Faculteit Ingenieurswetenschappen
Vakgroep Civiele techniek
Voorzitter: Prof. Dr. Ir. J. De Rouck

Combining genetic algorithms and boundary elements to optimize coastal aquifers' management using sheet pile walls

door

Koen Wildemeersch

Promotoren:
Prof. Dr. Ir. K. L. Katsifarakis (AUTH),
Prof. Dr. Ir. H. Peiffer (UGENT)

Scriptie ingediend tot het behalen van de academische graad van
master in de ingenieurswetenschappen bouwkunde
optie water- en transport

Academiejaar 2009–2010

This page intentionally left blank

Combining genetic algorithms and boundary elements to optimize coastal aquifers' management using sheet pile walls

Koen Wildemeersch

Promotoren: prof. dr. ir. Herman Peiffer, Kostas Katsifarakis
Begeleider:

Masterproef ingediend tot het behalen van de academische graad van
Master in de ingenieurswetenschappen: bouwkunde

Vakgroep Civiele techniek
Voorzitter: prof. dr. ir. Julien De Rouck
Faculteit Ingenieurswetenschappen
Academiejaar 2009-2010

UNIVERSITEIT
GENT

Foreword

Six years ago I had to make the decision whether to study computer or civil engineering. I decided to go for the latter, but found out that combining both fields of engineering is achievable and where most people do not agree, even very interesting. When professor Katsifarakis suggested to combine both worlds as a thesis, I did not have to think twice. This was exactly what I wanted.

This thesis was written while I was an Erasmus student in Greece. In total, I will have lived 10 months as a Greek (with a slightly different background) and now call this country my home away from home. I feel compelled to first thank my Greek friends. They made me feel at home, showed me good and special places, explained me their political problems, helped me out where my language skills where not sufficient, and so much more. Without them Erasmus would not have been as good an experience. *Efgaristo!*

Erasmus is in my opinion a really a great experience and I would strongly advise everybody to do it. It opens your eyes: new insights, a new culture, meeting a lot of people from all over the world. I consider myself very lucky with my flatmates and I want to thank them: Alex from France, Mari from Estonia, Jaime from Columbia and Xu from China. I cannot imagine a more diverse and interesting company. Together we lived our own '*Auberge Espaniol*'. Thank you for showing me your culture and sharing your friendship. *Merci, Aitäh, Graçias, Xie Xie!*

Writing a thesis is never a work done all by oneself. I especially want to thank my promoter professor Katsifarakis. My greek friends told me I had to consider myself lucky with this professor as a promoter and they where right. Thank you for sharing your knowledge and experience in the topic in such a modest and friendly way. Thank you as well for letting me go my own way and working out my own ideas. Next to academic help I also want to thank professor Katsifarakis for explaining and showing me his country. The help and information I got went much further than what was strictly necessary for my thesis alone. *Efgaristo para poli!*

I also want to thank the Aristotle University of Thessaloniki for accepting me as an Erasmus student, and Ghent University (Universiteit Gent) for accepting the Erasmus proposal. I also want to thank the Greek and Belgian Erasmus office. Being an Erasmus student brings along some extra issues and without the help received it would not have been possible. Thank you professor Peiffer (Ugent) to mentor my thesis. *Efgaristo, Bedankt!*

During the first month of my Erasmus exchange I attended a Greek language course at the University of Aegean, school of social sciences, on Mytiline island. Together with 25 other

people from all over Europe we learned the basics of the Greek language. Thank you Roula for teaching us! *Efgaristo poli!*

This was the third time I wrote a thesis and it is as a consequence the third time that I need to thank my parents. Without them none of this would have been possible in the first place. *Merci!*.

Writing in a language that is not your own brings along some problems, as does writing in general. Thank you Richard (United Kingdom) for going through my text and correcting the uncountable mistakes. Thank you Nikos (Greece) for reading my text from the point of view of an engineer. And thank you Mari (Estonia) for reading my text and giving my information about genetics. *Thank you, Efgaristo, Aitäh!*

The figures in this LaTeXthesis are all vector figures and I want to thank Ibe (Belgium) for his contribution. *Bedankt!*

I want to end with my life motto: ***Vive la vie en rose (Edith Piaff)***

Koen Wildemeersch
Thessaloniki
April 29, 2010

Copyright

Koen Wildemeersch, april 2010

Combining genetic algorithms and boundary elements to optimize coastal aquifers' management using sheet pile walls

Koen Wildemeersch

Supervisor(s): Kostas Katsifarakis, Herman Peiffer

Abstract— **This master's thesis combines genetic algorithms with a boundary element method that calculates the flow in a coastal aquifer. The goal of doing so is to optimize the total pumped flow of fresh water from the aquifer without sea water intrusion taking place. In order to improve the volume of water pumped, a sheet pile wall can be placed.**

Keywords—**Genetic algorithm, boundary element method, optimization, sheet pile wall, water management**

I. Introduction

CLIMATE change and human intervention have lead to a lack of fresh water. Fresh water can be found underground and extracted, but when this aquifer is close to the sea, special care should be taken not to create an inflow of saline water in the aquifer by extracting to much. This would eventually turn the fresh water into saline water, making the aquifer unusable for the extraction of fresh water. A good management of the aquifer is therefore required and it should be clear how much water can be extracted from the aquifer without having seawater intrusion.

One technique to calculate the flow in aquifers is to use a boundary element method. In this thesis the boundary element method will be used by a genetic algorithm to optimize the extracted flow from the aquifer by placing a sheet pile wall on the coastline. The genetic algorithm is used to find out what the best combination of a sheet pile wall and water extraction from different wells is. The algorithm designed is written in C#, and a pre- and post processor were designed so the user does not need to know any input syntax.

II. Theoretical background

A. Genetic algorithm

A genetic algorithm is a search and optimization technique based upon Darwin's theory of the survival of the fittest. A population of candidate solutions, represented each by a chromosome, is generated and their fitness is calculated. Based upon the fitness each chromosome is assigned, it has a different probability to be selected and to go to the next round. Just as with real chromosomes, they can undergo changes from one generation to another. Chromosomes in this thesis can undergo crossover mutation and antimetathesis with a constant or a linear probability. Selecting can take place in three ways: roulette wheel selection, ranking and constant selection. The changes made to the chromosome may result in a higher fitness function which give it a higher chance to survive. The algorithm is also designed in such a way that all variables can have their own subchromosome length.

The idea is that after a certain amount of generations the fittest chromosome dominates the population and the optimum candidate solution is found. To achieve this the fitness awarded to each chromosome is very important. The choice of the fitness function is hence very important and crucial to find very fit solutions. The chromosomes used in this thesis are represented by a binary, i.e a string of 1 and 0's. For every binary the integer value can be calculated and from that a double value is calculated knowing the upper and lower double value for the chromosome.

In order not to lose the fittest chromosome due to selection, crossover, mutation or antimetathesis, elitism is used to make sure that the fittest chromosome passes to the next generation without undergoing changes.

B. Boundary element method

The boundary element method is a technique used to solve differential equations of a function u, only knowing what are the conditions on the boundary of the the domain u is valid on. In this thesis the differential equation is the Poisson equation $\nabla^2 u = f$ which governs the flow in a homogeneous aquifer.

This thesis starts with the mathematical background needed in order to solve the differential equation and how to transform its analytical solution to a numerical solution that can be used for computation. The boundary of the domain is therefore discretized into a chain of boundary elements on which the boundary conditions are assumed to be constant.

The use of a boundary element method is very effective for adding the influence of wells and specific for this thesis the use of a sheet pile wall will be included in the boundary element. The boundary element method that is developed can be used for multiple boundary domains (multiple zones) with a constant transmissivity in each zone and for constant boundary conditions on the elements.

C. Combining both

The fitness function required for the genetic algorithm will be calculated by the boundary element method. This approach has been used before and is said to be the perfect marriage [1] by Harrouni, Ouazar et. al. It is correct to say that the genetic algorithm uses the boundary element method. The genetic algorithm will create chromosomes representing the flow rate extracted from wells and the beginning and end point of a sheet pile wall on the coastline. The double values of these chromosomes will be used as input for the boundary element method and with the results of the boundary element method a fitness function will be calculated. This fitness function uses the seawater intrusion calculated. When a lot of seawater intrusion was calculated the fitness will be low and vice versa.

D. Implementing a sheet pile wall

A sheet pile wall is a piece of the coastline were no inflow is allowed: $u_n = 0$. Implementing a sheet pile wall means that the user input needs to be modified. This is done by allowing the genetic algorithm to change the input data for the boundary element method. The sheet pile wall can start at a random point on the coast so it is not clear if the beginning and endpoint of the sheet pile wall will be the same as the boundary elements. To resolve this problem new boundary elements can be created and existing can be added.

E. Reducing the calculation work

During the test phase of the algorithm it became clear that some possible improvement could be made to prevent recalculating what had been calculated before, and thus reducing the calculation time and work. A first measurement was to store the fitness of chromosomes that had been calculated. When the same chromosome occurred for a second time its fitness could be read from the memory without going through the boundary element method again. When the chromosome had not yet been generated it could be that the coordinates of the wells had been calculated before. If so, the zone were the well was in would be stored and related to this set of coordinates. Especially in the case were the wells have a fixed position this leads to a very high calculation reduction.

Next to that, more calculation reduction was achieved by sorting the arrays used in the boundary element in such a way that parts of the arrays never needed to be calculated again.

III. Reliability of the designed algorithm

In a first step the boundary element method was designed without a sheet pile wall. For this algorithm a lot of school book examples are available and the solutions obtained with the algorithm were compared with the examples from the book. The results were satisfying.

In a second step, a genetic algorithm was developed. This algorithm was first tested for simple fitness functions that did not use the boundary element method. The algorithm did as was to be expected and in a third step the boundary element method and the genetic algorithm were combined. The candidate solutions obtained from the combined use where then compared to the results obtained via the traditional solving way (calculating each candidate solution).

In a last step the use of a sheet pile wall was implemented. This made it possible to change the user input of the boundary elements based upon the chromosome calculated by the genetic algorithm.

IV. Objectives

Originally three objectives were formulated. The first was to calculate the best combination of fresh water extraction through two wells with fixed coordinated for a given aquifer and known boundary conditions. This objective was set because the results could then be compared to that of Dr. Petala [2], who had studied this in here doctoral thesis. This objective was thus set to be sure that the algorithm worked in the way it was supposed to work.

The second objective was to include a sheet pile wall and see what the effect was on the maximum flow that could be extracted.

In a third and last objective the genetic algorithm was combined with the boundary element method that allowed the placement of a sheet pile wall, in order to optimize the aquifer. These last two objectives were taken together and are discussed in detail.

V. The aquifer studied

The aquifer studied in this thesis was studied before in the doctoral thesis of Dr. Petala [3]. It exists out of two zones with a different tranmissivity as depicted in figure (1). In zone 1, $T_1 = 0.001$ m/s and $T_2 = 0.003$ m/s in zone 2. Boundary AB represents the coastline (on which the sheet pile wall can be placed) and has a constant head boundary of $u = 0$ m. Lines ADF and BCE represent two impermeable boundaries $u_n = 0$ and line FE is a permeable boundary that provides inflow of fresh water due to the natural elevation: $u = 50$ m. u is the head and u_n the flux.

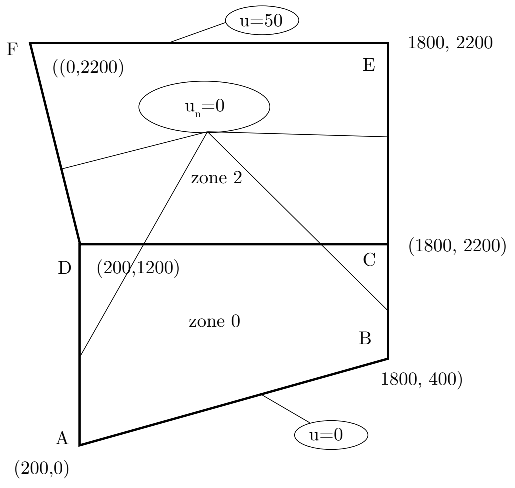

Fig. 1. Aquifer studied

VI. The fitness function used

For all the objectives one and the same fitness function were used. The fitness function used was designed for the first objective of this masters thesis and the doctoral thesis of Dr. Petala.

$$\Phi_K = \sum_{i=1}^{W} q_{w,i} - (70 \cdot \kappa - 7 \sum_{i=1}^{\kappa} T_i \cdot u_{n,i} \cdot l_i) \qquad (1)$$

In this function W is the number of wells (-), $q_{w,i}$ the flow in well i (m^3/s), κ the number of boundary elements that have sea water intrusion (-), T_i the transmissivity of zone i (m/s), $u_{n,i}$ the calculated flux for boundary element i (m^3/s) and l_i the length of the boundary element (m). The last summation is made for all $u_{n,i} > 0$, which represent inflow.

VII. Results

A. Objective one: Optimization of two wells with fixed coordinates

The results obtained from the algorithm could be compared to those of Dr. Petala's doctoral thesis [3]. In this thesis two wells were placed in the same zone: $W_1 = (500, 700)$ and $W_2 = (1400, 700)$. The best combination was then calculated to be $Q_1 = 0.031$ m^3/s and $Q_2 = 0.038$ m^3/s.

Here, two combinations of equal fitness (for a precision step of 0.00001 m^3/s) were found: $Q_1 = 0.03129$ m^3/s, $Q_2 = 0.03829$ m^3/s and $Q_1 = 0.03135$ m^3/s, $Q_2 = 0.03823$ m^3/s. The fitness for both solutions was 0.06958. The results were thus very satisfactory. The fact that two chromosomes showed to be as fit can be explained by the discontinuous search space and the fact that for both subchromosomes (Q_1 and Q_2) had the same length and the same upper and under values were used.

B. Objective two and three: Implementation of a sheet pile wall

Before running the algorithm, a set of good input parameters for the genetic algorithm was researched. Different factors were tested for the following input data: $PS = 50, NOG = 100, NOT = 10, P_c = 0.35, P_m = P_f = 0.06, \epsilon = 1$ and mutation and antimetathesis both took place in every generation. The sheet pile wall had a length of 1000 m. (PS = population size, NOG = number of generations, NOT = number of trials, P_c, P_m, P_f the crossover, mutation and antimetathesis probability, resp.)

A first parameter tested was the selection type used. Constant selection with a constant of 4 showed to be the best choice, bused upon the memory size and the required calculation time that showed to be the smallest. The number of fittest solution found was also the biggest using this selection technique.

A small test was made where mutation and antimetathesis could take place one per chromosome or once per gene. Once per gene showed not to be sufficient to find good results. On the other hand allowing mutation and antimetathesis for every gene proved to be much better.

The influence of the population size and the number of generations was considered. Increasing the population size did not result in finding extra fit solutions. Increasing the number of generations resulted in a few more fittest solutions found. Because only few extra were found and the number of trials increased by 50, the decision was made not to increase the number of generations carried out.

The second last parameter tested was to use mutation and antimetathesis interchangingly or not. Interchanging use resulted in less fit solutions found. The memory size was also smaller which indicated that the solution area was not searched enough. When for every generation, first mutation and then antimetathesis took place, the results proved to be better. There for mutation and antimetathesis was used in the last way.

The last parameter researched was called refreshment. An analysis of the fitness evolution had shown that the fitness sometimes not increased for a very long time. Therefore the idea was to inject new chromosomes in the population in the hope that they would lead to fitter chromosomes in the next generation. Three different injections were carried out: in a first a number of randomly populated chromosomes were added to the population size (similar to ranking). When refreshment took place soon after stabilization of ϕ, the number of fittest chromosomes found decreased. Allowing the algorithm more time before refreshing did not improve the results, but only caused more calculations to be carried out. The idea was then to refresh with highly fit chromosomes from the last generation. They would first be mutated or would first undergo antimetathesis with a probability of 100% in only one of the genes.The results found were less fit. Therefore the idea of refreshment was not used.

After having studied the settings for the genetic algorithm, the algorithm could be used to calculate objective 2 and 3. 5 different sheet pile wall lengths were studied = 200, 400, 600 and 800 m. For long sheet pile walls two groups of solutions seemed to be calculated. A first protected W_2 by placing the in front of this well. This lead to an increase of Q_2, but Q_1 was generally found to be less than was calculated in objective 2. The second group of solutions placed the sheet pile wall in between the two wells. Doing so both could extract more water from the aquifer. The first group was found to be always fitter than the last group.

For shorter sheet pile walls all runs point out that the sheet pile wall always protects W_2. There was a very clear relation between the length of the sheet pile wall and the total flow extracted: longer sheet pile walls lead to more extracted water without sea water intrusion.

C. Comparison to one extra well

In a last test, it was researched if it was possible to obtain the same improvements by using a third well , $W_3 = (1050, 750)$, instead of a sheet pile wall. The best result calculated were: $Q_1 = 0.0281, Q_2 = 0.0319, Q_3 = 0.0113$ m^3/s and the total flow rate was 0.07129 m^3/s. This result was only better compared to the use of a sheet pile wall of 200 m.

References

[1] K. El Harrouni, D. Ouazar, et al., *Groundwater: Boundary Element Techniques in Geomechanics*, eds G.D. Manolis & T.G. Davies. CMP/Elsevier, Amsterdam, 1993, pp. 243-94.

[2] K.L. Katsifarakis and Z. Petala, *Combining genetic algorithms and boundary elements to optimize coastal aquifers' management*, Journal of Hydrology. Elsevier, Amsterdam, 2006, pp. 200-207, doi. 10.1016/j.jhydrol.2005.11.016.

[3] Z. Petala, *Optimizing management of coastal aquifers by means of genetic algorithms*, (in Greek) PhD thesis, Department of Civil Engineering, Aristotle University of Thessaloniki, Greece, 2004. 260 pp.

Contents

Nomenclature and abbreviations

Nomenclature used for genetic algorithm

KK	Selection constant (-)	X	chromosome
G_{max}	integer that hold the run where the maximum was found (-)	Φ	fitness function
NOG	Number of generations (runs) (-)	ϕ	fitness
NOT	Number of trials (-)	ϕ_{ave}	average fitness
NOV	Number of variables (-)	ϕ_{min}	min fitness
P	Probability (-)	ϕ_{max}	maximum fitness
P	Double value of chromosome	ϕ_{off}	offline fitness
P_{max}	Maximum double value of chromosome	ϕ_{on}	online fitness
P_{min}	Minimum double value of chromosome	ΔP	double difference between two chromosomes
P_c	crossover probability (-)	λ	chromosome length (-)
P_m	mutation probability (-)	μ	constant used for calculating P
$_e$	end	ν	constant used for calculating P
$_b$	beginning	Σ	convergence velocity (-)
PS	population size (-)	Z	integer value of chromosome (-)
γ	generation (-)	$'$	after crossover
γ	Number of generations (runs) (-)	$''$	after mutation
χ	gene of chromosome	$\oplus$	concate

Nomenclature used for the boundary element method

A	array	spw_b	begin of the sheet pile wall (m)
B	array	spw_e	end of the sheet pile wall (m)
B_t	array	l_c	length of the coast (m)
$_c$	coastal	T	transmissivity (m/s)
$f(x,y)$	real function	T	transpone (matrix algebra)
$_f$	fixed	$\vec{u}$	vectorfield u
$g(x,y)$	real function	u	potential (m)
G_{ij}	array	u_n	$= \partial u / \partial n$, flux
$h(x,y)$	real function	$_w$	well
H_{ij}	array	w_k	weight factor (-)
$\hat{H}_{ij}$	array	x	first dimension of search area

h_{ij}	element of H (row i, column j)	y	second dimension of search area
$\vec{i}$	unit vector x axis	$x^{'}$	x coordinate in local axis system
$\vec{j}$	unit vector y axis	$y^{'}$	y coordinate in local axis system
k	number of colums in B_t matrix (-)	α	angle (rad)
l_j	length of boundary element (m)	β	angle (rad)
l	arch length (m)	∂	Dirac delta function
ln	natural logarithm	ϵ	radius (m)
m	number of unknown on the coastline (-)	Γ	boundary of surface Ω
n	number of colums in A matrix (-)	η	y coordinate of Q
$\vec{n}$	normal vector	Θ	angle (rad)
n_x	projection of $\vec{n}$ on the x axis (m)	κ	number of boundary lines with seawater intrusion
n_y	projection of $\vec{n}$ on the y axis (m)	Ω	domain
N	integer value representing a number (-)	ξ	x coordinate of Q
$P(x,y)$	source point	∇	$\frac{\partial}{\partial x}\vec{i}+\frac{\partial}{\partial y}\vec{j}$
$Q(x,y)$	density	∇^2	$\frac{\partial^2}{\partial x^2}+\frac{\partial^2}{\partial y^2}$
q	flow rate (m^3/s)	$\frac{\partial}{\partial n}$	$=\frac{\partial}{\partial x}n_x+\frac{\partial}{\partial y}n_y$
r	distance between two points (m)	$\wp$	delta Dirac function for well influence
s	path followed (m)	$\|\|\|\|$	norm (m)
		$\bar{\ }$	known

Nomenclature discussing the objectives

A	array	spw_b	begin of the sheet pile wall (m)
B	array	spw_e	end of the sheet pile wall (m)

Used abbreviations

BEM	Boundary Element Method	FEM	Finite Element Method
GA	Genetic Algorithm	FF	Fitness Function
RW	Roulette Wheel selection	C	Constant selection

Chapter 1

Introduction and objectives

Given a setup of wells that pump fresh water from an aquifer near the coastline, it will be studied how to increase the total freshwater flow pumped, without the intrusion of saline water, by using sheet pile walls.

The approach here is not to do field experiments but only to do a theoretical study. This study will be carried out by using a genetic algorithm that finds the best place for the sheet pile wall. By placing a sheet pile wall, seawater intrusion is hindered and more fresh water might be extracted. Interesting questions here are: 'How much more can be pumped by placing a sheet pile wall?', 'Where is the optimal location of the sheet pile wall?' and 'What is the best solution? Placing a sheet pile wall or installing an extra pump?'. To all these questions a theoretical solution will be researched.

In order to use a genetic algorithm to compute the optimization by a sheet pile wall, it is first necessary to find out what is the relation between the total flow pumped and the seawater intrusion. This relation will be calculated via a boundary element method. A simple computer algorithm program will be developed that can calculate the seawater inflow through the coastline border. Given a set of wells (their location and flow) the program will calculate the flow conditions at the coastal border. If there is inflow of saline water into the aquifer then the total flow pumped should be lowered. Theoretically, the best solution is found when there is zero inflow through the coastline.

The algorithm then needs to be extended so that it includes a sheet pile wall. It will then be possible to compute how much more fresh water can be pumped without having salt intrusion.

Using this algorithm, a genetic algorithm could then be developed to find the best optimization possible, i.e. the best location and length of the sheet pile wall in combination with the highest flow extracted. Combining the boundary element method with a genetic algorithm creates thus a powerful optimization tool. When adequate fitness functions are used it is possible to find the best combination in a minimum of time.

Three case studies will be made. In the first, the maximum flow pumped will be calculated without having seawater intrusion. The locations of the wells are constant but the flow pumped is variable and will be optimized. In the second case a sheet pile wall will be placed on the coastal border and its influence will be calculated. It will be computed how much flow increase this wall initiated and at what cost. In the third and final case the use of a sheet pile wall will be optimized. The best possible location and length will be computed, so that the flow pumped is maximal.

Chapter 2

Genetic algorithms

2.1 Genetic algorithms versus traditional solution finding

In this master thesis the traditional way of finding the (optimum) solution for a problem is left behind. Instead of calculating the solution in the range of all variables, an algorithm will be used that finds its own way to this (optimum) solution without calculating all the values.

The use of genetic algorithms (GA) became more important over the last few decades. On the moment of writing this thesis, GAs are not included in the education of civil engineers. For that reason a brief overview of the used terminology will be given. A lot of GAs might be developed, from very simple to what is called more complex. The GAs developed in this thesis are of both kinds and are also generation depended. This means they will change from generation to generation. GAs are used in a lot of domains but especially here they will be used to optimize the setup of wells and sheet pile walls.

GAs are mostly used in large solution spaces where calculating all candidate solutions would take a long time. It offers an alternative that does not need the computation of all candidate solutions and it is furthermore accepted to be efficient when the space is not perfectly smooth and unimodal. This means that there is not one (or more) smooth hill(s) where the best solution could take place. This is the case for both objectives two and three. If it would be clear beforehand where the solutions are concentrated it is probably not worth a GA. It is clear that in a homogeneous zone with only one well and very simple dimensions the use of GAs might be less interesting compared to the traditional approach of calculating the value of the unknown in a certain amount of points. When on the other hand the zone is divided into different subzones with their own transmissivity, T, the dimensions are irregular and there is more than one well, it might be less obvious how to find the best solution.

It should be clear that a genetic algorithm is not the best way to find the absolute optima, but should be used to find the near absolute optima. When the absolute optima is found the traditional approach can be used to find the absolute optima.

2.2 How do genetic algorithms work: analogy to natural genetics

Implementing GAs is using Darwin's theory on *survival of the fittest* to solve real life problems. The idea is that *generation* after generation the strongest species have the highest chances to survive. Each generation starts with a *genotype* that is selected by chance and that is modified, also by chance. This will most probably result in a change in its *phenotype*. Each generation ends after the phenotype is created. If the newly created chromosome is fitter, then it's chances to resist the dangers of its *environment* are higher. This chromosome is likely to survive and reproduce. It's *offspring* will most probably have this good change as well and will thus themselves have more chance to survive. They are, what Darwin called, *fitter*. Through evolution, the genotype will constantly change, and when to the better it will have more chances to survive. After a number of generations, called a *run*, the fittest genotypes should statistically dominate the less fitter ones which causes the latter to extinct.

Applying this idea to the problem of optimization means that a random population of solutions is selected and a *fitness function* is calculated for each one of them. The higher the fitness value, the higher the survival chances of the solution for the next generation. After a certain run the best solution is then likely to come forward.

2.3 Chromosomes and the binary system

A change in the genotype is in medical terms a change in the *chromosome*. Chromosomes are basic building stones and when some changes takes place in it it will change the genotype. A chromosome is here defined as a string of digits that represents one of the variables of the problem. Here, it might be the begin-coordinates of the sheet pile wall, the length of it, or the inflow in a well.

Chromosomes, although not necessary, will here be represented as a binary string. That is 0's and 1's. An example of a chromosome, X_1, might then be:

$$X_1 = 10010101001 \tag{2.1}$$

This binary represents an integer, Z_1, and the value is calculated as followed: Starting to count from the last position of the string towards the beginning:

$$Z_1 = (\text{int})X_1 = \sum_{\iota=1}^{\lambda} \left[(\chi(\iota)) \cdot 2^{\iota-1} \right] \tag{2.2}$$

Where (int) represents the integer value (in programming terminology this is called casting the binary) of chromosome X_1. λ is the number of digits χ in the chromosome. λ is 11 for X_1. The integer value of X_1 is thus:

$$Z_1 = 1193 \tag{2.3}$$

The unknowns in our problem are actually not integers but doubles (double precision). In order to work with doubles a technique called *linear mapping* is used. A real number, P, is transformed from a 10-base integer, Z, which had been transformed from a binary string, X, calculated before:

$$P = \mu Z + \upsilon \tag{2.4}$$

Z is calculated from X according eq. (2.2). μ and υ depend upon the location and the width of the space the solution is searched in and they are derived from the minimum and maximum values of P. Consider for example the sheet pile wall what will be used later on. This sheet pile wall will start between two real coordinates, P_{min} and P_{max} on the coastline. For both points, equation 2.4 can be written:

$$P_{min} = \mu Z_{min} + \upsilon \tag{2.5}$$

$$P_{max} = \mu Z_{max} + \upsilon \tag{2.6}$$

Keeping in mind that $X_{min} = 000000...$ and $X_{max} = 111111...$ it is then clear that $Z_{min} = 0$ and $Z_{max} = 2^{\lambda} - 1$. In eqs. (2.5) and (2.6) only μ and υ are unknown and can thus be derived. Their solution yields:

$$\mu = \frac{P_{max} - P_{min}}{2^{\lambda} - 1} \tag{2.7}$$

$$\upsilon = P_{min} \tag{2.8}$$

Knowing this eq. (2.4) becomes:

$$P = \left(\frac{P_{max} - P_{min}}{2^{\lambda} - 1}\right) Z + P_{min} \tag{2.9}$$

When for example the sheet pile wall can have coordinates between 10 m and 150 m, then X_1 would represent the real number P_1 as:

$$P = \left(\frac{150 - 10}{2^{11} - 1}\right) \cdot 1193 + 10 = 91.59 \tag{2.10}$$

The longer X is, the smaller the step between the double value of two chromosomes, ΔP, will be. Indeed, eq. (2.9) is not a continuous function and the collection of double values it depicts is not as well. Finding a good value for λ is thus finding a good balance between the accuracy required and the total calculation time of the GA. When λ is too low the optima might never be found because it can never be accessed.

The step between two chromosomes, $\Delta P = P_i - P_{i-1}$, will be the starting point to decide how long a chromosome should be:

$$\Delta P = \left(\frac{P_{max} - P_{min}}{2^{\lambda} - 1} \right) \tag{2.11}$$

For example, when looking for an optimal position of a sheet pile wall between two points on the coast, $A = 0$ m and $B = 500$ m, and the result should at least be precise on one meter the minimum chromosome length, λ_{min}, is calculated from:

$$\lambda_{min} \geq \frac{\ln \left(\frac{P_{max} - P_{min} + \Delta P}{\Delta P} \right)}{\ln 2} \qquad , \frac{P_{max} - P_{min} - \Delta P}{\Delta P} > 0 \tag{2.12}$$

When $\lambda = 8$, $\Delta P = 1.96$ m and the precision is not yet high enough. For $\lambda = 9$, $\Delta P = 0,98$ m, which then meets the required precision. $\lambda_{min} = 9$.

2.4 Operators

2.4.1 Selection

For every chromosome of the population a fitness function will be calculated. Based upon the individual fitness, and compared to the other fitness of the other chromosomes, a set of new chromosomes will be selected to go to the next generation.

The algorithm developed can select with three different selecting techniques: Roulette wheel selection, ranking and selection constant. The general idea of the method is explained. For the mathematical translation the reader is referred to the code in the back of this writing.

Roulette wheel

Roulette wheel selection is usually compared to the well known roulette game. A wheel is spun, and the numbered segment in which the ball comes to rest is the winning segment. The idea here is that the boxes become bigger with increasing fitness. Fitter chromosomes have a higher chance of being selected and hence to continue to the next round.

Ranking

Using ranking, all chromosomes are ordered according their fitness. The chromosome with the highest fitness is on the first place and the rest are ranked with descending fitness. From this list a certain percentage goes to the next generation and the other percentage is refreshed with new chromosomes. This method has the advantage of passing all the best solutions and inputting new chromosomes during all the generations. Operators like crossover and mutation (see later) are then only applied on a smaller group, which may result in not fine tuning the optimum solution.

Tournament selection

A number of chromosomes, KK, is selected with equal probability: $1/PS$. From this KK chromosomes, the fittest chromosome is passed to the next generation. In the first selection of KK chromosomes the fittest and the less fittest chromosome have equal probabilities of being selected. It is thus not unlikely that the KK selected chromosomes are not the fittest at all. This is done PS times so a new phenotype for the next generation is created. This technique allows less fit chromosomes to pass to the next generation.

2.4.2 Crossover

From one generation to another, chromosomes can crossover. This means that two chromosomes split on one place and that one part of the chromosome forms a new chromosome with another part of the other chromosome. The same happens with the two parts that remain and hence two new chromosomes have been created. Consider two chromosomes $X_1 = 10011001$ and $X_2 = 01110011$. They have been selected to go to the next generation and in between the two generations the chromosomes split after the second digit. 4 subchromosomes now exist: $X_{1,a} = 10$, $X_{1,b} = 011001$, $X_{2,a} = 01$ and $X_{2,b} = 110011$. Crossover means that $X_{1,a}$ and $X_{2,b}$ combine and the same happens with $X_{2,a}$ and $X_{1,b}$, so that two new chromosomes are created:

$$X_1^{'} = X_{1,a} \oplus X_{2,b} = 10 \oplus 110011 = 10110011 \tag{2.13}$$

$$X_2^{'} = X_{2,a} \oplus X_{1,b} = 01 \oplus 011001 = 01011001 \tag{2.14}$$

The $\oplus$ represents the concatenation of two subchromosomes and $X_1^{'}$ and $X_2^{'}$ are the two new chromosomes. In the algorithm developed later on, the string length for every variable is fixed through the generations and trials. Therefore, the place where the chromosomes are split is the same for both chromosomes. Doing so the newly generated chromosomes will always have the same length. When the length of the chromosomes would vary it would mean that the precision obtained would vary as well.

Splitting the chromosome can take place after the first binary and before the last. Thus, chromosome X_1 could be broken after the first until the seventh binary. This means there are $\lambda - 1$ possible break open positions. Crossover is applied to create new chromosomes and allow the generation of new chromosomes with, hopefully, a higher fitness and chance to survive than their parents.

The probability that crossover takes place is called the crossover probability, P_c. The higher P_c the more new chromosomes will be generated and more of the search space will be explored. Highly exploring the search space can give an answer to premature convergence, but over-exploring might also result in losing the (absolute) optimal solution again. A solution for this could be to store the fittest chromosome, this technique is called elitism and will be discussed later. Another approach is to change P_c during the generations. The algorithm developed allows to work with a linear crossover probability, $P_c(\gamma)$:

$$P_c(\gamma) = \frac{\gamma_e - \gamma}{\gamma_e - \gamma_b}(P_{c,e} - P_{c,b}) \tag{2.15}$$

$P_c(\gamma)$ is function of the generation it is in. $P_{c,e}$ is the crossover probability in the last (end) generation, γ_e, and $P_{c,b}$ in the first (begin) generation, γ_b. $P_c(\gamma)$ usually starts at a high value, to allow a a lot of different chromosomes to be created and towards the end of the run P_c is lowered so that the part of the search space with the, hopefully, optimum solution is further explored.

2.4.3 Mutation

Mutation happens in one chromosome and changes one of the chromosome's genes: a 1 will become a 0 and the other way around. The object is to further explore the search space. Consider a chromosome $X_3 = 10010011$ that is mutated in its second gene. The new chromosome $X_3'' = 11010011$ will now represent a totally different double value. This new chromosome might be in an area of the search space that was never searched in so far. In the last generation, crossover might not result in a new solution that is fitter. As an example, consider two chromosomes in the second last generation: $X_4 = 10001100$ and $X_5 = 10001100$. During the previous generations the fittest chromosomes survived and the population might thus exist of identical chromosomes, that are as fit. Crossing over X_4 and X_5 will thus not result in new information. If on the other hand, the chromosome is mutated a totally new chromosome will be generated.

The mutation probability, P_m, is usually chosen to be $\frac{1}{\lambda}$. The algorithm used in this master's thesis allows the user to use a fixed P_m as well as a linear changing $P_m(\gamma)$. The general idea is the same as described in subsection (2.4.2).

2.4.4 Antimetathesis

Anti metathesis was first proposed by Katsifarakis and Karpouzos [23] and can be used here as well. The probability with which antimetathesis takes place, P_f, is usually taken to be the same as P_m. When a gene of the chromosome is selected, its value will be changed from 1 to 0 or from 0 to 1, just as with mutation. Next to that the next gene is changed as well, based upon the new value of the selected gene. If the gene was changed to a 0, then the next gene will be a 1 and vice versa. Four possibilities exist: 1) 00 $\rightarrow$ 10, 2) 01 $\rightarrow$ 10, 3) 10 $\rightarrow$ 01, 4) 11 $\rightarrow$ 01.

The reasoning why to do this is explained with the following simple example. Suppose the exact solution is represented by the chromosome 1101 and that a very fit chromosome 1110 was found. Mutation can never lead to the exact chromosome but using antimetathesis the solution is found when the third gene was selected.

Antimetathesis and mutation are suggested to take place interchangingly.

2.4.5 Elitism

By applying selection, crossover and mutation it could be that the fittest solution disappears from the population again. Therefore the algorithm is equipped with a memory for the fittest chromosome. Before selection takes place, the fittest chromosome is stored and after all the operators took place it is added again to the population. In this way, the fittest chromosome can never disappear. This technique is called elitism. When elitism is used in this text it will be indicated by $\epsilon = 1$ and if not by $\epsilon = 0$.

2.5 A simple example

The idea of genetic algorithms might look abstract, but in fact it is a very logical approach. In a simple example, using selection, crossover and mutation, it is shown how things work.

In the example a population size, PS, of 4 chromosomes is considered. Every population thus has 4 chromosomes of which the chromosome length λ is chosen to be 4. The chromosome representation is binary. There will be three generations and the crossover probability P_c is constant over all generations and is 0.8. The last given is the mutation probability what is as suggested 0.25, calculated as $\frac{1}{PS}$.

The following happens, at random a first generation is created, each chromosome having the same probability:

$$\gamma(0) = \begin{cases} X_1 = 0010 \\ X_2 = 1010 \\ X_3 = 1101 \\ X_4 = 0101 \end{cases} \tag{2.16}$$

For all the chromosomes in the population, their fitness should be calculated. Consider the following fitness function Φ that equals the number of 1's in the chromosome. The fitness of the chromosomes is thus:

$$\Phi(\gamma(0)) = \begin{cases} \Phi(X_1) = 1 \\ \Phi(X_2) = 2 \\ \Phi(X_3) = 3 \\ \Phi(X_4) = 2 \end{cases} \tag{2.17}$$

Using, for example roulette wheel selection, the individual probability, P, of a chromosome going to the next generation (survival of the fittest!) is thus:

$$P(\gamma(0)) = \begin{cases} P(X_1) = 1/8 = 0.125 \\ P(X_2) = 2/8 = 0.250 \\ P(X_3) = 3/8 = 0.375 \\ P(X_4) = 2/8 = 0.250 \end{cases} \tag{2.18}$$

$\gamma(1)$ might then look like:

$$\gamma(1) = \begin{cases} X_1 = 1101 \\ X_2 = 1010 \\ X_3 = 0101 \\ X_4 = 0101 \end{cases} \tag{2.19}$$

By chance, the less fit solution has left the population, and was replaced by the fittest chromosome. Selecting again would probably result in another group of chromosomes. On this generation crossover is applied. Chromosomes X_1 and X_4 are selected by chance and crossover will take place ($P_c = 0.8$). The chromosomes split up after the third gene. The place where the chromosomes are split is also decided with equal probability. 4 chromosomes now exist: $X_{1,a} = 110$, $X_{1,b} = 1$, $X_{4,a} = 010$ and $X_{4,b} = 1$. Recombining gives us two new chromosomes: $X_1^{'} = 1101$ and $X_2^{'} = 0101$. In this notation the $'$ indicates the situation after crossover. Two more chromosomes need to be selected to have a fully populated population. Again by chance X_2 and X_3 were selected and crossed over after the first binary. The new chromosomes are thus $X_3^{'} = 1101$ and $X_4^{'} = 0010$. The population now looks like this:

$$\gamma(1)^{'} = \begin{cases} X_1^{'} = 1101 \\ X_2^{'} = 0101 \\ X_3^{'} = 1101 \\ X_4^{'} = 0010 \end{cases} \tag{2.20}$$

After crossover took place the chromosomes are mutated. The mutation probability is 0.25 and as a result only chromosome $X_4^{'}$ is mutated (binary is changed) in the second gene. The new chromosome is thus $X_4^{''} = 0110$. Where the $''$ indicates the chromosome after mutation took place, the situation is now:

$$\gamma(1)^{''} = \begin{cases} X_1^{''} = 1101 \\ X_2^{''} = 0101 \\ X_3^{''} = 1101 \\ X_4^{''} = 0110 \end{cases} \tag{2.21}$$

Using selection, crossover and mutation has increased the total fitness from the generation from 8 to 10, and there are now 2 chromosomes that already have a fitness of 3. Repeating the selecting, crossover and mutation operators, will thus statistically improve the overall fitness and the individual fitness. The last generation might look like this:

$$\gamma(3)'' = \begin{cases} X_1'' = 1101 \\ X_2'' = 1101 \\ X_3'' = 1111 \\ X_4'' = 0111 \end{cases} \tag{2.22}$$

It is thus clear that the maximum fitness, and thus the optimal solution, was found for chromosome X_3. If the number of runs would even be much bigger, then all chromosomes would evolve to become 1111. Although it must be mentioned that because of the mutation that takes place a chromosome with lower fitness might always occur in the population.

2.6 Test functions

Test functions are used to monitor the genetic algorithm and see how well it is performing. A lot of the test functions are available, some of them are more interesting than others. In what follows some of test functions are defined. They are implemented in the algorithm as well and will be used later in the case study.

2.6.1 φ_{max} as function of γ

A graph of φ_{max} as function of γ tells us if the algorithm has trouble finding better candidate solutions. If so it might be worth it to enlarge the population size PS, or choose another fitness function.

2.6.2 Off and on-line performance

The off-line performance, φ_{off}, shows the evolution of the average of the fitness of the best individual, φ_{max}, during the run, γ.

$$\varphi_{off}(\gamma) = \frac{1}{\gamma}\sum_{i=1}^{\gamma}\varphi_{max}(i) \tag{2.23}$$

The on-line performance, φ_{on}, gives the evolution of the average of all fitness functions φ_i during the run:

$$\varphi_{on}(\gamma) = \frac{1}{\gamma}\sum_{j=1}^{\gamma}\varphi_{ave}(\gamma) = \frac{1}{\gamma}\sum_{j=1}^{\gamma}\left[\frac{1}{PS}\sum_{i=1}^{PS}\varphi_i(j)\right] \tag{2.24}$$

2.6.3 Convergence velocity

This parameter shows if the GA made a lot of progress. Σ is called the convergence velocity. Γ is the last run.

$$\Sigma = \ln \sqrt{\frac{\varphi_{max}(\gamma = \Gamma)}{\varphi_{max}(\gamma = 0)}} \tag{2.25}$$

Because the algorithm is capable of working with both negative and positive fitness functions, a negative value might be passed to the ln function. To avoid this problem $\varphi_{max}(\gamma = 0)$ is set to a fixed value of one. The fitness added to do so is then also added to $\gamma = \Gamma$.

2.6.4 The run with maximum fitness

G_{max} is a parameter that stores during which generation the maximum fitness was obtained. G_{max} keeps track of the generation when the fittest solution was found. When elitism is used the fitness has to increase or remain at least the same from one generation to another. When elitism is not used, the fittest chromosome might disappear out of the population and the end solution might be less fit.

For example, the algorithm might be executed 100 times, with a number of generations of 50. When for all trials the optimum solution is found after maximum 15 generations, it is then clear that 15 is the number of trials needed to find the optimum. 35 trials are not needed anymore which reduces the calculation time.

Chapter 3

Boundary element method

3.1 Introduction

3.1.1 In this chapter

This chapter explains what the boundary element method is and why it is a good method for the objectives dealt within this writing. Before the mathematical formulation of the boundary element method is given, a few important aspects of the mathematical background are explained. The steps necessary to go from the mathematic formulation to the numerical implementation are also explained. The derived formula are only applicable for the boundary elements used in this thesis, which are constant boundary elements. The reader will thus find out step by step, how the method is built.

From the general method the extensions are made to include wells (point sources, which is very straight forward) and the implementation of a sheet pile wall (which requires some more work, since extra boundary elements can be created and existing elements might change). A section will deal with reducing the calculation time/load and a simple example will try to make things even more clear.

3.1.2 What is the boundary element method

Wikipedia describes the boundary element method as [22]: *'(...) a numerical computational method of solving linear partial differential equations which have been formulated as integral equations (i.e. in boundary integral form). It can be applied in many areas of engineering and science including fluid mechanics, acoustics, electromagnetics, and fracture mechanics. (...)'*

In simpler words it means that this method solves the Laplace (or Poisson) equation (the linear partial differential equation) where only input data is required on the boundary of the domain and therefore called boundary integral form. Solving this integral equation is done by discretizing the boundary and calculating the integrals in a numeric, rather than analytic way.

A lot of books are available concerning the basic principles of the boundary element method [1, 4, 5, 6] and also the website `http://www.iam.uni-stuttgart.de/bem`[15] gives a good introduction to the boundary element method. However for every specific problem these basic principles need to be extended.

3.1.3 Why the boundary element method? - Comparison to FEM

Other techniques, such as the finite element method (FEM), can be used instead of the boundary element method (BEM) that will be used here.

In a work, published by Donea and Huerta, on the use of finite element method for flow problems and the course manual *Eindige elementen methode*[2] (finite element method) written by professor Verhegge from Ghent University both provide the reader with more information about the use of the finite element method.

In this section the advantages of the BEM over the FEM are explained and as a result it will be clear that the use of the BEM is indeed a very good choice for the challenges that lay ahead.

Advantages

The biggest advantage of the BEM over the FEM is that no discretization of the inside domain is required, only the boundary of the domain should be discretizised. Thus, compared to the FEM, less equations and input data is needed. When the conditions at the domain boundary, called the boundary conditions, are known, the condition in any point in the domain can be calculated from the solution yielded for the boundary nodes.

The BEM is effective in computing the derivatives of the field function. When using the FEM, the accuracy drops, especially in areas or large gradients. Furthermore it is very easy to implement wells (concentrated force).

In my personal opinion, I also think the BEM method is easier to learn.

Disadvantages

The method requires that fundamental solution is known. There is no problem concerning the fundamental solution because the cases studied are always linear and the coefficients of the differential equation are constant. Superposition is thus at all times valid, and will be used to add to the wells.

A disadvantage of the Boundary element method is the fully populated and non-symmetric coefficient matrices of the linear algebraic equations that are produced. The FEM works with symmetric and not fully populated matrices, but the size of the matrices is bigger. Since most of the boundary elements remain unchanged during all generations, only parts of the fully populated matrices will be recalculated. This disadvantage will therefore disappear.

3.2 Mathematical background

To understand the theory of the boundary element method four mathematical concepts need to be explained. They are explained here and will be used in the next section. In this section also a fundamental solution will be derived that will as well be used in the next section.

