements very well for which we could not find any literature reference. One of them is the 6-bar mechanism shown in Figure 6-14b. It uses the same inverted parallelogram as Hart did for his true straight line mechanisms but it does not infringe upon his designs. This shows that our algorithm can be used to find novel solutions for existing problems.



(a) Forced double pendulum **(b)** 6-bar mechanism

Figure 6-14: Two original solutions to the straight line problem.

6-7-1 Discussion

Kuppens[13] did an experiment with similar settings. But he did not include robust optimization and had no fitness function size of the mechanism. His resulting mechanisms were mainly pendula while we have a wide range of different solutions. As we did not see a change in the final mechanisms with different settings that will probably not be the cause. The fitness function for the size of the mechanism does drive the system to design smaller mechanisms. True pendula are especially a good choice when the trajectory is far from the central axis. This is not really possible with the limitation on the size of the mechanism. Another cause for the lack of pendula can be that we asked for 2 degrees of freedom.

These 2 degrees of freedom can also be the cause that we did not see any perfect pendula. Every pendulum consisted of multiple mass elements as can be seen in Figure 6-13b or had other surplus elements. Experiments with a fitness function asking for a 1 degree of freedom could show if this was the cause. But as this takes more than 30 days to compute, this will not be possible within our project.

6-7-2 Conclusion

We have shown that with robust optimization the automated mechanism design program can create a wide variety of straight-line mechanism. It does not only produce mechanisms exactly the same as designs used for many decades in the industry. It also comes up with novel solutions that could be used.

6-8 Discussion

In the data from the experiments one particular event stood out. If we looked at the amount of function calls of the simulation function, most of the plots looked like in Figure 6-15. But occasionally the amount of function calls had extreme peaks ranging in the thousands of function calls in one generation as seen in the plots in Figure 6-16. The occurrence of these events can also be noticed when the algorithm is running. At these moments the algorithm can freeze for several hours in one generation. A possible explanation can be that within some populations, crossover has a very low chance of creating new mechanisms that fulfill the requirements. These requirements are if a mechanisms is connected to the ground, if all mass-elements are connected to each other, if more than one degree of freedom and if the simulation succeeds.

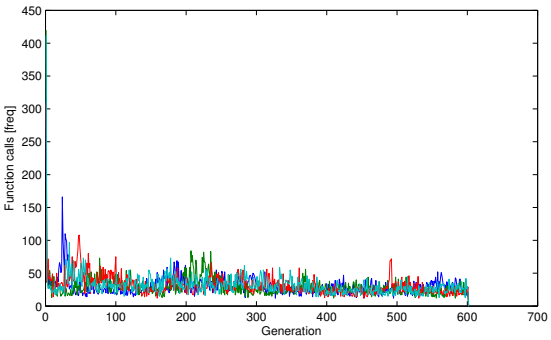
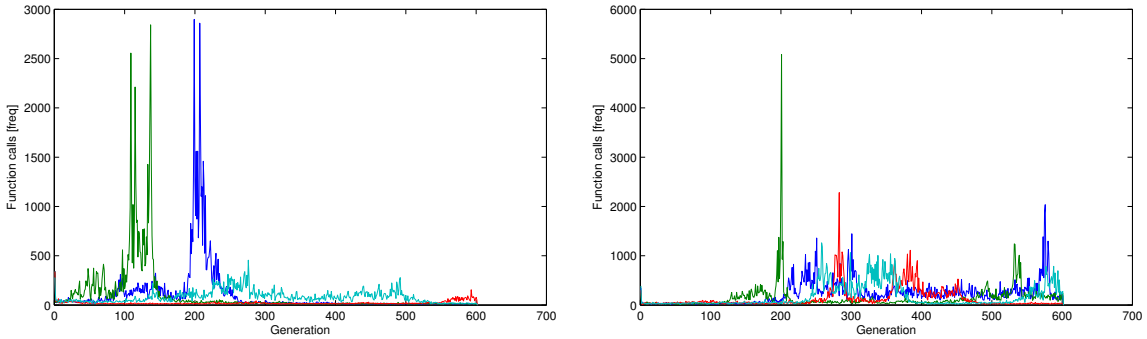
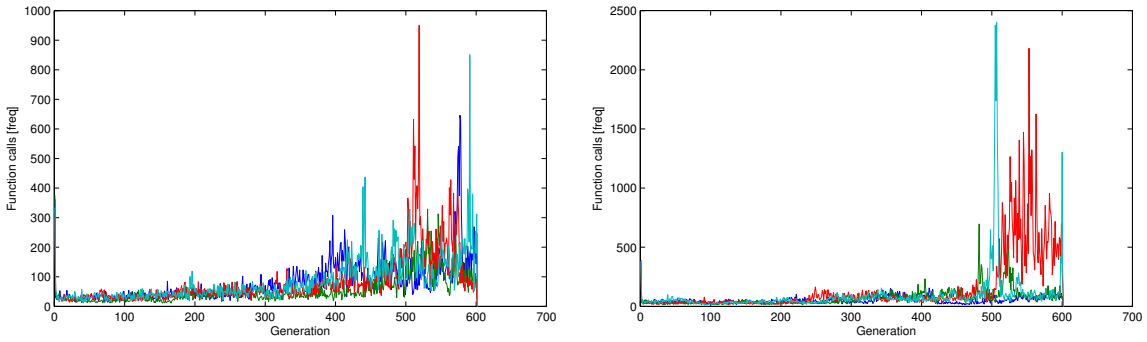


Figure 6-15: Function calls per generation of an automated mechanism design algorithm without robust optimization not showing extreme peaks.



(a) Function calls per generation with extreme peaks **(b)** Function calls per generation with extreme peaks



(c) Function calls per generation with extreme peaks **(d)** Function calls per generation with extreme peaks

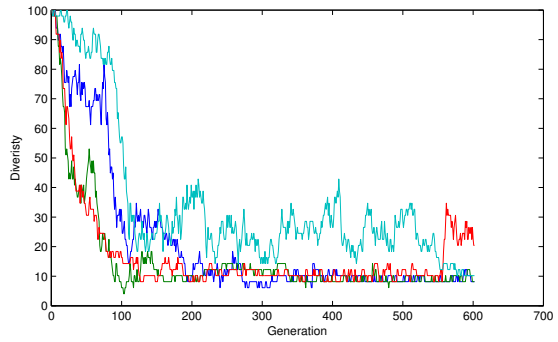
Figure 6-16: Function calls per generation from four experiments using the automated mechanism design program without robust optimization showing extreme peaks.

The most straightforward method of solving such a problem is by counting how many times the algorithm has tried to create a new mechanism with crossover. When this exceeds a certain value we create a mechanism without crossover. Two methods of creating a mechanism without crossover are possible: by taking a parent and using only mutation, or by creating a mechanism completely from scratch with random parameters.

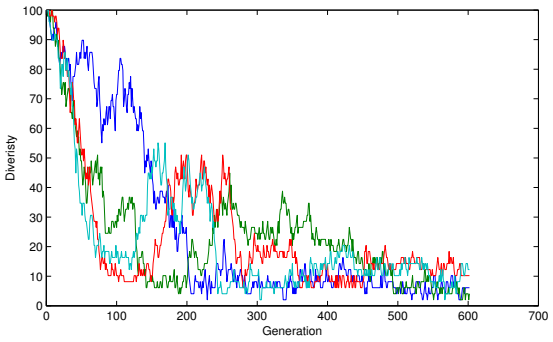
The peaks in function calls have a trend of appearing the same time as when that population has a low diversity. This can be seen in Figure 6-17 which are the diversity plots belonging to the plots in Figure 6-16. For example in Figure 6-16a in combination with 6-17a following the green line which represents an island. Here we see a drop in diversity around the 100'th generation, and a extreme in function calls around the same time.

At a point of low diversity the algorithm will most likely lock in a local optimum. As Kuppens[13] pointed out if we want to find the global optimum we need to try to keep the diversity high. He also tried the extinction method which increased diversity by replacing a part of the most common mechanism with new random mechanisms.

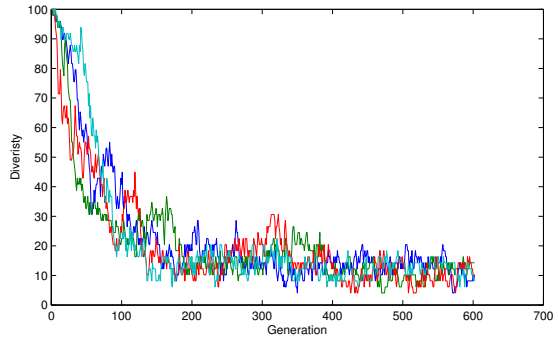
Knowing this, it is logical for our problem to choose the solution that increases the diversity if it is low. Therefore the second solution which replaces the mechanism with a new random one is the better choice. The other option has a higher chance of making small adjustments to the existing mechanisms which can fine-tune the best mechanism of that moment. But the low diversity means that that specific mechanism is well represented in the population. That means that it will already be used a lot for creating new mechanisms and more fine-tuning will probably have little influence.



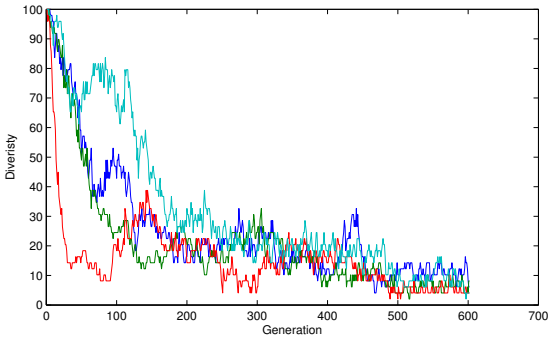
(a) Diversity per generation corresponding to Figure 6-16a



(b) Diversity per generation corresponding to Figure 6-16b



(c) FDiversity per ggeneration corresponding to Figure 6-16c



(d) Diversity per generation corresponding to Figure 6-16d

Figure 6-17: Diversity per generation from the four experiments in Figure 6-16 using the automated mechanism design program without robust optimization. These plots show a drop in diversity corresponding with the extreme peaks of function calls.

