Optimaliseren van positiemetingen op basis van randvoorwaarden

Floris De Smedt

4 mei 2009

Voorwoord

In 2004 begon ik aan een professionele bachelor opleiding electonica-ICT aan het ”Hoger Instituut der Kempen” in Geel. Ik eindigde mijn bachelor opleiding in 2007 met een eindwerk over de ontwikkeling van een honeypot.

Ik wou mijn studie echter vervolledigen met een masterjaar. Hiervoor begon ik in 2007 met het voorbereidende schakeljaar. Ik koos ervoor om mijn opleiding voort te zetten aan het ”De Nayer Instituut” in Sint-Katelijne-Waver. Het lessenpakket bevatte onder andere het vak ’programmeertechnieken’ waarin de programmeertaal C aan bod kwam.

In tegenstelling tot mijn vorig eindwerk koos ik voor mijn masterproef een onderwerp waarbij programmeren centraal stond. Essensium, waarbij ik deze masterproef heb gedaan, is bezig met een systeem voor positiebepaling. Aan de hand van afstandsmetingen vanuit basisstations (devices met een gekende positie die niet wijzigt tijdens de normale werking van het systeem) kan de positie van mobiele nodes worden bepaald. Door deze mobiele nodes op palletten, personen, wagens, ... te bevestigen kan ook hiervan de positie worden vastgelegd. Om de installatietijd zo kort mogelijk te houden, kunnen we het algoritme dat gebruikt wordt om de posities van de mobiele nodes te bepalen, ook gebruiken om de positie van de basisstations te bepalen. Het probleem hierbij is dat de positie niet nauwkeurig genoeg bepaald wordt. Mijn taak bestond uit het toevoegen van randvoorwaarden aan dit algoritme, zodat de positiebepaling nauwkeuriger wordt.

In tegenstelling tot het bachelor-eindwerk waar gewerkt kan worden met ”trial and error”, was het hier nodig om alle keuzes te onderbouwen. Zo was het nodig om wiskunde te combineren met het programmeren in C++. Ik heb me kunnen inwerken in de wiskundige ideeën die achter het bestaande algoritme zaten en was in staat om hier de nodige toevoegingen aan te doen.

Tot slot wil ik iedereen bedanken die me heeft geholpen met de verwezelijking van dit eindwerk.

In de eerste plaats mijn bedrijfspromotoren Huub Tubbax en Jan Olbrechts die me hebben ondersteund bij zowel de praktische implementatie van mijn masterproef, als het opstellen van de scriptie.

Ook wil ik mijn hogeschoolpromotor Ann Philips bedanken voor het nalezen en opvolgen van mijn scriptie. Hierbij werd ik op de nodige fouten en onduidelijkheden gewezen.

Ik bedank ook mijn ouders die me de mogelijkheid hebben gegeven te studeren. Zij hebben me ook geholpen met de spellingscontrole bij het nalezen van mijn scriptie.

Abstract

De masterproef gaat over de implementatie van het gebruik van randvoorwaarden bij een Real Time Location System.

We zijn vertrokken van een bestaand algoritme dat aan de hand van afstandsmetingen de posities van mobiele nodes kan bepalen. Door deze mobiele nodes aan paletten, personen, ... te bevestigen, kan ook hiervan de positie bepaald worden.

Om de installatietijd zo kort mogelijk te houden, kunnen we hetzelfde algoritme inzetten. Hierbij beschouwen we de basisstations tijdelijk als mobiele nodes.

Door randvoorwaarden toe te voegen aan het algoritme verbeteren we de nauwkeurigheid waarmee de positiebepaling gebeurt. Hierdoor zal ook tijdens de normale werking van het systeem, de positiebepaling van de mobiele nodes nauwkeuriger worden.

Abstract

This master thesis decribes the implementation of constraints in a Real Time Location System.

We started from an existing algorithm that uses distance measurements (from base-stations and/or other mobile devices) to determine the positions of mobile nodes (attached to the objects to localise). To reduce the setup-time, we can use the same algorithm to determine the positions of the base-stations. The base-stations temporarily act as mobile nodes during setup.

By adding constraints to the algorithm, we can improve the precision of the position calculation. This improves the precision of the position calculation during normal running of the system.

Short Summary

Introduction

Essensium, a company that is specialised in ASIC solutions for wireless sensor networks, works on an RTLS system. This system uses distance measurements between nodes (base-stations, which position is fixed, and/or mobile nodes which position can change) to determine the positions of the mobile nodes.

The precision of the position determination depends on the precision of the positions of the base-stations. To determine the positions of the base-stations (at setup), we can measure them manually. This procedure will be very time-intensive in a setup with a lot of base-stations.

Instead of measuring the positions, we can use the algorithm used for the determination of the mobile node position to determine the positions of the base-stations. During the setup, the base-stations of which we want determine the position act as mobile nodes. The use of the algorithm will simplify the setup procedure, but the most important advantage is the reduced setup-time.

The positions of the base-stations are fixed. We have more information about the position of base-stations than we have about mobile nodes. For example, we know if certain base-stations are attached to a wall so their position is on a certain line. Or we can know the distance between two nodes exacly so we shouldn't use the measurements of the nodes that have a measurement error. These are examples of constraints we can use to improve the position determination of the base-stations, which helps to improve the position determination of the mobile nodes when the system is in use. This master thesis describes the implementation of constraints in the algorithm.

Analysis of the algorithm

The input of the algorithm is the set of distance measurements between the nodes. With the distance measurements the algorithm determines the positions of the mobile nodes. Because there is an error on the measurements, there is no exact solution that fits all the distance measurements. We can indicate the difference between the calculated position and the distance measurements with a least squares formula.

The solution (the set of positions that is best fits the measurements) will be approached

in an iteration. In each iteration we move the nodes over a vector. This vector is different for each node and is calculated with the formula of Polak-Riviere for conjugate gradient. The use of conjugate gradient reduces the number of needed iterations needed to find the minimum of the least squares function.

The movement of the mobile nodes is the product of the conjugate gradient vector and a value. To determine this value we use a line method to determine the minimum of the least squares function on the conjugate gradient vectors. We search for the movement over the conjugate gradient of the mobile nodes such that the result of the least squares function is as small as possible.

After each iteration we compare the result of the least squares formule with the result of the last iteration. If the difference between these two is small we are close enough to the solution.

Constraints

In this algorithm we have to implement the use of constraints. To be independent of the release of the algorithm, we want to minimise the changes to the algorithm.

We began with the development of the class "Constraint". This is the parent-class of each constraint we want to use. In the Constraint-class we defined the necessary functions that are common for all constraints. With these functions we can manipulate the positions of the nodes and the conjugate gradient vector. With these functions we can ensure that after each iteration the nodes satisfy the constraints and if possible ensure that the nodes can't deviate from the constraints. By the use of inheritance we can call these functions (which are overwritten by the child-classes, which are the constraints we use) without knowing the type of the constraint in advance.

We use a combination of linked lists and circular lists to work with the constraints (the class "ConstraintsMatrix"). With the use of linked lists we can pass through the several constraints and nodes that need to satisfy a constraint. We use a circular list to pass through all the nodes that need to satisfy a certain constraint. The other circular list we use to pass through all the constraints a certain node need to satisfy. The constraints we implemented are:

- Line: As like the name indicates, this constraints represents a line. If a node needs to satisfy these constraints, its position needs to be on this line. The description of the line is known at the beginning of the algorithm. It is definded in the form 'ax+b' where 'a' and 'b' are the given parameters.

- VLine: because it is impossible to describe a vertical line the same way the Line constraint does, we define a VLine constraint which represents a vertical line. By the initialisation of the constraint we use one parameter to indicate the x-position of the vertical line.

- Afstand: This constraint gives the possibility to set a distance between two nodes manually. By manually define the distance between nodes, we don't have to use the

distance measurement with an error of the nodes themself. Also after setup, this constraint can be used. If two nodes are attached to the sides of a metal car, there can be a NLOS measurements so the determination of the positions is incorrect. By using the constraint "Afstand" we can improve this.

- ULine: If we know the nodes are on a line, but we don't know the description of the line, we use the constraint "ULine". By using least squares we can determine the line description that bests fit the positions of the mobile nodes. Because we are using coordinates, there can be an error on the x-position and on the y-position.

After each movement of the nodes we ensure they satisfy the constraints. We pass through the linked list of constraints, use a circular list to determine which nodes need to satisfy this constraints and call the function "initialise" of the constraint which changes the position of the nodes so they satisfy the constraint.

During the line method we search for the minimum of the least squares function. The least squares function uses the position of the nodes to calculate a result. Because we change the positions of the nodes after the movement over the conjugate gradient vector, we have to manipulate the positions that are used in the least squares calculation. Also during the line method we pass through the constraints so the positions that are used in the least squares formula are the same as the changed positions we become after the "initialise" function call.

Some constraints give us the possibility to manipulate the conjugate gradient vector so the nodes never deviate from the constraint. After the determination of the conjugate gradient vectors, we pass through the constraints to call the function "Execute" which manipulates the conjugate gradient vector if possible.

Conclusion

By using the algorithm we use to determine the positions of mobile nodes also for the determination of the positions of the base-stations during setup, we can reduce the setup-time. To improve the position determination we implement constraints.

Inhoudsopgave

1 Situering en doelstelling **xix**

1.1 Situering . . . xix

1.2 Meer over het LOST project . . . xx

1.3 Doelstelling . . . xxi

1.4 Overzicht van de hoofdstukken . . . xxii

2 Literatuurstudie **1**

2.1 C++ . . . 1

2.1.1 Inleiding . . . 1

2.1.2 Wat is C++ . . . 1

2.1.3 Dynamische arrays . . . 1

2.1.4 Gelinkte lijsten . . . 2

2.1.5 Overerving . . . 3

2.2 Numerieke methoden . . . 5

2.2.1 Inleiding . . . 5

2.2.2 C++ in de numerieke wiskunde . . . 5

2.2.3 Het vinden van nulpunten . . . 6

2.2.4 Zoeken naar extrema van functies . . . 9

2.2.5 Het modelleren van data . . . 13

2.2.6 Het kleinste kwadraat . . . 13

3 Analyse van de broncode **17**

3.1 Inleiding . . . 17

3.2 De probleemstelling . . . 17

3.2.1 Positiebepaling . . . 17

3.2.2 Beperkingen van het systeem . . . 18

3.3 Het iteratieve proces 19
3.3.1 Verkleinen van de totale fout 19
3.3.2 De gradiënt van de mobiele nodes 19
3.3.3 De toegevoegde gradiënt 20
3.3.4 Lijnminimalisatie 21
3.4 Conclusie 28

4 Randvoorwaarden 29
4.1 Waarom randvoorwaarden gebruiken 29
4.2 De toevoeging van randvoorwaarden 30
4.2.1 De klasse ”Constraint” 30
4.2.2 De klasse ”ConstraintMatrix” 32
4.3 Geïmplementeerde randvoorwaarden 41
4.3.1 Line 43
4.3.1.1 Line klasse 43
4.3.1.2 Execute 44
4.3.1.3 Initialise 45
4.3.1.4 Besluit 47
4.3.2 VLine 47
4.3.2.1 VLine klasse 47
4.3.2.2 Execute 48
4.3.2.3 Initialise 48
4.3.2.4 Besluit 49
4.3.3 Afstand 49
4.3.3.1 Afstand klasse 49
4.3.3.2 Initialise 50
4.3.4 ULine 51
4.3.4.1 ULine klasse 51
4.3.4.2 Bepalen van het lijn voorschrift 52
4.3.5 Conclusies 59

5 Implementatie 61
5.1 Inleiding 61
5.2 Aanroepen van ”initialise” 61

5.2.1 Na toevoeging in "Constraint Matrix" 62
5.2.2 Na elke verschuiving 63
5.2.3 Tijdens de iteratie van de lijnminimalisatie 64
5.2.4 In de functie "UnLine" 66
5.3 Aanroepen van "Execute" 67
5.4 Enkele voorbeelden 69
5.4.1 Beschikbare tools 69
5.4.2 Voorbeeld met Line en VLine 70
5.4.3 Voorbeeld Afstand 72
5.5 Conclusies 74

6 Besluit **75**

Bibliografie **77**

Verklarende lijst van afkortingen en symbolen

RTLS Staat voor ”Real Time Location System”. Dit is een systeem dat toelaat om de locatie van goederen, personen, ... te bepalen in een vooraf bepaalde tijd.

GUI Staat voor ”Graphical User Interface”. Een gebruikersinterface die de gebruiker in staat stelt om op een grafische manier te communiceren met een elektronisch apparaat zoals computers, PDA's, ...

NLOS Staat voor ”non-line-of-sight”. Deze term wordt gebruikt om het pad van een radio-golf aan te geven dat gedeeltelijk gehinderd wordt door obstakels.

ε De digitale voorstelling van getallen is met een eindige nauwkeurigheid. Met ε wordt aangegeven wat de kleinste waarde is zodat $1 + \varepsilon > 1$ waar is.

LOST LOcation System for Sensor Tracking, de naam van het systeem voor positiebepaling waaraan Essensium werkt.

GPS Global Positioning System. Dit is de naam van een satellietplaatsbepalingssysteem dat wereldwijd gebruikt wordt.

ToA Time Of Arrival, de tijd die nodig is om van een zender tot een ontvanger te geraken. Hierbij wordt gebruik gemaakt van de absolute tijd in plaats van een tijdsverschil.

RSS Received Signal Strength, de signaal sterkte die ontvangen wordt.

UWB Ultra Wideband, een technologie waarbij een grote hoeveelheid data op een korte tijdsspanne verzonden wordt. Dit gebeurt door de data over een grote bandbreedte te verspreiden.

WLAN Wireless Local Area Network, een netwerk van beperkte grootte dat twee of meer apparaten draadloos met elkaar verbindt.

Lijst van figuren

2.1 Een functie zonder nulpunt . . . 7
2.2 Oplossing via secant-methode . . . 8
2.3 Oplossing via false position . . . 8
2.4 Zoeken naar een minimum met golden section . . . 10
2.5 Zoeken naar een minimum met behulp van parabool fitting . . . 11
2.6 Toegevoegde gradiënt . . . 12
2.7 Least squares . . . 14

3.1 Probleemstelling . . . 18
3.2 Toegevoegde gradiënt . . . 20

4.1 Gelinkte lijst met 1 element . . . 37
4.2 Gelinkte lijst met twee elementen . . . 37
4.3 Begin van "Constraint Matrix" . . . 42
4.4 Voorbeeld van de "Constraint Matrix" na toevoeging van meerdere randvoorwaarden . . . 42
4.5 Projectie van een vector . . . 45
4.6 Projectie naar VLine . . . 48
4.7 Afstand randvoorwaarde . . . 51

5.1 Een voorbeeld van opstelling zonder het gebruik van randvoorwaarden . . . 71
5.2 Een voorbeeld van opstelling met twee constraints . . . 71
5.3 Een voorbeeld van opstelling met vier constraints . . . 72
5.4 Een opstelling met een meting waar een grote fout op zit. . . . 73
5.5 Het gebruik van de Afstand randvoorwaarde . . . 73

Hoofdstuk 1

Situering en doelstelling

1.1 Situering

Essensium is een bedrijf dat gespecialiseerd is in ASIC oplossingen voor draadloze sensor netwerken. Essensium biedt een brede waaier van technologieën aan die nodig zijn bij het opzetten van draadloze applicaties.

Essensium werkt aan de ontwikkeling van een RTLS. Hiermee wordt de positie van mobiele nodes bepaald. Door deze mobiele nodes te monteren op palletten, wagens, ... kan ook hiervan de positie bepaald worden.

De positiebepaling kan op verschillende manieren gebeuren. Zo is het mogelijk met afstandsmetingen tussen de nodes te werken, maar het is ook mogelijk dit te doen op basis van afstandsverschillen tussen de nodes. Het algoritme waarop deze masterproef gebaseerd is, maakt gebruik van afstandsmetingen. De rest van de tekst zal dan ook enkel deze methode van positiebepaling behandelen.

Voor dit product zijn er 2 typen nodes aanwezig:

- basisstations: hiervan is de positie gekend en op voorhand vastgelegd. Een basisstation zal afstandsmetingen doen naar de mobiele nodes.
- mobiele nodes: hiervan gaan we de positie bepalen aan de hand van afstandsmetingen, hetzij vanuit basisstations, hetzij vanuit andere mobiele nodes.

Essensium heeft een algoritme ontwikkeld voor deze positiebepaling. Deze implementatie is gemaakt in C++. De meeste methoden die hierin gebruikt worden zijn terug te vinden, en besproken, in de literatuur (Press et al. (2007) is hiervoor een zeer goede bron). Om de positiebepaling tot stand te brengen zijn een aantal methoden toegevoegd die specifiek zijn voor de applicatie (zoals het bepalen van een gradiënt, verschuiven van nodes, ...). Deze masterproef bouwt verder op deze bestaande implementatie.

Om op een grafische manier te kunnen werken met de C++ implementatie, heeft Essensium een GUI ontwikkeld. Deze GUI is ontwikkeld in python zodat hij een grotere flexibileit biedt en eenvoudig een grafische weergave van de resultaten kan genereren.

1.2 Meer over het LOST project

De positiebepaling binnen gebouwen is een moeilijke taak. Momenteel is er nog geen bestaande technologie die een relevante afstandsmeting (tot op beter dan een meter nauwkeurig) kan doen over grote afstanden. Het LOST (LOcation System for Sensor Tracking) project probeert hierin tegemoet te komen.

Voor het maken van een correcte positiebepaling in open lucht, bestaan er reeds technologieën als GPS. Deze positiebepaling is mogelijk door de line-of-sight die meestal aanwezig is. Dergelijke technologieën zijn echter niet in te zetten binnen gebouwen, omwille van de aanwezigheid van obstakels (kasten, bureaus, ...) zodat er reflecties ontstaan die elkaar beïnvloeden. Dit komt omdat deze technologieën meestal op narrow-band signalen gebaseerd zijn. Hierdoor zijn deze breed in het tijdsdomein en zullen reflecties (die op verschillende tijdsstippen aankomen) elkaar overlappen. Het onderscheiden van reflecties wordt zo een moeilijke taak waardoor er fouten ontstaan.

Een aantal technologieën voor binnen gebouwen, maken gebruik van RSS (received signal strength). Het gebruik van RSS is echter moeilijk doordat er geen sterk verband tussen de afstand en de signaalsterkte bestaat. Dit verband is ook sterk afhankelijk van de hardware implementatie. Een andere mogelijkheid is het gebruik van de propagation time. Hierbij wordt gemeten hoelang een signaal erover doet een andere node te bereiken. Het gebruik van UWB is hier zeer geschikt voor. Hierbij wordt de data over een grote bandbreedte verspreid, en binnen een zeer kort tijdsinterval verzonden waardoor het onderscheiden van reflecties veel eenvoudiger wordt. Het gebruik van UWB is echter kostenintensief omwille van het beperkte bereik. Hierdoor zijn er veel basisstations nodig wat een grote hardwarekost meebrengt. Ook worden bij UWB systemen de basisstations meestal onderling verbonden met draden voor synchronisatie. Dit brengt een grote installatiekost met zich mee.

Het LOST project baseert zich op het feit dat narrow-band technologieën als WLAN toch in kleine mate een wide-band signaal aanwezig hebben (wat eventueel nog kan worden gestimuleerd). Door dit wide-band signaal ook te gebruiken, kan de nauwkeurigheid van de afstandsmetingen worden verbeterd.

Het LOST project gebruikt ToA (Time of Arrival). Om het aankomende signaal te detecteren, wordt gezocht naar het begin van het pakket (waardoor het probleem van overlapping vermeden wordt). De nauwkeurigheid waarmee de propagation time (en dus de afstandsmeting) gemeten wordt is sterk afhankelijk van de tijdsresolutie van het systeem. Om een nauwkeurigheid van 1m te voorzien, moet er minstens gebruik worden gemaakt van een bandbreedte van 150MHz over het volledige datapad. Ook wordt niet de tijd gemeten tussen node A en node B, maar de tijd nodig om heen en weer te gaan (met een kleine offset waarin het pakket aanwezig is bij node B, waarna het wordt terug gezonden). Zo wordt voorkomen dat de nodes tot op de nano-seconde nauwkeurig moeten worden gesynchroniseerd. Wel zijn we hierbij afhankelijk van de clock drift tussen de twee nodes, maar deze is klein genoeg bij het gebruik van een 20ppm kristal. De afstand tussen de twee nodes kan berekend worden door formule 1.1, waarin t_B de tijd tussen vertrek en aankomst (roundtrip-time), t_A een vaste tijdsduur dat het pakket in node B aanwezig is

en c de lichtsnelheid voorstelt waarmee het signaal zich voortplant.

$$d = \frac{t_B - t_A}{2}.c \tag{1.1}$$

De verwerkingstijd van een afstandsmeting is 100μs. Dit wil zeggen dat elke seconde een afstandsmeting naar 10 000 nodes kan gebeuren, wat zorgt voor een grote schaalbaarheid van het LOST-project. De tijdsmetingen van t_A en t_B gebeuren met een nauwkeurigheid van 2ns. Hierdoor kan de afstand tussen twee nodes tot op 30cm nauwkeurig bepaald worden. Door een uitmiddeling van meerdere meetresultaten kan dit verbeterd worden tot op 10cm nauwkeurig.

De afstand waarbinnen het LOST project kan worden ingezet is voor in open lucht ongeveer 600m. Voor het gebruik binnen gebouwen is de afstand sterk afhankelijk van het aantal aanwezige obstakels. Het is ongeveer te vergelijken met WLAN. ((Tubbax et al. (2008))

1.3 Doelstelling

Het algoritme dat we kort besproken hebben, heeft als doel het vinden van de posities van mobiele nodes aan de hand van afstandsmetingen. Deze positiebepaling is afhankelijk van de nauwkeurigheid waarmee de positie van de basisstationss is bepaald. Elke onnauwkeurigheid van de positie van de basisstations zal voor een extra onnauwkeurigheid zorgen op de positiebepaling van de mobiele nodes (die gebaseerd zijn op een afstandsmeting vanuit deze basisstations). De posities van de basisstations kan op twee manieren bepaald worden:

- Manueel: we gaan de posities manueel opmeten.
- Berekenen: we gebruiken het positioneringsalgoritme voor het bepalen van de positie van basisstations. Deze procedure zal minder tijd in beslag nemen dan het manueel opmeten van de posities. Tijdens de installatie beschouwen we de basisstations waarvan we de positie willen bepalen als mobiele nodes.

Het berekenen van de posities (keuze 2) is de meest praktische methode van installatie. We zetten het positiebepalingsalgoritme ook voor de positiebepaling van de basisstations. We gaan tijdens installatie de basisstations waarvan we de positie niet exact weten, beschouwen als mobiele nodes.

Zoals we eerder al hebben aangehaald, is het echter zeer belangrijk dat de positiebepaling zeer nauwkeurig gebeurt. Deze masterproef bestaat uit het uitbreiden van het algoritme met randvoorwaarden zodat de positiebepaling nauwkeuriger wordt, wat rechtstreeks resulteert in een nauwkeurigere positiebepaling van de mobiele nodes.

Om dit te realiseren dienen we volgende stappen te zetten:

- Een grondige analyse van het bestaande algoritme: Gezien we dit algoritme willen uitbreiden, is het belangrijk dat we goed weten hoe dit werkt. Aan de hand

hiervan kunnen we bepalen welke factoren we kunnen manipuleren, en dus hoe we onze randvoorwaarden kunnen vertalen naar informatie die bruikbaar is voor het algoritme.

- Het modelleren van een randvoorwaarde: We moeten een model opstellen van een randvoorwaarde. In dit model leggen we vast hoe van randvoorwaarden kan worden gebruik gemaakt, welke functies we hiervoor moeten aanroepen, het aantal parameters dat een randvoorwaarden nodig heeft, op welke manier we onderscheid kunnen maken tussen verschillende randvoorwaarden, enz. We werken hiervoor een interface uit die onafhankelijk is van de randvoorwaarden die we gebruiken.
- Het opstellen van de randvoorwaarden: We moeten de randvoorwaarden die we willen implementeren opstellen vanuit het model. We voorzien elke randvoorwaarde van een aantal methoden die de manipulatie van de factoren die we bij de analyse hebben bepaald uitvoeren. De impact van de randvoorwaarde op de nauwkeurigheid van de resultaten is afhankelijk van de meerwaarde aan informatie die uit de randvoorwaarde kan worden afgeleid. Hoe meer informatie we kunnen afleiden uit een randvoorwaarde, hoe meer informatie we kunnen gebruiken om de nauwkeurigheid te verbeteren.
- Integratie: We moeten het gebruik van randvoorwaarden integreren in het bestaande algoritme. Hierbij proberen we zo min mogelijk wijzigingen aan te brengen in de oorspronkelijke implementatie, zodat het eenvoudig blijft deze integratie ook in nieuwere releases van het algoritme uit te voeren.