3.2.1 The Gauss-Green theorem

This theorem is essential for the boundary element method. Using this theorem it becomes possible to go from a domain integral to a boundary integral. The domain in the algorithm that will be developed later on is a 2D model. As explained before, good information is available about the 3D model as well, but only what is necessary for the boundary element developed later on will be discussed. The domain, Ω, thus only has two dimensions (x and y). Γ is defined as the boundary of Ω and in the domain a function $f = f(x, y)$ is valid. Fig. (3.1) depicts the composition. The integral of the derivative of f in respect to x over the domain Ω is noted as:

$$\int\limits_{\Omega} \frac{\partial f}{\partial x} \, \mathrm{d}\Omega \tag{3.1}$$

Because the boundary of the domain is known, eq. (3.1) can be written as a function of it's variables x and y. More precisely, the surface integral can be written as a double integral. For example first with respect to $x = f(y)$ and then with respect to y:

$$\int\limits_{\Omega} \frac{\partial f}{\partial x} \, \mathrm{d}\Omega = \int\limits_{y_1}^{y_2} \int\limits_{x_1(y)}^{x_2(y)} \frac{\partial f}{\partial x} \, \mathrm{d}x \, \mathrm{d}y = \int\limits_{y_1}^{y_2} \left(f(x_2, y) - f(x_1, y)\right) \, \mathrm{d}y \tag{3.2}$$

Figure (3.1) show that for every y_1 and y_2 the total boundary Γ is formed by two curves from s_1 and s_2. Furthermore the following relationship is clear, where s is measured in a counter-clockwise sense:

$$\cos\alpha = \frac{\mathrm{d}y}{\mathrm{d}s} = \frac{n_x}{||\vec{n}||} \Rightarrow \mathrm{d}y = n_x \, \mathrm{d}s \tag{3.3}$$

Eq. (3.2) can thus be expressed as a function of $\mathrm{d}s$, where $\vec{n}$ is the outward normal on Γ, and n_x its component according to the x-dimension:

$$\int\limits_{y_1}^{y_2} \left(f(x_2, y) - f(x_1, y)\right) \, \mathrm{d}y = \int\limits_{s_2} f(x_2, y) n_x \, \mathrm{d}s + \int\limits_{s_1} f(x_1, y) n_x \, \mathrm{d}s \tag{3.4}$$

The plus sign in the last term of eq. (3.4) is there because s_1 goes from y_2 to y_1. Turning the sense turns the sign. s_1 and s_2 together form Γ and thus can be written for s counter-clockwise over the entire of Γ:

$$\int\limits_{\Omega} \frac{\partial f}{\partial x} \, \mathrm{d}\Omega = \int\limits_{\Gamma} f(x, y) n_x \, \mathrm{d}s \tag{3.5}$$

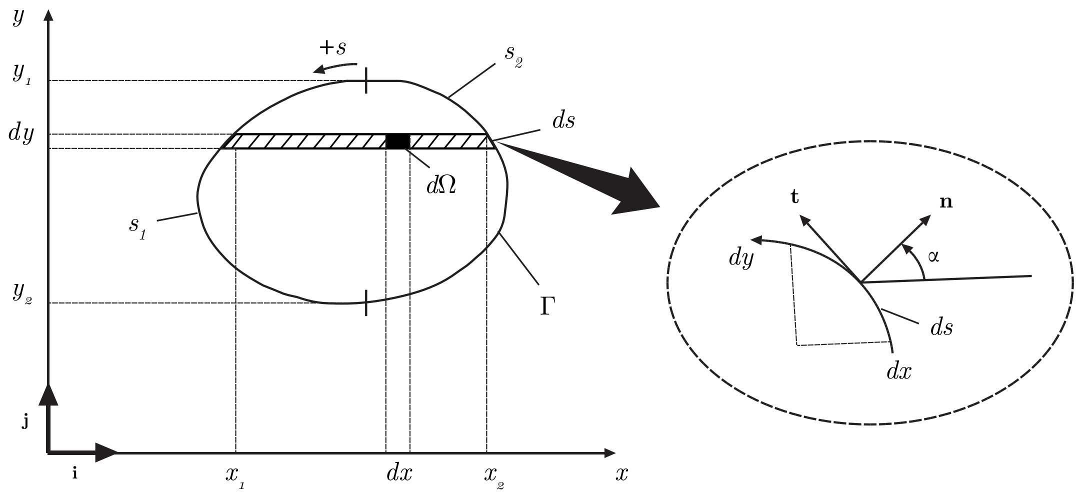

Figure 3.1: Domain Ω with boundary Γ

In a similar way the following equation can be derived, where n_y is the component of $\vec{n}$ along the y-dimension:

$$\int\limits_{\Omega} \frac{\partial f}{\partial y}\ \mathrm{d}\Omega = \int\limits_{\Gamma} f(x,y) n_y\ \mathrm{d}s \tag{3.6}$$

Equation. (3.5) for the function fg, where both f and g are function of x and y is then:

$$\begin{aligned}\int\limits_{\Omega} \frac{\partial (fg)}{\partial x}\ \mathrm{d}\Omega &= \int\limits_{\Gamma} (fg) n_x\ \mathrm{d}s \\ &= \int\limits_{\Omega} g\frac{\partial f}{\partial x}\ \mathrm{d}\Omega + \int\limits_{\Omega} f\frac{\partial g}{\partial x}\ \mathrm{d}\Omega\end{aligned} \tag{3.7}$$

And thus:

$$\int\limits_{\Omega} g\frac{\partial f}{\partial x}\ \mathrm{d}\Omega = \int\limits_{\Gamma} (fg) n_x\ \mathrm{d}s - \int\limits_{\Omega} f\frac{\partial g}{\partial x}\ \mathrm{d}\Omega \tag{3.8}$$

In an analogue way the relation for the partial of y is found:

$$\int\limits_{\Omega} g\frac{\partial f}{\partial y}\ \mathrm{d}\Omega = \int\limits_{\Gamma} (fg) n_y\ \mathrm{d}s - \int\limits_{\Omega} f\frac{\partial g}{\partial y}\ \mathrm{d}\Omega \tag{3.9}$$

The integration by parts is called the *Gauss-Green theorem.*

3.2.2 The divergence theorem of Gauss

A vector field $\vec{\mathbf{u}}$ is considered in the two dimensional space (x and y), with bound vectors $\vec{i}$ along the x- and $\vec{j}$ along the y-dimension. This $\vec{\mathbf{u}}$ is thus composed out of two vectors $u \cdot \vec{i}$ and $v \cdot \vec{j}$. $u(x,y)$ and $v(x,y)$ are the magnitude (scalar) of the vector. This vector field is notated as:

$$\vec{\mathbf{u}} = u(x,y)\vec{i} + v(x,y)\vec{j} = (u,v) \tag{3.10}$$

The normal $\vec{\mathbf{n}}$ can be written as well in that same space as:

$$\vec{\mathbf{n}} = n_x\vec{i} + n_y\vec{j} = (n_x, n_y) \tag{3.11}$$

When in eq. (3.5) $f = u$ and in eq. (3.6) $f = v$ is substituted and they are added together the following equation is yielded:

$$\int\limits_{\Omega} \frac{\partial u}{\partial x}\ \mathrm{d}\Omega + \int\limits_{\Omega} \frac{\partial v}{\partial y}\ \mathrm{d}\Omega = \int\limits_{\Omega} \left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}\right)\ \mathrm{d}\Omega = \int\limits_{\Gamma} (un_x + vn_y)\ \mathrm{d}s \tag{3.12}$$

The last term in eq. (3.12) can be written in vector notation:

$$\int\limits_{\Omega} \frac{\partial u}{\partial x}\ \mathrm{d}\Omega + \int\limits_{\Omega} \frac{\partial v}{\partial y}\ \mathrm{d}\Omega = \int\limits_{\Gamma} \vec{u} \cdot \vec{n}\ \mathrm{d}s \tag{3.13}$$

Introducing the vector ∇ defined as:

$$\nabla = \frac{\partial}{\partial x}\vec{i} + \frac{\partial}{\partial y}\vec{j} \tag{3.14}$$

equation (3.12) can be notated as:

$$\int\limits_{\Omega} \nabla \cdot \vec{\mathbf{u}}\ \mathrm{d}\Omega = \int\limits_{\Gamma} \vec{u} \cdot \vec{n}\ \mathrm{d}s \tag{3.15}$$

The $\cdot$ represents the dot product. $\nabla \cdot \vec{\mathbf{u}}$ is called the divergence of a vector field $\vec{\mathbf{u}}$ inside Ω and thus the name of the theorem.

3.2.3 Green's second identity

Consider eq. (3.8) where $f = \dfrac{\partial u}{\partial x}$ and $g = v$ and eq. (3.9) where $f = \dfrac{\partial u}{\partial y}$ and $g = v$. v and u are both function of x and y and are defined to be twice continuously differentiable in Ω and once on Γ:

$$\int\limits_{\Omega} v\frac{\partial^2 u}{\partial x^2}\,\mathrm{d}\Omega = \int\limits_{\Gamma} v\frac{\partial u}{\partial x}n_x\,\mathrm{d}s - \int\limits_{\Omega}\frac{\partial u}{\partial x}\frac{\partial v}{\partial x}\,\mathrm{d}\Omega \tag{3.16}$$

$$\int\limits_{\Omega} v\frac{\partial^2 u}{\partial y^2}\,\mathrm{d}\Omega = \int\limits_{\Gamma} v\frac{\partial u}{\partial y}n_y\,\mathrm{d}s - \int\limits_{\Omega}\frac{\partial u}{\partial y}\frac{\partial v}{\partial y}\,\mathrm{d}\Omega \tag{3.17}$$

Adding eq. (3.16) to eq. (3.17):

$$\int\limits_{\Omega} v\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right)\mathrm{d}\Omega = \int\limits_{\Gamma} v\left(\frac{\partial u}{\partial x}n_x + \frac{\partial u}{\partial y}n_y\right)\mathrm{d}s - \int\limits_{\Omega}\left(\frac{\partial u}{\partial x}\frac{\partial v}{\partial x} + \frac{\partial u}{\partial y}\frac{\partial v}{\partial y}\right)\mathrm{d}\Omega \tag{3.18}$$

Doing the same for eq. (3.8) where $f = \dfrac{\partial v}{\partial x}$ and $g = u$ added by eq. (3.9) where $f = \dfrac{\partial v}{\partial y}$, a similar equation as 3.18 is obtained:

$$\int\limits_{\Omega} u\left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2}\right)\mathrm{d}\Omega = \int\limits_{\Gamma} u\left(\frac{\partial v}{\partial x}n_x + \frac{\partial v}{\partial y}n_y\right)\mathrm{d}s - \int\limits_{\Omega}\left(\frac{\partial u}{\partial x}\frac{\partial v}{\partial x} + \frac{\partial u}{\partial y}\frac{\partial v}{\partial y}\right)\mathrm{d}\Omega \tag{3.19}$$

Subtracting eq. (3.19) from eq. (3.18):

$$\int\limits_{\Omega}\left[v\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) - u\left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2}\right)\right]\mathrm{d}\Omega = \int\limits_{\Gamma}\left[v\left(\frac{\partial u}{\partial x}n_x + \frac{\partial u}{\partial y}n_y\right) - u\left(\frac{\partial v}{\partial x}n_x + \frac{\partial v}{\partial y}n_y\right)\right]\mathrm{d}s \tag{3.20}$$

With the following definitions:

$$\nabla^2 = \nabla\cdot\nabla = \left(\frac{\partial}{\partial x}\vec{i} + \frac{\partial}{\partial y}\vec{j}\right)\cdot\left(\frac{\partial}{\partial x}\vec{i} + \frac{\partial}{\partial y}\vec{j}\right) = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} \tag{3.21}$$

And $\dfrac{\partial}{\partial n}$ defined as:

$$\frac{\partial}{\partial n} = \vec{n}\cdot\nabla = (n_x\vec{i} + n_y\vec{j})\cdot\left(\frac{\partial}{\partial x}\vec{i} + \frac{\partial}{\partial y}\vec{j}\right) = \frac{\partial}{\partial x}n_x + \frac{\partial}{\partial y}n_y \tag{3.22}$$

Equation (3.20) can be written in vector notation as:

$$\int\limits_{\Omega} (v\nabla^2 u - u\nabla^2 v)\ \mathrm{d}\Omega = \int\limits_{\Gamma} \left(v\frac{\partial u}{\partial n} - u\frac{\partial v}{\partial n} \right)\ \mathrm{d}s \tag{3.23}$$

∇^2 is called the Laplace operator or the harmonic operator and eq. (3.23) as Greens' reciprocal identity or Greens' second identity for the harmonic operator. This is probably the most important formula of the boundary element method.

3.2.4 The Dirac delta function

For the use in the application that will be developed further, a two dimensional Dirac delta function is needed. The two dimensional Dirac delta function, $\delta(Q - Q_0)$ is defined as:

$$\int\limits_{\Omega} \delta(Q - Q_0)h(Q)\ \mathrm{d}\Omega = h(Q_0) \tag{3.24}$$

In eq. (3.24) Q and Q_0 are both functions of x and y and they are located in Ω. $h(Q)$ is a continuous function in Ω and contains the point Q_0. Q_0 has fixed coordinates x_0 and y_0. Going through Ω only one point of the domain, Q_0, will lead to an increment of the integral. For all other points a 0 influence is applicable. This can also be written as:

$$\delta(Q - Q_0) = \begin{cases} 0, & Q \neq Q_0 \\ \infty, & Q = Q_0 \end{cases} \tag{3.25}$$

And when $h(Q) = 1$:

$$\int\limits_{\Omega} \delta(Q - Q_0)\ \mathrm{d}\Omega = 1 \tag{3.26}$$

3.2.5 The fundamental solution

The density of a source point P at a point Q is defined as:

$$f(Q) = \delta(Q - P) \tag{3.27}$$

and its potential $v(Q, P)$ satisfies:

$$\nabla^2 v = \delta(Q - P) \tag{3.28}$$

In what follows a solution of eq. (3.28) will be derived so that it is a fundamental solution of $\nabla^2 = 0$. To do so, eq. (3.28) is written in polar coordinates where the origin is at point P:

$$\frac{1}{r}\frac{\mathrm{d}}{\mathrm{d}r}\left(r\frac{\mathrm{d}v}{\mathrm{d}r}\right) = \delta(Q - P) \tag{3.29}$$

where:

$$r = \sqrt{(\xi - x)^2 + (\eta - y)^2} \tag{3.30}$$

(x, y) are the coordinates of P and (ξ, η) the coordinates of Q. The situation is depicted in fig. (3.2)

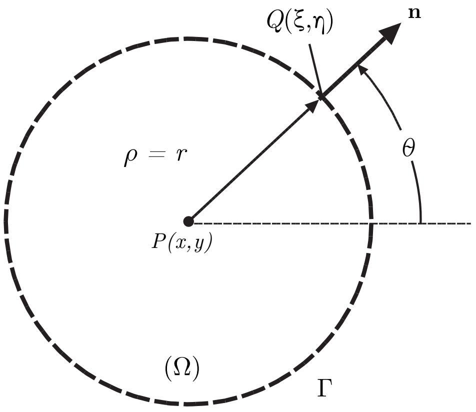

Figure 3.2: Density $Q(\xi, \eta)$ from source point $P(x, y)$

According to the definition of the Dirac delta function, its value is 0 for all positions where $Q \neq P$ and ∞ when $Q = P$. For all $r \neq 0$, $\delta(Q - P) = 0$ and eq. (3.29) is:

$$\frac{1}{r}\frac{\mathrm{d}}{\mathrm{d}r}\left(r\frac{\mathrm{d}v}{\mathrm{d}r}\right) = 0 \tag{3.31}$$

For this equation a lot of solutions exist. Integrating twice gives:

$$v = A \ln r + B \tag{3.32}$$

One particular solution is found by setting $B = 0$:

$$v = A \ln r \tag{3.33}$$

The value of A can be determined noticing that:

$$\frac{\partial v}{\partial r} = \frac{\partial v}{\partial n} = \frac{A}{r} \tag{3.34}$$

Furthermore, from fig. (3.2), $\mathrm{d}s = r\ \mathrm{d}\Theta$. Applying Green's identity for $u = 1$ and $v = A \ln r$:

$$-\int\limits_{\Omega} \nabla^2 v\ \mathrm{d}\Omega = \int\limits_{\Gamma} \frac{\partial v}{\partial n}\ \mathrm{d}s \tag{3.35}$$

Ω is the circle with center point P and radius r as depicted in fig. (3.2). ∇^2 is known from eq. (3.28) and $\dfrac{\partial v}{\partial r}$ from eq. (3.34) and thus:

$$-\int\limits_{\Omega} \delta(Q-P)\ \mathrm{d}\Omega = \int\limits_{0}^{2\pi} A\ \mathrm{d}\Theta \tag{3.36}$$

From this, with equation (3.26):

$$1 = 2\pi A \Rightarrow A = \frac{1}{2\pi} \tag{3.37}$$

The fundamental solution, v, is thus:

$$v = \frac{1}{2\pi} \ln r \tag{3.38}$$

This solution is called the free space Green's function.

3.3 Mathematical formulation of the boundary element method

3.3.1 Homogeneous equation

As mentioned before, solving the Laplace equation results in the solution for the problem where no point sources are applicable.

$$\nabla^2 u = 0 \overset{yields}{\rightarrow} u(x, y) \tag{3.39}$$

Consider now the following functions u and v that meet the conditions:

$$\nabla^2 u = 0 \tag{3.40}$$

and

$$\nabla^2 v = \delta(Q - P) \tag{3.41}$$

Eq. (3.41) was derived in section (3.2.5) and expresses the potential of a source point P at a point Q. Applying Green's identity (eq. (3.23)), where P lies inside Ω:

$$\int\limits_{\Omega} (v \cdot 0 - u \cdot \delta(Q-P)) \, \mathrm{d}\Omega = -\int\limits_{\Omega} (u \cdot \delta(Q-P)) \, \mathrm{d}\Omega = \int\limits_{\Gamma} \left(v\frac{\partial u}{\partial n} - u\frac{\partial v}{\partial n} \right) \, \mathrm{d}s \tag{3.42}$$

Using formula (3.24):

$$u(P) = -\int\limits_{\Gamma} \left(v\frac{\partial u}{\partial n} - u\frac{\partial v}{\partial n} \right) \, \mathrm{d}s \tag{3.43}$$

This equation is called the integral representation of the solution for the Laplace equation and is valid when P is inside Ω. The value of v, that is the fundamental solution of the Laplace equation, is known from section (3.2.5). The derivative $\dfrac{\partial v}{\partial n}$ becomes clear from figure (3.3):

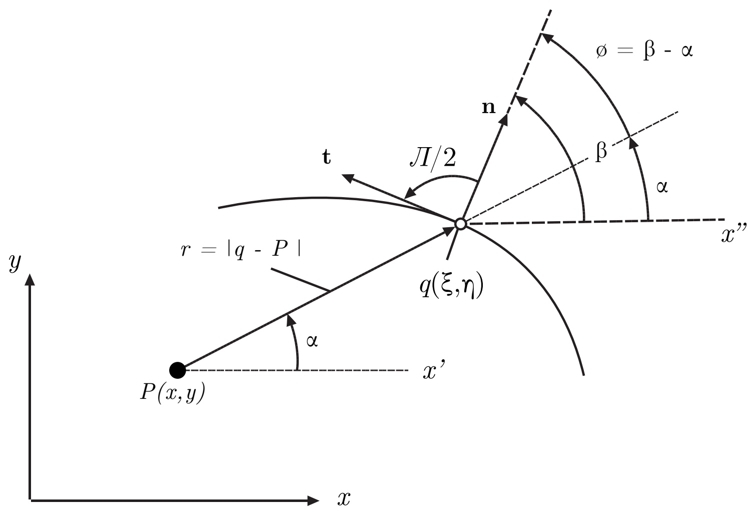

Figure 3.3: Derivative r to n

First the two following geometric relations are clear:

$$\cos\alpha = \frac{\xi - x}{r} \tag{3.44}$$

$$\sin\alpha = \frac{\eta - y}{r} \tag{3.45}$$

r is the length between P and Q:

$$r = \sqrt{(\xi - x)^2 + (\eta - y)^2} \tag{3.46}$$

Differentiating to x, resp y gives, and keeping in mind that when x and y increase ξ and η decrease:

$$\frac{\mathrm{d}r}{\mathrm{d}x} = -\frac{\mathrm{d}r}{\mathrm{d}\xi} = -\frac{\xi - x}{r} = -\cos\alpha \tag{3.47}$$

$$\frac{\mathrm{d}r}{\mathrm{d}y} = -\frac{\mathrm{d}r}{\mathrm{d}\eta} = -\frac{\eta - y}{r} = -\sin\alpha \tag{3.48}$$

Furthermore the relation to the outward normal on Γ can be deducted:

$$\cos\beta = \frac{n_x}{1} = n_x \tag{3.49}$$

$$\sin\beta = \frac{n_y}{1} = n_y \tag{3.50}$$

Knowing this the derivative of r with respect to n can be calculated:

$$\begin{aligned}\frac{\mathrm{d}r}{\mathrm{d}n} &= \frac{\mathrm{d}r}{\mathrm{d}\xi}n_x + \frac{\mathrm{d}r}{\mathrm{d}\eta}n_y \\ &= \frac{\mathrm{d}r}{\mathrm{d}\xi}\cos\beta + \frac{\mathrm{d}r}{\mathrm{d}\eta}\sin\beta \\ &= \cos\alpha\cos\beta + \sin\alpha\sin\beta \\ &= \cos(\beta - \alpha) \\ &= \cos\phi\end{aligned} \tag{3.51}$$

And thus the derivative of (3.38) with respect to n is:

$$\frac{\mathrm{d}v}{\mathrm{d}n} = \frac{1}{2\pi}\frac{\cos\phi}{r} \tag{3.52}$$

The integral representation also needs to be calculated for points P that are on Γ. To do so the approach is to start with a point P that is outside the domain and let the domain approach P. In the limit situation the domain will touch P and the later will thus be on the boundary. This situation is given in figure (3.4). The shortest distance possible between P and Ω^* is $\epsilon = r$. Ω^* is the part of Ω minus the part of Ω that belongs to the circle with center point in P and radius ϵ. It is clear that indeed, if ϵ approaches 0, that the domain approaches the point P, and eventually, when $\epsilon = 0$, P is on Γ. The total length of the arcs AP and PB is defined as l and the arch AB is defined as Γ_ϵ. Because of the circular boundary, the outward normal on Γ_ϵ is always pointed towards P and thus collides with the radius.

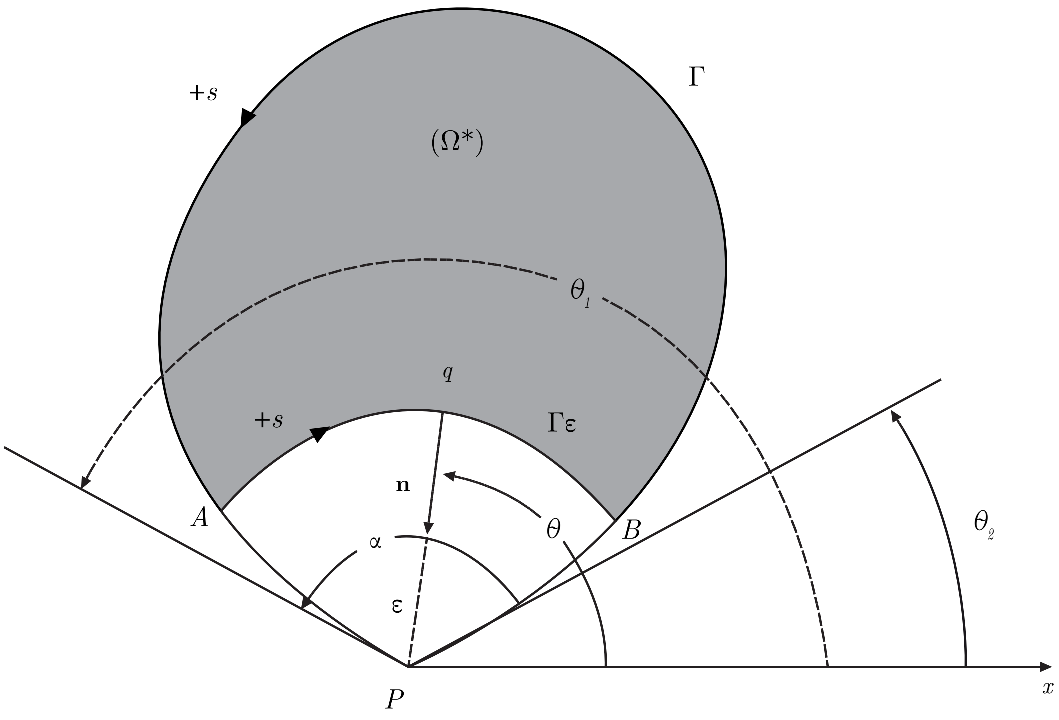

Figure 3.4: P outside of the domain

Writing once again Green's identity but now for the domain Ω^*, where u and v satisfy conditions (3.40) and (3.41):

$$\int_{\Omega^*} (v \cdot 0 - u \cdot 0)\, \mathrm{d}\Omega = 0 = \int_{\Gamma} \left(v\frac{\partial u}{\partial n} - u\frac{\partial v}{\partial n} \right)\, \mathrm{d}s \tag{3.53}$$

Indeed, according to the definition of the Dirac delta function, $\delta(Q - P) = 0$ where P is outside of Ω^*. Γ can be devided in two pieces: $\Gamma - l$ and Γ_ϵ and eq. (3.53) is thus:

$$0 = \int\limits_{\Gamma - l} \left(v\frac{\partial u}{\partial n} - u\frac{\partial v}{\partial n} \right) \, \mathrm{d}s + \int\limits_{\Gamma_\epsilon} \left(v\frac{\partial u}{\partial n} - u\frac{\partial v}{\partial n} \right) \, \mathrm{d}s \tag{3.54}$$

The situation of interest is when ϵ approaches 0. The first integral is simple:

$$\lim_{\epsilon \to 0} \int\limits_{\Gamma - l} \left(v\frac{\partial u}{\partial n} - u\frac{\partial v}{\partial n} \right) \, \mathrm{d}s = \int\limits_{\Gamma} \left(v\frac{\partial u}{\partial n} - u\frac{\partial v}{\partial n} \right) \, \mathrm{d}s \tag{3.55}$$

Because, from figure (3.4), it is clear that:

$$\lim_{\epsilon \to 0} (\Gamma - l) = \Gamma \tag{3.56}$$

The second integral of equation (3.54) is in the case where $\alpha = \pi$ is also straightforward. v and $\mathrm{d}v/\,\mathrm{d}n$ are known from eqs. (3.38) and (3.52) resp., and hence:

$$\lim_{\epsilon \to 0} \int\limits_{\Gamma_\epsilon} \left(v\frac{\partial u}{\partial n} - u\frac{\partial v}{\partial n} \right) \, \mathrm{d}s = \lim_{\epsilon \to 0} \int\limits_{\Gamma_\epsilon} \left(\frac{\ln r}{2\pi}\frac{\partial u}{\partial n} - u\frac{\cos\phi}{2\pi r} \right) \, \mathrm{d}s \tag{3.57}$$

Because $\mathrm{d}s = -r \,\mathrm{d}\phi$ and s over Γ_ϵ is always known when $r = \epsilon$ is known, because under all situations $\phi = \pi$. The last integral is thus reduced to:

$$\begin{aligned} \lim_{\epsilon \to 0} \int\limits_{\Gamma} \left(v\frac{\partial u}{\partial n} - u\frac{\partial v}{\partial n} \right) \, \mathrm{d}s &= \lim_{\epsilon \to 0} \left(\frac{\ln \epsilon}{2\pi}\frac{\partial u}{\partial n} - u\frac{\cos\pi}{2\pi\epsilon} \right) (\pi\epsilon) \\ &= \lim_{\epsilon \to 0} \left(0 - u\frac{-1}{2\pi\epsilon} \right) (\pi\epsilon) \\ &= \frac{1}{2}u(P) \end{aligned} \tag{3.58}$$

Knowing how the two integrals of eq. (3.54) evolve in the limit state to 0, the total limit is thus:

$$0 = \int\limits_{\Gamma} \left(v\frac{\partial u}{\partial n} - u\frac{\partial v}{\partial n} \right) \, \mathrm{d}s + \frac{1}{2}u(P) \Rightarrow \frac{1}{2}u(P) = -\int\limits_{\Gamma} \left(v\frac{\partial u}{\partial n} - u\frac{\partial v}{\partial n} \right) \, \mathrm{d}s \tag{3.59}$$

This equation is valid for source points P on the boundary of the domain, and when the boundary element is smooth ($\alpha = \pi$). This equation is called the boundary integral equation.

When at every point of the boundary u or u_n is known, the correspondening u_n or u can be found using this compatibility relation. As mentioned above, when P is outside Ω, $\delta(Q-P)$ is always zero for all possible Q's in Ω and thus:

$$-\int\limits_{\Omega} (u \cdot \delta(Q-P))\ \mathrm{d}\Omega = 0 = \int\limits_{\Gamma} \left(v\frac{\partial u}{\partial n} - u\frac{\partial v}{\partial n} \right)\ \mathrm{d}s \tag{3.60}$$

Three possible locations for P can thus occur:

1. P is inside Ω: eq. (3.43) is valid
2. P is on the boundary of Ω: eq. (3.59) is valid
3. P is outside of Ω: eq. (3.60) is valid

These three different situations can be written in one equation as:

$$\epsilon(P)u(P) = -\int\limits_{\Gamma} \left(v\frac{\partial u}{\partial n} - u\frac{\partial v}{\partial n} \right)\ \mathrm{d}s \tag{3.61}$$

Where:

$$\epsilon(P) = \begin{cases} 1 & \text{when } P \text{ inside the } \Omega \\ \frac{1}{2} & \text{when } P \text{ on } \Gamma \\ 0 & \text{when } P \text{ outside } \Omega \end{cases} \tag{3.62}$$

In the case of our mixed problem the following equations thus needs to be calculated:

$$\frac{1}{2}\bar{u} = -\int\limits_{\Gamma} \left(v\frac{\partial u}{\partial n} - \bar{u}\frac{\partial v}{\partial n} \right)\ \mathrm{d}s \qquad \text{on } \Gamma_1 \tag{3.63}$$

$$\frac{1}{2}u = -\int\limits_{\Gamma} \left(v\frac{\partial \bar{u}}{\partial n} - u\frac{\partial v}{\partial n} \right)\ \mathrm{d}s \qquad \text{on } \Gamma_2 \tag{3.64}$$

Where Γ_1 is the part of Γ where u is known, Γ_2 where $\dfrac{\mathrm{d}u}{\mathrm{d}n}$ is known and $\Gamma_1 + \Gamma_2 = \Gamma$.

3.3.2 Non homogeneous equation

When a well is added, as later on will be the case, $\nabla^2 \neq 0$. The Laplace equation is not valid anymore and a Poisson equation now describes the problem:

$$\nabla^2 u = f \qquad \text{in } \Omega \tag{3.65}$$

In this equation f is a function of x and y. Its value will later be discussed. In the following few lines it will be proven that the solution of equation (3.65) can be written as a sum of the solution u_0 of a homogeneous equation ($\nabla^2 u_0$) and a particular solution u_1 of the non homogeneous equation ($\nabla^2 u_1$):

$$u = u_0 + u_1 \tag{3.66}$$

The easiest way to prove this is by applying Green's identity where $\nabla^2 u = f$ (eq. 3.65) and $\nabla^2 v = \delta(Q-P)$ (eq. (3.28)):

$$\int\limits_{\Omega} v \cdot f - u \cdot \delta(Q-P)) \; \mathrm{d}\Omega = \int\limits_{\Gamma} \left(v\frac{\partial u}{\partial n} - u\frac{\partial v}{\partial n} \right) \; \mathrm{d}s \tag{3.67}$$

The second term from the left side of the equation is known from eq. (3.24), and for a smooth boundary (analogue to eq. (3.61):

$$\frac{1}{2}u(P) = \int\limits_{\Omega} (v \cdot f \; \mathrm{d}\Omega - \int\limits_{\Gamma} \left(v\frac{\partial u}{\partial n} - u\frac{\partial v}{\partial n} \right) \; \mathrm{d}s \tag{3.68}$$

The last part is exactly the solution of the homogeneous equation, and thus $\int\limits_{\Omega} v \cdot f \; \mathrm{d}\Omega$ is the solution of the non homogeneous solution:

$$u_0 = \int\limits_{\Gamma} \left(v\frac{\partial u}{\partial n} - u\frac{\partial v}{\partial n} \right) \; \mathrm{d}s \tag{3.69}$$

$$u_1 = \int\limits_{\Omega} v \cdot f \; \mathrm{d}\Omega \tag{3.70}$$

For a mixed problem, as is considered, the boundary conditions of the homogeneous are:

$$\bar{u} = u_0 + u_1 \tag{3.71}$$

$$\frac{\delta \bar{u}}{\delta n} = \frac{\delta u_0}{\delta n} + \frac{\delta u_1}{\delta n} \tag{3.72}$$

3.4 Numeric formulation

3.4.1 Discretization

From the previous chapter the analytical solution for the problem was obtained. For all boundary elements an equation similar to equation (3.61) can be written. It is the solution of the Laplace equation at that point p_i and is given by:

$$\frac{1}{2}u(p_i) = -\int_{\Gamma}\left[v(p_i,q)\frac{\partial u(q)}{\partial n_q} - u(q)\frac{\partial v(p_i,q)}{\partial n_q}\right]ds_q \tag{3.73}$$

This equation is valid only for constant line elements and will be used as the basic equation for the model. This equation now needs to be discretized so it can later be computed. Therefore Γ is divided into smaller pieces that all together form Γ again, this is shown in fig. (3.5).

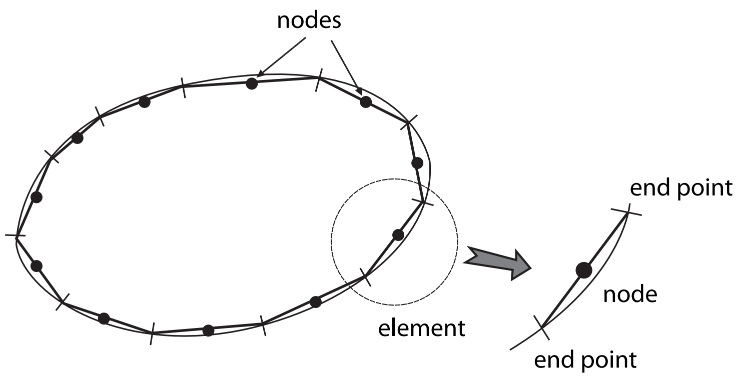

Figure 3.5: The use of constant line elements

For a point p_i, with u^i the value of u in point i, and $u_n = \partial u/\partial n$, equation (3.73) can be written as:

$$\frac{1}{2}u^i = -\sum_{j=1}^{N}\int_{\Gamma_j} v(p_i,q)\frac{\partial u(q)}{\partial n_q}ds_q + \sum_{j=1}^{N}\int_{\Gamma_j} u(q)\frac{\partial v(p_i,q)}{\partial n_q}ds_q \tag{3.74}$$

assuming that Γ is discretized in N parts. Figure (3.6) shows the situation.

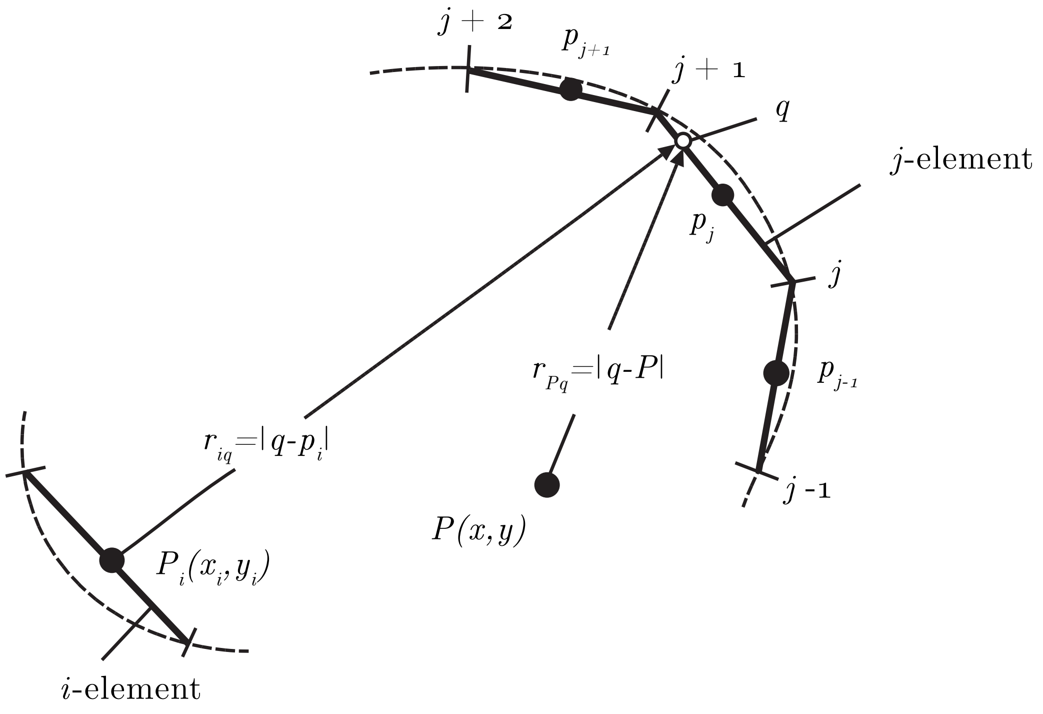

Figure 3.6: Nodal points p, q and P

Because only constant elements are to be used, u and u_n can be moved outside the integral, after placing all terms of u^i and u^j on the left hand side eq. (3.74) becomes:

$$-\frac{1}{2}u^i + \sum_{j=1}^{N}\left(\int_{\Gamma_j}\frac{\partial v(p_i,q)}{\partial n_q}ds_q\right)u^j = \sum_{j=1}^{N}\left(\int_{\Gamma_j}v(p_i,q)ds_q\right)u_n^j \tag{3.75}$$

Equation (3.75) can further be formulated as:

$$\sum_{j=1}^{N}H_{ij}u^j = \sum_{j=1}^{N}G_{ij}u_n^j \tag{3.76}$$

Where:

$$G_{ij} = \int_{\Gamma_j}v(p_i,q)ds_q \tag{3.77}$$

$$H_{ij} = \hat{H}_{i,j} - \frac{1}{2}\delta_{ij} \tag{3.78}$$

$$\hat{H}_{ij} = \int_{\Gamma_j} \frac{\partial v(p_i, q)}{\partial n_q} ds_q \tag{3.79}$$

δ_{ij} is the delta Kronecker function and is always 0 except when $(i = j)$, it then has the value of 1. Equation (3.76) is now almost ready to be computed, only $\hat{H}_{ij}$ and G_{ij} are still in their analytic shape and should be discretized.

3.4.2 H_{ij} and G_{ij}

$H_{i,j}$ and $G_{i,j}$ are evaluated for two different situations. A first is when $i = j$, and when the distance between source point and destination point is zero, called the diagonal elements and a second case where there is distance between the source and destination point: when $i \neq j$, called the off-diagonal elements.

Off-diagonal elements

The integrals are evaluated using Gauss Iteration. Doing so it is possible to approximate an integral as a summation:

$$\int_{-1}^{1} f(\xi)\ \mathrm{d}\xi \approx \sum_{k=1}^{n} f(\xi_k) w_k \tag{3.80}$$

In the algorithm developed 4 integration points will be used $(n = 4)$. The values of the abscissas ξ_k and the corresponding weight factor w_k are listed in table (3.1).

ξ_k	w_k
−0.861136311594053	+0.347854845137454
−0.339981043584856	+0.652145154862546
+0.339981043584856	+0.652145154862546
+0.861136311594053	+0.347854845137454

Table 3.1: 4 point Gauss integration - Abscissas and weights

In order to be able to use equation (3.80), x and y should be known as function of ξ. The approach is to start from a local system with axes $x^{'}$ and $y^{'}$ as depicted in figure (3.7). Depicted is an element j. It's two endpoints are $j(x_j, y_j)$ and $(j+1)(x_{j+1}, y_{j+1})$. Element j in the local system $(x^{'}, y^{'})$ is described by:

$$j(x^{'}, y^{'}) = (x^{'}, 0), \quad \text{Where } -\frac{l_j}{2} \leq x^{'} \leq \frac{l_j}{2} \tag{3.81}$$

And the relation between the local and the global system is thus:

$$x = \frac{x_{j+1} + x_j}{2} + \frac{x_{j+1} - x_j}{l_j} x^{'} \tag{3.82}$$

$$y = \frac{y_{j+1} + y_j}{2} + \frac{y_{j+1} - y_j}{l_j} x^{'}, \qquad -\frac{l_j}{2} \leq x^{'} \leq \frac{l_j}{2} \tag{3.83}$$

l_j is the length of the element (distance between begin and endpoint) and equals:

$$l_j = \sqrt{(x_{j+1} - x_j)^2 + (y_{j+1} - y_j)^2} \tag{3.84}$$

In the local system, $x^{'}$ varies from 0 to $\pm\frac{l_j}{2}$ (the local system has its origin in the middle of element j) and ξ varies from 0 to ± 1, so the relation between $x^{'}$ and ξ is the following:

$$\xi = \frac{2x^{'}}{l_j} \tag{3.85}$$

Equations (3.82) and (3.82) can now be written as function of ξ:

$$x(\xi) = \frac{x_{j+1} + x_j}{2} + \frac{x_{j+1} - x_j}{2} \xi \tag{3.86}$$

$$y(\xi) = \frac{y_{j+1} + y_j}{2} + \frac{y_{j+1} - y_j}{2} \xi \tag{3.87}$$

The only thing missing is the relation between s and ξ, but it is also clear from fig. (3.6):

$$\mathrm{d}s = \sqrt{\mathrm{d}x^2 + \mathrm{d}y^2} = \sqrt{\left(\frac{x_{j+1} - x_j}{2}\right)^2 + \left(\frac{y_{j+1} - y_j}{2}\right)^2} \,\mathrm{d}\xi = \frac{l_j}{2} \,\mathrm{d}\xi \tag{3.88}$$

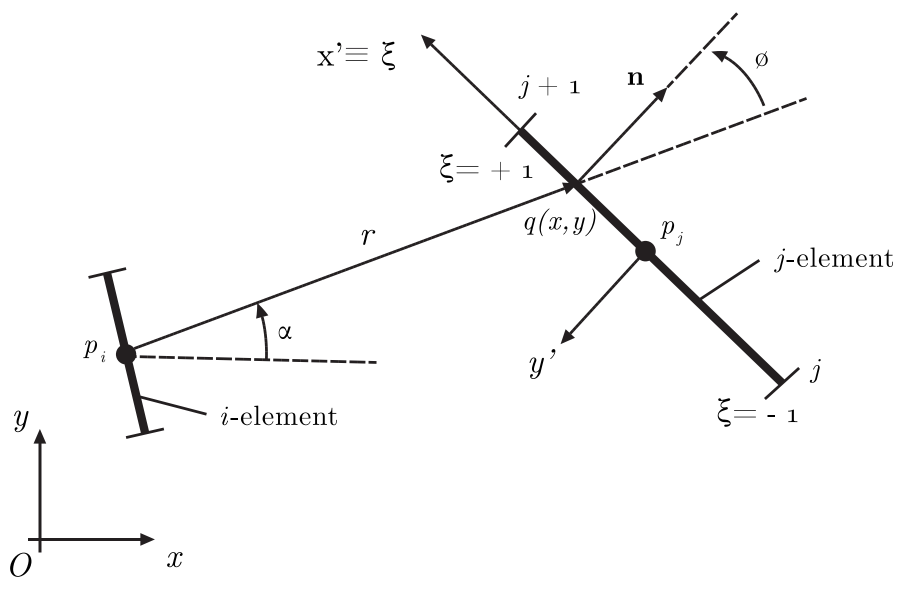

Figure 3.7: Global and local coordinate system

Eq. 3.77 can now be written as:

$$G_{ij} = \int_{\Gamma_j} v(p_i, q) ds_q = \int_{\Gamma_j} \frac{1}{2\pi} \ln[r(\xi)] \frac{l_j}{2} \, \mathrm{d}\xi = \frac{l_j}{4\pi} \sum_{k=1}^{n} \ln\left[r(\xi_k)\right] w_k \tag{3.89}$$

Where:

$$r(\xi_k) = \sqrt{\left(x(\xi_k) - x_i\right)^2 + \left(y(\xi_k) - y_i\right)^2} \tag{3.90}$$

For the off-diagonal elements of $H_{i,j}$, the relation between s and α is required. From fig(3.8):

$$\mathrm{d}s \cos\phi = r \, \mathrm{d}\alpha \Rightarrow \; \mathrm{d}s = \frac{r \, \mathrm{d}\alpha}{cos\phi} \tag{3.91}$$

Combining eq. (3.51) and 3.91:

$$\hat{H}_{ij} = \int_{\Gamma_j} \frac{\partial v}{\partial n} \, \mathrm{d}s = \int_{\Gamma_j} \frac{1}{2\pi} \frac{\cos\phi}{r} \, \mathrm{d}s = \int_{\Gamma_j} \frac{1}{2\pi} \, \mathrm{d}\alpha = \frac{a_{j+1} - a_j}{2\pi} \tag{3.92}$$

Where:

$$a_{j+1} = \arctan\left(\frac{y_{j+1} - y_i}{x_{j+1} - x_i}\right) \tag{3.93}$$

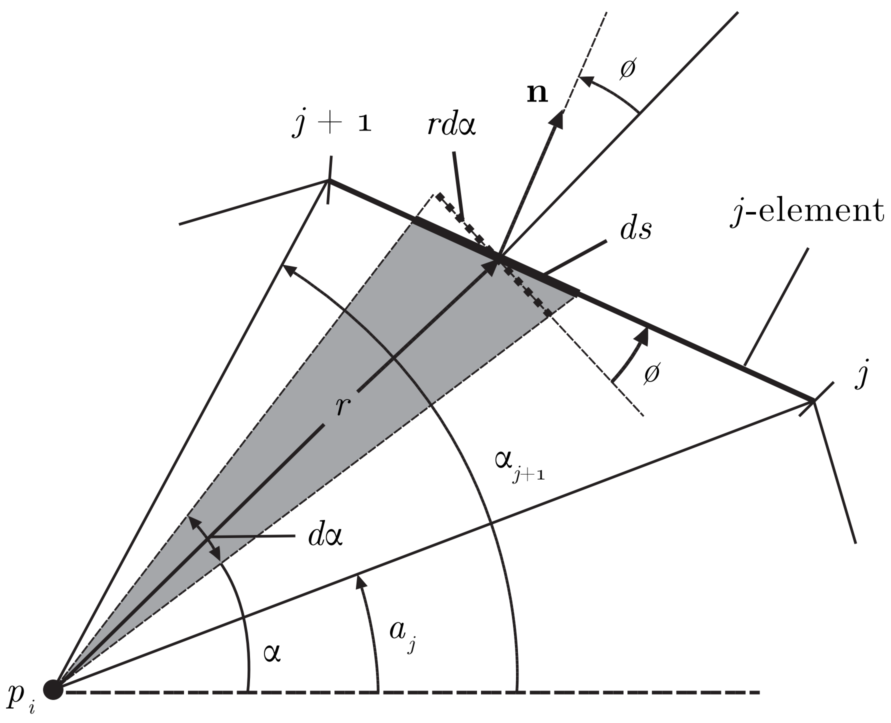

Figure 3.8: Relation between α and s

$$a_j = \arctan\left(\frac{y_j - y_i}{x_j - x_i}\right) \tag{3.94}$$

Diagonal elements

When $i = j$, the source and destination element are the same. This means that r is always on the line element and r is the distance from the center point to the point on the line element. For the mathematical formulation it is clear that $\phi = \frac{\pi}{2}$ or $\phi = \frac{3\pi}{2}$ for all r. As a result $\cos\phi$ is always 0.

$$r(\xi) = \frac{l_j}{2}|\xi| \tag{3.95}$$

the || represents the absolute value. r is always a positive value, that varies from 0 to $\frac{l_j}{2}$ as function of $x^{'}$ and thus in function of ξ from 0 to +1. With this:

$$G_{jj} = \int_{\Gamma_j} v \, \mathrm{d}s = \int_{\Gamma_j} \frac{1}{2\pi} \ln r \, \mathrm{d}s = 2 \int\limits_0^{l_j/2} \frac{1}{2\pi} \ln r \, \mathrm{d}r = \frac{l_j}{2\pi} \left[\ln\left(\frac{l_j}{2}\right) - 1 \right] \tag{3.96}$$

and:

$$\hat{H}_{jj} = \frac{1}{2\pi} \int\limits_{\Gamma_j} \frac{\cos\phi}{r} \, \mathrm{d}s = \frac{1}{2\pi} \int\limits_{-1}^{1} \frac{\cos\phi}{|\xi|} \, \mathrm{d}\xi = \frac{2}{2\pi} \left[\cos\phi \ln|\xi| \right]_0^1 = 0 \tag{3.97}$$

3.4.3 Multi-zone body or composite domain

The fundamental solution is only valid for homogeneous domains, and when the aquifer is not, it should be subdivided in different zones that are homogeneous or can be simplified to be so. Equation (3.74) is then valid for all the sub zones individually but extra information is available for the interfaces between two zones. On the boundary of Γ, u or u_n is known and thus one equation (3.74) can be written with one unknown. For points on the interface both u and u_n are unknown, there is thus only one equation and 2 unknown. For each point p_i on the interface however, two equations (3.74) can be written. One for the first zone, I, and one for the second zone ,II , p_i is in. There are thus 2 equations with 4 unknown ($u^{i,I}, u^{i,I}, u_n^{i,I}$ and $u_n^{i,II}$), however 2 additional equations are available from physical considerations:

- Continuity of the potential. The water height in one node is constant, and thus $u^{i,I}$ in the first zone equals $u^{i,II}$ in the second zone: $u^{i,I} = u^{i,II}$.
- Continuity of the flux. The net flow in a point is zero. What flows in from one zone has to go out in the other zone, $q_n^{i,I} + q_n^{i,II} = 0$. And thus $q_n^i = -q_n^{i,II}$. With Darcy's law this becomes $T_I \cdot u_n^{i,I} = -T_{II} \cdot u_n^{i,II}$ or $u_n^{i,II} = -\frac{T_I}{T_{II}} \cdot u_n^{i,I}$.

q is the flow and T the transmissivity. With this two extra relations per point, we now have as many linear unknown equations as there are unknown. In section (3.6) this is explained with an example.