Another problem surfaced with the creation of the fitness functions for the straight line mechanism. When programming the fitness functions a lot of systems had to be fine-tuned before it worked properly. A systematic method that can reduce the difficulty, uncertainty and time needed for this process is needed. In the experiments done later we used a different method that tries to solve these problems as explained below. This method consists of two parts: converting the requirements to mathematical functions and combining those functions.

To translate a requirement to a mathematical function we need two things. First we need a measure of the requirement which can be calculated by the algorithm. And secondly we need the desired value of that measure. A fitness function needs to be continues and always have a slope towards the optimum so that every adjustment translates in knowledge about reaching the optimum. This directly implies that the function needs to be unbounded which also removes possible errors that can occur if extreme values are given as input. Our algorithm tries to minimize the fitness value of a mechanism. Therefore our fitness functions should be zero at the desired point and positive otherwise. As we need to combine several different fitness functions they should be normalized so that you know between which values the you can expect a value. The normalization we applied will give a 0 when perfect and 1.2 when at its worst value.

By defining the measure as a physical or understandable value we can intuitively define our desires around it. As the requirements define what is optimal we can also use it to define which values around it are still acceptable and from when on it becomes unusable. Also, if the fitness measure is understandable we can define what reasonable the maximum value is that it can reach. As an example the requirements for the size of a mechanism are given in Table 6-5.

	low maximum	not wanted	tolerable	desired	tolerable	not wanted	high maximum
Fitness value	1.2	1	0.1	0	0.1	1	1.2
size of the mechanism	-0.2	0	0.2	1	1.3	1.7	10

Table 6-5: Input settings for the fitness function which defines the requirements for the total dimension of the mechanism, between these values the function will be linear.

Fishburn[42] and later Fan[43] showed that an good fitness function should be in the form of E^{-1} with E the error (real value minus the desired). This was based on an evolutionary optimization algorithm that searched for the highest fitness value which can go to infinite. As our has 0 fitness as a lower limit we will create a plateau around the desired value which is acceptable for the requirements. If we use the fitness values given in Table 6-5 and linearize the fitness function between these points, we get the Figure 6-18.

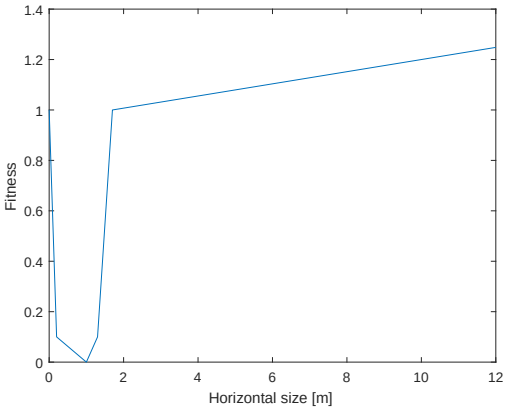


Figure 6-18: Fitness function of the size of a mechanism based on the values from Table 6-5.

We now have a systematic method of translating the requirement to a mathematical function. These functions still need to be combined to a single value with which the algorithm can work. Kuppens[13] did this by tuning the weights from Equation 6-13. At the basis of tuning these weights is that we know what the physical meaning of the several fitness functions are. We can use this to rank them on how important we think they are for our problem with 1 the most important and going up to less important. Sometimes we can not choose between which functions are more important which will result in functions with the same rank. From experience with tuning the straight line problem we saw that making the most important fitness function weights at least twice the next weight and letting the less important fitness function weights slowly decrease often works good (such as $[80, 40, 30, 20, 10]$ for the straight line problem). By using $w = r^{-1.5}$ with w the fitness function weight and r the rank of the fitness function, we get an approximation of the tuned weights we have seen before and it will roughly follow Fishburn's [42] advice. Another choice for combining the fitness functions can be the use of a Pareto front, using this in our algorithm should be investigated.

As a short literature survey did not hold much information, this proposed method is mostly based on experience. But as we want to further reduce the influence of the human on the design methods this will be an important part. Proper research should be done to automate the fitness function creation process.

In Chapter 6-2 we explained how to use the correlation coefficient as a measure for the straightness of a line. This measure is based on the work of Stojmenović [40]. He also proposed several other measures for straightness which could work better but this has not been investigated.

6-9 Conclusion

In this chapter we have verified that our robust optimization algorithm for genetic programming can generate robust mechanisms. This without sacrificing fitness or diversity. Our method will increase the computational costs but we have shown that it is less computational heavy than conventional robust optimization, while it will result in mechanisms with the same robustness and fitness. The optimal settings to get a good balance between robustness and computational efficiency are a memory size of 8 and noise magnitude of 2.

Chapter 7

Test-cases

In Chapter 6 we have shown that the robust optimization method we proposed works for a straight line optimization problem. But an automated mechanism design method has to function for a wide variety of problems. In this chapter we will take all the improvements and use it to find solutions to a couple of different problems. The settings for the robust evolutionary mechanism design method are given in Table 7-1 and every design problem is run 10 times.

During the first couple of runs with these test-cases the system sometimes froze for hours. We suspected the source of the problem to be the extreme peaks in function calls as discussed in Chapter 6-8. Therefore we limited the attempts to use crossover for creating a new mechanism to 100, after this a new random mechanism was created as child. After this modification the problem did not occur anymore.

To reduce the time needed to tune the fitness function settings, we used the method proposed in Chapter 6-8. The settings for this are further elaborated in their respective chapters.

Generations	1000
Migrations	100
Islands	8
Migrants	1
Population size	[10,10]
Neigbourhood size	[3,3]
New mechanisms	20
Crossover rate	100 %
Mutation rate	5 %
Loss rate	3 %
Mass element thickness	0.01 [m]
Hinge radius	0.01 [m]
Material density	1180 [kg/m^3]

Table 7-1: Settings for the evolutionary algorithm.

The first problem we will solve using the automated mechanism design method with robust optimization, is the straight line problem in Chapter 7-1. This has already been attempted during the verification in Chapter 6 but this time it will be done using more optimal algorithm settings and the altered fitness function tuning method. Secondly we will try to find a mechanism that can draw elliptic curves in Chapter 7-2. This has already been attempted by Kuppens[13]. Finally in Chapter 7-3 we will try to solve a problem proposed by a colleague who needed a mechanism that would increase the input frequency.

7-1 Straight line problem

As already explained in Chapter 6, the straight line problem tries to find a mechanism that draws a straight line. During the verification step we already solved this many times. However the settings used then were not ideal but good enough to compare the results between each other. This was done to preserve calculation time. In this chapter we will first define new fitness functions using the method explained in Chapter 6-8. After that we will solve the problem using the more optimal settings from Table 7-1, a memory size of 4 and noise magnitude of 2.

Fitness functions

The fitness functions for the straight line problem use the same measures as explained in Chapter 6-2. The difference in the fitness function is how the error is translated to a fitness value. This is done as explained in Chapter 6-8. For this method we use the input values from Table 7-2. The corresponding plots of these the resulting fitness functions are given in Appendix E in Figure E-1. These fitness functions are combined to a single value by appointing weights to them. This is done by first ranking the functions as done in Table 7-2 and than calculating the weights using $w = r^{-1.5}$ which results in the weights $[55.7\ 19.7\ 10.7\ 7.0\ 7.0]$.

	max low	not wanted	tolerable	desired	tolerable	not wanted	max high	rank
Fitness value	1.2	1	0.1	0	0.1	1	1.2	
Straightness	0	0.75	0.95	1	1.05	1.25	1.5	1
Trajectory length	0	0.5	0.8	1	2	5.5	10	2
Complexity	-63	-20	-7	0	7	20	63	3
Degrees freedom	-2	-1	0	1	2	6	7	4
Size of the mechanism	-0.2	0	0.2	1	1.3	1.7	10	4

Table 7-2: Input settings of the fitness functions for the straight line problem, between these values the functions are linear.

Experimental results

The problem was solved 10 times and the resulting fitness values of the best mechanisms are shown in Figure 7-1. How the mechanisms are developed over the course of the algorithm can be seen in Figure 7-2a where we plotted the fitness of the best mechanisms for every generation of all the 10 runs of the algorithm. We see that every run still improved the fitness of their best mechanism after 600 generations. Especially the red and turquoise algorithm runs show major improvements in their later generations. We see in Figure 7-2b that these runs of the algorithm also had a higher diversity for a longer time than the other runs. If this made the improvement in their later generations possible or has the same cause as the late improvement is not clear.

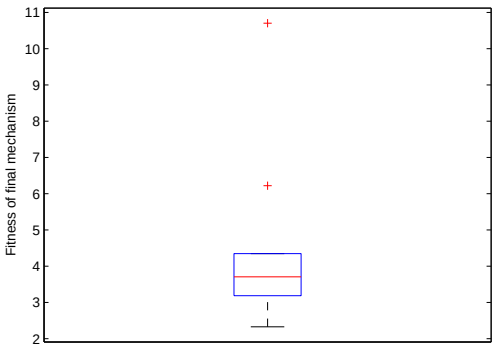
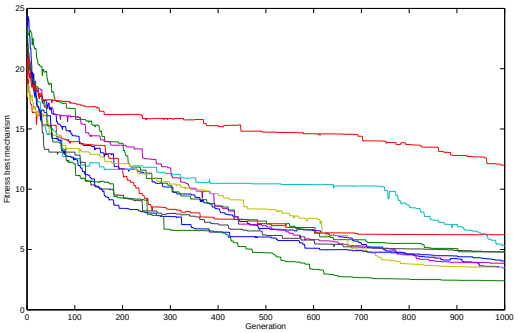
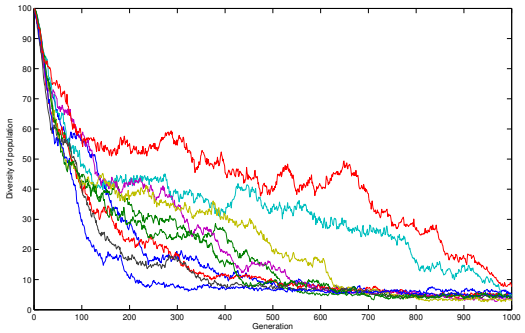


Figure 7-1: The fitness values of the final mechanisms from the 10 runs for the straight line problem.