1.4 Overzicht van de hoofdstukken

Eerder hebben we de deeltaken opgesomd die nodig zijn voor het uitbreiden van het algoritme. De volgende hoofdstukken zullen deze deeltaken behandelen:

- In hoofdstuk 2 doen we een literatuurstudie. Hierbij gaan we dieper in op de wiskundige principes waarop de methoden van het positioneringsalgoritme gebaseerd zijn.
- In hoofdstuk 3 maken we een analyse van het oorspronkelijke algoritme. We beginnen met de probleemstelling, die aangeeft wat het doel van het algoritme is en welke knelpunten dit met zich meebrengt. Vervolgens zullen we ingaan op de implementatie van het algoritme en op welke manier de knelpunten worden aangepakt.
- In hoofdstuk 4 bespreken we het gebruik van randvoorwaarden. Eerst bespreken we wat we onder een randvoorwaarde verstaan en waarom we deze nodig hebben. Vervolgens bespreken we hoe we een randvoorwaarde voorstellen met een klasse. Tot slot bespreken we de verschillende randvoorwaarden die we hebben opgesteld, en op welke wijze deze bijdrage tot een nauwkeuriger resultaat.
- In hoofdstuk 5 bespreken we de wijzigingen die we hebben aangebracht in het oorspronkelijke algoritme om van randvoorwaarden gebruik te maken. We bespreken

hoe deze randvoorwaarden worden aangesproken vanuit het algoritme om op de juiste momenten beroep te doen op de meerwaarde ervan.

- In hoofdstuk 6 overlopen we de belangrijkste realisaties van de masterproef. Tot slot halen we mogelijke verbeteringen aan.

Hoofdstuk 2

Literatuurstudie

2.1 C++

2.1.1 Inleiding

Mijn eindwerk bestaat uit het uitbreiden en optimaliseren van een algoritme voor positie bepaling dat is ontwikkeld in opdracht van Essensium. Dit algoritme bepaalt aan de hand van metingen van vaste devices en mobiele devices onderling de positie van mobiele devices. Deze code is ontwikkeld in de programmeertaal C++.

2.1.2 Wat is C++

C++ is door Bjarne Stroustrup in de jaren '80 ontwikkeld als een uitbreiding op de programmeertaal C. Als eerste kwam de toevoeging van klassen en objecten zodat we vanaf dan over een objectgeoriënteerde taal konden spreken. Al snel volgde verdere uitbreidingen als virtuele functies, operator overloading en de mogelijkheid om over te erven van verschillende klassen (waarin C++ in verschilt van veel andere programmeertalen). Doordat C++ een uitbreiding is van C bevat deze programmeertaal nog altijd de vele voordelen die C te bieden heeft. Het is namelijk nog altijd een programmeertaal die op laag niveau kan werken en dus zeer flexibel is, wat tot grote efficiëntie leidt. (wikibooks (2008)).

2.1.3 Dynamische arrays

C++ biedt de mogelijkheid om dynamische arrays te gebruiken. Hierbij wordt de grootte van de array tijdens runtime bepaald. Bij het gebruik van dynamische arrays wordt er tijdens runtime een stuk geheugen gereserveerd ter grootte van de op te slane elementen. Zo vermijden we het probleem dat je als programmeur een te grote of te kleine geheugenplaats reserveert.

```
int* a = NULL;
int n;
cin >> n;
a=new int[n];
for (int i=0; i<n; i++) {
    a[i] = 0;
}
. . .
delete [] a;
a = NULL;
/*
 * Volgende code is equivalent aan de vorige:
 * {
 *  int a[n];
 *  ...
 * }
 */
```

Het gebruik van 'new' kent zijn equivalent in C als de functie 'malloc'. Het is dan ook, net als in C, aan de gebruiker om het stuk gereserveerde ruimte vrij te geven. In C++ gebeurt dit met 'delete'.

Ook hier is het echter nodig om op een gegeven moment het aantal op te slane elementen te kennen. Dynamische arrays bieden nog niet de flexibiliteit van gelinkte lijsten, die tijdens runtime uitgebreid kunnen worden. Voor het bewaren van de gebruikte randvoorwaarden was nood aan deze meer flexibele oplossing.(Swartz (1994)).

2.1.4 Gelinkte lijsten

Bij gelinkte lijsten maken we een ketting van stukjes geheugen die elk een deel data bewaren en verwijzen naar het volgend stukje gebruikt geheugen van de ketting. Het bewerken van de gelinkte lijst bestaat uit het wijzigen van pointers en het vrijgeven of reserveren van stukken geheugen. Het is dus een zeer flexibele geheugenstructuur. Door het aantal verwijzingen naar andere stukken geheugen te vermeerderen kunnen ook extra manieren gebruikt worden om de verschillende elementen te doorlopen.

In een objectgeoriënteerde taal zijn 'data encapsulation' en 'data hiding' belangrijke onderwerpen. In plaats van elk element van de gelinkte lijsten publiek toegankelijk te maken, kunnen we deze meer afschermen zodat de interface naar de data in de klasse gecontroleerd is. Van buiten af kan een gebruiker van de gelinkte lijst klasse via methoden van de klasse data toevoegen of uitlezen, maar hij bepaalt niet op welke wijze binnen de klasse de elementen opgeslagen worden in de gelinkte lijst.

Het grote nadeel van gelinkte lijsten is dat het zoeken op een inefficiënte manier gebeurt. Het is niet mogelijk om rechtstreeks naar een element midden in te lijst te verwijzen wat

bij arrays wel kan. Eerst moeten alle voorgaande elementen doorlopen worden. Zo zullen bijvoorbeeld voor het aanspreken van het 6de element van een gelinkte lijst eerst de 5 voorgaande elementen moeten doorlopen worden, terwijl dit bij een array kan gebeuren door het element via array[5] aan te spreken is (bij een zero-origin array). (FunctionX.inc (2005)).

Om het zoeken te verbeteren maar toch de uitbreidbaarheid niet te verliezen, kan men ook gebruik maken van binaire bomen. Hierbij wordt elk element van 2 pointers voorzien die elk naar een nieuw element wijzen of naar 'NULL' indien er geen nieuwe elementen meer zijn. Om het zoeken te versnellen moet je een bepaalde structuur aanbrengen in de boom (er een zoekboom van maken). Zo kan je de eerste pointer naar hogere waarden en de andere pointer naar lagere waarden laten verwijzen. Zo wordt de zoek-efficiëntie verbeterd van $O(\frac{N}{2})$ naar $O(\log_2(N))$. Een element toevoegen aan een boom is echter complexer dan aan een gelinkte lijst (waar eenvoudig een extra pointer naar het tot dan toe laatste element kan worden bijgehouden). De keuze van opslag hangt dus sterk af van de operaties die het meest moeten gebeuren in het programma. (Crauwels (2008)).

In mijn eindwerk maak ik van gelinkte lijsten intensief gebruik. Zo zijn al de gebruikte randvoorwaarden en al de nodes die hieraan gekoppeld zijn, opgeslagen in een datastructuur die een combinatie is van gelinkte lijsten en circulaire lijsten (hierbij verwijst het laatste element naar het eerste element). Ook om de verschillende nodes die betrokken zijn in een randvoorwaarde door te geven aan een functie die deze verwerkt, maak ik gebruik van gelinkte lijsten.

2.1.5 Overerving

In een objectgeoriënteerde omgeving is het gebruikelijk dat klassen van elkaar overerven. Dit heeft als doel dat gemeenschappelijke eigenschappen niet geherdefinieerd hoeven te worden. Indien er bijvoorbeeld een klasse 'Vorm' bestaat, kunnen we bij het definiëren van een nieuwe klasse 'Cirkel' aangeven dat dit een vorm is, en zodoende over de eigenschappen van de klasse 'Vorm' dient te beschikken. Door het gebruik van overerving ontstaat er een hiërarchische structuur van klassen die met elkaar verwant zijn.

Bij het stijgen in de structuur, dus naar een meer algemene vorm van een klasse gaan (in het voorbeeld, van een 'Cirkel' een 'Vorm' maken) spreken we over upcasting. Bij deze bewerking spreekt het voor zich dat er zich weinig problemen kunnen voordoen doordat we naar een beperktere verzameling van functies en variabelen gaan. De omgekeerde bewerking, dalen in de hiërarchie (downcasting), spreekt echter niet voor zich. Het kan immers zijn dat er meerdere child-klassen zijn, en dan moet er uitgemaakt worden naar welke klasse er gegaan moet worden.

C++ voorziet hier functies voor. Men kan een onderscheid maken tussen 'dynamische casting' en 'statische casting'. Het grootste verschil is dat je bij dynamische casting moet gebruik maken van een echt polymorfe klassenstructuur. Dit wil zeggen dat er gebruik wordt gemaakt van virtuele functies. Dynamische casting gebruikt de informatie die in VTABLE is opgeslagen. VTABLE is een array die verwijzingen naar functies in de subklassen bevat. Deze functies zijn de geherdefinieerde versies van een oorspronkelijke functie in de basisklasse. Zo zal bijvoorbeeld de functie 'teken' een andere actie

moeten uitvoeren voor verschillende vormen. De functie ‘teken’ kan als virtueel worden aangemaakt in de basisklasse ‘Vorm’, zodat bij het aanmaken van een ‘Vorm’ object er een VTABLE wordt aangemaakt die verwijzingen bevat naar de verschillende versies van ‘teken’ van de verschillende subklassen (‘Vierkant’, ‘Cirkel’, ‘Driehoek’, ...). Hieruit kan dus worden afgeleid wat de subklassen zijn. Bij statische casting is hier geen nood aan. Het casten ligt dan in de handen van de programmeur, maar levert wel een kleine performantiewinst op.

Zowel statische als dynamische casting vereist echter dat tijdens het compileren al geweten is wat de child-klasse zal zijn. Het gebruik van downcasting gaat in tegen de filosofie van objectoriëntatie. Als we expliciet de overgang moeten maken van een basisklasse naar een child-klasse wil dit zeggen dat de functionaliteit van de basisklasse tekort schiet. Daarom wordt dit zo veel mogelijk vermeden.

Door gebruik te maken van een virtuele functies kan downcasting worden vermeden. Met behulp van de VTABLE kunnen we immers functies van een child-klasse aanroepen zonder de interne structuur van de child-klasse te kennen.

```
1  class Shape{
2   public:
3     void virtual function Draw();
4  };
5
6  class Square{
7   public:
8     void Draw(){
9       std::cout << "Geeft vierkant weer" << std::endl;
10    }
11 }
12
13 class Triangle{
14  public:
15    void Draw(){
16      std::cout << "Geeft driehoek weer" << std::endl;
17    }
18 }
19
20 void functie(){
21    Shape *p1 = new Triangle();
22    Shape *p2 = new Square();
23
24    p1->Draw();
25    p2->Draw();
26 }
```

In het voorbeeld is te zien hoe gebruik kan gemaakt worden van virtuele functies. In de

void functie worden twee pointers naar Shape aangemaakt, maar in plaats van deze naar een object van Shape te laten wijzen, laten we deze verwijzen naar een object van een subklasse. Door de functie 'Draw' virtueel te maken zal bij het aanroepen van deze functie op de pointers niet de Draw functie van 'Shape' worden aangeroepen, maar zal, afhankelijk van het type van pointer, in de VTABLE gekeken worden waar de overeenkomstige functie 'Draw' zich in die klasse bevindt. Zo zal dus de functie van de subklasse aangeroepen worden, en niet de Draw functie van de basisklasse.

Bij het toevoegen van de randvoorwaarden wordt voor elke randvoorwaarde een afzonderlijk object gemaakt van een klasse die overerft van de 'Constraint' klasse. De verwijzing naar deze objecten gebeurt echter met een pointer van het type 'Constraint'. Bij het aanroepen van een virtuele functie die in de 'Constraint' klasse is gedefinieerd, zal gekeken worden van welke klasse het object in werkelijkheid is, en zal de geherdefinieerde functie uit de juiste child-klasse worden uitgevoerd. Bij het compileren van de code is dus nog niet bepaald om welk type randvoorwaarde het zal gaan, maar wordt dit bepaald tijdens runtime. (linuxtopia (n.d.)).

2.2 Numerieke methoden

2.2.1 Inleiding

Wij vonden één bron die vrij volledig beschrijft hoe numerieke methoden kunnen aangewend worden in software (Press et al. (2007)). De meeste andere bronnen verwijzen hiernaar en baseren hun informatie hierop. In de loop van de jaren zijn er verschillende versies uitgebracht in de Numerical Recipes reeks, zo zijn er afzonderlijke uitgaven voor Fortran, C en C++. In 2007 is de meest recente uitgave verschenen, die niet meer op een programmeertaal gericht is, maar eerder op de methode van implementeren. Hierin wordt duidelijk omschreven wat de voordelen en zwaktes zijn van bepaalde oplossingsmethoden. De informatie omtrent numerische technieken is dan ook uit deze bron gehaald, behalve wanneer dit expliciet anders vermeld wordt.

2.2.2 C++ in de numerieke wiskunde

C++ is een taal die, zoals eerder vermeld, gebaseerd is op C, en dus dichtbij de hardware staat. Alhoewel het elementen bevat die bij hogere talen aanleunen is het nog altijd een programmeertaal die veelvuldig gebruikt wordt om systeem-software mee te schrijven. Alhoewel C++ dus niet ontworpen is met het oog op numerieke wiskunde (in tegenstelling tot talen zoals Fortran), kan het hier toch voor gebruikt worden.

Een belangrijk aspect bij het gebruik van numerieke wiskunde in C++ (en in andere programmeertalen) is de fout die gemaakt wordt. Een computer systeem is gebonden aan de gebruikte typen variabelen, die op hun beurt slechts over een beperkt aantal bits beschikken om waarden voor te stellen. We kunnen een onderscheid maken tussen gehele waarden (zoals int, long, ...) en floating point (decimale waarden zoals float, double, ...) numerieke waarden. Beide beschikken over een beperkt aantal bits om een waarde

voor te stellen. Het is dus niet mogelijk om een berekening tot een oneindige precisie te berekenen.

De fout die deze beperking met zich mee brengt, noemen we de ‘roundoff-error’ (afrondingsfout). Deze fout is enkel te wijten aan de beperking van het computersysteem (zo zal deze fout kleiner worden als er een groter aantal bits gebruikt wordt). Indien een afrondingsfout zich al vroeg in de berekeningen voordoet, kan deze zich voortzetten doorheen het programma en zo een groter effect bereiken dan oorspronkelijk het geval was. In dit geval spreken we over een ’onstabiel systeem’. Dat moeten we kost wat kost vermijden. Een ander type fout is de ’truncation-error’. Deze fout is te wijten aan het beperken van het aantal bewerkingen in een berekening. Zo zal je als programmeur bij het uitrekenen van een integraal niet werkelijk al de punten van een functie berekenen om tot een som te komen, maar zal je dit aantal beperken en een veronderstelling maken over hoe de functie tussen deze berekende waarden loopt. Hoe hoger het aantal punten waarvoor de functiewaarde berekend wordt, hoe zwaarder de gehele berekening van de integraal zal zijn, maar ook hoe correcter het resultaat de werkelijke integraal zal benaderen. De afwijking van de werkelijkheid noemen we de ’truncation-error’.

2.2.3 Het vinden van nulpunten

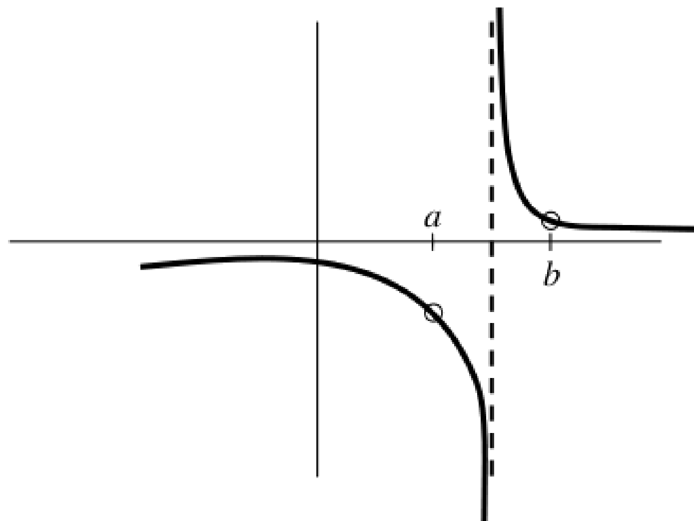
Het vinden van nulpunten van een vergelijking wordt regelmatig gebruikt bij het oplossen van vergelijkingen. Voor een aantal functies (bijvoorbeeld een 2de graadsfunctie) is bekend hoe de nulpunten gevonden kunnen worden. Door één van de twee leden gelijk te maken aan nul, kunnen met behulp van deze methoden de nulpunten, en dus de oplossing van de vergelijking, gevonden worden.

Om een oneindige cyclus te vermijden is het aan te raden om een zoekdomein voor het nulpunt te begrenzen (2 functiewaarden met een verschillend teken). In een begrensd zoekdomein hebben we grote kans een minimum te vinden.

Het vinden van een nulpunt gebeurt meestal door een bepaalde procedure iteratief op te roepen totdat een bepaalde voorwaarde bereikt is. Zo kan je bijvoorbeeld beginnen met de grenzen vast te leggen en naar het nulpunt toe te gaan door telkens het zoekdomein te halveren (deze techniek is bekend als ’bisection’). De kans dat zo exact op het nulpunt wordt gestoten, is vrij klein en zou dus een zeer hoge (oneindige) iteratie vergen. Daarom gaan we een voorwaarde stellen die het einde bepaalt. Indien het resultaat binnen een bepaald interval rond nul ligt, aanvaarden we dit als een nulpunt.

Het begrenzen van een nulpunt is niet altijd eenvoudig. Indien de functie niet continu is volstaat het niet om twee punten met verschillend teken te vinden, om te besluiten dat hier een nulpunt tussen zal liggen. Een voorbeeld hiervan is weergegeven in figuur 2.1. De functie is links van de asymptoot negatief, terwijl ze rechts van de asymptoot positief blijft. Om te vermijden dat we in een oneindige cyclus terecht komen gaan we het aantal iteraties waarin we naar het nulpunt toe werken, beperken.

De secant-methode (fig. 2.2) en de false-position-methode (fig. 2.3) kunnen beide gebruikt worden bij het vinden van een nulpunt. Beide zullen zelfs voor een constante functie sneller tot aan het nulpunt raken dan de hiervoor vermelde bisection-methode. In beide methoden



Figuur 2.1: Alhoewel beide grenzen een tegengestelde functiewaarde hebben ligt er toch geen nulpunt tussen. Het is geen continue functie

werken we met grenzen waartussen we de functie als een rechte beschouwen om tot een nieuwe grens te komen (het punt waar de rechte de X-as snijdt). Bij elke iteratie wordt één van de vorige grenzen vervangen door de nieuw bekomen waarde. Het enige verschil tussen beide methoden is dat de secant-methode altijd de oudste grens vervangt, terwijl de false-position-methode zorgt dat het nulpunt steeds tussen de grenzen zal blijven. Bij de secant-methode is het dus niet gegarandeerd dat er naar het nulpunt gegaan wordt. Wiskundig gezien zal de secant-methode sneller naar het nulpunt convergeren dan de false-position methode doordat er altijd nieuwe punten gebruikt worden, maar je loopt dus het gevaar dat het nulpunt niet gevonden wordt.

Een functie waarmee de meesten wel vertrouwd zijn, is de kwadratische vergelijking. Het voorschrift van deze functie is te zien in vergelijking 2.1. Het vinden van nulpunten van deze functie kan op verschillende manieren gebeuren (vergelijking 2.2 of vergelijking 2.3).

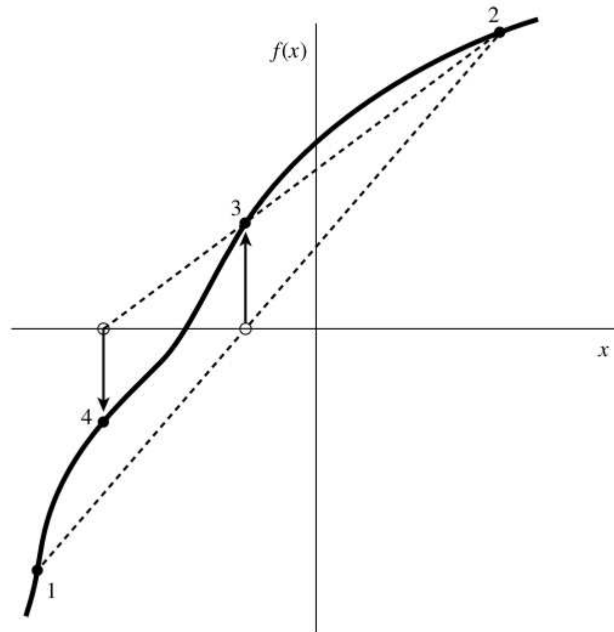
$$y = ax^2 + bx + c \tag{2.1}$$

$$x = \frac{-b\pm\sqrt{b^2 - 4ac}}{2a} \tag{2.2}$$

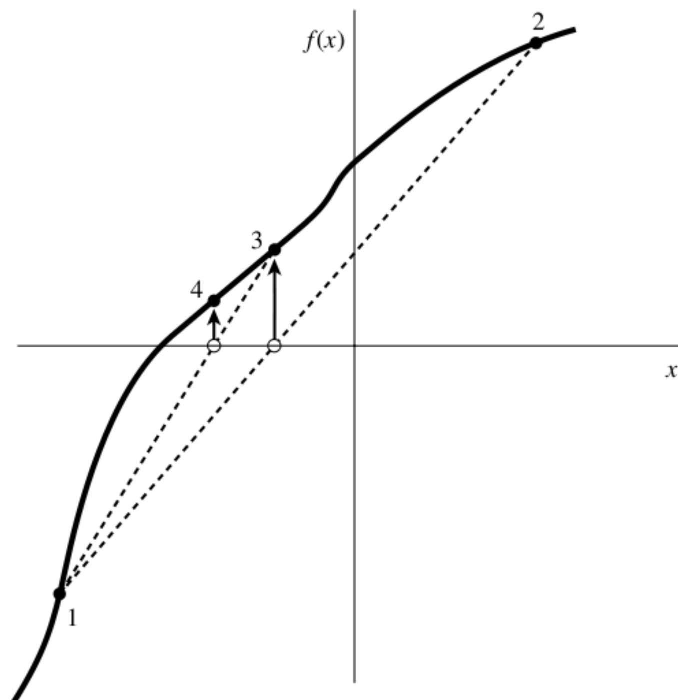
$$x = \frac{2c}{-b\pm\sqrt{b^2 - 4ac}} \tag{2.3}$$

Beide vergelijkingen beschikken over een gelijke term, die in beide gevallen een gevaar met zich mee brengt. Indien c of a zeer klein is, zal er een verschil gemaakt worden tussen 2 waarden die bijna gelijk zijn, dit kan tot een roundoff error leiden. Om dit te voorkomen gaan we met een tussenstap werken (vergelijking 2.4) om hierna tot beide nulpunten te komen (2.5 en 2.6).

$$q = -\frac{1}{2} * (b + sgn(b)\sqrt{b^2 - 4ac}) \tag{2.4}$$



Figuur 2.2: *Er wordt op zoek gegaan naar een nulpunt volgens de secant-methode. Zoals te zien is het niet gegarandeerd dat het nulpunt binnen de grenzen blijft*



Figuur 2.3: *Er wordt een nulpunt gezocht volgens de methode van false position. Zoals te zien blijft het nulpunt telkens tussen de grenzen, maar er wordt niet altijd gebruik gemaakt van de recentst gevonden grens*

$$x1 = \frac{q}{a} \tag{2.5}$$

$$x2 = \frac{c}{q} \tag{2.6}$$

In het boek van Press (Press et al. (2007)) wordt uitvoerig op dit thema ingegaan.

2.2.4 Zoeken naar extrema van functies

Het bepalen van een extremum (een minimum of een maximum) van een functie, heeft veel toepassingen. Met behulp van methoden om een extremum te vinden kan immers een optimaal (of worst-case) resultaat voor een functie gezocht worden. In dit hoofdstuk zullen verschillende methoden die aan bod kwamen bij het vinden van nulpunten terug komen, er zal immers blijken dat het vinden van nulpunten, en het vinden van extrema niet ver uiteen liggen.

Om te beginnen zijn de methoden voor het vinden van extrema in 2 groepen te verdelen.

