

Kevin Berget
Martine Grahl-Nielsen

Developing a Python program to design concrete elements using the strut and tie model

Master's thesis in Civil and Environmental Engineering
Supervisor: Daniel Cantero
June 2022

Kevin Berget
Martine Grahl-Nielsen

Developing a Python program to design concrete elements using the strut and tie model

Master's thesis in Civil and Environmental Engineering
Supervisor: Daniel Cantero
June 2022

Norwegian University of Science and Technology
Faculty of Engineering
Department of Structural Engineering



MASTER THESIS 2022

SUBJECT AREA: Concrete Structures	DATE: June 10 th , 2022	NO. OF PAGES: 73 + 33
--------------------------------------	---------------------------------------	--------------------------

TITLE:

Developing a Python program to design concrete elements using the strut and tie model

Utvikling av et Python program for å designe betongelementer ved bruk av stavmodellen

BY:

Kevin Berget
Martine Grahl-Nielsen



SUMMARY:

The strut and tie model (STM) is a method for designing concrete structures and is a valuable tool in regions where the use of standard beam theory is insufficient. STM uses the flow of forces through a structure to create an imaginary truss system and do design checks on this system. However simple using STM can be time-consuming. Therefore, this thesis has created a Python program to calculate and do necessary design checks of strut and tie models. The Python program is developed to make it easy to establish and check a given strut and tie model, using the design rules of Eurocode 2.

Some examples have been established to showcase that the calculation in the program is correct compared to hand-calculated problems. Using the program instead of hand calculation makes the calculation more efficient and with less chance of consequential error. The program also allows changing the STM fast and effectively to calculate many different STMs or iterate with minor changes on an existing one.

This thesis has also developed a graphical user interface (GUI) with the program as a base. The GUI will make the program more accessible for more people to establish and check strut and tie models.

Lastly, a small study has been conducted using the program. This study investigates simply supported deep beams with point loads and tries to optimize the STM. The result of this study shows that fewer vertical ties in the STM lead to less strain energy and thus "better" STMs, given that the design checks are okay.

Here is the link to the online repository of the developed program:
<https://gitlab.stud.idi.ntnu.no/martgrah/stm>

RESPONSIBLE TEACHER: Daniel Cantero, NTNU

SUPERVISOR(S): Daniel Cantero, NTNU

CARRIED OUT AT: Department of Structural Engineering

Abstract

The strut and tie model (STM) is a method for designing concrete structures and is a valuable tool in regions where the use of standard beam theory is insufficient. STM uses the flow of forces through a structure to create an imaginary truss system and do design checks on this system. However simple using STM can be time-consuming. Therefore, this thesis has created a Python program to calculate and do necessary design checks of strut and tie models. The Python program is developed to make it easy to establish and check a given strut and tie model, using the design rules of Eurocode 2.

Some examples have been established to showcase that the calculation in the program is correct compared to hand-calculated problems. Using the program instead of hand calculation makes the calculation more efficient and with less chance of consequential error. The program also allows changing the STM fast and effectively to calculate many different STMs or iterate with minor changes on an existing one.

This thesis has also developed a graphical user interface (GUI) with the program as a base. The GUI will make the program more accessible for more people to establish and check strut and tie models.

Lastly, a small study has been conducted using the program. This study investigates simply supported deep beams with point loads and tries to optimize the STM. The result of this study shows that fewer vertical ties in the STM lead to less strain energy and thus "better" STMs, given that the design checks are okay.

Here is the link to the online repository of the developed program:

<https://gitlab.stud.idi.ntnu.no/martgrah/stm>

Sammendrag

Stavmodellen er en metode for å designe betongkonstruksjoner, og er et verdifullt redskap i områder hvor det ikke er mulig å benytte vanlig bjelketeori. Metoden bruker fordelingen av krefter gjennom konstruksjonen for å danne et imaginært fagverkssystem, for så å gjøre kapasitetskontroller av disse. Til tross for stavmodellens enkelhet, kan den være ganske tidkrevende å bruke. Derfor har det i dette prosjektet blitt laget et Python-program for å beregne og gjennomføre nødvendige kapasitetskontroller av stavmodellen. Python-programmet er laget for å gjøre det enkelt å etablere og sjekke enhver stavmodell med utgangspunkt i reglene fra Eurokode 2.

Det har blitt laget noen eksempler for å vise at beregningene fra programmet samsvarer med håndberegninger. Ved bruk av programmet fremfor håndberegninger, blir utførelse av beregninger mer effektivt i tillegg til en lavere sjanse for følgefeil. Programmet gir også muligheten til å endre på stavmodellen raskt og enkelt, for å kunne regne mange forskjellige stavmodeller eller iterer eksisterende stavmodeller med små forandringer.

Det har også blitt utviklet et brukergrensesnitt til programmet. Dette brukergrensesnittet vil gjøre programmet mer tilgjengelig.

Til slutt har det blitt gjennomført en liten studie ved bruk av programmet. Denne studien undersøker fritt opplagte dype bjelker med punktlast for å prøve å finne den optimale stavmodellen. Resultatene herfra viser at færre vertikale strekkstaver i stavmodellen fører til mindre tøyingsenergi, og dermed en «bedre» stavmodell, gitt at kapasiteten er ok.

Her er lenken til oppbevaringsstedet til Python programmet:

<https://gitlab.stud.idi.ntnu.no/martgrah/stm>

Preface

This master's thesis is the final part of a master's degree in Civil and Environmental Engineering, written for the Department of Structural Engineering at the Norwegian University of Science and Technology (NTNU), Trondheim. The thesis was written over 20 weeks during the spring semester of 2022.

This thesis was chosen because of our interest in using coding to simplify calculations and methods used in structural engineering. Strut and tie modeling lends itself to being a prime candidate for developing a program around. In combination with relatively scarce information about strut and tie and the absence of a freely available program to design strut and tie, the choice fell on this part of concrete design.

In this thesis, we have learned a lot about strut and tie modeling and its design process and improved our skills in using Python and creating a program. Developing the program has been exciting and challenging.

We want to thank our supervisor Daniel Cantero, who presented this exciting topic and helped us to the best of his abilities.

Trondheim, June 10th, 2022

Kevin Berget

Martine Grahl-Nielsen

Table of content

1	Introduction	1
1.1	Background	1
1.2	Purpose of the thesis	1
1.3	Existing programs.....	1
1.4	Report Outline.....	2
2	Strut and tie model	3
2.1	General.....	3
2.2	Separate B- and D-regions	4
2.3	Develop the strut and tie model.....	6
2.4	Design the members	7
2.5	Minimize the strain	13
3	Software and Tools	14
3.1	General.....	14
3.2	Python	14
3.3	Spyder	14
3.4	Git.....	15
4	The program	16
4.1	General.....	16
4.2	Class setup	16
4.3	Node.....	17
4.4	Element	19
4.5	System.....	20
4.6	ElementForce	23
4.7	Checks	27
4.8	Plot	29
4.9	Outputs	30
5	User manual	31
5.1	General.....	31
5.2	Downloading the repository	31
5.3	Python script	32
5.4	Graphical user interface.....	37
6	Examples.....	42

6.1	General.....	42
6.2	Example 1: two-pile cap.....	42
6.3	Example 2: deep beam.....	46
6.4	Example 3: Deep beam with large openings and recess.....	49
7	Study	55
7.1	General.....	55
7.2	Study one: Symmetrical deep beam	56
7.3	Study two: Deep beam with non-centered point load	59
7.4	Discussion of the studies.....	63
7.5	Conclusion of the studies.....	63
8	Discussion	64
8.1	General.....	64
8.2	The process	64
8.3	Limitation	65
8.4	Examples	66
8.5	Study	66
8.6	Using the program	67
9	Conclusion	68
10	Future Work	69
	References	70
	Appendix.....	73

1 Introduction

1.1 Background

Strut and tie model/modeling (STM) is a method for designing complex concrete structures. It can be used in all concrete structures but is most effective where standard beam theory is inadequate. Strut and tie modeling is a simple but powerful method to design these regions, called D-regions. D-regions would generally need lots of advanced calculations to be designed. In contrast, STM can be done with hand calculations and a fraction of the calculations.

Even though the calculations are simple, there are a few drawbacks to using strut and tie modeling as a design tool. One of them is that the number of calculations rapidly increases in STMs with more than a couple of trusses, resulting in a long calculation process. Another drawback is that if the design fails in just one of the design checks, the whole STM fails, and a redesign is needed. Most of the calculations must be redone for each redesign, which adds a lot of time and resources. Lastly, since there are many ways of constructing an STM, some solutions may be better than others. So, iterations are needed to design a suitable structure using STM. Each iteration needs to go through all the previous steps in the design to get a satisfactory result. So, even though STM is a simple and effective method for concrete design, the complete procedure of an optimal design can take a long time and use many resources.

1.2 Purpose of the thesis

This thesis work aims to develop a Python program to calculate and control the capacity of any given strut and tie model. This program will help overcome the time- and resource-consuming part of strut and tie modeling. The benefits of the Python program are quickly checking if a given STM is okay and quickly changing the values of the STM to iterate and find the optimal solution. Later in the thesis, a small study will be completed using the developed program. This study is meant to learn something about the strut and tie model and its behavior and showcase some potential program uses.

1.3 Existing programs

During research for this thesis, papers and studies that had used software to calculate STM emerged. The programs used in these studies seemed more advanced than those developed in this project. However, these programs have been used to study and analyze different aspects of STM, and the programs were just tools for these analyses. Thus, these programs are not publicly available or even discussed in detail. The program developed in this thesis is meant to be a helping tool in the design process of STM and will be made publicly available.

A commercially available program for doing design with the strut and tie model, AStrutTie, was found during research for this thesis. However, this program being a commercial tool is quite expensive. Given that the nature of this thesis was to develop a freely available program, this program is not seen as a contender in the same market.

1.4 Report Outline

Section 1 is the introduction to the thesis. This section presents some background information and the purpose of the thesis.

Section 2 presents the theory of the strut and tie model. It will present how to establish an STM and go through all the necessary design checks and rules.

Section 3 will present the software and tools used to develop the program.

Section 4 will present the program. It will go through the classes, how they work and are built, and explain the code and workflow of the program.

Section 5 will present a user manual explaining one way of using the code. It also explains how to use the graphical user interface.

Section 6 will present some strut and tie examples. It will compare the results of precalculated examples with the results from the program.

Section 7 will present a short study using the program and trying to optimize some STMs for deep beams.

Section 8 presents the discussion. It outlines the process of developing the program, its limitations, and some additional comments on the examples and study.

Section 9 presents a conclusion of the whole thesis

Section 10 presents some suggestions for future work on this topic

2 Strut and tie model

2.1 General

Strut and tie modeling is an effective method for doing design checks on complicated concrete structures/elements. Instead of doing design checks for the entire element, STM focuses on the stress patterns in the element and designs according to them. The stress pattern is usually represented by an imaginary triangular truss pattern, where the compression trusses represent the concrete strut, and the tension trusses represent the steel reinforcement ties. STM can be applied in almost any concrete element design, in both serviceability limit state and ultimate limit state, even though it is most common in ultimate limit state design. However, STM is especially useful in zones or regions where it is impossible to use “normal” beam theory, so-called D-regions (Tae, 2021). Structures where plane sections remain plane cannot be designed using “normal” beam theory. Other calculations are required in these cases; this is where STM comes in.

The goal of STM is to determine the flow of the internal forces in the structure to perform the design checks in the critical areas of the element. An STM consists of nodes, ties, and struts. The struts and ties should follow the stress patterns in the element, and the nodes are the interfacial zones between struts and ties. It is necessary to do design checks on each of these components. An example of an STM is illustrated in Figure 1.

The strut and tie model is based on the lower bound theorem of limit analysis. The lower bound theorem states that if equilibrium can be found that balances the applied loads and is everywhere below or at the plastic moment value, the structure will not collapse or be at the point of collapse. In other words, the lower bound theorem ensures the strut and tie model is safe and conservative as long as the equilibrium and yield criteria are satisfied. The collapse load itself is the most significant load value to come out of an STM. The lower bound theorem is especially useful in tension-weak materials like concrete. (Chen & El-Metwally, 2017). This gives that if the STM is reasonably well designed and passes all the design checks, the result will be on the safe side of collapse.

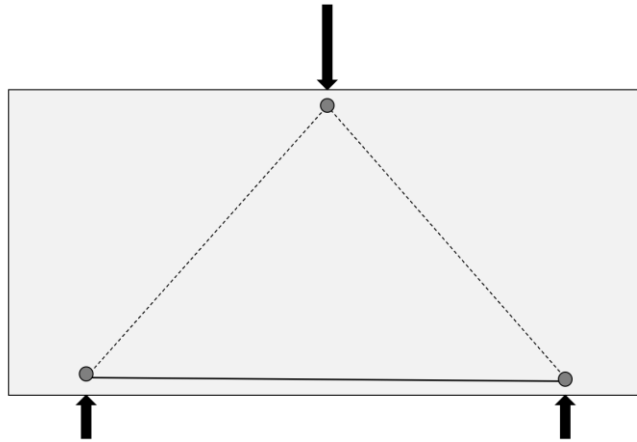


Figure 1: Example of a strut and tie model

There are several steps to follow when designing an STM. Later sections will discuss the design steps for the STM in more detail. The process for designing a strut and tie model is as follows:

1. Separate B- and D-regions, see section 2.2
2. Develop the STM, see section 2.3
3. Design the members of the STM, see section 2.4
4. Minimize the strain energy by optimizing the model, see section 2.5

2.2 Separate B- and D-regions

There are two different types of regions in a concrete structure. These regions are called B- and D-regions. In B-regions, also called Bernoulli regions, the strains are linear, making it possible to use existing methods for design checks and calculating required reinforcement. At D-regions, also called discontinuity or distribution regions, the strains are non-linear. Therefore, it is impossible to use the standard design rules for D-regions. Even though the strut and tie model can be used in both B- and D-regions, it is most useful in D-regions. Hence, D-regions will be the primary goal of this thesis.