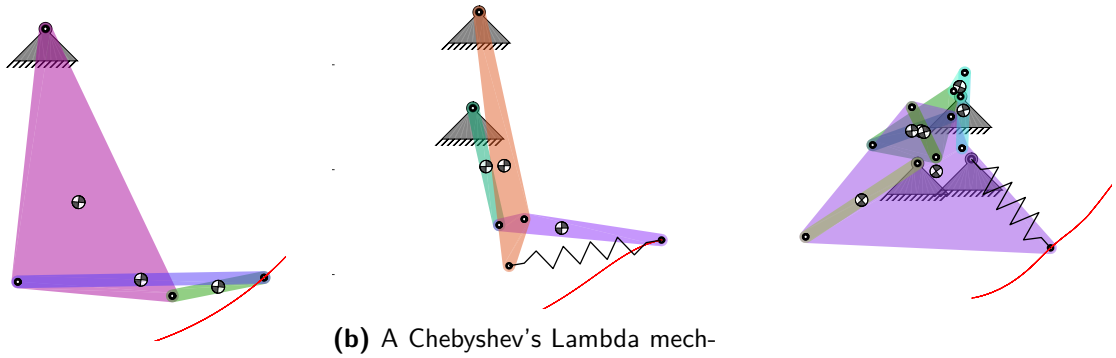
(a) The fitness values of the best mechanism of every generation.



(b) The diversity for each generation.

Figure 7-2: The fitness values and the diversity over the generations from the 10 runs for the straight line problem.

After classification of the final mechanisms we found seven pendula, one 4-bar mechanism (Chebyshev's Lambda mechanism) and two other. This is a better match with the results Kuppens[13] saw with his experiments. None of the mechanisms do create good approximations of a straight line. It is possible that we have a biased view as we only did 10 runs with the algorithm. The other possible cause can be that the fitness function that asked for 2 degrees of freedom applied a drive that reduced possibility of getting stuck in a local optimum.



(a) A pendulum.

(b) A Chebyshev's Lambda mechanism.

(c) A 8-bar mechanism.

Figure 7-3: Three of the mechanisms created by the automated mechanism design method solving the straight line problem.

7-2 Elliptic trajectory problem

Kuppens[13] tested his algorithm on a problem that tried to find a mechanism that would follow a elliptic trajectory. He found mechanisms that did a good approximation of an ellipse. With this he showed that this problem can function as a good test method for our automated mechanism design method. We will first explain how we calculate the fitness values. After that we will show the results and discuss the designed mechanisms.

Fitness functions

To calculate the fitness value for the elliptic trajectory problem we need to have a measure of how elliptic a trajectory is. Kuppens used the feature based approach of Coros [44]. As we will use the systematic fitness function tuning method from Chapter 6-8 we need to have a feeling what the values from the elliptic trajectory measure mean. The measure from Coros has several more tuning parameters inside which removes every connection it has with a real world value. This makes it impossible for us to determine if the value is tolerable and what we can expect as a maximum value. Therefore we cannot use this measure. Kuppens also proposed the modified Hausdorff distance[45] and the Fréchet distance[46]. But they have the same problem that the resulting values are difficult to understand.

In previous chapter we showed a method that can find a mechanism for a straight line. By defining the ellipse in terms of polar coordinates and than dividing it through its radius we can define the ellipse as a straight line. With this we can use a measure of straightness with which we have a feeling how it will behave and which values are acceptable. A problem with this method is that if only a part of an ellipse is recreated, it will not see this as an error. For this reason we also introduce a fitness function that gives the percentage of how much the trajectory encircles the center. In the straight line problem we had the fitness function for the length of the trajectory. This fitness function is to make sure that the final straight line is of a usable size. For the ellipse it would make more sense to use the surface area instead. The fitness functions for the complexity, degrees of freedom and size of the mechanism are exactly the same as for the straight line problem.

The settings to translate these measures to fitness values is given in Table 7-3. The resulting plots of these fitness function can be found in Appendix E in Figure E-2. Combining these fitness values was also done by ranking the fitness functions. The ranks as given in Table 7-3 resulted in the weights $[53.0\ 18.8\ 4.7\ 10.2\ 6.6\ 6.6]$ with which the fitness functions are combined to one fitness value. The other input settings for the algorithm are given in Table 7-1, a memory size of 4 and noise magnitude of 2.

	max low	not wanted	tolerable	desired	tolerable	not wanted	max high	rank
Fitness value	1.2	1	0.1	0	0.1	1	1.2	
Ellipse	-0.5	-0.3	-0.05	0	0.05	0.3	0.5	1
Trajectory circumvent	-2π	-0.3π	0π	0.25π	0.5π	0.8π	2π	2
Area ellipse	0	0.05	0.1	0.2	0.5	1	2	5
Complexity	-63	-20	-7	0	7	20	63	3
Degrees freedom	-2	-1	0	1	2	6	4	5
Size of the mechanism	-0.2	0	0.2	1	1.3	1.7	10	4

Table 7-3: Input settings of the fitness functions for the elliptic trajectory problem, between these values the functions are linear.

Experimental results

We ran the automated mechanism design method 10 times solving the elliptic trajectory problem. The fitness values of the resulting mechanisms are given in Figure 7-4. How the fitness values evolved over the generations is plotted in Figure 7-5a. If we compare this with the fitness from the straight line problem from Figure 7-2a we see some differences.

The first thing is that the fitness of most of the runs stops improving after 400 generations. If we look to the diversity of the algorithm for every generation as is given in Figure 7-5b we also see that it dropped a lot faster than the diversity of the straight line problem in Figure 7-2b. A possible cause for this fast decrease in diversity and stop of improvement of the fitness can be the method used to build the fitness functions or our method of creating a measure of how elliptic a trajectory is.

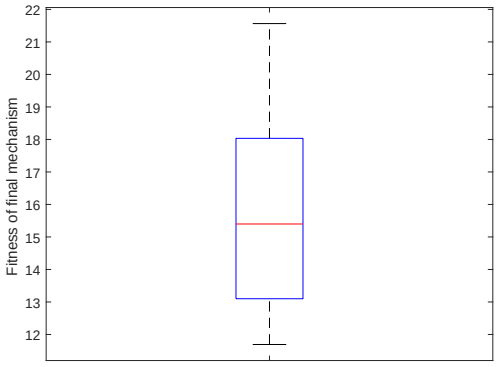
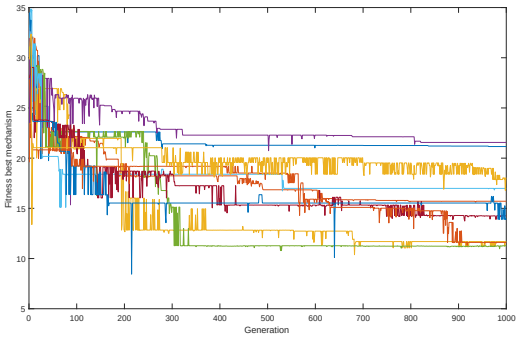
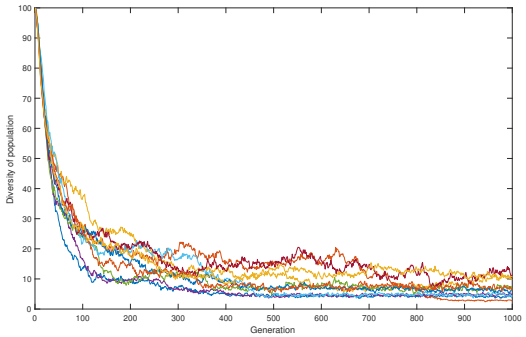


Figure 7-4: The fitness values of the final mechanisms from the 10 runs for the elliptic trajectory problem.

The second thing we can see in Figure 7-5a is a lot of spikes downwards. This is probably caused by the robust optimization algorithm. If the first fitness value in the memory is low and better than the best mechanism before it will show as a drop in the fitness plot. But if its second fitness calculation does result in an average fitness value higher than that of the best mechanism of previous generation, that mechanism will again be registered as best and the plot of the fitness value will go back up. At the end of the algorithm you see that this jiggering becomes smaller and happens less often. This is probably because most mechanisms in the population are similar to the best mechanism which will result in child mechanisms that are also similar and therefore more robust.



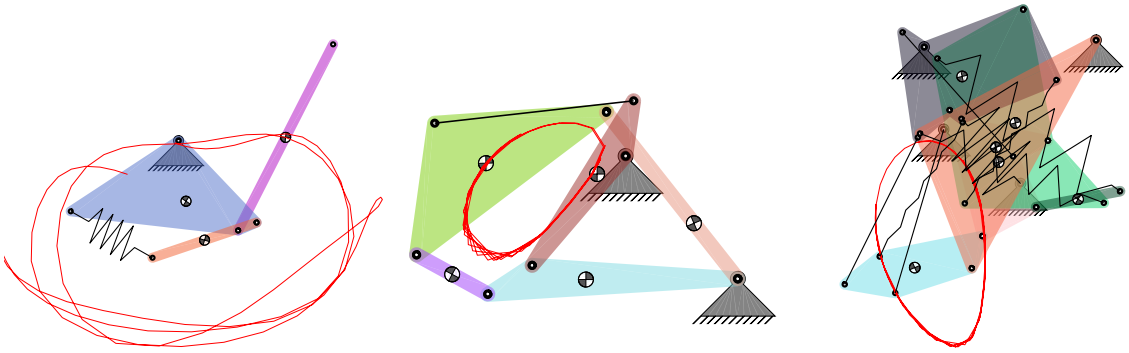
(a) The fitness values of the best mechanism of every generation.



(b) The diversity for each generation.

Figure 7-5: The fitness values and the diversity over the generations from the 10 runs for the elliptic trajectory problem.

The designs from the automated mechanisms design method resulted in four double pendula, five 4-bar mechanisms and the 7-bar mechanism in Figure and 7-6c. An example of the double pendula is given in Figure 7-6a. It is clear that this design will not result in a usable mechanism to draw elliptic lines. The 4-bar mechanisms, such as given in Figure 7-6b will make better mechanisms. Still the resulting trajectories are only rough approximations of an ellipse. They resemble the trajectory of a Chebyshev's Lambda mechanism which can be explained as their 4-bar setup is also an infringement on that design. The extra bars in Figure 7-6b are locked in a triangle and do not contribute to the movement.



