
UNIVERSITEIT • STELLENBOSCH • UNIVERSITY

# CAD-supported preliminary column force calculations in multi-storey buildings

Eliz-Mari Lourens

Paper presented in partial fulfilment
of the requirements for the degree of
Master of Science in Civil Engineering
at the University of Stellenbosch.

Supervisor: Dr. G.C. van Rooyen

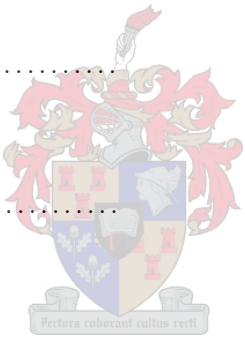
October 2006

Copyright © 2006 University of Stellenbosch
All rights reserved.

# Declaration

I, the undersigned, hereby declare that the work contained in this report is my own original work and that I have not previously in its entirety or in part submitted it at any university for a degree.

Signature: .....




Date: .....

# Synopsis

The predominately manual, time-consuming and error-prone procedure currently used in engineering offices for the calculation of preliminary column forces in multi-storey buildings constitutes the motive for the research described in this study. Identifying the current procedure as in need of improvement, techniques and prototype software posing a semi-automated alternative, are developed.

Influence areas used for load-assignment are established with the use of a Voronoi diagram calculated for a specific floor geometry. The forces transferred to the columns are based solely on the size of the influence areas thus calculated.

The definition of the floor geometry, as well as the definition of loads and other necessary input parameters, are performed in a CAD-system, into which the Voronoi functionality is integrated.

The accuracy of the forces obtained with the implemented procedure and, consequently, the accuracy of the forces as they are calculated in current practice, is determined through comparison with the results of finite element analyses. The comparative analysis of a sample of typical floor geometries allows an evaluation of the results and the identification of tendencies observed regarding the errors obtained.

It is concluded that calculating column forces based on influence areas, i.e. solving a geometrical problem without taking any stiffness properties into account, is unsafe. The implication hereof is twofold. Firstly, it serves as a warning concerning the technique currently used in practice and secondly, it steers the investigation in the direction of a finite element analysis: using the influence areas as a basis for automatic meshing, a semi-automated analysis can be performed relatively inexpensively, using plate elements.

# Samevatting

Voorlopige kolomkragte in multi-verdieping geboue word huidiglik op 'n tydrowende manier met die hand in ingenieurskantore bereken d.m.v. 'n proses waardeur foute maklik kan insluip. Hierdie feit dien as motivering vir die navorsing onderneem in hierdie studie. Deur die huidige proses te identifiseer as een wat sonder twyfel sal baat by 'n verbetering, is tegnieke en proto-tipe sagteware wat 'n semi-geautomatiseerde alternatief illustreer, ontwikkel.

Invloedsareas wat gebruik word om laste toe te ken aan die kolomme word verkry na aanleiding van 'n Voronoi diagram wat bereken kan word vir 'n spesifieke vloergeometrie. Die kragte wat oorgedra word na die kolomme word dus gebaseer op die grootte van die invloedsarea.

Die Voronoi funksionaliteit word geïntegreer in 'n CAD-sisteem om voorsiening te maak vir die geometriese definisie van die vloer, die definisie van laste, asook ander nodige invoerparameters.

Die akkuraatheid van die kragte wat verkry word d.m.v. die geimplementeerde prosedure en, gevolglik, die akkuraatheid van die kragte soos wat hulle bereken word in huidige praktyk, word vasgestel deur vergelyking met die resultate van eindige element analises. Die vergelyking maak die beoordeeling van die resultate en die identifisering van sekere neigings aangaande die foute wat gemaak word moontlik.

'n Gevolgtrekking word gemaak dat die berekening van kragte gebaseer op invloedsareas, m.a.w. waar 'n geometriese probleem opgelos word sonder dat enige styfheidseienskappe in berekening gebring word, onveilig is. Twee implikasies hiervan word identifiseer. Eerstens dien die gevolgtrekking as 'n waarskuwing rakende die tegniek wat huidiglik gebruik word in die praktyk en tweedens stuur dit die ondersoek in die rigting van 'n eindige element analise: deur die invloedsareas te gebruik as 'n basis vir automatiese netindeling, kan 'n semi-geautomatiseerde eindige element analise relatief goedkoop uitgevoer word deur gebruik te maak van plaat elemente.

# Acknowledgements

Dr. Gert van Rooyen, my supervisor. His guidance cultivated in me great respect for not only his knowledge, but also his person.
Anton Burger Eygelaar and Michael-John Deacon, for numerous helpful responses.
My father, for perspective.
John-James Burchell, Berry Meyer, my sister, Wouter Coetzee, and all my other friends, for providing excellent distraction at times when I needed it.

# Contents

**Declaration** ii

**Synopsis** iii

**Samevatting** iv

**Acknowledgements** v

**Contents** vi

**List of Figures** viii

**List of Tables** x

**1 Introduction** 1
- 1.1 Investigating the motive . . . 1
- 1.2 Defining objectives . . . 1
- 1.3 Limitations and scope of investigation . . . 2
- 1.4 Plan of development . . . 3

**2 The Voronoi diagram** 4
- 2.1 Definition . . . 5
- 2.2 Computation . . . 5
- 2.3 Implementation . . . 7
  - 2.3.1 Basic structure and initial considerations . . . 7
  - 2.3.2 Calculating the breakpoints . . . 8
  - 2.3.3 Special breakpoints: the start and end of the beach line . 12
  - 2.3.4 Representing the beach line . . . 13
  - 2.3.5 The handling of site events . . . 13
  - 2.3.6 The handling of circle events . . . 15
  - 2.3.7 Numerical issues . . . 17

**3 CAD-tools for approximate column forces** 18
- 3.1 The *CADemia* system . . . 18
- 3.2 Components . . . 19
- 3.3 Commands . . . 20
  - 3.3.1 Commands *DefineOuterBoundary* and *DefineInnerBoundary* . . . 20
  - 3.3.2 Command *ReadColumns* . . . 22

3.3.3 Command *AddColumn* . . . . . 22
3.3.4 Command *Calculate* . . . . . 22
3.3.5 Command *AddLoad* . . . . . 32
3.3.6 Command *Transfer* . . . . . 34

**4 Finite element modeling and comparative results 36**
4.1 An apartment building floor . . . . . 36
4.1.1 Model description . . . . . 36
4.1.2 The Voronoi diagram . . . . . 37
4.1.3 Comparative results . . . . . 37
4.2 Mezzanine floor of an industrial building . . . . . 40
4.2.1 Model description . . . . . 40
4.2.2 Voronoi diagram . . . . . 42
4.2.3 Comparative results . . . . . 42
4.3 A parking lot floor . . . . . 44
4.3.1 Model description . . . . . 44
4.3.2 Voronoi diagram . . . . . 44
4.3.3 Comparative results . . . . . 44

**5 Conclusions and recommendations 47**
5.1 Observed tendencies . . . . . 47
5.1.1 Columns situated on boundaries . . . . . 47
5.1.2 Hogging bending moments . . . . . 48
5.1.3 Uniform stiffness properties . . . . . 48
5.2 Aspects that need to be addressed . . . . . 49
5.3 Outstanding functionality . . . . . 50
5.3.1 Building definition . . . . . 50
5.3.2 Horizontal loading . . . . . 50
5.4 Final comments and recommendations . . . . . 50

**A Intersecting parabolas 52**

**B Midpoint of circle 55**

**C Text file format 56**

**D Parametric line-clipping 57**

**E Bounding line integration 58**

**F Layout of apartment building floor 61**

**G Comparative results for apartment building floor 62**

**H Layout of industrial floor 63**

**I Comparative results for industrial floor 64**

**J Comparative results for parking lot 65**

**Bibliography 66**

Pectora roborant cultus recti

# List of Figures


2.1 A Voronoi diagram . . . . . 5
2.2 The beach line . . . . . 6
2.3 A circle event . . . . . 7
2.4 Coordinate system . . . . . 8
2.5 The fundamental classes used to compute the Voronoi diagram . . 9
2.6 Intersections between parabolas: defining variables. . . . . 10
2.7 Definition of variables, continued. . . . . 10
2.8 Determination of breakpoint coordinates. . . . . 11
2.9 The start and end of the beach line. . . . . 12
2.10 Midpoint of circle through three points . . . . . 14

3.1 Components . . . . . 19
3.2 Parametric line clipping . . . . . 23
3.3 Creating the normal vector . . . . . 25
3.4 Distinguishing between segments . . . . . 26
3.5 Multiple areas of a single cell . . . . . 27
3.6 Building the Voronoi cells . . . . . 28
3.7 Assigning sides . . . . . 29
3.8 A floor geometry divided into influence areas based on the Voronoi diagram. . . . . 33
3.9 Adding a load and transferring the forces. . . . . 33

4.1 Mesh and boundary conditions for the apartment building floor. . 37
4.2 Voronoi diagram of the apartment building floor . . . . . 38
4.3 Displacement contour plots for the apartment building floor. . . . . 39
4.4 Deformed shape, applying a deformation scale factor of 1200. . . . 39
4.5 Hogging moments. . . . . 40
4.6 Surface outlines and boundary conditions for the industrial floor. . 41
4.7 Coffer dimensions and volume . . . . . 42
4.8 Displacement contour levels for the industrial floor. . . . . 43
4.9 Deformed shape, applying a deformation scale factor of 500. . . . . 43
4.10 Mesh and boundary conditions for the parking lot. . . . . 44
4.11 Voronoi diagram of the parking lot floor. . . . . 45
4.12 Displacement contour levels for the parking lot. . . . . 45
4.13 Deformed shape, applying a deformation scale factor of 500. . . . . 46

5.1 Extract from table of beam diagrams. . . . . 48
5.2 Horizontal Loading . . . . . 51

A.1 Intersections between parabolas: defining variables. . . . . 52
A.2 Definition of variables, continued. . . . . 53

B.1 Defining variables . . . . . 55

D.1 Parametric line-clipping . . . . . 57

# List of Tables


4.1 Average errors for the apartment building floor. . . . . 40
4.2 Average errors for the industrial floor. . . . . 43
4.3 Average errors for the parking lot. . . . . 45

5.1 Boundary vs internal forces. . . . . 48
5.2 Uniform vs non-uniform stiffness properties. . . . . 49
5.3 Percentage of computation time spent on a task. . . . . 49

# Chapter 1

# Introduction

In civil engineering practice, values of column forces in multi-storey buildings are often required before any detailed analysis of the structure has been performed. One of the reasons for this arises from the fast-tracked nature of the majority of construction projects: foundations are laid and base columns constructed whilst analysis and design are still in progress. A need for quick results when feasibility studies are performed or when evaluating the effect of design changes on supporting columns form other situations in which column forces are required, but where performing a detailed analysis to get these forces seems superfluous. Thus it was concluded that the development of an efficient tool for approximate column force calculations, in which the extensive input required in a finite element analysis is to be avoided, would be highly beneficial.

## 1.1 Investigating the motive

In an attempt to gain a better understanding of the problem, time was spent investigating current consulting engineering practice regarding the computation of preliminary column forces. It soon became clear that, if a tool capable of calculating these forces in an efficient manner could be developed, it would certainly serve as an improvement over the largely manual, time-consuming and error-prone procedure currently used: influence areas for the columns and/or walls of a floor are approximated using rulers and hand-held calculators and the forces thus obtained are accumulated with the use of spreadsheets.

## 1.2 Defining objectives

Having confirmed the motive for developing an efficient solution, the objectives of the task could be defined. These objectives were to:

1. Identify a way in which to efficiently calculate preliminary column forces without performing a detailed finite element analysis.
2. Implement the proposed procedure in prototype software.
3. Model a sample of typical floor geometries using the proposed procedure.

4. Model the sample using finite element software.

5. Evaluate results and identify any tendencies regarding the errors obtained.

## 1.3 Limitations and scope of investigation

It is to be expected that a complete solution of the stated problem extends beyond the scope of what can be achieved in a single master's thesis. As an initial approach it was decided to adopt the consulting engineering practice approach, i.e. to compute influence areas for load-bearing elements, but to do this in an efficient and optimal way. The result is that a geometrical problem is to be solved, without taking any stiffness properties into consideration. Furthermore, only vertical loading is to be considered. This leaves ample room for further study on this subject. For instance, if a manner in which to efficiently mesh the obtained influence areas can be developed, a relatively simple finite element analysis using plate elements can be performed and results may be drastically improved without the need for any additional input. This topic forms but one for future investigation and more shall be discussed at a later stage.

The stated objectives steers the investigation in the direction of two fields, namely that of dealing with geometric entities in a Computer Aided Draughting (CAD) environment, and computational geometry of planar surfaces.

***CAD:***

The load-bearing mechanisms of the buildings under consideration are identified in CAD drawings. In most cases the drawings are also available, and manipulated, in digital form using CAD software like AutoCAD[1]. It is obvious that an efficient solution to the stated problem has to be embedded in CAD software, thus extending the identification and design of the structural entities in a natural way. Consequently, it was decided to use a CAD environment which supports easy programmatic access for the development of the prototype software. This would suitably demonstrate the proposed solution and would also allow further development for supporting the transfer of information to and from other CAD systems. The *CADemia* system[2] developed at Bauhaus Universität Weimar, Germany, provides a CAD kernel with all the features required for the proposed project. It was thus decided to use this system as the basic CAD environment, and to extend it to provide the special features required for the proposed solution. Details of *CADemia* and the implemented extensions are described in later chapters.

***Computational geometry:***

The assignment of loads transferred by a floor to its supporting columns requires the solution of a number of problems pertaining to the geometry of the floor under consideration, and details of the loading applied to the floor. A survey of computational geometry literature[3, 4, 5] revealed a computable geometric structure, called the Voronoi diagram, which was considered optimal for the purpose of determining influence areas. This discovery essentially meant that a semi-automatic procedure could be developed which would remove the responsibility to draw influence areas from the engineer. Other geometric prob-

lems, for example the computation of the area of an arbitrarily shaped closed region or the intersection of loaded areas and influence areas, etc., would also have to be dealt with. The various problems, and their proposed solutions, are discussed in later chapters.

## 1.4 Plan of development

In the first chapter the Voronoi diagram is defined and the algorithm used for its computation, as well as the implementation of this algorithm, is described. The next chapter is dedicated to explaining how the Voronoi functionality was attached to a CAD-system. The chapter focuses on the development of numerous tools that were required in order to make the Voronoi functionality useful for computing column forces in engineering offices. The results obtained by performing finite element analyses of a chosen set of floors, and a comparison of these forces to the forces obtained using the prototype software, are presented in Chapter 4. The last chapter deals with the tendencies observed during a comparative analysis. The chapter also includes the mentioning of certain aspects that have to be addressed and outstanding functionality. Conclusions are drawn and recommendations made.

# Chapter 2

# The Voronoi diagram

The primary objective of the study is to identify a way in which to efficiently calculate preliminary column forces in multi-storey buildings. Column forces *per se* can be computed by performing a finite element analysis of the complete structure. However, this is certainly not an efficient way in which to determine preliminary results given the current tools supporting finite element modeling. Limited support is available for transferring geometric information from CAD files to FEM software and, furthermore, automated mesh generation is inadequate. As a result, the execution of a finite element analysis of a building is time consuming.

Two possible courses of action were identified:

1. *Automated finite element analysis*
   The performance of a simple, largely automated finite element analysis for which user input is minimized, represents a valid option. However, automated meshing is the topic of extensive research worldwide and was considered to fall outside the scope of this investigation. Furthermore, it was hoped that the geometric approach described below might provide insights that could be useful in automated meshing of building floors, and in that way pave the way for further investigations.

2. *Geometric solution*
   A geometric solution requires the identification of a way in which to mimic the procedure that is currently used in practice, namely the assignment of influence areas to columns. Such a solution would only be useful provided that the procedure can be automated and implemented using digital CAD data.
   This approach was chosen for the research described in this thesis.

Having chosen to solve the problem geometrically, an investigation to find a method to automate the subdivision of a floor geometry into influence areas was undertaken. The definition of the Voronoi diagram in Section 2.1 below makes it clear why this structure was considered an optimal solution for the creation of such influence areas. A Voronoi diagram is a versatile geometric structure and has applications in many more fields including social geography, physics and astronomy. In this application the diagram, by means of the subdivision of

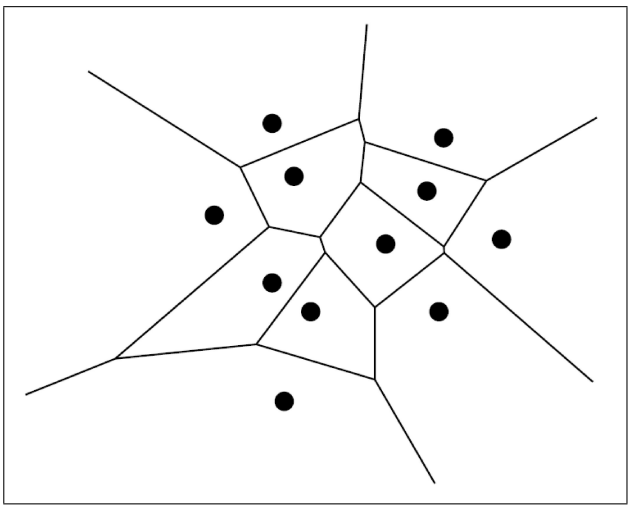


Figure 2.1: A Voronoi diagram

the floor into influence areas, provides a basis for automatic load assignment.

This chapter is dedicated to providing some theoretical background regarding these diagrams and to describing their computation and the way in which it was implemented in the prototype software.

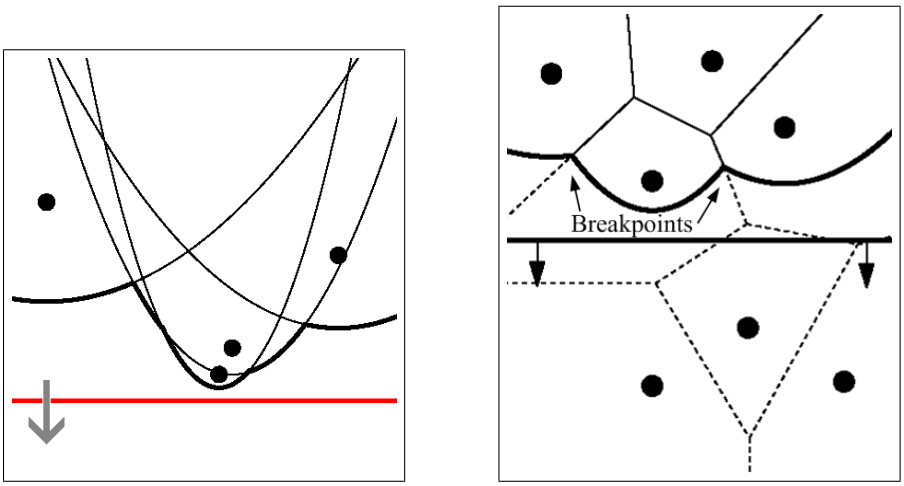
## 2.1 Definition

Let $P := p_1, p_2, \ldots, p_n$ be a set of $n$ distinct points in the plane. Each of these points represents a site. The Voronoi diagram of $P$ is defined as the subdivision of the plane into $n$ cells, a cell for each site in $P$, with the property that a point $q$ lies in the cell corresponding to a site $p_m$ if and only if $dist(q, p_m) < dist(q, p_n)$ for each $p_n \in P$ with $m \neq n$.