There are often multiple D-regions in a structure. D-regions are where there is a discontinuity in the geometry or the load on a structure. Examples of D-regions are, among others, deep beams, frame corners, dapped ends, corbels, or structures with holes. Examples of D-regions with geometry disturbances are shown in Figure 2, and load disturbances are shown in Figure 3.

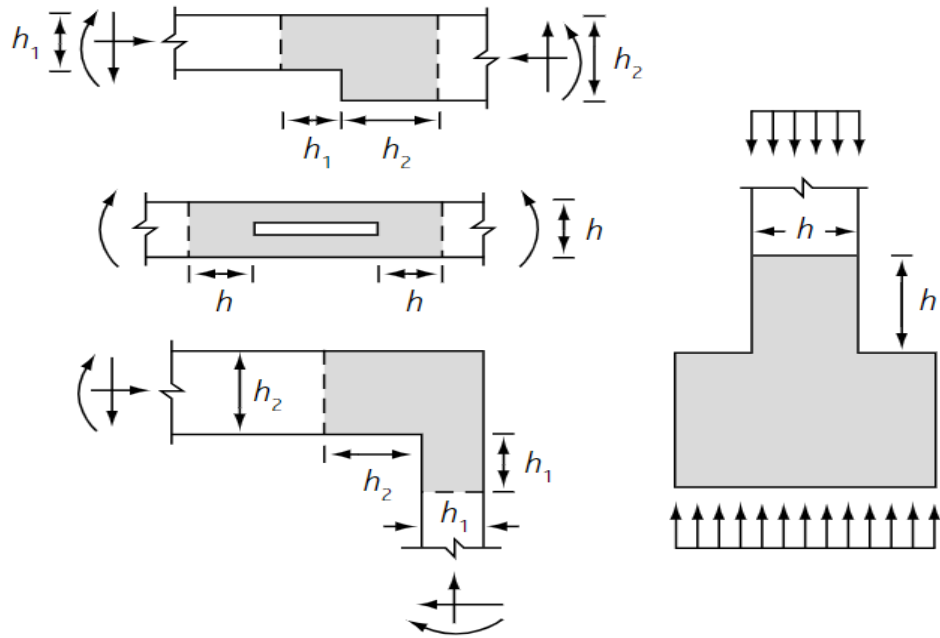


Figure 2 Examples of geometry D-regions. (Goodchild et al., 2014)

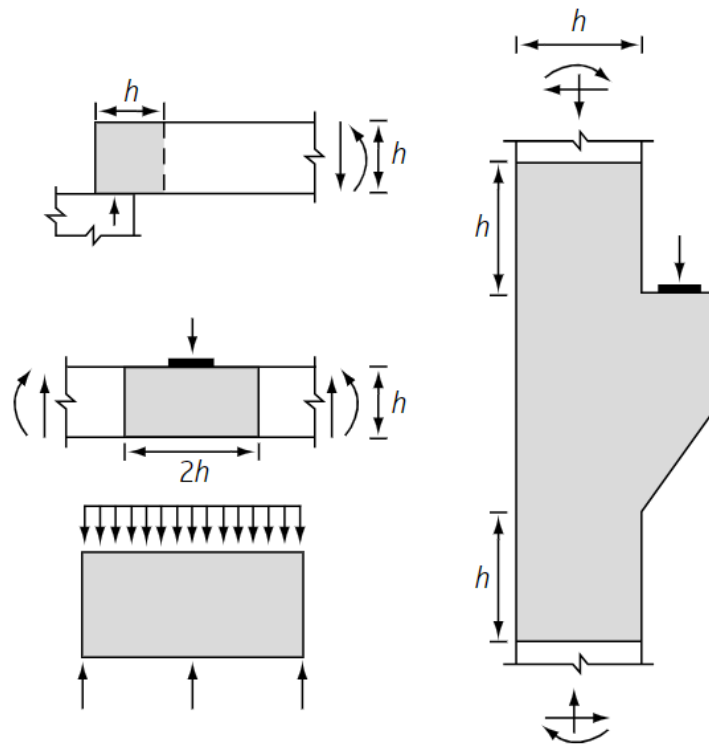


Figure 3 Examples of load and/or geometry D-regions, marked with grey. (Goodchild et al., 2014)

2.3 Develop the strut and tie model

The process of developing an STM is as follows:

1. Find the load paths in a structure
2. Choose the position of the struts, ties, and nodes
3. Optimizing the Solution

When creating an STM, it is essential to remember that the model should allow for sufficient cover over the reinforcement. In figures of strut and tie models, struts are represented by dashed lines, fully drawn lines represent ties, and dots represent the nodes.

The first step in developing an STM is determining the flow of forces through the structure. There are many ways to do this. Experience and common sense are one way (Tae, 2022). For example, in a simply supported deep beam with a central point load, it is simple to find the load path. In this case, the load-path will follow a triangular shape with compression in the diagonals and tension parallel to the structure in the lower section, like the example shown in Figure 1. Finite element analysis (FEA) is another way of determining the flow of forces in the structure. FEA uses numerical mathematical models to find the direction and magnitude of forces in elements or structures (English, 2019). FEA can be helpful in more complex concrete structures and load cases, as these can give more reliable results than human experience.

After determining the load paths, the next step is to place the struts and ties. Struts should be placed where there is compression and ties where there is tension. Struts can be placed wherever in a structure but should be placed where there is compression. Ties should be placed where there is tension in the structure. However, where struts are “imaginary” and represent concrete, ties represent the reinforcement in the structure. Hence, orienting the ties parallel or perpendicular to the element’s surface is common. This placement of the ties is done to help ease the structure’s construction, and dealing with angles in construction is laborious and a source of errors, but not impossible. Some other rules are implemented in the STM setup to avoid strain incompatibilities, like that the minimum angle between a strut and a tie meeting at a node should not be less than 35 degrees (Goodchild et al., 2015).

There are often several ways to create an STM. Figure 4 shows an example of one structure with several STM solutions. So, the last step to developing an STM is to decide the “best” solution. The total strain energy typically chooses the “best” strut and tie model. The STM with less strain energy results in a “better” STM and will be explained more in-depth in section 2.5.

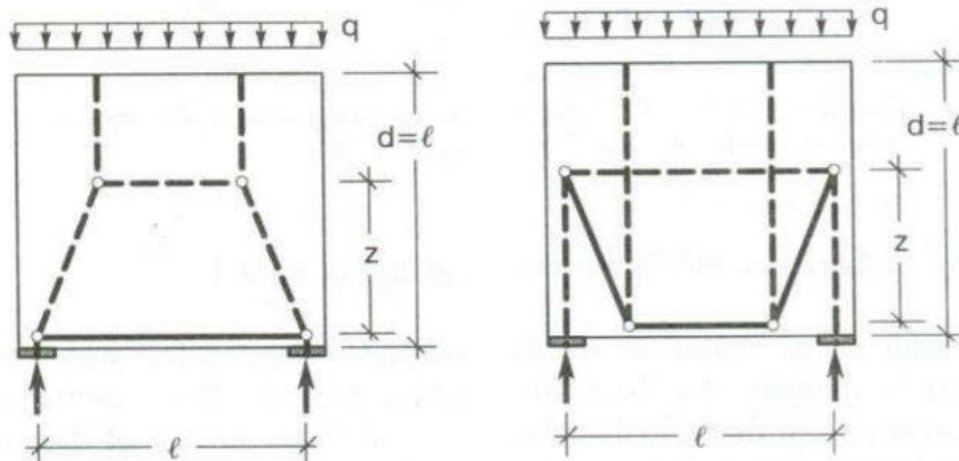


Figure 4: Example of possible valid strut and tie models for the same load case (Walter, 2017)

2.4 Design the members

2.4.1 General

When the STMs geometry is defined, the next step is to design the components. The components refer to the STMs struts, ties, and nodes. Some parameters must be established before the design checks. Some of them are stated, and others are calculated.

The parameters that need to be given before the design checks are:

- The concrete class
- The breadth of the concrete element
- The distance from the surface of the structure to the center of the tie
- External forces on the element
- The placement of the strut, ties, and nodes in the STM

From the given parameters, trigonometry and equilibrium equations can quickly determine the angles and forces in the STM. As for the dimensions of the trusses, it is a bit more complicated. The breadth of the trusses is the same as the breadth of the concrete element, but node geometry determines the width of the trusses.

This section will go through the design of each of the members in the STM: nodes, struts, and ties. The design checks will be done according to the checks in Eurocode 2.

2.4.2 Nodes

2.4.2.1 General

There are two ways to categorize nodes: smeared and concentrated nodes. Within large STMs with many nodes, smeared nodes are the most common. In these nodes, the concrete stress is usually not critical and, therefore, usually not checked.

Concentrated nodes are usually highly stressed and need to be carefully designed. According to EC2 6.5.4(3), concentrated nodes may develop where point loads are applied, at supports, in anchorage zones of prestressed tendons, and at bends in reinforcements, connections, and corners of members.

There are three types of concentrated nodes. Compression node without ties (CCC). Compression tension node with reinforcement in one direction (CCT). Compression tension node with reinforcement in two directions (CTT). A CCC-node is shown in Figure 5, CCT-node in Figure 6, and CTT-node is in Figure 7.

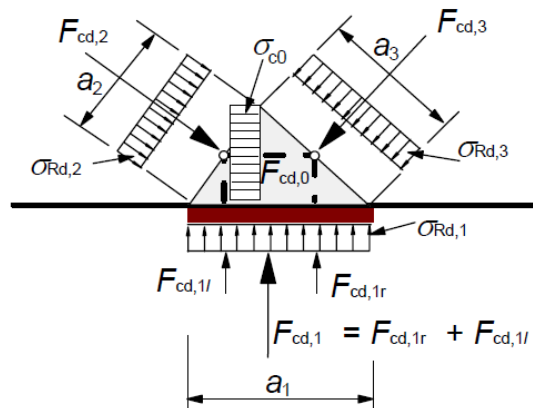


Figure 5 CCC-node from Eurocode 2

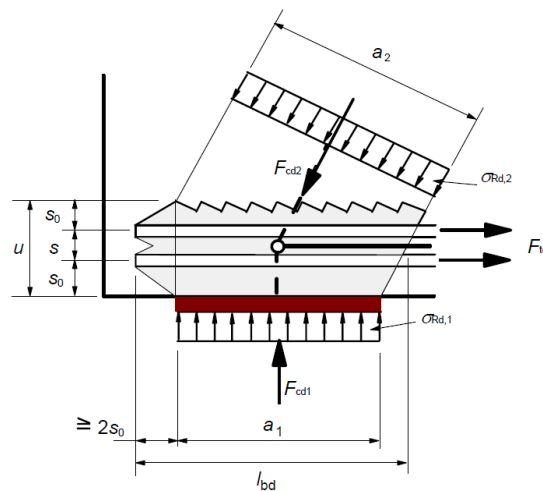


Figure 6 CCT-node, from Eurocode 2

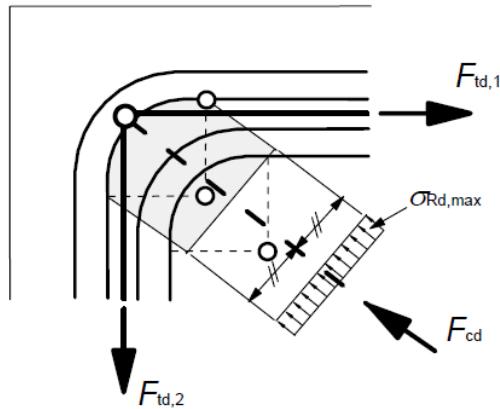


Figure 7 CTT-node, from Eurocode 2

2.4.2.2 Geometry of the node

The support length and the width of the ties determine the geometry of CCT-nodes at the supports. The support length is a given value, and the width of the ties (u) is usually double the distance between the concrete surface and the reinforcement's center. With these two values, see Figure 8, the width of the strut can be found by the formula:

$$a_2 = a_1 \sin \theta + u \cos \theta$$

(1)

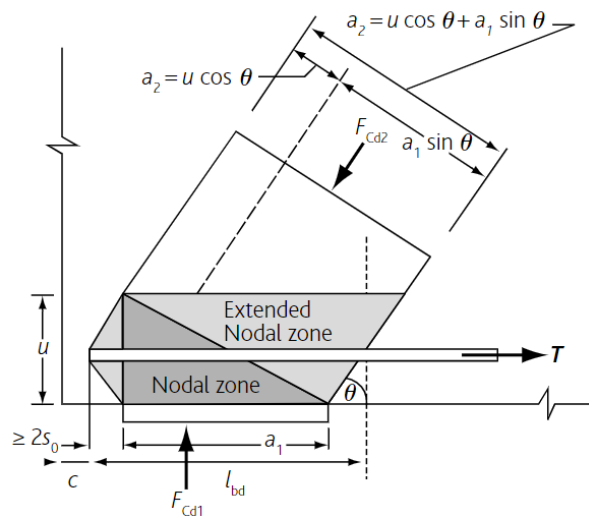


Figure 8: Example of a CCT-node (Goodchild et al., 2015)