(a) A double pendulum which does not make a good elliptic trajectory following mechanism.

(b) A Chebyshev's Lambda mechanism that follows approximately an elliptic curve with a flat side.

(c) A 7-bar mechanism that follows approximately an elliptic curve.

Figure 7-6: Three of the mechanisms created by the automated mechanism design method solving the elliptic trajectory problem.

7-3 Frequency multiplier problem

When looking for good test cases for our algorithm, a colleague suggested that we tried to solve a problem he had. He was looking for a mechanism that could double the input frequency without changing the amplitude of the input. As the previous two test-cases were trajectory problems, this proposed challenge is a good addition as it focuses more on the dynamics of a mechanism. To let the automated mechanism design method solve this problem we fist explain how we defined the fitness functions. After that we will show the results and discuss the final mechanisms.

Fitness functions

The mechanism that has to increase the input frequency will be used as a part of a larger design. Therefore it would be practical that the input and output of the mechanism are connected to the ground. Knowing this, only mechanisms with two or more connections to the ground will be a feasible solution. The algorithm will be able to find solutions without changes but to reduce the computation time we made sure that a mechanism did have this minimum required connections to the ground before we calculated its fitness.

At the center of the problem is the demanded increase of the input frequency to the output. To determine the best frequency ratio among the available connections to the ground we calculated for every of those connections the changes in direction and the time between those changes. The time that occurred most, and was larger than a minimum of 0.2 seconds, is then chosen as the frequency of that connection. After that the couple of connections with a ratio closest to the desired ratio was determined and used as the first fitness function for this problem.

The second demand was that the amplitude would stay the same. For this we calculated this ratio for the couple of connections found before. Finally to make sure that the resulting mechanism is also practical we need a minimum amplitude which is also introduced as a fitness function. These three functions combined with the drive for low complexity, one degree of freedom and a desired size of the algorithm are used as input for our algorithm. The settings for these functions are given in Table 7-4 and the resulting plots of these fitness functions are given in Appendix E in Figure E-3. Combining these fitness values was also done by ranking the fitness functions. The ranks as given in Table 7-4 resulted in the weights $[51.2\ 9.9\ 18.1\ 6.4\ 4.6\ 9.9]$ with which the fitness functions are combined to one fitness value. The other input settings for the algorithm are given in Table 7-1, a memory size of 8 and noise magnitude of 2.

	max low	not wanted	tolerable	desired	tolerable	not wanted	max high	rank
Fitness value	1.2	1	0.1	0	0.1	1	1.2	
Frequency ratio	0	0.8	1.2	1.5	2	2.5	10	1
Amplitude ratio	0	0.2	0.5	1	2	4	20	3
Amplitude	0	0.15π	0.3π	0.5π	0.8π	1.2π	2π	2
Complexity	-63	-20	-7	0	7	20	63	4
Degrees freedom	-2	-1	0	1	2	6	7	5
Size of the mechanism	-0.2	0	0.2	1	1.3	1.7	10	3

Table 7-4: Input settings of the fitness functions for the frequency multiplier problem, between these values the functions are linear.

Experimental results

The frequency multiplier problem was solved 10 times using our automated mechanism design method. The fitness of the resulting mechanisms can be seen in Figure 7-7. In Figure 7-8a we see how the fitness developed over the generations. In here we can clearly see similar aspects as we did see for the elliptic trajectory problem. We see the same peaks downwards which are probably the effects of the robust optimization. And we also see that the fitness stops improving quite fast compared to the straight line problem. However in contrast to the elliptic trajectory problem we still see some improvement of the fitness in the later generations.

In Figure 7-8b we see the diversity over the generations. It looks like this decreased even faster than the diversity of the elliptic trajectory curve which corresponds with the slow fitness improvements in Figure 7-8a.

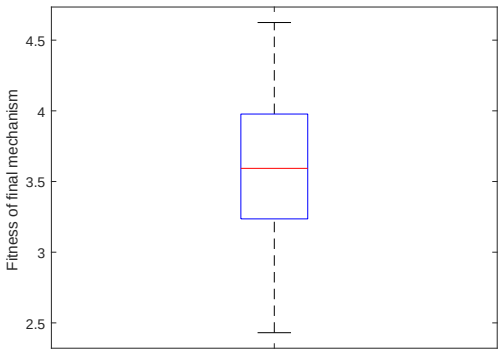
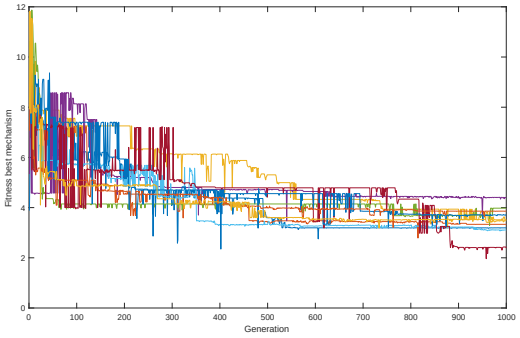
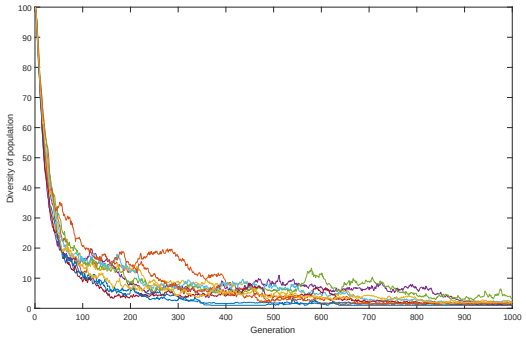


Figure 7-7: The fitness values of the final mechanisms from the 10 runs for the frequency multiplier problem.



(a) The fitness values of the best mechanism of every generation.

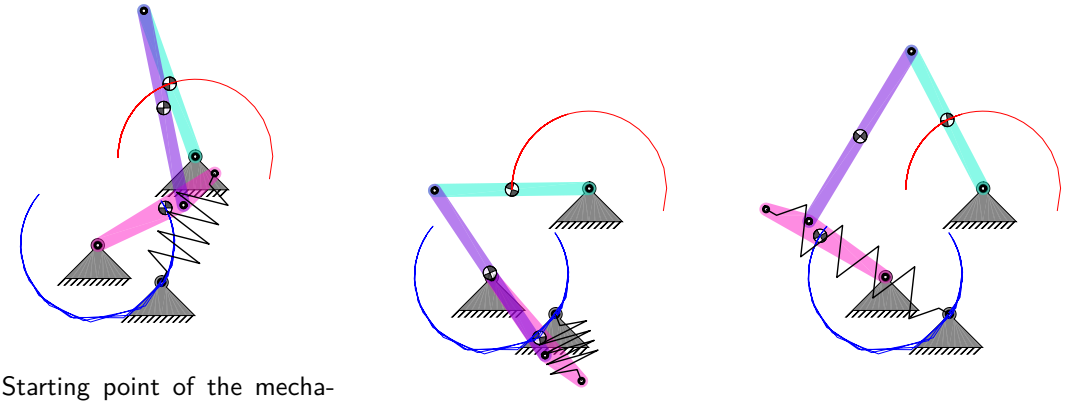


(b) The diversity for each generation.

Figure 7-8: The fitness values and the diversity over the generations from the 10 runs for the frequency multiplier problem.

Two of the mechanisms designed by our automated mechanisms design algorithm working on the frequency multiplier problem are shown in Figure 7-9 and 7-10. The algorithm designed six 4-bar mechanisms that positioned themselves in a singular position in which they switched the bar which would be on top. This way they constantly switch between a long and a short swing. At the end of the simulation one will have more long swings and the other more short swings. As we used the mode to calculate the dominant frequency of a hinge these mechanisms will have a good frequency ratio but are not usable as a real mechanisms for our problem. This means that our method of defining the frequency ratio can be improved.

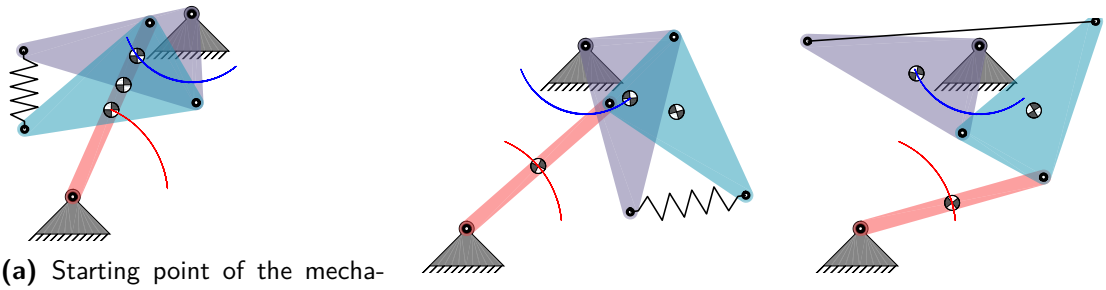
Three of the other mechanisms did follow the setup as seen in Figure 7-9. The working of these mechanisms are similar to the working of a piston. This is a beautiful method for fulfilling the frequency ratio requirement. A big downside is that this method does not fulfill the amplitude requirement. The driving hinge will always travel a smaller trajectory (or amplitude). This makes this mechanism useful if the amplitude requirement is not that strict.



(a) Starting point of the mechanism. **(b)** Mid point of the mechanism. **(c)** End point of the mechanism.

Figure 7-9: Mechanism designed by our algorithm solving the frequency multiplier problem.

The last mechanism our algorithm designed is seen in Figure 7-10. It works by unfolding two bars and than folding them back up, repeating this method on and on. This solves the frequency ratio requirement as it always doubles the input frequency. The amplitude ratio requirement is also solved. We can even influence the amplitude ratio by changing the dimensions of the smaller two bars (or triangles here) in contrast to the big bar. The amplitude itself cannot be bigger than 0.5π as otherwise the smaller bars will get locked in an overstretched position. But with an additional mechanism after the output the amplitude can be increased to the current demands making this an easily solvable problem. The only downside of this design is that it will only work for frequency ratios of 2. With this design we have shown that we can also get good working mechanisms for dynamical problems. We expected before running the automated mechanism design method that a solution would be a mechanism that exploits a singular position. We also thought that only very small points in the solution space would be able to do this and therefore we suspected that it would be a big challenge for our algorithm. Still it was able to find a mechanism that solves the problem in the first 10 runs.