- methoden die gebruik maken van een afgeleide (of een gradiënt als het over meerdere dimensies gaat).
- methoden die hier geen gebruik van maken

De methoden van de eerste groep zijn wat krachtiger, maar vergen een extra berekening voor het vinden van de afgeleide of de gradiënt. De extra kracht van de methode weegt dan ook niet altijd op tegen de nodige berekening. Indien we een functie in meerdere dimensies (er zijn verschillende variabelen waarvan de functie afhangt) dienen te vereenvoudigen, zal echter altijd gebruik gemaakt worden van een gradiënt om de richting te bepalen waarin zal gezocht worden.

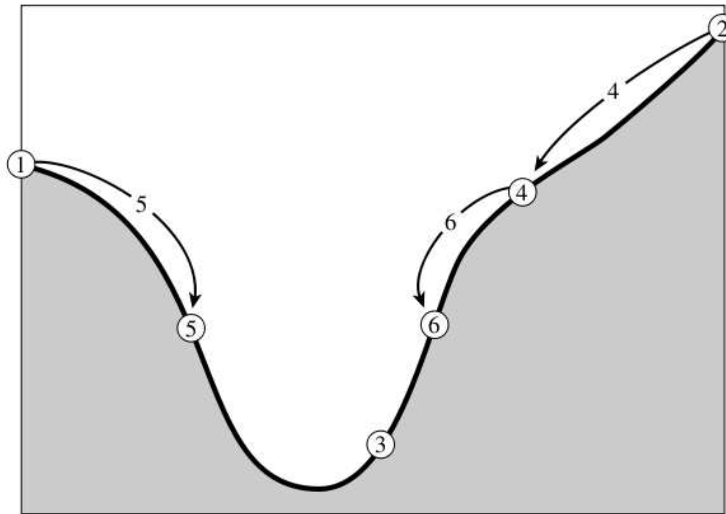
Er is een verschil tussen een lokaal extremum en een globaal extremum. Het vinden van een lokaal extremum is relatief eenvoudig, het vergt een methode die gelijkaardig is aan degene voor het vinden van nulpunten: stap voor stap naar een hogere/lagere functiewaarde toe gaan. Doordat deze methode slechts in een bepaald stuk van de functie op zoek gaat naar een extremum, zijn we niet zeker of dit ook het globale extremum is. Er zijn verschillende methoden om hier meer zekerheid over te krijgen.

- Totaal verschillende vertrekpunten nemen, vanuit elk vertrekpunt naar een extremum toe werken, en dan het beste resultaat selecteren als globaal extremum.
- Indien er een lokaal extremum gevonden is een grote stap hiervandaan doen en kijken of we tot hetzelfde extremum komen.

Bij het zoeken naar een extremum in meerdere dimensies steunen we op de methoden die dit voor één dimensie doen. Door dit te herhalen in verschillende dimensies komen we tot een extremum van de gehele functie. De methode waarop we steunen, kan gebruik maken

van een afgeleide, maar dit is geen noodzaak. Het is echter aan te raden hiervan gebruik te maken indien deze informatie beschikbaar is.

Net zoals bij het zoeken naar nulpunten, gaan we gebruik maken van grenzen. In plaats van deze zo te kiezen dat ze een verschillend teken hebben, gaan we een derde punt gebruiken dat een betere waarde oplevert (afhankelijk van het te zoeken extremum is dit hoger of lager). Een voorbeeld dat gebruik maakt van de gulden snede, is te bekijken in figuur 2.2.4. Door telkens de grenzen naar binnen toe te verleggen, verkleinen we het zoekgebied, en komen we uiteindelijk op een aanvaardbaar extremum.

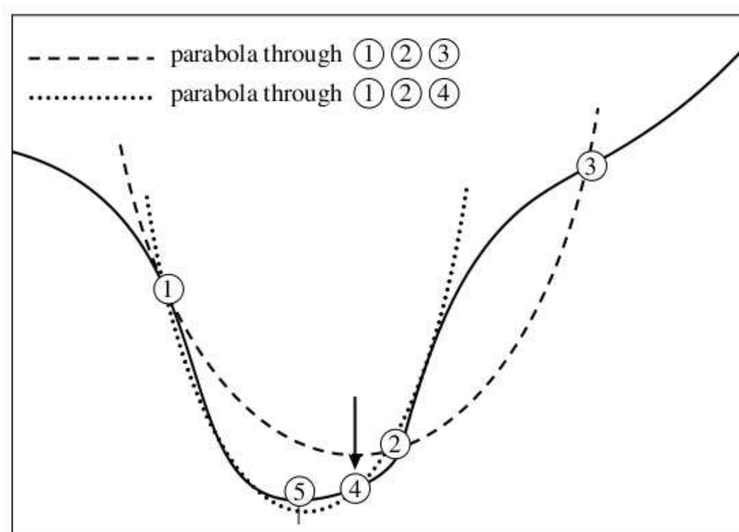


Figuur 2.4: *We gaan stap voor stap op zoek naar het minimum. We starten met de punten 1,2 en 3. Na een sprong komen we uit op punt 4 waardoor we de buitengrens naar binnen kunnen brengen. De volgende stap brengt ons bij punt 5 waarmee we die buitengrens naar binnen kunnen schuiven. Vervolgens komen we terecht bij punt 6 waardoor we weer een buitengrens naar binnen kunnen schuiven, ...*

De gulden-snede methode is geschikt voor elke functie. Indien we echter te maken hebben met een functie die een parabool benadert, kunnen we gebruik maken van betere methoden om tot een extremum te komen. Van een parabool is een minimum eenvoudig te bepalen aan de hand van formule 2.7. Door hierop in te spelen komen we tot een veel efficiëntere methode (fig. 2.5).

$$x = b - \frac{1}{2} * \frac{(b-a)^2 * (f(b) - f(c)) - (b-c)^2 * (f(b) - f(a))}{(b-a) * (f(b) - f(c)) - (b-c) * (f(b) - f(a))} \tag{2.7}$$

Het is onveilig uitsluitend gebruik maken van deze laatste methode. Stel bijvoorbeeld dat de drie punten op eenzelfde lijn liggen, dan zal de noemer nul zijn, en we bekomen een oneindig groot resultaat. Daarom gaan we een oplossingsmethode uitwerken die gebruik maakt van zowel een snelle methode (zoals degene die van de parabool gebruik maakt) als van een meer stabiele maar tragere methode als de gulden snede. Deze laatste dient als backup voor het geval de parabolische interpolatie geen goed resultaat oplevert.



Figuur 2.5: *Door telkens een extremum te berekenen van de parabool die gevormd kan worden met de 3 gekende punten, kunnen we sneller naar het minimum toegaan*

Door gebruik te maken van de afgeleide op het middelste punt, kunnen we bepalen in welk deelinterval we moeten zoeken (a-c of c-b). We kunnen echter niet enkel steunen op de afgeleiden van de functie, omdat dit ons niet genoeg informatie biedt. Zo zijn de afgeleiden voor een minimum en en een maximum beide nul, en uit de afgeleide kan ook niet bepaald worden of een bepaald punt gunstiger of minder gunstig is dan de grenzen en het middelste punt. De afgeleide biedt wel een grote hulp bij het vinden van extrema doordat een extremum altijd nul geeft als afgeleide. We kunnen dus direct gebruik maken van eerder geziene methoden voor het vinden van nulpunten.

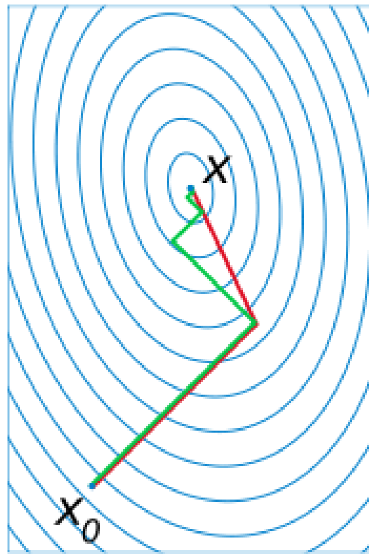
Nu we weten hoe gezocht kan worden naar extrema in één dimensie gaan we dit gebruiken om een extremum te zoeken in meerdere dimensies. Zoals eerder vermeld gebeurt dit door in verschillende richtingen/dimensies naar een extremum te zoeken. Dit kan op verschillende wijzen gebeuren.

Een eerste methode, en de meest inuïtieve, is het kiezen van de eenheidsvectoren. Indien de functie met meerdere veranderlijken werkt ,bijvoorbeeld x, y en z, gaan we achtereenvolgens naar een extremum volgens deze veranderlijken tot we het extremum vinden. Deze methode is voor veel functies geschikt, maar heeft ook een nadeel. Denk als voorbeeld aan een twee dimensionele functie die zeer traag daalt in beide dimensies. Bij het zoeken naar een extremum (in dit voorbeeld een minimum) zullen we in zeer veel kleine stappen naar het extremum toe gaan. Een gelijkaardig probleem doet zich voor bij het gebruik van de gradient als richting die, afhankelijk van de gebruikte één dimensionele methode, al dan niet al beschikbaar is. De gekozen richtingen zullen zich ook hier loodrecht op elkaar opvolgen, wat ook hier tot een zeer groot aantal iteraties kan leiden.

Een betere methode is om de gekozen richtingen te baseren op de voorgaande richtingen. Zoals zojuist beschreven zal bij het gebruik van enkel de gradiënt om de richting te selecteren de resulterende oplossing een opeenvolging van loodrechte richtingen bevatten. Door echter informatie van voorgaande richtingen te benutten, kunnen we tot een richting

komen die ons sneller tot het extremum zal brengen (figuur 2.6). (Wikipedia (2008)). De procedure hiervoor gaat als volgt:

- Het berekenen van de gradiënt als nieuwe richting.
- Het berekenen van een compensatiefactor (meestal voorgesteld met een gamma) die zal dienen om deze nieuwe richting bij te stellen. Dit gebeurt op basis van informatie van de vorige richting. Idealer zou zijn om ook richtingen van daarvoor te betrekken.
- Het aanpassen van de nieuwe richting aan de hand van de compensatiefactor en de laatst gevolgde richting.



Figuur 2.6: *De gradiënt aanpassen aan de hand van de gradiënt uit voorgaande stappen leidt met minder iteraties tot de oplossing. Aan de hand van de cirkels/elipsen worden de hoogtelijnen voorgesteld. Bij gebruik van het gradiënt (loodrecht op de hoogtelijnen) zijn er vijf stappen nodig om het minimum te bereiken. Bij gebruik van de toegevoegde gradiënt (aanpassen van de richting aan de hand van eerder gekozen richtingen) hebben we hier slechts twee stappen voor nodig.*

Verschillende wetenschappers hebben een formule opgesteld om de gamma factor te bepalen. De formule die hier gebruikt wordt, is degene van Polak-Ribiere, maar dit is dus niet de enige mogelijkheid.

$$g = -OudeRichting \tag{2.8}$$

$$x = NieuweRichting \tag{2.9}$$

$$gg = \sum g^2 \tag{2.10}$$

$$dgg = \sum (x + g) * x \tag{2.11}$$

$$gamma = dgg/gg \tag{2.12}$$

Ervaring wijst uit dat het gebruik van Polak-Ribiere minder iteraties nodig heeft dan Fletcher-Reeves bij een nietkwadratische functie. Gezien de functies waarvoor we een lokaal minimum zoeken zelden perfect kwadratisch zijn, is het gebruik van Polak-Ribiere aangeraden.

2.2.5 Het modelleren van data

Het modelleren van data is belangrijk bij de analyse van data. Bij modellering worden de data vergeleken met een functie die opgebouwd is uit een aantal parameters. Het modelleren kan als doel hebben om deze parameters te bepalen, en zo te besluiten welk model het beste de data voorstelt. Dit kan op zijn beurt als doel hebben een voorspelling te kunnen maken over toekomstige metingen van dezelfde aard. Een ander doel van modelleren kan zijn, de controle of de data aan een voorgesteld model voldoen. Hierbij zijn de parameters op voorhand bepaald en wordt er gekeken hoe de resultaten afwijken van het model.

Het modelleren is steeds gebaseerd op een figure-of-merit functie. Dit is een functie die aangeeft hoe goed de data overeenkomen met het model. Hoe deze functie geïnterpreteerd moet worden, hangt af van de context. Zo zullen Bayesianen met de figure-of-merit functie aangeven hoe groot de kans is dat de data overeenstemmen met het model. Bij een groot resultaat is de kans dus groot dat het model juist is voor de data. Frequentists geven echter met de merit functie aan hoe groot de afwijking is van het model. Dit wordt gebruikt bij de 'kleinste kwadraten methode' om de afstand tussen de data-punten en de functie die als benadering gekozen wordt te minimaliseren (fig 2.7). Voor beide geldt echter dat aan de hand van deze functie een extremum bepaald kan worden om hieruit de parameters van het best passende model af te leiden.

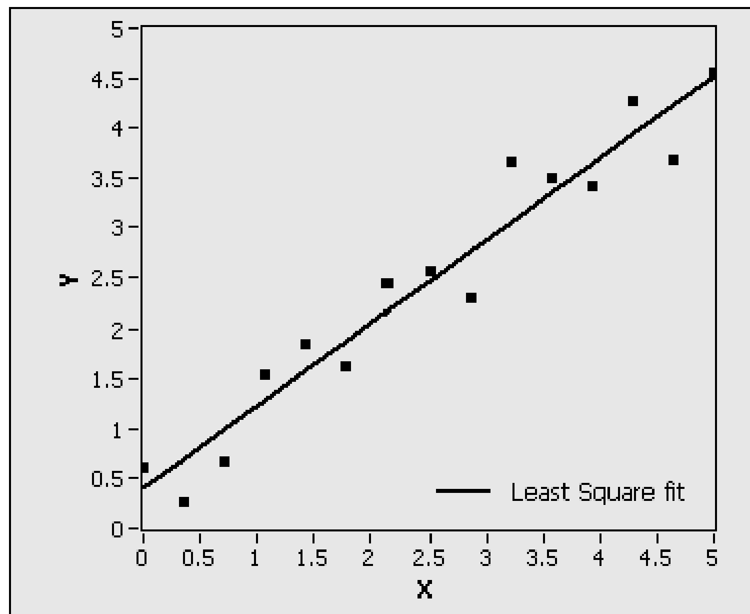
De gemodelleerde data komen nooit overeen met het vooropgestelde model, ze vertonen meetfouten. Het is belangrijk om te controleren in hoeverre deze data afwijken, en dus te kijken of het model geschikt is om de data weer te geven. Hiernaast is het ook belangrijk te weten hoe robuust deze parameters zijn. Dit geeft aan hoe groot de variantie van deze parameters zijn. Indien de variantie groot is, is er een grote kans dat de parameters van het correcte model zullen verschillen.

Een goede modelleringsmethode bestaat dus uit 3 elementen:

- Het bepalen van de beste parameters van het model
- De robuustheid van de parameters bepalen
- Controleren of het model geschikt is voor de data

2.2.6 Het kleinste kwadraat

Om de functie te vinden die het best overeenkomt met gegeven data, kan gebruik worden gemaakt van de methode van de kleinste kwadraten. Bij deze methode gaan we de som van de afstanden tussen de datapunten en het model minimaliseren. (Dirkse (2003)). Zoals



Figuur 2.7: *Door de som van de afstanden in het kwadraat tussen de punten en een lijn te minimaliseren kunnen we het functievoorschrift van de lijn vinden die het best bij deze punten past.*

eerder besproken dient er bij een goede modelering ook de goodness-of-fit (hoe geschikt het model is voor de data) bepaald te worden. Daarom gaan we de merit-functie van de standaard least-squares uitbreiden met informatie omtrent de spreiding van de fouten. Aan de hand hiervan kan de goodness-of-fit bepaald worden. De merit-functie die nu bekomen wordt, is chi-kwadraat (vergelijking 2.13). In deze vergelijking wordt van elk datapunt gekeken wat het verschil is tussen de y-waarde van de datapunten (y_i) en die van het model ($y(x_i|a_1...a_M)$). Die verschillen worden dan gekwadrateerd en opgeteld. Hierbij wordt verondersteld dat er zich enkel een meetfout voordoet op de y-coördinaat. Later zullen we ingaan op een methode die er rekening mee houdt dat er zich ook een fout op de x-coördinaat kan voordoen.

$$\chi^2 = \sum(\frac{y_i - y(x_i|a_1...a_M)}{\sigma_i})^2 \tag{2.13}$$

Nu kan naar het extremum van deze functie gezocht worden door de partiële afgeleide naar iedere parameter te bepalen. Als voorbeeld bepalen we een lijn met functievoorschrift $y = a + bx$. De chi-kwadraat functie hiervan staat in vergelijking 2.14. Deze vergelijking kan afgeleid worden naar beide parameters zodat we tot vergelijking 2.15 en vergelijking 2.16 komen. Deze worden gelijk aan nul gesteld doordat we op zoek zijn naar het extremum. Door in vergelijking 2.15 de constante termen (-2 en σ_i^2) naar het andere lid te brengen kunnen we ze laten wegvallen. Hierna kunnen we de term met a uit de sommatie halen en naar het andere lid brengen, de resulterende vergelijking is dan vergelijking 2.17. Door de term n, die het aantal datapunten aangeeft, terug naar het andere lid te brengen zien we dat de gemiddelde positie op de resulterende lijn zal liggen. De formule voor de a parameter resulteert in vergelijking 2.18.

$$\chi^2(a,b) = \sum(\frac{y_i - a - bx_i}{\sigma_i})^2 \tag{2.14}$$

$$0 = \frac{\partial\chi^2}{\partial a} = -2\sum\frac{y_i - a - bx_i}{\sigma_i^2} \tag{2.15}$$

$$0 = \frac{\partial\chi^2}{\partial b} = -2\sum\frac{x_i(y_i - a - bx_i)}{\sigma_i^2} \tag{2.16}$$

$$n * a = \sum(y_i - bx_i) \tag{2.17}$$

$$a = \bar{y} - b\bar{x} \tag{2.18}$$

Op eenzelfde manier kunnen we ook uit de vergelijking 2.16 de constante termen laten wegvallen. Na het product te bepalen van x_i met het gedeelte tussen de haken kunnen we de meest linkse term naar het andere lid brengen. Zo komen we tot vergelijking 2.19. Door de eerder gevonden vergelijking voor a in te vullen en b af te zonderen bekomen we hiervoor tenslotte ook de vergelijking voor de b factor (vergelijking 2.20).

$$\sum x_i y_i = \sum ax_i + \sum bx_i^2 \tag{2.19}$$

$$b = \frac{\sum y_i x_i - \sum x_i\bar{y}}{\sum x_i\bar{x} + \sum x_i^2} \tag{2.20}$$

Als we te maken krijgen met een systeem waarbij er fouten kunnen bestaan op zowel de x- als de y-coördinaat (zoals bij 2D-positiebepaling) moeten we de merit functie aanpassen. In de noemer moeten we een variantie-term toevoegen (vergelijking 2.21). We gebruiken nog steeds een lijn als model.

$$\chi^2(a,b) = \sum\frac{(y_i - a - bx_i)^2}{b^2\sigma_x^2 + \sigma_y^2} \tag{2.21}$$

Ook hier gaan we op zoek naar het extremum van de functie door de partiële afgeleide van beide parameters te bepalen en deze gelijk te stellen aan nul. Voor het bepalen van de a-parameter ondervinden we geen hinder van de extra term ($b^2\sigma_x^2$). Deze term is een constante zodat we ook hier op vergelijking 2.18 uitkomen.

Bij het berekenen van de partiële afgeleide van de b-parameter heeft deze term wel invloed. Het is geen constante meer maar een tweede-graadsterm. Doordat deze term in de noemer staat zal bij het afleiden de orde ervan stijgen. Op deze manier bekomen we een functie van een hogere graad dan de oorspronkelijke functie.

Omdat we niet aan de hand van de partiële afgeleide tot een formule van de b-parameter komen, gaan we het extremum van de b-parameter iteratief benaderen.

Het vinden van het lijnvoorschrift wordt op deze manier een iteratief proces. We gaan afwisselend een minimum zoeken voor a en b. Om te controleren of we het minimum goed benaderen vergelijken we het resultaat van de merit-functie van 2 opeenvolgende iteraties. Als het verschil hiertussen zo klein is dat ze nog slechts verschillen op basis van een afrondingsfout, besluiten we dat we het minimum goed genoeg hebben benaderd. Een iteratie voor het vinden van de parameters bestaat uit volgende stappen:

- Bereken a aan de hand van vergelijking 2.18. De b die hierin wordt gebruikt, is de laatst berekende waarde hiervoor.
- Benader het minimum van de merit-functie door enkel b te wijzigen.
- Vergelijk het resultaat van de merit-functie met het resultaat uit de vorige iteratie en besluit hieruit of het iteratieve proces mag worden stopgezet.

Hoofdstuk 3

Analyse van de broncode

3.1 Inleiding

In dit hoofdstuk bespreken we het algoritme van waaruit we vertrokken zijn. Dit algoritme is ontwikkeld door Essensium, en is in staat om een positiebepaling te doen van mobiele nodes aan de hand van afstandsmetingen tussen basisstations en mobiele nodes, en mobiele nodes onderling. In latere hoofdstukken komt aan bod welke uitbreidingen we gemaakt hebben om de resultaten van dit algoritme te verbeteren.

3.2 De probleemstelling

3.2.1 Positiebepaling

Het algoritme waarvan we vertrokken zijn is ontwikkeld door Essensium. Er zijn 2 typen nodes:

- basisstations: hiervan is de positie gekend en op voorhand vastgelegd. Een basisstation zal afstandsmetingen doen naar de mobiele nodes.
- mobiele nodes: hiervan gaan we positie bepalen aan de hand van afstandsmetingen, hetzij vanuit basisstations, hetzij vanuit andere mobiele nodes.

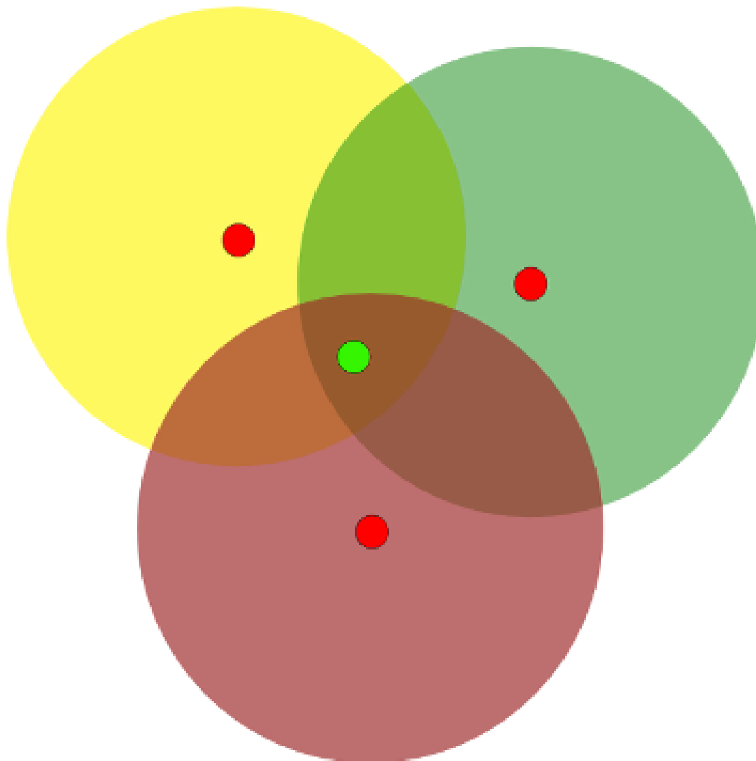
We kunnen onderscheid maken tussen 2 soorten van positiebepaling:

- ”Cooperative localization”: hierbij wordt er naast de metingen vanuit basisstations tot de mobiele nodes, ook rekening gehouden met metingen tussen de mobiele nodes onderling. Hierbij bepalen we de posities van alle mobiele nodes op hetzelfde moment.
- Klassieke positiebepaling: hierbij maken we geen gebruik van afstandsmetingen tussen de mobiele nodes. De relatieve posities van de nodes zijn niet relevant.

3.2.2 Beperkingen van het systeem

Om de posities van de mobiele nodes te bepalen dienen we een aantal vergelijkingen te combineren. De vergelijkingen worden opgebouwd uit de afstandsmetingen. Hierbij dienen we er rekening mee te houden dat de afstandsmetingen van een meetfout bevatten.

Het is uitgesloten om op een directe methode tot een exacte numerieke oplossing te komen. Door de aanwezige meetfouten is de kans groot dat deze zelfs niet bestaat (figuur 3.1). In plaats daarvan dient er via een iteratieve methode gewerkt te worden. Door in elke iteratie een stap dichter naar de oplossing te werken, bekomen we de oplossing die het best overeenkomt met de gemaakte afstandsmetingen, maar niet aan elke afstandsmeting hoeft te voldoen. We benaderen de oplossing.