Considering Figure 2.1, it can be seen that the Voronoi diagram consists of a) edges, forming bisectors of pairs of sites and b) vertices, defining intersections between these bisectors.

## 2.2 Computation

The algorithm used to compute the Voronoi diagram, a plane sweep algorithm commonly known as Fortune's algorithm, has been identified as optimal[3]. The algorithm involves sweeping a horizontal line - the sweep line - from top to bottom over the plane. As the sweep line moves downwards over the plane it strikes certain special points referred to as the event points. Relevant information regarding the structure of the diagram (i.e. its edges and vertices) is stored at these specific occurrences only, since it is only at the event points that the diagram structure changes. The following paragraphs are dedicated to explaining the algorithm in more detail.




(a) (b)

Figure 2.2: The beach line

A fundamental concept of the algorithm is the formation and development of the beach line: a line situated above the sweep line and consisting of a sequence of parabolic arcs. Referring to Figure 2.2a, it is easy to visualize that each parabola corresponding to a certain site $p_i$ above the sweep line bounds the locus of points that are closer to $p_i$ than to the sweep line. Thus the beach line can be seen as the line bounding the locus of points that are closer to a respective site above the sweep line than to the sweep line itself. The beach line is defined as the line passing through, for each $x$-coordinate, the lowest point of all parabolas.

As the sweep line moves from top to bottom the edges of the Voronoi diagram are traced out by the breakpoints separating the different parabolic arcs that form the beach line. Refer to Figure 2.2b for clarity.

The previously mentioned event points represent the points where, during the planar sweep, the combinatorial structure of the beach line changes. The first type of event, a site event, occurs when the sweep line strikes a new site. At this instance a new parabolic arc is formed which at first is simply a vertical line segment connecting the site to the beach line. As the downward movement of the sweep line progresses, this new parabola widens and the breakpoints at its beginning and end start tracing out a new edge. The edge is initially not connected to the rest of the Voronoi diagram above the sweep line and it is when such growing edges connect to form a vertex, that the second type of event, a circle event, takes place.

Referring to Figure 2.3, a circle event marks the disappearance of a parabolic arc from the beach line. This happens at an instant where a circle passing through points $p_i$, $p_j$ and $p_k$, with the vertex $q$ at its centre and its lowest point on the sweep line, can be drawn. To be more specific, a circle event can be

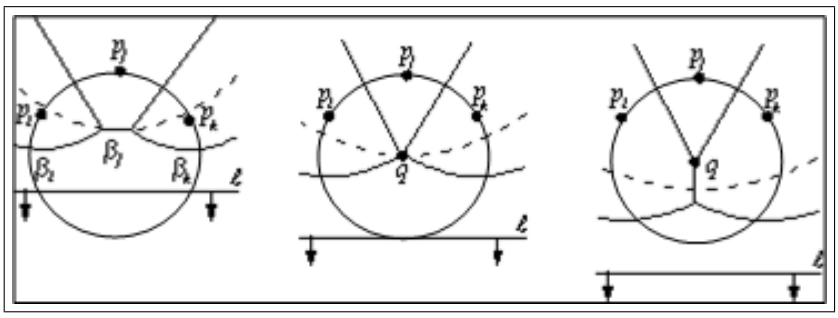


Figure 2.3: A circle event

defined as an event where the sweep line reaches the lowest point of a circle through three sites defining consecutive arcs on the beach line.

The construction of the Voronoi diagram can now be perceived to consist of a sequence of a) site events, when new edges start to grow and b) circle events, when two growing edges meet to form a vertex.

## 2.3 Implementation

### 2.3.1 Basic structure and initial considerations

As a starting point on implementation the structure of Fortune's algorithm is presented. Figure 2.4 defines the coordinate system used. The algorithm's basic strategy is as follows:

**while** *the event queue is not empty*
  **do** *remove the event with the largest y-coordinate (i.e. the next event) from the queue*
    **if** *the event is a site event*
      **then** *handleSiteEvent()*
      **else** *handleCircleEvent()*

It is necessary to set up two initial data structures before entering into this loop:

1. *The initial event queue*
   The initial event queue consists only of site events and is formed by simply sorting all defined sites in terms of descending y-coordinates. The circle events are detected and added to the queue only once the planar sweep has commenced.

2. *The initial beach line*
   A data structure representing the beach line has to be created before the planar sweep starts. This requires the creation of an arc for the topmost site or, in the case where there is more than one topmost site, the creation of multiple arcs for these sites. The topmost site(s) are identified from the initial event queue and stored in a list. Their coordinates are used

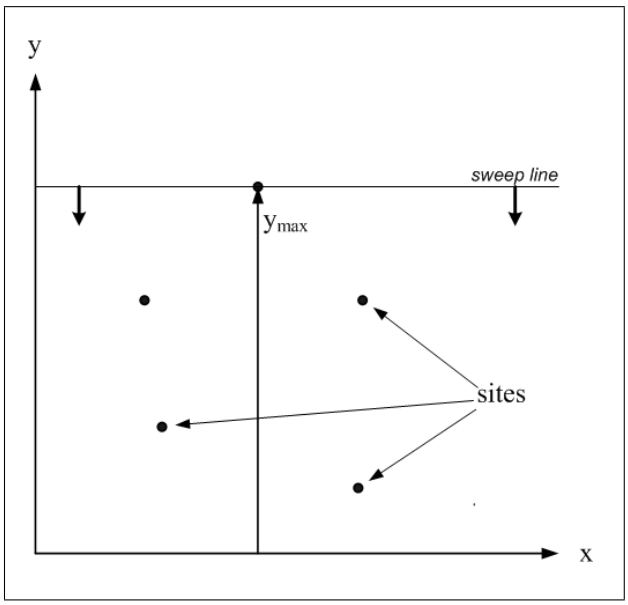



Figure 2.4: Coordinate system

to create an initial beach line where the position of the sweep line is taken as the position of the first site in the initial event queue that is not contained in the list of topmost site(s). This implies that the breakpoints separating the arcs of the initial beach line are calculated as if the sweep line is already in its 'second' position. Thus, when subsequently entering the abovementioned loop representing the planar sweep, these top site(s) are not handled and the sweep starts with the beach line in its second position.

Figure 2.5 shows the fundamental classes used to compute the Voronoi diagram. The following paragraphs are dedicated to explaining these classes, their key methods and the difficulties that surfaced during their development.

### 2.3.2 Calculating the breakpoints

A breakpoint is defined by two sites, the bisector of which it traces out. These two sites are used to calculate the breakpoint's coordinates which, of course, are different for every new position of the sweep line. Thus the *breakpoint()* method receives as parameter the current position of the sweep line and uses it to calculate the breakpoint's coordinates as follows (refer to Figure 2.6 and Figure 2.7):

$$X_0 = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \text{ and } Y_0 = \frac{(p_{jx} - X_0)^2}{2(p_{jy} - l_y)} + \frac{1}{2}(l_y + p_{jy})$$

$$\text{where} \quad a = \quad 1 - \frac{p_{iy} - l_y}{p_{jy} - l_y};$$

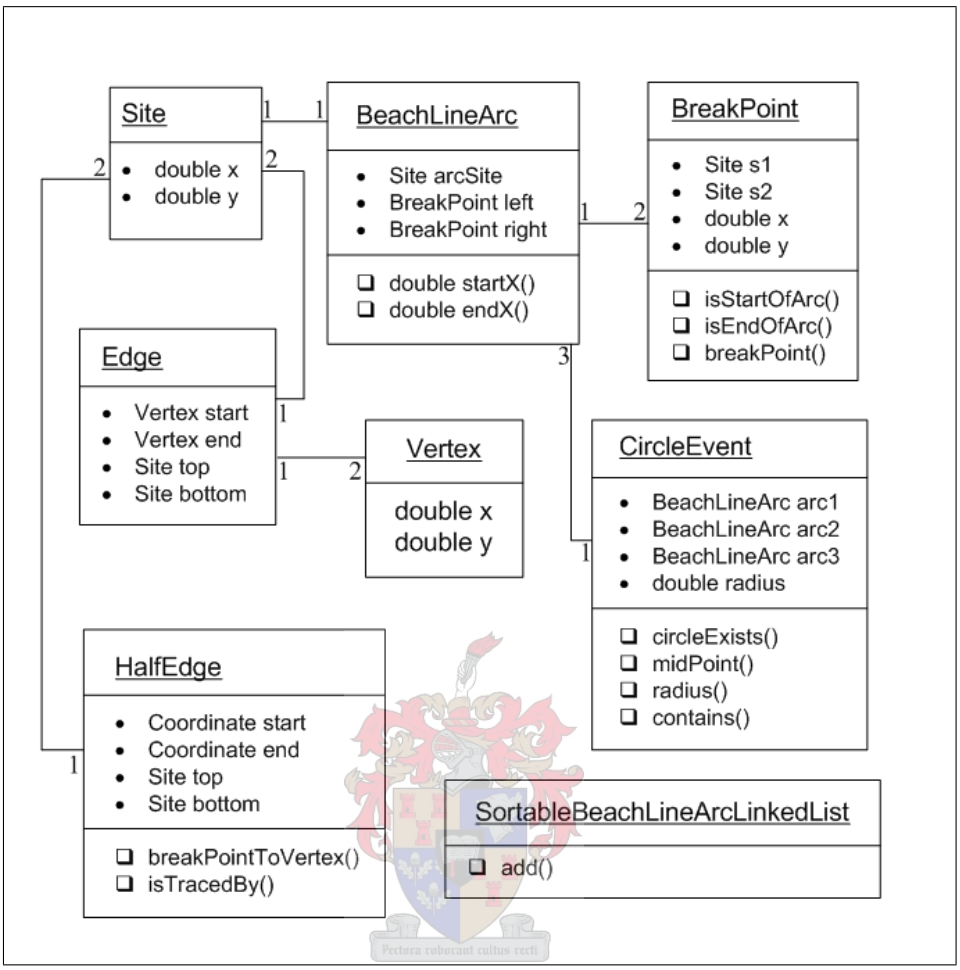



Figure 2.5: The fundamental classes used to compute the Voronoi diagram

$$b = 2p_{jx}\frac{p_{iy}-l_y}{p_{jy}-l_y} - 2p_{ix} \text{ and}$$
$$c = {p_{ix}}^2 - {p_{jx}}^2\frac{p_{iy}-l_y}{p_{jy}-l_y} - (p_{iy}-l_y)(p_{jy}-p_{iy}).$$

The derivation of these equations are presented in Appendix A.

Two possible values for $X_0$, and its corresponding $Y_0$, can be calculated, each defining one of the intersection points of the two parabolas, and it remains to be determined how the correct coordinate will be identified. Consider the creation of *BreakPoint* objects:

$\ldots =$ *new BreakPoint*$(s_1, s_2)$
*OR*
$\ldots =$ *new BreakPoint*$(s_2, s_1)$

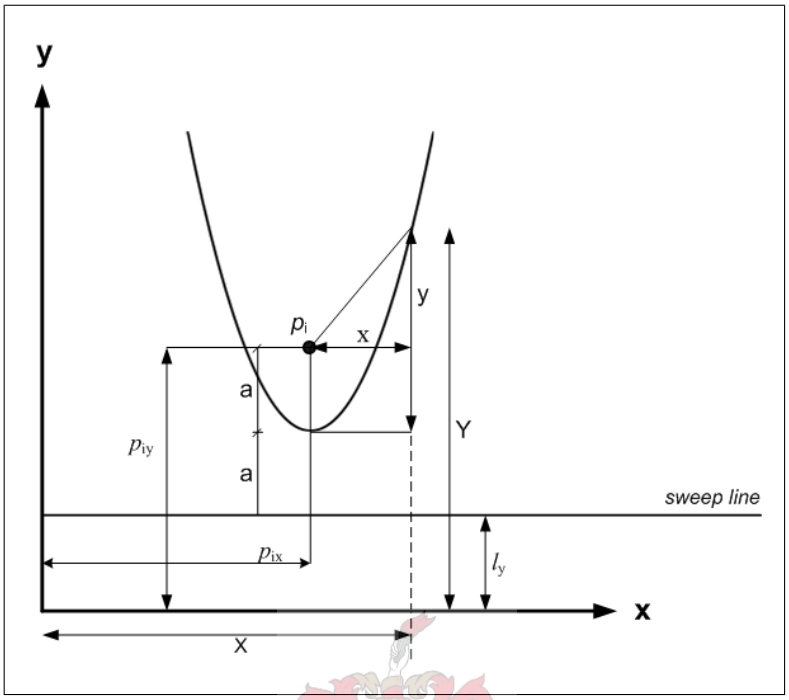



Figure 2.6: Intersections between parabolas: defining variables.

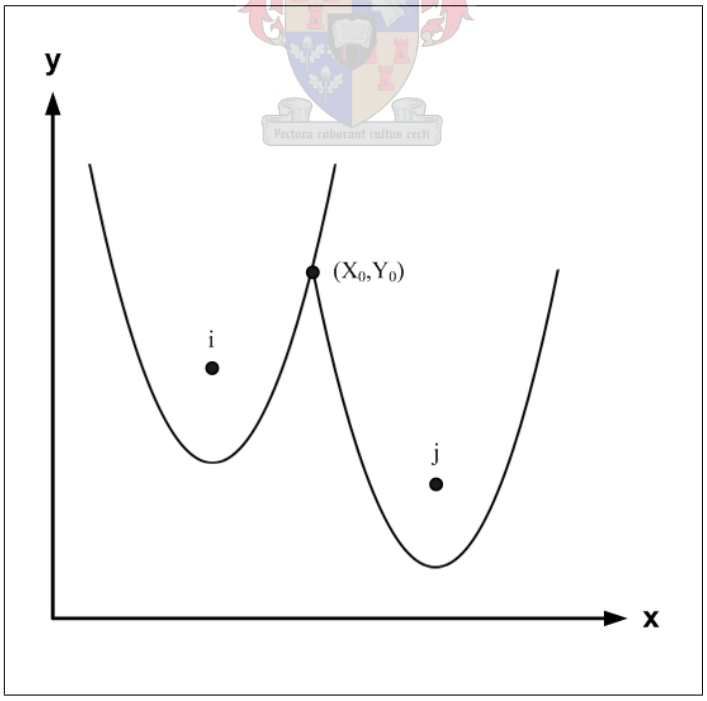



Figure 2.7: Definition of variables, continued.

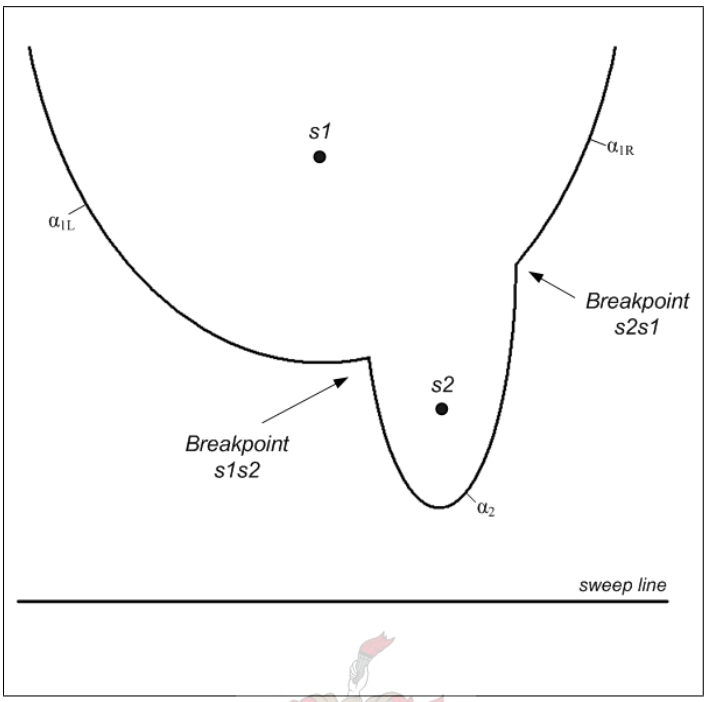



Figure 2.8: Determination of breakpoint coordinates.

Depending on the parameter sequence in which the *Site* objects are entered to create the breakpoint, the breakpoint will have an 'internal' site $s1$ and $s2$, e.g. the last breakpoint created in the example above will have site $s2$ as its 'internal' site $s1$. If the two possible breakpoints are then defined as illustrated in Figure 2.8, where breakpoint $s1s2$ represents the beach line transition from $\alpha_1$ to $\alpha_2$ and breakpoint $s2s1$ the transition from $\alpha_2$ to $\alpha_1$, the appropriate breakpoint can be identified as follows:

*IF internal site* $s1 <$ *internal site* $s2$
    *return* $s1s2$ *as the coordinates*
*IF internal site* $s1 >$ *internal site* $s2$
    *return* $s2s1$ *as the coordinates*

where *site* $s1 <$ *site* $s2$ if $s1_y > s2_y$, i.e. the site event $s1$ will take place before the site event $s2$.

There is one more aspect that can be mentioned regarding the *breakpoint()* method, namely the handling of certain special cases. The abovementioned equations don't hold true and special provision has to be made in the following situations:

1. *The y-values of the two sites are the same.*
   In this case the x-coordinate of the breakpoint cannot be calculated normally since $\frac{p_{iy}-l_y}{p_{jy}-l_y} = 1$ and thus $a = 0$. Instead it is calculated as follows:

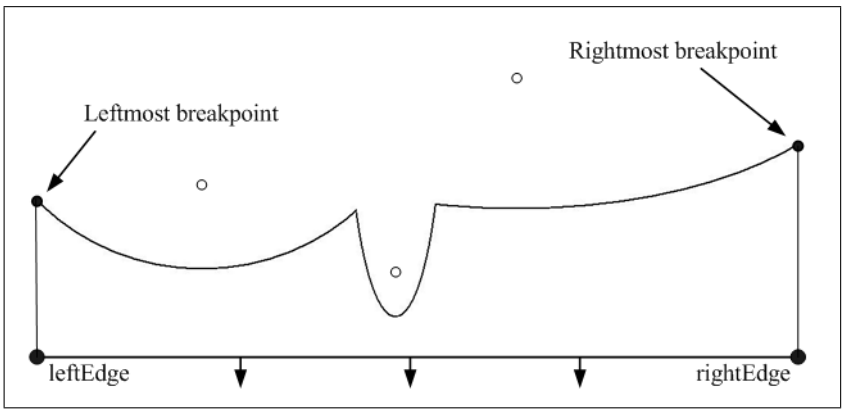



Figure 2.9: The start and end of the beach line.

$$X_0 = \frac{-c}{b}$$

2. *Both sites and the sweep line lie on the same y-value.*
   The breakpoint lies mid-way between the two points. The scenario requires special handling since $\frac{p_{iy}-l_y}{p_{jy}-l_y}$ cannot be calculated.