(a) Starting point of the mechanism. **(b)** Mid point of the mechanism. **(c)** End point of the mechanism.

Figure 7-10: Mechanism designed by our algorithm solving the frequency multiplier problem.

7-4 Discussion

The straight line mechanism clearly showed continues improvement of the fitness value over the course of the algorithm. However, with the elliptic trajectory problem and the frequency multiplier problem, this improvement slowed down very early. They also showed a steep drop in diversity in the first few generations while the diversity of the straight line problem slowly decreased over the generations. This drop of diversity is probably the cause of the very slow improvement of their fitness in the later generations. It is possible that this decrease of diversity is caused by our choice of fitness functions or fitness measures. But another very likely reason can be our method of tuning those fitness functions. Further research should be done to find a method that optimizes the working of the algorithm with minimal manual tuning of parameters.

7-5 Conclusion

Our robust optimization for automated mechanisms design has shown to find good solutions for the straight line problem and frequency multiplier problem. This was achieved with minimal tuning of the fitness function parameters. The results from the elliptic trajectory problem shows promise but also makes clear that we need to improve the algorithm. Still overall we see that our automated mechanisms design method is applicable for a wider range of problems.

Chapter 8

Discussion

The problem with current automated mechanism design methods is the gap between the resulting design and how it will function after manufacturing. To reduce this problem we did three improvements.

The first improvement was decreasing its computational costs by improving the incidence value calculation.

Secondly we improved the accuracy of the simulation by adding a better weight and inertia calculation for the mass elements. With this we introduced the material thickness and density as input settings for the algorithm reducing its design freedom. By letting the system choose its own thickness or density it is possible that better solutions can be found. If this is introduced it should be taken in account that these values should maintain within realistic bounds. The accuracy of the weight can be improved further by adding the weight of the hinges, springs and other small parts. However it is doubtful that this increased accuracy will be an overall improvement as it will also increase the computational cost. What will be a big improvement on the simulation accuracy is adding friction in the hinges. This will mean we will remove energy from the system which needs to be countered by adding actuators as an element. A successor working on our algorithm should investigate this option.

Our third and biggest improvement was the robust optimization which reduced the effect of production errors on the performance. As solution we proposed a new method of robust optimization specially designed for use in the genetic programming. In here we have based the shape of the noise magnitude on the ISO-2768-m norm [39]. As we did not investigate other shapes of these bounds we do not know the effects of it, or if it is even necessary. The sudden steps in the allowed error can introduce instability or other problems. The effect of the shape of the noise should be investigated further, for example by using ISO-268 for hinges or simplifying the system by using a linear noise magnitude. Besides that we now have only influenced the production dimensions of the mass elements, hinges and springs of the mechanisms. Other dimensions such as the thickness, density and spring stiffness, and in future versions maybe even the friction coefficient and wear, also have an effect on the performance of a mechanism. The effect on the fitness by disturbing those dimensions should be investigated. We now measured this effect by taking the difference between the outer most fitness value of the disturbed fitness calculations. This choice is heavily influenced by the total amount of fitness calculations done. If we choose for example the 3-sigma of the distribution we could reduce its influence on the amount of fitness calculations.

The comparison between the different settings of our algorithm is also influenced by the amount of experiments done per setting. We did only 30 of such experiments for each algorithm setting. Increasing this amount of experiments can show other effects of the robust optimization such as interaction of the variables which we were not able to check with our current data. This, and testing more different memory sizes and noise magnitudes, can results in better settings for our robust optimization algorithm but was not possible in our current time frame.

While making these adjustments to the algorithm some other things surfaced. A major problem we found was tuning the fitness functions in such a way that the algorithm would converge to a good solution. This required a lot of testing with the algorithm. This testing and evaluating the results removes the purpose of our algorithm as we could just have designed a good mechanisms in that time. We proposed and tested a method which did the tuning systematic and removed the need for testing. The results of this method were not satisfying as it probably decreased the convergence rate to a good solution. It is very important that a good method for systematic or even automatic fitness tuning is found to make this automated mechanisms design method usable. Other options on calculating the fittest mechanism, such as the pareto front, should be investigated.

Finally we noticed some extreme peaks in the plots of the function calls as explained in Chapter 6-8. We limited them by giving the breed function an upper bound of 100 after which it will stop trying to use crossover to create a child mechanism. We presumed that this was due to a low diversity which made it difficult for the crossover function to create valid mechanisms. As this is only a possible cause the real source of this problem should be investigated.

Chapter 9

Conclusion

We showed that we can make an evolutionary automated mechanism design program that generates practical designs whose performance will have minimal change when production tolerances are considered, without sacrificing much calculation time.

The minimization of the performance change due to production tolerances was achieved by a novel robust optimization method specifically useful in genetic programming. By letting this method update the fitness of the mechanism with noisy fitness values every generation, we showed that we can achieve the best results with a fitness memory-bank size of 8 fitness values. The fitness values in that memory-bank are calculated using noisy dimensions which show the best result when the noise is limited to 2 times the ISO-2678-m norm. This made the final mechanisms significantly more robust while minimizing the calculation costs which stayed between the 1.4 and 2.4 hours. By simplifying the incidence value calculation we further decreased the calculation time of the algorithm. And by improving the weight and moment of inertia calculations we increased the accuracy of the simulation of the mechanisms.

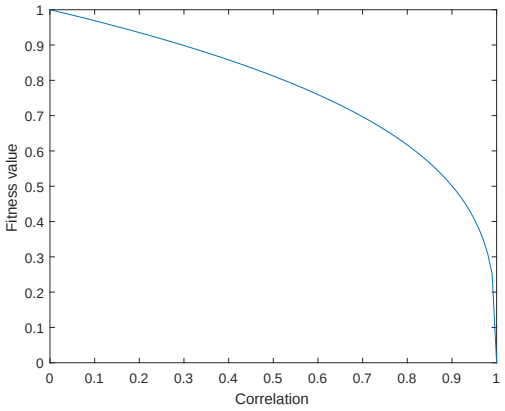
Appendix A

Used symbols

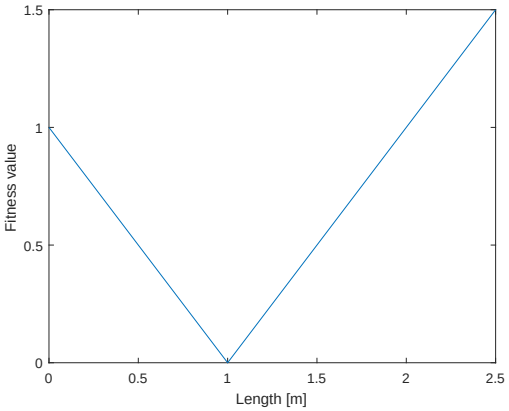
Symbol	Units	Description	First use
A	m^2	Surface area of mass element	Equation 4-1
C	$-$	Complexity	Equation 6-7
E	$-$	Error	Equation 6-4
F	$-$	Fitness function	Equation 5-1
F_{spring}	N	Spring force	Equation 2-1
f	$-$	Continues alternative of I	Equation 3-4
G_{n}	$-$	Ground element	Chapter 2-1-1
g	$-$	Continues alternative of I_{v2}	Equation 3-4
g_{n}	$-$	Amount of generations	Equation 6-1
H_{n}	$-$	Hinge connection	Chapter 2-1-1
I	$-$	Incidence value	Equation 2-2
I_{v2}	$-$	Incidence value v_2 part	Equation 3-2
I_{mass}	kgm^2	Mass moment of inertia	Equation 4-3
i	$-$	$[]^{th}$ value in a row	Equation 2-2
k	N/m	Spring constant	Equation 2-1
L_0	m	Free length	Equation 2-1
L_{d}	m	Length of trajectory	Equation 6-5
$L_{\text{Hor/Ver}}$	m	Horizontal or vertical size of mechanism	Equation 6-10
L_{Desired}	$-$	Desired value in fitness function	Equation 6-6
$L_{\text{Trajectory}}$	$-$	Desired value in fitness function	Equation 6-6
M_{n}	$-$	Mass element	Chapter 2-1-1
m_{mass}	kg	Weight of mass element	Equation 4-1
n	$-$	Numbering	Equation 2-3
$n_{\text{M/H/S}}$	m	Number of masses, hinges or springs	Equation 6-7
O_{M}	$-$	Origin of polygon	Figure 4-1
u	m	Distance	Equation 2-1
R	$-$	Robustness	Equation 5-1
r_{survival}	$\%$	Survival chance	Equation 6-1
S_{n}	$-$	Spring connection	Chapter 2-1-1
t	$-$	Time step	Equation 6-2
$t_{\text{thickness}}$	m	Thickness of mass element	Equation 4-1
v_{n}	$-$	Vertex number	Equation 2-2
$\bar{w}$	$-$	Vector with fitness weights	Equation 6-13
x_{i}	m	Horizontal location of connection	Equation 4-1
y_{i}	m	Vertical location of connection	Equation 4-1
α	rad	Angle	Figure 4-2b
δ	$-$	Noise	Equation 5-1
κ	rad	Curvature	Equation 6-2
ρ	kg/m^3	Density of mass element	Equation 4-1

Appendix B

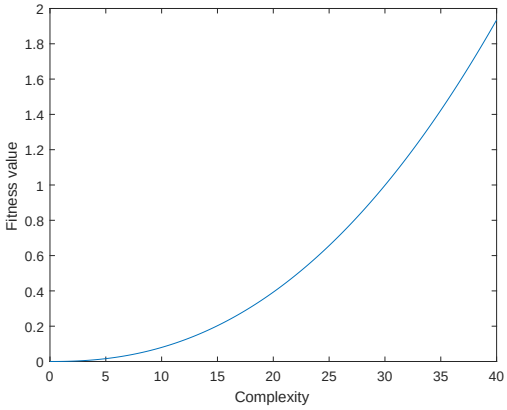
Plots fitness functions



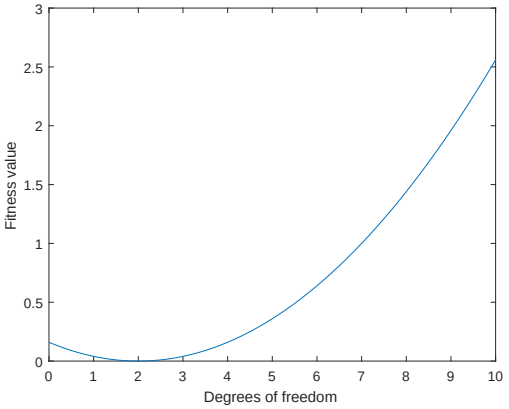
(a) Fitness function for the line curvature