For a CCC-node below a load, the length of the load is defined. However, the height of the node is not known. If it is a horizontal strut, as shown in Figure 9, the node's height is the same as the width of the horizontal strut. There are no rules to establish this width (Colorito et al., 2017). There are many ways of calculating this width. However, they are pretty tricky to calculate and have varying degrees of accuracy (Todisco, 2009). Thus, for simplicity's sake,

this thesis assumes the width of the flat strut to be the same as for the tie (u). A similar method to the CCT-node is used to get the dimensions of the diagonal strut, and the formula becomes:

$$a_2 = a_1 \sin \theta + u \cos \theta \tag{2}$$

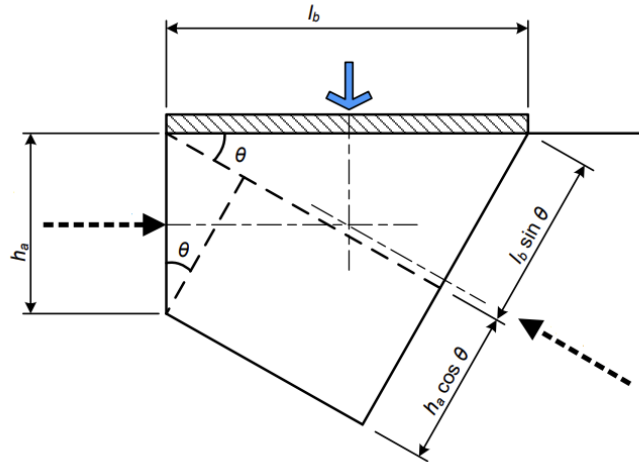


Figure 9: Geometry of a CCC-node with a flat strut (Colorito et al., 2017)

For a triangular CCC-node with an external force, the size of the struts is calculated by trigonometry and the sine rule (Varsity Tutors, 2007):

$$\frac{a}{\sin(A)} = \frac{b}{\sin(B)} = \frac{c}{\sin(C)} \tag{3}$$

Sometimes CCC-nodes have more than three struts meeting at a node. This is problematic as the node's design methods are based on just three struts at the node. When this happens, there is a need for an extra calculation step. This new step is to calculate resultants from the struts and the angle of these resultants until there are just three forces left, as demonstrated in Figure 10. External forces or flat/horizontal forces should, as far as possible, not be included in the resultants. When three forces have been achieved, the geometry is calculated the same way as described previously.

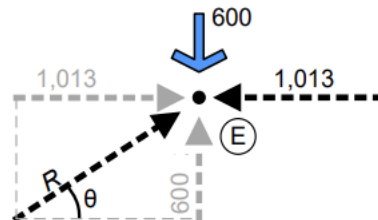


Figure 10: Example of combining two forces (Colorito et al., 2017)

A CCC-node without external force or limiting geometry is considered a smeared node; therefore, no calculations are necessary. (Colorito et al., 2017)

2.4.2.3 Design compressive stress

The design strength of these nodes is given in the equations (6.60-6.61) in Eurocode 2.

The design strength for CCC-nodes:

$$\sigma_{Rd,max} = k_1 v' f_{cd} \quad (4)$$

Where:

$$v' = 1 - f_{ck}/250 \quad (5)$$

The k_1 factor given in the national annex NA.6.5.4, and is equal to 1.

The design strength for CCT-nodes:

$$\sigma_{Rd,max} = k_2 v' f_{cd} \quad (6)$$

The k_2 factor given in the national annex NA.6.5.4, and is equal to 0,85.

The design strength for CTT-nodes:

$$\sigma_{Rd,max} = k_3 v' f_{cd} \quad (7)$$

The k_3 factor given in the national annex NA.6.5.4, and is equal to 0,75.

2.4.3 Struts

Section 6.5.2 in Eurocode 2 details the design of the struts. The first thing that must be checked is whether the strut experiences transverse tensile forces. If it does not, the design strength for the strut is given by equation (6.55) in EC2.

$$\sigma_{Rd,max} = f_{cd} \quad (8)$$

However, if this is not the case, the strut may experience transverse tensile forces, and a more rigorous method is not used. The new design strength of the strut is calculated by equation (6.56).

$$\sigma_{Rd,max} = 0.6 v' f_{cd} \quad (9)$$

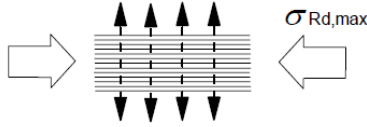


Figure 11: From Eurocode 2, strut with transverse tension

This check must be done at both ends of the strut, as there may be different stresses. In this thesis, it is assumed that all the struts may experience transverse tensile forces. This is done always to be conservative in the design checks.

2.4.4 Ties

EC2 6.5.3 describes the design of ties. Clause 6.5.3(1) in Eurocode 2 says the reinforcement should be limited by the rules in sections 3.2 and 3.3. These sections explain the rules and limitations of reinforcement and prestressing steel, respectively. With the use of ordinary Norwegian reinforcement and prestressing steel, these requirements are met. Thus, the design strength of the ties is given by:

$$\sigma_{Ed} = \frac{T}{A_s} < f_{yd} \tag{10}$$

Where:

$$f_{yd} = \frac{500}{1.5} = 434 \text{ N/mm}^2 \tag{11}$$

$$A_s = \frac{T}{f_{yd}} \tag{12}$$

Where:

- A_s is the reinforcement area
- T is the force in the ties

In this thesis, there is assumed sufficient anchorage.

2.5 Minimize the strain

Given that there are many ways to set up a strut and tie model, and many of them can satisfy the design criteria, there should be a way to know which model is the best. "Best" can be interpreted in different ways: most economical, easiest to construct, the strongest, etc. Usually, the best solution is the STM that minimizes the strain energy in the STM (Schlaich et al., 1987). The total strain energy is measured by summing each truss's strain energy. This gives the equation:

$$\text{Strain energy} = \sum F_i l_i \varepsilon_{mi} \quad (13)$$

Where:

- F is the force in the truss
- l is the length of the truss
- ε is the strain in the truss

In an STM, the strain energy from the compressive struts is small compared to the yield strains in steel. This gives that strain energy from struts can be excluded from the calculation, such that only the ties in the STM are used in the formula. (Hu et al., 2014) As the reinforcement amounts are chosen just to be at yielding, it is assumed that the stresses in the ties are the yield stress. Then the formula can be rewritten as:

$$\text{Strain energy} = \sum_i^{n^*} \frac{A_i l_i f_{yd}^2}{E} \quad (14)$$

Where:

- n^* is the number of ties
- E is the Young's modulus of steel reinforcement set to 200 GPa
- f_{yd} being the yield strength of steel reinforcement, see formula (11)

This rewritten formula shows that the strain energy of the STM is given by the term $A_i l_i$ times some constants. The term $A_i l_i$ can be seen as the total volume of steel in the STM, which gives that the minimizing of the strain energy also minimizes the steel usage and gives the "best" model. (Schlaich et al., 1987)

For now, iteration and trial and error are the methods to minimize the strain. The biggest problem with this is that all the design checks and strain energy calculations have to be done again for every iteration, making this iterating method very intensive, especially if done by hand. One of the ways to overcome this is with knowledge and experience, as they give an indicator of how the best model might look and therefore do not have to calculate the STMs known to be worse.

3 Software and Tools

3.1 General

In this day and age, there are many computer tools and software available to assist with lots of tasks and problems. Given that this thesis is developing a computer program to assist with a strut and tie modeling design, several external software and tools have been used. This section will explain the programming language, development environment, cross-platform collaboration, and file-sharing software used for this thesis.

3.2 Python

Python is an open-source high-level programming language (Python Software Foundation, 2012). It is designed with an emphasis on code readability. Python is mainly designed as an object-oriented programming language. The work on Python was started in the late 1980s, and the first release was Python 0.9.0 in 1991 (Python Software Foundation, 2022). It has later been revised, and new features have been introduced. In 2008, Python 3 was released, and in this thesis, Python 3.9 have been used.

Python is a widely used programming language. Since Python is so widely used, a lot of help and information exists online. There is also a large community that uses Python as their language and shares their codes. Thus, if there is a problem one wants to solve, the chances are that others have already done it or something similar that one can take inspiration from (Kuhlman, 2012).

One of the reasons Python was chosen as the programming language in this thesis is that the authors had some prior experience coding in Python. This, combined with the open-source and collaborative community surrounding Python, made it a prime candidate for this thesis.

3.3 Spyder

Spyder (Scientific PYthon Development EnviRonment) is an open-source, cross-platform integrated development environment for scientific programming in Python. Pierre Raybaut created it in 2009 (Raybaut, 2009). However, since 2012 it has been maintained and improved by scientific Python developers and the Spyder community. Spyder is designed by and for scientists, engineers, and data analysts. It has several Python packages geared towards scientific use already integrated (Spyder Doc Contributors, 2022).

Spyder was chosen as the development environment for this thesis because it is geared towards scientific programming. The already integrated packages have been of considerable help, and there has not been a need to download other packages than those already integrated with Spyder. Further, the variable explorer was another reason for using Spyder. It has been great for quality control during coding and code testing, and even debugging.

3.4 Git

3.4.1 General

Git is a version control system to handle any type of project. It is an open-source system developed by Linus Torvalds in 2005 (Spinellis, 2012). Git is a valuable tool when more than one person simultaneously codes on the same project. All the files and documents needed for a project are stored in a Git repository. The programmer then clones this repository to their computer, where local changes can be made. The programmer must commit and push the changes back to the repository (*GitLab*, 2022). With Git, it is possible to see the entire commit history and restore earlier commits. Some of the features of Git that are used in this thesis are branching, merging, committing changes, and seeing previous changes.

This project has three branches, one for each of the authors and the main branch. When the code is finished and working, it has been committed to the main, and the other person is responsible for implementing the changes into their branch. The reason Git was chosen is that it makes it easy to keep the code up to date when other authors have made changes. Another reason was to learn more about Git and get some experience using Git since this is a valuable tool to know.

3.4.2 GitLab

In this thesis, the program has been stored in the open-source software GitLab. GitLab is built on top of Git (O'Grady, 2018). GitLab is used to manage our repository and share it online. The reason for choosing GitLab was that the authors had some previous experience using GitLab. The user interface online makes it easy to use and understand.

Here is the link to the online GitLab repository:

<https://gitlab.stud.idi.ntnu.no/martgrah/stm>

3.4.3 GitHub Desktop

GitHub Desktop was used to access the cloned repository locally on the computer. GitHub is similar to GitLab, which is an open-source distributor of Git. GitHub Desktop is available for GitLab repositories. GitHub Desktop makes it easy to access the repository locally by using the user interface instead of command lines, which is the standard for Git. The user interface of the GitHub desktop is why it was chosen for this thesis since the user interface made it easy to use. The features used on GitHub Desktop are, commit, where the changes to the branches are committed and then pushed to transfer the changes to the server. When it has been pushed, a merge commits with the main is done, and when doing this, this procedure has been used: first merging the main into each branch to fix conflicts, and then merging the branches into the main. This way main will be up to date with the updates. This procedure ensures that any conflict is fixed before committing to the main. (*GitHub*, 2022)

4 The program

4.1 General

This program aims to speed up going through all the necessary steps in calculating an STM and, in other words, automizing the capacity checks of a given STM. It is then easier to find the most efficient STM since it is easy to test several alternatives of the STM.

When writing the program, it was intentionally written as general as possible, such that the project could take many different forms from there and not be limited. The code is easy to use for external people and easy to get results. It is also easy to expand the code and add features in the future for users and the developer.

The programming method used in this program is object-oriented programming (OOP). Objects are the fundamental building block in object-oriented programming. The most popular method used in OOP is class-based. This method is used in this program. A class is a coded template for the setup of an object. The class defines the object's initial values and variables, either set values or from a given input. If there are many objects with similar properties, different value classes are beneficial. Here is an example to explain the concept of classes and objects.

In this section, the geometry of the strut and tie model is referred to as a system, and the struts and ties are referred to as elements. The respective classes are called **System** and **Element** in the code. The variable names are marked with *cursive*, the functions with *cursive and bold*, and the classes are marked with **Bold** lettering. The first letter in the class names is upper case, and variables and functions have lower case first letters.

4.2 Class setup

When the program was written, the class setup was chosen to be easy to expand and adjust as it takes form. Therefore, the main classes for the setup of the STM are Node, Element, and System. These classes define the geometry of the model. Additional to these classes, there is a class to calculate the forces in the elements, one to do capacity checks, and one to plot the results.

The classes in the STM program are:

- **Node**, see section 4.3
- **Element**, see section 4.4
- **System**, see section 4.5
- **ElementForce**, see section 4.6
- **Checks**, see section 4.7
- **Plot**, see section 4.8

All the classes will be discussed in their section.