3. *The sweep line coincides with $p_j$.*
   At this instant the parabola defined by $p_j$ is a vertical line extending upwards from $p_j$ and the breakpoint is therefore directly above $p_j$. Again, this scenario requires special handling since $\frac{p_{iy}-l_y}{p_{jy}-l_y}$ cannot be calculated.

### 2.3.3 Special breakpoints: the start and end of the beach line

The previous section dealt with the calculation of breakpoints, specifically the mathematics behind the *breakpoint()* method and the important role played by the sequence in which *Site* objects are passed on to a breakpoint's constructor. This section presents another aspect of breakpoint implementation.

The beach line extends indefinitely to the left and right and it was therefore necessary to create two special breakpoints representing the start and end of the beach line. Since a breakpoint can only be defined by two sites, it was necessary to create a fictitious leftmost and rightmost site. These *leftEdge* and *rightEdge* objects, illustrated in Figure 2.9, are in fact not stationary and their coordinates are updated at each new position of the sweep line: they are given constant $x$-coordinates of $-\infty$ and $+\infty$, respectively, and their y-coordinates correspond to the current position of the sweep line. By placing them on the sweep line one ensures that the leftmost and rightmost breakpoint will also have $x$-coordinates of $-\infty$ and $+\infty$, respectively, because the parabolas used to calculate these breakpoints will always be vertical lines extending from the fictitious sites to the beach line.

### 2.3.4 Representing the beach line

The beach line, being comprised of a certain number of parabolic arcs, is represented by a *LinkedList* object that was extended to allow for a special way in which to add objects to the list. When a new site is encountered and a new parabolic arc is formed the composition of the linked list has to be altered. In the position where the new arc cuts into the beach line, three new parabolic arcs are created where originally there was only one. The original arc is dissected by the new arc.

It is worth mentioning that this replacement procedure causes the reference of the original arc to be lost, which, as will be explained at a later stage, plays an important role when handling circle events.

### 2.3.5 The handling of site events

The procedure to handle a site event is defined as follows:

1. Search the list of arcs to find the arc vertically above the current site. If a) the event queue contains a circle event that has a pointer to this arc and b) the current site lies within such a circle, the circle event must be removed from the queue.
2. Create the new parabolic arc for the site and place it into the arc list in the correct position.
3. Create a new instance of a *HalfEdge* object for the bisector separating the current site from the site responsible for the parabolic arc situated vertically above the current site.
4. Detection of new circle events: Checks are performed on two sets of triples of consecutive arcs in the arc list. The triple that has the new arc as its left arc is checked to see whether the breakpoints converge, followed by a check on the triple where the new arc is the right arc. If a circle event is detected, it is inserted into the event queue.

The definition of the first step in the list above cost a great deal of time since it differs from the definition specified in the literature used. The additional constraint, b), was added to the first step when it became clear that circle events were being lost. The following paragraphs serves as an attempt to explain the reason for this.

The first step originally stipulated that all circle events that have pointers to the arc vertically above the current site should be deleted. Consider the following:

- Circle events involve the meeting of two edges that are growing towards a common point, or vertex.
- Once the circle event has taken place, the two edges aren't growing anymore, but instead they have been replaced by a newly created edge that now grows from the vertex.

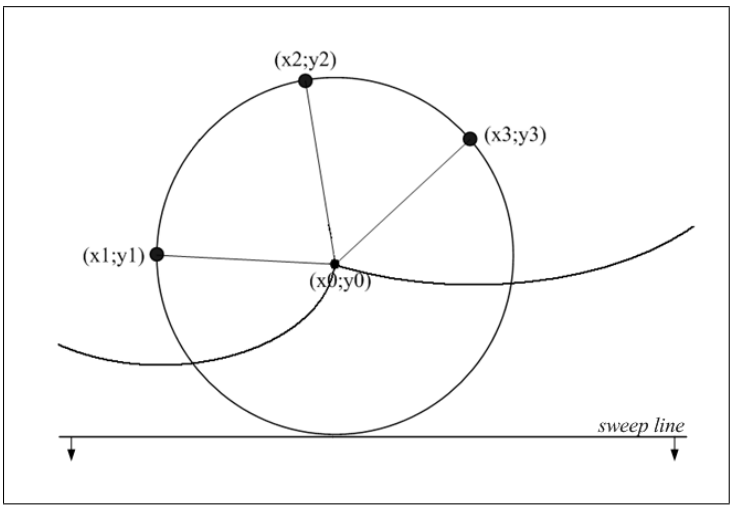



Figure 2.10: Midpoint of circle through three points

Consequently, it is possible for a circle event to render another circle event impossible. This depends on the order in which these circle events would take place, and typically happens when the first circle event connects edges that are also involved in the second event in such a way that the meeting of edges defined by the second circle event can no longer take place. The original definition implied that the scenario sketched above always occurs and that the circle event/s that would be detected in step 4, for instance, would in all cases overrule the circle events of step 1. This was found to be untrue. Instead it was found that the circle event should not be deleted in step 1 *if the current site lies outside such a circle*, because in these situations the meeting of edges should still take place. It immediately became apparent, once this modification was implemented, that there was indeed a flaw in the original step definition. The problem of edges not being terminated in vertices, and thus growing indefinitely, was solved.

Step two has already been explained in section 2.3.4, which brings us to step three: creating new *HalfEdge* objects. *HalfEdge* objects could have as their start and end attributes either *BreakPoint* or *Vertex* objects. Initially they are constructed using only two breakpoints and the references of the two sites of which they form the bisector, but through the occurrence of circle events one of their endpoints could be made permanent, i.e. could be transformed into a vertex. If more than one circle event operates on the same half edge then its start- as well as its endpoint would eventually be turned into vertices, thus transforming it into an *Edge* object. This process will shortly be explained in more detail.

Step 4 involves the detection of circle events in which a circle event is added to the event queue if the following is true (refer to Figure 2.10 for the definition of variables):

$$a_1 b_2 - b_1 a_2 < 0 \tag{2.1}$$

$$\begin{aligned} \text{where} \quad a_1 &= x_2 - x_1; \\ a_2 &= y_2 - y_1; \\ b_1 &= x_3 - x_2 \text{ and} \\ b_2 &= y_3 - y_2. \end{aligned}$$

The coordinates of the midpoint can then be calculated as follows:

$$\begin{Bmatrix} x_0 \\ y_0 \end{Bmatrix} = \frac{1}{a_1 b_2 - b_1 a_2} \begin{bmatrix} b_2 & -a_2 \\ -b_1 & a_1 \end{bmatrix} \begin{Bmatrix} c_1 \\ c_2 \end{Bmatrix}$$

where, additionally,

$$\begin{aligned} c_1 &= \frac{1}{2}(x_2^2 - x_1^2 + y_2^2 - y_1^2) \text{ and} \\ c_2 &= \frac{1}{2}(x_3^2 - x_2^2 + y_3^2 - y_2^2) \end{aligned}$$

It should be noted that inequality (2.1) is only valid for a coordinate system in which the positive $x$ and $y$ axes extend to the right and top, respectively. Flipping such a coordinate system around the $x$-axis would require the opposite inequality sign in the above equation.

The derivation of these equations are presented in Appendix B.

### 2.3.6 The handling of circle events

A circle event is handled in six steps:

1. Remove a) the disappearing arc from the arc list and b) all circle events that has a pointer to it from the event queue. Create a new breakpoint using the sites responsible for the arcs on each side of the disappearing arc.

2. Create a new vertex at the center of the circle causing the event and add it to the set of vertices.

3. Call *breakPointToVertex()* on the connecting half-edges.

4. Update the breakpoint references of the side arcs.

5. Use the new breakpoint (defined in 1.) and the new vertex (defined in 2.) to create a new *HalfEdge* record starting at the vertex.

6. Perform checks and take action in a manner analogous to step 4 under *handleSiteEvent()*, but for the triples of consecutive arcs use the triple that has a) the former left neighbor as its middle arc and b) the former right neighbor.

There are two points worth elaborating on here. Firstly, either the start- or the endpoint of each of the two connecting half-edges is transformed from a breakpoint to a vertex in step three. To assure that the correct breakpoint is transformed it is necessary to have the old breakpoint object as an input parameter for the *breakPointToVertex()* method so that the choice between breakpoints can be made at reference level.

```
public void breakPointToVertex(Vertex v, BreakPoint bp){
        ... if(start.equals(bp)){
                start = v;
                hasBeenClosed = true;
            }
        ... if(end.equals(bp)){
                end = v;
                hasBeenClosed = true;
            } ...
    }
```

Furthermore, this method should allow for the deletion of the half-edge and its subsequent replacement with a new *Edge* object in the case where both its start- and endpoints have been transformed into vertices, i.e. when *breakPoint-ToVertex()* is called for a second time.

```
        ... if(start instanceof Vertex && end instanceof Vertex){
                toBeRemoved.add(this);
                Edge e = new Edge((Vertex) start, (Vertex) end, top, bottom);
            } ...
```

The second point concerns step four, where one of the breakpoint references of each of the side arcs has to be updated. This poses a problem if one does not take into consideration the fact that, when a site event is handled, some arc references are lost. This makes it necessary to search through the arc list each time a breakpoint reference has to be updated and to then identify the breakpoint that should be updated by comparing it to the old breakpoint at reference level.

```
BreakPoint newBP = newBreakPoint(leftSite, rightSite);
...
BreakPoint oldBreakPointLeft = ((BeachLineArc) ((CircleEvent) currentSite).arc1).right;
BreakPoint oldBreakPointRight = ((BeachLineArc) ((CircleEvent) currentSite).arc3).left;
for(int z=0;z¡arcList.size();z++) {
        BeachLineArc arc = (BeachLineArc) arcList.get(z);
        if(arc.right.equals(oldBreakPointLeft)) {;
                arc.right = newBP;
        }
        if(arc.left.equals(oldBreakPointRight)) {
                arc.left = newBP;
        }
}
```

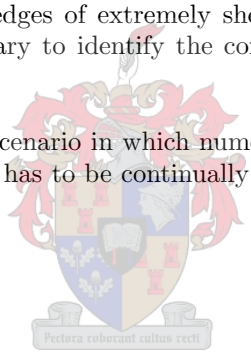
### 2.3.7 Numerical issues

Whether it is the determination of the event queue, the calculation of breakpoints or the detection of circle events, provision has to be made for loss of numerical accuracy caused by limited storage. In section 2.3.6, for instance, the necessity to make a choice between breakpoints on reference level in the *breakPointToVertex()* method was discussed. The alternative to this would be to identify a certain breakpoint by testing whether it falls within some 'small' circle drawn around the vertex it is growing towards. Although this initially seems like a valid solution one realizes, upon reflection, that there are numerical issues working against such a solution. Consider the following:

- Circle events are calculated using the raw data i.e. the defined coordinates of the site events. The coordinates of the vertex are thus calculated relatively accurately.
- Breakpoints, however, are calculated and re-calculated as parabola intersection points during the planar sweep, using a specific sweep line position.

These considerations, together with the fact that it is possible, for certain groupings of sites, to have edges of extremely short lengths, make it easy to understand why it is necessary to identify the correct breakpoint through its reference.

This illustrates but one scenario in which numerical considerations become important. Loss of accuracy has to be continually accounted for during implementation.



# Chapter 3

# CAD-tools for approximate column forces

The primary objective of the research was to identify a way in which to efficiently calculate preliminary column forces. A choice was made to find a geometric solution to this problem which would serve as a semi-automated alternative to the procedure used in current engineering practice. Voronoi diagrams were identified as ideal equipment for this task and a tool for calculating these diagrams given a set of predefined sites was developed. The question of how the Voronoi functionality can be made useful to compute column forces in engineering offices is addressed in this chapter.

The data available to solve the column-force problem comprises the CAD drawings of floor geometries. In many instances these drawings still have a preliminary status. The Voronoi functionality has to be used to geometrically divide each floor into influence areas, an area for each column. Consequently, a choice was made to proceed by attaching the Voronoi functionality to a CAD system. This would not only permit the geometrical definition of the floor, but would also provide an environment in which loaded areas could be defined.

This chapter is dedicated to explaining the integration of the Voronoi functionality into the chosen CAD system, as well as all subsequent tasks that have to be performed in order to arrive at a set of preliminary column forces.

## 3.1 The *CADemia* system

The *CADemia* system[2] is an open source CAD system under development at the Bauhaus University Weimar (BUW), Germany. One of the features of the system is its problem-orientated extensibility, which makes it ideal for use in the application under consideration. What is essentially required from the CAD system is the provision of an environment in which not only the Voronoi diagram can be calculated and displayed, but in which floor geometries, columns and loads can be defined. *CADemia*'s architecture applies the Model-View-Controller (MVC) paradigm, in which model components, their geometrical representation, and the commands that operate on them are clearly separated.

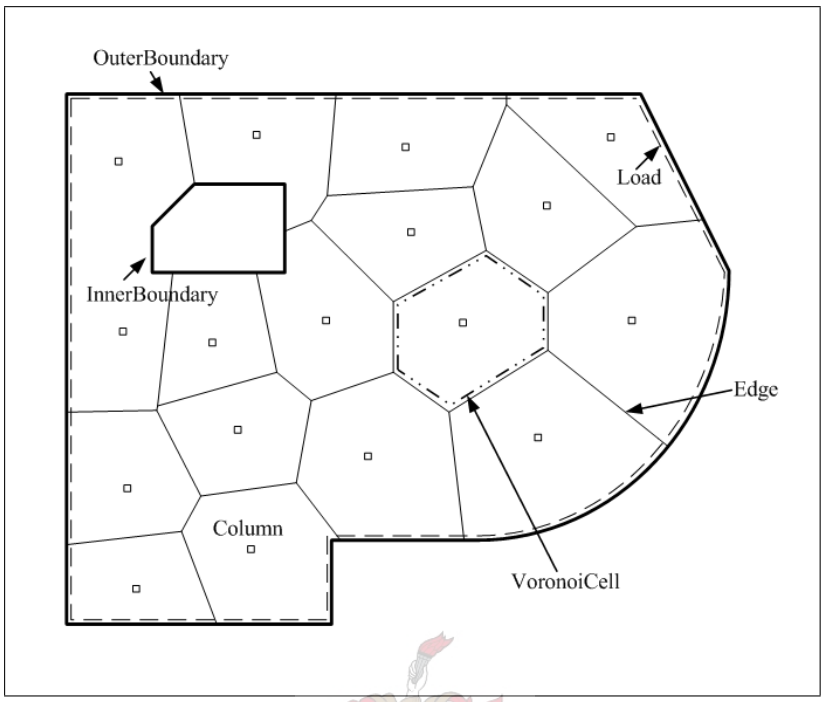



Figure 3.1: Components

In this way it provides an existing structure to which the intended functionality can be added. New components, e.g. columns and loads, and commands to define them, are integrated into *CADemia* by extending superclasses and implementing interfaces. The added components automatically become part of the CAD database.

The extensions to the system can be divided into two categories, namely

- the components added to the model and
- the commands added to define them.

## 3.2 Components

Referring to Figure 3.1, the following components are required:

***Column*** Represents columns and load-bearing walls in the structure and are the 'sites' of the Voronoi diagram.

***OuterBoundary*** Represents the outer boundary of a floor.

***InnerBoundary*** Represents the stairwells, lift shafts, etc. which cause openings in a floor.

***Edge*** Represents the edges of the Voronoi diagram.

***HalfEdge*** Represents the half-edges of the Voronoi diagram which are eventually transformed into *Edge's*.

***VoronoiCell*** Represents a closed influence area for a given column.

***Load*** Represents a loaded area.

The standardized structure of all components in *CADemia* is enforced through the *Component* interface, where the following methods are defined:

- *transformBy()*
- *getShapes()*.

These methods allow, respectively, for the transformation of a component and its visual representation by the graphical user interface. All of these classes were all either directly or indirectly made to implement *Component*. For the boundary classes, as well as classes *VoronoiCell* and *Load*, the implementation was done indirectly, i.e. by extending an already existing component, namely *ComponentGeneralPath*.

The commands required to instantiate these components, as well as commands providing other functionality, are discussed below.

## 3.3 Commands

Commands are represented by command objects. They represent user interaction and are used to insert, remove or change elements of the model. Commands are integrated into the *CADemia* environment by implementing the interface *Cmd*. This interface defines, amongst others, three essential methods:

- *doCmd()*,
- *undoCmd()* and
- *redoCmd()*.

The interface is implemented by a number of command-classes described below. With the exception of commands *Calculate* and *Transfer*, all commands are used to instantiate components.

### 3.3.1 Commands *DefineOuterBoundary* and *DefineInnerBoundary*

Through an extension of the existing *selectByPickIntersect* command in *CADemia*, these commands permit the definition of an outer boundary of the floor as well as any inner boundaries, e.g. lift shafts, it may have. Graphical components, whether they be line-, curve- or general path objects, are selected and have to be concatenated into a general path object representing the particular boundary. The concatenation functionality was developed separately since it is used not only in these two commands, but many times in the overall process.

The concatenation algorithm is described below.

The problem of concatenating various elements into a continuous general path is divided into a number of different tasks. Firstly, functionality has to be developed for creating a *GeneralPath* object from any number of geometric objects, where these objects are more specifically defined as objects that implement Java's *Shape* interface. Secondly, in order to avoid forcing the user to select these objects in a specific order, i.e. the order in which they would be appended to the general path, a way is required to sort a set of shapes representing a boundary geometrically. Thirdly, considering the fact that the set of shapes used to construct the general path may itself contain general path objects, we need to be able to 'explode' a general path object into its underlying shapes.

Class *GenPathSort* provides the functionalities described above. This class contains a number of methods, of which three were written for application in this specific context, i.e. as a tool for defining boundaries. Other methods of the class will be discussed at a later stage.

***nextShapeFromPoint()***
The method receives a set of shapes and a point object, which will be referred to as the 'frompoint'. It proceeds to search through the set of shapes for a shape that either starts or ends at the given frompoint. If the endpoint coincides with the frompoint, the shape is reversed. The first shape attaching to the frompoint and its endpoint are returned in an object array. While defining boundaries, it is implicitly assumed that no more than two shapes connect at the frompoint, i.e. that the boundary does not diverge, although the method does not check for this.

***explodePath()***
The method receives a general path. A list of shapes is returned, where these shapes form the atomic parts from which the given general path was constructed.

***constructSingleClosedPath()***
The method receives a set of shapes. A *GeneralPath* object is constructed using the given set of shapes. The shapes have to be lines, quadratic curves or cubic curves and together they have to form a closed path. The startpoint and the direction of the path are chosen arbitrarily. If the shapes do not form a single closed path, *null* is returned.

When commands *DefineOuterBoundary* and *DefineInnerBoundary* are called, geometric elements are:

- selected by drawing an intersecting window,
- exploded into their underlying shapes if necessary, and
- added to a set of shapes which is sorted geometrically and used to construct a *GeneralPath* object representing the boundary.