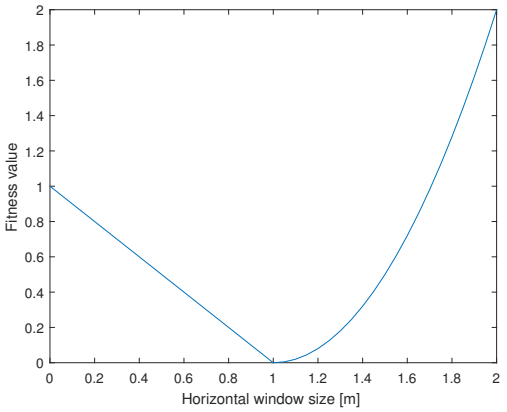
(b) Fitness function for the length of the line



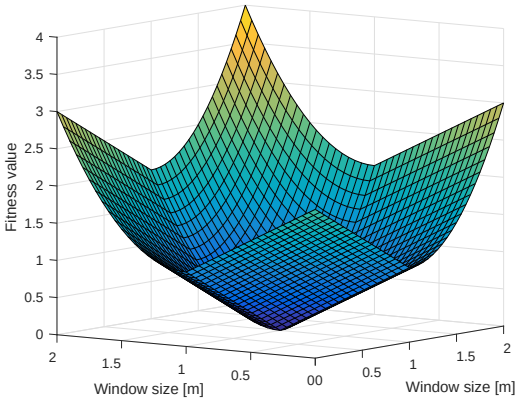
(c) Fitness function for the complexity of the mechanism



(d) Fitness function for the degrees of freedom of



(e) Fitness function for the size of the mechanism in one dimension



(f) Fitness function for the two dimensions of the window

Figure B-1: Plots of the fitness functions for the straight-line problem used to verify the robust optimization method.

Appendix C

Statistical tests memory-size

Fitness one-way ANOVA:

Source	SS	df	MS	F	Prob>F
Columns	170.36	5	34.0727	1.17	0.3247
Error	5057.3	174	29.0649		
Total	5227.66	179			

Diversity one-way ANOVA:

Source	SS	df	MS	F	Prob>F
Columns	5.92155e+07	5	1.18431e+07	0.74	0.5931
Error	2.7779e+09	174	1.5965e+07		
Total	2.83712e+09	179			

Robustness (transformed) one-way ANOVA:

Source	SS	df	MS	F	Prob>F
Columns	55.183	5	11.0367	2.83	0.0174
Error	678.056	174	3.8969		
Total	733.239	179			

Robustness (transformed) post-hoc Tukey:

1	1.0000	2.0000	−0.2188	1.2337	2.6861	0.1491
2	1.0000	3.0000	0.0605	1.5130	2.9655	0.0355
3	1.0000	4.0000	0.3075	1.7600	3.2125	0.0073
4	1.0000	5.0000	−0.2247	1.2278	2.6803	0.1530
5	1.0000	6.0000	−0.2981	1.1544	2.6069	0.2086
6	2.0000	3.0000	−1.1731	0.2794	1.7319	0.9941
7	2.0000	4.0000	−0.9261	0.5264	1.9788	0.9070
8	2.0000	5.0000	−1.4583	−0.0058	1.4466	1.0000
9	2.0000	6.0000	−1.5317	−0.0792	1.3733	1.0000
10	3.0000	4.0000	−1.2055	0.2470	1.6995	0.9967
11	3.0000	5.0000	−1.7377	−0.2852	1.1673	0.9935
12	3.0000	6.0000	−1.8111	−0.3586	1.0939	0.9816
13	4.0000	5.0000	−1.9847	−0.5322	0.9203	0.9030
14	4.0000	6.0000	−2.0581	−0.6056	0.8469	0.8428
15	5.0000	6.0000	−1.5259	−0.0734	1.3791	1.0000

Function calls Kruskal-Wallis:

	Source	SS	df	MS	Chi−sq	Prob>Chi−sq
1	Columns	203516.3	5	40703.3	74.96	9.4836e−15
2	Error	282468.7	174	1623.4		
3	Total	485985	179			

Function calls post-hoc Dunn's:

1	1.0000	2.0000	−67.3228	−27.9333	11.4562	0.4396
2	1.0000	3.0000	−96.1895	−56.8000	−17.4105	0.0004
3	1.0000	4.0000	−114.6228	−75.2333	−35.8438	0.0000
4	1.0000	5.0000	−125.7562	−86.3667	−46.9772	0.0000
5	1.0000	6.0000	−135.0562	−95.6667	−56.2772	0.0000
6	2.0000	3.0000	−68.2562	−28.8667	10.5228	0.3851
7	2.0000	4.0000	−86.6895	−47.3000	−7.9105	0.0066
8	2.0000	5.0000	−97.8228	−58.4333	−19.0438	0.0002
9	2.0000	6.0000	−107.1228	−67.7333	−28.3438	0.0000
10	3.0000	4.0000	−57.8228	−18.4333	20.9562	0.9396
11	3.0000	5.0000	−68.9562	−29.5667	9.8228	0.3466
12	3.0000	6.0000	−78.2562	−38.8667	0.5228	0.0564
13	4.0000	5.0000	−50.5228	−11.1333	28.2562	0.9996
14	4.0000	6.0000	−59.8228	−20.4333	18.9562	0.8736
15	5.0000	6.0000	−48.6895	−9.3000	30.0895	1.0000

Appendix D

Statistical tests noise magnitude

Fitness one-way ANOVA:

Source	SS	df	MS	F	Prob>F
Columns	160.56	5	32.1128	1.13	0.3477
Error	4957.14	174	28.4893		
Total	5117.7	179			

Diversity one-way ANOVA:

Source	SS	df	MS	F	Prob>F
Columns	3.47647e+07	5	6.95295e+06	0.54	0.7441
Error	2.23134e+09	174	1.28238e+07		
Total	2.26611e+09	179			

Robustness (transformed) one-way ANOVA:

Source	SS	df	MS	F	Prob>F
Columns	59.344	5	11.8688	3.25	0.0079
Error	636.151	174	3.656		
Total	695.495	179			

Robustness (transformed) post-hoc Tukey:

1.0000	2.0000	−0.2967	1.1102	2.5171	0.2156
1.0000	3.0000	0.1385	1.5454	2.9523	0.0216
1.0000	4.0000	0.3531	1.7600	3.1669	0.0049
1.0000	5.0000	−0.1686	1.2383	2.6452	0.1216
1.0000	6.0000	0.1028	1.5097	2.9165	0.0271
2.0000	3.0000	−0.9717	0.4352	1.8421	0.9510
2.0000	4.0000	−0.7571	0.6498	2.0567	0.7762
2.0000	5.0000	−1.2788	0.1281	1.5350	0.9998
2.0000	6.0000	−1.0074	0.3995	1.8063	0.9660
3.0000	4.0000	−1.1923	0.2146	1.6215	0.9980
3.0000	5.0000	−1.7140	−0.3071	1.0997	0.9894
3.0000	6.0000	−1.4427	−0.0358	1.3711	1.0000
4.0000	5.0000	−1.9286	−0.5217	0.8852	0.8984
4.0000	6.0000	−1.6572	−0.2504	1.1565	0.9959
5.0000	6.0000	−1.1355	0.2714	1.6783	0.9940

Function calls Kruskal-Wallis:

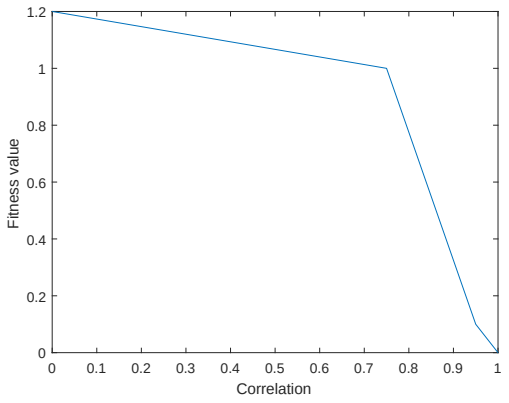
Source	SS	df	MS	Chi−sq	Prob>Chi−sq
Columns	2799.7	4	699.93	1.48	0.8296
Error	278437.8	145	1920.26		
Total	281237.5	149			

Function calls post-hoc Dunn's:

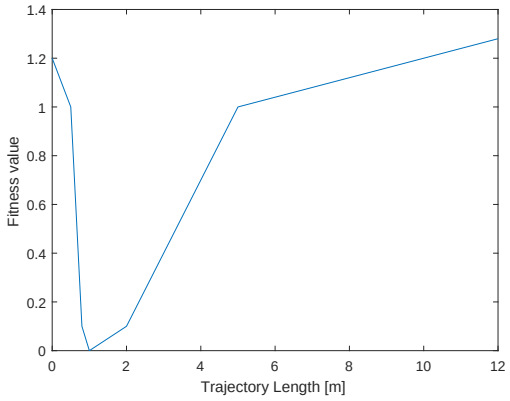
1.0000	2.0000	−31.5383	−0.1333	31.2716	1.0000
1.0000	3.0000	−34.7383	−3.3333	28.0716	1.0000
1.0000	4.0000	−37.0716	−5.6667	25.7383	0.9999
1.0000	5.0000	−43.1049	−11.7000	19.7049	0.9705
2.0000	3.0000	−34.6049	−3.2000	28.2049	1.0000
2.0000	4.0000	−36.9383	−5.5333	25.8716	0.9999
2.0000	5.0000	−42.9716	−11.5667	19.8383	0.9727
3.0000	4.0000	−33.7383	−2.3333	29.0716	1.0000
3.0000	5.0000	−39.7716	−8.3667	23.0383	0.9977
4.0000	5.0000	−37.4383	−6.0333	25.3716	0.9999