Figuur 3.1: *Doordat de afstandsmetingen naar de centrale node een meetfout bevatten (grote cirkels) is er geen oplossing (geen punt waar de drie afstandsmetingen snijden). We zoeken een oplossing die het best met de afstandsmetingen overeenkomt.*

Bij het bepalen van de posities dienen we een maatstaf te hanteren die aangeeft of een oplossing minder goed of beter is dan een andere oplossing. In de statistiek gebruikt met de term "function of merit" voor dit soort functie. Een oplossing bestaat uit een verzameling posities die overeenkomt met de berekende posities van de mobiele nodes. Als maatstaf gebruiken we een formule (zie formule 3.3) die het verschil maakt van de afstand tussen de berekende nodes volgens Pythagoras (zie formule 3.1) en de afstand die tussen de nodes is gemeten (de afstandsmeting die we als invoer van het algoritme gebruiken). Het verschil tussen beiden wordt gekwadrateerd (zodat elk verschil tot een positieve waarde wordt herleid) en wordt aangevuld met een waarde die zorgt dat we tot een goede oplossing

kunnen komen. Ervaring heeft uitgewezen dat de nodes de neiging hebben om zich van elkaar te verwijderen en zo vast komen te zitten in een lokaal minimum. Door de nodes aan het begin van het algoritme bij elkaar te houden en geleidelijk naar buiten toe laten gaan komen we tot het globale minimum. Er wordt vier keer gewerkt tot een oplossing, respectievelijk met een waarde voor 'ssigma' gelijk aan 3, 2.5, 1 en 0.

Voor al de metingen die hebben plaats gevonden, voeren we deze formule uit. De som hiervan is de gehanteerde function of merit. Verder in de tekst zullen we deze benoemen als "de totale fout". Hoe kleiner het resultaat is van deze functie, hoe beter we de oplossing benaderen.

$$d = pyth(a, b); \tag{3.1}$$

$$r = d^2 - Meting^2; \tag{3.2}$$

$$sum = \sum (r^2 + 8 * ssigma^2 * d^2); \tag{3.3}$$

3.3 Het iteratieve proces

3.3.1 Verkleinen van de totale fout

Zoals in de vorige paragraaf werd beschreven werken we via een iteratief proces om de posities van de mobiele nodes te bepalen. In elke stap voeren we een verschuiving uit van al de mobiele nodes, zodat de totale fout kleiner wordt en we dus een betere combinatie van posities verkrijgen. Deze stap wordt herhaald, tot de verbetering die een stap met zich mee brengt, zeer klein wordt. Het is niet mogelijk een oplossing te vinden die exact overeenstemt met al de afstandsmetingen, gezien deze een meetfout bevatten.

3.3.2 De gradiënt van de mobiele nodes

Om te bepalen in welke richting een mobiele node een verplaatsing moet ondergaan, maken we gebruik van de gradiënt van de mobiele node. Om tot een minimale totale fout te komen moet de positie van elke mobiele node overeenstemmen met de afstandsmetingen die gebeurd zijn. Gezien vanuit één mobiele node kunnen we een functie opstellen die aangeeft hoe goed deze node overeenstemt met de afstandsmetingen waarin deze node betrokken is. De gradiënt die we bepalen geeft de steilste helling aan van deze functie.

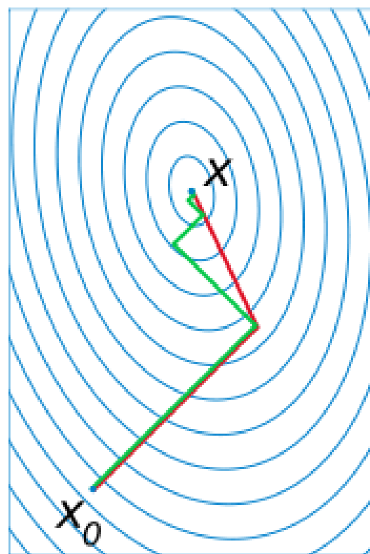
Het bepalen van de gradiënt gebeurt voor elke mobiele node afzonderlijk. Dit zorgt niet enkel dat elke mobiele node in de meest optimale richting zal verschuiven, maar ook dat voor elke mobiele node de gradiënt een verschillende grootte heeft (de gradiënt is een vector) die aangeeft in welke mate de verschuiving moet plaats vinden ten opzichte van andere mobiele nodes. Een node die dicht bij zijn eindpositie is, moet een kleinere verschuiving ondergaan dan een node die ver van zijn einddoel verwijderd is.

De methode waarop de gradiënt bepaald is, bestaat uit het maken van een som van vectoren. Voor elke mobiele node wordt elke afstandsmeting waarin de node betrokken

is, vergeleken met de afstand tussen de berekende posities (aan de hand van Pythagoras). Indien de afstandsmeting groter is, wil dit zeggen dat de mobiele nodes te dicht bij elkaar staan. We bekomen een vector die de mobiele node verder weg zal verschuiven over de lijn tussen beide nodes. Als de nodes te ver van elkaar verwijderd zijn, krijgen we een vector die de nodes naar elkaar toe duwt. Het is alsof er veren met een lengte van de afstandsmeting tussen de nodes bevestigd zijn. Als de nodes te kort bij elkaar staan duwt deze de nodes uit elkaar en als deze te ver van elkaar staan worden de nodes naar elkaar toe getrokken. Door dit voor al de afstandsmetingen te doen waarin de mobiele node betrokken is, kunnen we al de bekomen vectoren sommeren en bekomen de gradiënt van deze mobiele node.

3.3.3 De toegevoegde gradiënt

Om het aantal iteraties te beperken maken we gebruik van een toegevoegde gradiënt. Dit is een aangepaste versie van de gradiënt, die rekening houdt met eerder gevolgde richtingen. Zoals in figuur 3.2 is te zien komen we via deze methode in een kleiner aantal stappen tot het gewenste minimum. De gebruikte formule om de toegevoegde gradiënt te bepalen is opgesteld door Polak-Ribiere.



Figuur 3.2: *De gradiënt aanpassen aan de hand van de gradiënt uit voorgaande stappen leidt met minder iteraties tot de oplossing. Aan de hand van de cirkels/elipsen worden de hoogtelijnen voorgesteld. Bij gebruik van het gradiënt (loodrecht op de hoogtelijnen) zijn er vijf stappen nodig om het minimum te bereiken. Bij gebruik van de toegevoegde gradiënt (aanpassen van de richting aan de hand van eerder gekozen richtingen) hebben we hier slechts twee stappen voor nodig.*

3.3.4 Lijnminimalisatie

We hebben de gradiënt voor elke node bepaald, aan de hand hiervan hebben we de toegevoegde gradiënt berekend. De toegevoegde gradiënt geeft de richting aan waarin elke node zal verschuiven. Nu moeten we nog bepalen hoeveel elke node moet verschuiven. De grootte van de verschuiving bestaat uit het product tussen de grootte van de vector en een verschuivingswaarde die we bepalen. Doordat de toegevoegde gradiënt-vector voor elke node een verschillende grootte heeft, zal ook elke node over een verschillende afstand verplaatst worden.

Doordat we de verschuiving van elke node laten gebeuren over een vastgelegde vector, herleidt het bepalen van de verschuivingswaarde zich tot een 1-dimensionaal probleem. We kunnen dan ook beroep doen op lijnminimalisatie. Hoe dit werkt komt zo dadelijk aan bod.

Het vinden van de oplossing van het probleem kan voorgesteld worden door volgende pseudo-code:

```
1  VorigeTotaleFout = 10000000; // heel groot zetten
2  while(true){
3
4     // conjugate gradient bepalen
5     ForEach(mobiele_node){
6        gradient = BepaalGradient();
7        conjugate = BerekenConjugateGradient(gradient);
8        ConjugateGradient_Lijst.add(conjugate);
9     }
10
11    // verschuivingswaarde bepalen
12    verschuivingswaarde =
         DoeLijnMinimalisatie(ConjugateGradient_Lijst);
13
14    // verschuiving uitvoeren
15    ForEacht(mobiele_node){
16       DoeVerplaatsing(mobiele_node, verschuivingswaarde,
            ConjugateGradient_lijst.find(mobiele_node));
17    }
18
19    // kijken of we de iteratie kunnen beeindigen
20    TotaleFout = BerekenTotaleFout(Posities_Mobiele_Noden);
21    if(TotaleFout - VorigeTotaleFout < 0.001)
22       break; // stop de iteratie
23
24    // klaarzetten voor de volgende iteratie
25    VorigeTotaleFout = TotaleFout;
26 }
```

In het oorspronkelijke algoritme is de berekening van de verschuivingswaarde, en uitvoering van de verschuiving en de berekening van de fout geïntegreerd in de functie voor de lijnminimalisatie. De pseudo-code geeft echter duidelijker de nodige stappen aan.

De functie die we dienen te minimaliseren is van een ongekende graad die afhankelijk is van het aantal afstandsmetingen dat gebruikt wordt. Hierdoor kunnen we niet met zekerheid bepalen of er wel een minimum bestaat. Om te voorkomen dat we nodeloos zoekwerk doen, gaan we eerst een domein bepalen waarbinnen we met grote zekerheid kunnen zeggen dat er een minimum zal liggen. Dit domein kan bepaald worden door 3 punten te vinden waarvoor het middelste punt lager (bekomt een kleinere functie waarde) ligt dan de andere twee. Geheel zeker kunnen we dan nog niet zijn van het bestaan van het minimum gezien we rekening moeten houden dat de functie discontinu kan zijn. Om dit probleem te vermijden gaan we het aantal iteraties waarin we zoeken naar een minimum beperken (bemerk: deze iteratie is niet dezelfde als degene die we gebruiken voor positiebepaling. Hier zoeken we naar een minimum over een vector).

Voor het begrenzen van een zoekdomein waarbinnen een minimum ligt, proberen we gebruik te maken van parabolische extrapolatie. Deze techniek bestaat uit het berekenen van een minimum van een parabool waarvan drie gekende punten een element zijn. Indien deze methode een gunstig resultaat geeft komen we in een beperkt aantal iteraties aan het begrensde domein, indien dit niet zo is doen we beroep op ”default magnification”, om het zoekdomein uit te bereiden. De pseudo-code voor het begrenzen ziet er als volgt uit:

```
1  Bracket(&ax, &bx, &cx){
2   // ax en bx zijn begin waarden om de grenzen te beginnen
       zoeken, de grenzen komen terecht in ax en cx
3   GOLD = 1.618034;
4   fa = functie(ax); fb = functie(bx);
5   if(fb > fa){
6   // we willen dat ax een hoger resultaat geeft dan bx
7   // Zo zal ax waarschijnlijk een grens vormen van het
       domein
8     wissel(ax, bx);
9     wissel(fa, fb);
10  }
11  cx = bx + GOLD*(bx-ax); // we bepalen een cx om mee
       verder te werken
12  fc = functie(cx);
13
14  while(fb > fc){ // zolang we geen grens hebben gevonden
15    u = ParabolicExtrapolation(ax,bx,cx); // bepaal u via
         parabolic extrapolation
16    uimit = bx + GLIMIT*(cx-bx); // We leggen een grens om
         het zoekdomein in te bepalen
17
18    if(u tussen bx & cx){
```

```
19          if(fu < fc){ // minimum tussen bx en cx
20              ax = bx; fa = fb; // ax wordt op de grens gelegd
21              bx = u; fb = fu; // bx wordt de lager gelegen
                    waarde
22                  // cx ligt al op de andere grens
23              return;
24          }
25          else if(fu > fb){
26          // gezien ax al een hoger resultaat gaf dan bx
27          // wil dat zeggen dat er een minimum tussen
28          // ax en u ligt
29              cx = u;  fc = fu; // cx wordt een grens, ax is al
                     een grens
30                 // bx is al de lager gelegen waarde
31          return;
32          }
33          else { // parabolic extrapolation werkt niet
34              u = cx + GOLD*(cx-bx); // domein verder
                    uitbereiden weg van ax
35              fu = functie(u);
36          }
37      }
38      else if(u tussen ulimit & cx of verder){
39          // u wordt op ulimit gezet
40          u = ulimit; fu = functie(u);
41      }
42      else{ // parabolic extrapolation heeft geen zin
43          u = cx + GOLD*(cx-bx); fu = functie(u);
44      }
45   // grenzen doorschuiven
46      ax = bx; bx = cx; cx = u;
47      fa = fb; fb = fc; fc = fu;
48   }
49  }
```

Nu we een zoekdomein voor het minimum hebben bepaald, moeten we binnen dit domein het minimum bepalen. Het vinden van het minimum gebeurt door iteratief stappen te zetten die ons steeds dichter bij het minimum brengen. Gezien de functie voor het bepalen van de gradiënt al beschikbaar is, kunnen we hier ook beroep op doen bij het vinden het het minimum. De gradiënten die we berekenen zijn voor elke node afzonderlijk bepaald. Voor de lijnminimalisatie willen we echter kunnen beschikken over de afgeleide van de functie, zodat we deze gradiënten nog moeten verwerken hiernaar. Door te kijken naar de tekens van de gradiënt vector na de stap (de x en y waarde na de verschuiving) en die van voor de stap, kunnen we bepalen of de volgende stap in dezelfde richting moet

gebeuren (de tekens zijn gelijk) of dat de genomen stap te groot was en dus de volgende stap in tegengestelde richting moet gebeuren (tekens zijn gewisseld). De meest optimale situatie is als voor elke node de berekende gradiënt na de verschuiving loodrecht op de richtingsvector staat.

Het bepalen van de afgeleide van de totale fout over de lijn kan door volgende pseudo-code worden voorgesteld:

```
1 double df1dim(double verschuiving){
2    NieuweGradienten = BepaalGradienten(Verschoven_Punten);
3    ForEach(N_grad = NieuweGradienten && O_Grad =
        OudeGradienten){
4       afgeleide += N_grad.x * O_Grad.x + N_grad.y *
           O_Grad.y;
5    }
6    return afgeleide;
7 }
```

Door de overeenkomstige x- en y-waarde van de gradiënt van voor en na de verschuiving te vermenigvuldigen, detecteren we of de tekens verschillen (het product is in dat geval negatief, wat wil zeggen dat de verschuiving te groot is voor deze node). Deze bewerking voeren we voor elke node uit en sommeren we. Indien het resultaat van deze som negatief is, betekent dit dat we een te grote verplaatsing hebben uitgevoerd (het teken van de gradiënt van het merendeel van de nodes is gewisseld) en dus de verplaatsing moet verkleinen. In het perfecte geval wordt de som gelijk aan nul (de nieuwe gradiënt van de nodes is gemiddeld gezien loodrecht geplaatst op de vorige vector), en hebben we het minimum gevonden. Net als bij een afgeleide geeft een 0-resultaat een minimum aan.

De gebruikte methode om een minimum te bepalen is die van Brent. We maken gebruik van een implementatie die gebruik maakt van de afgeleide van de functie (de 'd' staat voor 'derivative').

Brent maakt gebruik van twee technieken, enerzijds probeert hij nulpunten te vinden van de afgeleide aan de hand van de Secant methode, en als dit niet werkt, wordt er teruggevallen op bisection als meer robuuste methode. Het vinden van een minimum is niet enkel het vinden van de nulpunten van de afgeleide, dit kan net zo goed tot een maximum leiden. We willen ook dat steeds binnen de afgebakende grenzen wordt gebleven. De kennis van de afgeleide zal dus in de hoofdzaak gebruikt worden om te achterhalen of in het interval (a,x) of (x,b) moet gewerkt worden, waarbij a en b de grenzen zijn en x het lager gelegen punt.

In het algoritme wordt er gebruik gemaakt van 6 punten binnen het interval:

- a: de ondergrens van het zoekdomein, deze wordt in de loop van het programma aangepast
- b: de bovengrens van het zoekdomein, deze wordt in de loop van het programma aangepast

- u: het laatste geëvalueerde punt
- v: de vorige waarde van w
- w: het punt dat de voorlaatste kleinste functiewaarde aangaf.
- x: het punt dat laagste functiewaarde tot dusver oplevert

Het algoritme zet eerst de gevonden grenzen om naar lokale variabelen a en b. Na een initialisatie van de 6 gebruikte variabelen kan de iteratie om het minimum te vinden van start gaan. Zoals eerder vermeld gebeurt dit in een iteratie met een beperkt aantal stappen om een oneindige lus te vermijden. We stellen een tolerantie in, deze bestaat uit 2 waarden:

- tol1: dit is de minimale stap die zal worden gezet in de iteratie. Deze waarde wordt bepaald aan de hand van formule 3.4.

  $$tol1 = TOL * abs(x) + ZEPS; \tag{3.4}$$

  TOL en ZEPS zijn respectievelijk gedefinieerd op 0.0002 en 0.00000000001. ZEPS heeft als doel te vermijden dat er een deling door nul ontstaat doordat het nulpunt van de functie exact nul zou zijn. Door het gebruik van een eindige precisie is er een round-off error op decimale waarden. Een vuistregel verteld ons dat het geen zin heeft om waarden te evalueren korter dan $\sqrt{\varepsilon}$ keer de centrale waarde. $\sqrt{\varepsilon}$ is de floating point precisie van het systeem.

- tol2: dit geeft de minimale afstand tussen a en b (de grenzen) aan en wordt bepaald aan de hand van formule 3.5. Het middelpunt ligt minimaal tol1 van de grenzen.

  $$tol2 = 2.0 * tol1 \tag{3.5}$$

We bepalen het midden van het interval (gemiddelde van a en b) en controleren of het verschil tussen x en het midden niet te klein is (regel 15 in dBrent algoritme), en indien dit zo is is het minimum gevonden op x.

Daarna controleren we of de vorige stap groot genoeg is om aan de minimum stapgrootte te voldoen (tol1 is gewijzigd aan het begin van de iteratie). Op lijn 33 in het algoritme dBrent gaan we kijken of de stap die volgt uit de Secant methode kleiner is dan de helft van de voorlaatste stap. Als de voorlaatste stap echter al niet voldoet aan de minimum te nemen stap, heeft het geen zin om de Secant methode nog uit te voeren aangezien we de berekende stap toch niet gaan aanvaarden. In dit geval maken we gebruik van de bisection methode om de nieuwe stap te bepalen (we maken nergens een vergelijking met de voorlaatste stap). Als de voorlaatste stap echter groter is dan de minimaal te nemen stap (tol1), maken we gebruik van de Secant methode in een poging een correcte stap te vinden (deze moet dan nog aan 3 voorwaarden voldoen om gebruikt te worden).

De Secant methode maakt telkens gebruik van 2 punten om de stap te bepalen. Hier hebben we echter 3 punten die we kunnen gebruiken (x, v en w) zodat we 2 mogelijke stappen kunnen bepalen die telkens x gebruiken. Voordat we de berekening van de stappen

doen, initialiseren we de stappen op het dubbele van de domein-breedte zodat deze zeker niet als stap kunnen gebruikt worden. Dit is nodig omdat we de berekening van de stappen niet willen doen als v of w samenvalt met x (dit zou leiden tot een deling door nul).

$$d1 = (w - x) * dx/(dx - dw); \tag{3.6}$$

$$d2 = (v - x) * dx/(dx - dv); \tag{3.7}$$

Om te controleren of een stap bruikbaar is moet deze aan 3 vereisten voldoen:

- De stap mag niet leiden tot een waarde die buiten de grenzen ligt.
- De stap moet een invers teken hebben van dx, zodat de stap in de juiste richting zal plaatsvinden.
- De stap moet groter zijn dan de minimum stap, indien dit niet het geval is nemen we de minimum stap.

Als voor beide stappen aan de richtlijnen voldaan wordt, zal de kleinste gebruikt worden (regel 23 in algoritme dBrent). Indien geen van beide voldoet, doen we beroep op de bisection methode (regel 37 in bisection methode). Nu rest ons nog het nemen van de stap, deze kan zowel door bisection als door de Secant methode bepaald zijn. Indien we merken dat na de stap de functiewaarde van het gevonden punt hoger ligt dan het vorige gevonden punt is het minimum gevonden (het vorige gevonden punt). Tijdens het algoritme kijken we naar de voorlaatste stap in plaats van de laatste stap om te kijken of de huidig berekende stap aanvaardbaar is. Het achterliggende idee is dat een misstap mag gebeuren, als deze later tot een beter resultaat zou kunnen leiden.

Aan het einde van elke iteratie schuiven we de grenzen naar binnen zodat het zoekdomein kleiner wordt voor de volgende iteratie.

```
1  dBrent(ax, bx, cx){ //de grenzen en het lager gelegen punt
2      x = bx;
3      if(ax < cx){
4          a = ax; b = cx;
5      }
6      else{
7          a = cx; b = ax
8      }
9
10     InitialiseVars(); // de 6 variabelen op hun waarde
           instellen
11
12     for(int iter = 0; iter < ITMAX; iter++){
13         xm = gem(a,b);
14         tol1 = TOL*abs(x)*ZEPS;
15         tol2 = 2*tol1;
```

```
16        if(abs(x-xm) <= tol2-0.5*(b-a){
17            return x;
18        }
19        if(abs(e) > tol1){
20            d1=d2=2*(b-a);
21            if(dw != dx) d1 = Secant(w,x);
22            if(dv != dx) d2 = Secant(v,x);
23
24            if(d1 && d2 bruikbaar){
25                d = min(d1,d2);
26            }
27            else if(d1 bruikbaar)
28                d = d1;
29            else
30                d = d2;
31            olde = e; //doorschuiven van de stappen
32            e = d;
33            if(abs(d) < abs(0.5*olde)){ // d is kleiner dan
                 de helft van voorlaatste stap
34                u = x + d;
35                if(u-a < tol2 || b-u < tol2){
36                    d = SIGN(tol1,xm-x); // minimale stap
                          gebruiken
37                }
38                else{ // stap is te groot
39                    BepaalInterval(dx); // is interval (a,x) of
                           (x,b)
40                    d = 0.5*e; // bisection
41                }
42            }
43        }
44        else{ // abs(e) is te klein
45            BepaalInterval(dx);// is interval (a,x) of (x,b)
46            d = 0.5 * e; // bisection
47        }
48
49        if(abs(d) > tol1)
50            ZetStap(d);
51        else{
52            ZetMinimumStap(tol1);
53        }
54
55        VerschuifGrenzen();
56    }
57    printf("te veel iteraties\n");
```

```
58 }
```

3.4 Conclusie

In dit hoofdstuk hebben we een analyse gedaan van het algoritme waarvan we zijn vertrokken. We zijn gestart met de probleemstelling, namelijk de bepaling van de posities van mobiele nodes aan de hand van afstandsmetingen. Vervolgens zijn we ingegaan op het iteratieve proces dat hiervoor gebruikt wordt en hebben we het multi-dimensionele probleem herleid tot een probleem in 1 dimensie. Tenslotte hebben we de uitwerking van het 1-dimensionele probleem uitgewerkt.

Hoofdstuk 4

Randvoorwaarden

4.1 Waarom randvoorwaarden gebruiken

Dit hoofdstuk behandelt het gebruik van randvoorwaarden. Aan de hand van randvoorwaarden geven we extra informatie (nieuwe voorwaarden voor de oplossing van het algoritme) zodat we een nauwkeuriger resultaat bekomen. Indien er een tekort aan informatie is om een correcte plaatsbepaling te doen (bijvoorbeeld slechts twee metingen in een 2D omgeving) kan een bijkomende randvoorwaarde zorgen dat er toch een correcte plaatsbepaling gebeurt.

Eerder hebben we al aangehaald dat het noodzakelijk is om de positie van de base-stations zo correct mogelijk te bepalen. Elke fout hierbij zal zich voortzetten in de afstandsmetingen die vanuit dit base-station gebeuren, en zo ook in de positiebepaling die afhankelijk is van deze afstandsmetingen. Stel dat we instellen dat een base-station zich op positie (10,0) bevindt, maar dat deze zich in werkelijkheid op positie (11,0) bevindt, dan zal elke afstandsmeting die vanuit dit base-station gemaakt is behandeld worden alsof deze vanaf positie (10,0) is gemaakt. Dit resulteert in een foutieve plaatsbepaling van de mobiele nodes.

Het bepalen van de positie van de base-stations (de setup) kan op twee manieren gebeuren:

- Manueel: we gaan de posities manueel opmeten. Deze methode is haalbaar voor een setup van enkele base-stations, maar is niet meer haalbaar in opstellingen waar meer dan 10 base-stations in betrokken zijn. In de realiteit kunnen we ook te maken hebben met opstellingen waar meer dan 50 base-stations in betrokken zijn, waarbij deze bevestigd zijn op een hoogte van 20-30 meter.

- Berekenen: we kunnen het algoritme dat we gebruiken voor het positioneren van de mobiele nodes ook gebruiken voor de plaatsbepaling van de base-stations. De posities van de base-stations resulteren na enkele seconden uit het algoritme, zodat de setup-tijd drastisch wordt ingekort.