### 3.3.2 Command *ReadColumns*

Column objects, representing the 'sites' of the Voronoi diagram, are read into the database from a text file. If duplicate sites are defined - a mistake easily made when the number of columns becomes large - the Voronoi diagram will not be calculated correctly and thus special provision is made to ensure that this does not happen.

The format of the text file is shown in Appendix C.

### 3.3.3 Command *AddColumn*

In addition to the *ReadColumns* command described above, this command can be used to add *Column* objects to the database through the graphical user interface. When prompted for the coordinates of the column, the user can define the position either by using the mouse or by typing.

### 3.3.4 Command *Calculate*

The Voronoi functionality is executed through the *Calculate* command. Apart from the implementation of Fortune's algorithm, which has already been discussed in section 2.3, this command required the development of various additional tools. Adjustments had to be made in order to effectively deploy the Voronoi functionality for use within the CAD-system and, more specifically, for the proposed application. These adjustments and additional developments will be discussed in the following paragraphs, in the order in which the need for them became apparent.

In section 2.3.7 the customary numerical issues that arise within applications like these were mentioned. A tolerance value has to be chosen through which one can account for inevitable inaccuracies. Since the problem is bounded, the tolerance value can be fitted to the specific problem by making it directly dependent on the size of the defined outer boundary: it is calculated, at the outset, as a function of the largest diagonal that can be drawn for the specific floor geometry.

A few more consequences are brought about by the presence of an outer boundary. Firstly, it dictates the sweep line's start- and end position for the planar sweep and, more importantly, it bounds the Voronoi diagram. The edges produced by the initial implementation extend past the boundaries of the floor and has to be 'trimmed' at the outer boundary.

#### *Trimming*

The *trim()* method is called after the Voronoi diagram has been calculated. It involves, amongst other things, the calculation of all intersections between the Voronoi edges and the defined outer boundary and/or the set of inner boundaries. These intersections, in conjunction with a way in which to identify the part of the edge that has to be thrown away, are used to 'cut' the Voronoi diagram into its bounded shape.

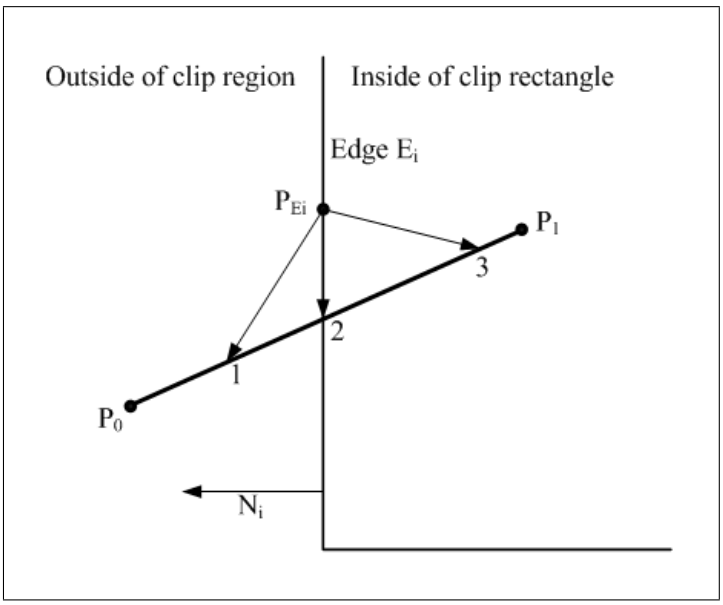



Figure 3.2: Parametric line clipping

The first difficulty that presents itself in this context is calculating the intersections in non-linear regions of the boundary. The *GeneralPath* objects representing the boundaries can contain quadratic- or cubic segments and there exists no easy way to calculate intersections between such segments and line objects. A decision was made to use line approximations of the actual *GeneralPath* objects when calculating intersections. Apart from this particular case, other reasons for using linear-approximated boundaries exist. Consequently the linear approximation is used throughout the computation.

What is essentially required for this task is a line-clipping algorithm. A variety of these algorithms can be found in literature and the Cyrus-Beck technique[3] was chosen. For reasons that will be explained shortly this algorithm could not be used to identify the line segment that should be thrown away, but is used only to calculate the intersection. Referring to Figure 3.2, the line that has to be clipped to the boundary is represented parametrically as follows:

$$P(t) = P_0 + t(P_1 - P_0)$$

where $t = 0$ at $P_0$ and $t = 1$ at $P_1$.

An arbitrary point $P_{E_i}$ on Edge $E_i$ is identified. Three vectors can be drawn from this point to three designated points on the line from $P_0$ to $P_1$: the intersection point, a point on the part of the line inside the clip region and a point on the part of the line outside the clip region. The value of the dot product of these vectors with the edge's outward normal, $N_i$, can be used to distinguish in which region a point lies, and to calculate the intersection.

$$\begin{aligned} N_i \bullet [P(t) - P_{E_i}] > 0 \quad &\text{for the vector} \quad P_{E_i} \rightarrow 1; \\ N_i \bullet [P(t) - P_{E_i}] = 0 \quad &\text{for the vector} \quad P_{E_i} \rightarrow 2 \text{ and} \\ N_i \bullet [P(t) - P_{E_i}] < 0 \quad &\text{for the vector} \quad P_{E_i} \rightarrow 3. \end{aligned}$$

To calculate the intersection, we need to use the second equation above and solve for the value of $t$ at point 2. The equation below is derived in Appendix D.

$$t = \frac{N_i \bullet [P_0 - P_{E_i}]}{-N_i \bullet D} \text{ where } D = P_1 - P_0.$$

Although the other two equations can theoretically be used to identify the part of the line that should be discarded, they are of no use in this context. The reason for this stems from the fact that, in this application, the same line can possibly have intersections with different boundaries and/or more than one intersection with a single boundary. The same Voronoi edge can, for example, run across an inner as well as the outer boundary or cross any boundary more than once as shown in Figure 3.4. As a result an alternative way is required to decide which parts of a Voronoi edge should be discarded.

Referring to the two endpoints of an edge, four possible scenarios exist:

- the startpoint, but not the endpoint, falls inside the floor geometry,
- the endpoint, but not the startpoint, falls inside the floor geometry,
- the startpoint and the endpoint fall inside the floor geometry, or
- the startpoint and the endpoint fall outside the floor geometry.

By identifying the applicable scenario, an edge can be dissected into however many pieces are necessary and the redundant parts can be identified and discarded. The algorithm is described below, following an explanation of how the Cyrus-Beck technique for calculating intersections was implemented.

The Cyrus-Beck algorithm requires the creation of a vector that serves as an outward normal to the boundary. Since it is only used to calculate the intersection, the 'outward' specification falls away and any vector that is normal to the boundary segment under consideration has to be created. This is accomplished in two steps. Refer to Figure 3.3. First a vector, $\overline{C}$, that is perpendicular to the plane is found by calculating the cross product of the boundary segment, $\overline{A}$, with any other vector, $\overline{B}$, which lies in the same plane:

$$\overline{C} = \overline{A} \times \overline{B}$$

The vector $\overline{B}$ can have two possible orientations. This is not strictly necessary - an orientation can be chosen at the outset - but it improves the accuracy of the components of the normal vector being calculated: by always choosing an orientation for $\overline{B}$ so as to maximize the angle between $\overline{A}$ and $\overline{B}$, the accuracy of the components of vector $\overline{C}$ is ensured which, in turn, ensures the accuracy

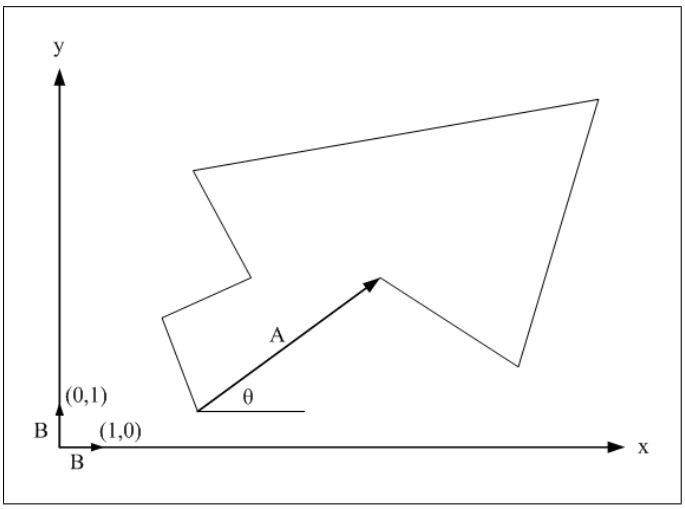



Figure 3.3: Creating the normal vector

of the normal. The choice between the two orientations is made by calculating the dot product $\overline{A} \bullet \overline{B}$. The orientation of $\overline{B}$ for which the angle, $\theta$, is largest, will produce the smallest value for this dot product.

In the second step the normal vector is calculated:

$$\overline{N} = \overline{A} \times \overline{C}$$

This vector, together with the start- and endpoints of the two lines, can be used to calculate the intersection using the equations presented earlier.

Figure 3.4 illustrates the four possible endpoint-position scenarios. The possibility that an edge can have more than one intersection with a boundary has repercussions beyond just the need for this compartmentalizing. If one considers the fact that the direction in which the boundary may be traversed is arbitrary and that the intersections will be found in an arbitrary order, it becomes clear that the intersections will have to be sorted after they have been calculated. For each edge, the intersections are stored in an array which is sorted from left to right (or from bottom to top in the case of a vertical line) once all the intersections have been calculated. The 'direction' that is thus enforced has to be correlated with the direction of the edge in order to use the proposed compartmentalizing to solve the problem. This, in turn, also demands that the edge is 'sorted': all edges have to run from left to right/bottom to top.

For each endpoint-position scenario, the sorted intersection array and the start- and endpoint of the *Edge* or *HalfEdge* is used to, with the help of a counter, create multiple *Edge* records if necessary, i.e. if intersection does indeed occur.

At the time when the trimming is done, i.e. after the Voronoi diagram has been calculated, there exists a set of *HalfEdge* as well as a set of *Edge* objects. During trimming the elements of both these sets are individually handled and

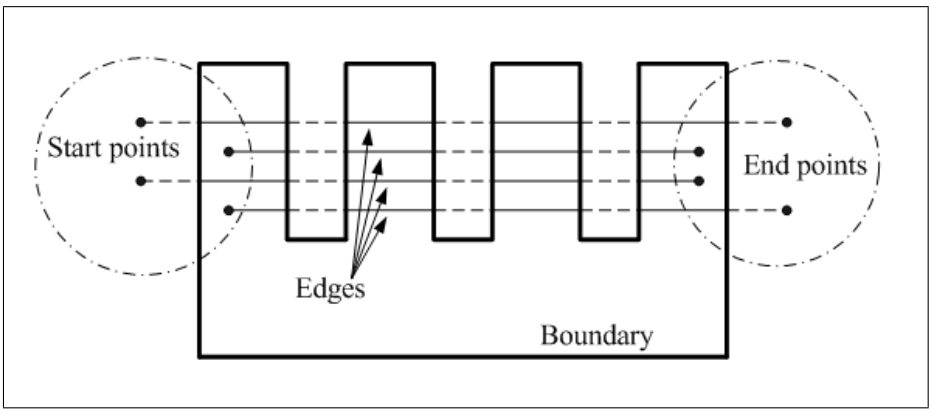



Figure 3.4: Distinguishing between segments

replaced by an appropriate number of *Edge* objects so that, upon completion, there exists only one set, comprising *Edge* objects.

Using the commands and functionalities described above, and given the geometry of a floor in a CAD-environment, the following can now be achieved:

- the definition of the outer boundary and/or inner boundaries of the floor,
- the definition of the positions of the supporting columns,
- the calculation of the Voronoi diagram, and
- the trimming of the Voronoi diagram to the defined boundaries.

What remains is to develop a way in which to place loads on the floor and to transfer the loads to the supporting columns. This is achieved by building Voronoi cells around each column and assigning the loads contained inside the cell to the column. This implies that the area of a cell has to be computed. The representation of Voronoi cells in such a way as to achieve the described objectives is discussed below.

***Building the Voronoi Cells***

If the Voronoi cells were to be represented by *GeneralPath* objects, the problem comprises moving from an outer boundary, a set of inner boundaries and a set of edges to a set of *GeneralPaths*. Whilst discussing the definition of the boundaries in section 3.3.1, class *GenPathSort* was described. It has methods that enable the concatenation of geometric objects into a *GeneralPath* object. If the individual geometric objects that form a cell can be identified, this functionality can be used to create a general path to represent the cell.

The first task is to split the boundaries into segments between points of intersection with the Voronoi edges. Once this has been done, i.e. once a set of boundary segments that each form a 'leg' of a Voronoi cell is available, the cells can be assembled. The intersections calculated whilst trimming are stored

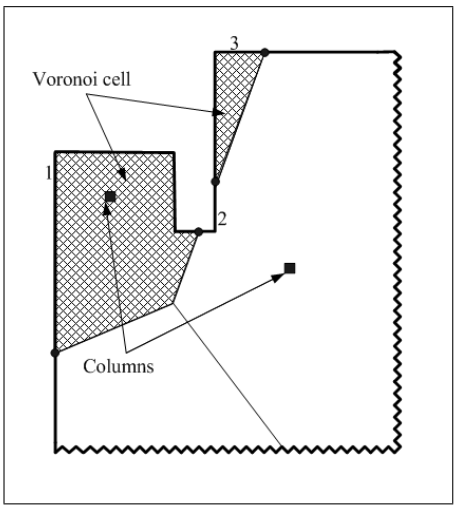



Figure 3.5: Multiple areas of a single cell

and used to perform the segmentation of the boundary. The *splitPath()* method, used to split a boundary into segments given a set of points on the boundary, was developed. This method receives the general path representing the boundary and the stored array of intersection points. It returns a set populated with the calculated segments, each of them represented as a general path. The method is called on each of the boundaries as soon as all the intersections, calculated during the trimming operation, are known.

Given the set of boundary segments and the set of Voronoi edges, what remains is to identify, for each column, the edges and boundary segments comprising its cell, and to then concatenate these elements into a general path. Referring to Chapter 2, each *Edge* object has as attributes the two columns it separates, making it easy to identify the edges belonging to a column. One could then argue that, by searching through the set of boundary segments for a segment or segments that close the discontinuities, if any, between the identified edges, the boundary segments of the cell can be identified. This is not the case.

Figure 3.5 illustrates a scenario in which there is more than one area contributing to a single Voronoi cell. If one were to identify the boundary segments belonging to this cell simply by searching for those that would close the discontinuities between edges, one too many segments - i.e. segment 2 - would be identified. It is clear that an alternative, more robust way in which to identify the different geometric elements comprising a cell is required.

The algorithm that was developed for the creation of the cells as well as the additional developments it necessitated, will be discussed using Figure 3.6. The fundamental principle employed by this algorithm rests on the ability of an edge to distinguish between its 'left' and its 'right' side. Prior to explaining the algorithm, more detail regarding this ability is presented.

**Assigning sides**

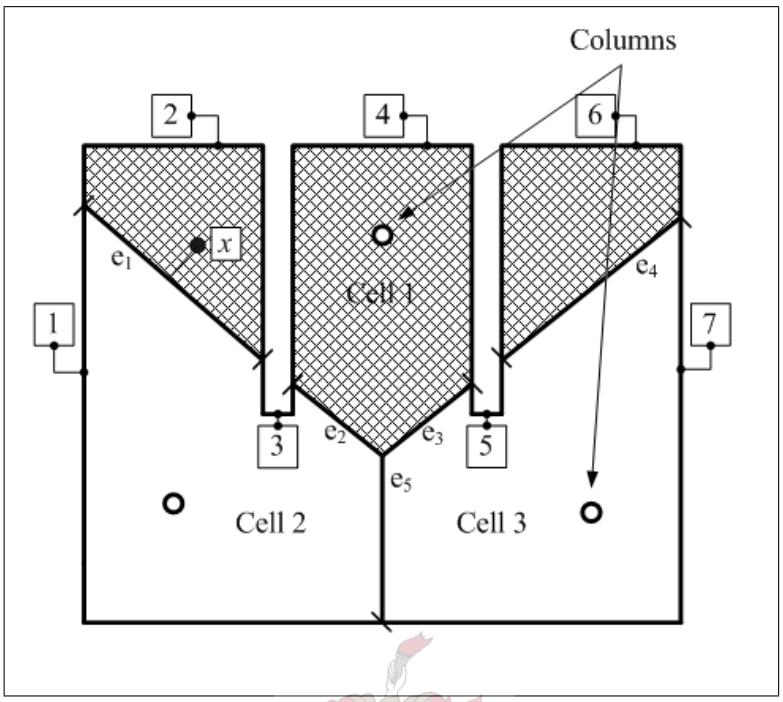



Figure 3.6: Building the Voronoi cells

If one were to consider cell 1 in Figure 3.6, it is easy to see that boundary segment 2 will have to be identified as forming a part of it. Conversely, for cell 2, this segment should not be identified. Apart from reinforcing the earlier statement concerning the identification of segments through closing discontinuities, this fact serves to illustrate that, if edge $e_1$ could provide information on whether a certain column lies on a certain 'side' of it, it could possibly provide the missing tool needed to solve the problem. This was indeed found to be the case and a method, *assignSides()*, was developed. The method is called after the Voronoi diagram has been calculated and before trimming commences.

Method *assignSides()* involves giving a value of either *'L'* or *'R'* to a *'topSide'* attribute that is defined for each edge and half-edge. The value is determined by simply calculating the cross product of vectors $\overline{A}$ and $\overline{B}$, running from the startpoint to the endpoint and the startpoint to the top column, respectively. Refer to Figure 3.7. If the resultant of this cross product points downwards, the edge's top column lies on its right side and, alternatively, if the resultant points upwards, the top column lies to the left of the edge.

During all subsequent calculations involving the edge sets, and especially whilst trimming, it is important to maintain the *topSide* attribute. When the Voronoi diagram is trimmed, and the original edge and half-edge sets are transformed into a new edge set, the attribute is inherited by every newly created edge. Furthermore, this attribute has to be updated whenever the direction of an edge is inverted, as is required during trimming. To maintain the attribute during such modifications, the *flip()* method was written. This method is called

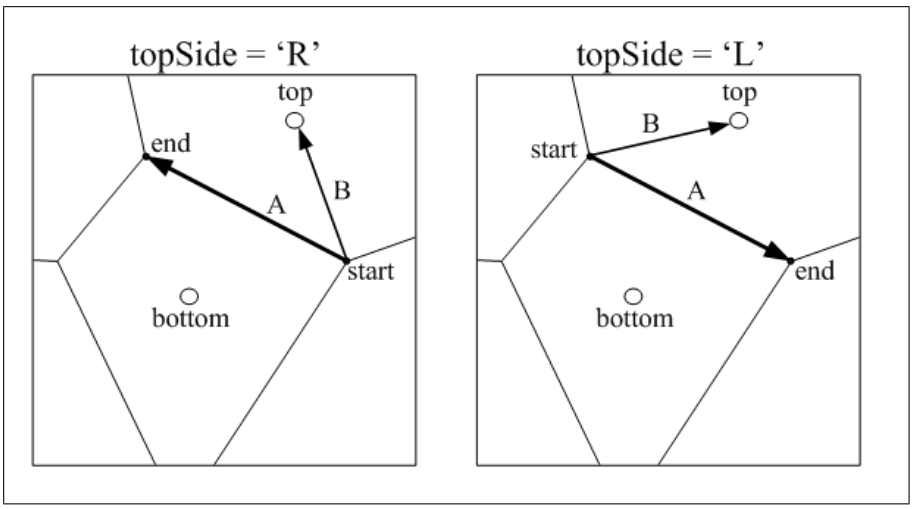



Figure 3.7: Assigning sides

on an edge whenever its start and end point has to be altered and ensures that the *topside* attribute gets updated accordingly.