3.4.4 Well influence

The boundary element method is especially useful when the load is applied on the boundary but it can also deal with loads inside the domain, called a *body force.* The influence of a well is such a load and it is very easy to apply when using the boundary element method. As analytically proven in section (3.3.2), the non homogeneous solution (because of the well) exists of the homogeneous solution calculated before and an extra term because of the well (superposition):

$$\underbrace{\sum_{j=1}^{N} H_{ij} u^j = \sum_{j=1}^{N} G_{ij} u_n^j}_{\text{homogeneous part}} + \underbrace{\sum_{w=1}^{N_w} \left(\wp \cdot \frac{Q_w}{2\pi T} \ln r_i \right)}_{\text{non homogeneous part}} \tag{3.98}$$

In this formula N_w is the number of wells and r_i is the distance from the well to the nodes p_i of the same zone of the well:

$$r_i = \sqrt{(x_i - x_w)^2 + (y_i - y_w)^2} \tag{3.99}$$

The non homogeneous part only affects the boundary elements that are in the same zone of the well. When the boundary element, p_i, is in the same zone as the well, then $\wp = 1$ and if not so $\wp = 0$.

3.4.5 Sheet pile wall

A sheet pile wall is a screen of piles that stops water from flowing according to its natural path. When such a wall is placed close to a boundary of the aquifer, water that tends to flow into the aquifer needs to go around it. Seawater infiltration is thus blocked and the wells can have a higher flow rate.

Implementing a sheet pile wall in the boundary element method means adding and or changing boundary elements through which no flow can exist: $q^i = 0$ and as a result $\bar{u}_n^i = 0$. The location of the sheet pile wall is generated by the genetic algorithm. It will generate a begin and endpoint for the sheet pile wall on the coastline. Based upon this begin and endpoint the boundary elements will constantly change. The boundary elements that were input by the user can thus be changed and need to be recalculated if necessary. In order not to recalculate all the boundary elements every time again, only those that have the property of being a coastal line will be recalculated. And also, the sheet pile wall can only be generated on such boundary elements. Moreover the boundary elements that are coastal lines have to be connected without occurrence of a non coastal boundary element in between. Good input data could then be as depicted in fig. (3.9). Boundary elements 0, 1 and 2 represent the coastline. On these three lines a sheet pile wall can be placed.

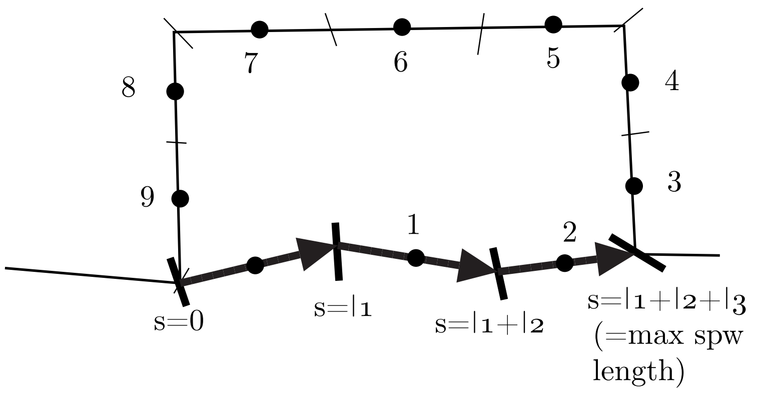

Figure 3.9: Path σ for sheet pile wall

Nine different situations my now occur for the combination of begin and endpoint. The first five take place when the begin and endpoint of the sheet pile wall is spawn on one and the same boundary element, they are listed in figure (3.10). A first possibility is that the begin and end point spawn are the same. In this case A) the length of the sheet pile wall is 0, and nothing should be changed to the boundary elements that were input. Another possibility only affecting one element is that the begin point is spawn on the begin point of the element, and the endpoint somewhere inside the element. In this situation the existing element needs to be split in two. One of the elements will get the property that $\bar{u}_n = 0$ and the other element will have the exact same boundary condition as the original element. The extreme point of the elements need to be recalculated and the array size will increase by one because of the extra element that was created. A similar thing happens in case C) the only difference with B) is how the boundary elements are created by the algorithm.

In case D) the sheet pile wall starts and end somewhere in the boundary element. Two extra elements should now be created. One on both sides of the existing boundary element that is now shortened in length and gets the boundary condition $\bar{u}_n = 0$. The newly created boundary elements get all their properties from the parent element, except for the extreme points and hence the length. The array size is incremented by 2. A last case that only affects one boundary element is when the beginpoint of the sheet pile wall and the boundary element are the same and at the same moment the same happens for the endpoint. No extra elements need to be created and only the boundary condition needs to be set to $\bar{u}_n = 0$.

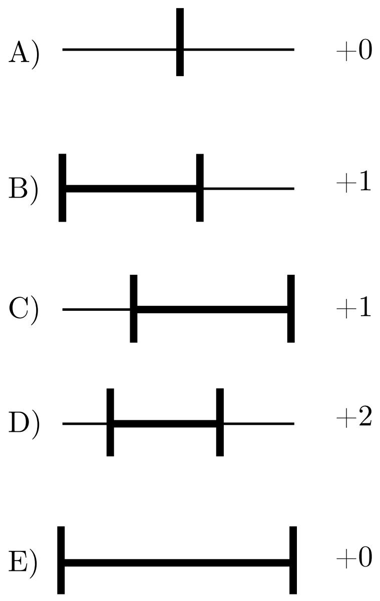

Figure 3.10: Changes to boundary elements when a sheet pile wall is used and the begin and end point of the sheet pile wall is on one boundary element only

4 other situations can occur when the begin and start point of the sheet pile wall are not on the same boundary element. At least two boundary elements are affected. Figure 3.11 shows the possibilities. In case A) the sheet pile wall ends inside a boundary element (the most right) and begins in the begin point of another element. The most right element will thus be split up in two new elements. One element becomes a sheet pile wall and the other inherits the properties of the former element. All the boundary elements in between the element where the sheet pile wall starts and ends keep their exact same properties, except that the boundary condition is changed to that of $\bar{u}_n = 0$. In this case the element that holds the beginning of the sheet pile wall is entirely a sheet pile wall and only it's boundary condition needs to be changed. A similar situation occurs in situation B), where only the first element that holds the sheet pile wall needs to be split up. In both cases 1 extra element is created and hence the array size increases by one.

In case C) both the begin and endpoint of the sheet pile wall are located inside a boundary element. As a result two extra boundary elements have to be created and the array size is incremented by two. In case D) the sheet pile wall starts in the begin point of a boundary element and ends in the endpoint of an element. No extra lines need to be created, only the boundary conditions need to be changed so that no water can flow through the elements.

The algorithm will thus first find out how many lines are affected by the sheet pile wall. If necessary it will split existing and add extra boundary elements and change the properties so that the elements behave as a sheet pile wall, and the newly created elements take the properties of the parent element.

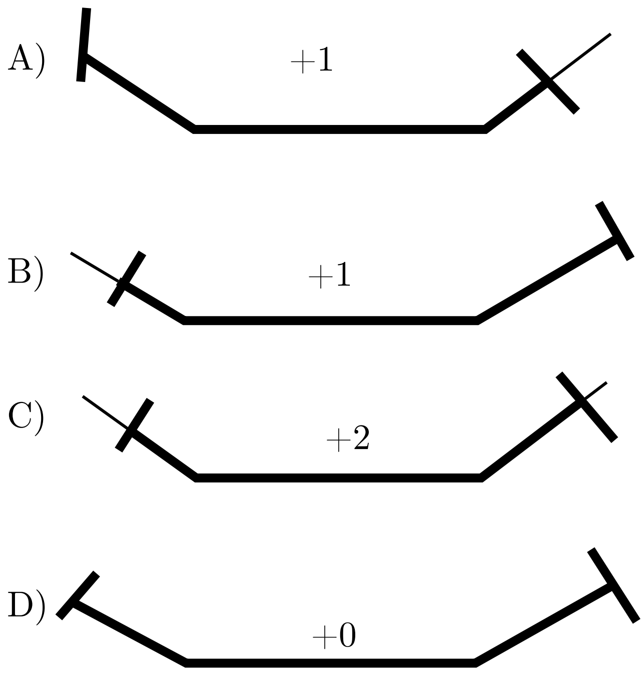

Figure 3.11: Changes to boundary element when a sheet pile wall is used and the begin and end point of the sheet pile wall affect more than one boundary element only

3.4.6 Gauss elimination

Solving equation (3.98) is done by using Gauss iteration. In a first step all the unknown should be brought to one side and all the known to the other side in the equality:

$$H \cdot u = G \cdot u_n \Rightarrow A \cdot X = B_t \cdot Y = B \tag{3.100}$$

A holds all the unknown values of H and G (u and u_n) and B_t all the known values of ($\bar{u}$ and $\bar{u}_n$). B_t and Y hold thus only known values and this matrix can be calculated. X holds all the unknown and when $A \cdot X = B$ is solved to X, the unknown are stored in the X vector. Solving this equation is done as previously mentioned by Gauss elimination.

Two potential problems may arise during the computation: divide by 0 error and round-off errors. Therefore Gauss elimination with partial pivoting is used. When partial pivoting is used all rows in the loop are compared with each other and the one that starts with the highest (absolute) value is brought in front position. Doing this, dividing by 0 is eliminated. In the case a column only has 0's in all the rows, the set of equations is unsolvable.

When multiple domain problems are considered the A matrix will have zones with only zeros there where nodes do not have a relationship with each other. Nodes from different zones don't have a h_{ij} and g_{ij} value. To deal with this gauss elimination is used where both rows and columns might change places. When two columns changes place, the X matrix changes, and when rows are changed of place the B matrix changes without affecting the B matrix.

3.5 Minimizing the calculation work

3.5.1 Calculating A and Bt immediately

Most calculations are made for the G and H matrix, and then transforming them to a A and B matrix based upon the known value of $\bar{u}$ of $\bar{u}_n$. Therefore the algorithm was designed in such a way it calculates A and B immediately. When adding a sheet pile wall, the A and B matrices will change. First of all its size will grow by one when the sheet pile wall begins inside a line, that is not on one of its extreme points. The same increment takes place when the sheet pile wall ends inside a line. The size of the array can thus be increased by one or by two.

The data stored in the matrices containing the information for the calculations also changes, but only there where the sheet pile wall is added. Figure (3.12) gives an example. There is thus no need to calculate the elements of A and B for the lines that are never changed.

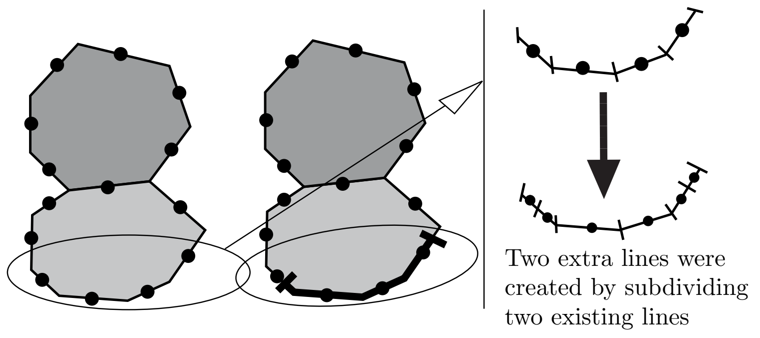

Figure 3.12: Creating extra lines by subdividing (sheet pile wall)

3.5.2 Reducing calculation time for A and B_t matrix

A first reduction already discussed previously is to calculate the A and B matrix without first calculating the H and G matrix. A serious improvement was realized in doing so, but the calculation work could be reduced even more. In the case that no sheet pile wall is used the values of A and B remain constant. The well influence is calculated by superposition. This superposition happens after A and B are calculated and before the equation $A \cdot X = B$ is solved.

In the case a sheet pile wall is used the size of A and B will vary because extra lines are generated for the sheet pile wall. However, for the line elements that are not on the coastline, the respective values can be copied. This means all elements in A and Bt where element i and j are not on the coastline can be copied into the new resized arrays A and Bt. Special attention is required for the location in the destination array because extra lines (and thus unknown and known) were added.

The algorithm will thus calculate four matrices even before the genetic algorithm is executed: uA, uB_t, $uplaatsX$ and $uplaatsB$. They are filled for the input data, thus without generating a sheet pile wall. In the case no sheet pile wall is used these four matrices can be used in the genetic algorithm without changing anything over all the runs. In the case that a sheet pile wall is used all the elements of uA and uB_t that are not on the coastline can be copied to the arrays A and Bt. The other elements of A and Bt need to be calculated every time again and are different for every chromosome combination.

The A and B arrays can be ordered in such a way that the part containing the non coastal line elements never need to be calculated again. Consider again the following matrix equation that was constructed before:

$$A \cdot X = B \qquad (B = B_t \cdot Y) \tag{3.101}$$

The matrices should be filled now in such a way that all the elements that remain constant during the generations are grouped together. In other words this means that all the lines that are not on the coast are grouped. X has than the following structure:

$$X = \left\{ \left\{ x_{f,1} \quad x_{f,2} \quad \cdots x_{f,n-1} \quad x_{f,n} \right\} \left\{ x_{c,1} \quad x_{c,2} \quad \cdots x_{c,m-1} \quad x_{c,m} \right\} \right\}^T \tag{3.102}$$

The index $_f$ represents all the unknown (u, u_n) for the line elements that are not coastal line elements. There are n unknown, two for each interface line element and one for the line elements not on the interface. They are (f)ixed. The index $_c$ stands for (c)oastal. The number of unknown for the coastal lines, m, is exactly the number of coastal lines, because, as stated previously, a line element that is on the interface can never be a coastal line.

Grouping all the non coastal line elements in the above part of the matrix X means that the corresponding values in the A matrix will be in the first n columns. When the A matrix (and thus the corresponding B_t matrix) is filled by starting on the first row and writing equations for the coastal line elements first, a upper left matrix is created that never needs to be calculated for the same aquifer. That this values are written in the upper left part of A has another advantage. When later a sheet pile wall is inserted the size of A will increase. There is no need to set up a new array with the new size, because the existing matrix can just be resized. Copying from one to another array is in that way bypassed. A now has the following structure:

$$A = \begin{bmatrix} \begin{bmatrix} a_{f1,f1} & \cdots & a_{f1,fn} \\ \vdots & \ddots & \vdots \\ a_{fn,f1} & \cdots & a_{fn,fn} \end{bmatrix} & \begin{bmatrix} a_{f1,c1} & \cdots & a_{f1,cm} \\ \vdots & \ddots & \vdots \\ a_{fn,c1} & \cdots & a_{fn,cm} \end{bmatrix} \\ \begin{bmatrix} a_{c1,f1} & \cdots & a_{c1,fn} \\ \vdots & \ddots & \vdots \\ a_{cm,f1} & \cdots & a_{cm,fn} \end{bmatrix} & \begin{bmatrix} a_{c1,c1} & \cdots & a_{c1,cm} \\ \vdots & \ddots & \vdots \\ a_{cm,c1} & \cdots & a_{cm,cm} \end{bmatrix} \end{bmatrix} \tag{3.103}$$

In the A matrix only 3 of the 4 zones need to be calculated over and over. When the number of non coastal lines is much larger than the number of coastal line elements a serious reduction is achieved.

A similar approach is to be followed for the B_t and Y matrices. B_t will have as many rows as there are equations available, to be more precise $(m + n)$. The number of columns, k, is the number of coastal lines that are not on the interface. For line elements that are the interface both u and u_n are unknown and therefore they are in the X matrix. As for X, Y can be divided in two zones, a first zone containing all the non coastal line elements and in the second all the coastal line elements.

$$Y = \left\{ \left\{ y_{f,1} \quad y_{f,2} \quad \cdots y_{f,k-1} \quad y_{f,k} \right\} \left\{ y_{c,1} \quad y_{c,2} \quad \cdots y_{c,m-1} \quad y_{c,m} \right\} \right\}^T \tag{3.104}$$

This results in a similar structure for $B_t t$:

$$B_t = \begin{bmatrix} \begin{bmatrix} bt_{f1,f1} & \cdots & bt_{f1,fk} \\ \vdots & \ddots & \vdots \\ bt_{fn,f1} & \cdots & bt_{fn,fk} \end{bmatrix} & \begin{bmatrix} bt_{f1,c1} & \cdots & bt_{f1,cm} \\ \vdots & \ddots & \vdots \\ bt_{fn,c1} & \cdots & bt_{fn,cm} \end{bmatrix} \\ \begin{bmatrix} bt_{c1,f1} & \cdots & bt_{c1,fk} \\ \vdots & \ddots & \vdots \\ bt_{cm,f1} & \cdots & bt_{cm,fk} \end{bmatrix} & \begin{bmatrix} bt_{c1,c1} & \cdots & bt_{c1,cm} \\ \vdots & \ddots & \vdots \\ bt_{cm,c1} & \cdots & bt_{cm,cm} \end{bmatrix} \end{bmatrix} \tag{3.105}$$

3.6 Simple example

In this example, a very basic aquifer will be dealt with. It consists out of two zones and 5 boundary elements as shown in figure (3.13). Boundary elements 0 and 1 are on the coast, and therefore they have a constant head condition ($\bar{u}$). Boundary elements 3 and 4 provide inflow because of a natural elevation. For those boundary elements $\bar{u}_n$. Zone I and II (each with their own transmissivity) have one boundary element in common, called the interface and that is boundary element 2.

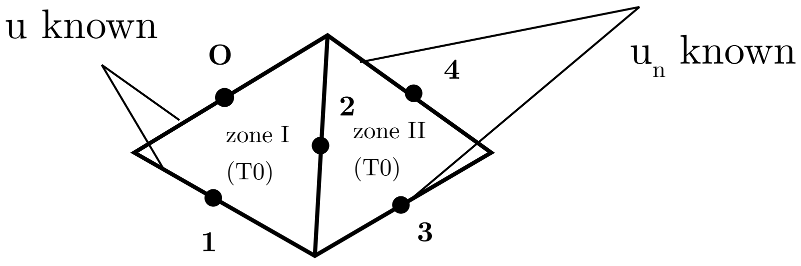

Figure 3.13: Multi-zone body

There are 6 equations (3.76) that can be written. One equation for every node on Γ and two for every node on the interface. Boundary elements 0 and 1 are only in direct contact with each other and the interface, therefore:

$$h_{00} \cdot \bar{u}^0 + h_{01} \cdot \bar{u}^1 + h_{02} \cdot u^{2,I} = g_{00} \cdot u_n^0 + g_{01} \cdot u_n^1 + g_{02} \cdot u_n^{2,I} \tag{3.106}$$
$$h_{10} \cdot \bar{u}^0 + h_{11} \cdot \bar{u}^1 + h_{12} \cdot u^{2,I} = g_{10} \cdot u_n^0 + g_{11} \cdot u_n^1 + g_{12} \cdot u_n^{2,I} \tag{3.107}$$

In this equation h_{xy} is calculated from (3.97) or (3.92) and $g_{x,y}$ from (3.96) or (3.89). $_x$ and $_y$ represent the boundary elements considered. In $u^{2,I}$ and $u_n^{2,I}$, I represents zone I. For the interface two equations can be written, one that expresses the relation with zone I and a second with zone II:

$$h_{20} \cdot \bar{u}^0 + h_{21} \cdot \bar{u}^1 + h_{22} \cdot u^{2,I} = g_{20} \cdot u_n^0 + g_{21} \cdot u_n^1 + g_{22} \cdot u_n^{2,I} \tag{3.108}$$
$$h_{22} \cdot u^{2,II} + h_{23} \cdot u^3 + h_{24} \cdot u^4 = g_{22} \cdot u_n^{2,II} + g_{23} \cdot \bar{u}_n^3 + g_{24} \cdot \bar{u}_n^4 \tag{3.109}$$

And for the boundary elements in the second zone:

$$h_{32} \cdot u^{2,II} + h_{33} \cdot u^3 + h_{34} \cdot u^4 = g_{32} \cdot u_n^{2,II} + g_{33} \cdot \bar{u}_n^3 + g_{34} \cdot \bar{u}_n^4 \tag{3.110}$$
$$h_{42} \cdot u^{2,II} + h_{43} \cdot u^3 + h_{44} \cdot u^4 = g_{42} \cdot u_n^{2,II} + g_{43} \cdot \bar{u}_n^3 + g_{44} \cdot \bar{u}_n^4 \tag{3.111}$$

Further, for boundary elements on the interface the following is known, because of the continuity of potential and flux:

$$u^{2,I} = u^{2,II} = u^2 \tag{3.112}$$

$$u_n^{2,I} = -\frac{k_{II}}{k_I} \cdot u_n^{2,II} = -k_{I,II} \cdot u_n^{2,II} = -k_{I,II} \cdot u_n^2 \tag{3.113}$$

These 6 equations can be written as one matrix equation. As explained in section (3.5.1), The matrix equation $A \cdot X = B_t \cdot Y$ will be constructed without first constructing $H \cdot u = G \cdot u_n$. Further more A, X, B_t and Y will be filled in such a way that the elements that never change are grouped as is explained in section (3.5.2). One possible X and Y vector could thus be:

$$X^T = \left\{u^2, u_n^2, u^3, u^4, u_n^0, u_n^1\right\} \tag{3.114}$$

$$Y^T = \left\{u_n^3, u_n^4, u^0, u^1\right\} \tag{3.115}$$

As it is supposed to be, X holds all the unknown and Y the unknown. The matrix A and B_t are thus:

$$A = \begin{bmatrix} h_{02} & -g_{02} & 0 & 0 & -g_{00} & -g_{01} \\ h_{12} & -g_{12} & 0 & 0 & -g_{10} & -g_{11} \\ h_{22} & -g_{22} & 0 & 0 & -g_{20} & -g_{21} \\ h_{22} & -g_{22} \cdot k_I/k_{II} & h_{23} & h_{24} & 0 & 0 \\ h_{32} & -g_{32} \cdot k_I/k_{II} & h_{33} & h_{34} & 0 & 0 \\ h_{42} & -g_{42} \cdot k_I/k_{II} & h_{43} & h_{44} & 0 & 0 \end{bmatrix} \tag{3.116}$$

$$B_t = \begin{bmatrix} 0 & 0 & -h_{00} & -h_{01} \\ 0 & 0 & -h_{10} & -h_{11} \\ 0 & 0 & -h_{20} & -h_{21} \\ g_{23} & g_{24} & 0 & 0 \\ g_{33} & g_{34} & 0 & 0 \\ g_{43} & g_{44} & 0 & 0 \end{bmatrix} \tag{3.117}$$

This means that for every element g_{ij} and h_{ij}, a check should be carried out in order to see if the element should be on the left or on the right side of the equality sign. If it changes side, a - sign is introduced. The position where it will be stored in A or B_t depends of the position of u or u_n in X or Y. All the values of Y are known and B can hence, B can be calculated as $B = B_t \cdot Y$. The formulation $A \cdot X = B$ has now been derived and can be solved for the vector X using Gauss elimination.

The third objective of this thesis requires the implementation of a sheet pile wall. A sheet pile wall can only be placed on the coast line, here boundary elements 0 and 1. They can thus never affect the values of h_{ij} and g_{ij} when both elements i and j are not a coastal boundary element. Figure (3.14) shows a possible sheet pile wall that affects both the boundary elements 0 and 1. The original boundary elements are shortened and their boundary condition changes to a known flux of 0. Two extra boundary elements need to be generated in order to make the zone closed again. The boundary conditions of 5 are the same as the original of 0 and the same happens for element 6 with the properties of 1.

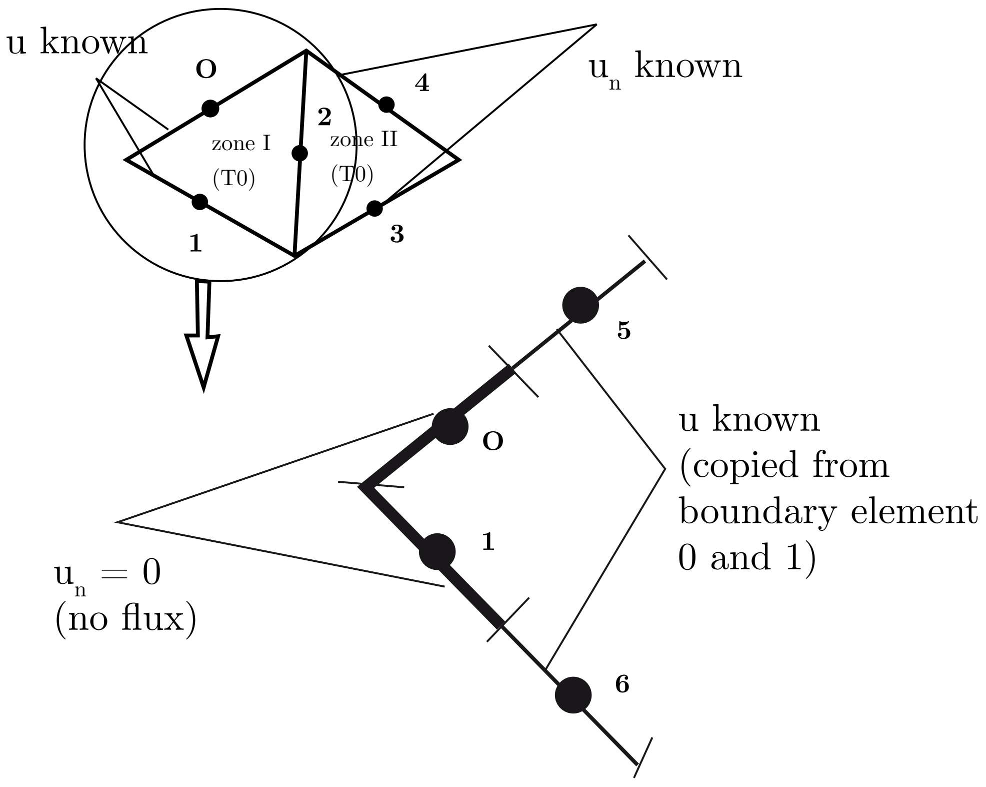

Figure 3.14: Multi-zone body (detail)

Two extra boundary elements bring along two extra unknown, but create two extra equations at the same time. Hence, X and Y will grow with two elements and they are now:

$$X^T = \{u^2, u_n^2, u^3, u^4, u^0, u^1, u_n^4, u_n^6\} \tag{3.118}$$

$$Y^T = \{u_n^3, u_n^4, u_n^0, u_n^1, u^5, u^6\} \tag{3.119}$$

X and Y have only changed for the coastal lines. The same happens for the A and B_t matrices where the relationship between two not coastal elements remains the same. They do thus not need to be recalculated over and over.

Chapter 4

Combined use of genetic algorithm and boundary element method

This chapter will explain how the genetic algorithm and the boundary element method are combined, it is how the genetic algorithm uses the boundary element method. From the previous chapters it is clear that a lot of calculations need to be carried out over and over. The calculation work carried out is already limited by calculating A and B_t without first calculating H and G and by only calculating the new elements of A and B_t. In the following section two memories will be introduced to further minimize the calculation load. After that a scheme is given that shows all the functions used in the algorithm. From this scheme the reader should understand exactly how the boundary element method is used by the genetic algorithm. For the full details of the algorithm the reader is referred to the back of this thesis.

4.1 Further minimization of the calculation work

4.1.1 Well memory

Finding out in what zone the well is located is a long procedure. It first needs to go through all the boundary elements to discover the elements around the well. Doing so it will find lines that in the worst case all belong to two zones. To find out in which of both zones the well is located also the neighbours of the last array of lines need to be found. This work is rather long and especially inefficient because the well can have maximum two degrees of freedom for its position (x and y) coordinate. When both are variable the number of different chromosomes for the well position is $2^\lambda \cdot 2^\lambda$. When only x or y is allowed to variate this number is only 2^λ. For a chromosome length of 8 this means 65536 or 256 possible well positions, resp.

Executing 10 trials each having a population size of 50 and being generated 100 times, thus resulting in 50000 fitness calculations it becomes clear that, especially in the case of one degree of freedom, storing the well chromosomes and their zone number will reduce the calculation time required.

In the case that x and y are not allowed to variate, their zone number should only be calculated once.

4.1.2 Chromosomes memory

In order to decrease the calculations that need to be carried out, the algorithm is provided with a memory. At the end of every generation the chromosomes that were created for the first time are stored in the memory, accompanied by the fitness of the chromosome. For every run it can then be checked if the chromosome has already occurred, and if so, it's fitness function does not need to be calculated anymore. When the chromosome has never been generated, then its fitness function will be calculated and stored away in the memory.

For example when working with two variables (Q_1 and Q_2 for example), each having a chromosome length of 8. There are in this case $(2^8) \cdot (2^8) = 65536$ different combinations possible. When 10 trials are executed, with a population size of 50 and 100 generations are carried out per trial, in average more than half of the 50000 calculations can be skipped because the fitness value was stored in the memory of the genetic algorithm. This also leads to a time reduction of 50%.

The advantage of memory is more noticeable for:

- a higher number of trials,
- shorter chromosomes (λ) (number of different chromosome possibilities $\approx^{NOV}$) and
- less variables, NOV, (number of different chromosome possibilities $\approx 2^\lambda$)

NOV is the number of variables.

4.2 Schema

Figures (4.1) and (4.2) shows how the boundary element method and the genetic algorithm are combined, or how the genetic algorithm uses the boundary element method to calculate the fitness it requires for its evolution. In the scheme the pre- and post processor are not included. The statistical data that is stored is also left out in order not to complicate the scheme. The functions mentioned in the scheme are the names as they are used in the algorithm. An out print of the algorithm (once again without pre- and postprocessor) is added to the back and the functions referred to are found in appendix (B).

Before the trials are started the input data is processed, this happens in the *CalculateInput* function. The length of the lines and the absolute coordinates of the nodes are calculated. Based upon the characteristics of every line, i.e. if the line is on the interface or on the coast the matrix X and Y are set up. This is done by the functions *CalculateUplaatsX* and *CalculateUplaatsY*:

Based upon the position of every line in X and Y, the arrays A and Bt are filled ($X \cdot A = Y \cdot B_t$). They are filled, as explained before in such a way that all the elements for non coastal boundary elements are grouped and can be used later on, without recalculating A and B_t over and over. A final function that is called is *CalculateLinOrderAndCumulLineEnd*. This function goes through all the boundary elements, finds out what lines are on the coast and finds out how they are in counterclockwise (anticlockwise) direction. This is necessary to know what boundary elements will be affected by placing a sheet pile. The order is the same during all runs.

For every trial a population of chromosomes (existing of subchromosomes) is generated by the function *generatePopulation*. The population size is one of the parameters of that function, together with the number of subchromosomes and the length of every subchromosome. For this first population the goal is to decide what exactly the fitness of the chromosome is. Before starting the calculations for every chromosome in the population, it is checked if the chromosome has never been calculated before. Every chromosome that was calculated before is stored in a memory together with its fitness. The fitness can, in the case of second occurrence, simply be read from the memory, without recalculation. In the case that the chromosome has never been generated before, its fitness will be calculated. The first step of this calculation is to find out if a sheet pile wall needs to be included. In the case this is the beginning and endpoint of the sheet pile wall should be calculated. The function *beginAndEndSpw* takes care of this. This function takes at least one chromosome as an argument. For the chromosomes that are passed a double value is calculated. When one chromosome is passed, the begin point of the sheet pile wall is calculated, and the length is constant. In the case two chromosomes are passed and the beginning and end points are calculated. This function also looks on which boundary element these beginning and endpoint are located. The *fillAffectedLines* finds out what boundary elements are affected by the sheet pile wall. Being affected means that the sheet pile is at least for one point on the boundary element.

The most important function when a sheet pile wall needs to be included is the *fillArray-WithValues* function. This function recalculates the boundary elements on the coast (length, node coordinates, boundary condition). This function thus adds one or two or no boundary elements. More details about this function can be found in the previous section.

Before the boundary element method is executed the zone for each well is calculated. A separated memory is available for the well positions. Every well position and corresponding zone, previously calculated is stored in the memory and when called a second the zone can be read from the memory without going through all of the boundary elements again.

All the necessary data is calculated now and the boundary elements can be triggered. The only purpose of the boundary element method is to calculate the fitness of the chromosome. Since new boundary elements might be added the X and Y vectors need to reviewed. They were filled in such a way that the coastal boundary elements were added to the end of the vector, and thus only the last part needs to be recalculated. *AddToPlaatsXandY* takes care of this job. Before the solution for $(A \cdot X = B_t \cdot Y)$ can be yield A and B_t should be filled. All the elements of A and B_t that express the relation between two elements that are not on the coast can just be copied (*CopyKnownValuesOfAandBt*) and the other values need to be calculated (*CalculatedAandBt*) since they might have changed or never have been calculated before. From Y and B_t, B can be calculated $(B = Bt \cdot Y)$ by function *CalculateB*. Before the function *SolveIntelligent* solves the equations $(A \cdot X = B)$ (using Gauss elimination), the influence of the well is added by *WellInfluenceSmart*. The final step of the boundary element method is to sort the unknown (u, u_n) that were found, based upon the type of boundary condition they represent.

All the previous work done was carried out to calculate on double value, namely the fitness of the chromosome. The void *CalculateFitnessFunction* calculates the fitness for the chromosome and stores it in the memory together with the inflow characteristics. This is done by the *fillCalculatedChromosomesAndInflowCharacteristics* function.

The entire cycle, starting with checking if the chromosome has ever been calculated before until storing the chromosome with its calculated fitness function and inflow characteristics is now done for every chromosome in the population. As a result, all chromosomes have now been assigned fitness and this fitness will be used to create a new generation. When elitism is used the fittest chromosome is stored before selection takes place, in order not to lose the fittest result. From all the chromosomes in the population a selection is made. This can happen in three ways. Using roulette wheel selection, ranking or by tournament method. A new population (with the same size) is selected and then chromosomes can undergo crossover (function *crossOver*) by chance. After chromosomes crossed over they are also submitted to mutation (function *mutation*). When elitism is used the fittest function is now added to the population again (deleting the last chromosome).

For this newly created population of chromosomes the fitness function is calculated again as described above. This is done for the number of generations. After the last generation a very fit chromosomes should have survived and the fittest is returned as the (optimum) solution.

```
//Scheme without pre and post processor

//1. To be called only once
    CalculateInput(…)
    CalculateUplaatsX(…)
    CalculateUplaatsY(…)
    CalculateAandBStart(…)
    CalculateLineOrderAndCumulLineEnd(…)

//2.  For every trial

    //2.a) Generate the initial population
    GeneratePopulation(…)

    //2.b) Calculate the fitness function for the chromosomes in the original population (γ = 0)

        BLOC A //block A calculates the fitness of each chromosome, using the boundary
        element method.

    //2. c) For every generation (γ = 1 .. NOG)

            2.c.1) If elitism is used: store fittest
            2.c.2) Selection (Roulette wheel, ranking, selection constant)
            2.c.3) Crossover(…)
            2.c.4) Mutation(…) and flip(…) //flip = antimetathesis  void
            2.c.5) If elitism is used: bring fittest back into the population

            BLOC A

            //next generation (→ 2.c)

    //next trial (→ 2)
```

Figure 4.1: Combined use of genetic algorithm and boundary element method

```
//BLOC A

//For all chromosomes in the population
    //1. Check if this chromosome has been calculated previously
    checkIfNeedsToBeCalculated(…)

        //1.a) should not be calculated → Read from memory and store fitness

        //1.b) should be calculated

            //1.b.1)  Check if a sheet pile wall is implemented

                //1.b.1.a) should not be calculated → GO TO 1.b.2)

                //1.b.1.a) should be calculated
                    beginAndEndSpw(…)
                    fillAffectedLines(…)
                    fillArrayWithValues(…)

            //1.b.2)  For all wells included:

                //1.b.2.a) should not be calculated → Read from memory

                //1.b.1.a) should be calculated
                    findOutZoneIntellegent(…)
                    fillCalculatedWellPosition(…)

            addToUplaatsXandY(…)
            copyKnownValuesOfAandBt(…)
            calculateAandBt(…)
            calculateB(…)
            wellInfluenceSmart(…)
            solveIntellegent(…)
            reorderSmart(…)
            calculateFitnessfunction(…)
            fillCalculatedeChromosomesAndInflowCharacteristics(…)

    //next chromosome → go to 1)
```

Figure 4.2: Combined use of genetic algorithm and boundary element method - A Block

Chapter 5

Application examples

The aquifer studied in this master's thesis has been studied before by Petala [24]. Figure (5.1) shows this aquifer and its boundary conditions. There are two zones, both with their own transmissivity T. $T_0 = 0.003$ m/s and $T_1 = 0.001$ m/s.

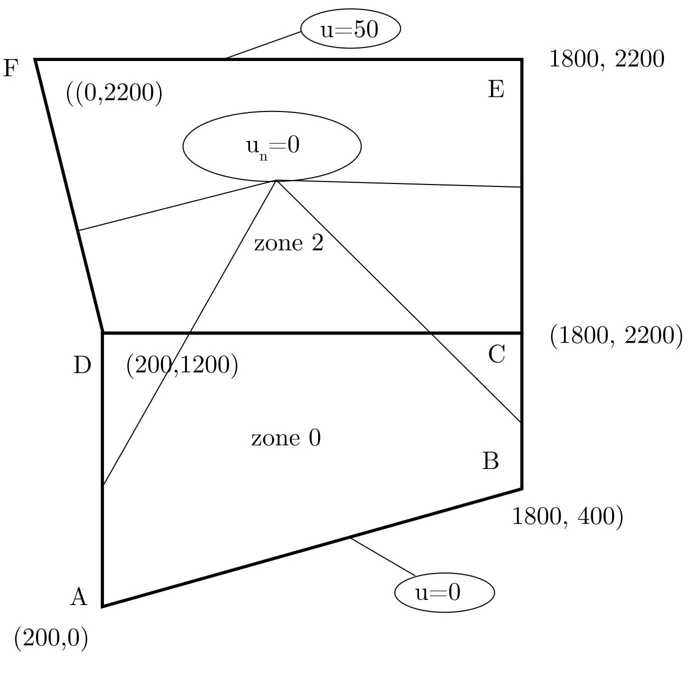

Figure 5.1: Aquifer studied

Line AB represents the coastline. Lines BCE and ADF are impermeable and line FE allows inflow from fresh water due to natural elevation. The only way for saline water to enter the aquifer is from the coast, through line AB. Natural flow is from zone 1 to zone 0 because of the height difference. 50 meters (fresh water) to 0 meters (saline water equivalent).

Before the genetic algorithm can use the boundary element method, the aquifer needs to be simplified to a chain of boundary elements that represent the aquifer. Lines AB, BC and DA belong only to zone 0, lines CE, EF and FD only to zone 1 and line CD belongs to both zone 1 and 0. This line is the interface of both zones. All lines now need to be subdivided in boundary elements and the subdivision should be high enough so that the solution is accurate enough so that no extra convergence of the results would be obtained by subdividing the boundary elements even more. This is tested by increasing the number of boundary elements and finding out what is the influence for the results found. When the increase of the number of boundary elements does not lead to improvements of the accuracy of the solutions calculated, called convergence, then a sufficient subdivision is reached. The more boundary elements used the longer the calculation time required.

The input of the aquifer counts 45 boundary elements. Line AB is discretized in 8 elements, as is the interface. BC counts 4, CE 5, FE 9, FD 5 and AD 6 elements.

5.1 Objective 1: optimal well flow for two fixed wells

In this case the developed software is used to calculate the optimal well configuration for two wells. Both wells have fixed coordinates, the first well, $W_1 = (500, 700)$ and the second $W_2 = (1400, 800)$. In a first attempt the flow is presumed to be between 0.01 and 0.05 m^3/s for both wells. The input parameters used are shown in table (5.1).

PS	50	P_c	0.35
NOG	100	$P_m = P_f$	0.111
NOT	10	ϵ	TRUE
		Selection type	Roulette wheel

Table 5.1: Input parameters

There are two unknown Q_1 and Q_2 each representing a chromosome. The length of the chromosome depends on the accuracy required and can be calculated according to eq. (2.12):

$$\lambda_{min} \geq \frac{\ln\left(\frac{0.05 - 0.01 + 0.0001}{0.0001}\right)}{\ln 2} = 8.64 \tag{5.1}$$

The chromosome length for both variables will be taken to be 9. The total combination of different chromosomes is thus $2^9 \cdot 2^9 = 2^{18} = 262144$. Even with two chromosomes with a short chromosome length, it becomes clear that the use of a genetic algorithm could come in use to reduce the calculation work, that is calculating the solution for the 262144 possibilities when the traditional way of solving the problem is used. One trial only calculates, at maximum 5000 candidate solutions. At maximum only 1.91% of the posibilities are calculated, and by using the memory the calculation works will even be less. $P_m = P_f$ is calculated as suggested: $1/\lambda = 0.111$. The fitness function used is the proposed fitness function by Katsifarakis and Petala [8], Φ_K:

$$\begin{aligned}\Phi_K &= \sum_{i=1}^{W} q_{w,i} - (70 \cdot \kappa - 7\sum_{i=1}^{\kappa} q_{w,i} \cdot l_i) \\ &= \sum_{i=1}^{W} q_{w,i} - (70 \cdot \kappa - 7\sum_{i=1}^{\kappa} T_i \cdot u_{n,i} \cdot l_i)\end{aligned} \tag{5.2}$$

The idea is to have high fitness when a lot of water is extracted from the wells. However, when seawater intrusion takes place, the fitness should be lowered again. In eq. (5.2), W is the total number of wells and κ represents the number of lines where u_n is positive (there is seawater intrusion). The summation only includes the κ elements boundary elements that have inflow.

5.1.1 Results

10 trials were carried out, no absolute optimum, but 10 very fit solutions were found. The fitness ranged between $\Phi_K \in [0.0689, 0.0695]$. The combinations of Q_1 and Q_2 are shown in table (5.2).

Trial	0	1	2	3	4
Φ_K	0.06900	0.06916	0.06892	0.06932	0.06924
Q_1	0.03059	0.03137	0.03059	0.03121	0.03145
Q_2	0.03841	0.03779	0.03834	0.03810	0.03779
G_{max}	85	19	77	71	83
Trial	5	6	7	8	9
Φ_K	0.06892	0.06947	0.06908	0.06939	0.06947
Q_1	0.03114	0.03121	0.03114	0.03106	0.03137
Q_2	0.03779	0.03826	0.03795	0.03834	0.03810
G_{max}	98	80	93	81	45

Table 5.2: Objective 2: Results for Φ_K, Q_1, Q_2 and G_{max}

The solutions were found sometimes near last generations. This indicates that there has not been absolute convergence and maybe the number of generations should be increased. In the following section the influence of the memory and the reduction in calculation will be discussed and then the exact solutions for this objective will be calculated.

5.1.2 The use of the memory per trial

Including a memory for the position of the well is here very effective, because only two calculations are required. Once for the position of W_1 and once for W_2. The position is fixed and the zone found during the first calculation can thus be used over and over. The number of well positions stored in the memory is 2, and from that moment on no new wells will be calculated.

Figure (5.2) shows the evolution of the number of calculations that are saved by using a memory as function of the generation for the first trial. During all generations, chromosomes that occur for the first time are stored together with their fitness. When the same chromosome is generated again (by crossover, mutation, antimetathesis and selection) the fitness function is just copied and its calculation can be skipped. As is to be expected there is a lot of spread, but the general trend is that the number of calculations that are saved during one generation increases as function of the generation. For the first trial alone 602 calculations were saved. This is a reduction of 12.04% compared to the calculations required when no memory was build in.