4.3 Node

A node is the connection point where the struts and ties meet. In this class, there are several callable variables. These variables are:

- *nodePosition*, the position of the node
- *nodeX*, the position of the node in the x-direction
- *nodeY*, the position of the node in the y-direction
- *forceMagnitude*, the applied external force
- *forceAngle*, the angle of the applied external force
- *forceWidth*, the width of the applied external force
- *isOkay*, whether the capacity of the node is sufficient when it is run through the design checks
- *forceX*, the external force in the x-direction
- *forceY*, the external force in the y-direction
- *nodeType*, the type of the node

When the node class is called, it calls the initialization function, see Figure 12. This function assigns values to the callable variables. The initialization function has several arguments, which will be listed here:

- *nodePosition*
- *forceMagnitude* (optional)
- *forceAngle* (optional)
- *forceWidth* (optional)

```

5 class Node:
6     def __init__(self,nodePosition_,forceMagnitude_= 0,forceWidth_= 0,
7                 forceAngle_=np.pi/2):
8         #The nodePosition must be a list that represent the coordinates to the node
9         #in integer numbers. The rest of the arguments should be integer numbers.
10        if(isinstance(nodePosition_, list)):
11            if(len(nodePosition_)==2):
12                if(isinstance(nodePosition_[0], int) and
13                    isinstance(nodePosition_[1], int)):
14                    self.nodePosition = nodePosition_
15                    self.nodeX = nodePosition_[0]
16                    self.nodeY = nodePosition_[1]
17            else:
18                print("The node list needs to contain integer numbers")
19        else:
20            print("The node list has to be of length 2")
21    else:
22        print("Then nodePosition needs to be of type list")
23    self.forceMagnitude = 0
24    self.forceAngle = 0
25    self.forceWidth = 0
26    self.isOkay=True
27    if(forceMagnitude_!=0):
28        self.addForce(forceMagnitude_,forceWidth_,forceAngle_)

```

Figure 12 Initialization function for **Node** class

A node must have a position in the x- and the y-direction, represented by a Python list. The *nodePosition* should be on the form [x,y], where x should be replaced by an integer number representing the node's position in the x-direction and y in the y-direction. There are three checks on the input argument to ensure that the *nodePosition* is written correctly. The first checks if it is a list, the second checks if the length of the list is equal to two, and the last checks if it contains integer numbers.

In each node, there is an option to apply external forces. The input *forceMagnitude*, *forceAngle*, and *forceWidth* define the force parameters. At the initialization function, the default of the *forceMagnitude* and the *forceWidth* is equal to zero, and the default of the *forceAngle* is equal to $\pi/2$. In the initialization function, the program checks if the *forceMagnitude* is not equal to zero, and if this is true, it calls on the function **addForce**, shown in Figure 13. As this program is made in Python, the positive y-direction is downward. Hence, the default direction of a force is straight downwards. Since there is a function **addForce**, it is possible to manually add a force by calling on this function with the node.

```

30     def addForce(self, forceMagnitude_, forceWidth_, forceAngle_=np.pi/2):
31         #adds force to node
32         if(isinstance(forceMagnitude_, int) and
33             isinstance(forceAngle_, float) and
34             isinstance(forceWidth_, int)):
35             self.forceMagnitude = forceMagnitude_
36             self.forceAngle = forceAngle_
37             self.forceWidth = forceWidth_
38             self.forceX = self.forceMagnitude*-mt.cos(forceAngle_)
39             self.forceY = self.forceMagnitude*mt.sin(forceAngle_)
40         else:
41             print("The forceMagnitude and forceWidth needs to be int," +
42                 " and the forceAngle float")

```

Figure 13 Function **addForce**

When a node is connected to other nodes in a system, it could also have a node type defined in the function **nodeType**. A node can only have a type if it is part of a system and the **ElementForce** class has been run. The input argument to this function must be an object of the type *system* to check what elements are connected to the node to define the type. The **nodeType** function checks what kind of trusses meet at the node and determines if the node is a CCC-, CCT-, or CTT-node. The *nodeType* will automatically be called by the **ElementForce** class. Thus it will not be necessary to call on this function.

4.4 Element

An element represents the trusses between two nodes, the connection between two nodes. These elements are known as the struts and ties in an STM. The callable variables in the Element class are:

- *node1*, the node at one end of the element
- *node2*, the node on the other end of the element
- *width*, the widths of the element at each end
- *isOkay*, whether the capacity of the element is sufficient when it is run through the design checks
- *forceMagnitude*, the internal force of the element

The initialization function requires two input arguments when the element class is called, see Figure 14. These arguments are *node1* and *node2*. *node1* and *node2* represent each side of the element.

```

74 class Element: #setup of element class, representing the trusses
75     def __init__(self,node1_,node2_):
76         self.node1 = node1_
77         self.node2 = node2_
78         self.width = []
79         self.isOkay=True

```

Figure 14 Initialization function for **Element** class

An element can have an internal force. The variable *forceMagnitude* represents this internal force. The internal force can only be calculated if the element is part of a system. When the system is finished, the class **ElementForce** can be called and will add the internal force of the element automatically using the *addForce* function. The class **ElementForce** will be discussed more in-depth in section 4.6.

In addition to *addForce*, there are two functions in the **Element** class. These functions are *returnOtherNode* and *addWidth*. *returnOtherNode* function takes the input argument of the type *node*, checks the element, and returns the *node* at the other end of the element. Suppose the input *node* is not part of the element. The following text will be printed to the console: "not valid node." *addWidth* will have an input argument of type number and add it to the *width* list. The *width* list represents the width of the element on each side. Calculating and adding the widths of the elements will be done by the class **Checks**, which will be discussed in section 4.7.

4.5 System

4.5.1 General

System is the class that contains all the strut, ties, and nodes. This class will represent the geometry of the entire strut and tie model. The callable variables in this class will be created in the initialization function for the **System** class, see Figure 15. The callable variables are:

- *nodes*, a list of all nodes
- *elements*, a list of all elements

```

96 class System: #setup for the STM system
97     def __init__(self):
98         self.nodes = [] #The list of all nodes in the system
99         self.elements = [] #The list of all elements in the system

```

Figure 15 Initialization function for the **System** class

There are no arguments in the initializations function in the **System** class, so to fill up the lists, the user should call on the functions to add nodes and elements in the class. It is also possible to call on the lists and use the append command. The advantage of using the functions in this class is that they will check if the object already exists and will not double store the objects.

4.5.2 Add nodes

To add nodes to the system, the user must call on the function **addNode**. The input argument must be an object of the type *node* in this function. To ensure that this *node* is not already in the *nodes* list, the function **existingNode**, see Figure 21, is called. A node with the same position should not be added to the *nodes* list. See Figure 16 for an image of the **addNode** function.

```
101     def addNode(self,node_): #adds nodes to the system
102         if(self.existingNode(node_.nodePosition) == False):
103             self.nodes.append(node_)
104         else:
105             print("This node already exist!")
```

Figure 16 Function **addNode**

4.5.3 Add elements

Elements can be added to the *elements* list by either calling on the function **addElementWithNodes**, see Figure 17, or the function **addElementWithElement**, see Figure 18. Two input arguments must be objects of type **Node** in **addElementWithNodes**. This function creates an element and then checks if this already exists in the *elements* list by using **existingElement** function, see Figure 22. The input nodes are also checked if they exist in the *nodes* list by using **existingNode**. If the nodes exist and the element does not, the element will be added to the *elements* list. By using the function **addElementWithElement**, the input argument must be a defined element. This function uses **addElementWithNodes** to do the same checks for this element.

```
107     def addElementWithNodes(self,node1_,node2_): #adds elements to the system
108         element = Element(node1_, node2_)
109         if(self.existingNode(node1_.nodePosition) == False):
110             print("This node1 does not exist. Create node first")
111         elif(self.existingNode(node2_.nodePosition) == False):
112             print("This node2 does not exist. Create node first")
113         elif(self.existingElement(element)):
114             print("This element already exist")
115         else:
116             self.elements.append(element)
```

Figure 17 Function **addElementWithNodes**

```
118     def addElementWithElement(self,element_): #adds elements to the system
119         node1 = element_.node1
120         node2 = element_.node2
121         self.addElementWithNodes(node1, node2)
```

Figure 18 Function **addElementWithElement**

4.5.4 Other functions

In addition to the mentioned functions, there are two more, ***connectedNodes***, see Figure 19, and ***elementsFromNode***, see Figure 20. The argument for ***connectedNodes*** must be of type *node*. This function will return a list with all the nodes connected to this node with elements. The argument for ***elementsFromNode*** must also be of type *node*. This function will return a list of all the elements connected to this node.

```
123     def connectedNodes(self,node_):
124         #returns which nodes are connected to another node by an element
125         connectedNodes = []
126         if(self.existingNode(node_.nodePosition) != False):
127             for element in self.elements:
128                 if(element.node1.nodePosition == node_.nodePosition):
129                     connectedNodes.append(element.node2)
130                 elif(element.node2.nodePosition == node_.nodePosition):
131                     connectedNodes.append(element.node1)
132             return connectedNodes
133         else:
134             print("This node is not part of the system")
```

Figure 19 Function ***connectedNodes***

```
136     def elementsFromNode(self,node_):
137         #returns which elements are connected to a node
138         connectedElements = []
139         if(self.existingNode(node_.nodePosition)):
140             for element in self.elements:
141                 if (node_.nodePosition == element.node1.nodePosition
142                     or node_.nodePosition == element.node2.nodePosition):
143                     connectedElements.append(element)
144             return connectedElements
145         else:
146             print("This node is not part of the system")
```

Figure 20 Function ***elementsFromNode***

```
148     def existingNode(self,nodePosition_):
149         #checks if a node exist in the system
150         for node in self.nodes:
151             if(node.nodePosition == nodePosition_):
152                 return node
153         return False
```

Figure 21 Function ***existingNode***

```

155     def existingElement(self,element_):
156         #checks if an element exist in the system
157         for element in self.elements:
158             if((element.node1.nodePosition==element_.node1.nodePosition and
159                element.node2.nodePosition==element_.node2.nodePosition) or
160                (element.node2.nodePosition==element_.node1.nodePosition and
161                 element.node1.nodePosition==element_.node2.nodePosition)):
162                 return element
163         return False

```

Figure 22 Function **existingElement**

4.6 ElementForce

4.6.1 General

The **ElementForce** class setup is different from the other classes, **Node**, **Element**, and **System**. The purpose of the **ElementForce** class is to calculate the internal forces of the elements in a system. The initialization function, shown in Figure 23, has an input argument of the type *system*, the object *system* created with the class **System**. Therefore, a system must be established before calling on the **ElementForce** class. After the initialization function is called, the elements will automatically have added the internal force into the objects **Element** in the *elements* list in the **System**. All the functions and the mathematics in this class will be discussed more in detail in this section.

```

166     class ElementForce: #calculates the forces in the trusses
167         def __init__(self,system_):
168             self.system = system_
169             self.solve()
170             self.addTypes()

```

Figure 23 Initialization function for **ElementForce** class

A truss system is not difficult to calculate by hand, but it is laborious. All that is needed is to calculate equilibrium at every node and use these results in further calculations at the other nodes. The sequential nature of these calculations offered some difficulty in the code. Firstly, one would have to locate a node with enough information to calculate the forces. Then there is a need to store these answers in a way that could easily be retrieved for calculations at the next node, which again might not have enough available information to be determined yet. This would result in an extensive, complicated code with many loops and if-statements.

To overcome this problem, the idea is to calculate everything with a set of equations. The equations are equilibrium equations for each node, in both directions (x and y), with the forces in the trusses as the unknowns. Calculating a large set of equations can be done using a matrix equation.

The steps of calculating the element forces:

1. Create the *elementMatrix*
2. Fill the *elementMatrix*
3. Create and fill the *forceMatrix*, for external forces
4. Establish and solve the equation
5. Add the result to the element objects and add type to node objects

4.6.2 Create the elementMatrix

The first step to this matrix-equation is to establish the matrix, which is done in the function ***elementMatrix***. The columns in this matrix represent the elements, so the number of columns equals the number of elements. The rows in this matrix represent the contribution from the element on the nodes in both the x- and y-direction. So, the two first rows represent the first node, where the first row is the x-direction, the second row is the y-direction, the second node is represented by rows three and four, and so on. This gives the number of rows equal two times the number of nodes. The matrix dimension then equals $2 \times \text{number of nodes} \times \text{number of elements}$. The *elementMatrix* is created in the function ***elementMatrix*** by creating a matrix consisting of zeros with the given dimension using the NumPy library.

4.6.3 Fill the elementMatrix

The next step is then to fill the matrix. The double for loop in the function ***elementMatrix*** systematically goes through the different cells of the *elementMatrix*. It calculates the desired value, which is the contribution from the element on the nodes. The first loop goes through the rows, thus the nodes in the x- and y-direction. The second loop goes through the columns, thus the elements. The first check in the first for loop establishes whether the x- or y-direction is desired in the given row before the second loop starts. The calculations will only be done if the element is connected to the node in question, or else it will remain equal to zero, as an element will not have a contribution to the node if they are not connected. If the element is connected to the node, the zero will be replaced by the returned numbers from the functions ***xDirection***, Figure 24, and ***yDirection***, Figure 25, depending on the direction. See equation (15) for the calculation of contribution in the x-direction and equation (16) for the calculation of contribution in the y-direction.