Apart from defining a *topSide* attribute for all edges and half-edges, additional functionality had to be put in place in order to effectively construct the Voronoi cells. Before presenting the algorithm, a method is discussed which was developed for merging two or more Voronoi edges. The need for this method will, however, only become apparent within the context of the algorithm.

**Merging edges**

Referring to Figure 3.6, cell 1, four edges can be identified - $e_1, e_2, e_3$ and $e_4$ - as belonging to the cell. The proposed algorithm requires that all edges used for the construction of a cell extend from one point on a boundary to another unless, of course, the edges have no intersections with the boundaries. For cell 1, this implies that edge $e2$ and $e3$ should be merged. The same can be said for edges $e2$ and $e5$ of cell 2. The *unifyEdges()* method receives a set of edges, i.e. the edges that were identified as belonging to a certain cell, merges edges where necessary, and returns a set comprised of the modified, merged edges, in the form of general paths. The functionality developed in class *GenPathSort* for concatenating geometric elements is used again, only now the elements do not have to form closed paths. Thus, instead of using *constructSingleClosedPath()*, a method called *constructPath()* was developed. This method returns an object array containing the general path as well as a boolean indicating whether the path is, in fact, closed.

The cell-building algorithm is presented on the next page.

```
for every column{
    find all edges that belong to the column
    concatenate edges where necessary → unified edge set
    find boundary segments that close the discontinuities between the edges →
      boundary set
    while the unified edge set is not empty{
        for every unified edge{
            for every boundary segment{
                if the boundary segment fits between the start and end vertices of
                  the edge{
                    if the area thus formed is on the same side of the edge as the
                      column{
                        add the boundary segment and edge to a result set
                        remove the edge from the edge set
                        remove the boundary segment from the boundary set
                        break
                    }
                    else{
                        remove the boundary segment from the boundary set
                        break
                    }
                }
            }
        }
        if the unified edge set is not empty{
            add the unified edge set to the result set
            add the boundary set to the result set
            clear the unified edge set
        }
    }
    using the result set, construct the Voronoi cell
}
```

Pectora roborant cultus recti

To illustrate the fundamental concept of this algorithm, i.e. the test it uses to distinguish between boundary segments that should be kept and those that should be discarded, refer to Figure 3.6. The essential distinction between segments is made by the *if*-statement in the innermost loop: **if** *the area thus formed is on the same side of the edge as the column.* For cell 1, this involves the following:

- creating an area using boundary segment 2 and edge $e1$,
- establishing, through edge $e_1$'s *topSide* attribute, that the column lies to its left,
- subsequently creating a point, $x$, to the left of the edge and
- testing whether this point is contained in the area.

In this specific case, the point is contained and thus the boundary segment will be kept. Conversely, using the same boundary segment in the context of

cell 2, the point would be created to the right of the edge, would thus not be contained in the area and boundary segment 2 would, rightly, be discarded.

The algorithm as described provides a robust way in which to identify the edges and boundary segments comprising a cell. These edges and segments are concatenated to create a Voronoi cell, i.e. a general path which can consist of more than one area and should be closed. For this purpose the method, *constructClosedPath()* is added to class *GenPathSort*.

***A Connected Edge***

The algorithm behind the construction of the Voronoi cells as well as the additions and most of the modifications it necessitated have been explained. There exists, however, another important modification that has not been described, namely that the *Edge* and *HalfEdge* objects as they exist after the Voronoi diagram has been calculated are not used when the cells are constructed. Instead *ConnectedEdge* objects are used, especially created for use in the process of building the cells. The following paragraphs are dedicated to explaining why this modification was necessary and serve to illustrate how the edges are transformed from their original representations into *ConnectedEdge*'s.

The reader is referred to the description of edge-merging. The aim of the *unifyEdges()* method is to concatenate a number of edges together into a general path where necessary. This general path, however, has to inherit the *topSide* attribute from the edges out of which it is constructed and thus Java's *GeneralPath* object cannot be used. One initially expects that this obstacle can be overcome simply by extending *GeneralPath* and adding the attribute, but this was found to be impossible, and there exist good reasons for this. Firstly, a *topSide()* attribute is worthless without its accompanying top and bottom attributes. Secondly, even if all three the attributes were accounted for in an extended *GeneralPath*, it would not be possible to assign meaningful values to them, since their values differ for each of the edges that we are trying to merge.

To account for these problems, class *ConnectedEdge* was written. The requirements were the following:

- The *topSide* attribute had to be maintained.
- The object could have one or more contributing edges.
- The class has to implement Java's *Shape* interface since the general path builders require shapes. The reason for this stems from the fact that the builders rely strongly on the use of a path iterator, which they receive from each contributing geometric object, to construct a path.

The first two requirements were satisfied by creating a set of contributing edges for each connected edge. This implies that, when edges are merged, each of them is added to this set by way of the connected edge's constructor. Whenever information on whether a specific column lies to the left or to the right of a certain connected edge is required, it can be extracted from any single connected edge in the set of contributing edges. Furthermore, since the possibility

of having more than one contributing edge necessitates representation by means of a general path, a general path builder is built into the class. This builder is called as part of the construction of a new connected edge and uses the set of contributing edges to create a general path.

The implementation of the *Shape* interface was achieved by simply diverting, by means of the general path, all the required methods to class *GeneralPath*.

It is important to note that, during computation, edge directions are frequently inverted and consequently an edge in the set of contributing edges could have one orientation at the time when the connected edge is constructed and another orientation shortly afterwards. Bearing this in mind, the general path builder is not called during construction, but rather each time the path is called upon, for example when the path's iterator is required.

The internal general path builder of connected edges has to be discussed in more detail. Although this builder uses the same algorithm as the one used in class *GenPathSort*, it was necessary to make some modifications since the geometric elements used to construct the path are, in this context, not standard *java.awt.geom* elements, but *Edge* objects. Whilst discussing the *DefineOuterBoundary* command, method *nextShapeFromPoint()*, which is frequently called during the construction of general paths, was mentioned. This method, for example, has to be altered since we require it to return *Edge*'s and not *Shape*'s. Furthermore, modifications to class *GenPathSort* were also required since the construction of the final general path representing the cell will happen here. *NextShapeFromPoint()* sometimes requires the inversion of a shape's direction - when the given 'frompoint' coincides with the shape's end point - as has been discussed earlier. If the shape is a connected edge, this inversion implies the inversion of all its contributing edges. Consequently it is necessary to test for the possibility of the next shape being a connected edge and handle such a scenario separately.

The methods and specializations described above make it possible to calculate the Voronoi diagram, performing the necessary trimming of the edges and building the Voronoi cells. Figure 3.8 shows a floor geometry as it would appear at the completion of the *Calculate* command.

### 3.3.5 Command *AddLoad*

In a manner analogous to that described for defining a boundary, *Load* objects are created and added to the database. The geometric elements defining the loaded area, e.g. the four line objects of the square in Figure 3.9, are selected by drawing intersecting windows. The lines, in conjunction with a $kN/m^2$ pressure value, are assembled into the geometrical object that defines a loaded area.

If the defined loaded area contains or is intersected by any inner boundaries, the *GeneralPath* object representing it should be altered accordingly. This is done by creating an *Area* object using the concatenated geometrical elements which were selected by the user and subsequently subtracting all inner boundaries from it. If the defined area does not contain or is not intersected by any

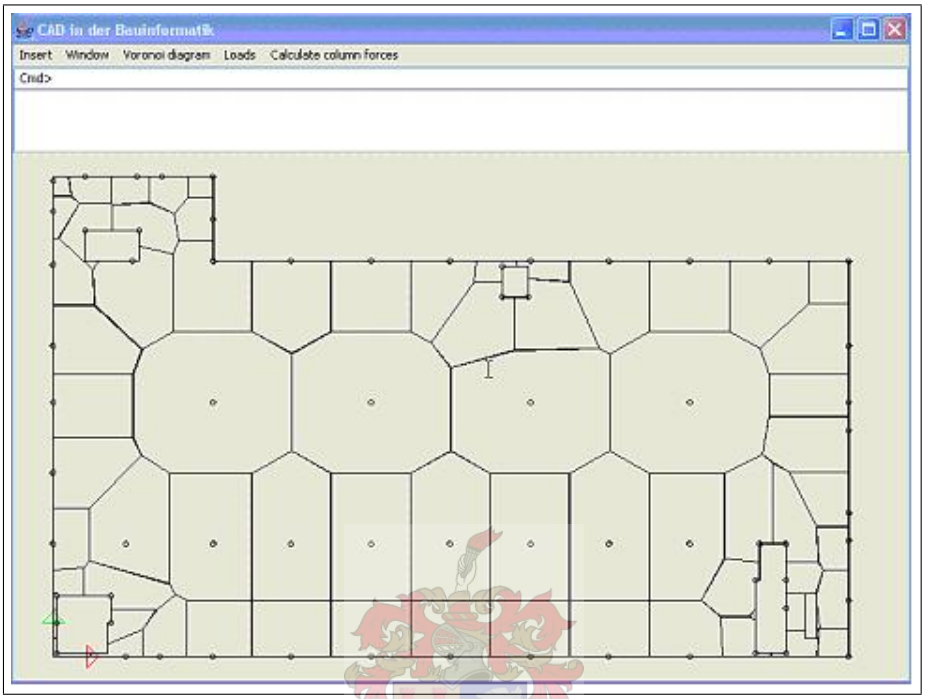



Figure 3.8: A floor geometry divided into influence areas based on the Voronoi diagram.

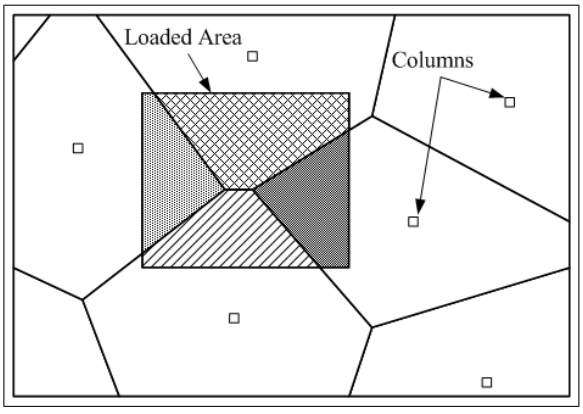



Figure 3.9: Adding a load and transferring the forces.

inner boundaries, the area would remain unchanged, If, however, the opposite is true, the area would be altered accordingly. The new *Area* object is then used to create the final *GeneralPath* representing the load.

### 3.3.6 Command *Transfer*

This command is used to transfer the effect of all loads to the columns. Central to the command is the calculation of cell areas by bounding-line integration, based on the Gauss divergence theorem:

$$\int x^m y^n \, dA = \frac{1}{m+n+2} \int x^m y^n (x dy - y dx)$$

By splitting this integral and setting $m = n = 0$ for the area itself:

$$A_{0,0} = \frac{1}{2} \sum_{e=1}^{n} \int_0^1 (x dy - y dx)$$

In Appendix E the expressions below for the representation of linear, quadratic and cubic segments in the above summation are derived:

*Linear:*

$$A = \frac{1}{2} \sum_{e=1}^{n} (x_{sp} y_{ep} - y_{sp} x_{ep})$$

*Quadratic:*

$$\begin{aligned} A = \tfrac{1}{2} \textstyle\sum_{e=1}^{n} \{ & -\tfrac{1}{3} C_{2,0} C_{2,1} x_{p1} y_{cp} + \tfrac{1}{3} C_{2,0} C_{2,1} x_{cp} y_{p1} - \tfrac{1}{3} C_{2,0} C_{2,2} x_{p2} y_{cp} \\ & + \tfrac{1}{3} C_{2,0} C_{2,2} x_{cp} y_{p2} + \tfrac{1}{3} C_{2,1} C_{2,2} x_{p1} y_{p2} - \tfrac{1}{3} C_{2,1} C_{2,2} x_{p2} y_{p1} \} \end{aligned}$$

*Cubic:*

$$\begin{aligned} A = \tfrac{1}{2} \textstyle\sum_{e=1}^{n} \{ & \tfrac{1}{5} C_{3,0} C_{3,1} x_{cp} y_{p1} + \tfrac{1}{10} C_{3,0} C_{3,2} x_{cp} y_{p2} + \tfrac{1}{10} C_{3,0} C_{3,3} x_{cp} y_{p3} \\ & - \tfrac{1}{5} C_{3,0} C_{3,1} x_{p1} y_{cp} - \tfrac{1}{30} C_{3,1} C_{3,2} x_{p2} y_{p1} + \tfrac{1}{5} C_{3,2} C_{3,3} x_{p2} y_{p3} \\ & - \tfrac{1}{10} C_{3,0} C_{3,3} x_{p3} y_{cp} - \tfrac{1}{10} C_{3,1} C_{3,3} x_{p3} y_{p1} - \tfrac{1}{5} C_{3,2} C_{3,3} x_{p3} y_{p2} \\ & + \tfrac{1}{30} C_{3,1} C_{3,2} x_{p1} y_{p2} - \tfrac{1}{10} C_{3,1} C_{3,3} x_{p1} y_{p3} - \tfrac{1}{10} C_{3,0} C_{3,2} x_{p2} y_{cp} \} \end{aligned}$$

Although these expressions permit the exact integration of the area, the last two are not used since straight-line approximations are used in non-linear regions of the boundary. The functionality has, however, been incorporated into the area calculations and can be used as soon as the need for approximating the boundary falls away, i.e. when intersections on non-linear segments of general paths can be calculated, and when a non-linear general path can be divided into

two or more general paths given a set of intersection points.

Once the cell areas have been calculated, they are intersected with each *Load* object to obtain the load increment produced by the load. The reader is again referred to Figure 3.9. The summation of these increments yield the load transferred to each column.

The CAD functionality of *CADemia*, together with the additional components and commands described in this chapter, yield a practical and effective set of tools for the computation of preliminary column forces based on the concept of influence areas for load assignment. The method used in engineering practice is thereby transferred to a computational platform which is far more effective, accurate and reliable than the tools currently in use. In this respect it can lead to a significant improvement in productivity. However, the accuracy of the method itself has to be evaluated. This is done in the following chapter.

# Chapter 4

# Finite element modeling and comparative results

The accuracy of the forces obtained using the prototype implementation is determined through comparison with the results of finite element analyses. For this exercise a sample of typical floor geometries are modeled using a commercially available FEM software package, DIANA. The floor geometries were chosen to each represent a different design in order to, considering the limitations imposed by the time available for this task, create a representative sample. Three floor geometries were modeled:

- the third floor of an apartment building,
- the mezzanine floor of an industrial building and
- a parking lot floor.

The modeling of these floors as well as, for each floor, two sets of comparative results, obtained by placing an appropriate live load on the floor, are presented in this chapter. For all individual value pairs percentage errors are calculated which are subsequently used to calculate an average error. These average errors are calculated for various subsets that are identified for the floor geometry, e.g. a set of internal columns and a set of edge- or boundary columns.

The results are analyzed and elaborated on in Chapter 5.

## 4.1 An apartment building floor

The first floor modeled can be seen as representing floors with uniform stiffness properties and irregular support groupings. Geometrically speaking, it has various lift shafts and stair cases inside an arbitrarily shaped outer boundary. A layout of the floor is presented in Appendix F.

### 4.1.1 Model description

The reinforced concrete slab is modeled using 3-node triangular plate bending elements, based on the Discrete Kirchhoff theory, adapted to take shear de-

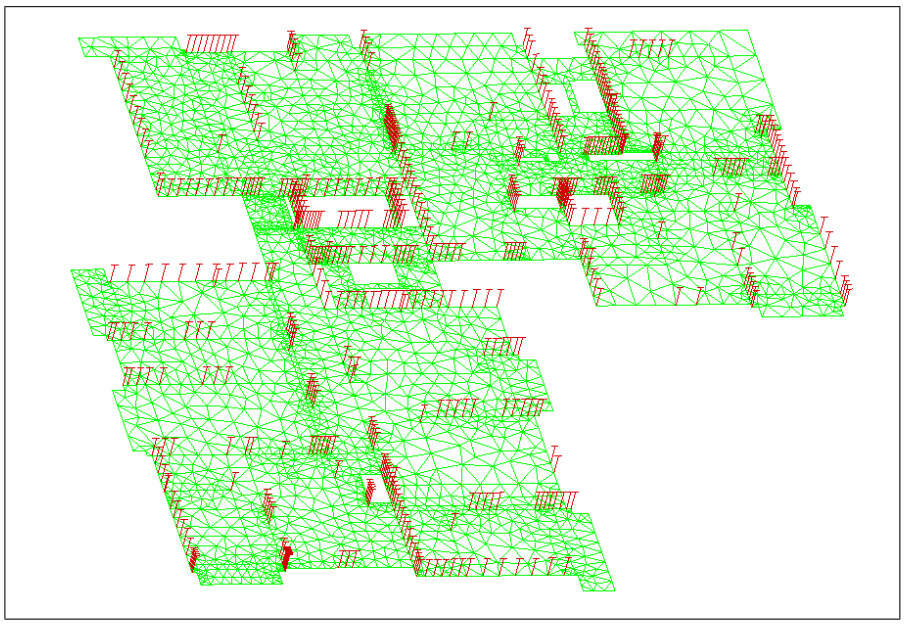


Figure 4.1: Mesh and boundary conditions for the apartment building floor.

formation into account. Where walls or columns are present, the nodes are restricted against translation in the $z$-direction. Figure 4.1 shows the mesh and boundary constraints.

Two physical properties, defined by the thickness of the plate elements, are assigned to two surfaces representing the 225mm slab of the main floor body and the 150mm slabs of the balconies, respectively. For each of these surfaces, an appropriate mass density for the material is assigned.

The material properties below are assigned to all surfaces of the model:

*Young's modulus* = 25GPa
*Poisson's ratio* = 0.2

A distributed load of $1.5kPa$[7] is placed on the floor.

### 4.1.2 The Voronoi diagram

Figure 4.2 shows the calculated Voronoi diagram. An outer boundary and a total of 14 inner boundaries are created to represent the geometry. Walls are represented by a series of linearly placed columns.