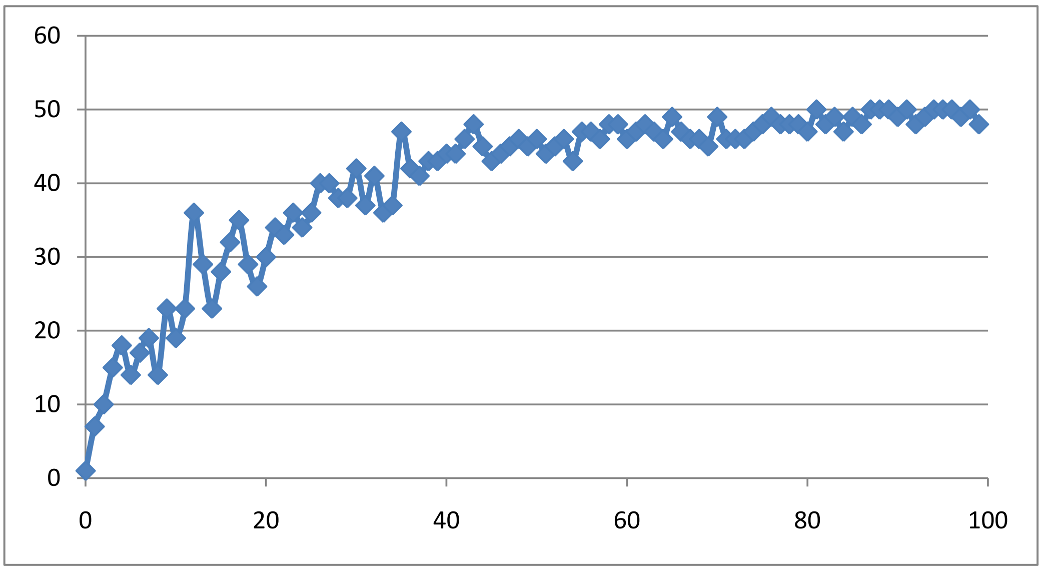

Figure 5.2: Calculations saved because of memory as function of the generation during the first trial

The software is programmed in such a way that it can perform different trials in order to achieve a statistical insight of the solutions obtained. The memory is not cleared after a trial is executed and the genetic algorithm can thus use what it learned from previous trials. Figure (5.3) shows the evolution of the number of calculations saved for the first 10 trials. In the 5th trial already 946 (18.92%) of all calculations are saved, and during the last trial the number of calculations saved is already 1381 (71.98%). The genetic algorithm is thus a good student or at least has a very good memory. The same excercise was carried out with two chromosomes of 8 genes. In the 5th trial already 55.06%, and during the last trial 71.98. This thus shows that shorter chromosomes will, drastically reduce the calculation. From ± 15 minutes ($\lambda = 9$) to ± 8 minutes ($\lambda = 8$).

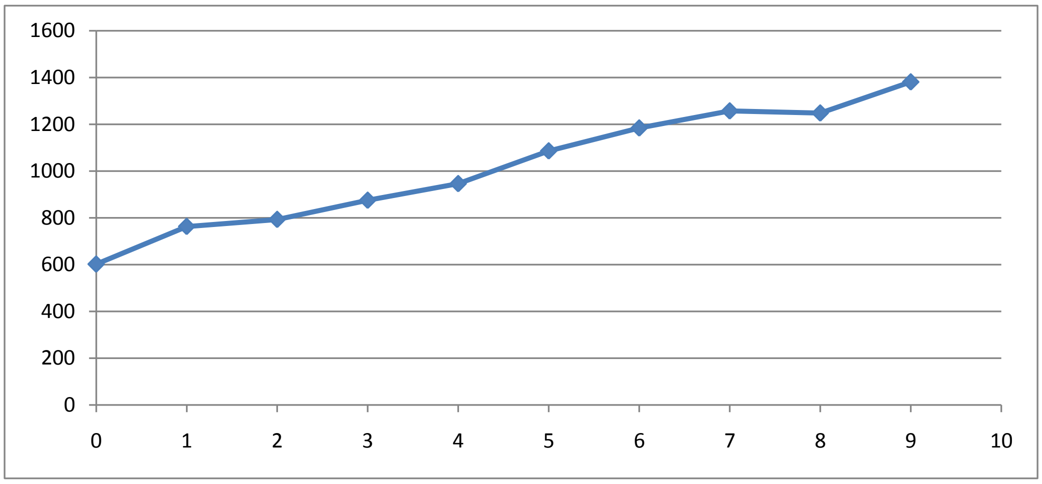

Figure 5.3: Calculations saved because of memory as function of the trial

5.1.3 Reducing calculation time for A and B_t matrix

Since there is no sheet pile wall included in this stage, the boundary elements will always remain the same. This means that the A and B_t matrix will always have the same values. The influence of the wells is added by superposition after calculating A and $B = B_t \cdot Y$. The script was thus optimized to handle this and the A and B_t matrix will thus only have been calculated once and not 5000 times per trial.

5.1.4 From good to optimum results

As stated before, a genetic algorithm should be used to find very fit solutions, but it is not sure that the solutions found are the absolute optimal solutions. Around the solutions found a traditional search should be used to find the optimum solution. Here a different approach will be used. After the first execution of the algorithm a second execution will take place, to fine tune the results.

From table (5.2) it is known that $Q_1 \in [0.03059, 0, 03145]$ and $Q_2 \in [0.03779, 0.03841]$. A second set of 10 trials will now be executed between those limits. $Q_1 : 0.030 \rightarrow 0.032$ and $Q_1 : 0.037 \rightarrow 0.039$. ΔP is left unchanged and the minimum chromosome length is calculated to be 4.24 and thus $\lambda = 5$, for both chromosomes. The total number of different chromosomes possible is 1024. These 1024 possibilities are smaller than the 5000 chromosomes that will be calculated every trial, and it is thus very likely that the results for all trials will be the same. The results are listed in table (5.3).

Trial	0	1	2	3	4
Φ_K	0.06958	0.06958	0.06958	0.06958	0.06958
Q_1	0.03129	0.03135	0.03129	0.03129	0.03135
Q_2	0.03829	0.03823	0.03829	0.03829	0.03823
G_{max}	1	4	15	6	5
Saved	4019	4964	4994	5000	5000
Trial	5	6	7	8	9
Φ_K	0.06958	0.06958	0.06958	0.06958	0.06958
Q_1	0.03135	0.03129	0.03135	0.03129	0.03135
Q_2	0.03823	0.03829	0.03823	0.03829	0.03823
G_{max}	11	4	0	3	5
Saved	4999	5000	5000	5000	5000

Table 5.3: Objective 2: fine tuned results for $\Phi_K, Q_1, Q_2, G_{max}$ and the number of calculations saved per trial

The fitness found is ten times the same, and even higher than was obtained before. It was surprising to find out that there are 2 chromosomes that are identically as fit, because there are 2 solutions found that are fit: ($Q_1 = 0.03129, Q_2 = 0.03829$) and ($Q_1 = 0.03135, Q_2 = 0.03823$). This is not the result of rounding mistakes as it was checked that both fitnesses are exactly the same, no matter how many digits after the comma were used. Exactly 5 of each chromosomes were found to be as fit, which shows again the statistical property of using genetic algorithms.

The memory size after all the runs was exactly 1024, the theoretical number of possibilities. So it is impossible that there was one chromosome that was fitter but never was selected. The last table also shows how many calculations were saved. From the fourth run on the number of calculations saved is 5000 except for trial number 5, where the algorithm selected a chromosome that had never been generated before.

The best solution is always found in the first 16 generations and thus the number of generations could safely be reduced to 25. This would lead to a calculation time that is about 4 times shorter. In this case this would mean that the calculation time would go from 23 seconds to approximately 6 seconds. It is thus very clear that the shorter the chromosome is, the shorter the calculation time will be, where a memory for the previous results is used.

It should be mentioned that no sea water intrusion took place in the solutions calculated.

5.2 Objective 2 and 3: implementation of a sheet pile wall - Input parameters

In this chapter the objective will be to provide a water recourse manager with relevant information for his decision making. This manager wants to extract more fresh water from the two existing wells used in objective 1. Therefore he wants to know if the use of a sheet pile could be beneficial.

For a given sheet pile length, the best optimum combination of q_1, q_2 and s_o will be researched. q_1 and q_2 is the flow extracted resp. from the first well, W_1, and the second, W_2. s_o is the begin point of the sheet pile wall on the coastline. The coastline goes from $s = 0$ (most left) to the end of the coast l_c (most right, and (l)ength of the (c)oast). Three variables thus exist and each candidate solution will be represented by a chromosome that has three sub chromosomes.

q_1 and q_2 are supposed to vary between 0.01 m^3/s and 0.05 m^3/s. More detailed information is required to make a better estimation of what will be the real range, but since no details are known for the aquifer studied this range is taken. In a first attempt ΔP between two candidate solutions is taken to be 0.001 m^3/s and as a result $\lambda_{1,2} = 6$ for both sub chromosomes.

The beginning position of the sheet pile wall is represented by the third sub chromosome. The length of the coast, l_c is 1649.34 m and the begin point can thus vary between 0 and $l_c - l_{spw}$ (this is computed automatically). Taking ΔP to be 20 m, $\lambda_3 = 5$ is sufficient when the sheet pile wall is 1000 meter long. The total chromosome has thus a length of 17 genes and therefore the mutation probability is taken to be $1/17 = 0.0588 \approx 0.6$.

In what follows the trials will be executed with: $PS = 50, NOG = 100, NOT = 10, P_c = 0.35, P_m = P_f = 0.06$ and $\epsilon = 1$ unless mentioned otherwise. Mutation and antimetathesis both take place for every generation. The algorithm developed has the possibility to run several trials. Since genetic algorithms are a statistical process it is good to know what happens if it is run multiple times. A low fitness for one trial can be excluded compared to the average. This approach is also very effective when combined with a memory because a lot of calculations can than just be skipped. The calculations carried out next are for a sheet pile wall with length 1000 m.

The fitness function used is the same as in the first objective and the results listed all have no saline water inflow.

5.2.1 Different selectors

The developed software allows the user to use three selection techniques: Roulette wheel selection, ranking and tournament selection. In this first section, all three will be used. The techniques, ranking and tournament selection require the input of a constant. Ranking constant will be carried out with $KK = 2, 3$ and 4 and tournament selection with $C = 15, 25$ and 35. The results are listed in tables (5.4) and (5.5).

case	$q_{1,min}$	$q_{1,max}$	Δq_1	$q_{2,min}$	$q_{2,max}$	Δq_2	ϕ_{max}	ϕ_{ave}	ϕ_{min}
$KK = 2$	0.0259	0.0310	0.0051	0.0417	0.0475	0.0057	0.0733	0.0730	0.0721
$KK = 3$	0.0246	0.0322	0.0076	0.0405	0.0487	0.0083	0.0740	0.0731	0.0727
$KK = 4$	0.0259	0.0329	0.0070	0.0392	0.0475	0.0083	0.0740	0.0728	0.0721
$C = 15$	0.0240	0.0373	0.0133	0.0348	0.0487	0.0140	0.0733	0.0724	0.0721
$C = 25$	0.0233	0.0322	0.0089	0.0398	0.0487	0.0089	0.0733	0.0727	0.0721
$C = 35$	0.0233	0.0316	0.0083	0.0405	0.0494	0.0089	0.0740	0.0730	0.0721
RW	0.0246	0.0329	0.0083	0.0392	0.0487	0.0095	0.0740	0.0726	0.0721
RW	0.0233	0.0360	0.0127	0.0360	0.0487	0.0127	0.0733	0.0723	0.0721

Table 5.4: Comparison selection methods for $P_m = P_f = 0.06$ per gene - Q and ϕ

case	Times found	G_{min}	G_{max}	Σ_{min}	Σ_{max}	memory size	Duration
$KK = 2$	6	12	63	0.000951	0.005682	31503	0:15:01
$KK =$	1	4	65	0.000635	0.00411	28333	0:14:09
$KK = 4$	2	3	78	0.000635	0.005054	23857	0:10:13
$C = 15$	1	0	74	0	0.003165	39095	0:18:03
$C = 25$	3	9	86	0.000951	0.00348	37861	0:19:35
$C = 35$	2	8	90	0.000951	0.004739	36756	0:17:14
RW	1	12	95	0.000635	0.013968	34274	0:16:09
RW	1	4	90	0.04746	0.013968	34318	0:17:30

Table 5.5: Comparison selection methods for $P_m = P_f = 0.06$ per gene - Times found G, Σ, memory size and duration

From these tables it is clear that the duration is function of the memory size. Calculating the chromosome's fitness (= going through BEM) takes time. Using tournament selection is faster than roulette wheel (RW) or ranking (C), and the higher KK is, the smaller the memory size. This can be explained because it is likely that taking the best out of 4 will sooner lead to convergence than selecting 3 or 2. More of the same chromosomes will be passed to the next generation which results in less crossover and hence less new chromosomes.

When using ranking, the number of chromosomes that pass to the next generation is related to the number of different chromosomes calculated. Passing more chromosomes allows less new chromosomes to be calculated. Passing only 15 chromosomes to the next generation, seems to prevent convergence of the results. The solution space is as a result bigger. $\Delta q_1 (= q_{1,max} - q_{1,min})$ and $\Delta q_2 (= q_{2,max} - q_{2,min})$ are high compared to the results obtained when 25 and 35 chromosomes that pass. As a result the average fitness is higher for $C = 35$ than for $C = 15$.

It also seems that there is a relationship between the number of different chromosomes calculated and the range of the solutions found $(\Delta q_1, \Delta q_2)$.

5.2.2 Influence of mutation and flip probability

One question that could be posed is if it is necessary to have mutation and flipping. In the previous subsection both took place with a probability of 6/100 for every gene of the chromosome. As a result some chromosomes were affected in multiple genes at the same time, creating a totally new chromosome. Most probably the search area will be better explored because of that, but maybe convergence will be made impossible. Tables (5.6) and (5.7) show the results.

From these tables it became clear that the higher KK is, the smaller the solution space became. The same is also visible with the use of the tournament selection.

Compared to mutation and flipping per gene, tournament selection now has a much smaller memory size, bringing the total calculation time under one minute. The same can be said for roulette wheel selection, but not for tournament selection, because then refreshment takes place anyway. The number of different chromosomes calculated is lower for all three selection methods.

For both $KK = 4$ and $C = 35$, ϕ_{max}, ϕ_{ave} and ϕ_{min} are bigger when mutation and flipping takes place per gene. Therefore it can be concluded that mutation and flipping is necessary to find fit chromosomes.

case	$q_{1,min}$	$q_{1,max}$	Δq_1	$q_{2,min}$	$q_{2,max}$	Δq_2	ϕ_{max}	ϕ_{ave}	ϕ_{min}
$KK = 2$	0.0144	0.0348	0.0203	0.0348	0.0500	0.0152	0.0733	0.0714	0.0644
$KK = 3$	0.0246	0.0348	0.0102	0.0367	0.0487	0.0121	0.0733	0.0717	0.0695
$KK = 4$	0.0271	0.0341	0.0070	0.0348	0.0449	0.0102	0.0733	0.0716	0.0689
$C = 15$	0.0233	0.0322	0.0089	0.0398	0.0494	0.0095	0.0740	0.0730	0.0721
$C = 25$	0.0233	0.0322	0.0089	0.0398	0.0494	0.0095	0.0733	0.0726	0.0721
$C = 35$	0.0278	0.0329	0.0051	0.0386	0.0462	0.0076	0.0740	0.0724	0.0714
RW	0.0290	0.0322	0.0032	0.0398	0.0443	0.0044	0.0733	0.0723	0.0714
RW	0.0252	0.0329	0.0076	0.0386	0.0481	0.0095	0.0733	0.0727	0.0714

Table 5.6: Comparison selection methods for $P_m = P_f = 0.06$ per chromosome - Q and ϕ

case	Times found	G_{min}	G_{max}	Σ_{min}	Σ_{max}	memory size	Duration
$KK = 2$	3	5	36	0.0003	0.0038	2569	0:01:05
$KK = 3$	1	2	78	0.0010	0.0035	2230	0:00:52
$KK = 4$	1	1	68	0.0003	0.0028	2371	0:00:57
$C = 15$	1	14	75	0.0010	0.0038	34092	0:16:42
$C = 25$	2	0	76	0.0000	0.0032	29151	0:13:28
$C = 35$	1	12	80	0.0003	0.0032	24270	0:11:07
RW	2	23	99	0.0006	0.0041	8917	0:03:50
RW	3	0	89	0.0000	0.0035	8714	0:03:49

Table 5.7: Comparison selection methods for $P_m = P_f = 0.06$ per chromosome - Times found G, Σ, memory size and duration

5.2.3 Fine tuning the results

From the previous subsections it became clear that KK and C needed to be high enough in order to find fit candidate solutions in a small solution space. $C = 15$, $C = 25$, $KK = 1$ and $KK = 2$ will therefore not be studied any more.

In this next step the solution space will further be researched. In order not to miss possible solutions the new search space will be the widest range for q_1 and q_2 found when using $KK = 4$, $C = 35$ and roulette wheel as a selector: $q_1 = [0.023, 0.036]$ and $q_2 = [0.036, 0.050]$. Increasing ΔP to 0.0005 results in a $\lambda_{min} = 5$ for both sub chromosomes. The same is done for the begin point of the sheet pile wall: $s_0 = [180, l_c - l_{spw}]$. λ_{spw} is kept the same and now represents a ΔP of 15 meters.

The total chromosome length now became 15 and $P_m = P_f$ is taken to be $1/15 = 0.667 \approx 0.07$. The total possible number of different chromosomes is now 32728, which is in the range of the memory size that was used for $C = 35$ in the previous subsection. NOT was now set to 50, in order to have more statistical data. The results of the new trials are listed in tables (5.8) and (5.9).

case	$q_{1,min}$	$q_{1,max}$	Δq_1	$q_{2,min}$	$q_{2,max}$	Δq_2	ϕ_{max}	ϕ_{ave}	ϕ_{min}
$KK = 4$	0.0276	0.0310	0.0034	0.0419	0.0464	0.0045	0.0740	0.0733	0.0728
$C = 35$	0.0238	0.0322	0.0084	0.0405	0.0491	0.0086	0.0740	0.0734	0.0727
RW	0.0234	0.0314	0.0080	0.0414	0.0495	0.0081	0.0740	0.0736	0.0728

Table 5.8: $l_{spw} = 1000$ (fine tune) - Q and ϕ

case	Times found	G_{min}	G_{max}	Σ_{min}	Σ_{max}	memory size	Duration
$KK = 4$	16	2	99	0.00021	0.002221	26014	0:15:11
$C = 35$	5	1	98	0.000161	0.001948	32013	0:21:03
RW	10	3	99	0.000194	0.002108	29639	0:19:47

Table 5.9: $l_{spw} = 1000$ (fine tune) - Times found G, Σ, memory size and duration

From the result obtained it seems that tournament selection is to be preferred. 16 out of 50 trials have resulted in the highest fitness found, where roulette wheel only has 10 out of 50 and Ranking only half of that. From the memory size it is clear that less different chromosomes need to be calculated to get more good results compared to C and RW. ϕ_{max}, ϕ_{max} and ϕ_{min} do not give preference to one of the three selecting methods, but Δq_1 and Δq_2 again are in favor of KK, since the solution area is much smaller. As a result the selection technique used later on in this thesis will be $KK = 4$.

5.2.4 Influence of the population size and number of generations

To see if the population size has influence, it is doubled to 100. The number of fittest found was 15, so the conclusion is that the original population size was already sufficient. The calculation time stayed under 25 minutes and 470171 out of 500000 calculations were saved. The memory size was thus 29829.

Using 150 generations, the number of fittest solutions found was 19 and 19 out of 50 found their fittest solution for $\gamma > 100$. The calculation was done in less than 20 minutes, and the memory size was 28079. Therefore it can be said that in this case increasing the number of generations has a bigger impact. But the extra calculation load, not only more generations but also more different chromosomes, lead to conclusion not to increase the number of generations.

5.2.5 Interchanging mutation and antimetathesis

In [23] it was stated that mutation and antimetathesis best take place interchangingly. The algorithm developed allows the user to decide whether to do so or not because of the following surprising results as listed in tables (5.10) and (5.11)

case	$q_{1,min}$	$q_{1,max}$	Δq_1	$q_{2,min}$	$q_{2,max}$	Δq_2	ϕ_{max}	ϕ_{ave}	ϕ_{min}
$KK = 4(i = 1)$	0.0251	0.0322	0.0071	0.0405	0.0482	0.0077	0.0740	0.0732	0.0727
$KK = 4(i = 0)$	0.0264	0.0310	0.0046	0.0419	0.0473	0.0054	0.0740	0.0735	0.0728

Table 5.10: Influence of interchangingly mutation and antimetathesis for $l_{spw} = 1000$ (fine tune) - Q and ϕ

case	Times found	G_{min}	G_{max}	Σ_{min}	Σ_{max}	memory size	Duration
$KK = 4(i = 1)$	10	0	80	0	0.002285	17070	0:10:50
$KK = 4(i = 0)$	16	0	99	0	0.002381	26505	0:16:44

Table 5.11: Influence of interchangingly mutation and antimetathesis for $l_{spw} = 1000$ (fine tune) - Times found G, Σ, memory size and duration

In this tables $i = 1$ means the algorithm was run with interchangingly using mutation and antimetathesis and $i = 0$ if first mutation and then antimetathesis took place for every generation. For $i = 1$ only 10 fit results were found where for $i = 0$ the number was 16. The number of unique chromosomes was also much lower (17070 compared to 26505) so the solution area was better searched for when first applying mutation and then antimetathesis. The average and minimum fitness function were also higher when $i = 0$ and the solution area (Δq_1,Δq_2) was smaller as well. In every aspect the use of antimetathesis after mutation seemed to be better.

Because these results were surprising, the comparison was made again using 250 trials in order to be sure not to have statistical influence. The results acknowledged the results listed before. Therefore the algorithm will be used with antimetathesis after mutation.

5.2.6 Refreshment

Figure (5.4) shows the fitness evolution of 6 trials for $KK = 4$.

The fitness evolution is clearly stepped. During different generations the fitness remains constant until a fitter chromosome is created by chance: two chromosomes crossed over and generated a fitter offspring, the chromosome was mutated or underwent antimetathesis and became fitter, or a combination. From this figure it seems that the generations before a jump in fitness takes place there is a temporary reduction, but this can not be because the fittest function is always passed from one generation to another. Some trials never seem to know an increase of fitness. One idea is to refresh the population with chromosomes. Three techniques are tested:

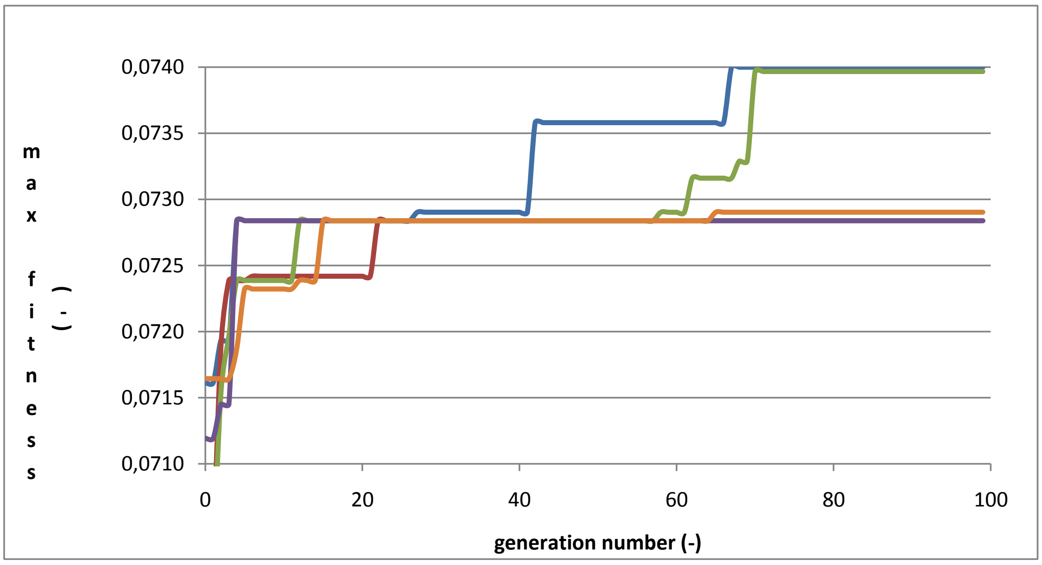

Figure 5.4: ϕ_{max} as function of γ

1. inputting new chromosomes, randomly created
2. inputting a number of mutated copies of the fittest chromosome from the last generation
3. inputting a number of flipped copies of the fittest chromosome from the last generation

All three methods have been implemented in the algorithm and can be used using roulette wheel and tournament selection. Table (5.12) shows the obtained results for three combinations carried out to see if there was a positive influence.

refresh					
Combination	Times found	ϕ_{max}	ϕ_{ave}	ϕ_{min}	memory
maxTimes = 35, new = 25	16	0,0740	0,0736	0,0728	30650
maxTimes = 35, new = 10	16	0,0740	0,0735	0,0728	28934
maxTimes = 15, new = 10	14	0,0740	0,0728	0,0737	30746
refresh with forced mutation					
Combination	Times found	ϕ_{max}	ϕ_{ave}	ϕ_{min}	memory
maxTimes = 35, new = 25	9	0,0740	0,0733	0,0728	24848
maxTimes = 35, new = 10	15	0,0740	0,0733	0,0728	25255
maxTimes = 15, new = 10	11	0,0740	0,0733	0,0728	24783
refresh with forced antimetathesis					
Combination	Times found	ϕ_{max}	ϕ_{ave}	ϕ_{min}	memory
maxTimes = 35, new = 25	14	0,0740	0,0732	0,0728	24116
maxTimes = 35, new = 10	10	0,0740	0,0732	0,0728	25027
maxTimes = 15, new = 10	13	0,0740	0,0733	0,0728	22831

Table 5.12: Influence of refreshing the population size for $KK = 4$

In the table 'maxTimes' is the number of generations that the maximum fitness is allowed not to increase. For every generation that the maximum fitness is not increasing a counter is incremented and when as high as maxTimes a number, 'new', of new chromosomes is added to the population size. Refreshing is programmed to take place after selection, mutation and antimetathesis took place. Refreshing with new chromosomes gave the best results. As was to be expected, more different chromosomes were created for a lower maxTimes and when a lot of new chromosomes were added.

Compared to the results obtained without refreshing (tables 5.8 and 5.9) ($\phi_{max} = 0.074, \phi_{ave} = 0.0733, \phi_{min} = 0.0728$, Times found = 16 and memory = 26014) no improvement was made. Refreshing with forced mutation and with forced antimetathesis is therefore not interesting. Refreshing with new chromosomes worked as well when the number of maxTimes allowed was high enough. When after 15 times the population was replenished with new chromosomes the number found was only 14, which indicates that the convergence progress was disturbed.

Since no real improvement was noticed no refreshing will take place in the following calculations.

5.3 Objective 2 and 3: implementation of a sheet pile wall - comparison for 5 different lengths

In the previous section, the use of one sheet pile was used. In real life it is not sufficient to only know results for one length. The management will want to make a comparison between different possibilities. For the aquifer studied here it is impossible to make detailed calculations but it is possible to make a comparison between different sheet pile wall lengths. In what follows the algorithm will be used to calculate 4 more sheet pile walls with a length of 800, 600, 400 and 200 m. The approach that leads to the optimum results is the same as applied before.

In a first step the algorithm is run for a search space that for sure holds the optimum solution. This will lead to a candidate solution space that is much smaller than the original search space. In a second step, the new search space will be searched again, but now with a higher precision (ΔP).

The initial search space has three variables Q_1, Q_2 and s_0. s_0 can range between the begin of the coast ($s = 0$) and $l_c - l_{spw}$ and the flow varies between 0.01 and 0.05 m^3/s in each well. $\Delta P = 0.002$ m^3/s for the flow and 20 m for the sheet pile wall. The sub chromosomes should then at least have a length of 5, 5 and 6 genes and the total chromosomes length is 16. In the case of the sheet pile wall of 200 m, the chromosome has one more gene to meet this step of 20 m. $P_m = P_f = 1/16\ (1/17) = 0.0625(0.06)$.

5.3.1 Sheet pile wall of 1000 m

The results for a sheet pile of length 1000 m are listed in table (5.13). They are the detailed version of the calculations in table (5.8) for $KK = 4$. In this table NOO is the number of occurrences. The total number of occurrences is 50.

$NOO(-)$	$\phi(-)$	Q_1(m^3/s)	Q_2(m^3/s)	$s_{0,min}(m)$	$s_{0,max}(m)$
16	0.07400	0.02761	0.04639	649.24	649.24
3	0.07397	0.02803	0.04594	649.24	649.24
1	0.07355	0.02761	0.04594	649.24	649.24
1	0.07345	0.02887	0.04458	649.24	649.24
1	0.07310	0.02761	0.04548	649.24	649.24
1	0.07294	0.02971	0.04323	649.24	649.24
4	0.07290	0.03013	0.04277	649.24	649.24
23	0.07284	0.03097	0.04187	452.46	588.69

Table 5.13: Results for $l_{spw} = 1000m$, second set of trials

For the fittest solutions the sheet pile wall is always placed as much to the right as possible. Good fitness is obtained by pumping most of it from W_2, so that is why the sheet pile wall is placed there. In less fitter solutions the sheet pile wall moves towards W_1 which allows pumping more from that well.

5.3.2 Sheet pile wall of 800 m

The results were very satisfactory since only two different fitnesses were found, the results are listed in table (5.14).

$NOO(-)$	$\phi(-)$	Q_1(m^3/s)	Q_2(m^3/s)	$s_{0,min}(m)$	$s_{0,max}(m)$
14	0.0729	0.0294	0.0435	849.2423	849.2423
29	0.0716	0.0319	0.0397	350.4809	539.2014
5	0.0716	0.0306	0.0410	444.8412	754.8820
1	0.0716	0.0281	0.0435	849.2423	849.2423
1	0.0716	0.0255	0.0461	849.2423	849.2423

Table 5.14: Results for $l_{spw} = 800m$, first set of trials

The fittest chromosome represented a sheet pile wall that started as much to the right as possible. Because the sheet pile wall was now only preventing inflow from W_2, Q_1 had dropped below the solution found in objective one. W_2 on the other hand could pump a lot without leading to sea water intrusion.

All the other trials resulted in a slightly less fit solution. 29 times a solution was found by placing a sheet pile wall somewhere on the coastline in between the two wells. Doing so, both wells can pump a little bit extra without leading to sea water intrusion, compared to objective 1.

From this first set of trials a new search area was constructed: $Q_1 \in [0.024, 0.032]$, $Q_2 \in [0.038, 0.048]$ and $s_0 \in [340, l_c - l_{spw}]$. ΔP was now decreased in order to have a finer solution domain. The new ΔP was taken to be 0.0005 m^3/s for the wells and 10 m for the sheet pile wall. To achieve this the sub chromosomes had to have a minimum of 5, 5 and 6 genes, creating a chromosome of 16. Table (5.15) lists the results for the second set of trials.

$NOO(-)$	$\phi(-)$	Q_1(m^3/s)	Q_2(m^3/s)	$s_{0,min}(m)$	$s_{0,max}(m)$
4	0.07329	0.02916	0.04413	849.24	849.24
3	0.07303	0.02890	0.04413	849.24	849.24
1	0.07258	0.02813	0.04445	849.24	849.24
1	0.07252	0.02968	0.04284	849.24	849.24
1	0.07245	0.02865	0.04381	849.24	849.24
1	0.07239	0.03019	0.04219	849.24	849.24
11	0.07232	0.03174	0.04058	437.00	461.25
28	0.07226	0.03071	0.04155	647.16	776.49

Table 5.15: Results for $l_{spw} = 800m$, second set of trials

The solutions with the highest fitness are these when a sheet pile wall is placed as much as possible to the end of the coast. 39 solutions are less fit and have the sheet pile wall placed in between the wells. Two groups of such solutions were found. The fittest ($\phi = 0.07232$) has a sheet pile wall with start point in the range of $s_0 \in [437.00, 461.25]$ m and the other solutions are ranged between $s_0 \in [647.16, 776.49]$. Both solution groups are within the range from the first set of trials, as it is supposed to be.

5.3.3 Sheet pile wall of 600 m

The results for the first set of trials is listed in table (5.16). Almost half of the time the fittest solution was found. The sheet pile wall is placed so that it is in front of the second well. As a result W_1 can not pump more than was calculated in objective 1. In fact the maximum flow pumped from this well is smaller than calculated in the first objective because of the influence of W_2 on the boundary nodes in front of W_1. The same table also shows in a very nice way what the relation between Q_1, Q_2 and s_0 is. As a general rule: the more pumped from W_2 the closer s_0 is placed towards it. This is also clear from table (5.17) that lists the second set of trials. The smaller search domain was prepared in a similar way as in the previous subsection: $Q_1 \in [0.026, 0.032]$, $Q_2 \in [0.038, 0.043]$ and $s_0 \in [260, 1040]$. Q_1, Q_2 were each represented by a sub chromosome with 4 genes and s_0 by 7 genes, in order to meet the same ΔP of 0.0005 m^3/s and 10 m. The total chromosome had a length of 15 (32768 different candidate solutions) and $P_m = P_f$ was set to be 0.07. The results in row 3 and 4 are not the same but they are different on more than 5 decimals after the comma. By rounding the values this difference became invisible.

$NOO(-)$	$\phi(-)$	Q_1(m^3/s)	Q_2(m^3/s)	$s_{0,min}(m)$	$s_{0,max}(m)$
22	0.0716	0.0306	0.0410	849.39	982.62
8	0.0703	0.0319	0.0384	266.47	682.84
4	0.0703	0.0306	0.0397	532.95	816.08
13	0.0703	0.0306	0.0397	632.88	1032.59
2	0.0690	0.0294	0.0397	749.46	816.08
1	0.0690	0.0268	0.0423	649.53	649.53

Table 5.16: Results for $l_{spw} = 600m$, first set of trials

The results from the second set of trials showed a very good convergence. 49 as fit chromosomes were found with the same flow rates. These solutions all placed the sheet pile wall in front of W_2. If the management wants W_1 to at least pump the same as in objective 1, then the engineer should return to the first set of trials and take a search area that only includes the solutions where Q_1 is bigger than calculated in objective 1.

$NOO(-)$	$\phi(-)$	Q_1(m^3/s)	Q_2(m^3/s)	$s_{0,min}(m)$	$s_{0,max}(m)$
49	0.07173	0.03040	0.04133	843.46	1027.72
1	0.07153	0.03120	0.04033	659.21	659.21

Table 5.17: Results for $l_{spw} = 600m$, second set of trials

5.3.4 Sheet pile wall of 400 m

From table (5.18) it becomes very clear in what way a genetic algorithm works. 24 very fit solutions were found, but from row 1 it is clear that it was possible to find even fitter solutions. Genetic algorithms are thus good solution finders, but they do not always return the fittest. To know the exact solution traditional calculations should then be carried out to explore the solution area around the fittest chromosomes found. Or as done here, a part of the search domain is further explored. The algorithm found as was expected protection of W_2 and lower values of Q_1. The last row lists solutions that are less fit than what was found without sheet pile wall.

$NOO(-)$	$\phi(-)$	$Q_1(\text{m}^3/\text{s})$	$Q_2(\text{m}^3/\text{s})$	$s_{0,min}(m)$	$s_{0,max}(m)$
10	0.0716	0.0306	0.0410	1050.95	1050.95
24	0.0703	0.0306	0.0397	793.17	1209.58
10	0.0703	0.0294	0.0410	1050.95	1229.41
6	0.0690	0.0294	0.0397	733.68	1150.10

Table 5.18: Results for $l_{spw} = 400m$, first set of trials

In a a second set of trials executed (ΔP as before) the trials all result in the same $\phi = 0.07140$ with $Q_1 = 0.03040$ and $Q_2 = 0.0410$. The sheet pile wall protected W_2 and $s_0 \in [1050.16, 1157.46]$. The reader might realize that the fitness has gone down. This can be explained by looking at the group of candidate solutions considered. In the second set of candidate solutions, $Q_1 = 0.0306$ was not an element. The closest was $Q_1 = 0.0304$ which results in a little less flow rate and hence a little bit less fit solution found.

5.3.5 Sheet pile wall of 200 m

In the last case, exactly in the same way as for the other lengths, the following results were calculated, listed in table (5.19). More than half of the results result in a sheet pile wall randomly generated between 57 m and 1449.24 m. Taking a closer look at the flows in the wells, the reader understands that the sheet pile is not being beneficial in these situations. It does not matter where it is placed because there will not be sea water intrusion in the first place, as was calculated in the first objective. 5 of the results lead to fitter solutions that are beneficial.

$NOO(-)$	$\phi(-)$	$Q_1(\text{m}^3/\text{s})$	$Q_2(\text{m}^3/\text{s})$	$s_{0,min}(m)$	$s_{0,max}(m)$
5	0.07032	0.03065	0.03968	1449.24	1449.24
29	0.06903	0.03065	0.03839	57.06	1449.24
13	0.06903	0.02935	0.03968	1426.42	1449.24
2	0.06774	0.02935	0.03839	992.79	1015.61
1	0.06774	0.02806	0.03968	992.79	992.79

Table 5.19: Results for $l_{spw} = 200m$, second set of trials

5.3.6 Summary

For five different sheet pile walls the best location of the sheet pile wall was calculated in order to optimize the low in both wells. Table (5.20) summarizes the results found in subsections (5.3.1) to (5.3.5).

$l_{spw}(m)$	$\phi(-)$	$Q_1(\mathrm{m}^3/\mathrm{s})$	$Q_2(\mathrm{m}^3/\mathrm{s})$	$s_{0,min}(m)$	$s_{0,max}(m)$
1000	0.07400	0.02761	0.04639	649.24	649.24
1000	0.07284	0.03097	0.04187	452.46	588.69
800	0.07329	0.02916	0.04413	849.24	849.24
800	0.07232	0.03174	0.04058	437.00	461.25
800	0.07226	0.03071	0.04155	647.16	776.49
600	0.07173	0.03040	0.04133	843.46	1027.72
600	0.07153	0.03120	0.04033	659.21	659.21
400	0.07140	0.03040	0.04100	1050.16	1157.46
200	0.07032	0.03065	0.03968	1449.24	1449.24

Table 5.20: Summary: results for $l_{spw} = 200 - 1000$ m

As was supposed to be ϕ increases with l_{spw}. Two groups of solutions were found for long sheet pile walls. The first group placed a sheet pile wall as much as possible to the right in order to protect W_2 and a second placed the sheet pile wall in between W_1 and W_2. In this first group Q_1 went well below the value calculated from the first objective, meaning that W_1 is not fully used. In the second group W_1 was protected and the flow could be higher again. When shorter sheet pile walls were used, W_2 was always protected by placing the sheet pile wall in front of it.

5.4 Sheet pile wall versus one extra well

The management can now, based upon the results from the previous section, decide to see if it is maybe not a better idea to use an extra well instead of a sheet pile wall. For example an old well W_3 might be located in zone 0 with coordinates $(1050, 750)$, and the management considers reopening it. Running the algorithm for this extra well, where Q_1, Q_2 and $Q_3 \in [0.01, 0.05]$ with $\Delta P = 0.002$ and $\lambda = 5$ for every sub chromosome lead to the results listed in table (5.21).

$NOO(-)$	$\phi(-)$	$Q_1(\mathrm{m}^3/\mathrm{s})$	$Q_2(\mathrm{m}^3/\mathrm{s})$	$Q_2(\mathrm{m}^3/\mathrm{s})$
49	0.0713	0.0281	0.0319	0.0113
1	0.0700	0.0255	0.0281	0.0165

Table 5.21: Influence of one extra well $W_3(1050, 750)$, second set of trials

Very good convergence was achieved (49/50 trials) and the total extracted flow was 0.0713 m^3/s. Comparing to the results when using a sheet pile wall (table (5.20)), it can be concluded that only in the case of a short sheet pile wall ($l_{spw} = 200$ m), the use of this extra well was found to be beneficial.

Chapter 6

Discussion and conclusions

This masters thesis combined the use of a genetic algorithm with a boundary element method with implementation of a sheet pile wall. As a result an application was developed with pre (database) and post processor (Microsoft Excel). While writing the algorithm some points of improvement became visible. Two memories were included. A first memory stored all the well positions calculated and a second all the chromosomes that were calculated. Doing so very big time and calculation reductions were achieved. In the first version a long time was spent on calculating the matrix equation $H \cdot u = G \cdot u_n$ and then in a second step sorting it to $A \cdot X = Y$ so that it could be solved by applying gauss elimination. A first improvement was not to calculate H and G but A and B directly. Next to that it was clear that big parts of A and B never changed during the generations. Therefore A and B were structured in such a way that all the elements that never changed were grouped together. They could then just be copied and a lot of calculation work was cut doing so.

The goal of doing this thesis was to find out what the influence could be of placing a sheet pile wall on an existing flow scheme pumped from two wells. In a first objective the maximum flow through the two existing wells was calculated in order not to have sea water intrusion. The results found were satisfactory: $Q_1 = 0.03129$ m^3/s and $Q_2 = 0.03829$ m^3/s and $Q_1 = 0.03135$ m^3/s and $Q_2 = 0.03823$ m^3/s. The fitness for both solutions was 0.06958, which was higher than obtained by Dr. Petala (0.069). That two chromosomes found to be exactly as fit can be explained by the discontinuous search space and the fact that both sub chromosomes (Q_1 and Q_2) had the same length and the same upper and under values were used.

The second and third objective were combined. Before running the algorithm, a set of good input parameters for the genetic algorithm was researched. Different factors were tested for the following input data: $PS = 50, NOG = 100, NOT = 10, P_c = 0.35, P_m = P_f = 0.06, \epsilon = 1$ and mutation and antimetathesis both took place in every generation. The sheet pile wall had a length of 1000 m.

A first parameter tested was the selection type used. Three selection methods were tested but using constant selection with KK = 4 showed to be better. Compared to roulette wheel selection and ranking, tournament selection had calculated a smaller amount of candidate solutions. The memory size and the required calculation time were thus smaller. A second argument to use $KK = 4$ was that the fittest solution found showed up more using this selection technique.

A small test was made where mutation and antimetathesis could take place one per chromosome or once per gene. Once per gene showed not to be sufficient to find good results. On the other hand allowing mutation and antimetathesis for every gene proved to be much better.

The influence of the population size and the number of generations was considered. Increasing the population size did not result in finding extra fit solutions. Increasing the number of generations resulted in a few more fittest solutions found. Because only few extra were found and the number of trials increased by 50, the decision was made not to increase the number of generations carried out.

The second last parameter tested was to use mutation and antimetathesis interchangingly or not. Interchanging use resulted in less fit solutions found. The memory size was also smaller which indicated that the solution area was not searched enough. When for every trial first mutation and then antimetathesis took place, the results proved to be better. There for mutation and antimetathesis was used in the last way.

The last parameter researched was called refreshment. Plotting $\phi_{max}(\gamma)$ showed that less fit solutions suffered from very long periods of not increasing their fitness. Therefore the idea was to inject new chromosomes in the population in the hope that they would lead to fitter chromosomes in the next generation. It was clear already from previous test that the algorithm sometimes needed a long time before a fitter chromosome was created. Therefore test were carried out that injected new chromosomes after a very short time of not having increased the fitness and after a longer period were the algorithm had more time to find fitter solutions. Three different injections were carried out: in a first a number of randomly populated chromosomes were added to the population size (similar to ranking). When refreshment took place soon after stabilization of ϕ, the number of fittest chromosomes found decreased. Allowing the algorithm more time before refreshing did not improve the results, but only caused more calculations to be carried out. The idea was then to refresh with highly fit chromosomes from the last generation. They would first be mutated or would first undergo antimetathesis with a probability of 100% in only one of the genes. No clear relation between the number of chromosomes refreshed and when done so could be made, but all the results were less fit compared to when no refreshment was used. Therefore the idea of refreshment was not used in the calculations that would be carried out next.

Now that the settings for the genetic algorithm were known objective 2 and 3 were studied. Using the algorithm 5 different sheet pile wall lengths were studied = 200, 400, 600 and 800 m. For long sheet pile walls two groups of solutions seemed to be calculated. A first protected W_2 by placing the sheet pile wall in front of this well. This lead to an increase of Q_2, but Q_1 was generally found to be less than was calculated in objective 2. The second group of solutions placed the sheet pile wall in between the two wells. Doing so both could extract more water from the aquifer. The first group was found to be always fitter than the last group. The decision maker will thus have to except if not fully using the capacity of W_1 is acceptable.

For shorter sheet pile walls the decision maker is not having a lot of choice because all runs point out that the sheet pile wall always protects W_2. There was a very clear relation between the length of the sheet pile wall and the total flow extracted: longer sheet pile walls lead to more extracted water without sea water intrusion. The results are listed in table (6.1).

$l_{spw}(m)$	$\phi(-)$	$Q_1(\mathrm{m}^3/\mathrm{s})$	$Q_2(\mathrm{m}^3/\mathrm{s})$	$s_{b,min}(m)$	$s_{b,max}(m)$
1000	0.07400	0.02761	0.04639	649.24	649.24
1000	0.07284	0.03097	0.04187	452.46	588.69
800	0.07329	0.02916	0.04413	849.24	849.24
800	0.07232	0.03174	0.04058	437.00	461.25
800	0.07226	0.03071	0.04155	647.16	776.49
600	0.07173	0.03040	0.04133	843.46	1027.72
600	0.07153	0.03120	0.04033	659.21	659.21
400	0.07140	0.03040	0.04100	1050.16	1157.46
200	0.07032	0.03065	0.03968	1449.24	1449.24

Table 6.1: Summary: results for $l_{spw} = 200 - 1000$ m

The algorithm was used a last time to solve an obvious question the decision maker would ask when seeing the previous results. One interesting question would be if it's not better to place an extra well. To test this a third well, $W_3 = (1050, 750)$, was added to the aquifer and the optimum solution calculated. The best results calculated were: $Q_1 = 0.0281, Q_2 = 0.0319, Q_3 = 0.0113$ m^3/s and the total flow rate was 0.07129 m^3/s. This result was only better compared to the use of a sheet pile wall of 200 m.

6.1 Reliability of the designed algorithm

In a first step the boundary element method was designed without a sheet pile wall. For this algorithm a lot of school book examples are available and the solutions obtained with the algorithm were compared with the examples from the book. The results were satisfying.

In a second step, a genetic algorithm was developed. This algorithm was first tested for simple fitness functions that did not use the boundary element method. The algorithm did as was to be expected and in a third step the boundary element method and the genetic algorithm were combined. The candidate solutions obtained from the combined use where then compared to the results obtained via the traditional solving way (calculating each candidate solution).

In a last step the use of a sheet pile wall was implemented. This made it possible to change the users input of the boundary elements based upon the chromosome calculated by the genetic algorithm.

6.2 Further research

In this thesis one fitness function was used, proposed by Katsifarakis, but different fitness functions could be developed as well. One possibility could be to include the cost and benefit of placing a sheet pile wall. Some tests have been done with a fitness function that includes these parameters as well but did not result in useful information. During those test both the length and the begin point of the sheet pile wall were a variable. The idea was to look for the best begin point and length of the sheet pile wall in combination with the best flow extracted from the two existing wells, in such a way that the sheet pile benefit was as high as possible. The results constantly led to a sheet pile wall over the entire length of the coast and maximum flow allowed in the wells or no extra flow in the wells and a sheet pile wall with length 0. In order to succeed in finding a good fitness function for this problem more information should be available about the aquifer in order to make the test realistic: How deep does the sheet pile wall need to go? How much water can be extracted from one well, how much can the aquifer provide?