Appendix E

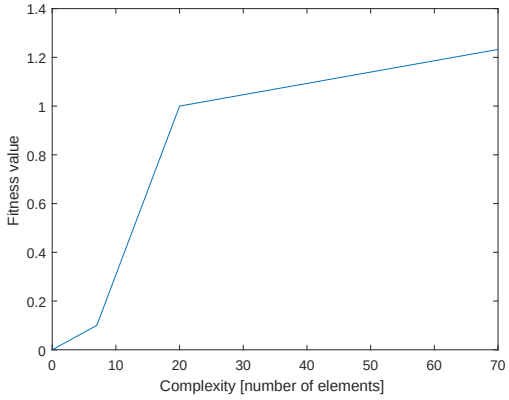
Plots from test-cases



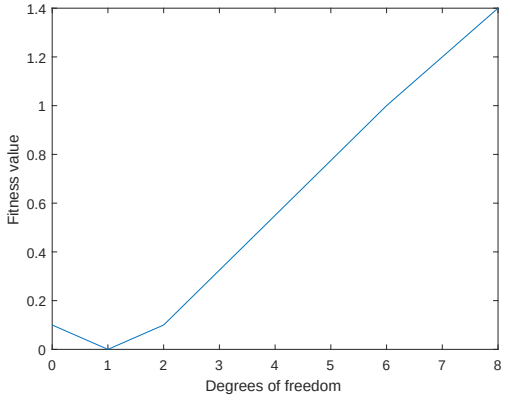
(a) Fitness function for the line curvature



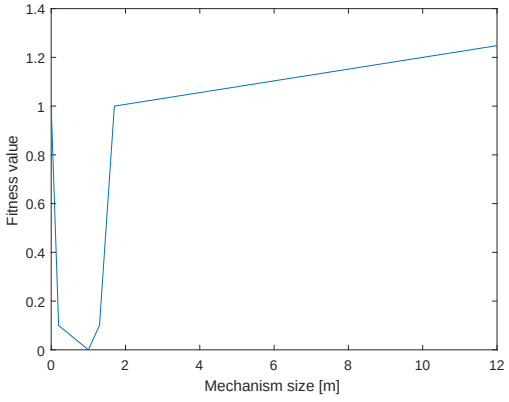
(b) Fitness function for the length of the line



(c) Fitness function for the complexity of the mechanism

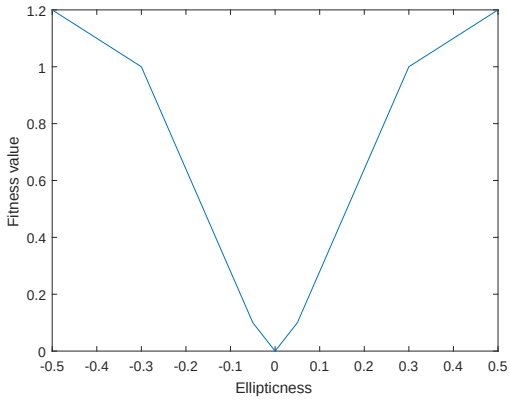


(d) Fitness function for the degrees of freedom of

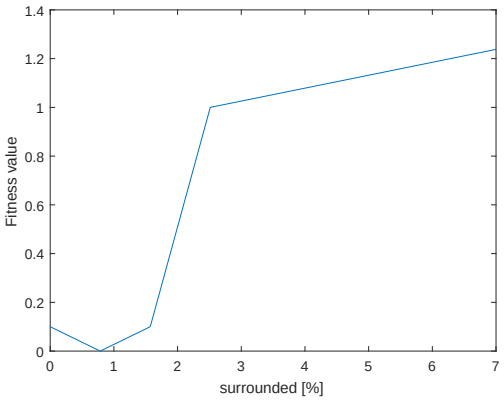


(e) Fitness function for the size of the mechanism in one dimension

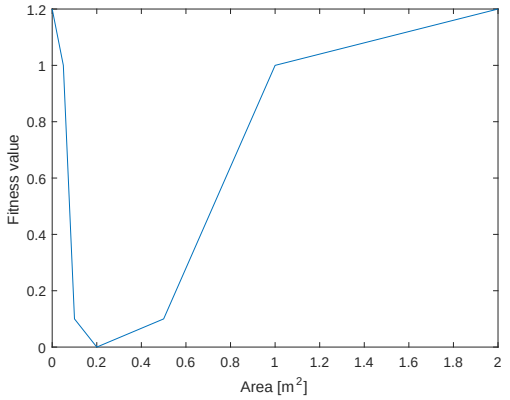
Figure E-1: Plots of the fitness functions for the straight-line problem used in first the test-case.



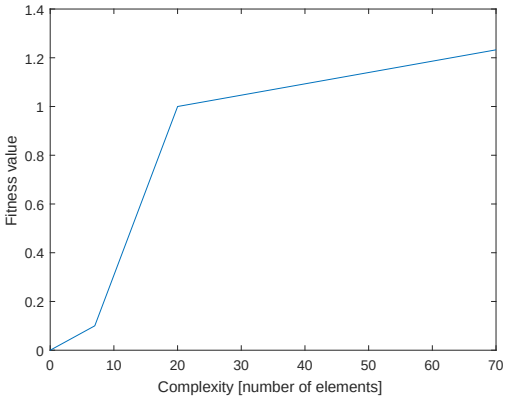
(a) Fitness function for the ellipticness



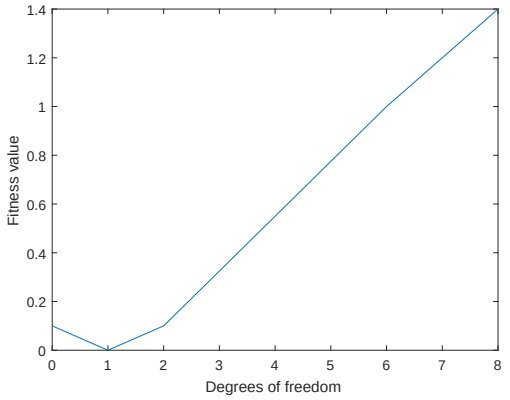
(b) Fitness function for the encircling of the center



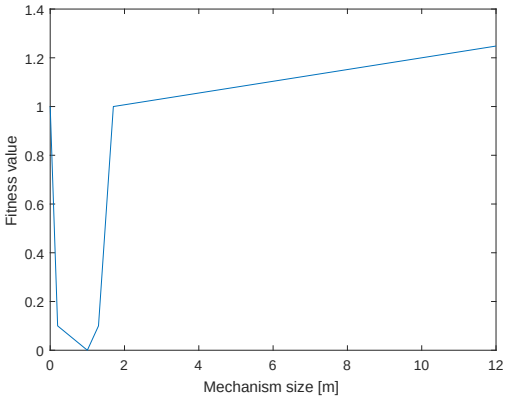
(c) Fitness function for the surface area of the data-points



(d) Fitness function for the complexity of the mechanism

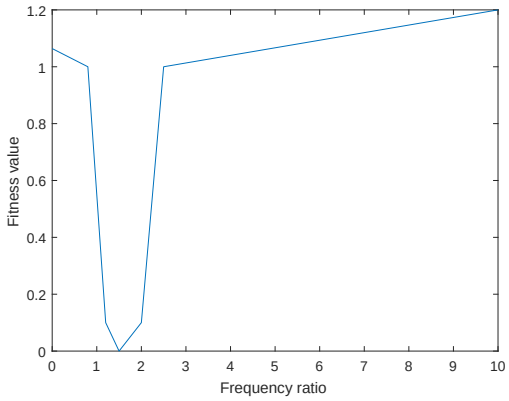


(e) Fitness function for the degrees of freedom of the mechanism

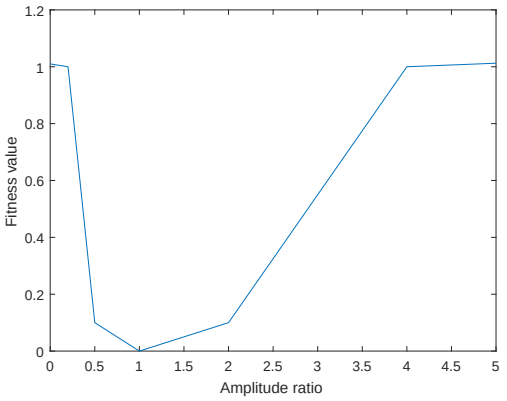


(f) Fitness function for the size of the mechanism in one dimension

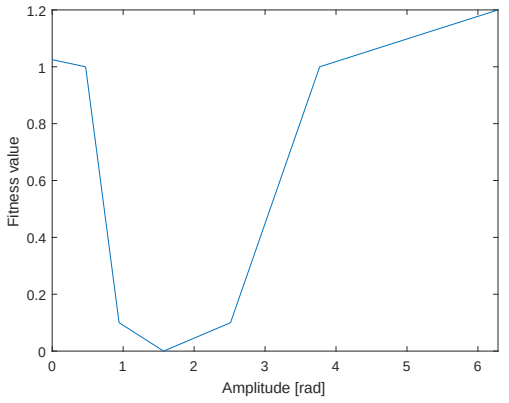
Figure E-2: Plots of the fitness functions for the elliptic trajectory problem used in first the test-case.