### 4.1.3 Comparative results

Displacement contour plots for the finite element model subjected to the $1.5kPa$ live load are presented in Figure 4.3. The displacements are in meters. Figure 4.4

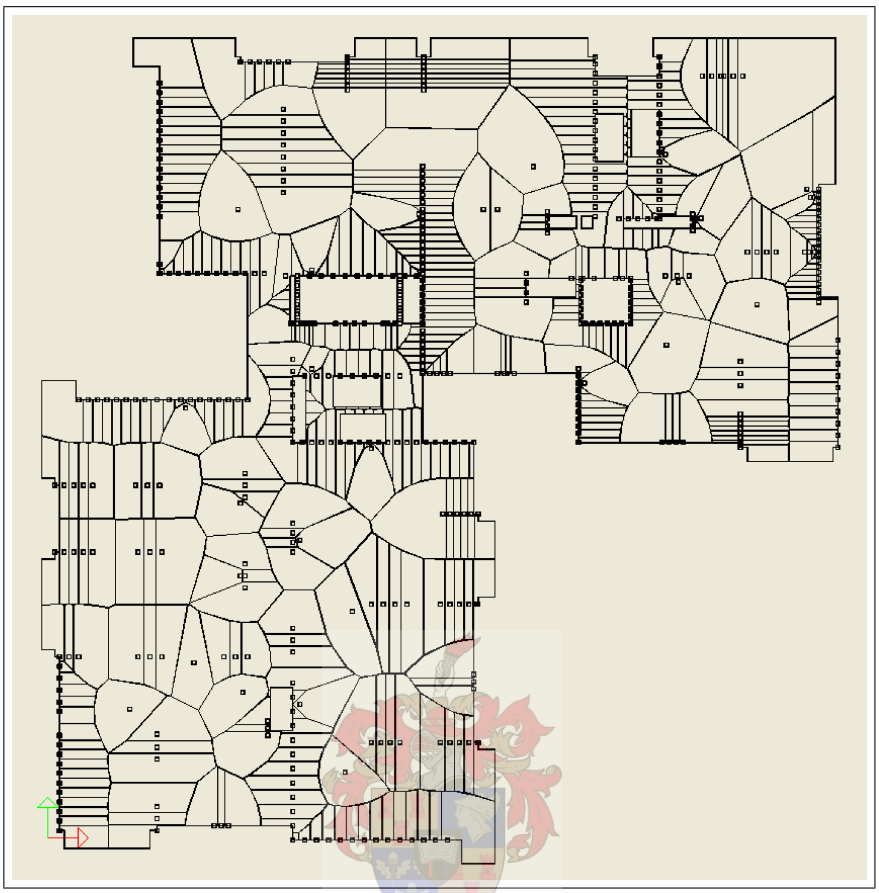


Figure 4.2: Voronoi diagram of the apartment building floor

shows the deformed shape using a deformation scale factor of 1200.

Where walls are present the nodal reaction forces obtained using DIANA, as well as the forces obtained for the linearly placed columns representing the walls in the prototype software, are transformed into $kN/m$ line loads. The two sets of column forces and line loads are tabulated in Appendix G, where a distinction between boundary walls, inner walls and columns are made and a percentage error is calculated for each of the pairs of values. Errors obtained for areas of the floor subjected to hogging bending moments are specifically marked. Greatly exaggerating deformation, such an area is illustrated by way of Figure 4.5.

Due to the DIANA reaction forces having relatively small values compared to the Voronoi forces in such areas, the errors are large and of little scientific value. Omitting them, an average error value for the column, wall, internal- and boundary wall subsets, are calculated.

The average error for the walls was found to be significantly higher than that of the columns and, additionally, the boundary walls' contribution to this

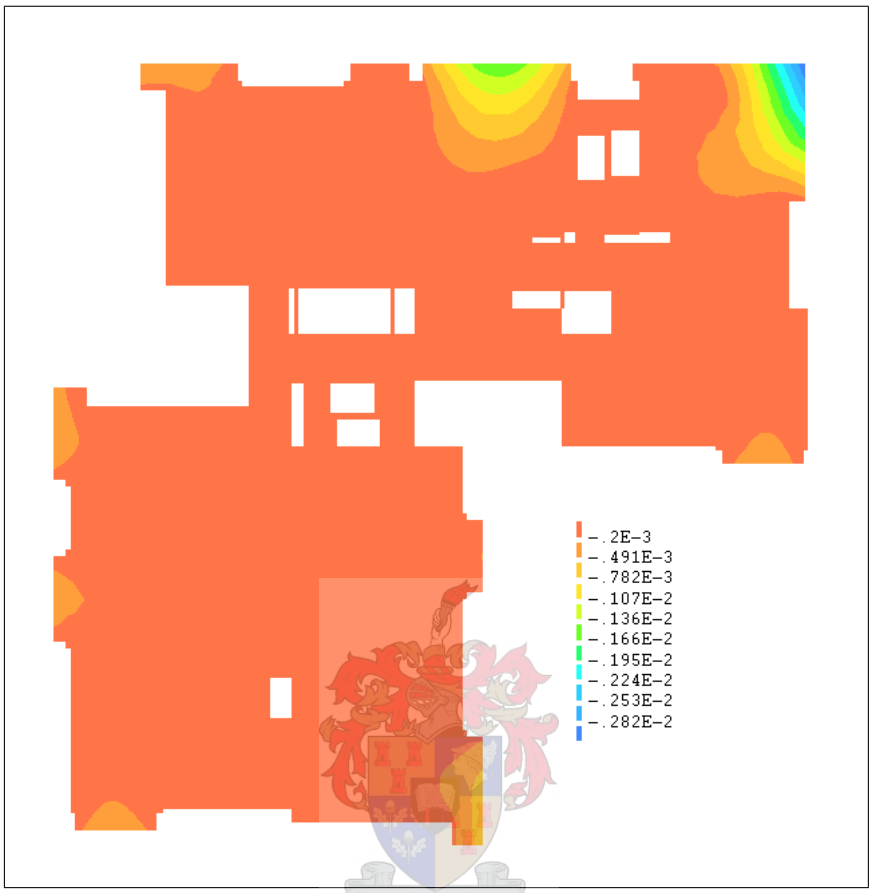



Figure 4.3: Displacement contour plots for the apartment building floor.

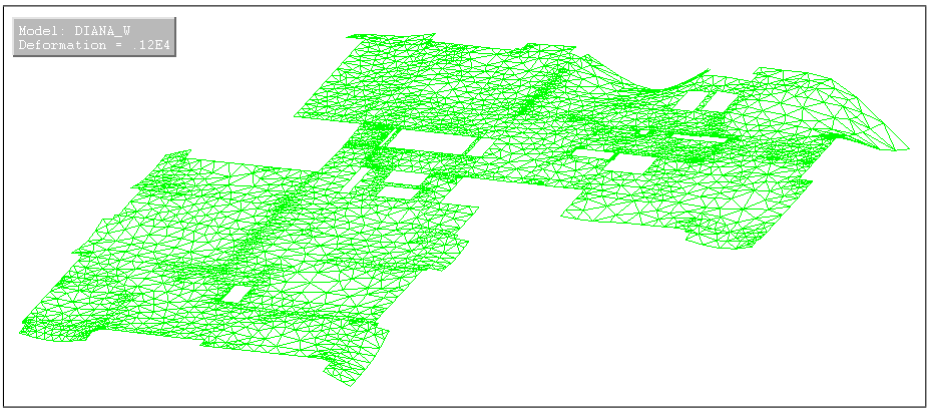



Figure 4.4: Deformed shape, applying a deformation scale factor of 1200.

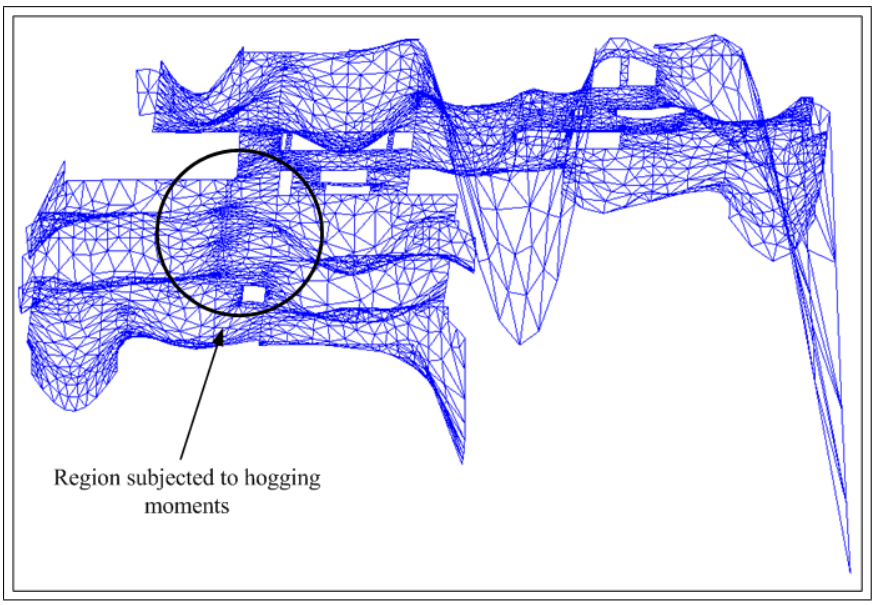



Figure 4.5: Hogging moments.

| Subset | Number of error values in set | Average error |
|---|---|---|
| Columns | 25 | 55% |
| Walls | 48 | 26% |
| Internal walls | 29 | 18% |
| Boundary walls | 19 | 38% |

Table 4.1: Average errors for the apartment building floor.

error larger than the internal walls' contribution. The results are tabulated in Table 4.1 and will be discussed in the following chapter.

## 4.2 Mezzanine floor of an industrial building

The second floor is an industrial coffer slab structure with varying stiffness and self-weight. Deep transfer beams, in both directions, separate the coffers. In the north-western and south-eastern corners are 250mm slabs. The columns are spaced fairly regularly, a number of inner boundaries are present and the outer boundary has an almost rectangular shape. Refer to Appendix H for a layout of the floor.

### 4.2.1 Model description

As in the previous model, 3-node triangular Discrete Kirchoff plate bending elements are used to model the floor. Where columns are present, the nodes are restricted against translation in the z-direction. Figure 4.6 shows the boundary constraints and surface outlines.

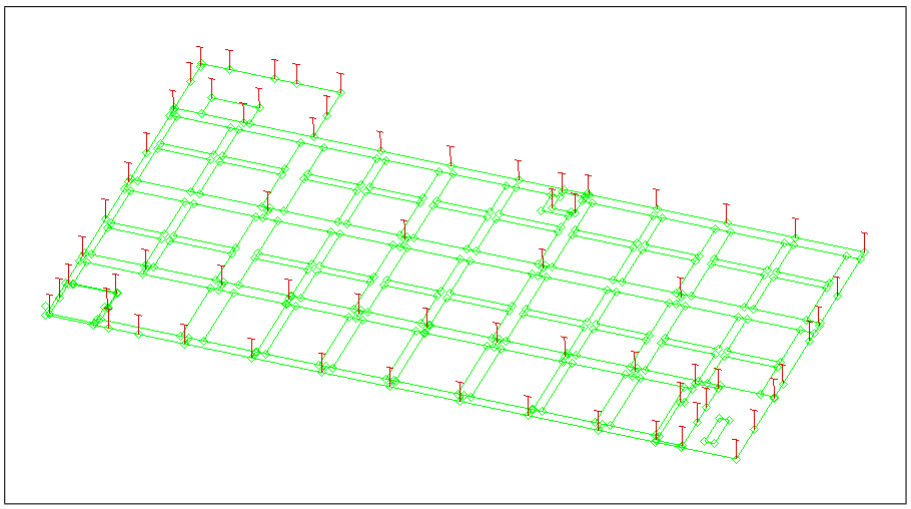


Figure 4.6: Surface outlines and boundary conditions for the industrial floor.

The flexural rigidity, $D$, of a plate element is analogous to the flexural stiffness $EI$ of a beam[6]:

$$D = \frac{Et^3}{12(1-\upsilon^2)}$$

where $t$ = the plate thickness and
$\upsilon$ = Poisson's ratio for the material.

Thus to counteract the added stiffness caused by the $(1-\upsilon^2)$ term, an equivalent plate thickness, $t_e$, is calculated as follows:

$$EI = \frac{Et_e^3}{12(1-\upsilon^2)}$$

$$\rightarrow t_e = \sqrt[3]{12I(1-\upsilon^2)}$$

This modification requires that an adjustment is made to the material's mass density:

$$volume \times \rho = volume_e \times \rho_e$$

$$\rightarrow \rho_e = \frac{volume}{volume_e} \times \rho.$$

The material and physical properties of the surfaces are adjusted to counteract the added stiffness caused by the use of the Kirchoff element. The moment of inertia used for this adjustment for the coffer surfaces is calculated using the dimensions presented in Figure 4.7. Figure 4.7 also supplies the information needed to calculate an equivalent mass density for the coffer material.

Equivalent plate thicknesses and mass densities for not only the coffers, but also the 250mm slabs and 550mm as well as 1000mm deep beams are calculated

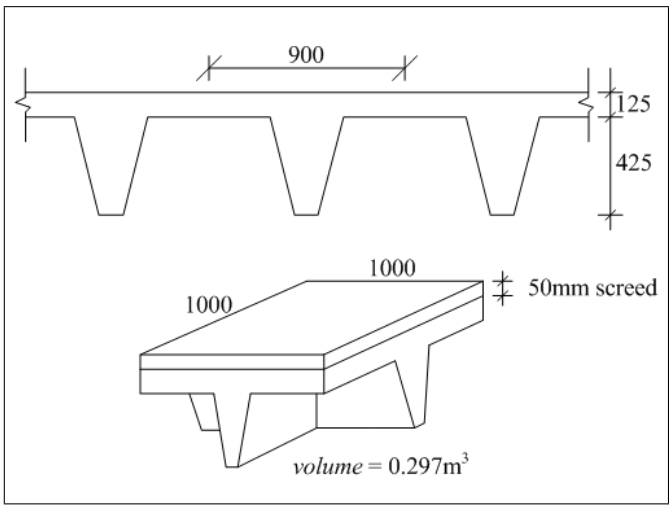



Figure 4.7: Coffer dimensions and volume

and assigned. Young's modulus and Poisson's ratio are, for each of the three defined 'equivalent' materials, assigned values of $25GPa$ and 0.2, respectively.

A distributed load of $16.12kPa$, comprised of service and live loads multiplied by their appropriate safety factors[7] is placed on the floor.

### 4.2.2 Voronoi diagram

The Voronoi diagram of this floor has already been presented as Figure 3.8 in Chapter 3. 67 Voronoi cells, one for each of the columns, are created and used to distribute the abovementioned load amongst the columns.

### 4.2.3 Comparative results

The results of the finite element analysis are presented by way of two figures. The first figure, Figure 4.8, shows displacement contour levels. Displacements are in meters. The second figure, Figure 4.9, shows the deformed shape. A deformation scale factor of 500 is applied.

The pairs of column force values are tabulated in Appendix I where a distinction is made between columns that fall on outer- or inner boundaries and internal columns. The percentage errors obtained for each of these pairs are presented in the appendix and their average values summarized in Table 4.2. Where the DIANA reaction forces point in an upwards direction the calculated errors are irrelevant and are, following an argument analogous to that presented in section 4.1.3., not used when calculating averages. At a quick glance the tendency for columns that fall on edges to produce greater errors than the internal columns is once again observed.

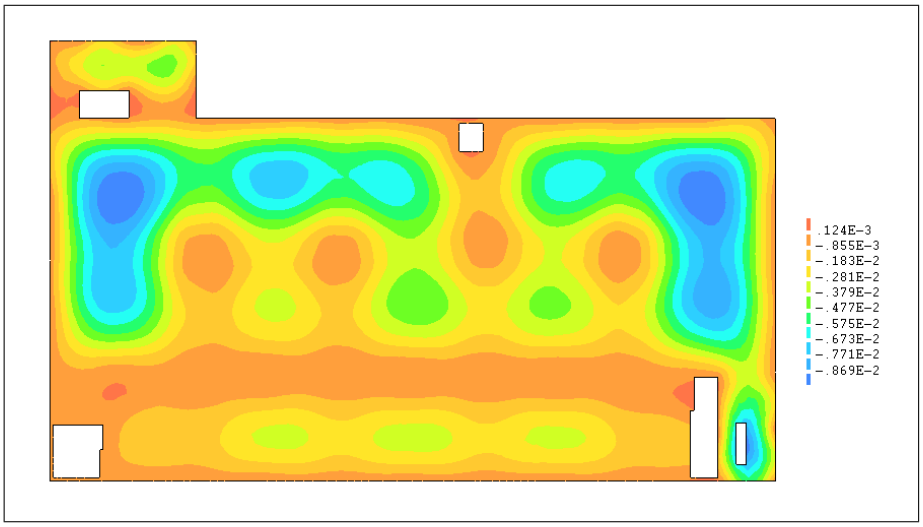



Figure 4.8: Displacement contour levels for the industrial floor.

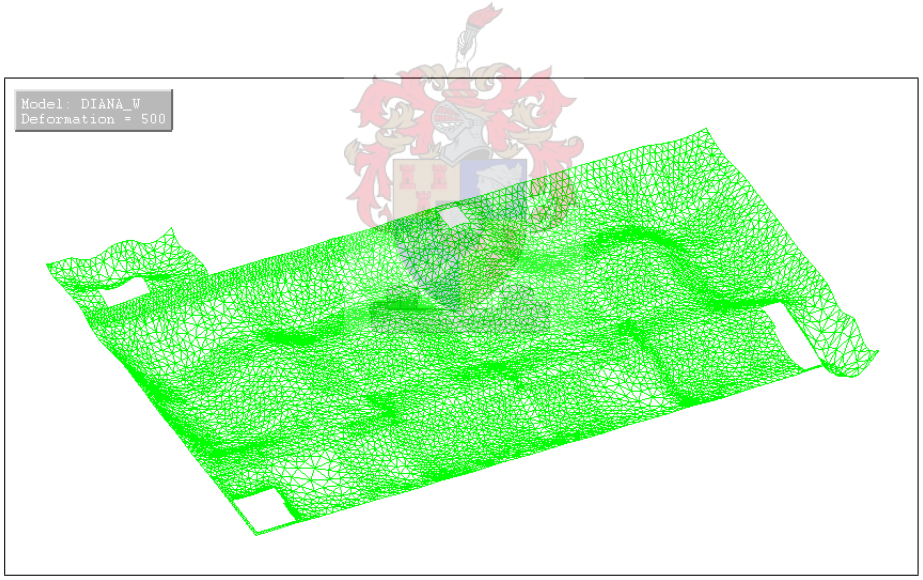



Figure 4.9: Deformed shape, applying a deformation scale factor of 500.

| Set/Subset | Number of error values in set | Average error |
|---|---|---|
| Columns | 63 | 65% |
| Internal columns | 12 | 21% |
| Edge columns | 51 | 76% |

Table 4.2: Average errors for the industrial floor.