It would be very interesting to further invest the influence of the parameters such as cross over, mutation, number of generations, antimetathesis, refreshing, refreshing with mutated copies of the fittest chromosome, ... The software that is developed allows the user to easily play with all these parameters and provides an excel file with the results. It would thus be an ideal start point for this research.

Very interesting as well would be to adapt the genetic algorithm so that it can calculate the best set of parameters itself. It would also be interesting to automatically do the search that was now done manually (gradually closing the search domain $(\delta q_1, \delta q_2, \delta s_b)$ and increasing ΔP).

The possibilities are in a way endless: 3D boundary element method, use of non constant boundary elements, self adapting genetic algorithms, other chromosome representations, pre-processor that allows the user to draw the boundary elements, postprocessor that output visual results, etc.

It must be mentioned as well, that the writer of this thesis is a civil engineer and not a computer engineer, the code written works and some mathematical improvement have been realized, but without any doubt there are improvements to be made in the syntax. One example is the memory the algorithm uses. It is accessed now by looping from the first to the last position in the array. Looping over 20000 positions takes a 'long' time and optimization is possible.

Appendix A

Post processor

Listed next are two worksheets of the post processor. The first sheet is called 'summary' and gives information about the input data, the results of the set of trials, some statistical information and the memory size. In a second called 'Results all trials' the best solution for every trial is given.

5 more work sheets are generated, but are not included here since they would take to many pages:

1. Detail calculations well saved
2. Detail calculations saved
3. Detail minimum fitness
4. Detail average fitness
5. Detail maximum fitness

In these worksheets, the user can follow how the memory is stored and how the fitness evolution went.

General	
Title:	Objective 3: l_spw = 1000, KK = 2
Author:	Koen Wildemeersch

Calculation Duration	
Starttime:	27 mei 2010 - 13:46:06
Endtime:	27 mei 2010 - 14:01:07
Duration:	0:15:01

Parameters genetic algorithm						
PS:	50	Selection method:	Selection constant			
NOG:	100	with constant:		2		
NOT:	10	Pc:	From:	0,35	To:	0,35
elitism:	TRUE	Pm:	From:	0,06	To:	0,06

fitness function used			
fitness function:	0		
C1:	70	C3:	0
C2:	7	C4:	0

Sheet pile wall		
Using sheet pile wall:	TRUE	
Using fixed sheet pile wall:	TRUE	
Length sheet pile wall:	1000	m
Min. bound. sheet pile wall:	0	m
Max. bound. sheet pile wall:	649,24	m
chr length start position:	5	
chr length for length spw:	-	m

Wells included

nr	Xmin	Xmax	Ymin	Ymax	Qmin	Qmax	chr_l
0	500	500	700	700	0,01	0,05	6
1	1400	1400	800	800	0,01	0,05	6

Best Result

trial Trial n° (-)	Well n° (-)	X (m)	Y (m)	Q (m3/s)
1	0	500	700	0,025873
	1	1400	800	0,04746

sb:	649,24	m
se:	1649,2	m
l:	1000	m
Max fitness:	0,0733	(-)
Tot. Inflow:	0	m3/s
NOLWI:	0	(-)
Gmax:	12	(-)
CV:	0,0016	(-)

Statistics

Lowest maximum fitness of all trials:	0,0721		
Average maximum fitness of all trials:	0,073		
Standard deviation on fitness:	0,0004		
minimum generations required to find max of trial:	63		
Calculations not carried out because of memory fitness:	18497	/	50000
memory size fitness:	31503		
Calculations not carried out because of memory well:	63004	/	63006
memory size wells:	2		

trial Trial n°	Max. Fitness	Well n°	X	Y	Q	CV	Tot. Inflow	NOLWI	Gmax	sb	se	l
(-)	(-)	(-)	(m)	(m)	(m3/s)	(-)	(m3/s)	(-)	(-)	(m)	(m)	(m)
0	0,072698413	0	500	700	0,030952	0,000951	0	0	34	586,41	1586,41	1000
		1	1400	800	0,041746							
1	0,073333333	0	500	700	0,025873	0,001585	0	0	12	649,24	1649,24	1000
		1	1400	800	0,047460							
2	0,072698413	0	500	700	0,030952	0,001585	0	0	26	565,47	1565,47	1000
		1	1400	800	0,041746							
3	0,072698413	0	500	700	0,030952	0,005682	0	0	63	586,41	1586,41	1000
		1	1400	800	0,041746							
4	0,073333333	0	500	700	0,025873	0,004110	0	0	54	649,24	1649,24	1000
		1	1400	800	0,047460							
5	0,072063492	0	500	700	0,029683	0,000951	0	0	29	502,64	1502,64	1000
		1	1400	800	0,042381							
6	0,073333333	0	500	700	0,027778	0,003795	0	0	60	649,24	1649,24	1000
		1	1400	800	0,045556							
7	0,073333333	0	500	700	0,029048	0,001585	0	0	59	649,24	1649,24	1000
		1	1400	800	0,044286							
8	0,073333333	0	500	700	0,029048	0,000951	0	0	17	649,24	1649,24	1000
		1	1400	800	0,044286							
9	0,073333333	0	500	700	0,027143	0,001585	0	0	23	649,24	1649,24	1000
		1	1400	800	0,046190							

Appendix B

Extract of source code

Included in this appendix is run.cs. This file includes all the functions that are needed for the calculation of the boundary element method and the genetic algorithm. The user interface is included in other files that have not been included to limit the size of this report.

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using System.Data.OleDb;
using System.Collections;
using System.IO;
using Excel = Microsoft.Office.Interop.Excel;

namespace KoenWildemeerschThesisWithInterface
{
    public partial class Run : Form
    {
        //variables that can be used all over the form (run.cs)

        //0. Date
        DateTime dateTimeBegin;
        DateTime dateTimeEnd;

        //1. Random
        static Random Random = new Random();

        //2. variables to be sized later (used after first setup)
        static double[][] line = new double[0][]; //after adding the SPW
        static double[] XN = new double[0]; //after adding the SPW
        static double[] YN = new double[0]; //after adding the SPW
        static int[][] zone = new int[0][]; //after adding the SPW
        static bool[] lineOnCoast = new bool[0]; //after adding the SPW
        static double[] L = new double[0]; //after adding the SPW
        static int[] K1 = new int[0]; //after adding the SPW
        static double[] BV = new double[0]; //after adding the SPW

        //3. Variables that contain the inputdata
        static int[] uK1 = new int[0]; //this array contains the type of boundary condition (0 = potential is known, 1 = flux is known)
        static double[] uBV = new double[0];
        static double[,] A = new double[0, 0];
        static double[,] Bt = new double[0, 0]; //before writing to B, write here
        static double[] B = new double[0];
        static double[] X = new double[0]; //array that holds the solutions af A.X = B
        static int[] plaatsB = new int[0];
        static int[] plaatsX = new int[0]; //array that holds all the position of the unknown
        static int[] uplaatsX = new int[0]; //for intitial
        static int[] uplaatsY = new int[0];
        double[] U = new double[0]; //array U holds the values of u after calculation
        double[] Un = new double[0]; //array Un holds the values of un after calculation
        static double[][] uline = new double[0][];
        static double[] uXN = new double[0];
        static double[] uYN = new double[0];
        static int[][] uzone = new int[0][]; // has the value of the zone(s) a nodepoint is in
        static double[][] well = new double[0][];
        static bool[][] hwell = new bool[0][];
        static int[] chrLengthWell = new int[0]; //stores the value of the chromosome length
        static double[] dmin = new double[0];
        static double[] dmax = new double[0];
        static double[] T = new double[0];
        static string[] Tname = new string[0];
        static bool[] ulineOnCoast = new bool[0];
        double[] uL = new double[0];
        int[] lineorder = new int[0];
        double[] cumulLineEnd = new double[0];
        double beginSpw = 0;
        double endSpw = 0;
        int lineBegin = 0;
        int lineEnd = 0;

        //parameters for GA
        int ps, numberofruns, numberOftrials, fitnessFunction, selectionType, selectionConstant, chr1_LengthSpw, chr2_LengthSpw, numberToRefresh, maxTimesTheSame;
        double pc_begin, pc_eind, pm_begin, pm_eind, C1, C2, C3, C4, spw_length, spw_min, spw_max;
        bool spw, elitism, fixed_spw_length, refresh, refreshByForcedMutation, refreshByForcedFlip,     ↙
    interchange;
        string projectName, author;

        //Arrays needed for the memory of the algorithm
        string[][] CalculatedChromosomes = new string[0][];
        double[][] CalculatedWellPosition = new double[0][];


        double[] CalculatedFitness = new double[0];
        double[] CalculatedWellZone = new double[0];
        double[] CalculatedTotalInflow = new double[0];
        int[] CalculatedTotalInflowNodes = new int[0];

        int CalculationsSaved = 0;
        int CalculationsSavedWell = 0;

        bool needsToBeCalculated = new bool();
        bool needsToBeCalculatedWell = new bool();

        double calculatedFitnessTemp = 0;


        public Run(int project_ID)
        {
            InitializeComponent();
            label1.Text = project_ID.ToString();

        }

        private void Run_Load(object sender, EventArgs e)
        {
            //Connect to database and fill the arrays
            //set the id
            string project_ID = label1.Text.ToString();

            //open the db
            OleDbConnection objConn = new OleDbConnection("Provider=Microsoft.JET.OLEDB.4.0;data source ↙
    =C:\\Users\\Koen Wildemeersch\\Desktop\\DataBase\\2000ThesisV11.mdb");
            objConn.Open();

            //1. fill the listview with the zones
            OleDbCommand objCommNUM = new OleDbCommand("select * from T WHERE [project_ID] = " +        ↙
    project_ID + "", objConn);
            OleDbCommand objComm = new OleDbCommand("select * from T WHERE [project_ID] = " +           ↙
    project_ID + "", objConn);

            OleDbDataReader objReaderNUM = objCommNUM.ExecuteReader();
            OleDbDataReader objReader = objComm.ExecuteReader();

            //1.a Count how many rows
            int sizeArray = 0;
            if (objReaderNUM.HasRows)
            {
                while (objReaderNUM.Read())
                {
                    sizeArray++;
                }
            }

            //1.b Resize
            Array.Resize(ref T, sizeArray);
            Array.Resize(ref Tname, sizeArray);

            //1.c Fill
            int iZone = 0;
            if (objReader.HasRows)
            {
                while (objReader.Read())
                {
                    T[iZone] = objReader.GetDouble(3);
                    Tname[iZone] = objReader.GetString(2);
                    iZone++;
                }
            }


            //2. fill the listview with the lines
            objCommNUM = new OleDbCommand("select * from lines WHERE [project_ID] = " + project_ID + "" ↙
    , objConn);
            objComm = new OleDbCommand("select * from lines WHERE [project_ID] = " + project_ID + "",   ↙
    objConn);

            objReaderNUM = objCommNUM.ExecuteReader();
            objReader = objComm.ExecuteReader();

            //2.a Count how many rows
            sizeArray = 0;
            if (objReaderNUM.HasRows)
            {
                while (objReaderNUM.Read())
                {
                    sizeArray++;
                }
            }

            //2.b Resize
            Array.Resize(ref uline, sizeArray);
            Array.Resize(ref uzone, sizeArray);
            Array.Resize(ref ulineOnCoast, sizeArray);
            Array.Resize(ref uK1, sizeArray);
            Array.Resize(ref uBV, sizeArray);

            //2.c Fill
            iZone = 0;
            if (objReader.HasRows)
            {
                while (objReader.Read())
                {
                    uline[iZone] = new double[4];
                    uzone[iZone] = new int[2];

                    uline[iZone][0] = objReader.GetDouble(2);
                    uline[iZone][1] = objReader.GetDouble(3);
                    uline[iZone][2] = objReader.GetDouble(4);
                    uline[iZone][3] = objReader.GetDouble(5);
                    uK1[iZone] = objReader.GetInt32(6);
                    uBV[iZone] = objReader.GetDouble(7);
                    uzone[iZone][0] = Array.IndexOf(Tname, objReader.GetString(8));
                    uzone[iZone][1] = Array.IndexOf(Tname, objReader.GetString(9));
                    if (uzone[iZone][0] == uzone[iZone][1])
                    {
                        uzone[iZone][1] = -1;
                    }
                    ulineOnCoast[iZone] = objReader.GetBoolean(10);
                    iZone++;
                }
            }

            //3. fill the array with the wells
            objCommNUM = new OleDbCommand("select * from wells WHERE [project_ID] = " + project_ID + "" ↙
    , objConn);
            objComm = new OleDbCommand("select * from wells WHERE [project_ID] = " + project_ID + "",   ↙
    objConn);

            objReaderNUM = objCommNUM.ExecuteReader();
            objReader = objComm.ExecuteReader();

            //3.a Count how many rows, and the dimension of dmin and dmax
            sizeArray = 0;
            int sizeD = 0;
            if (objReaderNUM.HasRows)
            {
                while (objReaderNUM.Read())
                {
                    if (objReaderNUM.GetDouble(3) != objReaderNUM.GetDouble(4))
                    {
                        sizeD++;
                    }
                    if (objReaderNUM.GetDouble(5) != objReaderNUM.GetDouble(6))
                    {
                        sizeD++;
                    }
                    if (objReaderNUM.GetDouble(7) != objReaderNUM.GetDouble(8))
                    {
                        sizeD++;
                    }
                    sizeArray++;
                }
            }

            //3.b Resize
            Array.Resize(ref well, sizeArray);
            Array.Resize(ref hwell, sizeArray);
            Array.Resize(ref chrLengthWell, sizeArray);
            Array.Resize(ref dmax, sizeD);
            Array.Resize(ref dmin, sizeD);


            //3.c Fill
            iZone = 0;
            int iDcounter = 0;

            if (objReader.HasRows)
            {
                while (objReader.Read())
                {
                    chrLengthWell[iZone] = objReader.GetInt32(9); //length of the  chromosomes for the ↙
    well

                    well[iZone] = new double[4];
                    hwell[iZone] = new bool[3];


                    if (objReader.GetDouble(3) == objReader.GetDouble(4))
                    {
                        well[iZone][0] = objReader.GetDouble(3);
                        hwell[iZone][0] = false;
                    }
                    else
                    {
                        hwell[iZone][0] = true;
                        dmin[iDcounter] = objReader.GetDouble(3);
                        dmax[iDcounter] = objReader.GetDouble(4);
                        iDcounter++;
                    }
                    if (objReader.GetDouble(5) == objReader.GetDouble(6))
                    {
                        well[iZone][1] = objReader.GetDouble(5);
                        hwell[iZone][1] = false;
                    }
                    else
                    {
                        hwell[iZone][1] = true;
                        dmin[iDcounter] = objReader.GetDouble(5);
                        dmax[iDcounter] = objReader.GetDouble(6);
                        iDcounter++;
                    }
                    if (objReader.GetDouble(7) == objReader.GetDouble(8))
                    {
                        well[iZone][2] = objReader.GetDouble(7);
                        hwell[iZone][2] = false;
                    }
                    else
                    {
                        hwell[iZone][2] = true;
                        dmin[iDcounter] = objReader.GetDouble(7);
                        dmax[iDcounter] = objReader.GetDouble(8);
                        iDcounter++;
                    }
                    iZone++;
                }//end while Read()
            }//end if there are rows

            //3.d Fill the other arrays, depending on the just resized arrays.
            Array.Resize(ref uL, uline.GetLength(0));
            Array.Resize(ref uXN, uline.GetLength(0));
            Array.Resize(ref uYN, uline.GetLength(0));



            //4. Load the GA settings

            //4.1. create the paramters
                //see begin

            //4.2. Assign the values from the db.

            objComm = new OleDbCommand("select * from GA WHERE [project_ID] = " + project_ID + "",
    objConn);
            objReader = objComm.ExecuteReader();

            if (objReader.HasRows)
            {
                while (objReader.Read())
                {
                    ps = objReader.GetInt32(2);
                    numberofruns = objReader.GetInt32(3);
                    numberOftrials = objReader.GetInt32(5);
                    pc_begin = objReader.GetDouble(6);
                    pc_eind = objReader.GetDouble(7);
                    pm_begin = objReader.GetDouble(8);
                    pm_eind = objReader.GetDouble(9);
                    elitism = objReader.GetBoolean(10);
                    spw = objReader.GetBoolean(11);
                    fitnessFunction = objReader.GetInt32(12);
                    selectionType = objReader.GetInt32(13);
                    selectionConstant = objReader.GetInt32(14);
                    C1 = objReader.GetDouble(15);
                    C2 = objReader.GetDouble(16);
                    C3 = objReader.GetDouble(17);
                    C4 = objReader.GetDouble(18);
                    fixed_spw_length = objReader.GetBoolean(19);
                    spw_length = objReader.GetDouble(20);
                    chr1_LengthSpw = objReader.GetInt32(21);
                    chr2_LengthSpw = objReader.GetInt32(22);
                    spw_min = objReader.GetDouble(23);
                    spw_max = objReader.GetDouble(24);
                    refresh = false;
                    refreshByForcedMutation = false;
                    refreshByForcedFlip = false;
                    interchange = false;
                    numberToRefresh = 10; //can be variable if successful
                    maxTimesTheSame = 10; //can be variable if successful
                }
            }//end if has rows

            //5.1. create the paramters
            //see begin

            //5.2. Assign the values from the db.

            objComm = new OleDbCommand("select * from project WHERE [ID] = " + project_ID + "",
    objConn);
            objReader = objComm.ExecuteReader();

            if (objReader.HasRows)
            {
                while (objReader.Read())
                {
                    projectName = objReader.GetString(1);
                    author = objReader.GetString(4);
                }
            }//end if has rows

            //6. Close the database
            objConn.Close();

            /*****************************************************************
             * Start the calculations
             *****************************************************************/

            //set max values for the progressbars
            progressBar1.Maximum = numberofruns;
            progressBar2.Maximum = numberOftrials;

            //calculate the begin time
            dateTimeBegin = DateTime.Now;

            int NumberOfSubchromoses = dmin.GetLength(0);
            if (spw == true)
            {
                if (fixed_spw_length == true)
                {//when a fixed length is set: only one chromosome (begin point) needs to be set
                    NumberOfSubchromoses = NumberOfSubchromoses + 1;
                }
                else
                {//length and beginpoint are variable
                    NumberOfSubchromoses = NumberOfSubchromoses + 2;
                }
                //1 extra subchromosome for the startposition, and one for the length
            }

            /*****************************************************************************************
             *
             * Calculations for the BEM (initial calculations)
             *
             *****************************************************************************************/

            //step 1: Calculate Node coordinates
            CalculateInput(uline, uL, uXN, uYN);

            //step 2: Calculate the dimensions of uplaatsX and uplaatsY
            int uNoU = totalNumberOfUnknown(uzone);
            int uNoK = 2 * uline.GetLength(0) - uNoU; //for every equation not on the interface there ↙
    is one known

            double[,] uA = new double[uNoU, uNoU];
            double[,] uBt = new double[uNoU, uNoK];


            Array.Resize(ref uplaatsX, uNoU);
            Array.Resize(ref uplaatsY, uNoK);

            //step 3: fill uplaatsX and uplaatsY
            int numberOfCoastlines = 0;
            for (int i = 0; i < ulineOnCoast.GetLength(0); i++)
            {
                if (ulineOnCoast[i] == true)
                {
                    numberOfCoastlines++;
                }
            }

            calculateUPlaatsX(ref uplaatsX, uzone, ulineOnCoast);
            calculateUPlaatsY(ref uplaatsY, uzone, ulineOnCoast);

            calculateAandBStart(ref uA, ref uBt, uplaatsX, uplaatsY, uK1, uzone, uline, uL, uXN, uYN, T ↙
    , ulineOnCoast);


            int S = numberOfCoastalElements(ulineOnCoast);

            Array.Resize(ref lineorder, S);
            Array.Resize(ref cumulLineEnd, S);

            calculateLineorderAndCumulLineEnd(uline, uL, ulineOnCoast, lineorder, cumulLineEnd);

            //assign spw_min and spw_max
            if (spw_min < 0)
            {
                spw_min = 0;
            }
            if (spw_max < 0)
            {
                if (fixed_spw_length == true)
                {
                    spw_max = cumulLineEnd[cumulLineEnd.GetLength(0) - 1] - spw_length;
                }
                else
                {
                    spw_max = cumulLineEnd[cumulLineEnd.GetLength(0) - 1];
                }
            }
            if (spw_max >= 0)
            {
                if (spw_max <= spw_min)
                {
                    MessageBox.Show("Sheet pile wall ends before it begins or has no length");
                }
            }

            /*****************************************************************************************
             *
             * Calculations for the GA
             *
             *****************************************************************************************/
            //set up the counters for the generations
            int detailnumCalculationSaved = 0;
            int detailnumCalculationSavedWell = 0;
            int TimesTheSame;

            //set up the arrays for the details of the different trials
            double[][] detailMaxFitness = new double[numberofruns][];
            double[][] detailMinFitness = new double[numberofruns][];
            double[][] detailAveFitness = new double[numberofruns][];
            int[][] detailCalculationSaved = new int[numberofruns][];
            int[][] detailCalculationSavedWell = new int[numberofruns][];

            //set the size of the jagged array

            for (int i = 0; i < numberofruns; i++)
            {
                detailMaxFitness[i] = new double[numberOftrials];
                detailMinFitness[i] = new double[numberOftrials];
                detailAveFitness[i] = new double[numberOftrials];
                detailCalculationSaved[i] = new int[numberOftrials];
                detailCalculationSavedWell[i] = new int[numberOftrials];
            }



            //set up the arrays for the differnt trials
            double[] trialMaxFitness = new double[numberOftrials];
            double[][] trialWell = new double[numberOftrials * well.GetLength(0)][];
            double[] trialConvergenceVelocity = new double[numberOftrials];
            double[] trialTotalInflow = new double[numberOftrials];
            double[] trialTotalNumberOflinesWithInflow = new double[numberOftrials];
            double[] trials = new double[numberOftrials];
            double[] triall = new double[numberOftrials];
            int[] trialBestGenFound = new int[numberOftrials];

            //set the dimension of the arrays in trialWell
            for (int w = 0; w < trialWell.GetLength(0); w++)
            {
                trialWell[w] = new double[3]; //X,Y,Q
            }

            /****************************************************************************************
            *
            * FOR EVERY TRIAL
            *
```

```
        ************************************************************************************/

        for (int trial = 0; trial < numberOftrials; trial++)
        {
            TimesTheSame = 0; //for every trial set to 0

            progressBar1.Value = progressBar1.Minimum;

            //variable that keeps track of the generation with highest fitnessfunction
            int fittestGenerationFound = 0;
            //set up the variables that are trial dependent

            double[] fitness = new double[ps];
            double elitefitness = 0;
            int numberOfElites = 1;
            string[][] elitechromosome = new string[numberOfElites][];

            for (int i = 0; i < numberOfElites; i++)
            {
                elitechromosome[i] = new string[NumberOfSubchromoses];
            }

            double[] avefitness = new double[numberofruns]; //average fitness for every run
            double[] maxfitness = new double[numberofruns]; //maximum fitness for every run
            double[] onlinefitness = new double[numberofruns]; //average of all the maxima after x 
    runs
            double[] offlinefitness = new double[numberofruns]; //average of all the maxima after x 
     runs
            double convergencevelocity = 0;

            string[][] chromosomes = new string[ps][];
            string[][] chromosomesTemp = new string[ps][];

            //assign there dimension already = amount of substrings
            for (int i = 0; i < ps; i++)
            {
                chromosomes[i] = new string[NumberOfSubchromoses];
                chromosomesTemp[i] = new string[NumberOfSubchromoses];
            }


            //create all the arrays.

            //first generate the chromosomes

            /* B. Generate the first generation of chromosomes
             * (SPW is a bool that tells if a chromosome should be created
             * for the SPW
             */

            generatepopulation(chromosomes, chrLengthWell, chr1_LengthSpw, chr2_LengthSpw, hwell, 
    spw);



            //calculate the double value of the chromosome
            for (int i = 0; i < ps; i++)
            { //thus for every population

                //check if should be calculated or not
                CheckIfNeedsToBeCalculated(ref CalculationsSaved, ref needsToBeCalculated, ref 
    calculatedFitnessTemp, chromosomes[i], CalculatedChromosomes, CalculatedFitness);

                if (needsToBeCalculated == false)
                {
                    fitness[i] = calculatedFitnessTemp;
                    detailnumCalculationSaved++;
                    detailnumCalculationSavedWell = detailnumCalculationSavedWell + well.GetLength 
    (0); //number of wells per chromosome, saved!
                }
                else
                {//it needs to be calculated
                    //fill in the variables of the well
                    int countD = 0; //counts what variable we are accessing from dmin and dmax
```

```
                        for (int w = 0; w < well.GetLength(0); w++)
                        {
                            for (int j = 0; j < 3; j++)
                            {
                                if (hwell[w][j] == true)
                                {
                                    well[w][j] = doubleChromosome(chromosomes[i][countD], dmin[countD], dmax[countD], chromosomes[i][countD].Length);
                                    countD++;//go to the next variable
                                }
                            } //end for ever the loop X, Y, Q, zone
                        }//end for every subchromosome


                        //calculate the SPW (and the changes to line, K1, BV, ...
                        if (spw == true)
                        {//if a sheetpilewall is to be included, the input data needs to be recalculated

                            //3. Calculated the beginning and the end of the SPW
                            beginSpw = 0;
                            endSpw = 0;
                            lineBegin = 0;
                            lineEnd = 0;

                            beginAndEndSPW(ref beginSpw, ref endSpw, ref lineBegin, ref lineEnd, lineorder, cumulLineEnd, chromosomes, i, fixed_spw_length, spw_length);
                            if (endSpw > cumulLineEnd[cumulLineEnd.GetLength(0) - 1])
                            {
                                MessageBox.Show("length problem");
                            }
                            //4. Calculates the number of lines that are affected
                            int Na = numberOfLinesAffected(lineorder, lineBegin, lineEnd);

                            //5. Fill an array with the affected lines
                            int[] affectedLines = new int[Na];
                            fillAffectedLines(lineorder, cumulLineEnd, Na, affectedLines, lineBegin, lineEnd);

                            //6. Calculate if extra equation because of begin of SPW
                            bool E1 = new bool();
                            E1 = extraLineForBeginSpw(cumulLineEnd, beginSpw, lineBegin, lineorder);

                            //7. Calculate if extra equation because of end of SPW
                            bool E2 = new bool();
                            E2 = extraLineForEndSpw(cumulLineEnd, endSpw, lineEnd, lineorder);

                            //8. Resize the arrays

                            int SizeArray = uline.GetLength(0);
                            if (E1 == true)
                            {
                                SizeArray++;
                            }
                            if (E2 == true)
                            {
                                SizeArray++;
                            }

                            //the exceptional case that beginSpw == endSpw (the SPW has than a lenght of 0)
                            if (beginSpw == endSpw)
                            {
                                //in this case nothing should actually happen
                                SizeArray = uline.GetLength(0);
                            }

                            //Resize arrays

                            Array.Resize(ref line, SizeArray);
                            Array.Resize(ref XN, SizeArray);
                            Array.Resize(ref YN, SizeArray);
                            Array.Resize(ref zone, SizeArray);
                            Array.Resize(ref lineOnCoast, SizeArray);
                            Array.Resize(ref L, SizeArray);
                            Array.Resize(ref K1, SizeArray);
                            Array.Resize(ref BV, SizeArray);

                            //fill array again
                            fillArrayWithValues(affectedLines, E1, E2, beginSpw, endSpw, lineorder);

                            //the number of coastal lines has changed
                            numberOfCoastlines = numberOfCoastalElements(lineOnCoast);

                        }//end if CheckBox4.checked == true
                        else
                        {//if no SPW is to be included, the valuef of uXy should be copied to Xy

                            //arrays opzetten = give them the original size again
                            int SizeArray = uline.GetLength(0);
                            Array.Resize(ref line, SizeArray);
                            Array.Resize(ref XN, SizeArray);
                            Array.Resize(ref YN, SizeArray);
                            Array.Resize(ref zone, SizeArray);
                            Array.Resize(ref lineOnCoast, SizeArray);
                            Array.Resize(ref L, SizeArray);
                            Array.Resize(ref K1, SizeArray);
                            Array.Resize(ref BV, SizeArray);

                            for (int k = 0; k < uline.GetLength(0); k++)
                            {
                                line[k] = new double[4];
                                zone[k] = new int[2];

                                for (int j = 0; j < 4; j++)
                                {
                                    Array.Copy(uline[k], j, line[k], j, 1);
                                }

                                Array.Copy(uXN, k, XN, k, 1);
                                Array.Copy(uYN, k, YN, k, 1);
                                Array.Copy(ulineOnCoast, k, lineOnCoast, k, 1);
                                Array.Copy(uL, k, L, k, 1);
                                Array.Copy(uK1, k, K1, k, 1);
                                Array.Copy(uBV, k, BV, k, 1);

                                for (int j = 0; j < 2; j++)
                                {
                                    Array.Copy(uzone[k], j, zone[k], j, 1);
                                }
                            }//end for k
                        }//else copy values when no SPW is used

                        //calculate the zonenumber of each well
                        for (int w = 0; w < well.GetLength(0); w++)
                        {
                            CheckIfNeedsToBeCalculatedWell(w, ref CalculationsSavedWell, ref 
    needsToBeCalculatedWell, ref well, CalculatedWellZone, CalculatedWellPosition);
                            if (needsToBeCalculatedWell == true)
                            {
                                findOutZoneIntellegint(ref well, w);
                                fillCalculatedWellPosition(well, w, ref CalculatedWellPosition, ref 
    CalculatedWellZone);
                            }
                            else
                            {
                                detailnumCalculationSavedWell++;
                            }
                        }

                        //this should happen for every chromosome
                        int NoU = totalNumberOfUnknown(zone);
                        int NoK = 2 * line.GetLength(0) - NoU; //for every equation not on the 
    interface there is one known

                        resizeMultiDimensionalArray(ref A, NoU, NoU);
                        resizeMultiDimensionalArray(ref Bt, NoU, NoK);
                        Array.Resize(ref B, NoU);
                        Array.Resize(ref X, NoU);
                        Array.Resize(ref uplaatsY, NoK);
                        Array.Resize(ref uplaatsX, NoU);
                        Array.Resize(ref U, line.GetLength(0));
                        Array.Resize(ref Un, line.GetLength(0));

                        bool[,] Acal = new bool[NoU, NoU];
                        bool[,] Btcal = new bool[NoU, NoK];


                        AddToUPlaatsXandY(ref uplaatsX, ref uplaatsY, zone, lineOnCoast,
    numberOfCoastlines);
                        CopyKnownValuesOfAandBt(uA, uBt, A, Bt);
                        calculateAandBt(uA, uBt, ref A, ref Bt, uplaatsX, uplaatsY, K1, zone, line, L,
    XN, YN, T, lineOnCoast,Acal, Btcal);
                        //calculateAandBdirect2(A, B, Bt, plaatsB, plaatsX, K1, BV, zone, line, L, XN,
    YN, T); //A ok, B Ok
                        calculateB(ref B, uplaatsY, Bt, BV);
                        wellinfluenceSmart(well, XN, YN, B, T, uplaatsX, zone);
                        //wellinfluence(well, XN, YN, B, T, plaatsX, zone); //needs to change as well!

                        solveInteliggent(A, B, X);
                        //reorder(BV, X, K1, U, Un, zone, plaatsX);
                        reorderSmart(BV, X, K1, U, Un, zone, uplaatsX);
                        calculatefitnessfunction(lineOnCoast, Un, fitness, i, chromosomes, dmin,
    fitnessFunction, C1, C2, C3, C4);
                        //Store chromosomes so they do not need to be recalculated
                        fillCalculatedChromosomesAndInflowCharacteristics(fitness[i], chromosomes[i],
    ref CalculatedFitness, ref CalculatedChromosomes, ref CalculatedTotalInflow, ref
    CalculatedTotalInflowNodes, Un, zone, lineOnCoast, L, T);

                    }//end if needsToBeCalculated
                }//end for every i (i = chromosome of the population)


                progressBar1.PerformStep();

                //detailed arrays
                detailMaxFitness[0][trial] = fitness.Max();
                detailMinFitness[0][trial] = fitness.Min();
                detailAveFitness[0][trial] = fitness.Average();
                detailCalculationSaved[0][trial] = detailnumCalculationSaved;
                detailCalculationSavedWell[0][trial] = detailnumCalculationSavedWell;

                //set back to 0
                detailnumCalculationSaved = 0;
                detailnumCalculationSavedWell = 0;

                //calculate average and maximum of the fitness
                avefitness[0] = fitness.Average();
                maxfitness[0] = fitness.Max();
                offlinefitness[0] = maxfitness[0];
                onlinefitness[0] = avefitness[0];

                //write the elite fitness
                elitefitness = fitness.Max();
                int IMax = Array.IndexOf(fitness, fitness.Max());

                //in any case it should be stored in the elitechromosome, it is the first run. Whatever
     chromosome will thus be the best so far
                for (int el = 0; el < numberOfElites; el++)
                {
                    for (int j = 0; j < NumberOfSubchromoses; j++)
                    {
                        elitechromosome[el][j] = String.Copy(chromosomes[IMax][j]);
                    }
                }

                //do for every generation (run =0 is the random generated chromosomes set
                for (int run = 1; run < numberofruns; run++)
                {

                    //write the population to a temp string[]

                    for (int i = 0; i < chromosomes.GetLength(0); i++)
                    {
                        for (int j = 0; j < chromosomes[i].GetLength(0); j++)
                        {
                            chromosomesTemp[i][j] = String.Copy(chromosomes[i][j]);
                        }
                    }


                    // select according the roulettewheel a chromosome
                    // Then cross them over
                    int NumberOfCrossOverCouples = (int)(Math.Floor((double)ps / 2)) * 2;

                    //Pc is constant during one run
                    double pc = Pc(run, ps, pc_begin, pc_eind);

                    //in the case of Roulettewheel selection
                    if (selectionType == 0)
                    {
                        for (int i = 0; i < NumberOfCrossOverCouples; i = i + 2)
                        {
                            int intChr1 = SelectByRoulettewheel(fitness);
                            int intChr2 = SelectByRoulettewheel(fitness);
                            for (int j = 0; j < chromosomesTemp[i].GetLength(0); j++)
                            {

                                chromosomes[i][j] = String.Copy(chromosomesTemp[intChr1][j]);
                                chromosomes[i + 1][j] = String.Copy(chromosomesTemp[intChr2][j]);
                            }
                            if (intChr1 != intChr2)
                            {//if they are the same, no new chromosome can be created by crossover.
                                crossover(chromosomes, i, pc );
                            }
                        }

                        if (ps % 2 != 0)
                        {
                            int intChr = SelectByRoulettewheel(fitness);
                            for (int j = 0; j < chromosomesTemp[ps - 1].GetLength(0); j++)
                            {
                                chromosomes[ps - 1][j] = String.Copy(chromosomesTemp[intChr][j]);
                            }
                        }
                    }//end if roulette wheel is the selectionoperator


                    if (selectionType == 1)
                    {
                        //Ranking

                        //1. How many of the population size will continue to the next generation ↙
    anyway?
                        int IntThatContinue = selectionConstant;

                        //2. Create and array that holds the fitness and the index
                        double[][] SortFitness = new double[fitness.GetLength(0)][];
                        for (int i = 0; i < fitness.GetLength(0); i++)
                        {
                            SortFitness[i] = new double[2];
                            double fit = fitness[i];
                            SortFitness[i][0] = fit;
                            SortFitness[i][1] = i;
                        }

                        //3. Sort the array, based upon its fitness...
                        IComparer myComparer = new ArrayComparer();
                        Array.Sort(SortFitness, myComparer);

                        //4. Fill the array with the chromosomes that continue anyway
                        for (int i = 0; i < IntThatContinue; i++)
                        {
                            int IndexChromosomeToCopy = (int)SortFitness[i][1];
                            for (int j = 0; j < chromosomes[0].GetLength(0); j++)
                            {
```

```
                            Array.Copy(chromosomesTemp[IndexChromosomeToCopy], j, chromosomes[i], j, 1);
                        }
                    }//end for all chromosomes that go to the next generation anyway

                    //5. Fill the other free spaces with fresh chromosomes.

                    for (int c = IntThatContinue; c < chromosomes.GetLength(0); c++)
                    {
                        int countSubChromosome = 0;
                        for (int i = 0; i < hwell.GetLength(0); i++)
                        {
                            for (int w = 0; w < 3; w++)
                            {
                                if (hwell[i][w] == true)
                                {
                                    chromosomes[c][countSubChromosome] = "";
                                    for (int j = 0; j < chrLengthWell[i]; j++)
                                    {
                                        int R = Random.Next(0, 2);
                                        chromosomes[c][countSubChromosome] = chromosomes[c][countSubChromosome] + R;
                                    }
                                    countSubChromosome++;//sub chromosome was made, so to the next one now
                                }
                            }
                        }

                        //for the sheet pile wall: chr1
                        if (spw == true)
                        {
                            if (chr1_LengthSpw != 0)
                            {
                                chromosomes[c][countSubChromosome] = "";
                                for (int j = 0; j < chr1_LengthSpw; j++)
                                {
                                    int R = Random.Next(0, 2);
                                    chromosomes[c][countSubChromosome] = chromosomes[c][countSubChromosome] + R;
                                }
                                countSubChromosome++;
                            }

                            if (chr2_LengthSpw != 0)
                            {
                                chromosomes[c][countSubChromosome] = "";
                                for (int j = 0; j < chr2_LengthSpw; j++)
                                {
                                    int R = Random.Next(0, 2);
                                    chromosomes[c][countSubChromosome] = chromosomes[c][countSubChromosome] + R;
                                }
                                countSubChromosome++;
                            }
                        }//end if spw == true
                    }//end for c

                    //6. Crossing over
                    for (int i = 0; i < NumberOfCrossOverCouples; i = i + 2)
                    {
                        crossover(chromosomes, i, pc);
                    }

                    //if uneven the last chromosome will not be crossed over.

                }//end Ranking

                if (selectionType == 2)
                {
                    int KK = selectionConstant();

                    for (int i = 0; i < NumberOfCrossOverCouples; i = i + 2)
                    {
                        int intChr1 = SelectByConstantSelection(fitness, KK);
                        int intChr2 = SelectByConstantSelection(fitness, KK);
                        for (int j = 0; j < chromosomesTemp[i].GetLength(0); j++)
                        {

                            chromosomes[i][j] = String.Copy(chromosomesTemp[intChr1][j]);
                            chromosomes[i + 1][j] = String.Copy(chromosomesTemp[intChr2][j]);
                        }

                        if (intChr1 != intChr2)
                        {//if they are the same, crossover cannot create a new chromosome
                            crossover(chromosomes, i, pc);
                        }
                    }

                    if (ps % 2 != 0)
                    {
                        int intChr = SelectByConstantSelection(fitness, KK);
                        for (int j = 0; j < chromosomesTemp[ps - 1].GetLength(0); j++)
                        {
                            chromosomes[ps - 1][j] = String.Copy(chromosomesTemp[intChr][j]);
                        }
                    }

                }

                if (interchange == true)
                {
                    //now mutate them
                    if (run % 2 == 0)
                    {
                        double pm = Pm(run, ps, pm_begin, pm_eind);
                        for (int i = 0; i < ps; i++)
                        {
                            mutation(chromosomes, i, pm);
                        }
                    }
                    else
                    {
                        //and now flip them
                        double pf = Pm(run, ps, pm_begin, pm_eind);
                        for (int i = 0; i < ps; i++)
                        {
                            flip(chromosomes, i, pf);
                        }
                    }
                }
                else
                {
                    //mutate
                    double pm = Pm(run, ps, pm_begin, pm_eind);
                    for (int i = 0; i < ps; i++)
                    {
                        mutation(chromosomes, i, pm);
                    }

                    //flip
                    double pf = Pm(run, ps, pm_begin, pm_eind);
                    for (int i = 0; i < ps; i++)
                    {
                        flip(chromosomes, i, pf);
                    }
                }

                //add the best one again!
                if (elitism == true)
                {
                    double maximumValue = fitness.Max();
                    int whereIsMaximum = Array.LastIndexOf(fitness, maximumValue);
                    for (int i = 0; i < chromosomes[0].GetLength(0); i++)
                    {
                        chromosomes[0][i] = String.Copy(chromosomesTemp[whereIsMaximum][i]);
                    }
                }

                if (refreshByForcedFlip == true && (selectionType == 0 || selectionType == 2))
                {
                    /* This function forces the best solution of the previous run to mutate,
                     * the place where mutation takes place is selected with equal probability)
                     */

                    if (TimesTheSame >= maxTimesTheSame)
                    {
                        // maximum value of last run
                        double maximumValue = fitness.Max();
                        int whereIsMaximum = Array.LastIndexOf(fitness, maximumValue);

                        for (int c = ps - numberToRefresh; c < ps; c++)
                        {
                            // Select subchromosome that will be mutate by chance
                            int R1 = Random.Next(0, chromosomes[0].GetLength(0));
                            // The length of the subchromosome
                            int length = chromosomes[0][R1].Length;
                            // the gene that will be mutated
                            int R2 = Random.Next(0, length-1);


                            //taking the sub chromosome that was selected
                            string subChrTemp = String.Copy(chromosomesTemp[whereIsMaximum][R1]);
                            //split in parts
                            string subChrB = subChrTemp.Substring(0, R2); //begin
                            string subChrM1 = subChrTemp.Substring(R2, 1); //to be flipped
                            string subChrM2 = subChrTemp.Substring(R2+1, 1); //to be flipped
                            string subChrE = subChrTemp.Substring(R2+2, (length - R2 - 2)); //end

                            //past back together
                            subChrTemp = subChrB + subChrM2 + subChrM1 + subChrE;

                            //store
                            for (int i = 0; i < chromosomes[0].GetLength(0); i++)
                            {
                                if (i != R1)
                                {
                                    chromosomes[c][i] = String.Copy(chromosomesTemp[whereIsMaximum][i]);
                                }
                                else
                                {
                                    chromosomes[c][i] = String.Copy(subChrTemp);
                                }
                            }
                        }//end for c
                    }//end if should be refreshed
                }//end refresh

                if (refreshByForcedMutation == true && (selectionType == 0 || selectionType == 2))
                {
                    /* This function forces the best solution of the previous run to mutate,
                     * the place where mutation takes place is selected with equal probability)
                     */

                    if (TimesTheSame >= maxTimesTheSame)
                    {
                        // maximum value of last run
                        double maximumValue = fitness.Max();
                        int whereIsMaximum = Array.LastIndexOf(fitness, maximumValue);

                        for (int c = ps - numberToRefresh; c < ps; c++)
                        {
                            // Select subchromosome that will be mutate by chance
                            int R1 = Random.Next(0, chromosomes[0].GetLength(0));
                            // The length of the subchromosome
                            int length = chromosomes[0][R1].Length;
                            // the gene that will be mutated
                            int R2 = Random.Next(0, length);

                            //taking the sub chromosome that was selected
                            string subChrTemp = String.Copy(chromosomesTemp[whereIsMaximum][R1]);
                            //split in parts
                            string subChrB = subChrTemp.Substring(0, R2); //begin
                            string subChrM = subChrTemp.Substring(R2, 1); //to be mutated
                            string subChrE = subChrTemp.Substring(R2 + 1, (length - R2 - 1)); //end
                            //mutate
                            if (subChrM == "1")
                            {
                                subChrM = "0";
                            }
                            else
                            {
                                subChrM = "1";
                            }
                            //past back together
                            subChrTemp = subChrB + subChrM + subChrE;

                            //store
                            for (int i = 0; i < chromosomes[0].GetLength(0); i++)
                            {
                                if (i != R1)
                                {
                                    chromosomes[c][i] = String.Copy(chromosomesTemp[whereIsMaximum][i]);
                                }
                                else
                                {
                                    chromosomes[c][i] = String.Copy(subChrTemp);
                                }
                            }
                        }
                    }//end if should be refreshed
                }//end refresh

                if (refresh == true && (selectionType == 0 || selectionType == 2))
                {
                        if (TimesTheSame >= maxTimesTheSame)
                        {
                            for (int c = ps - numberToRefresh; c < ps; c++)
                            {
                                int countSubChromosome = 0;
                                for (int i = 0; i < hwell.GetLength(0); i++)
                                {
                                    for (int w = 0; w < 3; w++)
                                    {
                                        if (hwell[i][w] == true)
                                        {
                                            chromosomes[c][countSubChromosome] = "";
                                            for (int j = 0; j < chrLengthWell[i]; j++)
                                            {
                                                int R = Random.Next(0, 2);
                                                chromosomes[c][countSubChromosome] = chromosomes[c][countSubChromosome] + R;
                                            }
                                            countSubChromosome++;//sub chromosome was made, so to the next one now
                                        }
                                    }
                                }

                                //for the sheet pile wall: chr1
                                if (spw == true)
                                {
                                    if (chr1_LengthSpw != 0)
                                    {
                                        chromosomes[c][countSubChromosome] = "";
                                        for (int j = 0; j < chr1_LengthSpw; j++)
                                        {
                                            int R = Random.Next(0, 2);
                                            chromosomes[c][countSubChromosome] = chromosomes[c][countSubChromosome] + R;
                                        }
                                        countSubChromosome++;
                                    }

                                    if (chr2_LengthSpw != 0)
                                    {
                                        chromosomes[c][countSubChromosome] = "";
                                        for (int j = 0; j < chr2_LengthSpw; j++)
                                        {
                                            int R = Random.Next(0, 2);
                                            chromosomes[c][countSubChromosome] = chromosomes[c][countSubChromosome] + R;
                                        }
                                        countSubChromosome++;
                                    }
                                }//end if spw == true
                            }
                        }//end if should be refreshed
                }//end refresh

                //calculate the new values of the unknown again

                for (int i = 0; i < ps; i++)
                {
                    //check if should be calculated or not
                    CheckIfNeedsToBeCalculated(ref CalculationsSaved, ref needsToBeCalculated, ref calculatedFitnessTemp, chromosomes[i], CalculatedChromosomes, CalculatedFitness);

                    if (needsToBeCalculated == false)
                    {
                        fitness[i] = calculatedFitnessTemp;
                        detailnumCalculationSaved++;
                        detailnumCalculationSavedWell = detailnumCalculationSavedWell + well.GetLength(0); //number of wells per chromosome, saved!
                    }
                    else
                    {//it needs to be calculated
                        int countD = 0; //counts what variable we are accessing from dmin and dmax
                        for (int w = 0; w < well.GetLength(0); w++)
                        {
                            for (int j = 0; j < 3; j++)
                            {
                                if (hwell[w][j] == true)
                                {
                                    well[w][j] = doubleChromosome(chromosomes[i][countD], dmin[countD], dmax[countD], chromosomes[i][countD].Length);
                                    countD++;//go to the next variable
                                }
                            } //end for ever the loop X, Y, Q, zone
                        }//end for every subchromosome



                        //calculate the SPW (and the changes to line, K1, BV, ...
                        if (spw == true)
                        {//if a sheetpilewall is to be included, the input data needs to be recalculated