We gaan het algoritme voor de positiebepaling van de mobiele nodes ook inzetten voor de positiebepaling van de base-stations bij de setup. Hierbij beschouwen we de base-

stations waarvan we de positie moeten bepalen, als mobiele nodes. Doordat over base-stations extra informatie kan verzameld worden, zijn we in staat om de resultaten van het algoritme nauwkeuriger te maken. Deze informatie wordt vertaald in de vorm van randvoorwaarden. Enkele mogelijke randvoorwaarden zijn:

- Het base station is gelegen tegen een muur (en dus op een lijn). Uit het grondplan kan afgeleid worden welke vergelijking deze lijn heeft.
- Het base station vormt een lijn met enkele andere base stations. Het voorschrift van de lijn kan gekend zijn (dan hebben we te maken met hetzelfde type randvoorwaarde als in het eerste voorbeeld) of ongekend zijn. In het laatste geval zal aan de hand van de berekende posities bepaald worden wat het voorschrift van de lijn is.
- Twee nodes liggen op een gekende afstand van elkaar.

4.2 De toevoeging van randvoorwaarden

Deze randvoorwaarden gaan we integreren waarbij we de aanpassingen zo beperkt mogelijk houden in het bestaande algoritme. Zo zorgen we ervoor dat het eenvoudig blijft om deze randvoorwaarden ook in nieuwere releases van het algoritme te gebruiken. De aanpassingen die we aanbrengen, worden besproken in hoofdstuk 5.

Voor het opstellen van de randvoorwaarden zelf, houden we met twee zaken rekening:

- We willen dat het eenvoudig blijft om nieuwe types randvoorwaarden te definieren. Hiervoor gaan we elke randvoorwaarde opbouwen vanuit een basisklasse ”Constraint”.
- Het algoritme mag niet afhankelijk zijn van het aantal randvoorwaarden dat we gebruiken. Hierbij willen we ook dat de gebruikte randvoorwaarden vlot kunnen aangesproken worden, zodat hierbij geen groot performantie-verlies optreedt.

4.2.1 De klasse ”Constraint”

Het model dat we gebruiken voor elke randvoorwaarde, is bepaald in de klasse ”Constraint”. Elke randvoorwaarde die we definiëren zal een child-klasse hiervan zijn. Zo zorgen we niet enkel dat elke randvoorwaarde dezelfde opbouw heeft, maar ook dat het algoritme onafhankelijk van het type van de gebruikte randvoorwaarden werkt. De klasse ”Constraint” ziet er als volgt uit:

```
1 class Constraint
2 {
3 public:
4         bool virtual initialise(Positions &P);
```

```
5          char virtual GiveType();
6          virtual ~Constraint(){};
7          bool virtual equals(Constraint *con);
8          void virtual GiveParams(double params[]);
9          void virtual Execute(Positions &P);
10 };
```

Elke randvoorwaarde die we later definiëren, erft over van deze klasse. Door de aanwezige functies "virtual" te maken zal at runtime bepaald worden uit welke child-klasse de functies moeten worden uitgevoerd. Elke child-klasse (en dus de randvoorwaarde) zal een eigen versie van deze functies bevatten. Bij het opstarten van het algoritme worden al de randvoorwaarden gedeclareerd als type "Constraint" (dus van de basisklasse), maar we koppelen hier een randvoorwaarde van een ander type (bijvoorbeeld Line) aan. We maken gebruik van polymorfisme. Bij het begin van het algoritme hoeft dus niet bekend te zijn welke randvoorwaarden er gebruikt worden, dit wordt at runtime bepaald. Een voorbeeld ter verduidelijking:

```
1 Constraint *con;
2 con = new Line();
3 std::cout << con.GiveType() << std::endl;
```

In dit voorbeeld zal de functie die in de Line klasse is gedefinieerd worden uitgevoerd, hoewel het type van de pointer "Constraint" is. Belangrijk om hier op te merken is dat at runtime bepaald wordt dat het hier over een lijn gaat. Indien de functie "GiveType" niet virtual is gedefinieerd, zal bij compilatie worden vastgelegd welke functie er moet worden uitgevoerd. Voor de compiler is het onmogelijk om te weten welk object er aan "con" zal worden toegekend (iets dat at runtime gebeurt). In dit geval zal de functie uit de Constraint-klasse worden uitgevoerd (wat niet gewenst is).

De functie "initialise" zorgt dat de punten die als parameter worden meegegeven, voldoen aan de randvoorwaarde. Indien de randvoorwaarde bijvoorbeeld een lijn is waar de punten op moeten liggen, zal "initialise" de posities van de nodes aanpassen zodat deze ook werkelijk op de lijn liggen.

De functie "GiveType" en "GiveParams" spreken voor zich. Ze geven respectievelijk het type randvoorwaarde (bestaande uit een letter) en de parameters ervan terug. Bij het aanmaken van een randvoorwaarde dienen deze gedefinieerd te worden. Indien we bijvoorbeeld een lijn willen gebruiken als randvoorwaarde, geven we als type randvoorwaarde de letter 'L' en als parameters geven we twee waarden die respectievelijk de a en de b in het lijnvoorschrift ax+b voorstellen.

De functies "GiveType" en "GiveParams" worden gebruikt in de functie "equals", die controleert of het opgegeven object gelijk is aan het oproepende object. We dienen er echter

rekening mee te houden dat het aantal parameters van verschillende randvoorwaarden kan verschillen. Zo beschikt een Line-object over twee parameters, die het functievoorschrift aangeven, terwijl een ULine-object over slechts 1 parameter beschikt dat een unieke identificatie van de randvoorwaarde voorstelt. Eerst dienen we GiveType op te roepen zodat we het type van beide randvoorwaarden kunnen vergelijken. Als deze hetzelfde zijn, zijn we ook zeker dat het aantal parameters overeen zal komen. Dan kunnen we zonder risico GiveParams oproepen en deze parameters vergelijken.

Het vergelijken van randvoorwaarden wordt gebruikt bij het invullen van de constraint matrix (4.2.2). Zo vermijden we dat er duplicaten zijn van randvoorwaarden.

De functie ”Execute” speelt in op de richtingsvector die gebruikt wordt. Zo kunnen we deze aanpassen aan het type en de parameters van de randvoorwaarde. Bij een randvoorwaarde van het type ”Line” kunnen we er bijvoorbeeld voor zorgen dat de mobiele nodes niet kunnen afwijken van de gedefinieerde lijn, en zodoende altijd aan de randvoorwaarde voldoen.

”Execute” komt echter niet in elke randvoorwaarde van pas, zo zijn er randvoorwaarden die geen informatie oplevert in verband met de te volgen richting. Een voorbeeld hiervan is de randvoorwaarde ”Afstand” waarbij een vastgelegde afstand tussen twee nodes bestaat. We hebben geen informatie over hoe deze nodes ten opzichte van elkaar moeten liggen, en kunnen dus geen aanpassingen aan de richtingsvectoren doen zodat de plaatsbepaling nauwkeuriger kan gebeuren. Bij dergelijke randvoorwaarden is de ”Execute” functie leeg en maken we dus enkel gebruik van de ”initialise” functie.

4.2.2 De klasse ”ConstraintMatrix”

Voor het bijhouden van randvoorwaarden in de loop van het algoritme gebruiken we een combinatie van gelinkte lijsten en circulaire lijsten. Deze dynamische structuren zorgen dat het aantal gebruikte randvoorwaarden eenvoudig kan worden uitgebreid. Hierbij moet aan vier eisen voldaan worden:

- Alle verschillende Constraints moeten doorlopen kunnen worden
- Al de verschillende nodes waar een randvoorwaarde op van toepassing is moeten eenvoudig te doorlopen zijn.
- De verschillende randvoorwaarden die op een bepaalde node van toepassing zijn moeten eenvoudig te doorlopen zijn.
- De verschillende nodes waarop dezelfde randvoorwaarde van toepassing is moeten eenvoudig te doorlopen zijn.

De structuur ”ConstraintN” die gebruikt wordt als een element van de constraint matrix, heeft zes variabelen. De ”ID” houdt de ID bij van de node, ”con” is een pointer naar de randvoorwaarde die op deze node van toepassing is. Gebruik maken van een pointer voorkomt dat eenzelfde randvoorwaarde meermaals gedefinieerd is. Er zijn dan nog vier pointers om de gelinkte en circulaire lijsten te implementeren (”nextID’ en ”nextCon”

zijn van toepassing voor gelinkte lijsten, "nextI" en "nextC" vormen de pointers van de circulaire lijsten).

```
struct ConstraintN{
        int ID;
        Constraint *con;
        ConstraintN *nextID;
        ConstraintN *nextCon;
        ConstraintN *nextI;
        ConstraintN *nextC;
};
```

Omdat er slechts één interface is voor de constraint matrix, voorzien we een klasse waarvan in het algoritme slechts 1 object wordt gemaakt.

```
class ConstraintMatrix
{
public:
        ConstraintMatrix(){
                head = NULL;
        }
        void ResetPointer(){
                pointer = head;
        }
        void AddConstraintN(int ID, Constraint *con);
        ConstraintN *GetIDsByConstraint(Constraint *con);
        ConstraintN *GetConstraintsByID(int ID);
        void FreeMatrix();
        void PrintMatrix();
        ConstraintN *Overloop(Constraint *&con); // geeft
           een LL van ID's terug die aan de con (by ref
           gewijzigd) moeten voldoen

private:
        ConstraintN *head;
        ConstraintN *pointer;
        ConstraintN* AddToList(ConstraintN *staart,
           ConstraintN *wijzer,char c);
        ConstraintN *GiveIDsByConstraintN(ConstraintN
           *wijzer);
};
```

Naast "head" is er nog een pointer naar een ConstraintN (genaamd "pointer"). Deze pointer wijst telkens naar het eerstvolgend onbehandeld element. Het gebruik ervan zal duidelijker worden aan de hand van de implementatie van de functie "Overloop":

```
ConstraintN *ConstraintMatrix::Overloop (Constraint *
   &con)
{
  ConstraintN *top; // zal de top bevatten van de LL die
     wordt terug gegeven
  if (pointer != NULL)
    {
      top = GiveIDsByConstraintN (pointer); //geeft een LL
          terug
      con = pointer->con; //con by ref wijzigen
      pointer = pointer->nextCon;
      return top;
    }
  else
    return (ConstraintN *) NULL;
}
```

We bemerken dat aan deze functie een pointer van het type "Constraint" by reference wordt meegegeven. Hierdoor kan binnen deze functie de pointer gewijzigd worden zodat deze na terugkeer van de functie naar een andere randvoorwaarde wijst. Dit is uitvoer die we krijgen naast de pointer naar een gelinkte lijst die de uitvoerwaarde van de functie is. Voor de eerste oproep verwijst "pointer" naar het eerste element van de gelinkte lijst van randvoorwaarden. Na de eerste oproep is de eerste randvoorwaarde behandeld en wijst de interne variabele "pointer" van het constraint matrix-object naar de tweede randvoorwaarde.

De functie "Overloop" geeft als resultaat een pointer naar het begin van een gelinkte lijst van de verschillende nodes waarop deze randvoorwaarde van toepassing is. We geven echter niet een verwijzing naar de data die in het constraint matrix-object aanwezig zijn terug. Voor deze gegevens reserveren we een nieuw stuk geheugen. Zo behouden we een beperkte interface naar de data van het constraint matrix-object en voorkomen misbruik ervan. Indien we een pointer naar een element binnen het constraint matrix-object zouden teruggeven, kan deze ook gebruikt worden om naar elementen te gaan waarop de randvoorwaarde niet van toepassing is, wat tot niet-consistente gegevens kan leiden.

Bij de eerste oproep zal de functie "Overloop" een gelinkte lijst van ID's terug geven van de eerste randvoorwaarde, welke via "con" aan te spreken wordt. De functie is zo geconstrueerd dat met een minimum aan commando's alle randvoorwaarden kunnen overlopen worden:

```
Constraint *con; // zal by ref aangepast worden
ConstraintN *top; // zal naar het eerste element van de
   gelinkte lijst wijzen
while(top = matrix->Overloop(con) != NULL){
   /*  doe iets met con en met de id's, te gebruiken adv
      top */
}
```

Het opbouwen van de gelinkte lijst die we teruggeven, gebeurt aan de hand van volgende functies.

```
ConstraintN* GiveIDsByConstraintN (ConstraintN * wijzer)
{
  ConstraintN *T, *kop, *staart;
  if (wijzer == NULL)
    return wijzer;

  T = wijzer;
  kop = NULL;
  kop = AddToList (kop, wijzer, 'I');
  wijzer = wijzer->nextI;
  staart = kop;
  while (wijzer != T) //als dezelfde zijn we de circulaire
      lijst rond
    {
      staart = AddToList (staart, wijzer, 'I');
      wijzer = wijzer->nextI;
    }
  return kop;
}
```

```
ConstraintN* AddToList (ConstraintN * staart, ConstraintN
   * wijzer, char c)
{
  ConstraintN *temp = new ConstraintN;  // het element dat
      in de LL zal komen
  temp->ID = wijzer->ID;
  temp->con = wijzer->con;
  temp->nextID = NULL;
  temp->nextCon = NULL;
```

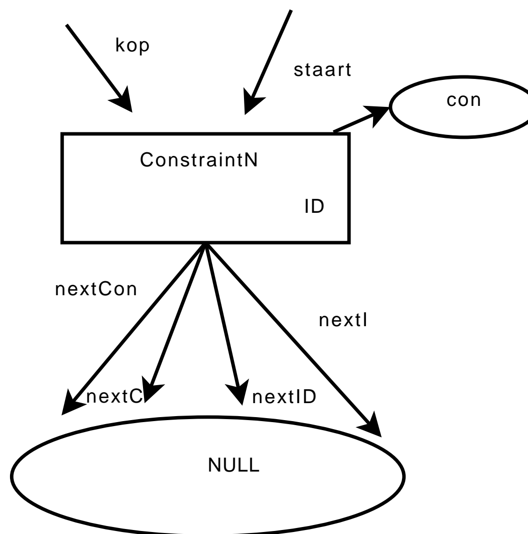
```
8      temp->nextC = NULL;
9      temp->nextI = NULL;
10     if (c == 'I' && staart != NULL)
11       {
12         staart->nextID = temp;
13       }
14     else if (c == 'C' && staart != NULL)
15         staart->nextCon = temp;
16     return temp;
17 }
```

De functie ”GiveIDsByConstraintN” geeft een gelinkte lijst terug. Elk element van de gelinkte lijst is van het type ”ConstraintN” en hiervan komt het ID-veld telkens overeen met de node-ID van een node die betrokken is in de opgegeven randvoorwaarde (de randvoorwaarde uit het ”ConstraintN”-element waar ”wijzer” naar verwijst). Met behulp van de functie ”Overloop” worden al de verschillende randvoorwaarden overlopen (in de ConstraintMatrix), en deze kunnen dan doorgegeven worden aan de functie ”GiveIDsByConstraint” die een gelinkte lijst aanmaakt die enkel elementen bevat met node-IDs die aan deze randvoorwaarde moeten voldoen. Deze gelinkte lijst staat los van de ConstraintMatrix en bevat slechts een stukje informatie hieruit.

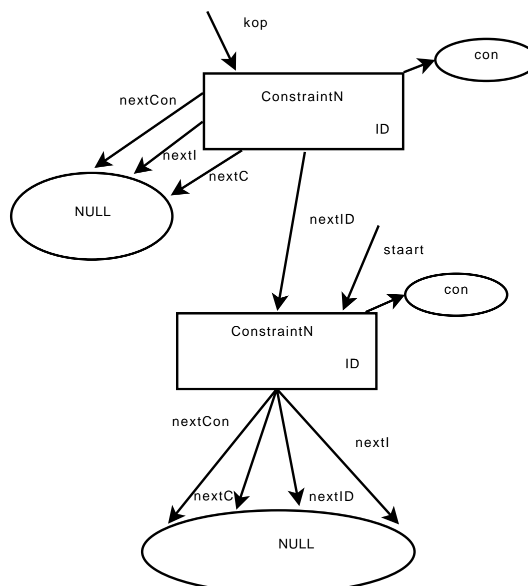
De methode begint met het toevoegen van het eerste element met behulp van de functie ”AddToList” en hier onmiddellijk met de pointer ”kop” naar te verwijzen. Het element dat wordt toegevoegd is hetgene waar met ”wijzer” naar wordt verwezen. We initialiseren ”staart” zodat deze naar hetzelfde element als ”top” wijst (het eerste element is momenteel ook het laatste). De gelinkte lijst die we tot nu toe hebben opgebouwd, is weergegeven in figuur 4.1. We zorgen dat ”staart” altijd naar het laatste element van de gelinkte lijst wijst. Vervolgens doorlopen we de circulaire lijst op basis van IDs (we volgen hiervoor telkens de pointer ”nextI”). Elk element van deze circulaire lijst voegen we toe aan de gelinkte lijst met behulp van ”AddToList”. Na het toevoegen van het tweede element ziet de gelinkte lijst er als figuur 4.2 uit. Als we merken dat de circulaire lijst volledig doorlopen is, geven we de gemaakte gelinkte lijst terug.

De functie ”AddToList” staat in voor het toevoegen van elementen aan de gelinkte lijst. Op basis van het laatste argument kan bepaald worden of we de circulaire lijst van IDs of van randvoorwaarden aan het overlopen zijn. Een gelinkte lijst van IDs zal aan de hand van een andere pointer doorlopen worden (”nextID”) dan een gelinkte lijst van randvoorwaarden (”nextCon”). Het element waar ”wijzer” naar verwijst wordt telkens toegevoegd aan de lijst. Hiervoor maken we een nieuw element aan, waar we vervolgens de inhoud van het toe te voegen element naar kopiëren. Tot slot wordt het toegevoegde element teruggegeven, dat als staart van de gelinkte lijst zal dienen.

Tot slot bespreken we hoe de te gebruiken randvoorwaarden worden opgenomen in de ”Constraint Matrix”. Hier bespreken we hoe een randvoorwaarde van het type ”Line” toegevoegd wordt, maar het toevoegen van andere randvoorwaarden is hiermee equivalent.



Figuur 4.1: *Na initialisatie en toevoeging van het eerste element, ziet de gelinkte lijst er zo uit.*



Figuur 4.2: *Het toevoegen van de elementen uit de circulaire lijst. In dit geval is de gelinkte lijst te doorlopen door NextID te volgen.*

```
bool AddLine(int ID, double a, double b)
{
    Positions p;
        Line *lijn = new Line(a,b);
        Point2D* punt = GetPos(ID); //huidige positie van
            de node opvragen
        p.push_back(*punt);
        lijn->initialise(p);
        punt->x = p.begin()->x;
        punt->y = p.begin()->y;
        mat->AddConstraintN(ID,lijn); // het toevoegen van
            de lijn in de matrix
    return true;
}
```

We beginnen met het aanmaken van een object van de ”Line” klasse met de nodige parameters. Dit object wordt toegevoegd in de ”Constraint Matrix” met behulp van de functie ”AddConstraintN”. Bij het toevoegen van een randvoorwaarde doen we direct ook een ”initialise” van dit object. Hiermee zorgen we ervoor dat de node die betrokken is in deze randvoorwaarde onmiddellijk voldoet aan de randvoorwaarde. Om de wijzigingen van de positie door te voeren werken we in 3 stappen:

- We vragen de positie op van de node en verwijzen hiernaar met behulp van ”punt” (dit is een pointer en zal wijzigingen dus direct doorvoeren in de werkelijke positie). De inhoud van ”punt” nemen we op in ”p” omdat de functie ”initialise” een parameter van het type ”Positions” verwacht.
- We geven ”p” door aan ”initialise” van het ”Line” object, welke de posities hiervan zal wijzigen (by reference) aan de hand van de ingestelde parameters van het object.
- De wijzigingen die in ”p” zijn gebeurd, zijn echter niet doorgevoerd in de opgeslagen positie. We voeren deze aanpassingen door met behulp van ”punt”.

```
void ConstraintMatrix::AddConstraintN (int ID, Constraint
    * con)
{
  ConstraintN *temp = new ConstraintN;
  ConstraintN *wijzer;
  int gevonden = 0;

  temp->ID = ID;
  temp->con = con;

```

```
10 //eerste element
11   if (head == NULL)
12     {
13       temp->nextID = NULL;
14       temp->nextCon = NULL;
15       temp->nextI = temp;
16       temp->nextC = temp;
17       head = temp;
18     }
19   else
20     {
21       wijzer = head;
22       do
23         {
24           if (wijzer->con->equals (con))
25             {
26               temp->nextI = wijzer->nextI;
27               wijzer->nextI = temp;
28               temp->nextCon = wijzer->nextCon;
29               gevonden = 1;
30               break;
31             }
32           if (wijzer->nextCon != NULL)
33             wijzer = wijzer->nextCon;
34         }
35       while (wijzer->nextCon != NULL);
36
37       if (wijzer->con->equals (con) && gevonden == 0)
38         {  // laatste element nog controleren
39           temp->nextI = wijzer->nextI;
40           wijzer->nextI = temp;
41           temp->nextCon = wijzer->nextCon;
42           gevonden = 1;
43         }
44
45       if (wijzer->nextCon == NULL && gevonden == 0)
46         { // De randvoorwaarde bestaat nog niet en wordt
             toegevoegd in de voorziene LL
47           wijzer->nextCon = temp;
48           temp->nextCon = NULL;
49           temp->nextI = temp;
50         }
51
52       // doorlopen van de ID's
53       gevonden = 0;
```

```
54        wijzer = head;
55        do
56          {
57            if (wijzer->ID == ID)
58              {
59                temp->nextC = wijzer->nextC;
60                wijzer->nextC = temp;
61                temp->nextID = wijzer->nextID;
62                gevonden = 1;
63                break;
64              }
65            if (wijzer->nextID != NULL)
66              wijzer = wijzer->nextID;
67          }
68        while (wijzer->nextID != NULL);
69
70        if (wijzer->ID == ID && gevonden == 0)
71          {  // laatste checken
72            temp->nextC = wijzer->nextC;
73            wijzer->nextC = temp;
74            temp->nextID = wijzer->nextID;
75            gevonden = 1;
76          }
77        if (wijzer->nextCon == NULL && gevonden == 0)
78          { // Er is nog geen enkele randvoorwaarde waarin
               deze Node betrokken is, de node wordt toegevoegd
                in de voorziene LL
79            wijzer->nextID = temp;
80            temp->nextID = NULL;
81            temp->nextC = temp;
82          }
83      }
84  }
```

Het toevoegen van een element in de ”Constraint Matrix” bestaat uit twee delen:

- Overlopen van de gelinkte lijst van randvoorwaarden, hier zijn twee mogelijkheden:
  - We vinden een duplicaat. We voegen het element in de circulaire lijst van IDs in (”nextI” van ”wijzer” naar het nieuwe element laten wijzen).
  - We vinden geen duplicaat. We voegen het element toe aan het einde van de gelinkte lijst van randvoorwaarden (”nextCon” van wijzer wijst naar het nieuwe element) en initialiseren de circulaire lijst van IDs zodat deze naar zichzelf wijst.

- Overlopen van de gelinkte lijst van IDs, ook hier zijn twee mogelijkheden:
  - We vinden deze ID al ergens (ook in een andere randvoorwaarden betrokken). In dit geval voegen we het element toe in de circulaire lijst van randvoorwaarden ("nextC" wijst naar het nieuwe element).
  - Er is nog geen enkele randvoorwaarde gebruikt waar deze ID in betrokken is. In dit geval voegen we het element onderaan de gelinkte lijst van IDs toe en initialiseren de circulaire lijst van randvoorwaarden zodat deze naar zichzelf wijst.

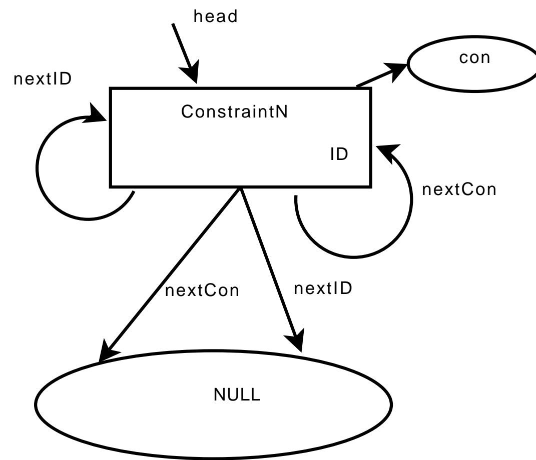
In figuur 4.3 zien we hoe de "Constraint Matrix" er uitziet na toevoeging van de eerste randvoorwaarde. Er is nog maar één randvoorwaarde aanwezig waar slechts één ID in betrokken is. Beide pointers voor de gelinkte lijst verwijzen daarom naar NULL. Ook in de circulaire lijsten zijn er geen andere elementen, zodat deze naar het eigen element verwijzen.