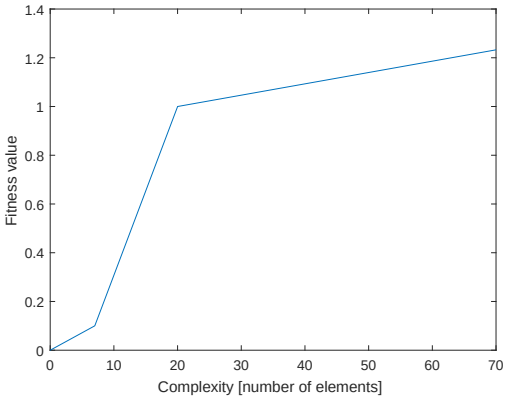
(a) Fitness function for the frequency ratio



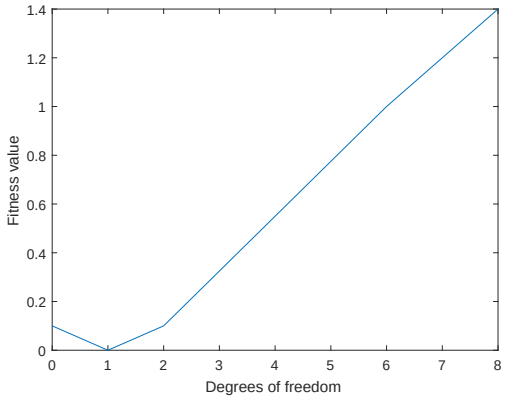
(b) Fitness function for the amplitude ratio



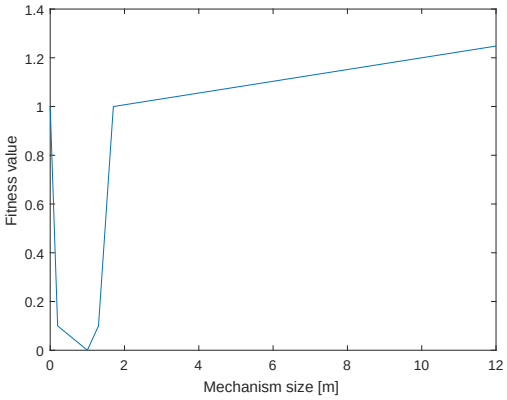
(c) Fitness function for the amplitude length



(d) Fitness function for the complexity of the mechanism



(e) Fitness function for the degrees of freedom of the mechanism



(f) Fitness function for the size of the mechanism in one dimension

Figure E-3: Plots of the fitness functions for the frequency multiplier problem used in third the test-case.

Bibliography

[1] M. Plooij and M. Wisse, "A novel spring mechanism to reduce energy consumption of robotic arms," *IEEE International Conference on Intelligent Robots and Systems*, pp. 2901–2908, 2012.

[2] M. Meggiolan and P. Saccardo, "Rim for a bicycle wheel and bicycle wheel comprising such rim," 2011.

[3] B. Y. B. Gunn and T. E. Peet, "Four Geometrical Problems from the Moscow Mathematical Papyrus," *The Journal of Egyptian Archaeology*, vol. 15, no. 3, pp. 167–185, 1929.

[4] R. J. Gillings, "Problems 1 to 6 of the Rhind Mathematical Papyrus," *The Mathematics Teacher*, vol. 55, no. 1, pp. 61–69, 1962.

[5] A. Purcell and J. Gero, "Drawings and the design process," *Design Studies*, vol. 19, no. 4, pp. 389–430, 1998.

[6] M. H. Morgan, *Vitruvius the Ten Books on Architecture*. 1914.

[7] S. Tornincasa and F. D. Monaco, "the Future and the Evolution of CAD," in *International Research/Expert Conference 'Trends in the Development of Machinery and Associated Technology'*, no. September, p. 18, 2010.

[8] J. R. Koza, M. A. Keane, M. J. Streeter, T. P. Adams, and L. W. Jones, "Invention and creativity in automated design by means of genetic programming," *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, vol. 18, no. 03, pp. 245–269, 2004.

[9] F. A. Firestone, "A new analogy between mechanical and electrical systems," *Acoustical Society of America*, vol. 4, pp. 249–267, 1932.

[10] K. Sims, "Evolving virtual creatures," *Siggraph '94*, vol. SIGGRAPH ', no. July, pp. 15–22, 1994.

[11] C. Leger, *Automated Synthesis and Optimisation of Robot Configurations: An Evolutionary Approach*. PhD thesis, 1999.

[12] I. C. Staal, *Evolutionary Mechinisms: Automated synthesis of robotic mechanisms by an Evolutionary Algorithm*. PhD thesis, Delft University of Technology, 2014.

[13] P. Kuppens, *Automated Robot Design With Artificial Evolution*. PhD thesis, Delft Univirsity of Technology, 2016.

[14] G. A. Hollinger and J. M. Briscoe, "Genetic optimization and simulation of a piezoelectric pipe-crawling inspection robot," *Proceedings - IEEE International Conference on Robotics and Automation*, vol. 2005, no. April, pp. 484–489, 2005.

[15] L. Brodbeck, S. Hauser, and F. Iida, "Morphological Evolution of Physical Robots through Model-Free Phenotype Development," *Plos One*, vol. 10, no. 6, p. e0128444, 2015.

[16] S. Balakirsky, S. Carpin, G. Dimitoglou, and B. Balaguer, "From simulation to real robots with predictable results: Methods and examples," *Performance Evaluation and Benchmarking of Intelligent Systems*, no. i, pp. 113–137, 2009.

[17] a. Boeing and T. Bräunl, "Leveraging multiple simulators for crossing the reality gap," *2012 12th International Conference on Control, Automation, Robotics and Vision, ICARCV 2012*, vol. 2012, no. December, pp. 1113–1119, 2012.

[18] E. Kranendonk, "A realistic approach to evolution based automated mechanism design," tech. rep., 2016.

[19] S. Kota and S. J. Chiou, "Conceptual design of mechanisms based on computational synthesis and simulation of kinematic building blocks," *Research in Engineering Design*, vol. 4, no. 2, pp. 75–87, 1992.

[20] A. S. HORNBY, A. P. COWIE, and J. W. LEWIS, *Oxford advanced learner's dictionary of current English*. London: Oxford University Press, 1974.

[21] L. C. Schmidt and J. Cagan, "GGREADA: A graph grammar-based machine design algorithm," *Research in Engineering Design*, vol. 9, pp. 195–213, 1997.

[22] G. Renner and A. Ekárt, "Genetic algorithms in computer aided design," *Computer-Aided Design*, vol. 35, no. 8, pp. 709–726, 2003.

[23] T. Back, U. Hammel, and H.-P. Schwefel, "Evolutionary computation: comments on the history and current state," *IEEE Transactions on Evolutionary Computation*, vol. 1, no. 1, pp. 3–17, 1997.

[24] I. Rechenberg, *Evolutionsstrategie Optimierung technischer Systeme nach Prinzipien der biologishen Evolution*. Frommann-Holzboog, 1973.

[25] T. Friedrich, P. Oliveto, D. Sudholt, and C. Witt, "Analysis of diversity-preserving mechanisms for global exploration*," *Evolutionary Computation*, vol. 17, no. 4, pp. 455–476, 2009.

[26] E. Alba and M. Tomassini, "Parallelism and evolutionary algorithms," *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 5, pp. 443–462, 2002.

[27] T. Krink, B. H. Mayoh, and Z. Michalewicz, "A Patchwork Model for Evolutionary Algorithms with Structured and Variable Size Populations," in *Proceedings of the 1st Annual Conference on Genetic and Evolutionary Computation*, vol. 2, pp. 1321–1328, 1999.

[28] G. W. Greenwood, G. B. Fogel, and M. Ciobanu, "Emphasizing extinction in evolutionary programming," in *Proceedings of the 1999 Congress on Evolutionary Computation, CEC 1999*, vol. 1, pp. 666–671, 1999.

[29] A. N. Godwin, "Simple calculation of moments of inertia for polygons," *International Journal of Mathematical Education in Science and Technology*, vol. 11, no. 4, pp. 577–586, 1980.

[30] H.-G. Beyer and B. Sendhoff, "Robust optimization - A comprehensive survey," *Computer Methods in Applied Mechanics and Engineering*, vol. 196, no. 33-34, pp. 3190–3218, 2007.

[31] E. Kranendonk, "Automated mechanism design," tech. rep., 2016.

[32] G. Taguchi, *Introduction to Quality Engineering*. 1989.

[33] S. Tsutsui, A. Ghosh, and Y. Fujimoto, "A robust solution searching scheme in genetic search," *Parallel Problem Solving from Nature*, vol. 1141, pp. 543–552, 1996.

[34] Y. Jin and B. Sendhoff, "Trade-off between performance and robustness: An evolutionary multiobjective approach," *Evolutionary Multi-Criterion Optimization*, vol. 2632, pp. 237–251, 2003.

[35] H.-G. Beyer and B. Sendhoff, "Evolution Strategies for Robust Optimization," in *2006 IEEE International Conference on Evolutionary Computation*, pp. 1346–1353, 2006.

[36] A. Thompson, "Evolutionary techniques for fault tolerance," in *International Conference on CONTROL*, no. 427, pp. 2–5, 1996.

[37] A. Thompson and P. Layzell, "Evolution of robustness in an electronics design," *Evolvable Systems: From Biology to Hardware, Proceedings*, vol. 1801, pp. 218–228, 2000.

[38] A. B. Owen, *Monte Carlo theory, methods and examples*. 2013.

[39] L. and fits Technical Committee ISO/TC 3, "NEN-ISO 2768-1," tech. rep., 1990.

[40] M. Stojmenović, A. Nayak, and J. Zunic, "Measuring linearity of a finite set of points," in *2006 IEEE Conference on Cybernetics and Intelligent Systems*, pp. 222–227, 2006.

[41] A. Field, *Discovering Statistics Using IBM SPSS Statistics*. 2013.

[42] P. C. Fishburn, *Nonlinear Preferences And Utility Theory*. Prentice Hall / Harvester Wheatsheaf (July 1988), 1988.

[43] W. Fan, E. A. Fox, P. Pathak, and H. Wu, "The Effects of Fitness Functions on Genetic Programming-Based Ranking Discovery For Web," *Journal of the American Society for Information Science and Technology*, vol. 55, no. 7, pp. 628–636, 2004.

[44] S. Coros, B. Thomaszewski, G. Noris, S. Sueda, M. Forberg, R. W. Sumner, W. Matusik, and B. Bickel, "Computational Design of Mechanical Characters," *ACM Trans. Graph.*, vol. 32, no. 4, pp. 83:1–83:12, 2013.

[45] M.-P. Dubuisson and a.K. Jain, "A modified Hausdorff distance for object matching," *Proceedings of 12th International Conference on Pattern Recognition*, vol. 1, no. 1, pp. 566–568, 1994.

[46] H. Alt and M. Godau, "Computing the Frechet Distance between two Polygon Curves," *International Journal of Computational Geometry & applications*, vol. 5, no. 1-2, pp. 75–91, 1995.