                            //3. Calculated the beginning and the end of the SPW
                            beginSpw = 0;
                            endSpw = 0;
                            lineBegin = 0;
                            lineEnd = 0;

                            beginAndEndSPW(ref beginSpw, ref endSpw, ref lineBegin, ref lineEnd, lineorder, cumulLineEnd, chromosomes, i, fixed_spw_length, spw_length);
                            if (endSpw > cumulLineEnd[cumulLineEnd.GetLength(0) - 1])
                            {
                                MessageBox.Show("length problem");
                            }
                            //4. Calculates the number of lines that are affected
                            int Na = numberOfLinesAffected(lineorder, lineBegin, lineEnd);

                            //5. Fill an array with the affected lines
                            int[] affectedLines = new int[Na];
                            fillAffectedLines(lineorder, cumulLineEnd, Na, affectedLines, lineBegin, lineEnd);

```

```
                                //6. Calculate if extra equation because of begin of SPW
                                bool E1 = new bool();
                                E1 = extraLineForBeginSpw(cumulLineEnd, beginSpw, lineBegin, lineorder) ↙
    ;

                                //7. Calculate if extra equation because of end of SPW
                                bool E2 = new bool();
                                E2 = extraLineForEndSpw(cumulLineEnd, endSpw, lineEnd, lineorder);

                                //8. Resize the arrays

                                int SizeArray = uline.GetLength(0);
                                if (E1 == true)
                                {
                                    SizeArray++;
                                }
                                if (E2 == true)
                                {
                                    SizeArray++;
                                }

                                //the exceptional case that beginSpw == endSpw
                                if (beginSpw == endSpw)
                                {
                                    //in this case nothing should actually happen
                                    SizeArray = uline.GetLength(0);
                                }

                                //Resize arrays

                                Array.Resize(ref line, SizeArray);
                                Array.Resize(ref XN, SizeArray);
                                Array.Resize(ref YN, SizeArray);
                                Array.Resize(ref zone, SizeArray);
                                Array.Resize(ref lineOnCoast, SizeArray);
                                Array.Resize(ref L, SizeArray);
                                Array.Resize(ref K1, SizeArray);
                                Array.Resize(ref BV, SizeArray);

                                //fill array again
                                fillArrayWithValues(affectedLines, E1, E2, beginSpw, endSpw, lineorder) ↙
    ;

                                //the number of coastal lines has changed
                                numberOfCoastlines = numberOfCoastalElements(lineOnCoast);
                            }//end if CheckBox4.checked == true
                            else
                            {//if no SPW is to be included, the valuef of uXy should be copied to Xy

                                for (int k = 0; k < uline.GetLength(0); k++)
                                {
                                    line[k] = new double[4];
                                    zone[k] = new int[2];

                                    for (int j = 0; j < 4; j++)
                                    {
                                        Array.Copy(uline[k], j, line[k], j, 1);
                                    }

                                    Array.Copy(uXN, k, XN, k, 1);
                                    Array.Copy(uYN, k, YN, k, 1);
                                    Array.Copy(ulineOnCoast, k, lineOnCoast, k, 1);
                                    Array.Copy(uL, k, L, k, 1);
                                    Array.Copy(uK1, k, K1, k, 1);
                                    Array.Copy(uBV, k, BV, k, 1);

                                    for (int j = 0; j < 2; j++)
                                    {
                                        Array.Copy(uzone[k], j, zone[k], j, 1);
                                    }
                                }//end for k
                            }//else copy values when no SPW is used
```

```
                        //calculate the zonenumber of each well
                        for (int w = 0; w < well.GetLength(0); w++)
                        {
                            CheckIfNeedsToBeCalculatedWell(w, ref CalculationsSavedWell, ref needsToBeCalculatedWell, ref well, CalculatedWellZone, CalculatedWellPosition);
                            if (needsToBeCalculatedWell == true)
                            {
                                findOutZoneIntellegint(ref well, w);
                                fillCalculatedWellPosition(well, w, ref CalculatedWellPosition, ref CalculatedWellZone);
                            }
                            else
                            {
                                detailnumCalculationSavedWell++;
                            }
                        }


                        //this should happen for every chromosome
                        int NoU = totalNumberOfUnknown(zone);
                        int NoK = 2 * line.GetLength(0) - NoU; //for every equation not on the interface there is one known


                        //A and B matrix (square matrix, with dimension of G and H = dimension XM)
                        resizeMultiDimensionalArray(ref A, NoU, NoU);
                        resizeMultiDimensionalArray(ref Bt, NoU, NoK);
                        Array.Resize(ref B, NoU);
                        Array.Resize(ref X, NoU);
                        Array.Resize(ref uplaatsY, NoK);
                        Array.Resize(ref uplaatsX, NoU);
                        Array.Resize(ref U, line.GetLength(0));
                        Array.Resize(ref Un, line.GetLength(0));

                        bool[,] Acal = new bool[NoU, NoU];
                        bool[,] Btcal = new bool[NoU, NoK];

                        //calculatePlaatsB(plaatsB, zone);
                        //calculatePlaatsX(plaatsX, zone);
                        AddToUPlaatsXandY(ref uplaatsX, ref uplaatsY, zone, lineOnCoast, numberOfCoastlines);
                        CopyKnownValuesOfAandBt(uA, uBt, A, Bt);
                        calculateAandBt(uA, uBt, ref A, ref Bt, uplaatsX, uplaatsY, K1, zone, line, L, XN, YN, T, lineOnCoast, Acal, Btcal);
                        //calculateAandBdirect2(A, B, Bt, plaatsB, plaatsX, K1, BV, zone, line, L, XN, YN, T); //A ok, B Ok
                        calculateB(ref B, uplaatsY, Bt, BV);
                        wellinfluenceSmart(well, XN, YN, B, T, uplaatsX, zone);
                        //wellinfluence(well, XN, YN, B, T, plaatsX, zone); //needs to change as well!
                        solveInteliggent(A, B, X);
                        //reorder(BV, X, K1, U, Un, zone, plaatsX);
                        reorderSmart(BV, X, K1, U, Un, zone, uplaatsX);
                        calculatefitnessfunction(lineOnCoast, Un, fitness, i, chromosomes, dmin, fitnessFunction, C1, C2, C3, C4);
                        //Store chromosomes so they do not need to be recalculated
                        fillCalculatedChromosomesAndInflowCharacteristics(fitness[i], chromosomes[i], ref CalculatedFitness, ref CalculatedChromosomes, ref CalculatedTotalInflow, ref CalculatedTotalInflowNodes, Un, zone, lineOnCoast, L, T);
                    }//end if needs to be recalculated
                }




                //store details
                detailMaxFitness[run][trial] = fitness.Max();
                detailMinFitness[run][trial] = fitness.Min();
                detailAveFitness[run][trial] = fitness.Average();
```

```
                    detailCalculationSaved[run][trial] = detailnumCalculationSaved;
                    detailCalculationSavedWell[run][trial] = detailnumCalculationSavedWell;

                    //check if the fitness found is higher
                    if (detailMaxFitness[run][trial] == detailMaxFitness[run-1][trial])
                    {
                        TimesTheSame++;
                    }
                    else
                    {
                        TimesTheSame = 0;
                    }

                    //reset detailnumCalculationSaved and detailnumCalculationSavedWell
                    detailnumCalculationSaved = 0;
                    detailnumCalculationSavedWell = 0;

                    //calculate maximum and average fitness of this generation
                    avefitness[run] = fitness.Average();
                    maxfitness[run] = fitness.Max();
                    if (maxfitness[run] < maxfitness[run - 1])
                    {
                        MessageBox.Show("Maxima werd niet overgenomen!");
                    }
                    else if (maxfitness[run] > maxfitness[run - 1])
                    {
                        elitefitness = fitness.Max();
                        IMax = Array.IndexOf(fitness, fitness.Max());

                        for (int el = 0; el < numberOfElites; el++)
                        {
                            for (int j = 0; j < NumberOfSubchromoses; j++)
                            {
                                Array.Copy(chromosomes[IMax], j, elitechromosome[el], j, 1);
                            }
                        }

                        //in this generation the best was found
                        fittestGenerationFound = run;
                    }


                    //calculate f_off and f_on
                    calculateOfflinePerformance(offlinefitness, run, maxfitness);
                    calculateOnlinePerformance(onlinefitness, run, avefitness);

                    //print offlinefitness and onlinefitness
                    //printOfflinePerformance(offlinefitness);
                    //printOnlinePerformance(onlinefitness);
                    //printavefitness(avefitness);
                    //printmaxfitness(maxfitness);
                    progressBar1.PerformStep();
                }//end run


                convergencevelocity = calculateConvergenceVelocity(maxfitness);
                double startOfSheetpilewall = 0;
                double lengthOfSheetpilewall = 0;
                double[] dWhereIsMax = new double[dmin.GetLength(0)]; //to store the double values

                //calculate the place where the maximum fitness occured
                if (elitism == false)
                {
                    int IndexOfMaximum = Array.IndexOf(fitness, fitness.Max());

                    for (int d = 0; d < dmin.GetLength(0); d++)
                    {
                        dWhereIsMax[d] = doubleChromosome(chromosomes[IndexOfMaximum][d], dmin[d], dmax ↙
    [d], chromosomes[IndexOfMaximum][d].Length);
                    }
                    if (fixed_spw_length == true)
                    {
                        startOfSheetpilewall = doubleChromosome(chromosomes[0][chromosomes[0].GetLength ↙
    (0) - 1], spw_min, spw_max, chromosomes[0][chromosomes[0].GetLength(0) - 1].Length);
```

```
                    lengthOfSheetpilewall = spw_length;
                }
                else
                {
                    startOfSheetpilewall = doubleChromosome(chromosomes[0][chromosomes[0].GetLength(0) - 2], spw_min, spw_max, chromosomes[0][chromosomes[0].GetLength(0) - 2].Length);
                    lengthOfSheetpilewall = (doubleChromosome(chromosomes[0][chromosomes[0].GetLength(0) - 1], 0, 1, chromosomes[0][chromosomes[0].GetLength(0) - 1].Length)) * (spw_max - startOfSheetpilewall);
                }

            }//end if checkbox3 was not checked.
            else
            { //the checkbox was checked
                for (int d = 0; d < dmin.GetLength(0); d++)
                {
                    dWhereIsMax[d] = doubleChromosome(elitechromosome[0][d], dmin[d], dmax[d], elitechromosome[0][d].Length);
                }

                if (spw == true)
                {
                    if (fixed_spw_length == true)
                    {
                        startOfSheetpilewall = doubleChromosome(elitechromosome[0][elitechromosome[0].GetLength(0) - 1], spw_min, spw_max, elitechromosome[0][elitechromosome[0].GetLength(0) - 1].Length);
                        lengthOfSheetpilewall = spw_length;
                    }
                    else
                    {
                        startOfSheetpilewall = doubleChromosome(elitechromosome[0][elitechromosome[0].GetLength(0) - 2], spw_min, spw_max, elitechromosome[0][elitechromosome[0].GetLength(0) - 2].Length);
                        lengthOfSheetpilewall = (doubleChromosome(elitechromosome[0][elitechromosome[0].GetLength(0) - 1], 0, 1, elitechromosome[0][elitechromosome[0].GetLength(0) - 1].Length)) * (spw_max - startOfSheetpilewall);
                    }
                }
            }//end else: the checkbox was checked

            //fill the trial for the inflow caracteristics

            /* for the maximum fitness of the last run and the identical chromosomes copy
             * find the index in the CalculatedTotalInflow and store the values in
             * trialTotalInflow and trialTotalNumberOflinesWithInflow
             */

            for (int j = 0; j < CalculatedFitness.GetLength(0); j++)
            {//j is the counter representing the CalculatedFitness

                if (fitness.Max() == CalculatedFitness[j])
                {
                    //multiple chromosomes might have the same fitness so it should be checked if their subchromosomes are identical
                    int numOk = 0;
                    for (int s = 0; s < CalculatedChromosomes[j].GetLength(0); s++)
                    {
                        if (chromosomes[Array.IndexOf(fitness, fitness.Max())][s] == CalculatedChromosomes[j][s])
                        {
                            numOk++;
                        }
                    }//end for s

                    if (numOk == CalculatedChromosomes[j].GetLength(0))
                    {//this is the index that we are looking for
                        trialTotalInflow[trial] = CalculatedTotalInflow[j];
                        trialTotalNumberOflinesWithInflow[trial] = CalculatedTotalInflowNodes[j];
                        j = CalculatedFitness.GetLength(0);
                    }
                }//end if (fitness[i] == Calculatedfitness[j])
            }//end for each chromosome in the store matrices

            //fill the trial arrays.
            trialMaxFitness[trial] = fitness.Max();
            trialConvergenceVelocity[trial] = convergencevelocity;
            trials[trial] = startOfSheetpilewall;
            triall[trial] = lengthOfSheetpilewall;
            trialBestGenFound[trial] = fittestGenerationFound;

            int dd = 0;

            for (int i = 0; i < well.GetLength(0); i++)
            {
                for (int j = 0; j < 3; j++)
                {
                    if (hwell[i][j] == false)
                    {
                        trialWell[trial * well.GetLength(0) + i][j] = well[i][j];
                    }
                    else
                    {
                        trialWell[trial * well.GetLength(0) + i][j] = dWhereIsMax[dd];
                        dd++;
                    }
                }
            } //end filling well
            progressBar2.PerformStep();
        }//end of all trial

        //write the report showint the results and the best found
        trialreportxls(ps, numberofruns, pc_begin, pc_eind, pm_begin, pm_eind, trialMaxFitness,     ↙
trialWell, trialConvergenceVelocity, trialTotalInflow, trialTotalNumberOflinesWithInflow,            ↙
trialBestGenFound, trials, triall, CalculationsSaved, NumberOfSubchromoses, CalculationsSavedWell,  ↙
CalculatedFitness.GetLength(0), CalculatedWellZone.GetLength(0),detailMaxFitness, detailMinFitness, ↙
 detailAveFitness, detailCalculationSaved, detailCalculationSavedWell, C1, C2, C3, C4,              ↙
fixed_spw_length, spw_length);
        MessageBox.Show("Trials completed");

    }//end Run_Load

    //Other functions

    public void CalculateInput(double[][] uline, double[] uL, double[] uXN, double[] uYN)
    {
        /* line[i][0] = x coordinate of the left endpoint of line i
         * line[i][1] = y coordinate of the left endpoint of line i
         * line[i][2] = x coordinate of the right endpoint of line i
         * line[i][3] = y coordinate of the right endpoint of line i
         */

        for (int i = 0; i < uline.GetLength(0); i++)
        {
            uL[i] = Math.Sqrt(Math.Pow((uline[i][2] - uline[i][0]), 2) + Math.Pow((uline[i][3] -    ↙
uline[i][1]), 2));
            uXN[i] = (uline[i][0] + uline[i][2]) / 2;
            uYN[i] = (uline[i][1] + uline[i][3]) / 2;
        }
    }//end CalculateInput

    public void calculateUPlaatsX(ref int[] plaatsuX, int[][] uzone, bool[] ulineOnCoast)
    {
        int i = 0;

        //for all the nodes not on the interface
        for (int I = 0; I < uzone.GetLength(0); I++)
        {
            if (ulineOnCoast[I] == false)
            {
                uplaatsX[i] = I; //nodes have to be numbers from one to N, and always increased by  ↙
1.
                i++;
                if (uzone[I][1] != -1)
                {
                    uplaatsX[i] = I;
                    i++;
                }
            }
        }
        //second write all the nodes that are on the coastline
        for (int I = 0; I < uzone.GetLength(0); I++)
        {
            if (ulineOnCoast[I] == true)
            {
                uplaatsX[i] = I;
                i++;
            }
        }
    }//end calculateUPlaatsX

    public void calculateUPlaatsY(ref int[] plaatsuY, int[][] uzone, bool[] ulineOnCoast)
    {
        int i = 0;

        //for all the nodes not on the coastline and interface
        for (int I = 0; I < uzone.GetLength(0); I++)
        {
            if (ulineOnCoast[I] == false)
            {
                if (uzone[I][1] == -1)
                {
                    uplaatsY[i] = I;
                    i++;
                }
            }
        }
        //second write all the nodes that are on the coastline
        for (int I = 0; I < uzone.GetLength(0); I++)
        {
            if (ulineOnCoast[I] == true)
            {
                uplaatsY[i] = I;
                i++;
            }
        }
    }//end calculateUPlaatsY

    public void calculateAandBStart(ref double[,] uA, ref double[,] uBt, int[] uplaatsX, int[] uplaatsY, int[] uK, int[][] uzone, double[][] uline, double[] uL, double[] uXN, double[] uYN, double[] T, bool[] ulineOnCoast)
    {
        for (int I = 0; I < uzone.GetLength(0); I++)
        {
            int rij = Array.IndexOf(uplaatsX, I);

            //write first equation: for node on interface or not, it is the same
            for (int J = 0; J < uzone.GetLength(0); J++)
            {
                if (uzone[J][0] == uzone[I][0] || uzone[J][1] == uzone[I][0])
                {
                    //when J is on the interface
                    if (uzone[J][1] != -1)
                    {
                        //is J defined in same zone as I (otherwise problem with L and g*(-To/T1)
                        if (uzone[J][0] == uzone[I][0])
                        { //they are defined in the same zone: no problem
                            if (I == J)
                            {
                                uA[rij, Array.IndexOf(uplaatsX, J)] = -0.5; // = h
                                uA[rij, Array.LastIndexOf(uplaatsX, J)] = -uL[J] / (2 * Math.PI) * (Math.Log(uL[J] / 2) - 1); // =-g
                            }
                            else
                            {
                                uA[rij, Array.IndexOf(uplaatsX, J)] = Hon(uXN[I], uline[J][0], uline[J][2], uYN[I], uline[J][1], uline[J][3]); // = h
                                uA[rij, Array.LastIndexOf(uplaatsX, J)] = -Gon(uXN[I], uline[J][0], uline[J][2], uYN[I], uline[J][1], uline[J][3], uL[J]); // =-g
                            }
                        }
                        else
```

```
                            { //they are not defined in the same zone: pay attention!
                                if (I == J)
                                {
                                    uA[rij, Array.IndexOf(uplaatsX, J)] = -0.5;// =h
                                    uA[rij, Array.LastIndexOf(uplaatsX, J)] = -uL[J] / (2 * Math.PI) * (Math.Log(uL[J] / 2) - 1) * (-T[uzone[J][0]] / T[uzone[J][1]]);//-g
                                }
                                else
                                {
                                    uA[rij, Array.IndexOf(uplaatsX, J)] = Hon(uXN[I], uline[J][2], uline[J][0], uYN[I], uline[J][3], uline[J][1]);// =h
                                    uA[rij, Array.LastIndexOf(uplaatsX, J)] = -Gon(uXN[I], uline[J][2], uline[J][0], uYN[I], uline[J][3], uline[J][1], uL[J]) * (-T[uzone[J][0]] / T[uzone[J][1]]); //-g

                                }
                            }

                        }

                        //when J is not on the interface
                        else
                        {
                            //there can be no problem with L or g*(-To/T1), K1 decides

                            if (uK1[J] == 0) //u is given so colums should be changed
                            {
                                if (I == J)
                                {
                                    uA[rij, Array.IndexOf(uplaatsX, J)] = -uL[J] / (2 * Math.PI) * (Math.Log(uL[J] / 2) - 1); //-g
                                    uBt[rij, Array.IndexOf(uplaatsY, J)] = 0.5; //-h
                                }
                                else
                                {
                                    uA[rij, Array.IndexOf(uplaatsX, J)] = -Gon(uXN[I], uline[J][0], uline[J][2], uYN[I], uline[J][1], uline[J][3], uL[J]); //-g
                                    uBt[rij, Array.IndexOf(uplaatsY, J)] = -Hon(uXN[I], uline[J][0], uline[J][2], uYN[I], uline[J][1], uline[J][3]); //-h
                                }
                            }
                            else //no problem, colums can stay. (uK1[J] == 1)
                            {
                                if (I == J)
                                {
                                    uA[rij, Array.IndexOf(uplaatsX, J)] = -0.5; //h
                                    uBt[rij, Array.IndexOf(uplaatsY, J)] = uL[J] / (2 * Math.PI) * (Math.Log(uL[J] / 2) - 1); //g
                                }
                                else
                                {
                                    uA[rij, Array.IndexOf(uplaatsX, J)] = Hon(uXN[I], uline[J][0], uline[J][2], uYN[I], uline[J][1], uline[J][3]); //h
                                    uBt[rij, Array.IndexOf(uplaatsY, J)] = Gon(uXN[I], uline[J][0], uline[J][2], uYN[I], uline[J][1], uline[J][3], uL[J]); //g
                                }
                            }
                        }
                    }
                }//end for all J


                //write second equation: only for nodes on the interface
                if (uzone[I][1] != -1)
                {
                    rij = Array.LastIndexOf(uplaatsX, I);

                    //write second equation: only for nodes I on the interface
                    for (int J = 0; J < uzone.GetLength(0); J++)
                    {
                        //check if an equation should be written towards this point
                        if (uzone[J][0] == uzone[I0][1] || uzone[J][1] == uzone[I][1])
                        {


```

```
                        //when J is on the interface
                        if (uzone[J][1] != -1)
                        {
                            //is J defined in same zone as I (otherwise problem with L and g*(-To/T1)
                            if (uzone[J][0] == uzone[I][1])
                            { //they are defined in the same zone: no problem

                                if (I == J)
                                {
                                    uA[rij, Array.IndexOf(uplaatsX, J)] = -0.5; // = h
                                    uA[rij, Array.LastIndexOf(uplaatsX, J)] = -uL[J] / (2 * Math.PI) * (Math.Log(uL[J] / 2) - 1); // =-g, voorlopig geen teken wissel
                                }
                                else
                                {
                                    uA[rij, Array.IndexOf(uplaatsX, J)] = Hon(uXN[I], uline[J][0], uline[J][2], uYN[I], uline[J][1], uline[J][3]); // = h
                                    uA[rij, Array.LastIndexOf(uplaatsX, J)] = -Gon(uXN[I], uline[J][0], uline[J][2], uYN[I], uline[J][1], uline[J][3], uL[J]); // =-g
                                }

                            }
                            else
                            { //they are not defined in the same zone: pay attention!

                                if (I == J)
                                {
                                    uA[rij, Array.IndexOf(uplaatsX, J)] = -0.5;// =h
                                    uA[rij, Array.LastIndexOf(uplaatsX, J)] = -uL[J] / (2 * Math.PI) * (Math.Log(uL[J] / 2) - 1) * (-T[uzone[J][0]] / T[uzone[J][1]]); //-g
                                }
                                else
                                {
                                    uA[rij, Array.IndexOf(uplaatsX, J)] = Hon(uXN[I], uline[J][2], uline[J][0], uYN[I], uline[J][3], uline[J][1]);// =h
                                    uA[rij, Array.LastIndexOf(uplaatsX, J)] = -Gon(uXN[I], uline[J][2], uline[J][0], uYN[I], uline[J][3], uline[J][1], uL[J]) * (-T[uzone[J][0]] / T[uzone[J][1]]); // -g
                                }
                            }

                        }

                        //when J is not on the interface
                        else
                        {
                            //there can be no problem with L or g*(-To/T1), K1 decides

                            if (uK1[J] == 0) //u is given so colums should be changed
                            {
                                if (I == J)
                                {
                                    uA[rij, Array.IndexOf(uplaatsX, J)] = -uL[J] / (2 * Math.PI) * (Math.Log(uL[J] / 2) - 1); //-g
                                    uBt[rij, System.Array.IndexOf(uplaatsY, J)] = 0.5; //-h
                                }
                                else
                                {
                                    uA[rij, Array.IndexOf(uplaatsX, J)] = -Gon(uXN[I], uline[J][0], uline[J][2], uYN[I], uline[J][1], uline[J][3], uL[J]); //-g
                                    uBt[rij, Array.IndexOf(uplaatsY, J)] = -Hon(uXN[I], uline[J][0], uline[J][2], uYN[I], uline[J][1], uline[J][3]); //-h
                                }
                            }
                            else //no problem, colums can stay.
                            {
                                if (I == J)
                                {
                                    uA[rij, Array.IndexOf(uplaatsX, J)] = -0.5; //h
                                    uBt[rij, Array.IndexOf(uplaatsY, J)] = uL[J] / (2 * Math.PI) * (Math.Log(uL[J] / 2) - 1); //g
                                }
                                else
                                {
                                    uA[rij, Array.IndexOf(uplaatsX, J)] = Hon(uXN[I], uline[J][0], uline[J][2], uYN[I], uline[J][1], uline[J][3]); //h
                                    uBt[rij, Array.IndexOf(uplaatsY, J)] = Gon(uXN[I], uline[J][0], uline[J][2], uYN[I], uline[J][1], uline[J][3], uL[J]); //g
                                }
                            }

                        }
                    }//end if equation should be written
                }
            }
        }//end for all nodes I

    }//end calculateAandBtStart

    public void calculateLineorderAndCumulLineEnd(double[][] uline, double[] uL, bool[] ulineOnCoast, int[] lineorder, double[] cumulLineEnd)
    {
        //1. Temp store all elements on the coastline
        int[] onCoast = new int[lineorder.GetLength(0)];
        int counter = 0;
        for (int i = 0; i < uline.GetLength(0); i++)
        {
            if (ulineOnCoast[i] == true)
            {
                onCoast[counter] = i; //write away the number of the line that is on the coast
                counter++;
            }
        }

        //2. Sort them from beginning to end
        int[][] numberOfTimesUsed = new int[onCoast.GetLength(0)][];

        for (int l = 0; l < onCoast.GetLength(0); l++)
        {
            //for all lines on the coastline check how many times there left and right node is used
            numberOfTimesUsed[l] = new int[2];

            for (int j = 0; j < onCoast.GetLength(0); j++)
            {
                //left node of the line
                if ((uline[onCoast[j]][0] == uline[onCoast[l]][0] && uline[onCoast[j]][1] == uline[onCoast[l]][1]) || (uline[onCoast[j]][2] == uline[onCoast[l]][0] && uline[onCoast[j]][3] == uline[onCoast[l]][1]))
                {
                    numberOfTimesUsed[l][0]++;
                }

                //right node of the line
                if ((uline[onCoast[j]][0] == uline[onCoast[l]][2] && uline[onCoast[j]][1] == uline[onCoast[l]][3]) || (uline[onCoast[j]][2] == uline[onCoast[l]][2] && uline[onCoast[j]][3] == uline[onCoast[l]][3]))
                {
                    numberOfTimesUsed[l][1]++;
                }
            }
        }

        //find out where the line starts and ends
        int LineStart = 0;
        int LineEnd = 0;

        for (int i = 0; i < numberOfTimesUsed.GetLength(0); i++)
        {
            if (numberOfTimesUsed[i][0] == 1)
            {
                if (LineStart != 0)
                {
                    MessageBox.Show("Multiple possibilities for line beginning");
                }
                else
                {
                    LineStart = onCoast[i];
                }
            }

            if (numberOfTimesUsed[i][1] == 1)
            {
                if (LineEnd != 0)
                {
                    MessageBox.Show("Multiple possibilities for line ending");
                }
                else
                {
                    LineEnd = onCoast[i];
                }
            }
        }

        //find the lineorder and store away in array int lineorder[]
        lineorder[0] = LineStart;
        cumulLineEnd[0] = uL[LineStart];

        //A. Calculate lineorder

        for (int t = 1; t < onCoast.GetLength(0); t++)
        {
            for (int l = 0; l < onCoast.GetLength(0); l++)
            {
                //find where the end of the line t is the same of the beginning of line l
                if (uline[lineorder[t - 1]][2] == uline[onCoast[l]][0] && uline[lineorder[t - 1]]
[3] == uline[onCoast[l]][1])
                {
                    lineorder[t] = onCoast[l];
                    cumulLineEnd[t] = cumulLineEnd[t - 1] + uL[onCoast[l]];
                    l = lineorder.GetLength(0);
                }
            }
        }


        //last point! This should be exactly the end point because otherwise a mistake was made
        if (uline[lineorder[lineorder.GetLength(0) - 2]][2] == uline[LineEnd][0] && uline[lineorder
[lineorder.GetLength(0) - 2]][3] == uline[LineEnd][1])
        {
            lineorder[lineorder.GetLength(0) - 1] = LineEnd;
            cumulLineEnd[lineorder.GetLength(0) - 1] = cumulLineEnd[lineorder.GetLength(0) - 2] +
uL[LineEnd];
        }
        else
        {
            MessageBox.Show("coastline is not calculated correctly!");
        }

    }//end calculateLineorderAndCumulLineEnd

    public void generatepopulation(string[][] chromosomes, int[] chrLengthWell, int chr1_LengthSpw,
 int chr2_LengthSpw, bool[][] hwell, bool spw)
    {
        for (int ps = 0; ps < chromosomes.GetLength(0); ps++)
        {
            int countSubChromosome = 0;
            for (int i = 0; i < hwell.GetLength(0); i++)
            {
                for (int w = 0; w < 3; w++)
                {
                    if (hwell[i][w] == true)
                    {
                        chromosomes[ps][countSubChromosome] = "";
                        for (int j = 0; j < chrLengthWell[i]; j++)
                        {
                            int R = Random.Next(0, 2);
                            chromosomes[ps][countSubChromosome] = chromosomes[ps]
[countSubChromosome] + R;
                        }
                        countSubChromosome++;//sub chromosome was made, so to the next one now
                    }
                }
            }

            //for the sheet pile wall: chr1
            if (spw == true)
            {
                if (chr1_LengthSpw != 0)
                {
                    chromosomes[ps][countSubChromosome] = "";
                    for (int j = 0; j < chr1_LengthSpw; j++)
                    {
                        int R = Random.Next(0, 2);
                        chromosomes[ps][countSubChromosome] = chromosomes[ps][countSubChromosome] + R;
                    }
                    countSubChromosome++;
                }

                if (chr2_LengthSpw != 0)
                {
                    chromosomes[ps][countSubChromosome] = "";
                    for (int j = 0; j < chr2_LengthSpw; j++)
                    {
                        int R = Random.Next(0, 2);
                        chromosomes[ps][countSubChromosome] = chromosomes[ps][countSubChromosome] + R;
                    }
                    countSubChromosome++;
                }
            }


        }//end for every ps
    } //end generatepopulation

    public void CheckIfNeedsToBeCalculated(ref int numberOfValuesSaved, ref bool needsToBeCalculated, ref double calculatedFitnessTemp, string[] chromosome, string[][] Calculatedchromosomes, double[] Calculatedfitness)
    {
        int numberOfSubChromosomes = chromosome.GetLength(0);
        needsToBeCalculated = true; //a test will be performed to see if calculation is required

        //see if the fitnessvalue is already in the Calculatedfitness matrix
        for (int j = 0; j < Calculatedfitness.GetLength(0); j++)
        {//j is the counter representing the CalculatedFitness

            //multiple chromosomes might have the same fitness so it should be checked if their subchromosomes are identical
            int numOk = 0;
            for (int s = 0; s < numberOfSubChromosomes; s++)
            {
                if (chromosome[s] == Calculatedchromosomes[j][s])
                {
                    numOk++;
                }
                else
                {
                    s = numberOfSubChromosomes; //if one is not in it, that it can not be the same any way
                }
            }//end for s

            if (numOk == numberOfSubChromosomes)
            {//then there is no need to recalculate
                needsToBeCalculated = false;
                calculatedFitnessTemp = Calculatedfitness[j];
                j = Calculatedfitness.GetLength(0); //so the for loop ends
                numberOfValuesSaved++; //this chromosome does not need to be recalculated
            }

        }//end for each chromosome in the stored matrices

    }//end CheckIfNeedsToBeCalculated

    public void beginAndEndSPW(ref double beginSpw, ref double endSpw, ref int lineBegin, ref int lineEnd, int[] lineorder, double[] cumulLineEnd, string[][] chromosomes, int r, bool fixed_spw_length, double spw_length)
    {
        double lengthSpw = 0;
        if (fixed_spw_length == true)
        {
            //beginSpw from 0 to l_coast - l_spw
            beginSpw = doubleChromosome(chromosomes[r][chromosomes[r].GetLength(0) - 1], spw_min, spw_max, chromosomes[r][chromosomes[r].GetLength(0) - 1].Length);
            lengthSpw = spw_length;
        }
        else
        {
            beginSpw = doubleChromosome(chromosomes[r][chromosomes[r].GetLength(0) - 2], spw_min, spw_max, chromosomes[r][chromosomes[r].GetLength(0) - 2].Length);
            //length is procentualy calculated from distance beginning to distance end
            lengthSpw = doubleChromosome(chromosomes[r][chromosomes[r].GetLength(0) - 1], 0, 1, chromosomes[r][chromosomes[r].GetLength(0) - 1].Length) * (spw_max - beginSpw);
        }

        endSpw = beginSpw + lengthSpw;

        //calculate on what line the SPW begins
        bool foundLine = false;
        for (int i = 0; i < lineorder.GetLength(0); i++)
        {

            if (beginSpw < cumulLineEnd[i])
            {
                lineBegin = lineorder[i];
                i = lineorder.GetLength(0);
                foundLine = true;
            }
        }
        if (foundLine == false)
        {
            lineBegin = lineorder[lineorder.GetLength(0) - 1];
            //MessageBox.Show("beginLine is not smaller than end of the sheetpilewall");
        }

        //calculate on what line the SPW endss
        for (int i = 0; i < lineorder.GetLength(0); i++)
        {
            if (endSpw <= cumulLineEnd[i])
            {
                lineEnd = lineorder[i];
                i = lineorder.GetLength(0);
            }
        }
        if (beginSpw > endSpw)
        {
            MessageBox.Show("Something seriously went wrong calculating the begin and end coordinates of the spw!");
        }
    } //end beginAndEndSpw

    public void fillAffectedLines(int[] lineorder, double[] cumulLineEnd, int numberOflinesAffected , int[] affectedLines, int lineBegin, int lineEnd)
    {
        int counter = 0;
        int t = Array.IndexOf(lineorder, lineBegin);
        bool onSWP = new bool();
        onSWP = true;
        while (onSWP == true)
        {
            if (lineorder[t] == lineEnd)
            {

                affectedLines[counter] = lineorder[t];
                counter++;
                onSWP = false;
            }
            else
            {

                affectedLines[counter] = lineorder[t];
                counter++;
            }
            t++; //go to next line
        }
    }//end fillAffectedLines

    public void fillArrayWithValues(int[] affectedLines, bool E1, bool E2, double beginSpw, double endSpw, int[] lineorder)
    {

        //1. for all lines that are not affected, just copy
        for (int i = 0; i < uline.GetLength(0); i++)
        {

            line[i] = new double[4];
            zone[i] = new int[2];

            //step 1: copy all the information that is not affected

            if (Array.IndexOf(affectedLines, i) == -1)
            {//for all lines that are not affected
                //a) line
                for (int j = 0; j < 4; j++)
                {
                    Array.Copy(uline[i], j, line[i], j, 1);
                }

                //b) uXN, uYN, uLineOnCoast,uL,uK1,uBV

                Array.Copy(uXN, i, XN, i, 1);
                Array.Copy(uYN, i, YN, i, 1);
                Array.Copy(ulineOnCoast, i, lineOnCoast, i, 1);
                Array.Copy(uL, i, L, i, 1);
                Array.Copy(uK1, i, K1, i, 1);
                Array.Copy(uBV, i, BV, i, 1);

                //c) uzone
                for (int j = 0; j < 2; j++)
                {
                    Array.Copy(uzone[i], j, zone[i], j, 1);
                }
            }//if they are not affected
            else
            {//when the line is affected (at least part of it is on the SPW)

                //Calculate Sbegin and Send
                double Send = cumulLineEnd[Array.IndexOf(lineorder, i)]; ;
                double Sbegin = Send - uL[i];

                //Possibility 1: lineBegin == lineEnd
                if (affectedLines.GetLength(0) == 1)
                {
                    if (beginSpw != endSpw)
                    {
                        //a) Sbegin == beginSpw && Send == endSpw
                        if (Sbegin == beginSpw && Send == endSpw)
                        {
                            //intire line changes to become SPW, no new line is created
                            //a) line
                            for (int j = 0; j < 4; j++)
                            {
                                Array.Copy(uline[i], j, line[i], j, 1);
                            }

                            //b) uXN, uYN, uLineOnCoast,uL,uK1,uBV

                            Array.Copy(uXN, i, XN, i, 1);
                            Array.Copy(uYN, i, YN, i, 1);
                            Array.Copy(ulineOnCoast, i, lineOnCoast, i, 1);
                            Array.Copy(uL, i, L, i, 1);
                            //BV and K should not be copied but set manualy
                            K1[i] = 1;
                            BV[i] = 0;

                            //c) uzone
                            for (int j = 0; j < 2; j++)
                            {
                                Array.Copy(uzone[i], j, zone[i], j, 1);
                            }
                        }//end if Sbegin == beginSpw && Send == endSpw
                        else
                        {
                            int row = line.GetLength(0) - 1;

                            //first have a look at the end
                            if (E2 == true)
                            {
                                //1. calculate begin of the line
                                double Dx = Dsx(uline, uL, cumulLineEnd, i, endSpw, lineorder);
                                double Dy = Dsy(uline, uL, cumulLineEnd, i, endSpw, lineorder);
                                double Xs = uline[i][0] + Dx;
                                double Ys = uline[i][1] + Dy;
                                double Length = Math.Sqrt(Math.Pow(Dx, 2) + Math.Pow(Dy, 2));

                                //ALFA) Write Existing part that is on the SPW (begin original line to S)
                                //a) line

                                Array.Copy(uline[i], 0, line[i], 0, 1);
                                Array.Copy(uline[i], 1, line[i], 1, 1);
                                line[i][2] = Xs;
                                line[i][3] = Ys;

                                //b) uXN, uYN, uLineOnCoast,uL,uK1,uBV

                                XN[i] = (line[i][2] + line[i][0]) / 2;
                                YN[i] = (line[i][3] + line[i][1]) / 2;
                                Array.Copy(ulineOnCoast, i, lineOnCoast, i, 1);
                                L[i] = Length;
                                K1[i] = 1;
                                BV[i] = 0;

                                //c) uzone
                                for (int j = 0; j < 2; j++)
                                {
                                    Array.Copy(uzone[i], j, zone[i], j, 1);
                                }

                                //BETA) Write the NEW part that is not part of the SPW (S to end orginal of line) // the extra line!
                                //a) line
                                line[row] = new double[4];
                                line[row][0] = Xs;
                                line[row][1] = Ys;
                                Array.Copy(uline[i], 2, line[row], 2, 1);
                                Array.Copy(uline[i], 3, line[row], 3, 1);


                                //b) uXN, uYN, uLineOnCoast,uL,uK1,uBV
                                XN[row] = (line[row][2] + line[row][0]) / 2;
                                YN[row] = (line[row][3] + line[row][1]) / 2;

                                Array.Copy(ulineOnCoast, i, lineOnCoast, row, 1);
                                L[row] = uL[i] - Length;
                                Array.Copy(uK1, i, K1, row, 1);
                                Array.Copy(uBV, i, BV, row, 1);

                                //c) uzone
                                zone[row] = new int[2];
                                for (int j = 0; j < 2; j++)
                                {
                                    Array.Copy(uzone[i], j, zone[row], j, 1);
                                }
                                row--; //only if an extra line was added!
                            }//end if E2 == true
                            else
                            {
                                //the line is SPW until the end of the line (the beginning is regarded later)
                                Array.Copy(uline[i], 0, line[i], 0, 1);
                                Array.Copy(uline[i], 1, line[i], 1, 1);
                                Array.Copy(uline[i], 2, line[i], 2, 1);
                                Array.Copy(uline[i], 3, line[i], 3, 1);
                                Array.Copy(uXN, i, XN, i, 1);
                                Array.Copy(uYN, i, YN, i, 1);
                                Array.Copy(uL, i, L, i, 1);
                                Array.Copy(ulineOnCoast, i, lineOnCoast, i, 1);
                                K1[i] = 1;//became SPW
                                BV[i] = 0;//became SPW

                                for (int j = 0; j < 2; j++)
                                {
                                    Array.Copy(uzone[i], j, zone[i], j, 1);
                                }


                            }// if E2 != true (just copy but change BV, K1)
                            if (E1 == true)
                            {
                                //1. calculate begin of the new line
                                double Dx = Dsx(uline, uL, cumulLineEnd, i, beginSpw, lineorder);
                                double Dy = Dsy(uline, uL, cumulLineEnd, i, beginSpw, lineorder);
                                double Xs = uline[i][0] + Dx;
                                double Ys = uline[i][1] + Dy;
                                double Length = Math.Sqrt(Math.Pow(Dx, 2) + Math.Pow(Dy, 2));

                                //ALFA) Write the NEW part that is not on the SPW (begin original line to S) // the extra line!
                                //a) line
                                line[row] = new double[4];
                                Array.Copy(uline[i], 0, line[row], 0, 1);
                                Array.Copy(uline[i], 1, line[row], 1, 1);
                                line[row][2] = Xs;
                                line[row][3] = Ys;

                                //b) uXN, uYN, uLineOnCoast,uL,uK1,uBV

                                XN[row] = (line[row][2] + line[row][0]) / 2;
                                YN[row] = (line[row][3] + line[row][1]) / 2;
                                Array.Copy(ulineOnCoast, i, lineOnCoast, row, 1);
                                L[row] = Length;
                                Array.Copy(uK1, i, K1, row, 1);
                                Array.Copy(uBV, i, BV, row, 1);

                                //c) uzone
                                zone[row] = new int[2];
                                for (int j = 0; j < 2; j++)
                                {
                                    Array.Copy(uzone[i], j, zone[row], j, 1);
                                }

                                //BETA) Change begin coordinates and length of uXY[i] (S to end end of the already adapted line uL[i])
                                //a) line

                                line[i][0] = Xs;
                                line[i][1] = Ys;
                                //x en y coordinate of the end of line i are already set

                                //b) uXN, uYN, uLineOnCoast,uL,uK1,uBV
                                XN[i] = (line[i][2] + line[i][0]) / 2;
                                YN[i] = (line[i][3] + line[i][1]) / 2;

                                L[i] = Math.Sqrt(Math.Pow(line[i][2] - line[i][0], 2) + Math.Pow(line[i][3] - line[i][1], 2));
                                //uK1 and uBV had already been set
                            }//end if E1 == true
                            else
                            {
                                //nothing needs to change anymore, because it already happend in ↙
    the if or else condition for E2==true
                            }
                        }//end if Sbegin != beginSpw || Send != endSpw
                    }//end beginSpw =! endSpw (when are the same nothing should happen)
                    else
                    {//when beginSpw == endSpw ==> copy the data
                        //a) line
                        for (int j = 0; j < 4; j++)
                        {
                            Array.Copy(uline[i], j, line[i], j, 1);
                        }

                        //b) uXN, uYN, uLineOnCoast,uL,uK1,uBV

                        Array.Copy(uXN, i, XN, i, 1);
                        Array.Copy(uYN, i, YN, i, 1);
                        Array.Copy(ulineOnCoast, i, lineOnCoast, i, 1);
                        Array.Copy(uL, i, L, i, 1);
                        Array.Copy(uK1, i, K1, i, 1);
                        Array.Copy(uBV, i, BV, i, 1);

                        //c) uzone
                        for (int j = 0; j < 2; j++)
                        {
                            Array.Copy(uzone[i], j, zone[i], j, 1);
                        }
                    }
                }//end if beginline == lineEnd


                //Possibility 2: lineBegin != lineEnd
                else
                {
                    if (i == lineBegin)
                    {//is begin SPW
                        if (beginSpw == Sbegin)
                        {
                            /* The entire line is SPW
                             * copy most, but change K1 and BV
                             * no extra line needs to be calculated
                             */

                            //a) line
                            for (int j = 0; j < 4; j++)
                            {
                                Array.Copy(uline[i], j, line[i], j, 1);
                            }

                            //b) uXN, uYN, uLineOnCoast,uL,uK1,uBV

                            Array.Copy(uXN, i, XN, i, 1);
                            Array.Copy(uYN, i, YN, i, 1);
                            Array.Copy(ulineOnCoast, i, lineOnCoast, i, 1);
                            Array.Copy(uL, i, L, i, 1);
                            //BV and K should not be copied but set manualy
                            K1[i] = 1;
                            BV[i] = 0;

                            //c) uzone
                            for (int j = 0; j < 2; j++)
                            {
                                Array.Copy(uzone[i], j, zone[i], j, 1);
                            }
                        }//end if (beginSpw == Sbegin)
                        else
                        {
                            int row = line.GetLength(0) - 1;
                            //calculate on what row the extra line should be stored
                            if (E2 == true)
                            {
                                row--;
                            }

```

```
                                //a new line is to be added, and the existing to be changed
                                //1. calculate begin of the new line
                                double Dx = Dsx(uline, uL, cumulLineEnd, i, beginSpw, lineorder);
                                double Dy = Dsy(uline, uL, cumulLineEnd, i, beginSpw, lineorder);
                                double Xs = uline[i][0] + Dx;
                                double Ys = uline[i][1] + Dy;
                                double Length = Math.Sqrt(Math.Pow(Dx, 2) + Math.Pow(Dy, 2));

                                //ALFA) Write the NEW part that is not on the SPW (begin original line to S) // the extra line!
                                //a) line
                                line[row] = new double[4];
                                Array.Copy(uline[i], 0, line[row], 0, 1);
                                Array.Copy(uline[i], 1, line[row], 1, 1);
                                line[row][2] = Xs;
                                line[row][3] = Ys;

                                //b) uXN, uYN, uLineOnCoast,uL,uK1,uBV

                                XN[row] = (line[row][2] + line[row][0]) / 2;
                                YN[row] = (line[row][3] + line[row][1]) / 2;
                                Array.Copy(ulineOnCoast, i, lineOnCoast, row, 1);
                                L[row] = Length;
                                Array.Copy(uK1, i, K1, row, 1);
                                Array.Copy(uBV, i, BV, row, 1);

                                //c) uzone
                                zone[row] = new int[2];
                                for (int j = 0; j < 2; j++)
                                {
                                    Array.Copy(uzone[i], j, zone[row], j, 1);
                                }

                                //BETA) Change begin coordinates and length of uXY[i] (S to end)
                                //a) line

                                line[i][0] = Xs;
                                line[i][1] = Ys;
                                Array.Copy(uline[i], 2, line[i], 2, 1);
                                Array.Copy(uline[i], 3, line[i], 3, 1);
                                Array.Copy(ulineOnCoast, i, lineOnCoast, i, 1);
                                //b) uXN, uYN, uLineOnCoast,uL,uK1,uBV
                                XN[i] = (line[i][2] + line[i][0]) / 2;
                                YN[i] = (line[i][3] + line[i][1]) / 2;

                                L[i] = Math.Sqrt(Math.Pow(line[i][2] - line[i][0], 2) + Math.Pow(line[i][3] - line[i][1], 2));
                                K1[i] = 1;
                                BV[i] = 0;

                                for (int j = 0; j < 2; j++)
                                {
                                    Array.Copy(uzone[i], j, zone[i], j, 1);
                                }