In figuur 4.4 zien we een voorbeeld hoe de "Constraint Matrix" er na verloop van tijd kan uitzien. Niet al de pointers zijn getekend omdat dit te onoverzichtelijk zou worden. In dit schema zijn drie randvoorwaarden betrokken en drie verschillende nodes (verschillende IDs). De gelinkte lijst om de verschillende randvoorwaarden te doorlopen is duidelijk te herkennen aan de linker kant. Bij elke randvoorwaarde die we tegen komen herkennen we een zijsprong naar rechts. Dit zijn de verschillende nodes (IDs) die in dezelfde randvoorwaarde zijn betrokken. Deze circulaire lijst is te doorlopen door de pointer "nextI" te volgen. Ook zien we een voorbeeld van een circulaire lijst van verschillende randvoorwaarden waarin één bepaalde node betrokken is. Deze circulaire lijst verbindt de randvoorwaarden die op de node met ID '1' van toepassing zijn. Deze circulaire lijst is te doorlopen door de pointer "nextC" te volgen. Ten slotte is ook de gelinkte lijst om de verschillende IDs te doorlopen herkenbaar. Deze verbindt de verschillende nodes (IDs) waarop een randvoorwaarde van toepassing is met behulp van de pointer "nextID". Deze tekening is echter niet volledig:

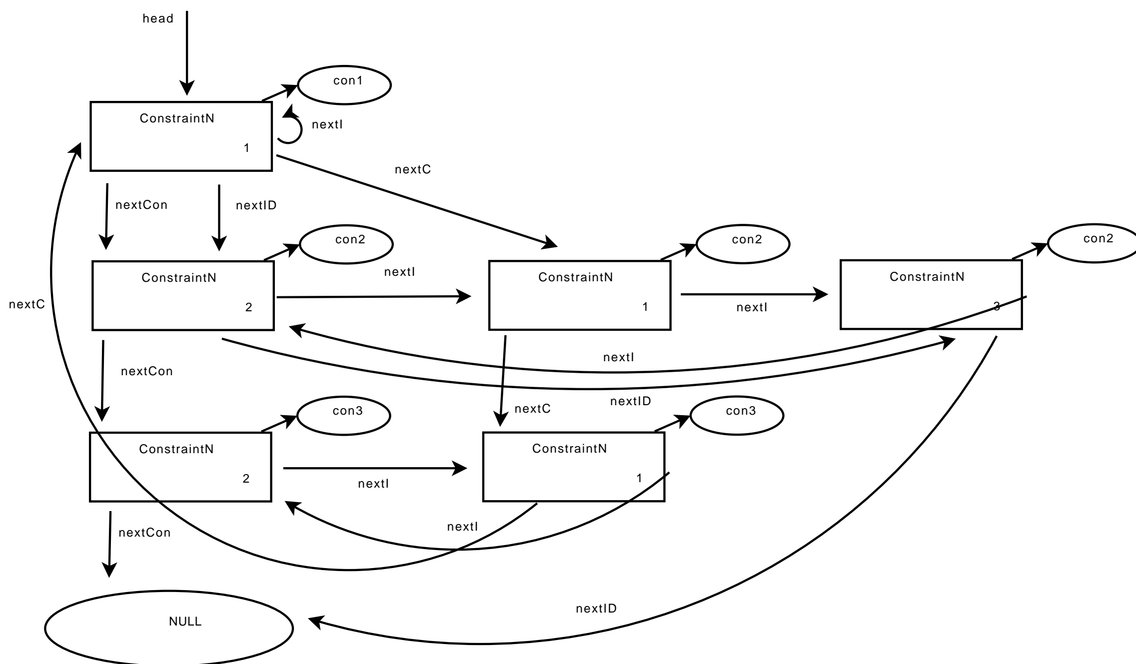
- Er ontbreekt een circulaire lijst die de verschillende randvoorwaarden doorloopt waarin de node met ID '2' is betrokken.
- Ieder element zou een pointer "nextID" moeten hebben naar het volgende element in de gelinkte lijst van nodes (IDs) en hetzelfde geldt voor de gelinkte lijst van verschillende randvoorwaarden.

4.3 Geïmplementeerde randvoorwaarden

In de vorige sectie hebben we aangegeven hoe we randvoorwaarden voorstellen en aanspreken. Tot nu toe hebben we enkel met een algemene definitie van een randvoorwaarde gewerkt. Binnen het algoritme maken we gebruik van meer specifieke varianten ervan. Deze zijn child-klassen van "Constraint".



Figuur 4.3: *De constraint matrix na het toevoegen van de eerste randvoorwaarde.*



Figuur 4.4: *Een voorbeeld hoe de "Constraint Matrix" er na toevoeging van 3 randvoorwaarden, waarbij verschillende IDs betrokken zijn, uitziet.*

4.3.1 Line

4.3.1.1 Line klasse

De eerste randvoorwaarde is de Line randvoorwaarde. Zoals de naam al aangeeft, stelt deze een lijn voor. Indien een node aan deze randvoorwaarde voldoet, wil dit zegggen dat de node op de lijn ligt die gedefineerd wordt aan de hand van de randvoorwaarde parameters. Bij deze randvoorwaarde is het voorschrift van de lijn bekend. Uiteraard kunnen meerdere nodes aan deze randvoorwaarde voldoen. Voor elk van hen dient dan afzonderlijk gedefinieerd te worden dat hij moet voldoen aan de "Line" randvoorwaarde met voorschrift ax+b (de parameters zijn a en b). Bij het invoegen in de constraint matrix wordt gedetecteerd of deze nodes aan dezelfde randvoorwaarde voldoen om duplicaten in het geheugen te vermijden. Zoals eerder vermeld heeft een Line randvoorwaarde twee parameters die samen het lijnvoorschrift voorstellen.

```
1  class Line : public Constraint
2  {
3  public:
4          Line(double a, double b);
5          bool initialise(Positions &P);
6          double geta() { return l_a;};
7          double getb() { return l_b;};
8          bool equals(Constraint *con);
9          void Execute(Positions &P);
10         void setVoorschrift(double a, double b){
11                 l_a = a;
12                 l_b = b;
13         }
14 private:
15         double GetCoord(double x);
16         double l_a,l_b;
17 protected:
18         char GiveType();
19         void GiveParams(double params[])
20         {
21                 params[0] = l_a;
22                 params[1] = l_b;
23         }
24         void Projection(double& x,double& y);
25
26 };
```

Bij het vergelijken van deze child-klasse van de Constraint klasse zien we dat hier enkele functies zijn bijgekomen:

- geta en getb: deze kunnen gebruikt worden om de parameters op te vragen.
- setVoorschrift: deze kan gebruikt worden om het opgegeven voorschrift aan te passen. Indien deze randvoorwaarde rechtstreeks (dus niet via een andere randvoorwaarde) wordt gebruikt is deze functie zinloos. Het voorschrift van een lijn wijzigt immers niet meer. Maar de ULine randvoorwaarde (4.3.4.1) die later aan bod komt, maakt intern een Line object aan waarvan het functievoorschrift pas later gekend is, zodat het nodig is dit na uitvoering van de constructor van Line nog te wijzigen.
- GetCoord: Deze functie wordt gebruikt om de y-coordinaat van een opgegeven x-waarde te bepalen aan de hand van het lijnvoorschrift. Bijvoorbeeld bij een voorschrift 2x+1 zal een aanroep van GetCoord(2) de waarde 5 terug geven.

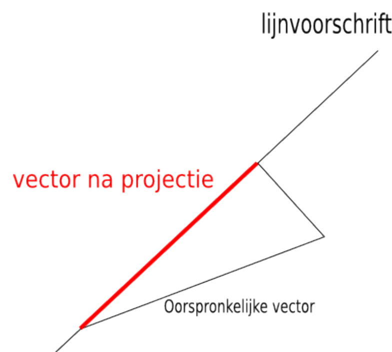
4.3.1.2 Execute

Deze functie past de richtingsvector aan, aan de hand van de randvoorwaarde (type is Line). Bij een ”Line” randvoorwaarde kunnen we ervoor zorgen dat een node op elke ogenblik (in elke verschuiving) blijft voldoen aan de randvoorwaarde. Via ”initialise” zetten we bij de eerste iteratie de node op de lijn, en door de richtingsvector zo aan te passen dat deze altijd aan het Line-voorschrift voldoet, zorgen we dat de node hier ook aan zal blijven voldoen. De actie die dient te worden uitgevoerd, is een projectie van de richtingsvector op de lijn.

```
1 void Line::Execute (Positions & P)
2 {
3   for (Positions::iterator it = P.begin (); it != P.end
      (); it++)
4     {
5       Projection (it->x, it->y);
6     }
7 }
```

Hiervoor hebben we voor elke node die aan de randvoorwaarde dient te voldoen (deze informatie halen we uit het constraint matrix-object) de richtingsvector geprojecteerd op het lijn-voorschrift. Hierna zien we hoe deze projectie wordt uitgevoerd (als parameters geven we de x- en y-component van de richtingsvector mee).

```
1 void Line::Projection (double &x, double &y)
2 {
3   double x_loc = 3.0;
4   if(x_loc == x) //veiligheid
5         x_loc+=0.001;
6   double y_loc = GetCoord (x_loc);
```



Figuur 4.5: *De richtingsvector wordt geprojecteerd op het lijnvoorschrift van Line.*

```
7
8     double Beta = atan2 (y_loc - GetCoord (0), x_loc);
9     double Alpha = atan2 (y, x);
10    double Gamma = Alpha - Beta;
11    double Length = sqrt (x * x + y * y) * cos (Gamma);
12
13    x = Length * cos (Beta);
14    y = Length * sin (Beta);
15  }
```

Eerst bepalen we de hoek tussen de lijn en de te projecteren vector. Dit gebeurt in 3 stappen:

- We bepalen welke hoek de voorgeschreven lijn maakt tegenover de x-as. Hiervoor verleggen we de lijn eigenlijk naar de oorsprong (GetCoord(0) geeft aan hoeveel we de lijn naar onder moeten schuiven om door de oorsprong te gaan).
- We bepalen welke hoek de richtingsvector maakt met de x-as.
- Het verschil van beide hoeken geeft aan welke hoek er tussen beide ligt.

Als deze hoek gekend is kunnen we berekenen wat de lengte van de vector na projectie zal zijn. Aan de hand van deze vector en de hoek die het lijn-voorschift bepaalt, berekenen we de x- en y-component van de geprojecteerde vector, welke by reference gewijzigd worden.

4.3.1.3 Initialise

De verplaatsing van een node naar de lijn kan op twee manieren.

- De node ligt op de y-as, hiervoor gaan we de corresponderende y-coordinaat bepalen en de node hierop plaatsen. Dit is een verschuiving over de y-as. Het is dus niet de kortst mogelijke verschuiving (orthogonale).

- De node ligt niet op de y-as. Nu kiezen we toch voor een orthogonale verschuiving, en verschuiven de node over een zo kort mogelijk afstand (loodrecht) naar de lijn.

Indien de node in de oorsprong ligt, wat gebeurt bij initialisatie, dienen we een eenvoudige berekening te doen. Gezien deze functie quasi zeker slechts éénmalig vanuit de oorsprong wordt uitgevoerd, en dit plaats vindt bij initialisatie waarbij de node nog ver van zijn eindbestemming ligt, zou het verspilling van rekentijd zijn om hier een complexe berekening uit te voeren om de verschuiving minimaal te houden.

Bij gebruik van de gewone Line-randvoorwaarde heeft de initialisatiefunctie vanuit een andere coordinaat geen effect, gezien de nodes in de loop van het algoritme niet meer afwijken van het Line-voorschrift. Bij de ULine-randvoorwaarde die later aan bod komt (4.3.4.1), heeft dit echter wel zin. Bij ULine wordt het voorschrift van de lijn pas bepaald aan het einde van het algoritme (als de nodes zeer dicht bij hun eindbestemming liggen). Als we dan nog een verschuiving gaan uitvoeren is het van groot belang dat deze minimaal blijft, en dus orthogonaal (loodrecht) dient te gebeuren. We zien dat de berekening van de orthogonale verschuiving zeer hard lijkt op deze van de projectie-functie.

```
1  bool Line::initialise (Positions & P)
2  {
3    double offset = GetCoord(0);
4    for (Positions::iterator it = P.begin (); it != P.end
        (); it++)
5      {
6          if(it->x == 0) // eerste geval
7                  it->y = GetCoord(0);
8          else{ // tweede geval
9                  it->y -= GetCoord(0);
10                 double Beta = atan2(it->y,it->x);
11                 double Alpha =
                      atan2(GetCoord(3)-GetCoord(0),3);
12                 double Gamma = Beta - Alpha;
13                 double Lengte =
                      sqrt(it->x*it->x+it->y*it->y)*cos(Gamma);
14                 it->x = Lengte * cos(Beta);
15                 it->y = Lengte * sin(Beta);
16                 it->y+=GetCoord(0);
17         }
18     }
19   return true;
20 }
```

4.3.1.4 Besluit

De Line randvoorwaarde stelt de ligging van een eenvoudige lijn voor waarvan het functievoorschrift op voorhand gegeven is. Het is waarschijnlijk de randvoorwaarde die het meest gebruikt zal worden. Naast het gebruik als afzonderlijke randvoorwaarde is het ook de basis voor de ULine randvoorwaarde die later aan bod komt.

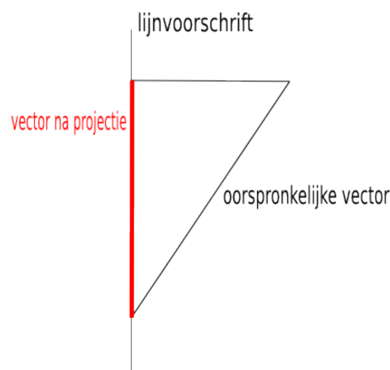
4.3.2 VLine

4.3.2.1 VLine klasse

De VLine is een "afgeslankte" versie van de Line randvoorwaarde. Deze randvoorwaarde stelt een verticale lijn voor. In tegenstelling tot de Line randvoorwaarde is er maar 1 parameter waarmee we de randvoorwaarde kunnen beschrijven, de x-coordinaat waarop de verticale lijn ligt. Deze randvoorwaarde wordt gebruikt doordat van een verticale lijn geen functievoorschrift is op te stellen (een oneindig grote richtingscoëficiënt).

```
1  class VLine : public Constraint
2  {
3  public:
4          VLine(double x);
5          bool initialise(Positions &P);
6          double GetCoord(double x);
7          double getx() { return l_x;};
8          bool equals(Constraint *con);
9          void Execute(Positions &P);
10 private:
11         double l_x;
12 protected:
13         char GiveType();
14         void GiveParams(double params[])
15         {
16                 params[0] = l_x;
17         }
18         void Projection(double& x,double& y);
19 };
```

De VLine randvoorwaarde gelijkt op de "Line" randvoorwaarde, op de aanwezigheid van de tweede parameter bij de Line-randvoorwaarde (en bijhorende functies) na zijn ze zelfs identiek.



Figuur 4.6: De richtingsvector wordt geprojecteerd op het lijnvoorschrift van VLine

4.3.2.2 Execute

De implementatie van de Execute functie is identiek met die van Line. Voor elke node die op de verticale lijn dient te liggen gebeurt er een projectie. Deze projectie is echter veel eenvoudiger dan de driehoeksmetingen die we bij de Line randvoorwaarde gebruiken.

```
void
VLine::Projection (double &x, double &y)
{
  x = 0;
}
```

De projectie gebeurt altijd loodrecht naar de verticale lijn. Dit komt overeen met het schrappen van de x-component van de richtingsvector, zodat enkel de y-component nog over blijft.

4.3.2.3 Initialise

Ook de "initialise" functie is zeer eenvoudig. Elke node wordt naar de verticale lijn verplaatst. Het enige dat dient te gebeuren, is de x-coordinaat van het punt gelijkstellen aan de parameter van "VLine" (de ligging van de verticale lijn).

```
bool VLine::initialise (Positions & P)
{
  for (Positions::iterator it = P.begin (); it != P.end
     (); it++)
    {
      it->x = l_x;
    }
```

```
7     return true;
8   }
```

4.3.2.4 Besluit

De VLine randvoorwaarde wordt gebruikt voor verticale lijnen, waarvoor geen functievoorschrift kan opgesteld worden. Doordat het een ”afgeslankte” versie van de Line randvoorwaarde is, zijn ook de functies zeer vereenvoudigd.

4.3.3 Afstand

4.3.3.1 Afstand klasse

De Afstand randvoorwaarde stelt, zoals de naam al aangeeft, een afstand voor. Aan de hand van deze randvoorwaarde kan je opgeven welk afstand er tussen twee opgegeven nodes ligt. Deze randvoorwaarde is gebaseerd op een probleem uit de praktijk. Één van de lokaties waar het systeem voor positie bepaling gebruikt zal worden, is in een magazijn waarbij er twee mobiele nodes aan weerszijden van een metalen wagen gemonteerd zijn. Doordat elk van beide mobiele nodes voor een onrechtstreekse meting zorgt met twee base stations wijken de resultaten sterk af. De oplossing hiervoor is het gebruik van de Afstand randvoorwaarde, hierbij geven we de afstand tussen de twee mobiele nodes op (lengte van de wagen) en maken geen gebruik meer van de onrechtstreekse meting.

Ook tijdens de setup kan deze randvoorwaarde gebruik worden. Met behulp van deze randvoorwaarde kunnen we de afstand tussen twee nodes opgeven (doordat we deze manueel hebben opgemeten of uit de plattegrond van het gebouw hebben kunnen opmaken). In het hoofdstuk Implementatie geven we hier een voorbeeld van (5.4.3).

De Afstand klasse ziet er als volgt uit

```
1   class Afstand : public Constraint
2   {
3   public:
4           Afstand(int afstand);
5           bool initialise(Positions &P);
6           void Execute(Positions &P){};
7           bool equals(Constraint *con);
8           double GetAfstand(){return l_afstand; }
9   private:
10          int l_afstand;
11          void Verplaats(double &x1,double &y1, double &x2,
              double &y2);
12  protected:
```

```
13         void GiveParams(double params[]){
14                 params[0] = l_afstand;
15         }
16         char GiveType(){return 'A';}
17 };
```

De functie Execute is hier leeg en zal bij uitvoeren geen wijzigingen doen aan de richtingsvectoren. We hebben geen informatie hoe we de richtingsvector kunnen verbeteren aan de hand van de Afstand randvoorwaarde.

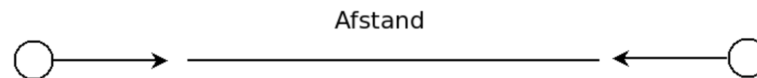
4.3.3.2 Initialise

De posities van de nodes kunnen we wel manipuleren. We kunnen ervoor zorgen dat deze voldoen aan de randvoorwaarde door deze naar elkaar toe te brengen, of van elkaar te verwijderen. De "initialise" functie ziet er zeer gelijkend uit met die van Line en VLine. Voor elke node moet er echter een verplaatsing gebeuren in plaats van een projectie.

Bij de verplaatsing gaan we de twee nodes die betrokken zijn in de randvoorwaarde, met een gelijke afstand naar elkaar toe brengen (of van elkaar verwijderen). Dit gebeurt door eerst de te verplaatsen afstand te bepalen (Pythagoras tussen de twee nodes en dan het verschil maken met de afstand die er moet tussen zijn). Als deze afstand negatief is betekent dit dat de nodes te kort bij elkaar staan en met een negatieve afstand naar elkaar moeten verschoven worden, wat neerkomt op het verwijderen van elkaar. Om de verschuiving zelf uit te voeren, moeten we weten onder welke hoek de nodes ten opzichte van elkaar staan, gezien de verplaatsing wordt opgeslagen aan de hand van gewijzigde x- en y-coordinaten. Als de nodes horizontaal naast elkaar liggen zal enkel de x-coordinaat wijzigen (de hoek is 0 graden), als de nodes verticaal boven elkaar liggen, zal enkel de y-coordinaat moeten wijzigen (de hoek is 90 graden). Figuur 4.7 geeft een grafische weergave van de randvoorwaarde Afstand.

```
1  bool Afstand::initialise (Positions & P)
2  {
3
4    for (Positions::iterator it = P.begin (); it != P.end
       (); it += 2) // Er wordt per 2 posities gewerkt
5      {
6        Verplaats (it->x, it->y, (it + 1)->x, (it + 1)->y);
7      }
8
9    return true;
10 }
```

```
1  void Afstand::Verplaats (double &x1, double &y1, double
      &x2, double &y2)
2  {
3    double teVerschuiven =
4      (sqrt ((x2 - x1) * (x2 - x1) + (y2 - y1) * (y2 - y1))
         - l_afstand) / 2;
5    double Hoek = atan2 (y2 - y1, x2 - x1);
6    x1 += teVerschuiven * cos (Hoek);
7    x2 -= teVerschuiven * cos (Hoek);
8    y1 += teVerschuiven * sin (Hoek);
9    y2 -= teVerschuiven * sin (Hoek);
10 }
```



Figuur 4.7: *De nodes worden met een gelijke afstand naar elkaar toe gebracht.*

4.3.4 ULine

4.3.4.1 ULine klasse

De laatste randvoorwaarde die geimplementeerd is, is de ULine randvoorwaarde. Deze lijkt op de Line randvoorwaarde, maar het lijnvoorschrift is niet vooraf bekend. Deze randvoorwaarde kan gebruikt worden wanneer men weet dat de nodes op een lijn staan (bijvoorbeeld een balk/kabel tegen het plafond) maar het te moeilijk of te tijdsintensief is om het voorschrift ervan te bepalen. Aan de hand van de posities die resulteren uit het standaard algoritme, kan het lijnvoorschrift gezocht worden dat het best aan deze coordinaten voldoet. Deze coordinaten worden dan aan een object van het type "Line" toegekend en aan de hand hiervan kan de "initialise" functie van dit Line object worden aangeroepen zodat alle nodes een orthogonale verplaatsing naar de lijn doen.

```
1  class ULine: public Constraint
2  {
3  public:
4          ULine(int id);
5          double Lijnberekening(Positions &P, double
              voorschrift[]);
6          bool initialise(Positions &P);
7          void Execute(Positions &P);
```

```
8           bool equals(Constraint *con);
9   private:
10          Line *l_l;
11          int l_id;
12          double b_l, a_l;
13          Positions P;
14          double X,Y;
15  protected:
16          double ChiSquareFunction (double a, double b,
               Positions P);
17          double CalculateB (double a, int n);
18          double nzSIGN(const double& a, const double& b);
19          void minBracket (double &ax, double &bx, double
               &cx);
20          void brent (const double &ax, const double &bx,
               const double &cx, double &xmin, double &fret);
21          double f1dim(double a);
22          void GiveParams(double params[]);
23          char GiveType();
24  };
```

De ULine randvoorwaarde wordt gekenmerkt door slechts één parameter, een ID. Aan de hand van deze ID wordt bepaald tot welke ULine een node hoort. Het lijnvoorschrift zal bepaald worden uit de posities van al de nodes die bij een ULine horen en dus met dezelfde ID zijn aangemaakt.

In tegenstelling tot de Line-randvoorwaarde is de Execute functie hier leeg. Tijdens het algoritme kennen we het voorschrift van de lijn nog niet. We kunnen de richtingsvector dus niet beinvloeden om hieraan te voldoen.

Het aantal functies is groter dan bij de andere randvoorwaarden. Dit komt doordat het bepalen van het lijn voorschrift vrij complex is.

4.3.4.2 Bepalen van het lijn voorschrift

Zoals in de vorige paragraaf besproken gaan we het functievoorschrift pas bepalen als elke node zich op de positie bevindt die resulteert uit het standaard algoritme. We wachten hiermee tot het einde omwille van twee redenen:

- Het bepalen van het lijnvoorschrift is een rekenintensieve taak, dus moeten we de uitvoering ervan minimaliseren.
- Aan het einde van het algoritme heeft elke node een positie die zeer kort bij de 'echte' positie (en dus de lijn) ligt, zodat het lijnvoorschrift dat we hieruit kunnen afleiden grote kans heeft de 'echte' lijn te benaderen.

Hieraan is echter ook een nadeel verbonden. Als we gebruik maken van Cooperative Localisation (de onderlinge afstand tussen mobiele nodes wordt ook in rekening gebracht) wordt de positie van de nodes die niet op de lijn liggen niet aangepast aan de verbeterde nauwkeurigheid van degene die wel op de lijn liggen.