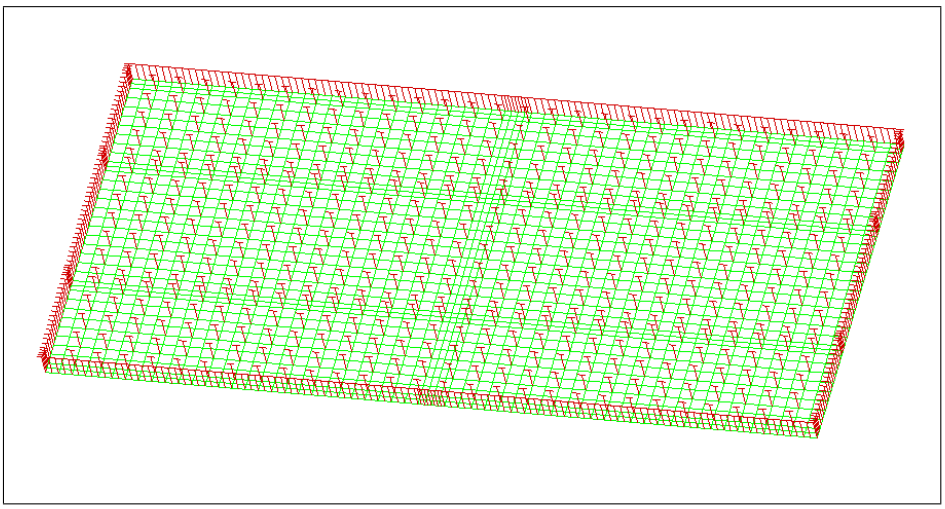


Figure 4.10: Mesh and boundary conditions for the parking lot.

## 4.3 A parking lot floor

The last floor can be seen as representing floors with uniform stiffness properties as well as regular support groupings: a 225mm post-tensioned concrete slab is supported by columns placed on a fairly regular $30 \times 19$ grid. The floor has a rectangular shape and no inner boundaries are present.

### 4.3.1 Model description

Due to simple geometry the meshing of this floor did not require the use of triangular elements, as was the case with the previous two floor geometries. Eight-node quadrilateral plate bending elements, based on the Mindlin-Reissner theory, could be used. The appropriate plate thickness and mass density as well as values for Young's modulus and Poisson's ratio are assigned as in the previous two analyses. Constraints against translation in the z-direction are applied to nodes where walls or columns are present. Figure 4.10 shows the mesh and boundary constraints.

### 4.3.2 Voronoi diagram

The Voronoi diagram of this floor is comprised of nothing more than a number of rectangular cells. As for the apartment building floor, the walls are represented by a number of linearly placed columns. Refer to Figure 4.11.

### 4.3.3 Comparative results

Displacement contour levels and deformation plots are presented in Figure 4.12 and Figure 4.13, respectively. Displacements are in meters and a scale factor of 500 is applied in Figure 4.13. Appendix J contains the pairs of column force values and the calculated errors. Average errors for the set of internal columns and the set of walls are calculated and presented in Table 4.3.

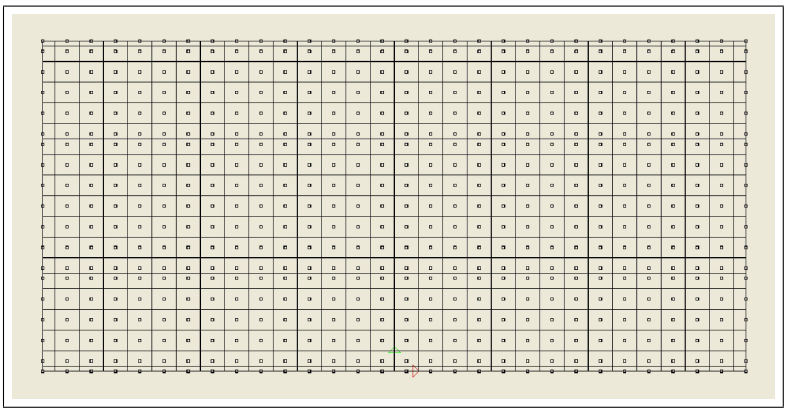


Figure 4.11: Voronoi diagram of the parking lot floor.

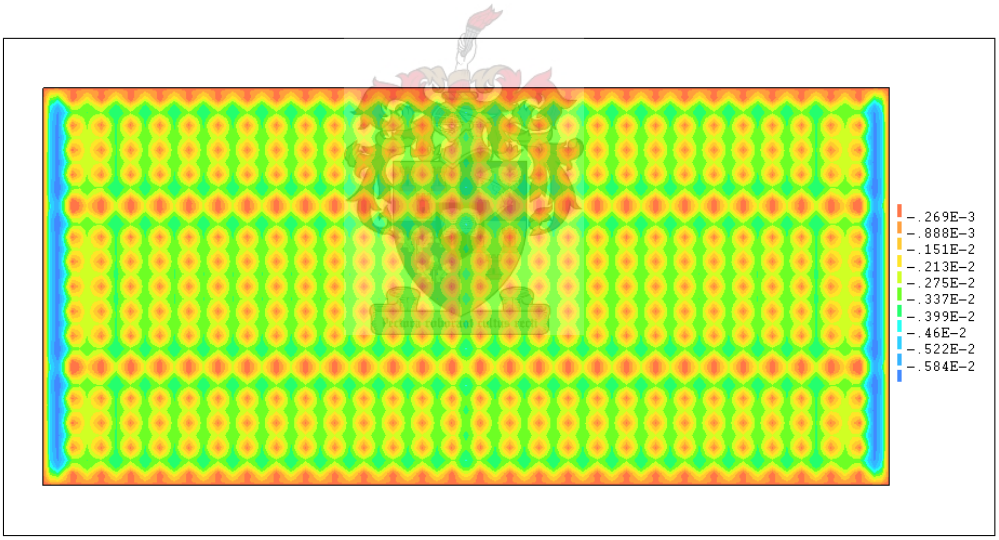



Figure 4.12: Displacement contour levels for the parking lot.

| **Subset** | **Number of error values in set** | **Average error** |
|---|---|---|
| Internal columns | 476 | 4.5% |
| Walls | 4 | 56% |

Table 4.3: Average errors for the parking lot.

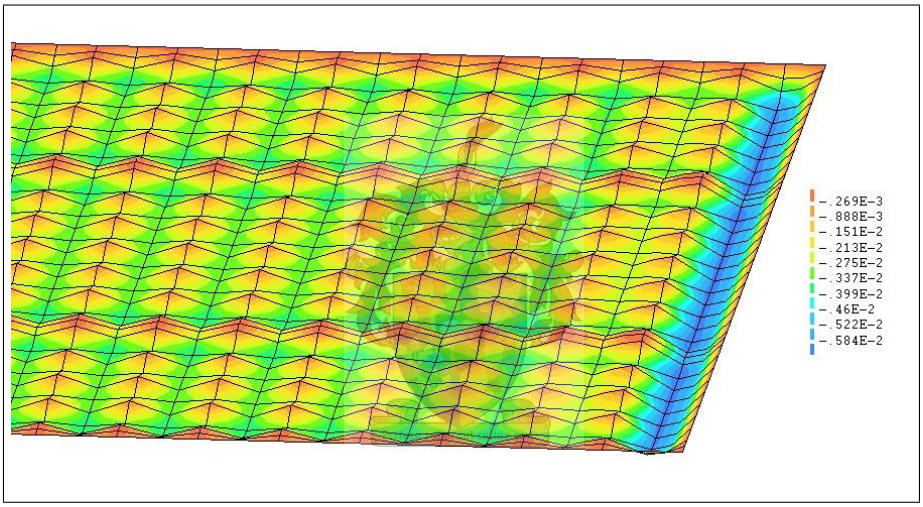



Figure 4.13: Deformed shape, applying a deformation scale factor of 500.

# Chapter 5

# Conclusions and recommendations

The column forces obtained using the Voronoi diagram were compared to the results of finite element analyses in the previous chapter. Before continuing to discuss the tendencies observed through this comparison, it is important to remind the reader of the third set of values available for comparison, namely the forces as they would be calculated in current engineering practice, as was explained in section 1.1. The developed software, serving as a semi-automated alternative to this procedure, is based on exactly the same principles as those used in practice, but the forces are calculated faster and more accurately in an integrated CAD environment. Considering this, this study could also be seen as providing insight into the accuracy with which preliminary forces are calculated in industry. Consequently, this chapter will be used not only to present a number of tendencies observed regarding the errors obtained, but also to make judgements concerning the method used in practice and, lastly, to give guidelines as to how this study should be continued.

## 5.1 Observed tendencies

Referring mainly to the tables presented in the previous chapter, in which average errors were calculated for a number of subsets for each floor, three tendencies were identified. Each of them will be discussed separately.

### 5.1.1 Columns situated on boundaries

All of the floors, in varying degrees, showed a tendency to produce greater errors for columns or walls situated on the boundaries. In Table 5.1 the errors are presented so as to highlight this tendency.

A reason for such a tendency can be found by examining Figure 4.13 presented in the previous chapter. The deformations in the first span of the parking lot slab are considerably larger than those in the subsequent spans, a fact that can be attributed to the lack of continuity existing at the boundary, which

| | *Average errors* | |
|---|---|---|
| | Boundary columns | Internal columns |
| Apartment building | 38 | 18 |
| Industrial building | 76 | 21 |
| Parking lot | 56 | 4.5 |

Table 5.1: Boundary vs internal forces.

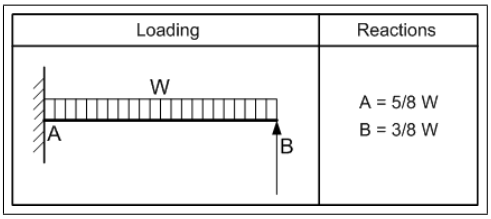



Figure 5.1: Extract from table of beam diagrams.

causes a reduction in the slab's resistance to rotation. One expects the prototype software to produce forces that are smaller than the actual forces in the boundary regions since, for a regular grid, the forces allocated to the boundary columns are exactly half of what is allocated to the internal columns because the influence areas are halved at the boundary. The opposite was found to be true, i.e. it was found that in most cases the Voronoi technique produces forces that are larger than the actual forces at the boundaries. However, by comparing the reactions produced at the first row of columns and the reactions produced at the boundary to the reactions given in Figure 5.1 for a propped cantilever, the results are better understood.

### 5.1.2 Hogging bending moments

This tendency has been discussed to some extent in the previous chapter (see section 4.1.3). The reaction forces in certain areas of the floor subjected to hogging moments can assume not only very small values, but can in some cases even point in a downwards direction. The errors obtained for such reaction force pairs were omitted when calculating averages because their extremely large values would only serve to distort the average whilst they actually have little scientific value in such a context. They do, however, still exist and it can be confidently stated that the prototype software, through its inability to take stiffness properties into account, will generally produce large errors in regions subjected to hogging bending moments. It is also worth noting that these errors will always be on the safe side.

### 5.1.3 Uniform stiffness properties

Considering the parking lot slab as representative of floors having uniform stiffness properties and the industrial floor with its deep transfer beams as representative of floors with non-uniform stiffness properties, Table 5.2 is created.

| | *Average errors* | |
|---|---|---|
| | Uniform stiffness | Non-uniform stiffness |
| Internal columns | 4.5 | 21 |
| Boundary columns or walls | 56 | 76 |

Table 5.2: Uniform vs non-uniform stiffness properties.

| | *Percentage* |
|---|---|
| Calculating the Voronoi diagram | 9.1% |
| Trimming | 1.8% |
| Building the Voronoi cells | 87.5% |

Table 5.3: Percentage of computation time spent on a task.

The table serves as confirmation of a rather intuitive conclusion that can be drawn regarding the role played by the extent in which a floor's stiffness properties varies. When stiffness properties are relatively uniform, the errors obtained using the prototype software can be expected to be smaller than those obtained when stiffness varies. The presence of transfer beams, for instance, has a great influence on the way in which the total load on a floor is divided amongst the load-bearing elements.

## 5.2 Aspects that need to be addressed

There are certain aspects that can be improved upon in the current implementation. One of these aspects has already been discussed in section 3.3.4 under *Trimming* and concerns the line approximations used for curves. It was stated that linear-approximated boundaries are used because intersections between lines and *GeneralPath's* cannot be calculated. Although not to a great extent, the approximation influences the accuracy with which areas are calculated (consider section 3.3.6 where the calculation of areas based on the Gauss divergence theorem is discussed). Consequently, accuracy could be improved by developing a way in which the abovementioned intersections can be calculated, thus allowing for the exact integration of the area.

The next aspect concerns the algorithm implemented for building the Voronoi cells, thoroughly discussed in section 3.3.4 under *Building the Voronoi cells*. The overall time currently needed for the calculation of the forces is not satisfactory. Computation time for the apartment building floor, for instance, having more than 500 columns for which cells have to be calculated, amounts to 141s. Table 5.3 illustrates, percentagewise, the time spent on the three main tasks comprising the calculation.

It is clear that the optimization of the software would primarily involve optimizing the construction of the Voronoi cells. The reason for the extensive time required to build the cells stems from the fact that the algorithm requires, for each column, an iteration over the complete set of boundary segments where,

furthermore, a path iterator is called and traversed for each boundary segment.

Due to time-constraints the task of developing and implementing a new and improved algorithm could not be performed. (The reader is referred back to the difficulties that were encountered in trying to develop a robust algorithm to handle the problem). This aspect should, however, be seen as one of the first that should be addressed as part of a continuation of the study.

## 5.3 Outstanding functionality

Apart from having to improve the algorithm currently implemented for constructing the Voronoi cells, other tasks that would have to be performed if the study were to be continued, can be identified.

### 5.3.1 Building definition

The functionality as it exists at present solves a two-dimensional problem and needs to be extended to allow for the definition of the building in three dimensions. It is only when this extension has been carried out that the vertical accumulation of the forces can be calculated - a task that can be dealt with as a flow problem in the column network.

### 5.3.2 Horizontal loading

Horizontal forces, that would account for earthquake and wind loading, are currently not taken into consideration. Referring to Figure 5.2, a suggestion is made as to how these forces could, as a separate exercise involving a greatly simplified finite element model, be accounted for.

Viewing the floors as lumped masses, the distributed horizontal loading is concentrated into horizontal forces applied only at floor elevations. A bar and framework model of the columns and bracings can be set up without great effort. An assumption is made that the floors themselves act as rigid bodies and they are consequently not explicitly modeled, but rather their effect on the model is accounted for with the use of constraint equations.

This simplified model can be analyzed to determine the column forces due to horizontal loading. These forces can subsequently be superimposed upon the vertical forces to obtain the total column force in each column.

## 5.4 Final comments and recommendations

Having developed and tested software for the calculation of preliminary column forces based on the method used in current engineering practice, and having seen the errors produced by calculating the forces in this manner, a few more important conclusions can be drawn.

Firstly, relying solely on influence areas to calculate column forces is unsafe. Secondly, the alternative previously mentioned, namely the performance of a simple, largely automated finite element analysis, is suggested. Taking this

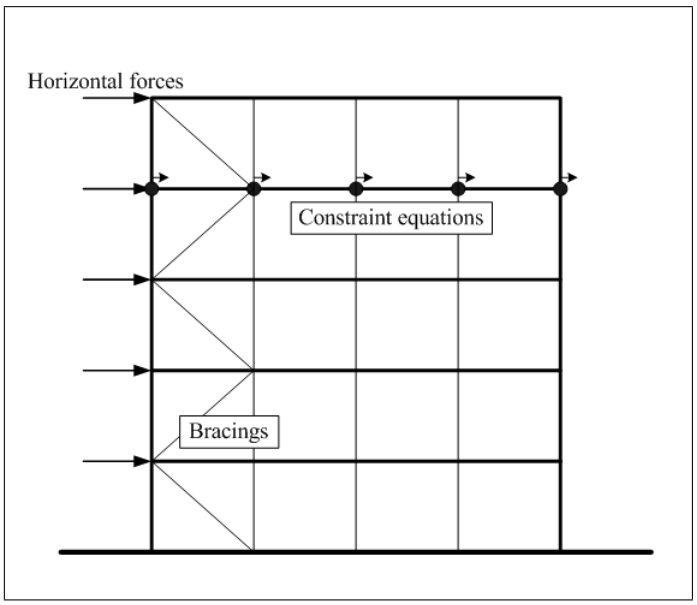



Figure 5.2: Horizontal Loading

course of action would involve using the influence areas as a basis for automatic meshing and subsequently performing a finite element analysis using plate elements.

It is concluded that the calculation of column forces based on the subdivision of floors into influence areas may be dangerous. Even when a floor has uniform stiffness properties, which supports an inclination to believe that solving the problem geometrically is acceptable, the errors obtained are far greater than what can be accepted.

# Appendix A

# Intersecting parabolas

*Derivation of coordinates of parabola intersection.*

$x^2 + (y-a)^2 = (y+a)^2$
$\rightarrow x^2 + y^2 - 2ay + a^2 = y^2 + 2ay + a^2$
$\Rightarrow x^2 = 4ay$

$a = \frac{1}{2}(p_{iy} - l_y)$

$l_y + a + y = Y$
$\rightarrow y = Y - l_y - \frac{1}{2}(p_{iy} - l_y)$
$= Y - \frac{1}{2}(l_y + p_{iy})$

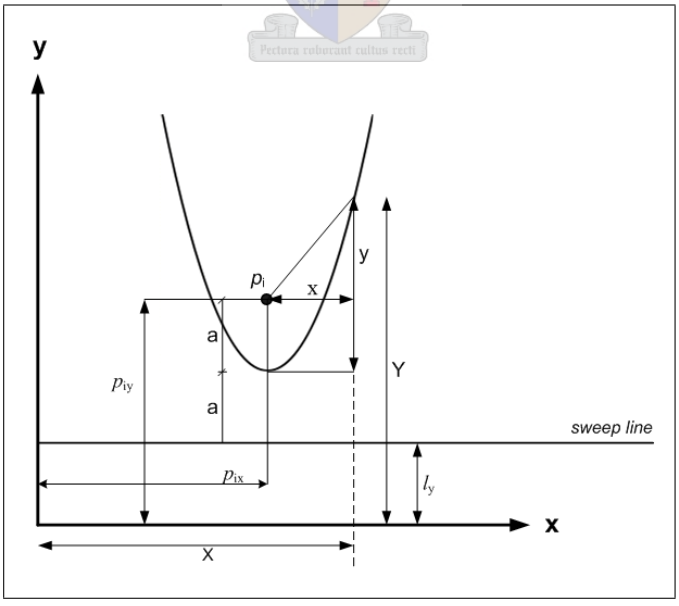



Figure A.1: Intersections between parabolas: defining variables.