                            }//end if (beginSpw != Sbegin)
                        }//end if i == lineBegin
                        else if (i == lineEnd)
                        {//is end SPW
                            if (endSpw == Send)
                            {
                                /* The entire line is SPW
                                 * copy most, but change K1 and BV
                                 * no extra line needs to be calculated
                                 */

                                //a) line
                                for (int j = 0; j < 4; j++)
                                {
                                    Array.Copy(uline[i], j, line[i], j, 1);
                                }

                                //b) uXN, uYN, uLineOnCoast,uL,uK1,uBV

```

```
        Array.Copy(uXN, i, XN, i, 1);
        Array.Copy(uYN, i, YN, i, 1);
        Array.Copy(ulineOnCoast, i, lineOnCoast, i, 1);
        Array.Copy(uL, i, L, i, 1);
        //BV and K should not be copied but set manualy
        K1[i] = 1;
        BV[i] = 0;

        //c) uzone
        for (int j = 0; j < 2; j++)
        {
            Array.Copy(uzone[i], j, zone[i], j, 1);
        }



    }//end if (endSpw == Send)
    else
    {
        int row = line.GetLength(0) - 1;

        //a new line is to be added, and the existing to be changed
        //1. calculate begin of the new line
        double Dx = Dsx(uline, uL, cumulLineEnd, i, endSpw, lineorder);
        double Dy = Dsy(uline, uL, cumulLineEnd, i, endSpw, lineorder);
        double Xs = uline[i][0] + Dx;
        double Ys = uline[i][1] + Dy;
        double Length = Math.Sqrt(Math.Pow(Dx, 2) + Math.Pow(Dy, 2));

        //ALFA) Write the NEW part that is not on the SPW (S to end line) // ↙
the extra line!
        //a) line
        line[row] = new double[4];
        Array.Copy(uline[i], 2, line[row], 2, 1);
        Array.Copy(uline[i], 3, line[row], 3, 1);
        line[row][0] = Xs;
        line[row][1] = Ys;

        //b) uXN, uYN, uLineOnCoast,uL,uK1,uBV

        XN[row] = (line[row][2] + line[row][0]) / 2;
        YN[row] = (line[row][3] + line[row][1]) / 2;
        Array.Copy(ulineOnCoast, i, lineOnCoast, row, 1);
        L[row] = uL[i] - Length;
        Array.Copy(uK1, i, K1, row, 1);
        Array.Copy(uBV, i, BV, row, 1);

        //c) uzone
        zone[row] = new int[2];
        for (int j = 0; j < 2; j++)
        {
            Array.Copy(uzone[i], j, zone[row], j, 1);
        }

        //BETA) The existing line is now shortened and is SPW
        //a) line

        line[i][2] = Xs;
        line[i][3] = Ys;
        Array.Copy(uline[i], 0, line[i], 0, 1);
        Array.Copy(uline[i], 1, line[i], 1, 1);

        //b) uXN, uYN, uLineOnCoast,uL,uK1,uBV
        XN[i] = (line[i][2] + line[i][0]) / 2;
        YN[i] = (line[i][3] + line[i][1]) / 2;
        Array.Copy(ulineOnCoast, i, lineOnCoast, i, 1);
        L[i] = Length;
        K1[i] = 1;
        BV[i] = 0;

        for (int j = 0; j < 2; j++)
        {
            Array.Copy(uzone[i], j, zone[i], j, 1);
        }
```

```
                        }// end if (endSpw != Send)
                    }//end if (i == lineEnd)
                    else
                    {//line is not holding end or begin but is just SPW

                        //a) line
                        for (int j = 0; j < 4; j++)
                        {
                            Array.Copy(uline[i], j, line[i], j, 1);
                        }

                        //b) uXN, uYN, uLineOnCoast,uL,uK1,uBV

                        Array.Copy(uXN, i, XN, i, 1);
                        Array.Copy(uYN, i, YN, i, 1);
                        Array.Copy(ulineOnCoast, i, lineOnCoast, i, 1);
                        Array.Copy(uL, i, L, i, 1);
                        //BV and K should not be copied but set manualy
                        K1[i] = 1;
                        BV[i] = 0;

                        //c) uzone
                        for (int j = 0; j < 2; j++)
                        {
                            Array.Copy(uzone[i], j, zone[i], j, 1);
                        }

                    }//end if line is not holding end or begin but is just SPW
                }//end if beginline != lineEnd


            }//end for all lines that are affected
        }//end for every line i loop
    }//end fillArrayWithUnchangedValues

    public void CheckIfNeedsToBeCalculatedWell(int w, ref int numberOfValuesSavedWell, ref bool ↙
needsToBeCalculatedWell, ref double[][] well, double[] CalculatedWellZone, double[][]           ↙
CalculatedWellPosition)
    {
        needsToBeCalculatedWell = true; //a test will be performed to see if calculation is     ↙
required

        //see if the fitnessvalue is already in the Calculatedfitness matrix
        for (int j = 0; j < CalculatedWellPosition.GetLength(0); j++)
        {//j is the counter representing the CalculatedFitness
            if (well[w][0] == CalculatedWellPosition[j][0])
            {
                if (well[w][1] == CalculatedWellPosition[j][1])
                {
                    needsToBeCalculatedWell = false; //no need to recalculate
                    Array.Copy(CalculatedWellZone, j, well[w], 3, 1);//assign the value
                    j = CalculatedWellPosition.GetLength(0); //stop the search
                    numberOfValuesSavedWell++; //calculation saved
                }
            }
        }//end for each chromosome in the store matrice
    }//end CheckIfNeedsToBeCalculatedWell

    public void findOutZoneIntellegint(ref double[][] bron, int w)
    {

        /*******************************************************************
         * function valid for wells that are on the interface or in any of the subdomains
         * when well is on the boundary an error will occur!
         * Situations like this will never occur because the conditions on the boundary
         * are fixed! a well should thus never be positionated there!
         */
        //for each well, the zonenumber will be stored here
        int[][] zoneNumber = new int[bron.GetLength(0)][];

        zoneNumber[w] = new int[2];

        //variables needed for this function
        double[][] linesWithSameXunder = new double[0][]; //first position is for the number of the ↙
```

```
 line
            double[][] linesWithSameXabove = new double[0][]; //second position is for the distance between the well and the line
            double[][] linesWithSameYleft = new double[0][];
            double[][] linesWithSameYright = new double[0][];

            double YXw = 0;
            double XYw = 0;
            double m = 0; //rico of the line

            //variable necessary to check if on interface or boundary!
            bool found = new bool();
            found = false;


            //check all the lines in the project
            for (int l = 0; l < line.GetLength(0); l++)
            {
                //check the X-coordinates
                if ((bron[w][0] >= line[l][0] && bron[w][0] <= line[l][2]) || (bron[w][0] <= line[l][0] && bron[w][0] >= line[l][2]))
                {
                    //1. calculate Y(Xw) (X is known, Y is unknown)
                    if (line[l][0] == line[l][2])
                    { //m would be give devide by 0 error
                        YXw = YN[l];
                    }
                    else
                    {
                        m = (line[l][3] - line[l][1]) / (line[l][2] - line[l][0]);
                        YXw = m * (bron[w][0] - line[l][0]) + line[l][1];
                    }

                    //2. Fill in the array linesWith...
                    if (YXw == bron[w][1])
                    {
                        //increase size by one
                        Array.Resize(ref linesWithSameXabove, linesWithSameXabove.GetLength(0) + 1);
                        Array.Resize(ref linesWithSameXunder, linesWithSameXunder.GetLength(0) + 1);

                        //create new element
                        linesWithSameXabove[linesWithSameXabove.GetLength(0) - 1] = new double[2];
                        linesWithSameXunder[linesWithSameXunder.GetLength(0) - 1] = new double[2];

                        //insert values
                        linesWithSameXabove[linesWithSameXabove.GetLength(0) - 1][0] = l;
                        linesWithSameXabove[linesWithSameXabove.GetLength(0) - 1][1] = 0;
                        linesWithSameXunder[linesWithSameXunder.GetLength(0) - 1][0] = l;
                        linesWithSameXunder[linesWithSameXunder.GetLength(0) - 1][1] = 0;
                    }

                    if (YXw > bron[w][1])
                    { //above it
                        Array.Resize(ref linesWithSameXabove, linesWithSameXabove.GetLength(0) + 1);
                        linesWithSameXabove[linesWithSameXabove.GetLength(0) - 1] = new double[2]; //first position for its zone, and second for its X coordinate, later on used to calculate the closest line
                        linesWithSameXabove[linesWithSameXabove.GetLength(0) - 1][0] = l;
                        linesWithSameXabove[linesWithSameXabove.GetLength(0) - 1][1] = Math.Abs(YXw - bron[w][1]);
                    }

                    if (YXw < bron[w][1])
                    { //above it
                        Array.Resize(ref linesWithSameXunder, linesWithSameXunder.GetLength(0) + 1);
                        linesWithSameXunder[linesWithSameXunder.GetLength(0) - 1] = new double[2];
                        linesWithSameXunder[linesWithSameXunder.GetLength(0) - 1][0] = l;
                        linesWithSameXunder[linesWithSameXunder.GetLength(0) - 1][1] = Math.Abs(YXw - bron[w][1]);
                    }

                }
                //check the Y-coordinates
                if ((bron[w][1] >= line[l][1] && bron[w][1] <= line[l][3]) || (bron[w][1] <= line[l][1]
    && bron[w][1] >= line[l][3]))
                {
                    //1. calculate X(Yw) (Y is known, X is unknown)
                    if (line[l][0] == line[l][2])
                    { //m would be give devide by 0 error
                        XYw = XN[l];
                    }
                    else
                    {
                        m = (line[l][3] - line[l][1]) / (line[l][2] - line[l][0]);
                        if (m == 0)
                        {
                            XYw = XN[l];
                        }
                        else
                        {
                            XYw = (bron[w][1] + m * line[l][0] - line[l][1]) / m;
                        }
                    }

                    //2. Fill in the array linesWith...
                    if (XYw == bron[w][0])
                    {
                        //increase size by one
                        Array.Resize(ref linesWithSameYleft, linesWithSameYleft.GetLength(0) + 1);
                        Array.Resize(ref linesWithSameYright, linesWithSameYright.GetLength(0) + 1);

                        //create new element
                        linesWithSameYleft[linesWithSameYleft.GetLength(0) - 1] = new double[2];
                        linesWithSameYright[linesWithSameYright.GetLength(0) - 1] = new double[2];

                        //insert values
                        linesWithSameYleft[linesWithSameYleft.GetLength(0) - 1][0] = l;
                        linesWithSameYleft[linesWithSameYleft.GetLength(0) - 1][1] = 0;
                        linesWithSameYright[linesWithSameYright.GetLength(0) - 1][0] = l;
                        linesWithSameYright[linesWithSameYright.GetLength(0) - 1][1] = 0;
                    }

                    if (XYw > bron[w][0])
                    { //right of it it
                        Array.Resize(ref linesWithSameYright, linesWithSameYright.GetLength(0) + 1);
                        linesWithSameYright[linesWithSameYright.GetLength(0) - 1] = new double[2];
                        linesWithSameYright[linesWithSameYright.GetLength(0) - 1][0] = l;
                        linesWithSameYright[linesWithSameYright.GetLength(0) - 1][1] = Math.Abs(XYw -  ↙
  bron[w][0]);
                    }

                    if (XYw < bron[w][0])
                    { //left of it
                        Array.Resize(ref linesWithSameYleft, linesWithSameYleft.GetLength(0) + 1);
                        linesWithSameYleft[linesWithSameYleft.GetLength(0) - 1] = new double[2];
                        linesWithSameYleft[linesWithSameYleft.GetLength(0) - 1][0] = l;
                        linesWithSameYleft[linesWithSameYleft.GetLength(0) - 1][1] = Math.Abs(XYw -    ↙
  bron[w][0]);
                    }

                }//end check Y-coordinates
            }//end for all lines

            //The arrays should now be sorted
            sortJarredArray(linesWithSameXabove);
            sortJarredArray(linesWithSameXunder);
            sortJarredArray(linesWithSameYleft);
            sortJarredArray(linesWithSameYright);

            /* on the first position of each array is now the smallest distance
             * between the well and the lines, going through all of them will
             * result in the zone that the well is in!
             */

            found = false;

            //posibility 1: well is on a line linesWith...[0][1] = 0
            if (linesWithSameXabove[0][1] == 0 || linesWithSameYleft[0][1] == 0)
            {
                //on the interface or on the boundary
                if (linesWithSameXabove[0][1] == 0)
                {
                    zoneNumber[w][0] = zone[(int)linesWithSameXabove[0][0]][0];
                    zoneNumber[w][1] = zone[(int)linesWithSameXabove[0][0]][1];
                }
                if (linesWithSameYleft[0][1] == 0)
                {
                    zoneNumber[w][0] = zone[(int)linesWithSameYleft[0][0]][0];
                    zoneNumber[w][1] = zone[(int)linesWithSameYleft[0][0]][1];
                }
                found = true;
            }

            else
            {
                //find the zone (4 equal zone numbers)
                //check if rightminP, underminP, leftminP in een van de twee zone elementen aboveminP zijn 1ste zone zitten hebben

                if (zone[(int)linesWithSameXabove[0][0]][0] == zone[(int)linesWithSameYright[0][0]][0] || zone[(int)linesWithSameXabove[0][0]][0] == zone[(int)linesWithSameYright[0][0]][1])
                {
                    if (zone[(int)linesWithSameXabove[0][0]][0] == zone[(int)linesWithSameXunder[0][0]][0] || zone[(int)linesWithSameXabove[0][0]][0] == zone[(int)linesWithSameXunder[0][0]][1])
                    {
                        if (zone[(int)linesWithSameXabove[0][0]][0] == zone[(int)linesWithSameYleft[0][0]][0] || zone[(int)linesWithSameXabove[0][0]][0] == zone[(int)linesWithSameYleft[0][0]][1])
                        {
                            if (zone[(int)linesWithSameXabove[0][0]][0] != -1)
                            {
                                zoneNumber[w][0] = zone[(int)linesWithSameXabove[0][0]][0];
                                found = true;
                            }
                            else { zoneNumber[w][0] = -1; }
                        }
                        else { zoneNumber[w][1] = -1; }
                    }
                    else { zoneNumber[w][1] = -1; }
                }
                else { zoneNumber[w][1] = -1; }

                if (zone[(int)linesWithSameXabove[0][0]][1] == zone[(int)linesWithSameYright[0][0]][0] || zone[(int)linesWithSameXabove[0][0]][1] == zone[(int)linesWithSameYright[0][0]][1])
                {
                    if (zone[(int)linesWithSameXabove[0][0]][1] == zone[(int)linesWithSameXunder[0][0]][0] || zone[(int)linesWithSameXabove[0][0]][1] == zone[(int)linesWithSameXunder[0][0]][1])
                    {
                        if (zone[(int)linesWithSameXabove[0][0]][1] == zone[(int)linesWithSameYleft[0][0]][0] || zone[(int)linesWithSameXabove[0][0]][1] == zone[(int)linesWithSameYleft[0][0]][1])
                        {
                            if (zone[(int)linesWithSameXabove[0][0]][1] != -1)
                            {
                                if (found != true)
                                {
                                    zoneNumber[w][1] = zone[(int)linesWithSameXabove[0][0]][1];
                                    found = true;
                                }
                                else
                                {
                                    /* here is the problem that it might be that the well is located
                                     * in a zone that is located in an other zone. The for lines around
                                     * the well will thus have exactly the same two zones! An extra eq
                                     * will now decide in what region it is located
                                     */
                                    bool second = new bool();
                                    second = false;

                                    if (linesWithSameXabove.GetLength(0) > 1)
                                    {
                                        if (linesWithSameXabove[0][0] == linesWithSameXabove[1][0])
                                        {
                                            if (linesWithSameXabove.GetLength(0) > 2)
                                            {
                                                if (zoneNumber[w][0] == zone[(int)linesWithSameXabove[2][0]][0] || zoneNumber[w][0] == zone[(int)linesWithSameXabove[2][0]][1])
                                                {
                                                    zoneNumber[w][0] = zoneNumber[w][1];
                                                    zoneNumber[w][1] = -1;
                                                }
                                                else
                                                {
                                                    zoneNumber[w][1] = -1;
                                                }
                                                second = true; //found out what is the exact zone
                                            }
                                        }
                                        else
                                        {
                                            if (zoneNumber[w][0] == zone[(int)linesWithSameXabove[1][0]][0] || zoneNumber[w][0] == zone[(int)linesWithSameXabove[1][0]][1])
                                            {
                                                zoneNumber[w][0] = zoneNumber[w][1];
                                                zoneNumber[w][1] = -1;
                                            }
                                            else
                                            {
                                                zoneNumber[w][1] = -1;
                                            }
                                            second = true; //found out what is the exact zone
                                        }
                                    }//end for the 1st point (above the well)
                                    if (second == false)
                                    {//for the second point: right of the well
                                        if (linesWithSameYright.GetLength(0) > 1)
                                        {
                                            if (linesWithSameYright[0][0] == linesWithSameYright[1][0])
                                            {
                                                if (linesWithSameYright.GetLength(0) > 2)
                                                {
                                                    if (zoneNumber[w][0] == zone[(int)linesWithSameYright[2][0]][0] || zoneNumber[w][0] == zone[(int)linesWithSameYright[2][0]][1])
                                                    {
                                                        zoneNumber[w][0] = zoneNumber[w][1];
                                                        zoneNumber[w][1] = -1;
                                                    }
                                                    else
                                                    {
                                                        zoneNumber[w][1] = -1;
                                                    }
                                                    second = true; //found out what is the exact zone

                                                }
                                            }
                                            else
                                            {
                                                if (zoneNumber[w][0] == zone[(int)linesWithSameYright[1][0]][0] || zoneNumber[w][0] == zone[(int)linesWithSameYright[1][0]][1])
                                                {
                                                    zoneNumber[w][0] = zoneNumber[w][1];
                                                    zoneNumber[w][1] = -1;
                                                }
                                                else
                                                {
                                                    zoneNumber[w][1] = -1;
                                                }
                                                second = true; //found out what is the exact zone
                                            }
                                        }
                                    }//end if second is false for 2nd point
                                    if (second == false)
                                    {//for the 3th point (under)
                                        if (linesWithSameXunder.GetLength(0) > 1)
                                        {
                                            if (linesWithSameXunder[0][0] == linesWithSameXunder[1][0])
                                            {
                                                if (linesWithSameXunder.GetLength(0) > 2)
                                                {
                                                    if (zoneNumber[w][0] == zone[(int)linesWithSameXunder[2][0]][0] || zoneNumber[w][0] == zone[(int)linesWithSameXunder[2][0]][1])
                                                    {
                                                        zoneNumber[w][0] = zoneNumber[w][1];
                                                        zoneNumber[w][1] = -1;
                                                    }
                                                    else
                                                    {
                                                        zoneNumber[w][1] = -1;
                                                    }
                                                    second = true; //found out what is the exact zone
                                                }
                                            }
                                            else
                                            {
                                                if (zoneNumber[w][0] == zone[(int)linesWithSameXunder[1][0]][0] || zoneNumber[w][0] == zone[(int)linesWithSameXunder[1][0]][1])
                                                {
                                                    zoneNumber[w][0] = zoneNumber[w][1];
                                                    zoneNumber[w][1] = -1;
                                                }
                                                else
                                                {
                                                    zoneNumber[w][1] = -1;
                                                }
                                                second = true; //found out what is the exact zone
                                            }
                                        }
                                    }//end if second is false for 3th point
                                    if (second == false)
                                    {//for the 4th point (left)
                                        if (linesWithSameXunder.GetLength(0) > 1)
                                        {
                                            if (linesWithSameYleft[0][0] == linesWithSameYleft[1][0])
                                            {
                                                if (linesWithSameYleft.GetLength(0) > 2)
                                                {
                                                    if (zoneNumber[w][0] == zone[(int)linesWithSameYleft[2][0]][0] || zoneNumber[w][0] == zone[(int)linesWithSameYleft[2][0]][1])
                                                    {
                                                        zoneNumber[w][0] = zoneNumber[w][1];
                                                        zoneNumber[w][1] = -1;
                                                    }
                                                    else
                                                    {
                                                        zoneNumber[w][1] = -1;
                                                    }
                                                    second = true; //found out what is the exact zone
                                                }
                                            }
                                            else
                                            {
                                                if (zoneNumber[w][0] == zone[(int)linesWithSameYleft[1][0]][0] || zoneNumber[w][0] == zone[(int)linesWithSameYleft[1][0]][1])
                                                {
                                                    zoneNumber[w][0] = zoneNumber[w][1];
                                                    zoneNumber[w][1] = -1;
                                                }
                                                else
                                                {
                                                    zoneNumber[w][1] = -1;
                                                }
                                                second = true; //found out what is the exact zone
                                            }
                                        }
                                    }//end if second is false for 4th point

                                    //if still false: then give error
                                    MessageBox.Show("Was trying to find the exact zone as an inclosed zone but failed");
                                }
                            }
                            else { zoneNumber[w][1] = -1; }
                        }
                        else { zoneNumber[w][1] = -1; }
                    }
                    else { zoneNumber[w][1] = -1; }
                }
                else { zoneNumber[w][1] = -1; }
            }

            if (found == false)
            {
                MessageBox.Show("An error occured, it was impossible to retrieve the zonenumber");
            }




            //store the zone in all well[w][3]

            if (zoneNumber[w][0] == -1 && zoneNumber[w][1] == -1)
            {
                MessageBox.Show("No zone found");
            }
            else if (zoneNumber[w][0] == -1 || zoneNumber[w][1] == -1)
            {
                //one zone is found
                if (zoneNumber[w][0] == -1)
                {
                    bron[w][3] = (int)zoneNumber[w][1];
                }
                else
                {
                    bron[w][3] = (int)zoneNumber[w][0];
                }
            }

        }//end findOutZoneIntelligent

        public void fillCalculatedWellPosition(double[][] well, int i, ref double[][] 
    CalculatedWellPosition, ref double[] CalculatedWellZone)
        {
            bool copy = new bool();

            copy = true; //A test will find out if it should be set to false

            //see if the fitnessvalue is already in the Calculatedfitness matrix
            for (int j = 0; j < CalculatedWellPosition.GetLength(0); j++)
            {//j is the counter representing the CalculatedFitness

                if (well[i][0] == CalculatedWellPosition[j][0])
                {
                    //multiple well positions with the correspondending x value may exist, the y should 
     be checked as well
                    if (well[i][1] == CalculatedWellPosition[j][1])
                    {
                        copy = false;
                        j = CalculatedWellPosition.GetLength(0);
                    }
                }//end if (fitness[i] == Calculatedfitness[j])
            }//end for each chromosome in the store matrices

            if (copy == true)
            {
                //0. New size of the arrays
                int newSize = CalculatedWellPosition.GetLength(0) + 1;

                //1. Resize the CalculatedWellZone and fill
                Array.Resize(ref CalculatedWellZone, newSize);
                Array.Copy(well[i], 3, CalculatedWellZone, newSize - 1, 1);

                //2. Resize the CalculatedChromosomes and fill
                Array.Resize(ref CalculatedWellPosition, newSize);
                CalculatedWellPosition[newSize - 1] = new double[2];
                for (int s = 0; s < 2; s++)
                {
                    Array.Copy(well[i], s, CalculatedWellPosition[newSize - 1], s, 1);
                }
            }//end if (copy == true)
        }//end void fillCalculatedChromosomes

        public void resizeMultiDimensionalArray(ref double[,] original, int rows, int cols)
        {
            double[,] newArray = new double[rows, cols];
            original = newArray;
        }//end resizeMultiDimensionalArray

        public void AddToUPlaatsXandY(ref int[] uplaatsX, ref int[] uplaatsY, int[][] zone, bool[] lineOnCoast, int numberOfCoastLines)
        {
            int i = uplaatsX.GetLength(0) - numberOfCoastLines;
            int j = uplaatsY.GetLength(0) - numberOfCoastLines;

            //for all the nodes not on the interface
            for (int I = 0; I < zone.GetLength(0); I++)
            {
                if (lineOnCoast[I] == true)
                {
                    uplaatsX[i] = I; //nodes have to be numbers from one to N, and always increased by 1.
                    uplaatsY[j] = I;
                    i++;
                    j++;

                    if (zone[I][1] != -1)
                    {
                        MessageBox.Show("Error while calculating uplaatsX");
                    }
                }
            }
        }//addToUPlaatsXandY

        public void CopyKnownValuesOfAandBt(double[,] uA, double[,] uBt, double[,] A, double[,] Bt)
        {
            //first copy everything for uA to A
            for (int i = 0; i < uA.GetLength(0); i++)
            {
                for (int j = 0; j < uA.GetLength(1); j++)
                {
                    double tempElement = uA[i, j];
                    A[i, j] = tempElement;
                }
            }

            //second copy everything from uBt to Bt
            for (int i = 0; i < uBt.GetLength(0); i++)
            {
                for (int j = 0; j < uBt.GetLength(1); j++)
                {
                    double tempElement = uBt[i, j];
                    Bt[i, j] = tempElement;
                }
            }

        }//end CopyKnownValuesOfAandBt

        public void calculateAandBt(double[,] uA, double[,] uBt, ref double[,] A, ref double[,] Bt, int[] uplaatsX, int[] uplaatsY, int[] K, int[][] zone, double[][] line, double[] L, double[] XN, double[] YN, double[] T, bool[] lineOnCoast, bool[,] Acal, bool[,] Btcal)
        {
            for (int I = 0; I < zone.GetLength(0); I++)
            {
                int rij = Array.IndexOf(uplaatsX, I);

                //write first equation: for node on interface or not, it is the same
                for (int J = 0; J < zone.GetLength(0); J++)
                {
                    if (lineOnCoast[I] == true || lineOnCoast[J] == true)
                    {
                        if (zone[J][0] == zone[I][0] || zone[J][1] == zone[I][0])
                        {
                            //when J is on the interface
                            if (zone[J][1] != -1)
                            {
                                Acal[rij, Array.IndexOf(uplaatsX, J)] = true;
                                Acal[rij, Array.LastIndexOf(uplaatsX, J)] = true;

                                //is J defined in same zone as I (otherwise problem with L and g*(-To/T1)
                                if (zone[J][0] == zone[I][0])
                                { //they are defined in the same zone: no problem
                                    if (I == J)
                                    {
                                        A[rij, Array.IndexOf(uplaatsX, J)] = -0.5; // = h
                                        A[rij, Array.LastIndexOf(uplaatsX, J)] = -L[J] / (2 * Math.PI) * (Math.Log(L[J] / 2) - 1); // =-g
                                    }
                                    else
                                    {
                                        A[rij, Array.IndexOf(uplaatsX, J)] = Hon(XN[I], line[J][0], line[J][2], YN[I], line[J][1], line[J][3]); // = h
                                        A[rij, Array.LastIndexOf(uplaatsX, J)] = -Gon(XN[I], line[J][0], line[J][2], YN[I], line[J][1], line[J][3], L[J]); // =-g
                                    }
                                }
                                else
                                { //they are not defined in the same zone: pay attention!
                                    if (I == J)
                                    {
                                        A[rij, Array.IndexOf(uplaatsX, J)] = -0.5;// =h
                                        A[rij, Array.LastIndexOf(uplaatsX, J)] = -L[J] / (2 * Math.PI) * (Math.Log(L[J] / 2) - 1) * (-T[zone[J][0]] / T[zone[J][1]]);//-g
                                    }
                                    else
                                    {
                                        A[rij, Array.IndexOf(uplaatsX, J)] = Hon(XN[I], line[J][2], line[J][0], YN[I], line[J][3], line[J][1]);// =h
                                        A[rij, Array.LastIndexOf(uplaatsX, J)] = -Gon(XN[I], line[J][2], line[J][0], YN[I], line[J][3], line[J][1], L[J]) * (-T[zone[J][0]] / T[zone[J][1]]); //-g

                                    }
                                }

                            }

                            //when J is not on the interface
                            else
                            {
                                //there can be no problem with L or g*(-To/T1), K1 decides
                                Acal[rij, Array.IndexOf(uplaatsX, J)] = true;
                                Btcal[rij, Array.IndexOf(uplaatsY, J)] = true;

                                if (K1[J] == 0) //u is given so colums should be changed
                                {
                                    if (I == J)
                                    {
                                        A[rij, Array.IndexOf(uplaatsX, J)] = -L[J] / (2 * Math.PI) * (Math.Log(L[J] / 2) - 1); //-g
                                        Bt[rij, Array.IndexOf(uplaatsY, J)] = 0.5; //-h
                                    }
                                    else
                                    {
                                        A[rij, Array.IndexOf(uplaatsX, J)] = -Gon(XN[I], line[J][0], line[J][2], YN[I], line[J][1], line[J][3], L[J]); //-g
                                        Bt[rij, Array.IndexOf(uplaatsY, J)] = -Hon(XN[I], line[J][0], line[J][2], YN[I], line[J][1], line[J][3]); //-h
                                    }
                                }
                                else //no problem, colums can stay. (uK1[J] == 1)
                                {
                                    if (I == J)
                                    {
```

```
                                    A[rij, Array.IndexOf(uplaatsX, J)] = -0.5; //h
                                    Bt[rij, Array.IndexOf(uplaatsY, J)] = L[J] / (2 * Math.PI) * (Math.Log(L[J] / 2) - 1); //g
                                }
                                else
                                {
                                    A[rij, Array.IndexOf(uplaatsX, J)] = Hon(XN[I], line[J][0], line[J][2], YN[I], line[J][1], line[J][3]); //h
                                    Bt[rij, Array.IndexOf(uplaatsY, J)] = Gon(XN[I], line[J][0], line[J][2], YN[I], line[J][1], line[J][3], L[J]); //g
                                }
                            }
                        }
                    }
                }//end if one of the line elements is on the coast
            }//end for all J


            //write second equation: only for nodes on the interface
            if (zone[I][1] != -1)
            {
                rij = Array.LastIndexOf(uplaatsX, I);

                //write second equation: only for nodes I on the interface
                for (int J = 0; J < zone.GetLength(0); J++)
                {
                    if (lineOnCoast[I] == true || lineOnCoast[J] == true)
                    {

                        //check if an equation should be written towards this point
                        if (zone[J][0] == zone[I][1] || zone[J][1] == zone[I][1])
                        {
                            //when J is on the interface
                            if (zone[J][1] != -1)
                            {
                                Acal[rij, Array.IndexOf(uplaatsX, J)] = true;
                                Acal[rij, Array.LastIndexOf(uplaatsX, J)] = true;

                                //is J defined in same zone as I (otherwise problem with L and g*(-To/T1)
                                if (zone[J][0] == zone[I][1])
                                { //they are defined in the same zone: no problem

                                    if (I == J)
                                    {
                                        A[rij, Array.IndexOf(uplaatsX, J)] = -0.5; // = h
                                        A[rij, Array.LastIndexOf(uplaatsX, J)] = -L[J] / (2 * Math.PI) * (Math.Log(L[J] / 2) - 1); // =-g, voorlopig geen teken wissel
                                    }
                                    else
                                    {
                                        A[rij, Array.IndexOf(uplaatsX, J)] = Hon(XN[I], line[J][0], line[J][2], YN[I], line[J][1], line[J][3]); // = h
                                        A[rij, Array.LastIndexOf(uplaatsX, J)] = -Gon(XN[I], line[J][0], line[J][2], YN[I], line[J][1], line[J][3], L[J]); // =-g
                                    }

                                }
                                else
                                { //they are not defined in the same zone: pay attention!

                                    if (I == J)
                                    {
                                        A[rij, Array.IndexOf(uplaatsX, J)] = -0.5;// =h
                                        A[rij, Array.LastIndexOf(uplaatsX, J)] = -L[J] / (2 * Math.PI) * (Math.Log(L[J] / 2) - 1) * (-T[zone[J][0]] / T[zone[J][1]]); //-g
                                    }
                                    else
                                    {
                                        A[rij, Array.IndexOf(uplaatsX, J)] = Hon(XN[I], line[J][2], line[J][0], YN[I], line[J][3], line[J][1]);//h
                                        A[rij, Array.LastIndexOf(uplaatsX, J)] = -Gon(XN[I], line[J][2], line[J][0], YN[I], line[J][3], line[J][1], L[J]) * (-T[zone[J][0]] / T[zone[J][1]]); //-g
                                    }
                                }

                            }

                            //when J is not on the interface
                            else
                            {
                                Acal[rij, Array.IndexOf(uplaatsX, J)] = true;
                                Btcal[rij, System.Array.IndexOf(uplaatsY, J)] = true;
                                //there can be no problem with L or g*(-To/T1), K1 decides

                                if (K1[J] == 0) //u is given so colums should be changed
                                {
                                    if (I == J)
                                    {
                                        A[rij, Array.IndexOf(uplaatsX, J)] = -L[J] / (2 * Math.PI)  ↙
* (Math.Log(L[J] / 2) - 1); //-g
                                        Bt[rij, Array.IndexOf(uplaatsY, J)] = 0.5; //-h
                                    }
                                    else
                                    {
                                        A[rij, Array.IndexOf(uplaatsX, J)] = -Gon(XN[I], line[J][0] ↙
, line[J][2], YN[I], line[J][1], line[J][3], L[J]); //-g
                                        Bt[rij, Array.IndexOf(uplaatsY, J)] = -Hon(XN[I], line[J]   ↙
[0], line[J][2], YN[I], line[J][1], line[J][3]); //-h
                                    }
                                }
                                else //no problem, colums can stay.
                                {
                                    if (I == J)
                                    {
                                        A[rij, Array.IndexOf(uplaatsX, J)] = -0.5; //h
                                        Bt[rij, Array.IndexOf(uplaatsY, J)] = L[J] / (2 * Math.PI)  ↙
* (Math.Log(L[J] / 2) - 1); //g
                                    }
                                    else
                                    {
                                        A[rij, Array.IndexOf(uplaatsX, J)] = Hon(XN[I], line[J][0], ↙
 line[J][2], YN[I], line[J][1], line[J][3]); //h
                                        Bt[rij, Array.IndexOf(uplaatsY, J)] = Gon(XN[I], line[J][0] ↙
, line[J][2], YN[I], line[J][1], line[J][3], L[J]); //g
                                    }
                                }

                            }
                        }//end if equation should be written
                    }
                }//end if one of them is true
            }//end for all J
        }//end for all nodes I

    }//end calculateAandBt

    public void calculateB(ref double[] B, int[] uplaatsY, double[,] Bt, double[] BV)
    {
        // fill the B array
        for (int I = 0; I < Bt.GetLength(0); I++)
        {
            B[I] = 0;

            for (int J = 0; J < Bt.GetLength(1); J++)
            {
                B[I] = B[I] + Bt[I, J] * BV[uplaatsY[J]];
            }//end for loop J
        }//end for loop I
    }//end calculateB

    public void wellinfluenceSmart(double[][] well, double[] XN, double[] YN, double[] B, double[]  ↙
T, int[] uplaatsX, int[][] zone)
    {
        //loop through all (w)ells
        for (int w = 0; w < well.GetLength(0); w++)
        {
            int rij = 0; //counter that indicated the row in B (for every well start from the first ↙
 equation

            for (int I = 0; I < XN.GetLength(0); I++)
            {
                if (zone[I][0] == well[w][3]) //are they in the same zone
                {
                    rij = Array.IndexOf(uplaatsX, I);
                    B[rij] = B[rij] - (well[w][2] / (2 * Math.PI * T[(int)well[w][3]])) * Math.Log(Math.Sqrt(Math.Pow(XN[I] - well[w][0], 2) + Math.Pow(YN[I] - well[w][1], 2)));
                }

                if (zone[I][1] != -1)
                {
                    if (zone[I][1] == well[w][3]) //are they in the same zone
                    {
                        rij = Array.LastIndexOf(uplaatsX, I);
                        B[rij] = B[rij] - (well[w][2] / (2 * Math.PI * T[(int)well[w][3]])) * Math.Log(Math.Sqrt(Math.Pow(XN[I] - well[w][0], 2) + Math.Pow(YN[I] - well[w][1], 2)));
                    }
                } //end if on the interface
            } //end for all elements I
        }//end for all wells
    }//end wellinfluence

    public void solveInteligggent(double[,] A, double[] B, double[] X)
    {
        /* this script works for square matrices, with whatever dimensions.
         * When an element on the diagonal is zero, colums will be swapt in order not to have problems
         */

        //variables needed...
        double[] Atemp = new double[A.GetLength(1)]; // temporary array for the switch
        double Btemp = 0; // temporary array for the switch
        double sf = 0; //factor for scaling
        Boolean found = new Boolean();

        for (int I = 0; I < (A.GetLength(0) - 1); I++)//the last line (and colum) should not be done
        {
            found = true; //at the start of each run set it true, when A[I,I] != 0 it will be set to false
            //row per row we will work
            //find maximum value of the colum, starting from the row where we are on (I)

            if (A[I, I] == 0)
            { //there a problem, there is a zero on a place we do not like it at all!
                found = false; //there is a zero on A[I,I]

                //1) look if there is in this colum a row that has a value different of 0
                for (int i = I + 1; i < A.GetLength(0); i++)
                {
                    if (A[i, I] != 0) //when this value is not zero we will swap rows and use this row to make the rest 0
                    {

                        //de rij met de maxima wegschrijven in de matrixes Atemp and Btemp

                        for (int j = I; j < A.GetLength(0); j++) //for row I, write all colomvalues starting at J to temp array
                        {
                            //eerst de A matrix
                            Atemp[j] = A[i, j]; //wegschrijven array met waarden van de rij waar niet nul
                        }

                        Btemp = B[i];

                        //daarna de rijen verwisselen (1: overschrijf de rij met de maximale nummers, 2: overschrijf de beschouwde rij)
                        for (int j = I; j < A.GetLength(0); j++)
                        {
                            //eerst de A matrix
                            A[i, j] = A[I, j];
```

```
                                A[I, j] = Atemp[j];

                            }

                            //de matrix B herschikken
                            B[i] = B[I];
                            B[I] = Btemp;

                            found = true; //Yes we found a value different from 0! Hoera!
                            i = A.GetLength(0); //set i high enough to stop the search for a value that is not zero
                        }
                    } //end changing rows to get A[I,I] != 0

                    //2) in the worst situation there were only 0's in the colum, we then should to colum changed
                    if (found == false)
                    {
                        /* we did not find a row with a value different from 0! So now we will try by changing a colum
                         * Look to the first colum on the right, if in it has values on its rows that are not zero, then
                         * swap, if there are non, check with the colum one time more on the right of it, and so on,
                         * if even the last colum only exists of 0... then give an error message. something went wrong
                         * if we by this succeeded in creating a non A[I,I] element, we put found on true !!!
                         * don't forget the X matrix (the B matrix remains unchanged by colum operations)
                         */



                    }

                    //3) if found is still false, then give an error message en stop the progress
                    if (found == false)
                    {
                        MessageBox.Show("An error occured, the matrix is singular! Proces stopped and no solution was found!");
                        I = A.GetLength(0); //set I high enough to stop the cycle!
                    }
                }


                /* We are now sure that there is no 0 on the A[I,I] and can use the value of A[I,I] to
                 * empty the rows below it!
                 */
                if (found == true)
                {
                    //a non zer0 A[I,I] value was found: we can now use it to eliminate the values in the colums of the rows under it!

                    /* Start not at I, but at I+1, because the Ith row is the one used
                    * to make the others 0 in the Jth colum
                    * j starts at J, dont forget matrix B!
                    */

                    for (int i = I + 1; i < A.GetLength(0); i++)
                    {
                        sf = (A[i, I] / A[I, I]);

                        //eerste de A matrix
                        for (int j = I; j < A.GetLength(1); j++)
                        {
                            A[i, j] = A[i, j] - sf * A[I, j];
                        }

                        //daarna de B matrix
                        B[i] = B[i] - sf * B[I];
                    }
                }
            }

            //emptying X
            for (int i = 0; i < X.GetLength(0); i++)
            {
                X[i] = 0;
            }


            //Matrices have new been ordened, they can now be used by backsolving it to X
            for (int k = X.GetLength(0) - 1; k >= 0; k--)
            {

                double sum = 0;
                for (int j = k + 1; j < X.GetLength(0); j++)
                {
                    sum = sum + A[k, j] * X[j];
                }
                X[k] = (B[k] - sum) / A[k, k];
            }
        }//end solveInteliggent

        public void reorderSmart(double[] BV, double[] X, int[] K1, double[] U, double[] Un, int[][]
    zone, int[] uplaatsX)
        {
            /* This function places the calculated and know values of u in the U vector
             * and the values of un in the Un vector
             * Herefore it uses the BV vector (with the known values) and the X vector
             * with the calculated values. The K1 vector keeps track of what was given
             * and makes the decission to write to U or to Un
             */

            for (int i = 0; i < zone.GetLength(0); i++)
            {
                // are we dealing with a point on the intersection? Then u and u_n should be written
                if (zone[i][1] != -1)
                {
                    U[i] = X[Array.IndexOf(uplaatsX, i)];
                    Un[i] = X[Array.LastIndexOf(uplaatsX, i)];
                }
                else
                {
                    if (K1[i] == 0)
                    {
                        U[i] = BV[i];
                        Un[i] = X[Array.IndexOf(uplaatsX, i)];
                    }
                    else
                    {
                        U[i] = X[Array.IndexOf(uplaatsX, i)];
                        Un[i] = BV[i];
                    }
                }
            }

        }//end reorderSmart

        public void calculatefitnessfunction(bool[] lineOnCoast, double[] Un, double[] fitness, int
    chromosomeCounter, string[][] chromosomes, double[] dmin, int fitnessFunction, double C1, double C2
    , double C3, double C4)
        {

            /* Pay attention that when working with multiple zones, that then the numbering
             * of lineOnfCoast is the same of the lines, otherwise the wrong lines will be
             * selected...
             */

            if (fitnessFunction == 0)
            {
                //fitnessfunction according Katsifarakis

                double sumQ = 0;
                for (int w = 0; w < well.GetLength(0); w++)
                {
                    sumQ = sumQ + well[w][2];
                }



                double PEN = 0;
                double B = 0;
                int k = 0;

                for (int s = 0; s < lineOnCoast.GetLength(0); s++)
                {
                    if (lineOnCoast[s] == true)
                    {
                        if (Un[s] > 0)
                        {
                            if (zone[s][0] != -1 && zone[s][1] == -1)
                            {
                                B = B + Un[s] * L[s] * T[zone[s][0]];
                            }
                            else if (zone[s][1] != -1 && zone[s][0] == -1)
                            {
                                B = B + Un[s] * L[s] * T[zone[s][1]];
                            }
                            else
                            {
                                MessageBox.Show("Zone undifined");
                            }
                            k++;
                        }
                    }
                }
                //nog aanpassen! well niet zeker in zone 0!
                PEN = (C1 * k + C2 * B);
                fitness[chromosomeCounter] = sumQ - PEN;
            }

            if (fitnessFunction == 1)
            {
                //fixed input parameters

                 //in euro per liter second
                double pricespwpersquremeter = 174;                 //in euro per m²
                double tinyear = 10;                                //number of years (in years)
                double pricewater = 0.1;                          //in m³/s
                double t = tinyear * 365 * 24 * 60 * 60;            //in s
                double h = 10;                                      //height of the spw in meter

                // 1. extra income because of extra water flow

                double IncomeWater = 0;
                int d = 0; //counter for the dmin array

                for (int w = 0; w < well.GetLength(0); w++)
                {
                    if (hwell[w][2] == true)
                    {
                        IncomeWater = IncomeWater + (well[w][2] - dmin[d]);
                        d++;
                    }
                }

                IncomeWater = IncomeWater * pricewater * t; // ( m³/s * Euro/m³ * s = Euro )

                //2. extra cost because of the spw that needs to be constructed
                double beginSpw = doubleChromosome(chromosomes[chromosomeCounter][chromosomes ↙
            [chromosomeCounter].GetLength(0) - 2], spw_min, spw_max, chromosomes[chromosomeCounter][chromosomes ↙
            [chromosomeCounter].GetLength(0) - 2].Length);
                double lengthSpw;
                if (fixed_spw_length == true)
                {
                    lengthSpw = spw_length;
                }
                else
                {
                    lengthSpw = doubleChromosome(chromosomes[chromosomeCounter][chromosomes ↙
    [chromosomeCounter].GetLength(0) - 1], 0, 1, chromosomes[chromosomeCounter][chromosomes ↙
    [chromosomeCounter].GetLength(0) - 1].Length) * (spw_max - beginSpw);
                }
                double CostSpw = lengthSpw * h * pricespwpersquremeter; // (m * m * euro/m² = euro)

                //3 & 4. inflow through all coastal lines (B) and number of boundary elements with ↙
    inflow (k)

                double B = 0;
                int k = 0;

                for (int s = 0; s < lineOnCoast.GetLength(0); s++)
                {
                    if (lineOnCoast[s] == true)
                    {
                        if (Un[s] > 0)
                        {
                            if (zone[s][0] != -1 && zone[s][1] == -1)
                            {
                                B = B + Un[s] * L[s] * T[zone[s][0]];
                            }
                            else if (zone[s][1] != -1 && zone[s][0] == -1)
                            {
                                B = B + Un[s] * L[s] * T[zone[s][1]];
                            }
                            else
                            {
                                MessageBox.Show("Zone undifined");
                            }
                            k++;
                        }
                    }
                }// end for all lines

                //4. Calculate fitness
                fitness[chromosomeCounter] = C1 * IncomeWater - (C2 * CostSpw + C3 * k + C4 * B*t);

            }
            if (fitnessFunction == 2)
            {
                //scaled fitness function
                //fixed input parameters

                //in euro per liter second
                double pricespwpersquremeter = 174;                  //in euro per m²
                double tinyear = 10;                                 //number of years (in years)
                double pricewater = 0.1;                            //in m³/s
                double t = tinyear * 365 * 24 * 60 * 60;             //in s
                double h = 10;                                       //height of the spw in meter

                // 1. extra income because of extra water flow

                double IncomeWater = 0;
                double maxIncomeWater = 0;

                int d = 0; //counter for the dmin array

                for (int w = 0; w < well.GetLength(0); w++)
                {
                    if (hwell[w][2] == true)
                    {
                        maxIncomeWater = maxIncomeWater + (dmax[d] - dmin[d]);
                        d++;
                    }
                }

                maxIncomeWater = maxIncomeWater * t * pricewater;

                d = 0; //counter for the dmin array

                for (int w = 0; w < well.GetLength(0); w++)
                {
                    if (hwell[w][2] == true)
                    {
                        IncomeWater = IncomeWater + (well[w][2] - dmin[d]);
                        d++;
                    }
                }

                IncomeWater = IncomeWater * pricewater * t; // ( m³/s * Euro/m³ * s = Euro )