Calculating the contribution for x-direction:

$$\frac{x_2 - x_1}{L} \quad (15)$$

```
197     def xDirection (self,node1_,node2_): #calculates the contribution in x-direction
198         x1=node1_.nodeX
199         x2=node2_.nodeX
200         y1=node1_.nodeY
201         y2=node2_.nodeY
202         L = mt.sqrt((x2-x1)**2+(y2-y1)**2)
203         if L == 0: #failsafe
204             return 0
205         return (x2-x1)/L
```

Figure 24 Function **xDirection**

Calculating the contribution for y-direction:

$$\frac{y_2 - y_1}{L} \quad (16)$$

```
207     def yDirection (self,node1_,node2_): #calculates contribution i y-direction
208         x1=node1_.nodeX
209         x2=node2_.nodeX
210         y1=node1_.nodeY
211         y2=node2_.nodeY
212         L=mt.sqrt((x2-x1)**2+(y2-y1)**2)
213         if L == 0: #failsafe
214             return 0
215         return (y2-y1)/L
```

Figure 25 Function **yDirection**

The signs of the contributions are essential, so it is vital to use the correct nodes in the functions **xDirection** and **yDirection** as the arguments *node1* and *node2*. The current node should be *node1*, and the node on the other side of the element should be *node2*. All the contributions with opposite directions have different signs and give the correct results since the signs for the contributions are globally the same at all nodes.

4.6.4 Create and fill the forceMatrix

The *forceMatrix* has a dimension of one column and two times the number of nodes rows. So, each number represents the node in x-direction and y-direction, the same as the rows in *elementMatrix*. The *forceMatrix* is created by creating a matrix consisting of zeros with the given dimension using the NumPy library. The **forceMatrix** function then goes through each node and checks if it has an external load. The user should have already added the external loads into the *node* objects. If the *node* has a force, it changes the number in the *forceMatrix* by calling on the variables in the node objects *forceX* and *forceY*.

4.6.5 Establish and solve the equation

The equation to solve this matrix problem is:

$$A * x = B \tag{17}$$

In this case, for solving the matrix problem, the A-matrix in equation (17) represents the *elementMatrix*, and the B-matrix represents the *forceMatrix*. The x is the vector of the unknown internal forces of the element. The dimension of the x vector should be the same number as the number of columns in matrix A. In this case, this is equal to the number of elements. When solving this equation, the unknown x vector will give the resulting forces of the elements. The *elementMatrix* problem is solved in the **Solve** function. This **solve** function is shown in Figure 26.

```
227     def solve(self): #solves the matrix equation, and add the truss forces into
228         elementMatrix = self.elementMatrix()
229         forceMatrix = self.forceMatrix()
230         self.elementForce = la.lstsq(elementMatrix, forceMatrix , rcond=None)
231         res=False
232         for force in self.system.nodes:
233             l=len(str(int(force.forceMagnitude)))
234             if self.elementForce[1] > 10**(l-1) and l !=1:
235                 res=True
236         if res: #checks if residual is too high
237             self.system.residual="\nHigh residual in the calculation of forces
238         else:
239             self.system.residual=""
240         i = 0
241         for force in self.elementForce[0]:
242             self.system.elements[i].addForce(force)
243             i+=1
244         return self.elementForce
```

Figure 26 Function **solve**

The problem with this setup is that there are more equations than unknowns, known as an overdetermined system (akrowne, 2013b). An overdetermined system of this kind is always inconsistent (it has no solution). This is combatted with the ordinary least square method, which finds the approximate solution of an overdetermined system (akrowne, 2013a). In NumPy linear algebra library in Python, a command called `lstsq` returns the least-squares solution to the matrix problem, shown in Figure 26 in code line 230. The `lstsq` command outputs the approximate solution of the problem and some other parameters. One of these parameters is the residual, which indicates how accurate the solution is (NumPy Developers, 2022). Despite the approximate results, testing has shown that the residuals are relatively low if there is nothing wrong with the system. While if there is something wrong with the equilibrium, the residual will usually be orders of magnitude higher than the external forces. This implies that the results are accurate to a satisfactory level when the residual is “low”.

Lastly, a check of the residual has been implemented, which prints a warning if this value is significant.

4.6.6 Add the result to the element objects and add type to the node objects

The solve function is called on in the initialization function, which will be done automatically after calculating the *elementForce* matrix. The **solve** function, see Figure 26, contains a loop to go through the values in the resulting *elementForce* list and add the forces to the corresponding element objects. After the solve function is called, the **addTypes** function, see Figure 27, is called. This function loops through each node in the system and calls the function **nodeType** in the **Node** class to add the node-type to the nodes.

```
246     def addTypes(self): #adds the nodetypes to the node objects after the truss
247         for node in self.system.nodes:
248             node.nodeType(self.system)
```

Figure 27 Function **addTypes**

4.7 Checks

The **Checks** class is the class where all the design checks of the system are done. This class should be called after the **ElementForce** class has been run. The initialization of this class takes four arguments:

- *system*, should be of the type system
- *thickness*, the breadth of the concrete element
- *conClass*, the concrete class
- *c*, the distance from the concrete surface to the center of the tie

These variables are further used to calculate some inherent values needed for the design checks. The rest of the initialization function is used as a setup for the program's outputs. See Figure 28

```
251 class Checks:
252     def __init__(self,system_,thicness_,conClass_,c_):
253         #performs design checks for all nodes and elements in the system
254         self.system=system_
255         self.thickness=thicness_
256         self.conClass=conClass_*0.85/1.5
257         self.v=1-self.conClass/250
258         self.c=c_
259         self.output_strings = []
260         self.is_okay=True
261         self.check()
262         string=self.system.residual
263         self.output_strings.append(string)
264         print(string)
```

Figure 28 Initialization function for **Checks** class

The function **check** is the primary operator of the **Checks** class. The **check** function systematically goes through each node and element in the system to perform design checks.

As mentioned in 2.4.2, there are many different types of nodes and geometries these nodes can have. Because of this, many if-statements try to accommodate all the possible node types. Inside these if-statements, the geometry of the nodes is determined, with forces and sizes dimensions calculated as explained in 2.4.2.2. Sizes of the trusses are saved in their respective element objects by using the previously mentioned **addWidth** function. The calculated geometry is used as input arguments in the functions **CCC**, **CCT**, or **CTT**, depending on which type of node it is.

These functions perform the final design checks for the given node type and return if the node's capacity is okay or not. If the capacity is not okay, the functions change the node object's internal variable *isOkay* from true to false for later use, and the global variable *is_okay* to false to signify that the system fails. These functions can be seen exemplified with the function **CCC** in Figure 29.

```
485     def CCC(self, F1, F2, F3, width1, width2, width3, bredd, strength, v, num):
486         #design check of CCC node
487         sigma_1=(F1/(bredd*width1))           #stresses
488         sigma_2=(F2/(bredd*width2))
489         sigma_3=F3/(bredd*width3)
490         sigma_M=v*strength                    #Ec2. formula(6.60) for ccc-node
491         if sigma_1 > sigma_M or sigma_2 > sigma_M or sigma_3 > sigma_M:
492             string = "The capacity of node " +str(num)+" is not enough."
493             print (string)
494             self.output_strings.append(string)
495             self.is_okay=False
496             return False
497         else:
498             string = "The capacity of node " +str(num)+" is OK!"
499             print(string)
500             self.output_strings.append(string)
501             return True
```

Figure 29 Function **CCC**

After the nodes have been checked, the same is done for the trusses. All the trusses are categorized as either struts or ties. Hence, the strut function checks the trusses experiencing compressive forces, and the **tie** function designs the trusses experiencing tensile forces. The **strut** function checks if the capacity of the strut is okay using the widths previously calculated when the nodes were checked. Some of the trusses do not have two widths if one of the ends is at a smeared node. This is handled by the for loop in the function. Suppose the capacity of the strut is not okay. In that case, the element's internal variable *isOkay* is changed from true to false, and the global variable *is_okay* to false to signify that the system as a whole fails. See Figure 30. The **tie** function calculates the required reinforcement area in the tie for the calculated force.

```

534     def strut(self, element, thickness, strength, v, num):
535         #design check of strut
536         isokay=True
537         sigma_M=0.6*v*strength # Ec2. formula(6.56) for strut with transverse
538         for i in element.width: #checking both ends of the strut
539             sigma=element.forceMagnitude/(i*thickness)
540             if sigma > sigma_M:
541                 isokay=False
542
543         if isokay == False:
544             string = "The capacity of strut " +str(num)+" is not enough."
545             print (string)
546             self.output_strings.append(string)
547             self.is_okay=False
548             return False
549         else:
550             string = "The capacity of strut " +str(num)+" is OK!"
551             print(string)
552             self.output_strings.append(string)
553             return True

```

Figure 30 Function **strut**

Lastly, the class call for the function **strainEnergy**, which calculates the system's total strain energy as described in section 2.5. The **strainEnergy** is shown in Figure 31.

```

624     def strainEnergy(system): #calculates the total strain energy in the system
625         strain=0
626         for i in system.elements:
627             if i.forceMagnitude < 0:
628                 l=mt.sqrt((i.node1.nodeX-i.node2.nodeX)**2+
629                     (i.node1.nodeY-i.node2.nodeY)**2)
630                 strain+=abs(i.forceMagnitude)*l*434/200000*10**(-3)
631         return strain

```

Figure 31 Function **strainEnergy**

4.8 Plot

The **Plot** class exists to draw the system and the calculated results to have a more user-friendly output than just a bunch of numbers. The initializing of the class uses the system as the only input argument before running the internal function **draw**. The **draw** function uses the Python package matplotlib to draw the system. This includes the nodes, elements, external forces as arrows, the internal forces of the elements, and numbering the nodes and elements in a plot. This drawing is constructive to visualize the system and uncover any input mistakes. The **draw** function is also where the variable *isOkay* in the objects is used, as the function draws these objects in red if their design checks are not met. Lastly, the **draw** function returns the whole plot as a figure to be able to be drawn in the GUI program window.

4.9 Outputs

The result from the program will be printed out on the console. What will be printed:

- State of nodes
- State of struts
- Required reinforcement
- If the system is sufficient
- Strain energy (only if the capacity of the system is enough)
- Residual (only if high)

All the printed lines will happen in the **Checks** class. All the nodes and each strut and tie will have one printed line each. The nodes and strut will either be OK or not enough. The printed line for the tie is the required reinforcement for each tie. If any nodes or struts fail, the system's total capacity will not be enough. The strain energy will be printed if the system's capacity is enough. A warning will also be printed if the residual from the calculation in the class **ElementForce**.

5 User manual

5.1 General

There are several ways to create the strut and tie model using the program created for this thesis. The methods can be divided into two main groups. The first method is creating a Python file and importing all the necessary classes. Then manually calling the necessary functions and classes to set up the system and calling on the necessary calculations. This method is more open, and the user has more freedom to make it personal to different models. The other method is using the user interfaces. This method is more fixed and created for users who are not comfortable with coding in Python and can easily create simple strut and tie models. Both methods will be discussed further in later sections.

It is essential to install all the necessary packages before using the code. The code is developed in Spyder, with most of the necessary packages already downloaded and implemented into the program.

The packages used:

- NumPy
- math
- matplotlib
- Tkinter
- sys

5.2 Downloading the repository

5.2.1 General

The first step of using the repository is downloading it. The code is stored in a GitLab web repository which can be accessed from this link:

<https://gitlab.stud.idi.ntnu.no/martgrah/stm>

The repository should be downloaded as a zip or cloned locally on the computer. The easiest is downloading the zip and unpack it locally on the computer. See Figure 32 for a screenshot from the GitLab web repository. The download button and clone button are inside the dotted box. The branch should be main.

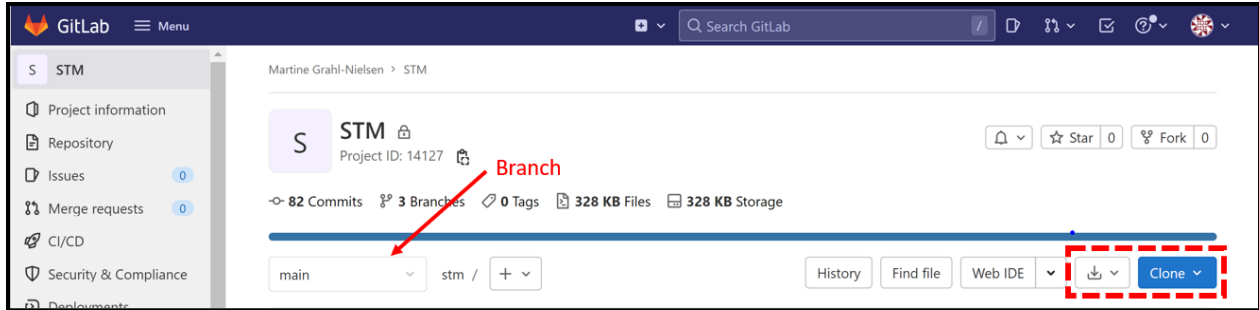


Figure 32 GitLab repository

5.2.2 Repository content

The repository contains:

- STM file
- GUI file
- models folder
- examples folders

The STM file is the main code, where all the calculations are done. The GUI file is the graphical user interface file that gives the user the program with the GUI. The models folder is where users should save their strut and tie models. Furthermore, the examples folder is where the examples are saved that are used in the examples in section 6 and the studies in section 7.

5.3 Python script

5.3.1 General

The main workflow of establishing the strut and tie model using this program can be:

1. Importing the file
2. System setup
3. Calculating the element force
4. Do the design checks
5. Plotting the results

This section shows a general guide on how the code can be used and how to set up models. The user can also use the code as a base and extend the code for more complex strut and tie models. The possibilities are many, and there is no limit to the use of the code.