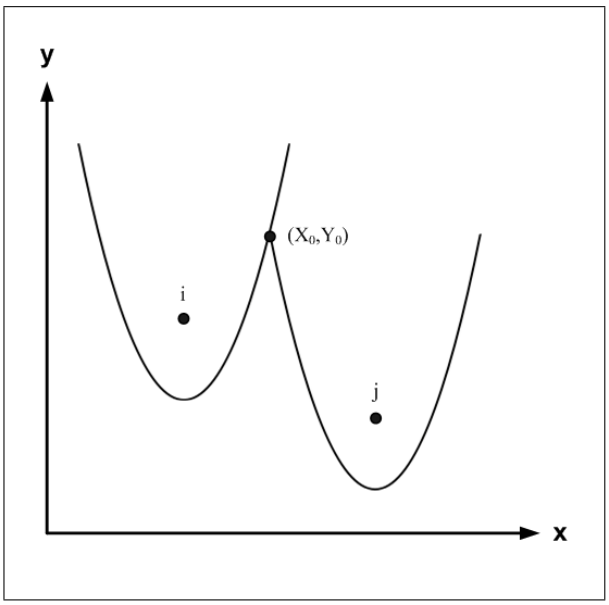



Figure A.2: Definition of variables, continued.

$p_{ix} + x = X$
$\rightarrow x = X - p_{ix}$

$\Longrightarrow (X - p_{ix})^2 = 4ay$
$= 2(p_{iy} - l_y)(Y - \frac{1}{2}(l_y + p_{iy}))$

For two points $i$ and $j$ the intersections point is $(X_0, Y_0)$.

1) $(X_0 - p_{ix})^2 = 2(p_{iy} - l_y)(Y_0 - \frac{1}{2}(l_y + p_i y))$
2) $(p_{jx} - X_0)^2 = 2(p_{jy} - l_y)(Y_0 - \frac{1}{2}(l_y + p_j y))$

From 2):

$Y_0 = \frac{(p_{jx} - X_0)^2}{2(p_{jy} - l_y)} + \frac{1}{2}(l_y + p_{jy})$

Substitute $Y_0$ into 1):

$(X_0 - p_{ix})^2 = 2(p_{iy} - l_y)[\frac{(p_{jx} - X_0)^2}{2(p_{jy} - l_y)} + \frac{1}{2}(l_y + p_{jy}) - \frac{1}{2}(l_y + p_{iy})]$

$X_0^2 - 2p_{ix}X_0 + p_{ix}^2 = \frac{p_{iy} - l_y}{p_{jy} - l_y}(p_{jx} - X_0)^2 + (p_{iy} - l_y)(p_{jy} - p_{iy})$

$= \frac{p_{iy} - l_y}{p_{jy} - l_y}(p_{jx}^2 - 2p_{jx}X_0 + X_0^2) + (p_{iy} - l_y)(p_{jy} - p_{iy})$

$[1 - \frac{p_{iy} - l_y}{p_{jy} - l_y}]X_0^2 + [2p_{jx}\frac{p_{iy} - l_y}{p_{jy} - l_y} - 2p_{ix}]X_0 + [p_{ix}^2 - p_{jx}^2\frac{p_{iy} - l_y}{p_{jy} - l_y} - (p_{iy} - l_y)(p_{jy} - p_{iy})] = 0$

Thus

$Y_0 = \frac{(p_{jx} - X_0)^2}{2(p_{jy} - l_y)} + \frac{1}{2}(l_y + p_{jy})$ and $X_0 = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$

where $a = 1 - \frac{p_{iy}-l_y}{p_{jy}-l_y}$;

$b = 2p_{jx}\frac{p_{iy}-l_y}{p_{jy}-l_y} - 2p_{ix}$ and

$c = p_{ix}^2 - p_{jx}^2\frac{(p_{iy}-l_y)}{(p_{jy}-l_y)} - (p_{iy}-l_y)(p_{jy}-p_{iy})$

# Appendix B

# Midpoint of circle

*Derivation of the midpoint coordinates of a circle, given three points on its circumference.*

$(x_0 - x_1)^2 + (y_0 - y_1)^2 = (x_0 - x_2)^2 + (y_0 - y_2)^2 = (x_0 - x_3)^2 + (y_0 - y_3)^2$
$\rightarrow x_0^2 - 2x_0x_1 + x_1^2 + y_0^2 - 2y_0y_1 + y_1^2 = x_0^2 - 2x_0x_2 + x_2^2 + y_0^2 - 2y_0y_2 + y_2^2$
$\rightarrow 2(x_2 - x_1)x_0 + 2(y_2 - y_1)y_0 + x_1^2 - x_2^2 + y_1^2 - y_2^2 = 0$

Similarly:

$2(x_3 - x_2)x_0 + 2(y_3 - y_2)y_0 + x_2^2 - x_3^2 + y_2^2 - y_3^2 = 0$

Thus

$$\begin{bmatrix} (x_2 - x_1) & (y_2 - y_1) \\ (x_3 - x_2) & (y_3 - y_2) \end{bmatrix} \begin{Bmatrix} x_0 \\ y_0 \end{Bmatrix} = \frac{1}{2} \begin{Bmatrix} x_2^2 - x_1^2 + y_2^2 - y_1^2 \\ x_3^2 - x_2^2 + y_3^2 - y_2^2 \end{Bmatrix}$$

or

$$\begin{bmatrix} a_1 & a_2 \\ b_1 & b_2 \end{bmatrix} \begin{Bmatrix} x_0 \\ y_0 \end{Bmatrix} = \begin{Bmatrix} c_1 \\ c_2 \end{Bmatrix}$$

$$\Longrightarrow \begin{Bmatrix} x_0 \\ y_0 \end{Bmatrix} = \frac{1}{a_1b_2 - b_1a_2} \begin{bmatrix} b_2 & -a_2 \\ -b_1 & a_1 \end{bmatrix} \begin{Bmatrix} c_1 \\ c_2 \end{Bmatrix}$$

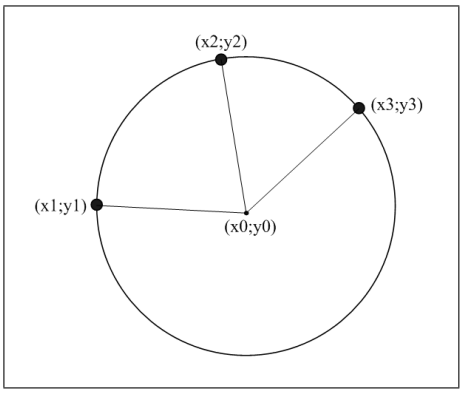



Figure B.1: Defining variables

# Appendix C

# Text file format

*Extract from Columns.txt:*
(x-coordinate, y-coordinate)

| | |
|---|---|
| 6.4 | 0.0 |
| 9.4 | 0.0 |
| 14.1 | 0.0 |
| 20.9 | 0.0 |
| 28.0 | 0.0 |
| 61.82 | 0.3 |
| 64.45 | 0.3 |
| 70.0 | 12.7 |
| 56.0 | 35.0 |
| 9.6 | 42.5 |
| 14.1 | 42.5 |
| 0.0 | 22.5 |
| 7.4 | 42.5 |
| 9.6 | 42.5 |
| 61.82 | 6.8 |
| 0.0 | 34.7 |
| 41.8 | 31.8 |

# Appendix D

# Parametric line-clipping

*Derivation of intersection coordinates of lines and edges.*

$P(t) = P_0 + t(P_1 - P_0)$

Solve for the value of $t$ at the intersection of $P_0P_1$ with the edge:

$N_i \bullet [P_t - P_{E_i}] = 0$

Substitute for $P(t)$:

$N_i \bullet [P_0 + t(P_1 - P_0) - P_{E_i}] = 0$

Group terms and distribute the dot product:

$N_i \bullet [P_0 - P_{E_i}] + N_i \bullet t[P_1 - P_0] = 0$

Let $D = P_1 - P_0$ be the vector from $P_0$ to $P_1$, and solve for $t$:

$t = \frac{N_i \bullet [P_0 - P_{E_i}]}{-N_i \bullet D}$

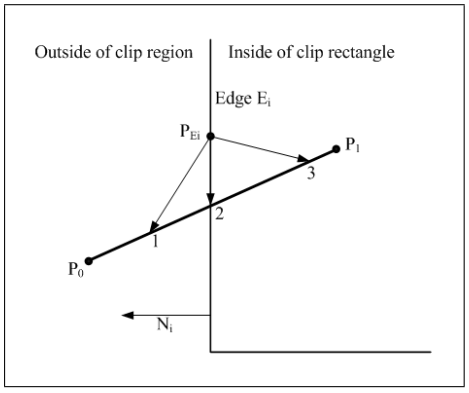



Figure D.1: Parametric line-clipping

# Appendix E

# Bounding line integration

*Derivation of bounding line integration formulae for linear, quadratic and cubic segments based on the Gauss divergence theorem.*

Gauss integration theorem:

$\int x^m y^n \, dA = \frac{1}{m+n+2} \int x^m y^n (xdy - ydx)$

Split the integral:

$\frac{1}{m+n+2} \int x^m y^n (xdy - ydx) = \frac{1}{m+n+2} \sum_{e=1}^{n} \int x^m y^n (xdy - ydx)$

$\implies A_{m,n} = \frac{1}{m+n+2} \sum_{e=1}^{n} \int_0^1 a_k x^m y^n \, dt$ where $xdy - ydx = a_k dt$

Substitute $m = n = 0$ for the area itself:

$A_{0,0} = \frac{1}{2} \sum_{e=1}^{n} \int_0^1 a_k dt = \frac{1}{2} \sum_{e=1}^{n} \int_0^1 (xdy - ydx)$

**1) For linear segments:**

$P(t) = sp(1-t) + ep(t), \ 0 \leq t \leq 1$

where $sp$ = the startpoint and $ep$ = the endpoint.

$\rightarrow x(t) = x_{sp}(1-t) + x_{ep}t$ and
$y(t) = y_{sp}(1-t) + y_{ep}t$

$\frac{dx}{dt} = x_{ep} - x_{sp}$
$\rightarrow dx = (x_{ep} - x_{sp})dt$

$\frac{dy}{dt} = y_{ep} - y_{sp}$
$\rightarrow dy = (y_{ep} - y_{sp})dt$

$(xdy - ydx) = [(x_{sp}(1-t) + x_{ep}t)(y_{ep} - y_{sp}) - (y_{sp}(1-t) + y_{ep}t)(x_{ep} - x_{sp})]dt$

$= (x_{sp}y_{ep} - y_{sp}x_{ep})dt$
$\rightarrow a_k = x_{sp}y_{ep} - y_{sp}x_{ep}$

$A = \frac{1}{2}\sum_{e=1}^{n}(x_{sp}y_{ep} - y_{sp}x_{ep})$

**2) For quadratic segments:**

Parametric equation for a quadratic segment, from the *Java API*:

$P(t) = B_{2,0}cp + B_{2,1}p1 + B_{2,2}p2,\ 0 \leq t \leq 1$

where $cp$ = the most recently specified (current) point,
$p1$ = the first control point and
$p2$ = the final interpolated control point.

$B_{n,m}$ = $m$th coefficient of $n$th degree Bernstein polynomial
$= C_{n,m}t^m(1-t)^{n-m}$.
$C_{n,m}$ = Combination of $n$ things, taken $m$ at a time
$= \frac{n!}{m!(n-m)!}$.

Thus $P(t) = [C_{2,0}(1-t)^2]cp + [C_{2,1}t(1-t)]p1 + [C_{2,2}t^2]p2$

$\rightarrow x(t) = C_{2,0}(1-t)^2x_{cp} + C_{2,1}t(1-t)x_{p1} + C_{2,2}t^2x_{p2}$ and
$y(t) = C_{2,0}(1-t)^2y_{cp} + C_{2,1}t(1-t)y_{p1} + C_{2,2}t^2y_{p2}$.

$\frac{dx}{dt} = C_{2,0}x_{cp}(2t-2) + C_{2,1}x_{p1}(1-2t) + C_{2,2}x_{p2}(2t)$
$\rightarrow dx = [C_{2,0}x_{cp}(2t-2) + C_{2,1}x_{p1}(1-2t) + C_{2,2}x_{p2}(2t)]dt$

Similarly,

$dy = [C_{2,0}y_{cp}(2t-2) + C_{2,1}y_{p1}(1-2t) + C_{2,2}y_{p2}(2t)]dt$

$$xdy - ydx = [C_{2,0}C_{2,1}x_{cp}y_{p1} - 2C_{2,0}C_{2,1}x_{cp}y_{p1}t + 2C_{2,0}C_{2,1}x_{cp}y_{p1}t^2 + 2C_{2,0}C_{2,2}x_{cp}y_{p2}t + 2C_{2,0}^2x_{cp}y_{cp}t^3 - C_{2,0}C_{2,1}x_{p1}y_{cp} + 2C_{2,0}C_{2,1}x_{p1}y_{cp}t - 2C_{2,0}C_{2,2}x_{p2}y_{cp}t]dt$$

$A = \frac{1}{2}\sum_{e=1}^{n}\int_0^1(xdy - ydx)$
$= \ldots$
$= \frac{1}{2}\sum_{e=1}^{n}[-\frac{1}{3}C_{2,0}C_{2,1}x_{p1}y_{cp} + \frac{1}{3}C_{2,0}C_{2,1}x_{cp}y_{p1} - \frac{1}{3}C_{2,0}C_{2,2}x_{p2}y_{cp} + \frac{1}{3}C_{2,0}C_{2,2}x_{cp}y_{p2} + \frac{1}{3}C_{2,1}C_{2,2}x_{p1}y_{p2} - \frac{1}{3}C_{2,1}C_{2,2}x_{p2}y_{p1}]$

**3) For cubic segments:**

Parametric equation for a cubic segment, from the *Java API*:

$P(t) = B_{3,0}cp + B_{3,1}p1 + B_{3,2}p2 + B_{3,3}p3$

where $cp$ = the most recently specified (current) point,
$p1$ = the first control point,

$p2$ = the second control point and
$p3$ = the final interpolated control point.

$B_{n,m}$ = $m$th coefficient of $n$th degree Bernstein polynomial
$= C_{n,m}t^m(1-t)^{n-m}$.
$C_{n,m}$ = Combination of $n$ things, taken $m$ at a time
$= \frac{n!}{m!(n-m)!}$.

Thus $P(t) = C_{3,0}(1-t)^3 cp + C_{3,1}t(1-t)^2 p1 + C_{3,2}t^2(1-t)p2 + C_{3,3}t^3 p3$

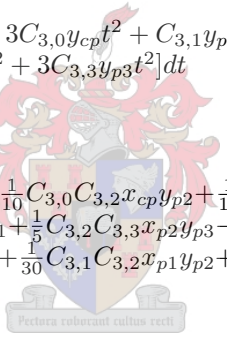
$\rightarrow x(t) = C_{3,0}x_{cp}(1-t)^3 + C_{3,1}x_{p1}t(1-t)^2 + C_{3,2}x_{p2}t^2(1-t) + C_{3,3}x_{p3}t^3$ and
$y(t) = C_{3,0}y_{cp}(1-t)^3 + C_{3,1}y_{p1}t(1-t)^2 + C_{3,2}y_{p2}t^2(1-t) + C_{3,3}y_{p3}t^3$

$$\begin{aligned}\frac{dx}{dt} =& -3C_{3,0}x_{cp} + 6C_{3,0}x_{cp}t - 3C_{3,0}x_{cp}t^2 + C_{3,1}x_{p1} - 4C_{3,1}x_{p1}t + 3C_{3,1}x_{p1}t^2 \\ &+ 2C_{3,2}x_{p2}t - 3C_{3,2}x_{p2}t^2 + 3C_{3,3}x_{p3}t^2 \\ \rightarrow dx =& [-3C_{3,0}x_{cp} + 6C_{3,0}x_{cp}t - 3C_{3,0}x_{cp}t^2 + C_{3,1}x_{p1} - 4C_{3,1}x_{p1}t + 3C_{3,1}x_{p1}t^2 \\ &+ 2C_{3,2}x_{p2}t - 3C_{3,2}x_{p2}t^2 + 3C_{3,3}x_{p3}t^2]dt\end{aligned}$$

Similarly,

$$\begin{aligned}dy =& [-3C_{3,0}y_{cp} + 6C_{3,0}y_{cp}t - 3C_{3,0}y_{cp}t^2 + C_{3,1}y_{p1} - 4C_{3,1}y_{p1}t + 3C_{3,1}y_{p1}t^2 \\ &+ 2C_{3,2}y_{p2}t - 3C_{3,2}y_{p2}t^2 + 3C_{3,3}y_{p3}t^2]dt\end{aligned}$$

$$\begin{aligned}A =& \tfrac{1}{2}\textstyle\sum_{e=1}^{n}\int_0^1 (xdy - ydx) \\ =& \ldots \\ =& \tfrac{1}{2}\textstyle\sum_{e=1}^{n}[\tfrac{1}{5}C_{3,0}C_{3,1}x_{cp}y_{p1} + \tfrac{1}{10}C_{3,0}C_{3,2}x_{cp}y_{p2} + \tfrac{1}{10}C_{3,0}C_{3,3}x_{cp}y_{p3} - \tfrac{1}{5}C_{3,0}C_{3,1}x_{p1}y_{cp} \\ &- \tfrac{1}{30}C_{3,1}C_{3,2}x_{p2}y_{p1} + \tfrac{1}{5}C_{3,2}C_{3,3}x_{p2}y_{p3} - \tfrac{1}{10}C_{3,0}C_{3,3}x_{p3}y_{cp} - \tfrac{1}{10}C_{3,1}C_{3,3}x_{p3}y_{p1} \\ &- \tfrac{1}{5}C_{3,2}C_{3,3}x_{p3}y_{p2} + \tfrac{1}{30}C_{3,1}C_{3,2}x_{p1}y_{p2} + \tfrac{1}{10}C_{3,1}C_{3,3}x_{p1}y_{p3} - \tfrac{1}{10}C_{3,0}C_{3,2}x_{p2}y_{cp}]\end{aligned}$$



# Appendix F

# Layout of apartment building floor

# Appendix G

# Comparative results for apartment building floor

# Appendix H

# Layout of industrial floor

# Appendix I

# Comparative results for industrial floor

# Appendix J

# Comparative results for parking lot

# Bibliography


[1] www.autodesk.com/autocad: AutoCAD software.

[2] www.cademia.de: CADEMIA - das Open Source CAD System.

[3] M. de Berg, M. van Kreveld, M. Overmars, O. Schwarzkopf, *Computational Geometry - Algorithms and Applications*, Springer, Berlin Heidelberg, 1997.

[4] L. Ammeraal, *Computer Graphics for Java Programmers*, John Wiley & Sons, Chichester New York, 1998.

[5] J.D. Foley, A. van Dam, S.K. Feiner, J.F. Hughes, R.L. Phillips, *Introduction to Computer Graphics*, Addison-Wesley Publishing Company, Reading New York, 1994.

[6] R.D. Cook, D.S. Malkus, M.E. Plesha, R.J. Witt, *Concepts and Applications of Finite Element Analysis*, John Wiley & Sons, New York, 2002.

[7] *SABS 0160-1989 (as amended 1990, 1991 and 1993): South African standard code of practice for the general procedures and loadings to be adopted in the design of buildings*, Council of the South African Bureau of Standards.


Pectora roborant cultus recti