Het bepalen van het lijnvoorschrift gebeurt aan de hand van een ”function of merit”, die gebaseerd is op Chi-kwadraat functie (vergelijking 4.1). We zien twee termen in de noemer. Bij gebruik van coordinaten kan er een fout in twee richtingen optreden, er kan een afwijking op zowel de y-coordinaat als op de x-coordinaat bestaan. Als we nu het voorschrift voor a en b willen afleiden (partiële afgeleide naar beide termen), bemerken we echter dat we enkel voor b een liniaire term kunnen vinden (vergelijking 4.2). Uit deze vergelijking kunnen we afleiden dat de lijn die we zoeken door het middelpunt van de punten zal gaan. Dit is eenvoudiger te zien als we vergelijking 4.2 herschrijven als vergelijking 4.3

$$\chi^2(a,b) = \sum \frac{(y_i - b - ax_i)^2}{a^2\sigma_x^2 + \sigma_y^2} \tag{4.1}$$

$$b = 1/n(\sum y_i - a \sum x_i) \tag{4.2}$$

$$\bar{y} = a\bar{x} + b \tag{4.3}$$

Als we partieel afleiden naar a, is de term in de noemer niet meer constant. Dit betekent dat het resultaat van de afleiding zal stijgen in orde in plaats van dalen. Hieruit volgt dat de functie complexer wordt in plaats van eenvoudiger. Om toch tot een waarde voor a te komen gaan we deze iteratief benaderen. Het vinden van de parameters voor de lijn gebeurt in een iteratie waarvan elke stap uit 3 delen bestaat:

- Het bepalen van b uit de gemiddelde waarden en de huidige waarde van a.
- Een lijnminimalisatie uitvoeren op de Chi-kwadraat functie om een minimale waarde van a te vinden bij de huidige waarde van b. Doordat b constant is, is er nog maar één variabele zodat we kunnen terugvallen op een 1-dimensionale methode (lijnminimalisatie).
- We vergelijken het resultaat van de Chi-kwadraat functie met dat van de vorige iteratie. Als het verschil zeer klein is, stoppen we de iteratie.

Voor het bepalen van a maken we ook hier gebruik van MinBracket en de Brent methode. Hier hebben we echter geen toegang tot de afgeleide van de functie, zodat we de eerder besproken versie van Brent niet kunnen gebruiken. De MinBracket functie kan echter wel herbruikt worden, maar in het oorspronkelijke algoritme is deze binnen een klasse gedefinieerd zodat deze moeilijk toegankelijk wordt. We hebben er dan ook voor gekozen om een duplicaat hiervan te voorzien binnen de ULine klasse.

```
double ULine::Lijnberekening (Positions &P, double
   voorschrift[])
{
this->P = P;

  bool NotReady = true;
  double vorige = 10000000;     //groot instellen
  double result = 0;
  double ax = 0, bx = 0, cx = 0;
  int n = 0;
  double fret = 0;
  a_l = b_l = 0;                               //initialiseer op
     oorsprong
  int teller = 0;

//gemiddelde berekenen
  for (Positions::iterator it = P.begin (); it != P.end
     (); it++)
    {
      n++;
      X += it->x;
      Y += it->y;
    }
  Y /= n;
  X /= n;

//begin iteratie
  while (NotReady)
    {
      b_l = CalculateB (a_l, n);
      ax=cx=0.0;
      bx=2.0;
      minBracket (ax, bx, cx); //grenzen zitten in ax en
         cx, bx is de laagste
      brent (ax, bx, cx, a_l, fret);
      result = ChiSquareFunction (a_l, b_l, P);
      if ((vorige - result) < 0.01  )
        NotReady = false;

      vorige = result;
    }
  voorschrift[0] = a_l;
  voorschrift[1] = b_l;

  return result/n;
```

```
}
```

Het bepalen van een lijn die het beste bij een set van punten past, is een modellering. Het is belangrijk om naast het vinden van de parameters ook te controleren of het model wel geschikt is, en of de gevonden parameters robuust zijn.

Dat het model geschikt is, weten we zeker, omdat we weten dat in werkelijkheid deze punten op een lijn liggen (dat is de reden dat we de randvoorwaarde gebruiken). De robuustheid van de parameters kunnen we controleren door naar het gemiddelde resultaat van de Chi-kwadraat functie te kijken (het resultaat van Lijnberekening). Als deze waarde klein is, betekent dit dat gemiddeld gezien de punten zeer kort bij het gevonden lijnvoorschrift liggen. Een kleine wijziging van de parameters zal dan ook een grote wijziging in het resultaat van de Chi-kwadraat functie teweegbrengen. Als we de afstand voorstellen op een Gauss-curve, is deze in dit geval smal. We kunnen met grote zekerheid zeggen dat de gevonden parameters goed overeenkomen met die van de werkelijke lijn.

Als het gemiddelde resultaat echter groot is, wil dit zeggen dat gemiddeld gezien de nodes ver weg liggen van de gevonden lijn. Het effect van een wijziging aan de parameters zal hier veel kleiner zijn, zodat er dus een grotere spreiding mogelijk is. Hier zijn we minder zeker dat ons model wel exact genoeg overeenkomt met het werkelijke voorschrift van de lijn. De parameters zijn hier minder robust. Als we de afstanden van de nodes tot de lijn voorstellen in een gauss-curve zal deze breder zijn.

Voordat we een verplaatsing doen van de nodes naar de gevonden lijn controleren we of het resultaat van Lijnberekening klein genoeg is. Als de waarde klein is, wil dit zeggen dat de spreiding op de variabelen klein is, zodat we zeker zijn van de correctheid van het model. Dit geeft ons een extra zekerheid dat we er goed aan zullen doen dit lijnvoorschrift te gebruiken.

```
bool ULine::initialise (Positions & P)
{
  double voorschrift[2];
  double resultaat = Lijnberekening (P,voorschrift);
  if(resultaat < 0.01){
   l_l->setVoorschrift (voorschrift[0], voorschrift[1]);
   l_l->initialise (P);
  }

  return true;
}
```

```
double ULine::CalculateB (double a, int n)
```

```
{
  return (Y - a * X);
}
```

```
double ULine::ChiSquareFunction (double a, double b,
   Positions P)
{
  double waarde = 0;
  double VarX;
  double VarY = VarX = 1;        //kan later aangepast
     worden

  for (Positions::iterator it = P.begin (); it != P.end
     (); it++)
    {
      waarde += ((it->y - b - a * it->x) * (it->y - b - a
         * it->x)) / (VarY * VarY + a * a * VarX * VarX);
    }
  return waarde;
}
```

De Brent methode die we hier gebruiken, kan geen gebruik maken van de afgeleide. Net als degene die gebruikt wordt bij de lijnminimalisatie van de totale fout over de richtingsvector, bevat deze methode twee technieken. Als snelle methode wordt er hier gebruik gemaakt van parabolische extrapolatie, gezien we hier naar een minimum zoeken en niet naar een nulpunt van de gradiënt. Als meer robuuste methode wordt hier gebruik gemaakt van de golden mean in plaats van bisection (was de robuuste methode waarop gesteund werd bij de lijnminimalisatie) .

De Brent methode en de MinBracket methode maken gebruik van een functie die de functiewaarde in één dimensie aangeeft voor een opgegeven a waarde (f1dim). Dit komt neer op het oproepen van ChiSquareFunction waarvan enkel de eerste parameter niet constant is.

```
double ULine::f1dim (double a)
{
  return ChiSquareFunction (a, b_l, P);
}
```

```
1  void ULine::brent (const double &ax, const double &bx,
      const double &cx, double &xmin, double &fret)
2  {
3    double TOL = 0.0002;
4    int ITMAX = 1000000;
5    double CGOLD = 0.3819660;
6    double ZEPS = 0.0000000001;    //1.0e-10;
7    double d = 0.0;
8    double e = 0.0;
9
10   double a, b;
11   if (ax < cx)
12     {
13       a = ax;
14       b = cx;
15     }
16   else
17     {
18       a = cx;
19       b = ax;
20     }
21
22   double x = bx;
23   double w = bx;
24   double v = bx;
25   double fx = f1dim (x);
26   double fw = fx;
27   double fv = fx;
28
29   double xm, tol1, tol2;
30
31   double r, p, q, etemp, u, fu;
32
33   for (int iter = 0; iter < ITMAX; iter++)
34     {
35       xm = 0.5 * (a + b);
36       tol1 = TOL * fabs (x) + ZEPS;
37       tol2 = 2.0 * tol1;
38 if (fabs (x - xm) <= (tol2 - 0.5 * (b - a)))
39         {
40           xmin = x;             // stel xmin in op dit
              minimum
41           fret = fx;            // stel fret in op de
              overeenkomstige fout
42           return;
```

```
43          }
44        if (fabs (e) > tol1)
45          {
46           //parabolic extrapolation
47            r = (x - w) / (fx - fv);
48            q = (x - v) * (fx - fw);
49            p = (x - v) * q - (x - w) * r;
50            q = 2.0 * (q - r);
51            if (q > 0.0)
52              p = -p;
53            q = fabs (q);
54            etemp = e;
55            e = d;
56            if (fabs (p) >= fabs (0.5 * q * etemp) || p <= q
                  * (a - x) || p >= q * (b - x))
57              d = CGOLD * (e = (x >= xm ? a - x : b - x));
58            else
59              {
60                d = p / q;
61                u = x + d;
62                if (u - a < tol2 || b - u < tol2)
63                  d = nzSIGN (tol1, xm - x);
64              }
65          }
66        else
67          {
68            d = CGOLD * (e = (x >= xm ? a - x : b - x));
69          }
70        u = (fabs (d) >= tol1 ? x + d : x + nzSIGN (tol1,
             d));
71        fu = f1dim (u);
72
73        if (fu <= fx)
74          {
75            if (u >= x)
76              a = x;
77            else
78              b = x;
79            v = w;
80            w = x;
81            x = u;
82            fv = fw;
83            fw = fx;
84            fx = fu;
85          }
```

```
86          else
87            {
88              if (u < x)
89                a = u;
90              else
91                b = u;
92              if (fu <= fw || w == x)
93                {
94                  v = w;
95                  w = u;
96                  fv = fw;
97                  fw = fu;
98                }
99              else if (fu <= fv || v == x || v == w)
100               {
101                 v = u;
102                 fv = fu;
103               }
104           }
105       }
106   std::cout << "Too many iterations in routine brent" <<
        std::endl;
107   xmin = x;
108   fret = fx;
109 }
```

4.3.5 Conclusies

In dit hoofdstuk hebben we randvoorwaarden behandeld. Eerst hebben we gezien hoe een randvoorwaarde wordt voorgesteld en welke functies nodig zijn om hem te gebruiken. Daarna hebben we de klasse "ConstraintMatrix" besproken, die gebruikt wordt om de randvoorwaarden tijdens het algoritme aan te kunnen spreken. Ten slotte hebben we de randvoorwaarden besproken die geïmplementeerd zijn. De randvoorwaarden die we besproken hebben zijn:

- Line: Deze stelt een lijn voor waarvan het functievoorschrift gekend is.
- VLine: Deze stelt een verticale lijn voor. Van een verticale lijn kan geen functievoorschrift worden opgesteld.
- Afstand: Deze geeft de mogelijkheid de afstand tussen twee nodes op te geven.
- ULine: Deze stelt een lijn voor waarvan het functievoorschrift bepaald moet worden aan de hand van de positie van nodes die op deze lijn liggen.

Hoofdstuk 5

Implementatie

5.1 Inleiding

In hoofdstuk 4 hebben we de verschillende randvoorwaarden besproken en hoe deze in de ”Constraint Matrix” opgenomen worden. In dit hoofdstuk bespreken we hoe we het gebruik van randvoorwaarden integreren in het algoritme. We hebben gezien dat er slechts twee factoren zijn (de richtingsvector en de positie) waarop we inspelen. Om op deze factoren in te spelen hebben we twee functies voorzien in de klasse ”Constraint”:

- ”initialise” die de positie van een node wijzigt zodat deze aan de randvoorwaarde voldoet.
- ”Execute” die de richtingsvector die is opgesteld voor een node wijzigt zodat de node na verschuiving over deze vector nog steeds aan de randvoorwaarde zal voldoen.

In dit hoofdstuk behandelen we vanuit welke functie we een oproep doen van ”initialise” en ”Execute”.

Aan het einde van het hoofdstuk geven we enkele voorbeelden van resultaten uit de GUI.

5.2 Aanroepen van ”initialise”

De functie ”initialise” zorgt dat de nodes waarop de randvoorwaarde van toepassing is aan de randvoorwaarde voldoen. We willen dat op elk moment aan de randvoorwaarde is voldaan zodat het vinden van een resultaat-oplossing ook optimaal verloopt.

We roepen de functie ”initialise” van op 4 plaatsen op:

- Bij het toevoegen van een randvoorwaarde in de ”Constraint Matrix”.
- Na de verschuiving van de nodes.
- Tijdens de lijnminimalisatie (zie 3.3.4). Zo zorgen we dat tijdens het berekenen van de fout (zie formule 3.3) rekening wordt gehouden met de posities waar de nodes zullen terecht komen na verschuiving.

- In de aanroep van de functie ”UnLine”, deze behandelt de initialisatie voor de randvoorwaarde ”ULine”.

Het aanpassen van de coordinaten gebeurt telkens in 3 stappen:

- Opzoeken van de posities die aangepast moeten worden en deze bewaren in een variabele van het type ”Positions”
- De initialise functie uitvoeren waarbij we de variabele meegeven als parameter. De posities zullen by reference gewijzigd worden.
- De wijzigingen aan de posities ook doorvoeren op de bewaarde posities.

5.2.1 Na toevoeging in ”Constraint Matrix”

Het toevoegen van een randvoorwaarde in de ”Constraint Matrix” werd al in hoofdstuk 4 besproken. Hier benadrukken we de aanroep van ”initialise” en de 3 stappen die hierbij komen kijken.

```
1  bool CGraph::AddLine(int ID, double a, double b)
2  {
3      Positions p;
4      Line *lijn = new Line(a,b);
5
6      //stap 1
7      Point2D* punt = GetPos(ID);
8      p.push_back(*punt);
9
10     //stap 2
11     lijn->initialise(p);
12
13     //stap 3
14     punt->x = p.begin()->x;
15     punt->y = p.begin()->y;
16
17     mat->AddConstraintN(ID,lijn);
18     return true;
19 }
```

Bij het toevoegen van een randvoorwaarde willen we dat de nodes onmiddellijk aan de randvoorwaarde voldoen. Zo zorgen we ervoor dat vanaf de eerste iteratie op een correcte manier naar een oplossing gezocht wordt. Bij de randvoorwaarde ”Line” en ”VLine” zal de node zelfs niet meer van de randvoorwaarde afwijken. Dit komt doordat de richtingsvector aangepast wordt aan het lijnvoorschrift van de randvoorwaarde (waardoor enkel verplaatsingen over de lijn voorkomen).

5.2.2 Na elke verschuiving

```
double dlinmin(){
/*
 * lijnminimalisatie doen en de verschuiving uitvoeren
 */
   m_pGraph->mat->ResetPointer();
   ConstraintN *kop,*w;
   Constraint *con;
   while(kop = m_pGraph->mat->Overloop(con)){
         if(con->GiveType() == 'U')
         continue;
      if(kop == NULL)
         break;
      w = kop;
      Positions p;
      p.clear();

      //stap 1
      while(w != NULL){
         Point2D *punt = m_pGraph->GetPos(w->ID);
         p.push_back(*punt);
         w = w->nextID;
      }
      //stap 2
      con->initialise(p);
      w = kop;
      Positions::iterator iPos = p.begin();
      //stap 3
      while(w != NULL && iPos != p.end()){
         m_pGraph->SetPos(w->ID,*iPos);
         w = w->nextID;
         iPos++;
      }
   }
/*
 * berekenen van de totale fout en terug geven
 */
}
```

Zoals we in hoofdstuk 3 hebben uitgelegd, gebeurt het zoeken naar een oplossing door iteratief de nodes korter naar een oplossing te brengen. Voorgaande code wordt in elke iteratie uitgevoerd nadat de verschuiving van de nodes heeft plaats gevonden. Zo zorgen we ervoor dat na elke iteratie terug voldaan wordt aan de randvoorwaarden. De bere-

kening van de totale fout gebeurt hierdoor op basis van een set van punten die aan alle randvoorwaarden voldoen.

We beginnen met de aanroep van ”ResetPointer”. Dit zorgt dat de interne variabele ”pointer” naar het eerste element (de eerste randvoorwaarde) van de ”Constraint Matrix” verwijst. Aan de hand van ”Overloop” gaan we de verschillende randvoorwaarden overlopen. Hierbij komt een verwijzing naar de randvoorwaarde in ”con” terecht en verwijst ”kop” naar het eerste element van een gelinkte lijst van elementen die betrokken zijn in deze randvoorwaarde. Het aanpassen van de posities gebeurt ook hier in 3 stappen:

- We vragen de posities van de nodes die betrokken zijn in de randvoorwaarde op uit de ”Constraint Matrix” en bewaren deze in ”p” (van het type Positions).
- De posities van de nodes worden doorgegeven aan de functie ”initialise” van de randvoorwaarde die we momenteel behandelen.
- De functie ”initialise” heeft de wijzigingen aan de posities doorgevoerd in ”p”. We voeren deze veranderingen ook door op de opgeslagen posities.

Voor randvoorwaarden van het type ”ULine” voeren we hier geen ”initialise” uit. Deze zullen pas behandeld worden als de posities hun eindbestemming beter benaderen.

5.2.3 Tijdens de iteratie van de lijnminimalisatie

Tijdens het uitvoeren van de lijnminimalisatie (het zoeken naar de optimale verschuiving over de richtingsvectoren) wordt gebruik gemaakt van de berekening voor de totale fout. Het berekenen van de totale fout gebeurt op basis van de posities waar de nodes zouden staan bij een bepaalde verschuiving (zie het algoritme ”dBrent” in hoofdstuk 3). Door enkel gebruik te maken van de vorige 2 gevallen van ”initialise” komen we in het volgende scenario:

- Elke node staat op een positie zodat aan de randvoorwaarden is voldaan.
- We berekenen de richtingsvector en de optimale verschuivingswaarde.
- We voeren de verschuiving uit.
- We verplaatsen de nodes zodat deze terug aan de randvoorwaarden voldoen.

Direct na de verschuiving verplaatsen we de nodes weer zodat deze aan de randvoorwaarden voldoen. Bij het bepalen van de verschuivingswaarde houden we echter geen rekening met de wijziging van de node posities om aan de randvoorwaarden te voldoen. We kunnen dit oplossen door ”initialise” ook binnen de lijnminimalisatie aan te roepen. Zo zorgen we ervoor dat de fout (die we minimaal willen maken in de lijnminimalisatie) wordt berekend op basis van de posities waar de nodes werkelijk terecht zullen komen.

```
double df1dim(const double& x){
 /*
  *  Maak een set van posities bij een bepaalde
     verschuivingswaarde x
  */

 //aanpassen van de coordinaten aan de randvoorwaarden
 m_pGraph->mat->ResetPointer();
 ConstraintN *kop,*w;
 Constraint *con;
 while(kop = m_pGraph->mat->Overloop(con)){
   if(con->GiveType() == 'U')
      continue;
   if(kop == NULL)
      continue;
   w = kop;
   Positions p;
   p.clear();

   //stap1
   while(w != NULL){
      Point2D punt = m_tempPos[w->ID-1];
      p.push_back(punt);
      w = w->nextID;
   }

   //stap 2
   con->initialise(p);

   w = kop;
   Positions::iterator iPos = p.begin();

   //stap 3
   while(w != NULL && iPos != p.end()){
      m_tempPos[w->ID-1] = *iPos;
      w = w->nextID;
      iPos++;
   }
 }

 /*
  *  Bereken de afgeleide van de functie aan de hand van
     de posities en geef deze terug
  */
}
```

Het stukje code om de posities aan de randvoorwaarden aan te passen is identiek als degene die in de vorige listing gebruikt werd. We herkennen ook hier de 3 stappen en we herkennen zelfs 3 overkoepelende delen in de functie:

- Ophalen van de posities en een verschuiving hierop toepassen.
- Aanpassen van de verschoven punten aan de randvoorwaarden. Dit is code die we toevoegen in de bestaande code. Dit zorgt dat vanuit het algoritme we functies van de randvoorwaarden worden gebruikt.
- Een resultaatwaarde berekenen aan de hand van de posities (in dit geval de afgeleide).

De functie ”f1dim” die ook gebruikt wordt in zowel ”MinBracket” als in ”dBrent”, is bijna identiek aan de functie ”df1dim”. De functie ”f1dim” zal echter niet een afgeleide berekenen maar de totale fout.

5.2.4 In de functie ”UnLine”

De laatste plaats waar we ”initialise” zien terug komen is in de functie ”UnLine”. In deze functie gaan we het lijnvoorschrift van een ”ULine” randvoorwaarde bepalen. We roepen deze functie aan na het iteratieve proces. Op dit moment staan al de nodes kort bij hun eindpositie, en heeft het lijnvoorschrift dat we berekenen grote kans om het juiste te zijn.

```
1  void UnLine(){
2   m_pGraph->mat->ResetPointer();
3   ConstraintN *kop, *w;
4   Constraint *con;
5
6   while(kop = m_pGraph->mat->Overloop(con)){
7     if(con->GiveType() != 'U') //!!!
8        continue;
9     if(kop == NULL)
10       continue;
11    w = kop;
12    Positions p;
13    p.clear();
14
15    //stap 1
16    while(w != NULL){
17       Point2D *punt = m_pGraph->GetPos(w->ID-1);
```

```
18         p.push_back(*punt);
19         w = w->nextID;
20      }
21      // stap 2
22      con->initialise(p);
23
24      w = kop;
25      Positions::iterator iPos = p.begin();
26
27      //stap 3
28      while(w != NULL && iPos != p.end()){
29         m_pGraph->SetPos(w->ID-1,*iPos);
30         w = w->nextID;
31         iPos++;
32      }
33   }
34 }
```

We zien hier bijna identieke code terug komen. Het verschil waar we op willen wijzen, is dat hier enkel randvoorwaarden van het type ”ULine” behandeld worden (lijn 7). De rest van de functie is identiek aan die rond andere oproepen van ”initialise”. De functie ”initialise” van de klasse ”ULine” is uitgebreider dan die van andere randvoorwaarden. Bij oproep van ”initialise” gebeurt het volgende:

- Uit de posities die zijn mee gegeven, wordt het lijnvoorschrift bepaald dat het beste past bij deze posities.
- Het lijnvoorschrift wordt bewaard in een intern object van het type ”Line”.
- Als het bepalen van het lijnvoorschrift een goed resultaat heeft opgeleverd, roepen we ”initialise” van het ”Line” object aan, zodat de nodes verplaatst worden naar het lijnvoorschrift.

Ook hier moeten we na de oproep van ”initialise” de gewijzigde posities nog kopiëren naar de opgeslagen posities.

5.3 Aanroepen van ”Execute”

Met de aanroep van ”Execute” wijzigen we de richtingsvector aan de hand van de randvoorwaarden. Er is slechts 1 plaats waar we deze functie moeten aanroepen, namelijk bij het bepalen van de richtingsvectoren.

```
ErrorMap FDxLSsmctr(Positions* pPositions){
 /*
  * De gradi\"ent bepalen, in dit geval aan de hand van
     het veer systeem
  */

   m_pGraph->mat->ResetPointer();
   ConstraintN *kop,*wijzer;
   Constraint *con;
   while(kop = m_pGraph->mat->Overloop(con)){
      if(kop == NULL)
         continue;
      wijzer = kop;
      Positions p;
      p.clear();

      //stap 1
      while(wijzer != NULL){
         ErrorMap::iterator iError =
            errors.find(wijzer->ID);
         p.push_back(iError->second);
         wijzer = wijzer->nextID;
      }

      //stap 2
      con->Execute(p);

      wijzer = kop;
      Positions::iterator iPos = p.begin();

      //stap 3
      while(wijzer != NULL && iPos != p.end()){
         ErrorMap::iterator iError =
            errors.find(wijzer->ID);
         iError->second = *iPos;
         wijzer = wijzer->nextID;
         iPos++;
      }
   }

 /*
  * Terug geven van de ErrorMap
  */
}
```

Deze functie geeft een "ErrorMap" terug, dit is gewoon een combinatie van een "ID" van een node en een "Position" welke de richtingsvector bevat. We zien ook hier de 3 stappen terugkomen:

- Het bepalen van de gradiënt voor elke node en deze bewaren in een ErrorMap
- Het aanpassen van de gradiënt in de ErrorMap voor nodes die in een randvoorwaarde betrokken zijn die hier invloed op kunnen uitoefenen.
- Het teruggeven van de ErrorMap.