The folder models are created to store the models the user creates. This folder consists of two Python files. One is called template, and the purpose of this file is to be used as a template that the user can use as a base for creating models. The other file is called user_manual, which is the model created for this user manual to show an example.

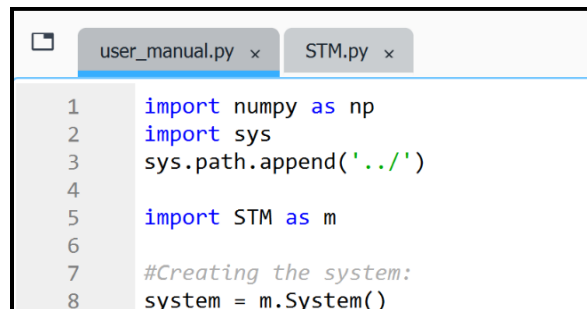
5.3.2 Importing the STM file

The first step is to import the STM file. Importing the STM file into a new Python file is the best method to create a model because it can then be saved as a single model, and the user can create several models and compare the results. Another possibility is to use the STM file and create the strut and tie model below the code. Then it is essential to create variable names so they are easily distinguished from each other.

Since the `user_manual` is not in the same folder as the STM file, we need to add this folder. We use the package `sys` and the command `sys.path.append('./')`. See code lines two and three in Figure 33 and Figure 34 for the `sys` import. If the model is created in another folder than `models`, the argument: `'./'` needs to be changed.

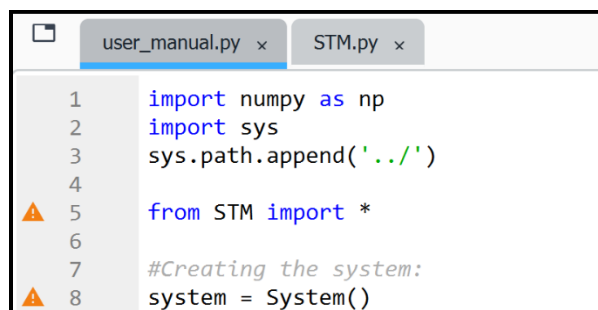
Here are two ways to import the STM file:

- `import STM as m`
- `from STM import *`



```
user_manual.py x STM.py x
1 import numpy as np
2 import sys
3 sys.path.append('./')
4
5 import STM as m
6
7 #Creating the system:
8 system = m.System()
```

Figure 33 Import STM as m



```
user_manual.py x STM.py x
1 import numpy as np
2 import sys
3 sys.path.append('./')
4
5 from STM import *
6
7 #Creating the system:
8 system = System()
```

Figure 34 Import STM as *

If the first option is used, `import STM as m`, every time a function or class from STM is called on must begin with `m`. The `m` can be changed to the user's preference. See Figure 33 for this option. This method is used in this user manual.

Using the second option, `from STM import *`, the class and function names can be called on directly. This can create a warning triangle but will not affect the code if written correctly. See Figure 34 for an image from Spyder on this importing method.

5.3.3 System setup

The system setup should define the geometry of the strut and tie model. This consist of nodes and elements. It is important to remember that the coordinate system is defined by x to the right and y downwards when using the code. One way to set up the system:

1. Create the system
2. Create the nodes
3. Apply force to nodes
4. Add the nodes
5. Create/add the elements

The first step is to create the system. This is done by calling on the **System** class and defining this object as a variable. This is done in code line 8 in Figure 33 and Figure 34.

The next step is to create the nodes. The nodes need to be separate variables since they need to be callable to add force and create the elements. The **Node** class must be called and defined as a variable. The input argument must be a two-dimensional list representing the coordinates in the x- and y-direction, both in mm. See Figure 35 for how this is done.

```
10 #Creating the nodes:  
11 node1 = m.Node([0,0])  
12 node2 = m.Node([3000,0])  
13 node3 = m.Node([1500,-1500])
```

Figure 35 Creating the nodes

After the nodes are defined, the external forces can be applied to the nodes. This is done by calling on the function **addForce** in the **Node** class. The first argument is the magnitude of the force in N. The second is the width of the force in mm, and the last is the angle in radians. The magnitude of the force should be a positive integer; thus, the force will always point toward the node. Increasing angles will rotate the force counter-clockwise with the zero-angle coming straight from the right. Figure 36 illustrates the different *forceAngle* to demonstrate the direction of the force. The angle has a default equal to $\pi/2$, representing the force downwards, as shown in code line 18 in Figure 37, and defining the angle to be equal to $3\pi/2$ the force will be upwards, as shown in code lines 16 and 17 in Figure 37.

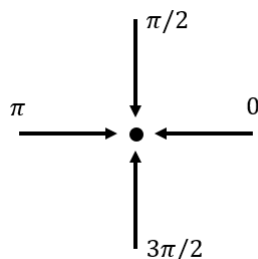


Figure 36 Illustrating the *forceAngle* on the node

```

15 #Applying force to the nodes:
16 node1.addForce(10000,400,3*np.pi/2)
17 node2.addForce(10000,400,3*np.pi/2)
18 node3.addForce(20000,400)

```

Figure 37 Applying force to the nodes

After the force is defined, the nodes must be added to the system. This is done by calling on the function **addNode** in the class **System**. See Figure 38 for how this is done.

```

20 #Adding the nodes to the system:
21 system.addNode(node1)
22 system.addNode(node2)
23 system.addNode(node3)

```

Figure 38 Adding the nodes to the system

The next step is to create/add the elements. The elements do not need to be defined as a variable. This is because the element does not need to be callable later. Therefore, there are two ways to add the element to the system:

- **addElementWithNodes**
- **addElementWithElement**

Figure 39 shows the different ways to create/add the elements. By using **addElementWithNodes**, the two input arguments must be of type *node*, representing one node on each side of the element. It is important to use predefined nodes, so the same *node* objects are in the *nodes* list in the system. By using **addElementWithElement**, the input argument must be of type *element*.

```

25 #Create/add the elements:
26 element1 = m.Element(node1, node2)
27
28 system.addElementWithElement(element1)
29 system.addElementWithElement(m.Element(node2, node3))
30 system.addElementWithNodes(node1,node3)

```

Figure 39 Create/add the elements

5.3.4 Calculating the element force

After the system is defined, the next step is to find the truss system's internal forces, the elements' internal forces. This is done by calling on the class **ElementForce**. The forces will be added to the object element when this class is called. See Figure 40 on how to call on the **ElementForce** class.

```

32 #Calculating the element force:
33 e = m.ElementForce(system)

```

Figure 40 Calling on the ElementForce

Defining **ElementForce** as a variable is advised, as it will be shown in the variable explorer in Spyder. With the variable explorer, it is possible to see the list of all the internal forces of

the elements in a list, shown in Figure 41. How to get the list: In variable explorer double click e (the variable name) ->Double click elementForce ->Double-click the first index (0). Here it is also possible to check the residual from the force calculations by double-clicking the second index (1) instead of the first.

	0
0	-10000
1	14142.1
2	14142.1

Figure 41 elementForce list

5.3.5 Do the design checks

When the system is established, and the elements' internal forces are added, the next step is to call on the **Checks** class. This class will do all the necessary design checks for the model. See how to call on the class **Checks** in Figure 42. The arguments in the **Checks** class are first the system, then the breadth of the cross-section of the concrete element in mm, then the concrete class number, then the distance from the concrete surface to the center of the tie in mm.

```
35 #Do the design checks:
36 m.Checks(system, 400, 30, 50)
```

Figure 42 Calling on the Checks

The results of the design checks will be printed out in the console field. It will print if the node capacity is sufficient or not, the required reinforcement of the ties, if the struts are sufficient, and the strain energy. See the printed lines from the user_manual example in Figure 43

```
The capacity of node 1 is OK!
The capacity of node 2 is OK!
The capacity of node 3 is OK!
The capacity of node 4 is OK!

Required reinforcement area in tie 1 is 922mm^2
The capacity of strut 2 is OK!
The capacity of strut 3 is OK!
The capacity of strut 4 is OK!

The capacity of the system is enough

The strain energy in the system is 3472 J
```

Figure 43 The printed lines to the console

5.3.6 Plotting the results

The last step is to plot the results. To do this, the **Plot** class must be called. See Figure 44 on how to call on the **Plot** class. The input argument is the system.

```
38 #Plotting the results:  
39 m.Plot(system)
```

Figure 44 Calling on the Plot

After the **Plot** class has been called, the plot will be shown as the one in Figure 45. If the resulting nodes, struts, or ties are insufficient, they will be red.

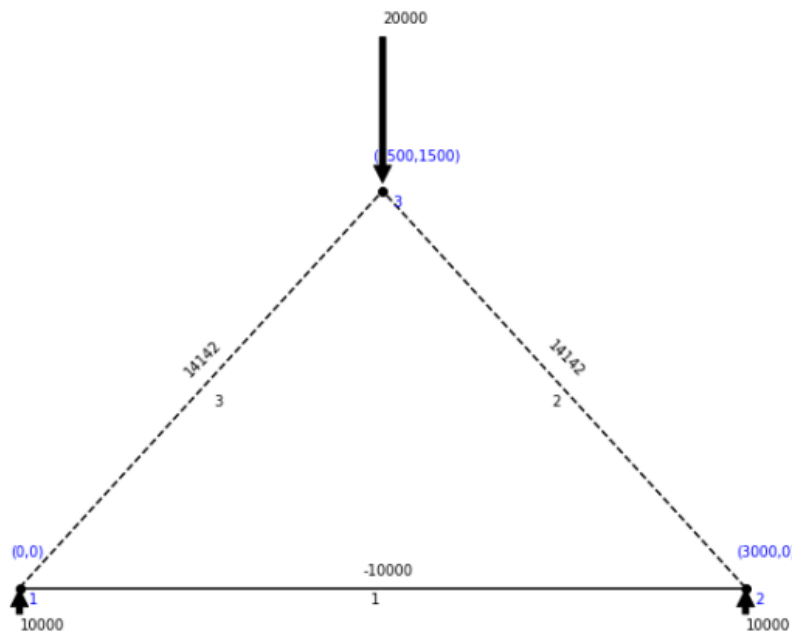


Figure 45 The Plot of the established STM

5.4 Graphical user interface

5.4.1 General

The graphical user interface has been created using the Python package Tkinter. Tkinter is a framework already built into the Python standard library (Python, 2022). The framework Tkinter consists of buttons, entry boxes, frames, among many other features. It is easy to use and works great for this purpose.

There are four separate windows in the graphical user interface (GUI). In The first three windows, the user is asked to submit some data to create the strut and tie model, and the last window will show the results.

The four windows are:

1. Number of nodes
2. System setup
3. Concrete specification
4. Results

An example is followed through the next sub-chapters to show how the GUI can be used. This is just an example, and there are endless possibilities to set up the STM.

5.4.2 Window one: Number of nodes

In window one, the user is asked to add the required number of nodes for the strut and tie model. The number of nodes is the number of connection points between strut and ties. The input must be an integer number. If not, the submit button will not activate.

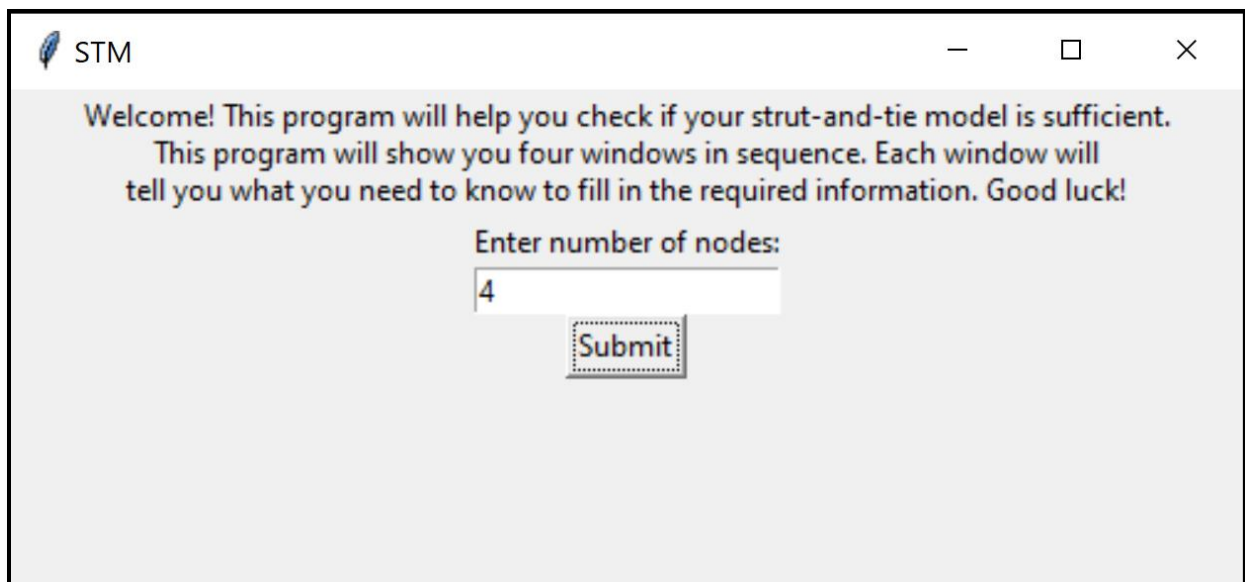


Figure 46 The first window in the GUI

5.4.3 Window two: System setup

In the second window, the user is required to fill in a lot more information. This window is built up from a grid format. The rows represent a node, and the columns can be divided into three sections. The three sections are:

- Location of node in coordinate system, both x and y
- Which node it is connected to by strut or tie
- Force parameters

It is important to notice that the coordinate system is a right-handed coordinates system, which gives positive x to the right and positive y upwards. All the input numbers should be positive or negative integer numbers.