                //2. extra cost because of the spw that needs to be constructed
                double beginSpw = doubleChromosome(chromosomes[chromosomeCounter][chromosomes[chromosomeCounter].GetLength(0) - 2], spw_min, spw_max, chromosomes[chromosomeCounter][chromosomes[chromosomeCounter].GetLength(0) - 2].Length);
                double lengthSpw;
                if (fixed_spw_length == true)
                {
                    lengthSpw = spw_length;
                }
                else
                {
                    lengthSpw = doubleChromosome(chromosomes[chromosomeCounter][chromosomes[chromosomeCounter].GetLength(0) - 1], 0, 1, chromosomes[chromosomeCounter][chromosomes[chromosomeCounter].GetLength(0) - 1].Length) * (spw_max - beginSpw);
                }

                double CostSpw = lengthSpw * h * pricespwpersquremeter; // (m * m * euro/m² = euro)
                double maxCostSpw = cumulLineEnd[cumulLineEnd.GetLength(0) - 1] * h * pricespwpersquremeter;

                //3 & 4. inflow through all coastal lines (B) and number of boundary elements with inflow (k)

                double B = 0;
                int k = 0;
                int kmax = 0;

                for (int s = 0; s < lineOnCoast.GetLength(0); s++)
                {
                    if (lineOnCoast[s] == true)
                    {
                        if (Un[s] > 0)
                        {
                            if (zone[s][0] != -1 && zone[s][1] == -1)
                            {
                                B = B + Un[s] * L[s] * T[zone[s][0]];
                            }
                            else if (zone[s][1] != -1 && zone[s][0] == -1)
                            {
                                B = B + Un[s] * L[s] * T[zone[s][1]];
                            }
                            else
                            {
                                MessageBox.Show("Zone undifined");
                            }
                            k++;
                        }
                        kmax++;
                    }
                }// end for all lines

                //4. Calculate fitness
                fitness[chromosomeCounter] = C1 * IncomeWater/maxIncomeWater - C2 * CostSpw/maxCostSpw - C3 * k/kmax - C4 * B;
            }

            //fitnessfunction to see if the optimal length is found
            //double beginSpw = doubleChromosome(chromosomes[chromosomeCounter][chromosomes[chromosomeCounter].GetLength(0) - 2], 0, cumulLineEnd[cumulLineEnd.GetLength(0) - 1], chromosomes[chromosomeCounter][chromosomes[chromosomeCounter].GetLength(0) - 2].Length);

            //length is procentualy calculated from distance beginning to distance end
            //double lengthSpw = doubleChromosome(chromosomes[chromosomeCounter][chromosomes[chromosomeCounter].GetLength(0) - 1], 0, 1, chromosomes[chromosomeCounter][chromosomes[chromosomeCounter].GetLength(0) - 1].Length) * (cumulLineEnd[cumulLineEnd.GetLength(0) - 1] - beginSpw);
            //fitness[chromosomeCounter] = lengthSpw*lengthSpw;
```

```


    }//end calculatefitnessfunction

    public void fillCalculatedChromosomesAndInflowCharacteristics(double fitness, string[] chromosome, ref double[] Calculatedfitness, ref string[][] Calculatedchromosomes, ref double[] CalculatedTotalInflow, ref int[] CalculatedTotalInflowNodes, double[] Un, int[][] zone, bool[] lineOnCoast, double[] L, double[] T )
    {
        bool copy = new bool();
        int numberOfSubChromosomes = chromosome.GetLength(0);

            copy = true; //A test will find out if it should be set to false

            /* see if the fitnessvalue is already in the Calculatedfitness matrix
             * This has already been checked before the function is called, because the
             * function is only called when the matrices where calculated. The value that
             * will be insert, will thus be a new one for sure, because otherwise it would
             * never have been calculated in the first place
             */


            //for (int j = 0; j < Calculatedfitness.GetLength(0); j++)
            //{//j is the counter representing the CalculatedFitness

            //    if (fitness == Calculatedfitness[j])
            //    {
            //        //multiple chromosomes might have the same fitness so it should be checked if their subchromosomes are identical
            //        int numOk = 0;
            //        for (int s = 0; s < numberOfSubChromosomes; s++)
            //        {
            //            if (chromosome[s] == Calculatedchromosomes[j][s])
            //            {
            //                numOk++;
            //            }
            //        }//end for s

            //        if (numOk == numberOfSubChromosomes)
            //        {//then it should not be copied because they had been copied before already
            //            copy = false;
            //        }
            //    }//end if (fitness[i] == Calculatedfitness[j])
            //}//end for each chromosome in the store matrices

            if (copy == true)
            {
                //0. New size of the arrays
                int newSize = Calculatedfitness.GetLength(0) + 1;

                //1. Resize the Calculatedfitness and fill
                Array.Resize(ref Calculatedfitness, newSize);
                Calculatedfitness[newSize - 1] = fitness;
                //Array.Copy(fitness, i, Calculatedfitness, newSize - 1, 1);

                //2. Resize the CalculatedChromosomes and fill
                Array.Resize(ref Calculatedchromosomes, newSize);
                Calculatedchromosomes[newSize - 1] = new string[numberOfSubChromosomes];
                for (int s = 0; s < numberOfSubChromosomes; s++)
                {
                    Array.Copy(chromosome, s, Calculatedchromosomes[newSize - 1], s, 1);
                }

                //3.A Resize the CalculatedTotalInflow and CalculatedTotalInflowNodes
                Array.Resize(ref CalculatedTotalInflow, newSize);
                Array.Resize(ref CalculatedTotalInflowNodes, newSize);

                //3.B Calculate the value of the inflow and the number of boundary elements with inflow

                double B = 0;
                int k = 0;

                for (int s = 0; s < lineOnCoast.GetLength(0); s++)
```

```
                    {
                        if (lineOnCoast[s] == true)
                        {
                            if (Un[s] > 0)
                            {
                                if (zone[s][0] != -1 && zone[s][1] == -1)
                                {
                                    B = B + Un[s] * L[s] * T[zone[s][0]];
                                }
                                else if (zone[s][1] != -1 && zone[s][0] == -1)
                                {
                                    B = B + Un[s] * L[s] * T[zone[s][1]];
                                }
                                else
                                {
                                    MessageBox.Show("Zone undifined");
                                }
                                k++;
                            }
                        }
                    }
                    CalculatedTotalInflow[newSize-1] = B;
                    CalculatedTotalInflowNodes[newSize-1] = k;

                }//end if (copy == true)
        }//end void fillCalculatedChromosomes

        public void crossover(string[][] chromosomes, int row, double pc)
        {

            //crossover on only one
            //generate random number to see if crossover taks place

            /* calculate random between 0 and 1, to see if crossover takes place
             * If crossover takes place it taks place for all the substrings!
             */

            double R = Random.NextDouble();

            if (R <= pc) //crossover should take place
            {
                //in what chromosome crossover should take place
                int R1 = Random.Next(0, chromosomes[0].GetLength(0));

                for (int subchr = 0; subchr < chromosomes[0].GetLength(0); subchr++){
                    if (subchr == R1)
                    {
                        //length
                        int l = chromosomes[0][subchr].Length;

                        //1. Calculate the place where crossover should take place
                        int AA = Random.Next(1, l);

                        //2. Do the crossovert

                        string deel1Chromosome1 = chromosomes[row][subchr].Substring(0, AA);
                        string deel2Chromosome1 = chromosomes[row][subchr].Substring(AA, l - AA);
                        string deel1Chromosome2 = chromosomes[row + 1][subchr].Substring(0, AA);
                        string deel2Chromosome2 = chromosomes[row + 1][subchr].Substring(AA, l - AA);

                        chromosomes[row][subchr] = deel1Chromosome1 + deel2Chromosome2;
                        chromosomes[row + 1][subchr] = deel1Chromosome2 + deel2Chromosome1;
                    }
                    if (subchr > R1)
                    {
                        string tempStr = chromosomes[row][subchr];

                        //just switch
                        Array.Copy(chromosomes[row+1],subchr,chromosomes[row],subchr,1);
                        chromosomes[row + 1][subchr] = tempStr;
                    }
                }


            }//end when crossover should be carried out
        }//end crossover

        //public void flip(string[][] chromosomes, int row, double pm)
        //{
        //    double R0 = Random.NextDouble();
        //    if (R0 <= pm)
        //    {
        //        // Select subchromosome that will be mutate by chance
        //        int R1 = Random.Next(0, chromosomes[row].GetLength(0));
        //        // The length of the subchromosome
        //        int length = chromosomes[row][R1].Length;
        //        // the gene that will be mutated
        //        int R2 = Random.Next(0, length - 1);


        //        //taking the sub chromosome that was selected
        //        string subChrTemp = String.Copy(chromosomes[row][R1]);
        //        //split in parts
        //        string subChrB = subChrTemp.Substring(0, R2); //begin
        //        string subChrM1 = subChrTemp.Substring(R2, 1); //to be flipped
        //        string subChrM2 = subChrTemp.Substring(R2 + 1, 1); //to be flipped
        //        string subChrE = subChrTemp.Substring(R2 + 2, (length - R2 - 2)); //end
        //        //flip according Katsifarakis
        //        if (subChrM1 == "0")
        //        {
        //            subChrM1 = "1";
        //            subChrM2 = "0";
        //        }
        //        else
        //        {
        //            subChrM1 = "0";
        //            subChrM2 = "1";
        //        }
        //        //past back together
        //        subChrTemp = subChrB + subChrM1 + subChrM2 + subChrE;

        //        //store
        //        chromosomes[row][R1] = String.Copy(subChrTemp);
        //    }
        //}//end flip


        public void flip(string[][] chromosomes, int row, double pm)
        {
            for (int subchromosome = 0; subchromosome < chromosomes[0].GetLength(0); subchromosome++)
            {
                //calculate the length
                int l = chromosomes[row][subchromosome].Length;
                int[] chromosome_in_pieces = new int[l];

                //cut the string into peaces and convert it to 10-int
                for (int i = 0; i < l; i++)
                {
                    chromosome_in_pieces[i] = Convert.ToInt32(chromosomes[row][subchromosome].Substring ↙
    (i, 1), 10);
                }

                //calculate random between 0 and 1

                for (int i = 0; i < l-1; i++)
                {

                    double R = Random.NextDouble();
                    if (R <= pm)
                    {
                        if (chromosome_in_pieces[i] == 0)
                        {
                            chromosome_in_pieces[i] = 1;
                            chromosome_in_pieces[i+1] = 0;
                        }
                        else //set it to be zero
                        {
                            chromosome_in_pieces[i] = 0;
                            chromosome_in_pieces[i + 1] = 1;
                        }
                    }//end when mutation should be carried out
                } //end for loop


                //make string from all arrayvalues

                string resultaat = "";

                for (int i = 0; i < l; i++)
                {
                    resultaat = resultaat + chromosome_in_pieces[i].ToString();
                }

                chromosomes[row][subchromosome] = resultaat;
            }
        }//end flip


        //public void mutation(string[][] chromosomes, int row, double pm)
        //{
        //    double R0 = Random.NextDouble();
        //    if (R0 <= pm)
        //    {
        //        // Select subchromosome that will be mutate by chance
        //        int R1 = Random.Next(0, chromosomes[0].GetLength(0));
        //        // The length of the subchromosome
        //        int length = chromosomes[0][R1].Length;
        //        // the gene that will be mutated
        //        int R2 = Random.Next(0, length);

        //        //taking the sub chromosome that was selected
        //        string subChrTemp = String.Copy(chromosomes[row][R1]);
        //        //split in parts
        //        string subChrB = subChrTemp.Substring(0, R2); //begin
        //        string subChrM = subChrTemp.Substring(R2, 1); //to be mutated
        //        string subChrE = subChrTemp.Substring(R2 + 1, (length - R2 - 1)); //end
        //        //mutate
        //        if (subChrM == "1")
        //        {
        //            subChrM = "0";
        //        }
        //        else
        //        {
        //            subChrM = "1";
        //        }
        //        //past back together
        //        subChrTemp = subChrB + subChrM + subChrE;

        //        //store
        //        chromosomes[row][R1] = String.Copy(subChrTemp);

        //    }//end if R0 < Pm
        //}//end mutation

        public void mutation(string[][] chromosomes, int row, double pm)
        {
            for (int subchromosome = 0; subchromosome < chromosomes[0].GetLength(0); subchromosome++)
            {
                //calculate the length
                int l = chromosomes[row][subchromosome].Length;
                int[] chromosome_in_pieces = new int[l];

                //cut the string into peaces and convert it to 10-int
                for (int i = 0; i < l; i++)
                {
                    chromosome_in_pieces[i] = Convert.ToInt32(chromosomes[row][subchromosome].Substring ↙
    (i, 1), 10);
                }

                //calculate random between 0 and 1

                for (int i = 0; i < l; i++)
                {

                    double R = Random.NextDouble();
                    if (R <= pm)
                    {
                        if (chromosome_in_pieces[i] == 0)
                        {
                            chromosome_in_pieces[i] = 1;
                        }
                        else //set it to be zero
                        {
                            chromosome_in_pieces[i] = 0;
                        }
                    }//end when mutation should be carried out
                } //end for loop


                //make string from all arrayvalues

                string resultaat = "";

                for (int i = 0; i < l; i++)
                {
                    resultaat = resultaat + chromosome_in_pieces[i].ToString();
                }

                chromosomes[row][subchromosome] = resultaat;
            }
        }//end mutation

        public void calculateOfflinePerformance(double[] offlinefitness, int run, double[] maxfitness)
        {
            offlinefitness[run] = 0;
            for (int i = 0; i < run + 1; i++)
            {
                offlinefitness[run] = offlinefitness[run] + maxfitness[i];
            }
            offlinefitness[run] = offlinefitness[run] / (run + 1);
        }//end calculateOfflinePerformance

        public void calculateOnlinePerformance(double[] onlinefitness, int run, double[] avefitness)
        {
            onlinefitness[run] = 0;
            for (int i = 0; i < run + 1; i++)
            {
                onlinefitness[run] = onlinefitness[run] + avefitness[i];
            }
            onlinefitness[run] = onlinefitness[run] / (run + 1);
        }//end calculateOnlinePerformance

        public void sortJarredArray(double[][] array)
        {

            double[] tempArray0 = new double[array.GetLength(0)]; //stores the linenumber
            double[] tempArray1 = new double[array.GetLength(0)]; //stores the distance line to well
            double[] tempArray1Sorted = new double[array.GetLength(0)]; //stores the distance line to well, this array will be sorted
            double[][] sortedArray = new double[array.GetLength(0)][];

            //1. save all double values in a 1 dimensional array
            for (int i = 0; i < array.GetLength(0); i++)
            {
                tempArray0[i] = array[i][0];
                tempArray1[i] = array[i][1];
                tempArray1Sorted[i] = array[i][1];
            }

            //2. Sort the tempArray[]
            Array.Sort(tempArray1Sorted);

            //3. Find the original index in array[]
            for (int i = 0; i < array.GetLength(0); i++)
            {
                sortedArray[i] = new double[2];
                sortedArray[i][0] = array[Array.LastIndexOf(tempArray1, tempArray1Sorted[i])][0];
                sortedArray[i][1] = tempArray1Sorted[i];
            }

            //4. Copy the values from sorted to the original array.
            for (int i = 0; i < array.GetLength(0); i++)
            {
                for (int j = 0; j < array[i].GetLength(0); j++)
                {
                    array[i][j] = sortedArray[i][j];
                }
            }
        }//end sortJarredArray

        public void InflowCharacteristics(int row, double[] L, double[] T, double[] Un, int[][] zone, bool[] lineOnCoast, ref double[] totalInflow, ref int[] totalInflowNodes)
        {
            double B = 0;
            int k = 0;

            for (int s = 0; s < lineOnCoast.GetLength(0); s++)
            {
                if (lineOnCoast[s] == true)
                {
                    if (Un[s] > 0)
                    {
                        if (zone[s][0] != -1 && zone[s][1] == -1)
                        {
                            B = B + Un[s] * L[s] * T[zone[s][0]];
                        }
                        else if (zone[s][1] != -1 && zone[s][0] == -1)
                        {
                            B = B + Un[s] * L[s] * T[zone[s][1]];
                        }
                        else
                        {
                            MessageBox.Show("Zone undifined");
                        }
                        k++;
                    }
                }
            }
            totalInflow[row] = B;
            totalInflowNodes[row] = k;
        }//end InflowCharacteristics

        public void trialreportxls(int ps, int numberofruns, double pc_begin, double pc_eind, double pm_begin, double pm_eind, double[] trialMaxFitness, double[][] trialWell, double[] trialConvergenceVelocity, double[] trialTotalInflow, double[] trialTotalNumberOflinesWithInflow, int[] trialBestGenFound, double[] trials, double[] triall, int CalculationsSaved, int NumberOfSubchromoses, int CalculationsSavedWell, int memoryFitness, int memoryWell, double[][] detailMaxFitness, double[][] detailMinFitness, double[][] detailAveFitness, int[][] detailCalculationSaved, int[][] detailCalculationSavedWell, double C1, double C2, double C3, double C4, bool fixed_spw_length, double spw_length)
        {

            //giving the name of the file

            dateTimeEnd = DateTime.Now;
            string time = dateTimeEnd.ToString("yyyy-MM-dd (HH-mm-ss)");
            string nameDoc = "report" + time + ".xls";

            //open the XLS

            Excel.Application xlApp = default(Excel.Application);
            Excel.Workbook xlWorkBook = default(Excel.Workbook);
            Excel.Worksheet xlWorkSheet = default(Excel.Worksheet);

            try
            {
                object misValue = System.Reflection.Missing.Value;

                xlApp = new Excel.Application();
                xlWorkBook = xlApp.Workbooks.Open(@"C:\Users\Koen Wildemeersch\Desktop\SjabloomThesis.
    xls", misValue, misValue, misValue, misValue, misValue, misValue, misValue, misValue, misValue, 
    misValue, misValue, misValue, misValue, misValue);
                xlWorkSheet = xlWorkBook.Worksheets.get_Item(1);

                //1. general
                xlWorkSheet.Cells[3, 3] = projectName;
                xlWorkSheet.Cells[4, 3] = author;

                //2. Calculation Duration

                //calculating the time it took
                TimeSpan ts = (dateTimeEnd - dateTimeBegin);
                string durationtime = new DateTime(ts.Ticks).ToString("HH:mm:ss");

                xlWorkSheet.Cells[7, 3] = dateTimeBegin.ToString("dd MMM yyyy - HH:mm:ss");
                xlWorkSheet.Cells[8, 3] = dateTimeEnd.ToString("dd MMM yyyy - HH:mm:ss");
                xlWorkSheet.Cells[9, 3] = durationtime;

                xlWorkSheet.Cells[12, 3] = ps;
                xlWorkSheet.Cells[13, 3] = numberofruns;
                xlWorkSheet.Cells[14, 3] = trialMaxFitness.GetLength(0);
                xlWorkSheet.Cells[15, 3] = elitism;

                //selection method
                if (selectionType == 0)
                {
                    xlWorkSheet.Cells[12, 6] = "Roulette wheel";
                    xlWorkSheet.Cells[13, 6] = "-";
                }
                if (selectionType == 1)
                {
                    xlWorkSheet.Cells[12, 6] = "Ranking";
                    xlWorkSheet.Cells[13, 6] = selectionConstant;
                }
                if (selectionType == 2)
                {
                    xlWorkSheet.Cells[12, 6] = "Selection constant";
                    xlWorkSheet.Cells[13, 6] = selectionConstant;
                }

                xlWorkSheet.Cells[14, 6] = pc_begin;
                xlWorkSheet.Cells[14, 9] = pc_eind;
                xlWorkSheet.Cells[15, 6] = pm_begin;
                xlWorkSheet.Cells[15, 9] = pm_eind;

                //3. fitness function

                xlWorkSheet.Cells[18, 3] = fitnessFunction;
                xlWorkSheet.Cells[19, 3] = C1;
                xlWorkSheet.Cells[20, 3] = C2;
                xlWorkSheet.Cells[19, 8] = C3;
                xlWorkSheet.Cells[20, 8] = C4;

                //4. sheet pile wall


                xlWorkSheet.Cells[23, 3] = spw;
                if (spw != false)
                {
                    xlWorkSheet.Cells[24, 3] = fixed_spw_length;
                    if (fixed_spw_length == true)
                    {
                        xlWorkSheet.Cells[25, 3] = spw_length;
                        xlWorkSheet.Cells[28, 3] = chr1_LengthSpw;
                        xlWorkSheet.Cells[29, 3] = "-";
                    }
                    else
                    {
                        xlWorkSheet.Cells[25, 3] = "Over entire coastline (between lower and upper 
    bound)";
                        xlWorkSheet.Cells[28, 3] = chr1_LengthSpw;
                        xlWorkSheet.Cells[29, 3] = chr2_LengthSpw;
                    }
                    if (spw_min <= 0)
                    {
                        xlWorkSheet.Cells[26, 3] = "0";
                    }
                    else
                    {
                        xlWorkSheet.Cells[26, 3] = spw_min;
                    }
                    if (spw_max <= 0)
                    {
                        xlWorkSheet.Cells[27, 3] = cumulLineEnd[cumulLineEnd.GetLength(0)-1];
                    }
                    else
                    {
                        xlWorkSheet.Cells[27, 3] = spw_max;
                    }

                }
                else
                {
                    xlWorkSheet.Cells[24, 3] = "-";
                    xlWorkSheet.Cells[25, 3] = "-";
                    xlWorkSheet.Cells[26, 3] = "-";
                    xlWorkSheet.Cells[27, 3] = "-";
                    xlWorkSheet.Cells[28, 3] = "-";
                    xlWorkSheet.Cells[29, 3] = "-";
                }

                //7. Statistics


                xlWorkSheet.Cells[56, 6] = trialMaxFitness.Min();
                xlWorkSheet.Cells[57, 6] = trialMaxFitness.Average();
                xlWorkSheet.Cells[58, 6] = StandardDeviation(trialMaxFitness);
                xlWorkSheet.Cells[59, 6] = trialBestGenFound.Max();

                int numberOfCalculations = ps * numberofruns * trialMaxFitness.GetLength(0);

                xlWorkSheet.Cells[60, 6] = CalculationsSaved;
                xlWorkSheet.Cells[60, 7] = "/";
                xlWorkSheet.Cells[60, 8] = numberOfCalculations;
                xlWorkSheet.Cells[61, 6] = memoryFitness;

                xlWorkSheet.Cells[62, 6] = CalculationsSavedWell;
                xlWorkSheet.Cells[62, 7] = "/";
                xlWorkSheet.Cells[62, 8] = ((numberOfCalculations * well.GetLength(0)) -      ↙
  CalculationsSaved * well.GetLength(0));
                xlWorkSheet.Cells[63, 6] = memoryWell;

                //6. best result
                //find out where is the best solution?
                int IndexBext = Array.IndexOf(trialMaxFitness, trialMaxFitness.Max());

                if (spw == true)
                {
                    xlWorkSheet.Cells[46, 3] = trials[IndexBext];
                    xlWorkSheet.Cells[47, 3] = trials[IndexBext] + triall[IndexBext];
                    xlWorkSheet.Cells[48, 3] = triall[IndexBext];
                }
                else
                {
                    xlWorkSheet.Cells[46, 3] = "-";
                    xlWorkSheet.Cells[47, 3] = "-";
                    xlWorkSheet.Cells[48, 3] = "-";
                }
                xlWorkSheet.Cells[44, 3] = IndexBext;
                xlWorkSheet.Cells[49, 3] = trialMaxFitness[IndexBext];
                xlWorkSheet.Cells[50, 3] = trialTotalInflow[IndexBext];
                xlWorkSheet.Cells[51, 3] = trialTotalNumberOflinesWithInflow[IndexBext];
                xlWorkSheet.Cells[52, 3] = trialBestGenFound[IndexBext];
                xlWorkSheet.Cells[53, 3] = trialConvergenceVelocity[IndexBext];

                xlWorkSheet.Cells[44, 4] = 0;
                xlWorkSheet.Cells[44, 5] = trialWell[IndexBext * well.GetLength(0)][0];
                xlWorkSheet.Cells[44, 6] = trialWell[IndexBext * well.GetLength(0)][1];
```

```
            xlWorkSheet.Cells[44, 7] = trialWell[IndexBext * well.GetLength(0)][2];


            int r = 44;
            //if the number of wells is different from 0, extra lines need to be writen for them
            if (well.GetLength(0) > 1)
            {

                for (int w = 1; w < well.GetLength(0); w++)
                {
                    //insert a new row
                    //xlWorkSheet.Rows.Insert(Microsoft.Office.Interop.Excel.XlDirection.xlDown, r+ 2);
                    r++;

                    //write the row
                    xlWorkSheet.Cells[r, 4] = w;
                    xlWorkSheet.Cells[r, 5] = trialWell[IndexBext * well.GetLength(0) + w][0];
                    xlWorkSheet.Cells[r, 6] = trialWell[IndexBext * well.GetLength(0) + w][1];
                    xlWorkSheet.Cells[r, 7] = trialWell[IndexBext * well.GetLength(0) + w][2];

                }
            }

            //5. writing the wells.

            //counter for dmin and dmax
            int dd = 0;
            r = 37;
            for (int i = 0; i < well.GetLength(0); i++)
            {
                xlWorkSheet.Cells[r, 2] = i;

                for (int j = 0; j < 3; j++)
                {
                    if (hwell[i][j] == false)
                    {
                        xlWorkSheet.Cells[r, 3 + j * 2] = well[i][j];
                        xlWorkSheet.Cells[r, 3 + j * 2 + 1] = well[i][j];
                    }
                    else
                    {
                        xlWorkSheet.Cells[r, 3 + j * 2] = dmin[dd];
                        xlWorkSheet.Cells[r, 3 + j * 2 + 1 ] = dmax[dd];
                        dd++;
                    }
                }
                xlWorkSheet.Cells[r, 9] = chrLengthWell[i];
                r++; //so we know what is the next line to write
            }

            //2. Write all results
            xlWorkSheet = (Excel.Worksheet)xlWorkBook.Worksheets.get_Item(2);

            int row = 3;

            for (int trial = 0; trial < trialMaxFitness.GetLength(0); trial++)
            {
                xlWorkSheet.Cells[row, 1] = trial;
                xlWorkSheet.Cells[row, 2] = trialMaxFitness[trial];
                xlWorkSheet.Cells[row, 3] = 0;
                xlWorkSheet.Cells[row, 4] = trialWell[trial * well.GetLength(0)][0];
                xlWorkSheet.Cells[row, 5] = trialWell[trial * well.GetLength(0)][1];
                xlWorkSheet.Cells[row, 6] = trialWell[trial * well.GetLength(0)][2];
                xlWorkSheet.Cells[row, 7] = trialConvergenceVelocity[trial];
                xlWorkSheet.Cells[row, 8] = trialTotalInflow[trial];
                xlWorkSheet.Cells[row, 9] = trialTotalNumberOflinesWithInflow[trial];
                xlWorkSheet.Cells[row, 10] = trialBestGenFound[trial];
                if (spw == true)
                {
                    xlWorkSheet.Cells[row, 11] = trials[trial];
                    xlWorkSheet.Cells[row, 12] = trials[trial] + triall[trial];
                    xlWorkSheet.Cells[row, 13] = triall[trial];
                }
```

```

                    //if the number of wells is different from 0, extra lines need to be writen for ↙
    them
                    if (well.GetLength(0) > 1)
                    {
                        for (int w = 1; w < well.GetLength(0); w++)
                        {
                            row++;
                            xlWorkSheet.Cells[row, 3] = w;
                            xlWorkSheet.Cells[row, 4] = trialWell[trial * well.GetLength(0) + w][0];
                            xlWorkSheet.Cells[row, 5] = trialWell[trial * well.GetLength(0) + w][1];
                            xlWorkSheet.Cells[row, 6] = trialWell[trial * well.GetLength(0) + w][2];
                        }
                    }
                    row++;

                }//end every trial to write report

                //3. Well Calculations Saved
                xlWorkSheet = (Excel.Worksheet)xlWorkBook.Worksheets.get_Item(3);
                for (int i = 0; i < detailCalculationSavedWell[0].GetLength(0); i++)
                {
                    xlWorkSheet.Cells[1, i + 2] = i;
                }
                row = 2;
                for (int i = 0; i < detailCalculationSavedWell.GetLength(0); i++)
                {
                    xlWorkSheet.Cells[row, 1] = i;
                    for (int j = 0; j < detailCalculationSavedWell[0].GetLength(0); j++)
                    {
                        xlWorkSheet.Cells[row, j + 2] = detailCalculationSavedWell[i][j];
                    }
                    row++;
                }

                //4. Calculations Saved
                xlWorkSheet = (Excel.Worksheet)xlWorkBook.Worksheets.get_Item(4);
                for (int i = 0; i < detailCalculationSaved[0].GetLength(0); i++)
                {
                    xlWorkSheet.Cells[1, i + 2] = i;
                }
                row = 2;
                for (int i = 0; i < detailCalculationSaved.GetLength(0); i++)
                {
                    xlWorkSheet.Cells[row, 1] = i;
                    for (int j = 0; j < detailCalculationSaved[0].GetLength(0); j++)
                    {
                        xlWorkSheet.Cells[row, j + 2] = detailCalculationSaved[i][j];
                    }
                    row++;
                }


                //5. Detail min Fitness
                xlWorkSheet = (Excel.Worksheet)xlWorkBook.Worksheets.get_Item(5);
                for (int i = 0; i < detailMinFitness[0].GetLength(0); i++)
                {
                    xlWorkSheet.Cells[1, i + 2] = i;
                }
                row = 2;
                for (int i = 0; i < detailMinFitness.GetLength(0); i++)
                {
                    xlWorkSheet.Cells[row, 1] = i;
                    for (int j = 0; j < detailMinFitness[0].GetLength(0); j++)
                    {
                        xlWorkSheet.Cells[row, j + 2] = detailMinFitness[i][j];
                    }
                    row++;
                }


                //6. Detail max fitness
                xlWorkSheet = (Excel.Worksheet)xlWorkBook.Worksheets.get_Item(6);
                for (int i = 0; i < detailAveFitness[0].GetLength(0); i++)
                {
                    xlWorkSheet.Cells[1, i + 2] = i;
                }
                row = 2;
                for (int i = 0; i < detailAveFitness.GetLength(0); i++)
                {
                    xlWorkSheet.Cells[row, 1] = i;
                    for (int j = 0; j < detailAveFitness[0].GetLength(0); j++)
                    {
                        xlWorkSheet.Cells[row, j + 2] = detailAveFitness[i][j];
                    }
                    row++;
                }


                //7. Detail max fitness
                xlWorkSheet = (Excel.Worksheet)xlWorkBook.Worksheets.get_Item(7);
                for (int i = 0; i < detailMaxFitness[0].GetLength(0); i++)
                {
                    xlWorkSheet.Cells[1, i + 2] = i;
                }
                row = 2;
                for (int i = 0; i < detailMaxFitness.GetLength(0); i++)
                {
                    xlWorkSheet.Cells[row, 1] = i;
                    for (int j = 0; j < detailMaxFitness[0].GetLength(0); j++)
                    {
                        xlWorkSheet.Cells[row, j + 2] = detailMaxFitness[i][j];
                    }
                    row++;
                }



                xlWorkBook.SaveAs(nameDoc, Excel.XlFileFormat.xlWorkbookNormal, misValue, misValue,  ↙
    misValue, misValue, Excel.XlSaveAsAccessMode.xlExclusive, misValue, misValue, misValue, misValue,  ↙
    misValue);
                xlWorkBook.Close(true, misValue, misValue);
                xlApp.Quit();

                releaseObject(xlWorkSheet);
                releaseObject(xlWorkBook);
                releaseObject(xlApp);
            }
            finally
            {
                if (xlApp != null)
                    releaseObject(xlApp);
                if (xlWorkBook != null)
                    releaseObject(xlWorkBook);
                if (xlWorkSheet != null)
                    releaseObject(xlWorkSheet);
            }

            if (System.IO.File.Exists(nameDoc))
            {
                if (MessageBox.Show("Would you like to open the excel file?", this.Text,  ↙
    MessageBoxButtons.YesNo, MessageBoxIcon.Question) == DialogResult.Yes)
                {
                    try
                    {
                        System.Diagnostics.Process.Start(nameDoc);
                    }
                    catch (Exception ex)
                    {
                        MessageBox.Show("Error opening the excel file." + Environment.NewLine +
                          ex.Message, this.Text, MessageBoxButtons.OK, MessageBoxIcon.Error);
                    }
                }
            }

        }//end function write trialreportxls

        private void releaseObject(object obj)
        {
            if (obj == null)
                throw new ArgumentNullException("obj");
            try
            {
                System.Runtime.InteropServices.Marshal.ReleaseComObject(obj);
            }
            catch { }
        }



        //statics

        static int totalNumberOfUnknown(int[][] zone)
        {
            /* First of all the total number of unknown should be calculated:
            * for all nodes there is an eqation, and for the nodes on the interface
            * there is an extra. The number of unknown is thus the dimension of XM +
            * the number of arrays zone where zone[I][1] != -1
            */
            int number = zone.GetLength(0); //one equation per node in any case

            for (int i = 0; i < zone.GetLength(0); i++)
            {
                if (zone[i][1] != -1)
                { //if it is different from -1 it means it is on the interface so an extra eq is needed
                    number++;
                } //end if
            }//end for i
            return number;
        }//end totalNumberOfUnknown

        static int numberOfCoastalElements(bool[] ulineOnCoast)
        {
            int number = 0;
            for (int i = 0; i < ulineOnCoast.GetLength(0); i++)
            {
                if (ulineOnCoast[i] == true)
                {
                    number++;
                }
            }
            return number++;
        }//end numberOfCoastalElements

        static double Gon(double x0, double x1, double x2, double y0, double y1, double y2, double lj)
        {

            //values of /xi (k) and w (k) (for 4 (k=0,1,2 or 3) point Gauss integration)
            double[] xi = new double[4] { -0.861136311594053, -0.339981043584856, 0.339981043584856, 0.
    861136311594053 };
            double[] w = new double[4] { 0.347854845137454, 0.652145154862546, 0.652145154862546, 0.
    347854845137454 };
            double x_xi;    //X coordinate as function of xi
            double y_xi;    //Y coordinate as function of xi
            double r_xi;    //r
            double sum = 0;     // sum necessary for calculating G

            //calculate the summation

            for (int k = 0; k < 4; k++)
            {
                x_xi = (x2 + x1) / 2 + (x2 - x1) / 2 * xi[k];
                y_xi = (y2 + y1) / 2 + (y2 - y1) / 2 * xi[k];
                r_xi = Math.Sqrt(Math.Pow((x_xi - x0), 2) + Math.Pow((y_xi - y0), 2));
                sum = sum + Math.Log(r_xi) * w[k];
            }
            return lj / (4 * Math.PI) * sum; //G is calculated correctly
        }//end Gon

        static double Hon(double x0, double x1, double x2, double y0, double y1, double y2)
        {
            double DY1 = y1 - y0;
            double DX1 = x1 - x0;
            double DY2 = y2 - y0;
            double DX2 = x2 - x0;
            double DL1 = Math.Sqrt(DX1 * DX1 + DY1 * DY1);
            double cos1 = DX1 / DL1;
            double sin1 = DY1 / DL1;
            double DX2R = DX2 * cos1 + DY2 * sin1;
            double DY2R = -DX2 * sin1 + DY2 * cos1;
            return (Math.Atan2(DY2R, DX2R) / (2 * Math.PI));
        }//end Hon

        static double doubleChromosome(string chromosome, double dmin, double dmax, int lengthchromosome)
        {
            double I32; //for very high exponents C# makes mistakes with int, therefore use double
            double dchromosome;

            //calculate the length
            int l = chromosome.Length;
            int[] chromosome_in_pieces = new int[l];

            //cut the string into peaces and convert it to 10-int
            for (int i = 0; i < l; i++)
            {
                chromosome_in_pieces[i] = Convert.ToInt32(chromosome.Substring(i, 1), 10);
            }

            //now go through the chromosome and calculate the int value

            I32 = 0;
            for (int i = 0; i < l - 1; i++)
            {
                I32 = I32 + Math.Pow(2 * chromosome_in_pieces[i], (l - 1 - i));
            }

            //for the last bit
            I32 = I32 + chromosome_in_pieces[l - 1];

            //from the int calculate the double

            dchromosome = (dmax - dmin) / (Math.Pow(2, l) - 1) * I32 + dmin;

            return dchromosome;
        }//end doubleChromosome

        static int numberOfLinesAffected(int[] lineorder, int lineBegin, int lineEnd)
        {
            int numberOfLinesAffected = 0;
            int t = Array.IndexOf(lineorder, lineBegin);
            bool onSWP = new bool();
            onSWP = true;
            while (onSWP == true)
            {
                if (lineorder[t] == lineEnd)
                {
                    numberOfLinesAffected++;
                    onSWP = false;
                }
                else
                {
                    numberOfLinesAffected++;
                }
                t++; //go to next line
            }
            return numberOfLinesAffected;
        }//end numberOfLinesAffected

        static bool extraLineForBeginSpw(double[] cumulLineEnd, double beginSpw, int lineBegin, int[] lineorder)
        {
            bool extraForBeginSpw = new bool();
            extraForBeginSpw = false;
            //when begin is not on the end/begin point of the original line a subdivision is to be made

            if (Array.IndexOf(lineorder, lineBegin) == 0)
            {
                if (beginSpw != 0)
                {//the statistic posibility that the SPW starts in the beginning of the coastline
                    extraForBeginSpw = true;
                }
            }
            else
            {
                if (beginSpw != cumulLineEnd[Array.IndexOf(lineorder, lineBegin) - 1])
                {
                    extraForBeginSpw = true;
                }
            }
            return extraForBeginSpw;
        }//end extraLineForBeginSpw

        static bool extraLineForEndSpw(double[] cumulLineEnd, double endSpw, int lineEnd, int[] lineorder)
        {
            bool extraforEndSpw = new bool();
            extraforEndSpw = false;
            if (endSpw != cumulLineEnd[Array.IndexOf(lineorder, lineEnd)])
            {
                extraforEndSpw = true;
            }
            return extraforEndSpw;
        }//end extraLineForEndSpw;

        static int SelectByRoulettewheel(double[] fitness)
        {
            //1. find the minimum value of the fitnessfunction
            double minFitness = fitness.Min();
            double maxFitness = fitness.Max();
            int NumOfMin = 0;
            int NumOfMax = 0;
            bool areAllAsFit = new bool();
            areAllAsFit = true;
            double[] probability = new double[fitness.GetLength(0)];
            double pmin;

            //2. Calculate the probability that will be given to that minimum fitness

            //2.1 Find out if all chromosomes are as fit
            if (minFitness != maxFitness)
            {
                areAllAsFit = false;
            }

            if (areAllAsFit == true)
            {
                //same probability

                pmin = 1 / Convert.ToDouble(fitness.GetLength(0));
                for (int c = 0; c < fitness.GetLength(0); c++)
                {
                    probability[c] = pmin;
                }
            }
            else
            {
                pmin = 1 / Math.Pow(fitness.GetLength(0), 2);

                //3. Calculate a the ratio of the lowest and hightest probability
                //3.1. Calculate factor
                double dmax = Math.Abs(minFitness - maxFitness);
                double suml = 0;
                double factor = 0;
                for (int c = 0; c < fitness.GetLength(0); c++)
                {
                    if (fitness[c] == minFitness)
                    {
                        NumOfMin++;
                    }
                    else if (fitness[c] == maxFitness)
                    {
                        NumOfMax++;
                    }
                    else
                    {
                        suml = suml + Math.Abs((fitness[c] - minFitness) / (dmax));
                    }
                }
                factor = (1 / pmin - NumOfMin - (fitness.GetLength(0) - (NumOfMin + NumOfMax)) + suml) ↙
  / (NumOfMax + suml);

                for (int c = 0; c < fitness.GetLength(0); c++)
                {
                    if (fitness[c] == minFitness)
                    {
                        probability[c] = pmin;
                    }
                    else if (fitness[c] == maxFitness)
                    {
                        probability[c] = pmin * factor;
                    }
                    else
                    {
                        probability[c] = pmin + (factor - 1) * pmin * Math.Abs((fitness[c] -         ↙
  minFitness) / (dmax));
                    }
                }
            }//end if not as fit

            double sumProb = probability.Sum();
            if (sumProb < 0.95 || sumProb > 1.05)
            {
                MessageBox.Show("Error During probability calculation! ( " + sumProb + " )");
            }
            //4. With there probabilities used RouletteWheel and select one chromosome
            //select a chromosome via roulette wheel selection
            double tempMaximum = 0;
            int selectedchromosome = 0;

            //calculate random between 0 and 1
            double R = Random.NextDouble();

            for (int i = 0; i < fitness.GetLength(0); i++)
            {
                tempMaximum = tempMaximum + probability[i];
                if (tempMaximum > R)
                {
                    selectedchromosome = i; //this is the index of the selected element
                    i = fitness.GetLength(0); //stop the loop
                }
            }

            //5. Return this chromosome
            return selectedchromosome;




        }//end selectByRoulettewheel

        static int SelectByConstantSelection(double[] fitness, int KK)
        {
            int[] KKChromosome = new int[KK];
            double[] KKfitness = new double[KK];

            //1. Select KK chromosomes
            for (int k = 0; k < KK; k++)
            {
                int R = Random.Next(0, fitness.GetLength(0));
                KKChromosome[k] = R;
                Array.Copy(fitness, R, KKfitness, k, 1);
            }

            //2. find the maxima fitness
            int IndexMaxFitness = Array.IndexOf(KKfitness, KKfitness.Max());
            int IndexSelectedChromosome = KKChromosome[IndexMaxFitness];

            //3. return the index of the selected chromosome
            return IndexSelectedChromosome;
        }//end SelectByConstantSelection

        static double Pc(int run, int ps, double pc_begin, double pc_eind)
        {
            return pc_begin - ((pc_begin - pc_eind) / ps) * run;
        }//end Pc

        static double Pm(int run, int ps, double pm_begin, double pm_eind)
        {
            return pm_begin - ((pm_begin - pm_eind) / ps) * run;
        }//end Pm

        static double calculateConvergenceVelocity(double[] maxfitness)
        {
            double B = 1;
            double diff = B - maxfitness[0];
            double A = maxfitness[maxfitness.GetLength(0) - 1] + diff;
            return Math.Log(Math.Sqrt(A / B));
            //return Math.Log(Math.Sqrt(maxfitness[maxfitness.GetLength(0) - 1] / maxfitness[0]));
        }//end calculateConvergenceVelocity

        static double Dsx(double[][] uline, double[] uL, double[] cumulLineEnd, int lineNumber, double ↙
    S, int[] lineorder)
        {//calculates delta s accordint the x-as
            double Dsx = 0;
            double ls = uL[lineNumber] - (cumulLineEnd[Array.IndexOf(lineorder, lineNumber)] - S);
            Dsx = ls * (uline[lineNumber][2] - uline[lineNumber][0]) / uL[lineNumber];
            return Dsx;
        }//end Dsx

        static double Dsy(double[][] uline, double[] uL, double[] cumulLineEnd, int lineNumber, double ↙
    S, int[] lineorder)
        {//calculates delta s accordint the x-as
            double Dsy = 0;
            double ls = uL[lineNumber] - (cumulLineEnd[Array.IndexOf(lineorder, lineNumber)] - S);
            Dsy = ls * (uline[lineNumber][3] - uline[lineNumber][1]) / uL[lineNumber];
            return Dsy;
        }//end Dsy

        static double StandardDeviation(double[] trialMaxFitness)
        {
            double SumOfSqrs = 0;
            double average = trialMaxFitness.Average();
            for (int i = 0; i < trialMaxFitness.GetLength(0); i++)
            {
                SumOfSqrs += Math.Pow((trialMaxFitness[i] - average), 2);
            }
            return Math.Sqrt(SumOfSqrs / (trialMaxFitness.GetLength(0) - 1));
        }//end StandardDevition
    }
}
```

Bibliography

[1] J.T. Katsikadelis. *Boundary elements. theorie and applications.* Elsevier, 2002.

[2] B. Verhegghe. *Elementenmethode in de toegepaste mechanica.* Universiteit Gent, 2008.

[3] H. Peiffer. *Grondwater en contaminenten stroming.* Universiteit Gent.

[4] C.A. Brebbia and J. Dominguez. *Boundary elements an introductory course.* Witpress, 1998.

[5] G. Beer. *Programming the boundary element method. An introduction for engineers.* Wiley, 2001.

[6] F. Paris and J. Canas. *Boundary element method. Fundamentals and applications.* Oxford University Press, 1997.

[7] J.C.F. Telles C.A. Brebbia and L.C. Wrobel. *Boundary element techniques.* Springer-Verlag, 1984.

[8] K.L. Katsifarakis and Z. Petala. Combining genetic algorithms and boundary elements to optimize coastal aquifers' management. In *Journal of Hydrology*, pages 200–207. Elsevier, 2006.

[9] N. Theodosiou K.L. Katsifarakis, D.K. Karpouzos. Combined use of bem and genetic algorithms in groundwater flow and mass transport problems. In *Engineering Analysis with boundary elements*, pages 555–565. Elsevier, 1999.

[10] Qbasic tutorial. `http://westcompsci.pima.edu/cis100`.

[11] The qbasis station. `http://www.qbasicstation.com/`.

[12] K. Wildemeersch. Grondwaterstandvariaties in zeedijken ten gevolge van de getijdenwerking. Master's thesis, KHBO, 2008.

[13] C. A. Brebbia. *The boundary element method for engineers.* Pentech press, 1978.

[14] M. Mitchell. *An introduction to genetic algorithms.* MIT Press, 1998.

[15] Universität Stuttgart. Boundary element methods. `http://www.iam.uni-stuttgart.de/bem/`, 2004.

[16] Food and agriculture organisation of the United Nations. *Seawater intrusion in coastal aquifers.* FAO, 1997.

[17] Coley D. A. *An introduction to genetic algorithms for scientists and engineers.* World Scientific, 2005.

[18] Centre for Civil Engineering Research and Codes. *Backgrounds of numerical modelling of geotechnical constructions, part 3.* CUR, 2000.

[19] D. Ouazar A.H.-D. Cheng. Groundwater optimization and parameter estimation by genetic algorithms and dual reciprocity boundary element method. In *Engineering Analysis with boundary elements*, pages 287–296. Elsevier, 1997.

[20] et al. K. El Harrouni, D. Ouazar. Groundwater. In *Boundary Element Techniques in Geomechanics*, pages 243–294. Elsevier, 1993.

[21] P. Tolikas E. Sidiropoulos. Genetic alorithms and cellular automata in aquifer management. In *Applied Mathematical modelling 32*, pages 617–640. Elsevier, 2008.

[22] Wikipedia. Boundary element methods. `http://en.wikipedia.org/wiki/Boundary_element_method`, 2010.

[23] K.L. Katsifarakis and D.K. Karpouzos. Minimization of pumping cost in zoned aquifers by means of genetic algorithms. In *Proceedings of the international conference on protection and restoration of the environment IV*, pages 61–68, 1998.

[24] Z. Petala. *Optimizing management of coastal aquifers by means of genetic algorithms (in Greek).* PhD thesis, Department of Civil Engineering, Aristotle University of Thessaloniki, Greece, 2004.

List of Figures

List of Tables

This page intentionally left blank