Het beïnvloeden van de richtingsvector is hier enkel opgenomen in de functie die de richtingsvectoren opstelt. Er wordt dus geen rekening gehouden met de aanpassing die het gebruik van de toegevoegde gradiënt met zich meebrengt. Doordat we enkel de randvoorwaarden die een lijn voorstellen hebben voorzien van een "Execute" functie (van de andere randvoorwaarden is de "Execute" functie leeg gebleven zodat deze geen wijzigingen aanbrengen) hebben we hier geen hinder van. De voorgaande richting is namelijk ook volgens het lijnvoorschrift. Indien we randvoorwaarden definiëren die geen lijn voorstellen en toch een invloed uitoefenen op de richtingsvector zullen we na het opstellen van de toegevoegde gradiënt vector de vectoren moeten bijsturen voor de randvoorwaarden. Het is echter niet mogelijk om de aanroep van "Execute" enkel na het opstellen van de toegevoegde gradiënt te doen, omdat er voor de lijnminimalisatie beroep wordt gedaan op de gradiënt en niet op de toegevoegde gradiënt.

5.4 Enkele voorbeelden

5.4.1 Beschikbare tools

Om een grafische voorstelling van de resultaten te maken, heb ik gebruik gemaakt van een GUI. Deze GUI is ontwikkeld in python. Voor het opzetten van een simulatie zijn er 5 files nodig:

- initialise: een bestand dat voor elke node de beginpositie bevat.
- measurements: de afstanden die tussen de nodes gemeten wordt. Er is een functie voorzien om deze te genereren (de werkelijke afstand + een willekeurige foutwaarde).
- basestations: deze file bevat een lijst van de basestations en hun positie.
- true: de werkelijke positie van de node. Aan de hand hiervan kunnen we een vergelijking maken van de berekende positie en de werkelijke positie en zo een besluit vormen over de gemaakte verbeteringen.
- constraint: in dit bestand definiëren we de te gebruiken randvoorwaarden en op welke nodes deze betrekking hebben.

Essensium heeft een API en een simulator ontwikkeld. De simulator kan afstandsmetingen genereren die de werkelijkheid weerspiegelen. Zo kunnen we met behulp van de API gebruik maken van afstandsmetingen afkomstig van een simulatie, of gebruik maken van werkelijke afstandsmetingen.

Op dit ogenblik hebben we het gebruik van randvoorwaarden nog niet volledig geïntegreerd in deze API, dus de afstandsmetingen uit volgende voorbeelden zijn niet afstandsmetingen uit de simulator. Voor het genereren van de afstandsmetingen hebben we gebruik gemaakt van een functie die in het oorspronkelijke algoritme aanwezig was. Deze functie genereerd de afstandsmetingen op basis van een opgesteld model die de werkelijkheid weerspiegelt.

5.4.2 Voorbeeld met Line en VLine

De situatie waarvan we vertrekken, is een opstelling waar de basisstations in een raster zijn geplaatst. Deze kunnen bijvoorbeeld bevestigd zijn aan het plafond van een magazijn. Het raster is 32 meter breed en 20 meter diep. In figuur 5.1 zien we wat het resultaat als we het algoritme uitvoeren zonder gebruik van randvoorwaarden. De gemiddelde afwijking die we bekomen is 20,5 cm per node en de maximale afwijking is bijna 41 cm.

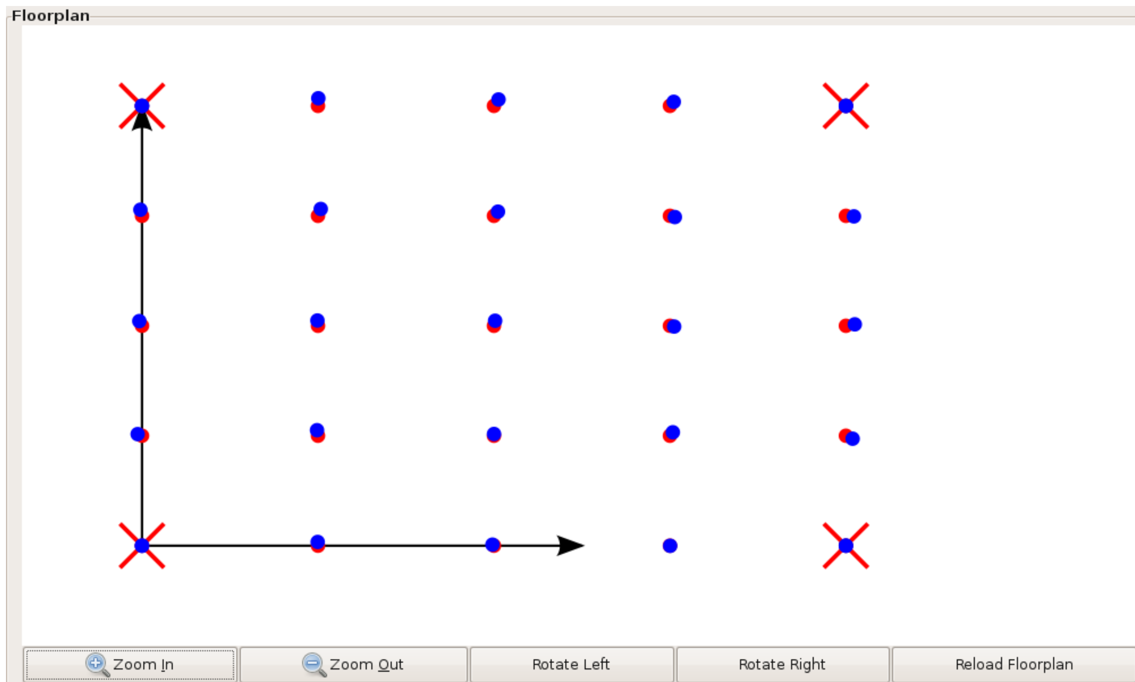
Door gebruik te maken van randvoorwaarden kunnen we dit resultaat verbeteren. Eerst maken we gebruik van twee Line randvoorwaarden (figuur 5.2). De gemiddelde afwijking die we nu bekomen is nog slechts 15,3 cm per node. Dit wil zeggen dat we gemiddeld meer dan 5 cm korter bij de werkelijke positie liggen. We zien echter ook dat de maximale afstand verhoogd is tot bijna 46 cm. Dit maximum vindt niet plaats bij de nodes waarop de randvoorwaarden van toepassing zijn. De nodes waarop geen randvoorwaarde van toepassing is, worden wat naar links en naar rechts geduwd. Hierdoor krijgen de nodes aan de linker- en rechterkant van de afbeelding een grotere afwijking.

We kunnen de resultaten nog verbeteren door ook gebruik te maken van VLine randvoorwaarden op de andere muren (figuur 5.3). Door de extra randvoorwaarden, worden de resultaten verbeterd tot een gemiddelde van 7,8 cm per node (wat een verbetering van 7,5 cm betekend ten opzichte van het gebruik van enkel de twee Line randvoorwaarden). De oorzaak van deze grote verbetering, is dat het gebruik van VLine randvoorwaarden de nodes naar binnen duwen. Hierdoor compenseren we het probleem dat bij het gebruik van enkel de Line randvoorwaarden ontstond. De maximum afwijking is ook gereduceerd tot 18 cm wat minder dan de helft is van de resultaten zonder het gebruik van randvoorwaarden.

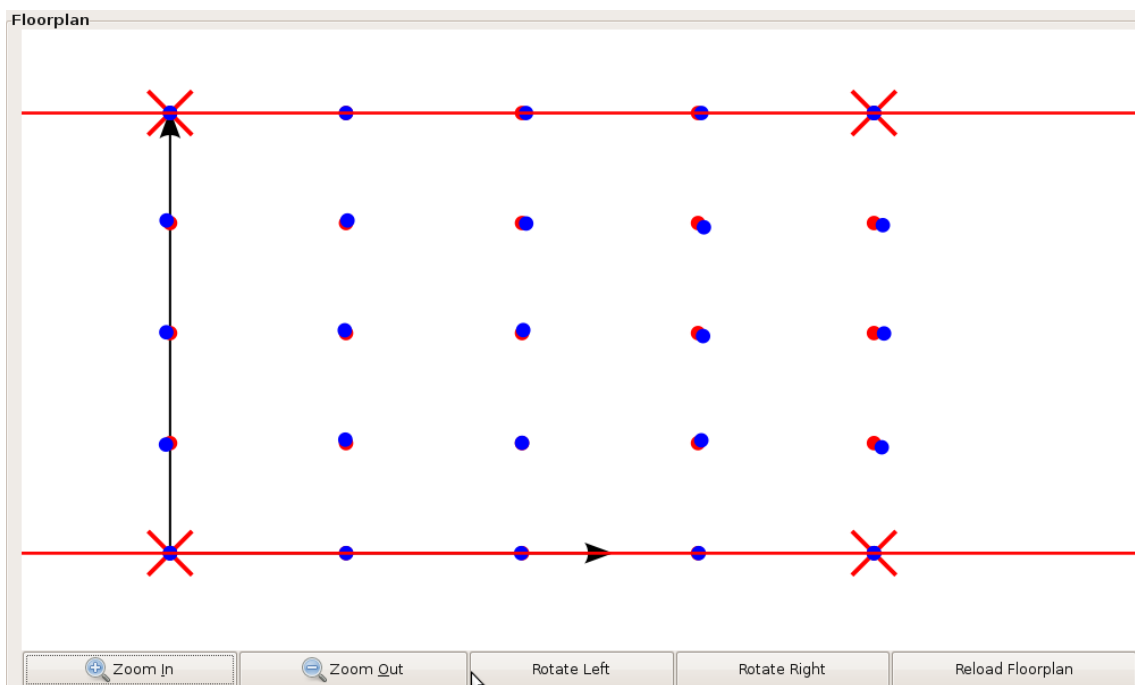
De resultaten worden samengevat in tabel 5.1.

Tabel 5.1: *Overzicht van de verbeteringen met de Line en de VLine randvoorwaarden*

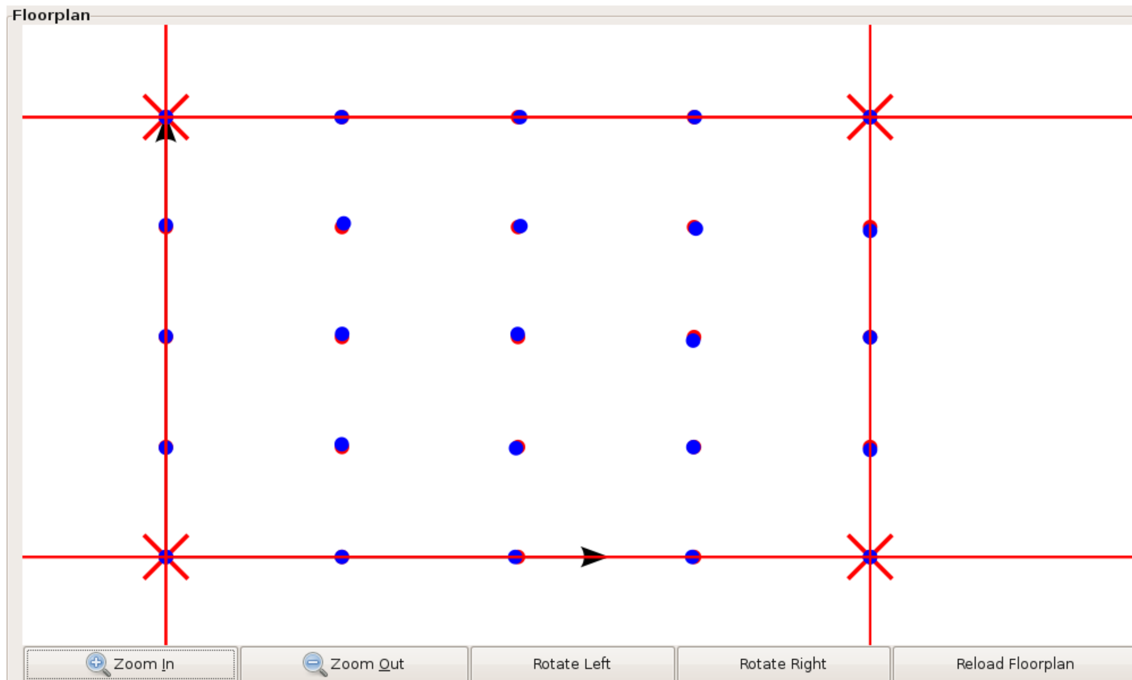
Randvoorwaarden	gemiddelde afwijking per node	maximale afwijking
geen	20,5cm	41cm
2 Line	15,3cm	46cm
2 Line en 2 VLine	7,8cm	18cm



Figuur 5.1: *Als we geen randvoorwaarden gebruiken, is het resultaat van de positie bepaling niet optimaal.*



Figuur 5.2: *De resultaten van de positiebepaling kunnen verbeterd worden door het gebruik van twee Line randvoorwaarden.*



Figuur 5.3: *In dit voorbeeld zien we de posities die berekend worden bij gebruik van zowel twee Line randvoorwaarden als twee VLine randvoorwaarden.*

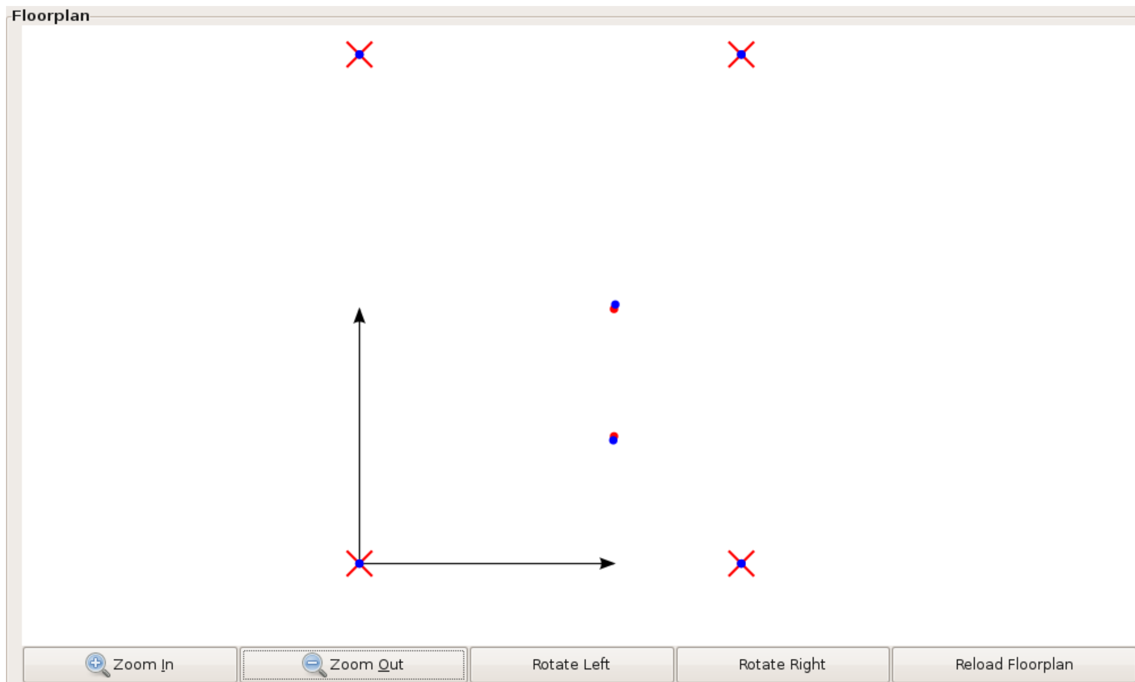
5.4.3 Voorbeeld Afstand

In deze opstelling wordt de afstandsmeting tussen de twee centrale nodes gehinderd (bijvoorbeeld door een balk die zich tussen deze twee nodes bevindt). In plaats van de werkelijke 10 meter geeft de afstandsmeting 14 meter aan. Het resultaat is te zien in figuur 5.4. De gemiddelde afwijking is 11,5 cm, hierbij worden de basisstations die als fixed tijdens de berekening beschouwd zijn (aangeduid met kruisjes) ook in rekening gebracht. Als we enkel de twee centrale nodes in rekening brengen, hebben we een gemiddelde afwijking van 34,4 cm per node.

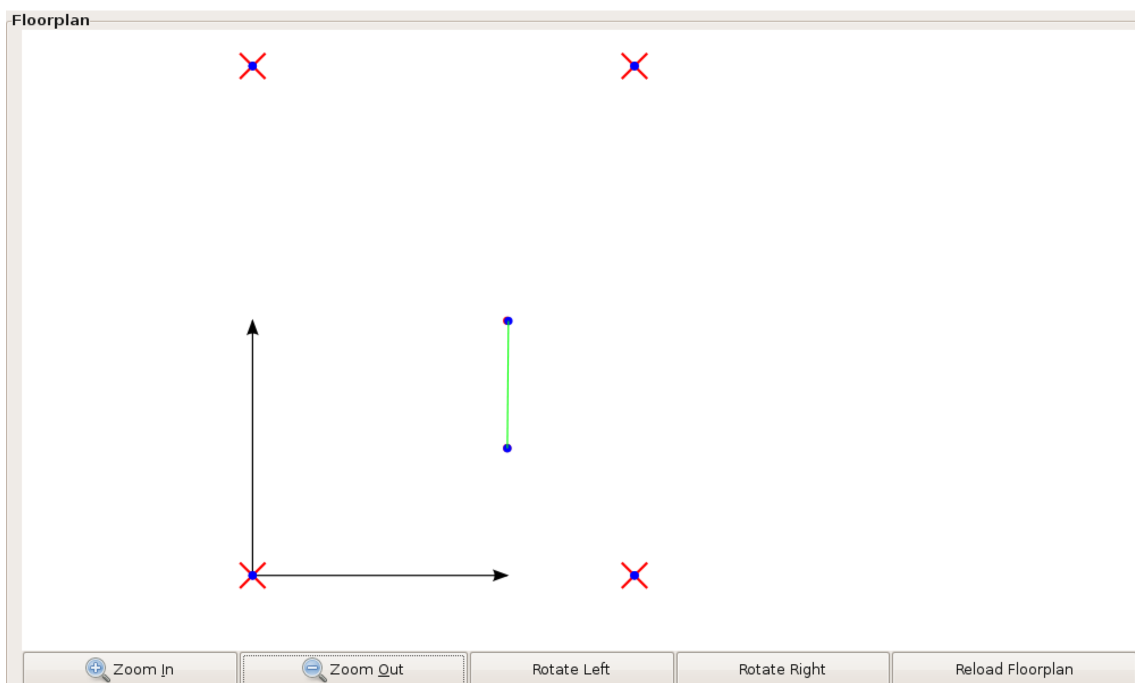
Een verbetering kan bekomen worden door de meting tussen deze twee nodes niet te gebruiken. Zo verbeteren we het resultaat tot een gemiddelde afwijking (waarbij we enkel de centrale nodes in rekening brengen) van 10,8 cm.

Een nog betere mogelijkheid is om de afstand tussen deze nodes exact te bepalen, en gebruik te maken van de randvoorwaarde Afstand. Het resultaat hiervan is weergegeven in figuur 5.5. Bij gebruik van de randvoorwaarden Afstand, verbeteren we de gemiddelde afwijking tot 5,8cm per node. Zelfs ten opzichte van de keuze om de foutieve afstandsmeting te negeren, is dit een grote verbetering in nauwkeurigheid.

In tabel 5.2 worden de resultaten van dit scenario samengevat.



Figuur 5.4: *Tussen de twee centrale nodes wordt de afstandsmeting gehinderd. Dit zorgt voor een grote fout bij de positiebepaling.*



Figuur 5.5: *We geven aan de hand van de Afstand randvoorwaarde de afstand tussen de twee centrale nodes op. Dit geeft een grote verbetering van het resultaat.*

Tabel 5.2: Overzicht van de verbeteringen met de Afstand randvoorwaarde

Randvoorwaarden	gemiddelde afwijking per node
geen	34,4cm
geen + negeren afstandsmeting	10,8cm
Afstand	5,8cm

5.5 Conclusies

In dit hoofdstuk hebben we besproken hoe we de randvoorwaarden uit hoofdstuk 4 integreren in het algoritme. We hebben besproken vanuit welke functies we een aanroep doen naar "initialise" en naar "Execute". Bij een aanroep van die functies hebben we gebruik gemaakt van drie stappen om tot een aangepaste posities/richtingsvector te komen.

Tot slot hebben we enkele voorbeelden gegeven waarin de randvoorwaarden gebruikt worden. In de besproken scenario's zien we telkens minstens een halvering van de gemiddelde afwijking. Hieruit kunnen we besluiten dat het gebruik van randvoorwaarden zeer effectief is om de positiebepaling nauwkeuriger te maken.

Hoofdstuk 6

Besluit

Het doel van deze masterproef was om randvoorwaarden toe te voegen aan een bestaande implementatie van een algoritme, zodat de positiebepaling nauwkeuriger kon gebeuren. Hierbij is het belangrijk dat dit op een doordachte manier gebeurt.

We zijn begonnen met een grondige analyse te doen van het algoritme. Hieruit konden we opmaken op welke manier we van afstandsmetingen konden komen tot een set van posities die hierbij het beste past en welke problemen kwamen kijken bij het vinden van deze oplossing. Hieruit hebben we kunnen afleiden op welke factoren we best inspelen om de resulterende set van posities bij te stellen.

De ontwikkeling van de randvoorwaarden hebben we onafhankelijk gehouden van de oorspronkelijke code. Zo behouden we de mogelijkheid om deze randvoorwaarden ook te gebruiken vanuit nieuwere releases van het algoritme voor positie-bepaling. De werking van het algoritme blijft zo ook onafhankelijk van het aantal of de types gebruikte randvoorwaarden.

De manier waarop we randvoorwaarden bijhouden in de loop van het algoritme is door ze te bewaren in een combinatie van gelinkte en circulaire lijsten. Het dynamische karakter van deze methode zorgt dat het uitbreiden van het aantal gebruikte randvoorwaarden nooit tot geheugen overschrijvingen leidt. De flexibiliteit van deze combinatie zorgt ook dat we vlot de verschillende randvoorwaarden kunnen aanspreken en deze op verschillende manieren kunnen overlopen.

Door een vaste basisklasse te gebruiken voor elke randvoorwaarde zorgen we dat het toevoegen van nieuwe randvoorwaarden beperkt blijft tot het invullen van enkele functies. Door de nodige functies van deze basisklasse ”virtual” te declareren zorgen we dat het type randvoorwaarde transparant blijft voor het algoritme. Het is pas at runtime dat bepaald wordt met welke randvoorwaarden we te maken hebben.

Tot slot hebben we het aantal toevoegingen aan het algoritme beperkt gehouden. Zo zorgen we dat de integratie van de ontwikkelde randvoorwaarden ook in nieuwere releases beperkt zal blijven.

De toevoeging van randvoorwaarden zorgt dat de resultaten van het algoritme nauwkeuriger worden. In de scenario's die we hebben aangehaald (5.4) zagen we dat de gemiddelde afwijking minstens gehalveerd werd. We kunnen hieruit besluiten dat het gebruik van

randvoorwaarden bij de installatie van het systeem, de nauwkeurigheid waarmee de basisstations bepaald worden veel verbeterd. Dit zal ook de positiebepaling van mobiele nodes ten goede komen.

Tot slot nog enkele verbeteringen die aangebracht kunnen worden

- De ”initialise” functie van de randvoorwaarde ULine opnemen in de iteratie. Hierdoor zal de stijging in nauwkeurigheid ook invloed hebben bij de positiebepaling van nodes die niet aan de randvoorwaarde hoeven te voldoen.
- Voorzien dat een node ook aan meerdere randvoorwaarden kan voldoen.

Bibliografie

Crauwels, H. (2008), *Programmeertechnieken*.

Dirkse, D. (2003), 'de kleinste kwadraten methode', http://home.hccnet.nl/david.dirkse/klkw/klkw.html.

FunctionX.inc (2005), 'Linked lists', http://www.functionx.com/cpp/articles/linkedlist.htm.

linuxtopia (n.d.), 'Downcasting', http://www.linuxtopia.org/online_books/programming_books/thinking_in_c++/Chapter15_028.html.

Press, W. H., Teukolsky, S. A., Vetterling, W. T. & Flannery, B. P. (2007), *Numerical recipes third edition*, Cambridge.

Swartz, F. (1994), 'C++ notes: Dynamic allocation of arrays', http://www.fredosaurus.com/notes-cpp/newdelete/50dynamalloc.html.

Tubbax, H., Wouters, J., Olbrechts, J., Debacker, P., Spiessens, P., Stubbe, F., Danneels, J., Bauwelinck, J., Yin, X., Torfs, G. & Vandewege, J. (2008), 'A Novel Positioning Technique for 2,4 GHz ISM Band'.

wikibooks (2008), 'C++ programming/all chapters', http://en.wikibooks.org/wiki/C\%2B\%2B_Programming/All_Chapters.

Wikipedia (2008), 'Conjugate gradient method', http://en.wikipedia.org/wiki/Fletcher-Reeves.