Figure 47 illustrates how the window is built. The blue box illustrates the grid's row, which represents a node. The red box illustrates the coordinates of each node. The coordinates must be a positive or negative integer number of the node's placement in mm. The green box represents the nodes the current node is connected to and is filled with checkboxes. For example, in Figure 48, the first node is connected to nodes 2 and 3. The yellow box represents the external force on the node. The force width is in mm, and the magnitude is in newtons.

System setup

Input: Node coordinates, connected node(s), force at node.
Right handed coordinate system, positive x to the right and positive y upwards.

Nodes	X [mm]	Y [mm]	1	2	3	Force magnitude [N]	Force width [mm]
1			<input type="checkbox"/>	<input type="checkbox"/>			
2				<input type="checkbox"/>			
3							

Submit

Figure 47 Illustration of how the System setup window is built

The GUI checks when the user tries to press the button to submit to see if the user has filled in all the required information. The checks done are as follows:

- All coordinates are filled
- The force magnitude is filled correctly
- The force width is filled if the magnitude is applied
- The model is in vertical equilibrium

If the coordinates are not filled correctly with an integer number, this line will be printed to the window:

"All positions need to be filled with integer number"

If the force magnitude is filled incorrectly, this line will be printed to the window:

"The magnitude must be filled with integer number."

If the force width is not filled when the force magnitude is filled correctly, this line will be printed to the window:

"Force width must be filled with integer number when force magnitude is applied."

To check if the element is in vertical equilibrium, it takes all the forces and checks if that is equal to zero. If this is not true, this line will be printed on the window:

"The structure needs to be at vertical equilibrium"

If there are more than one error in the input information, it does not always give all the error output lines. First, fix the error given and then try to submit again and fix the next one until it goes through. There are no checks to see if the user has checked any of the boxes to connect them.

Nodes	X [mm]	Y [mm]	1	2	3	4	Force magnitude [N]	Force width [mm]
1	-2000	0	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	400000	400
2	2000	0	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	400000	400
3	-1000	1000	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	-400000	400
4	1000	1000	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	-400000	400

Figure 48 The second window in the GUI

5.4.4 Window three: Concrete specification

In this window, the user is required to submit three inputs to specify the concrete element. The required inputs are:

- Concrete class
- Breadth of the concrete element
- Distance from concrete surface to center of tie

All the inputs should be an integer number. Nothing will happen when the submit button is pressed if this is not true. The concrete class represents the classification of the concrete. The breadth of the concrete element is the breadth of the cross-section of the element. The last one is the distance from the surface to the center of the tie.

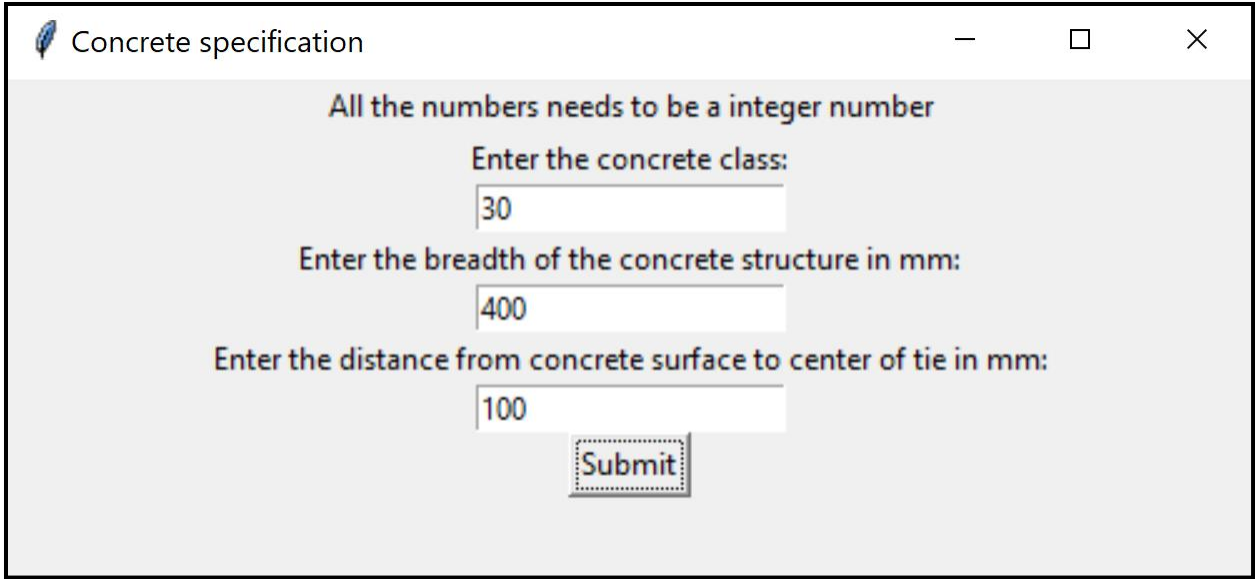


Figure 49 The third window in the GUI

5.4.5 Window four: Results

The fourth and last window will show the results of the STM. This window is divided into two, where the left part is the text to inform if the nodes, struts, ties, and the system in total are sufficient. In this section, the strain energy will also be given, and if there is a high residual from calculating the internal force, this will also be printed. The STM will be plotted in the right section of the fourth window. If there are parts of the STM that is not sufficient, it will be written in the left part and marked with red in the right part.

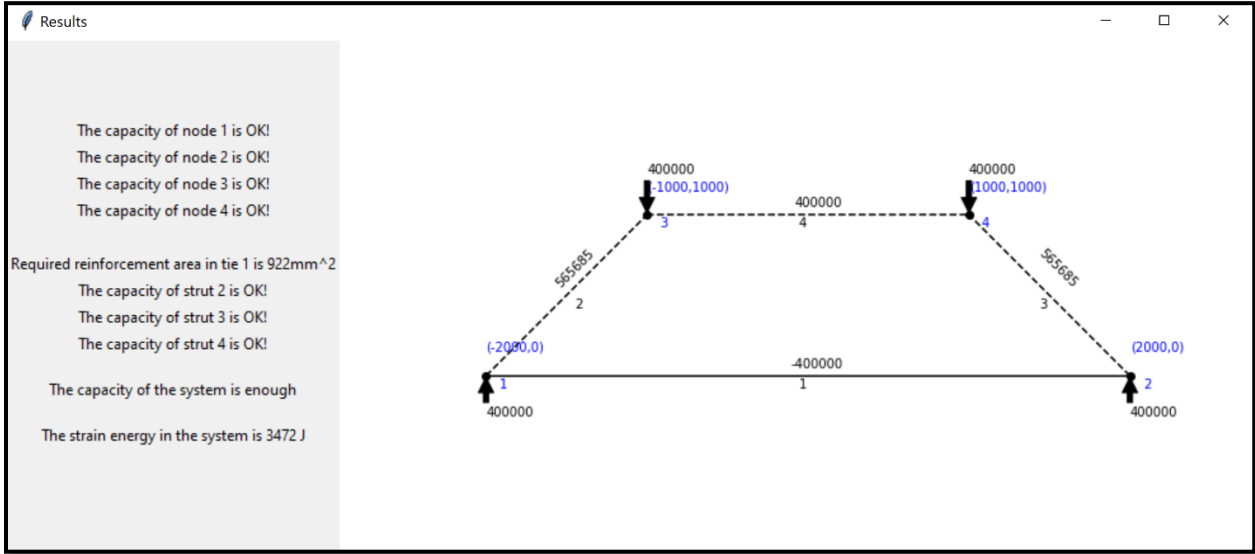


Figure 50 The fourth window in the GUI

6 Examples

6.1 General

To demonstrate that the program works as intended, three examples of strut and tie models will be established in this section. These STM examples are found online, with hand calculations to compare with the result from the program.

6.2 Example 1: two-pile cap

6.2.1 General

This example is a two-pile cap, see Figure 51 (Goodchild et al., 2015). The two-pile cap example should carry a load of 2500 kN on a 500 mm square column. The diameter of the pile caps is 600 mm in diameter.

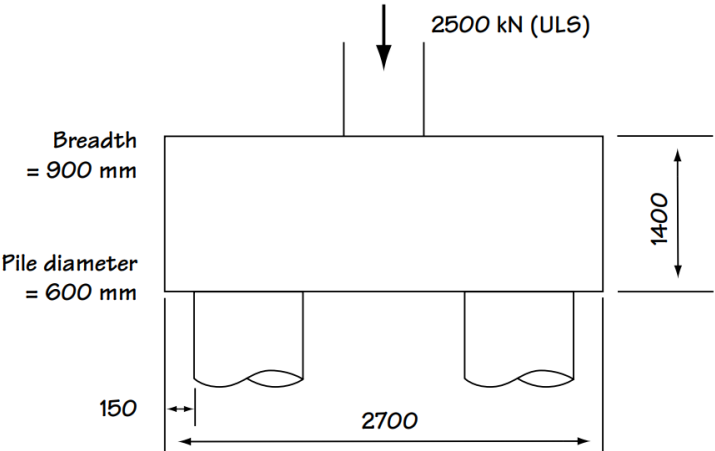


Figure 51 Two-pile cap (Goodchild et al., 2015)

6.2.2 Model setup

The model setup is shown in Figure 52. The program does not have an option to have a circular bearing plate, so to fix this, the supports' bearing plate is changed to $300^2 * \frac{\pi}{900} = 314$.

The following information is known to set up the STM:

- Coordinates of node, Figure 52
- Point load, 2 500 kN
 - Support forces: 1 250 kN each cap
- Breadth, 900 mm
- Width of the bearing plates, $b_1 = 500$ mm and $b_2 = b_3 = 314$ mm
- The concrete class, $f_{ck} = 30$ MPa
- Distance to center of tie, $c = 50$ mm

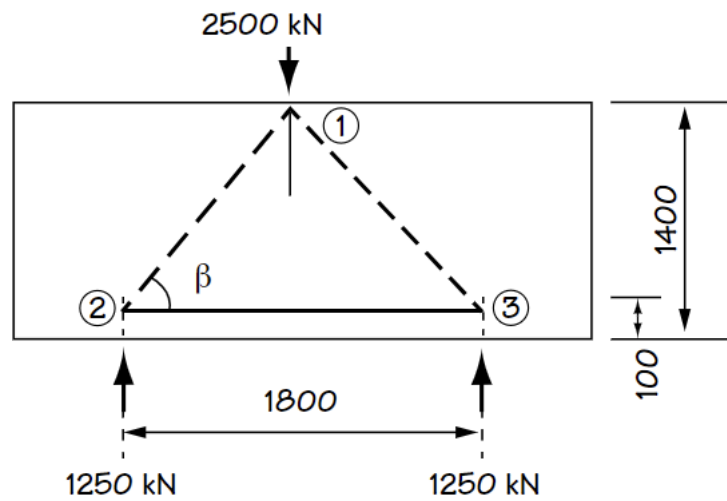


Figure 52 Geometry of STM for example 1 (Goodchild et al., 2015)

```

1  from STM import*
2  import numpy as np
3  system=System()
4
5  node1=Node([0,0])
6  node2=Node([-900,1300])
7  node3=Node([900,1300])
8
9  system.addNode(node1)
10 system.addNode(node2)
11 system.addNode(node3)
12
13 system.addElementWithNodes(node2, node1)
14 system.addElementWithNodes(node1, node3)
15 system.addElementWithNodes(node2, node3)
16
17 node2.addForce(1250000, 314, 3*np.pi/2)
18 node3.addForce(1250000, 314, 3*np.pi/2)
19 node1.addForce(2500000, 500)
20
21 ElementForce(system)
22 Checks(system, 900, 30, 50)
23 Plot(system)
24

```

Figure 53 STM code setup for example 1

6.2.3 Results

The output result lines from the program are given in Figure 54, and the plotted result from the program is given in Figure 55. This example will compare the results of the elements and the required reinforcement.

```

The capacity of node 1 is OK!
The capacity of node 2 is OK!
The capacity of node 3 is OK!

The capacity of strut 1 is OK!
The capacity of strut 2 is OK!
Required reinforcement area in tie 3 is 1994mm^2

The capacity of the system is enough

The strain energy in the system is 3380 J

```

Figure 54 Printed output of example 1

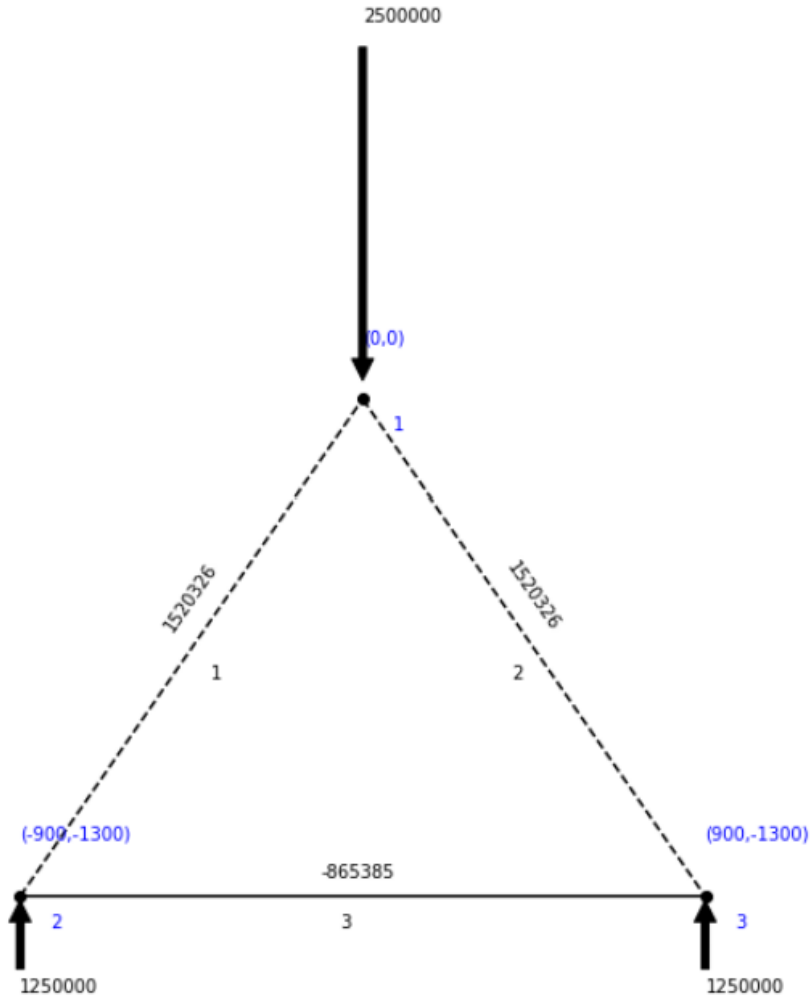


Figure 55 Plotted STM of example 1

From the output of the program and the hand calculations shown in Table 1, it is evident that the results are the same. The forces in the trusses have been calculated to a satisfactory level. The required reinforcement is about the same, and all the design checks of the system are suitable for both cases.

Element	Program		Example		Difference
	Force	Status	Force	Status	
1 Strut	1520 kN	Yes	1520 kN	Yes	0 kN
2 Tie	1520 kN	Yes	1520 kN	Yes	0 kN
3 Strut	-865 kN	Yes	-866 kN	Yes	1 kN
Reinforcement	1996 mm ²		1991 mm ²		5 mm ²

Table 1 Comparing internal forces and required reinforcement of elements in example 1

6.3 Example 2: deep beam

6.3.1 General

Example 2 is a deep beam with a uniformly distributed load, shown in Figure 56. For STM design, calculate distributed load as two point loads equal to 810 kN each. The size of the point loads is assumed to be the same at the supports.

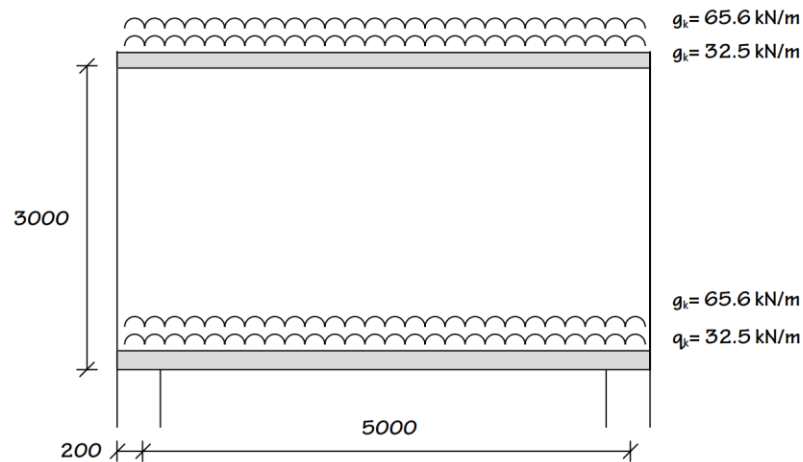


Figure 56 Example 2: deep beam (Goodchild et al., 2015)

6.3.2 Model setup

The setup of the STM is shown in Figure 57. The program cannot handle struts from the top down to the nodes, so the forces are instead placed directly on the nodes.

The following information is known to set up the STM:

- Coordinates of node, Figure 57
- Two point loads, 810 kN each
 - Support forces: 810 kN each
- Breadth, 250 mm
- Width of the bearing plates, 376 mm (all)
- The concrete class, $f_{ck} = 25$
- Assuming distance from the surface to tie,
- $c = 180 \text{ mm}$

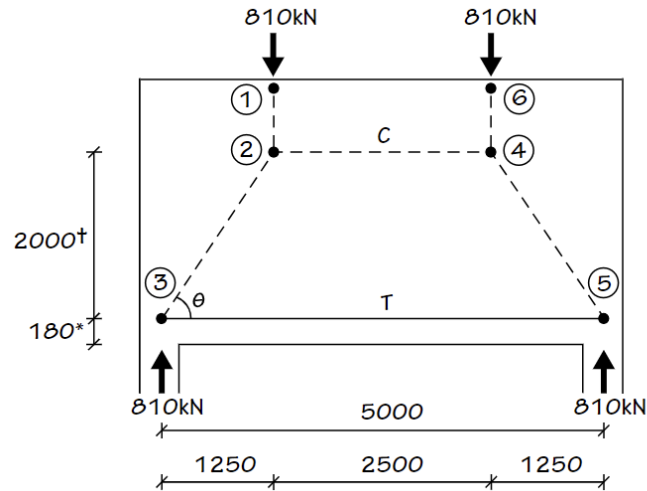


Figure 57 Geometry of STM for example 2 (Goodchild et al., 2015)

```

1  import sys
2  sys.path.append('../')
3
4  from STM import*
5  import numpy as np
6  system=System()
7
8  node1=Node([0,0])
9  node2=Node([1250,-2000])
10 node3=Node([5000,0])
11 node4=Node([3750,-2000])
12
13
14 system.addNode(node1)
15 system.addNode(node2)
16 system.addNode(node3)
17 system.addNode(node4)
18
19 system.addElementWithNodes(node1, node2)
20 system.addElementWithNodes(node1, node3)
21 system.addElementWithNodes(node2, node4)
22 system.addElementWithNodes(node3, node4)
23
24 node1.addForce(810000, 376, 3*np.pi/2)
25 node2.addForce(810000, 376)
26 node3.addForce(810000, 376, 3*np.pi/2)
27 node4.addForce(810000, 376)
28
29 e=ElementForce(system)
30 Checks(system, 250, 25, 180)
31 Plot(system)

```

Figure 58 STM code setup for example 2

6.3.3 Results

The output result lines from the program are given in

Figure 59, and the plotted result from the program is given in Figure 60. This example will compare the results of the elements and the required reinforcement.

```
The capacity of node 1 is OK!  
The capacity of node 2 is OK!  
The capacity of node 3 is OK!  
The capacity of node 4 is OK!  
  
The capacity of strut 1 is OK!  
Required reinforcement area in tie 2 is 1167mm^2  
The capacity of strut 3 is OK!  
The capacity of strut 4 is OK!  
  
The capacity of the system is enough  
The strain energy in the system is 5492 J
```

Figure 59 Printed output lines for example 2

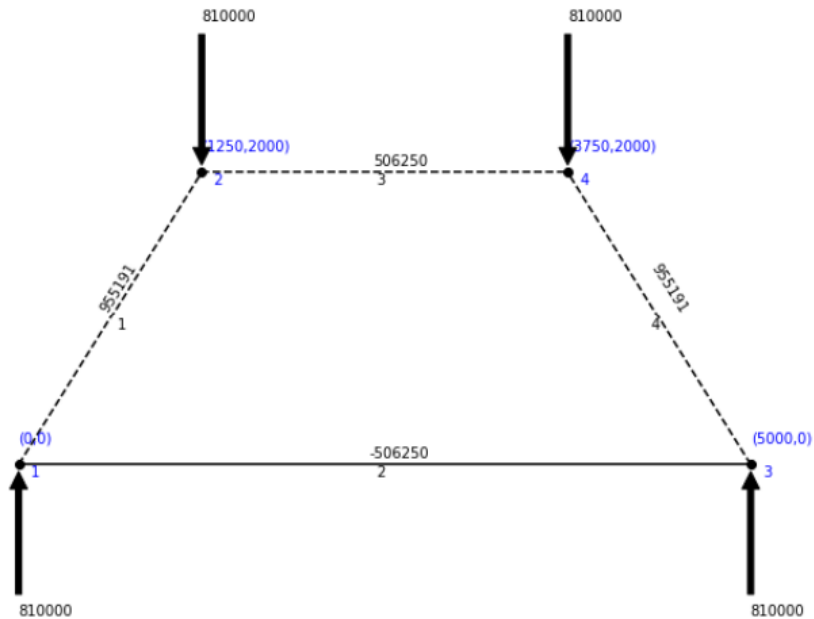


Figure 60 Plotted result for example 2

It is evident that the program does the same as the hand calculations, as shown in Table 2. The internal forces calculated in the program are the same as the calculated internal forces in the example. The required reinforcement differs only with 3 mm².

Element	Program		Example		Difference
		Okay		Okay	
1 Strut	955 kN	Yes	955 kN	Yes	0 kN
2 Tie	-506 kN	Yes	-506 kN	Yes	0 kN
3 Strut	506 kN	Yes	506 kN	Yes	0 kN
4 Strut	955 kN	Yes	955 kN	Yes	0 kN
Reinforcement	1167 mm ²		1164 mm ²		3 mm ²

Table 2 Comparing results for example 2

6.4 Example 3: Deep beam with large openings and recess

6.4.1 General

This example is a deep beam with a large opening and recess (Ghoraba et al., 2020). The geometry of the beam is illustrated in Figure 61. The beam is designed to carry a nominal load of 2667 kN.

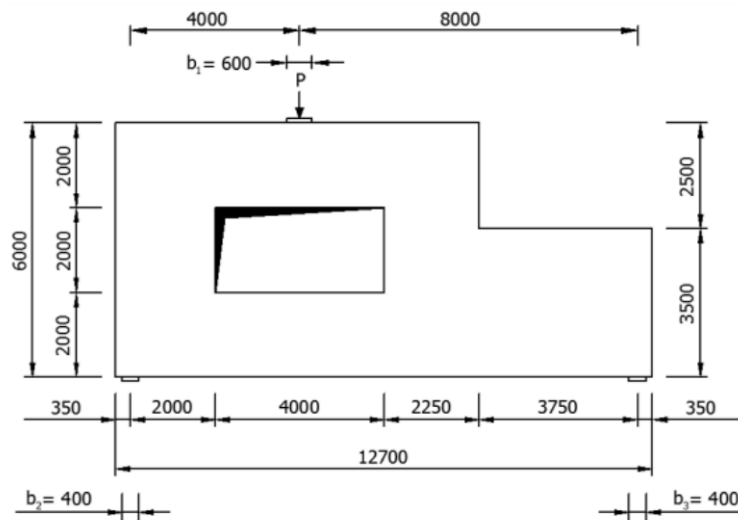


Figure 61 Geometry of deep beam with large opening and recess (Ghoraba et al., 2020)

6.4.2 Model setup

The geometry of the STM is given in Figure 62. The model can be established using the program with all the given information. The given information:

- Coordinates of node, Figure 62
- Nominal load, 2 667 000 N
 - Support forces: $V_1 = 1\,706\,737\text{ N}$ and $V_2 = 853\,262\text{ N}$
- Breadth, 305 mm
- Width of the bearing plates, $b_1 = 600\text{ mm}$ and $b_2 = b_3 = 400\text{ mm}$
- The concrete class, 55
- $c = 200$

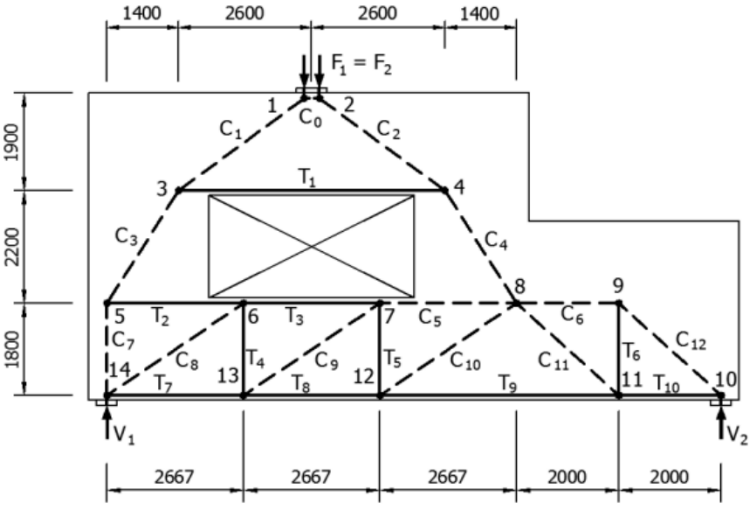


Figure 62 Geometry of the STM for example 3 (Ghoraba et al., 2020)

```

1  from STM import *
2  from numpy import pi
3
4  system = System()
5
6  node1 = Node([4000,0],2560000,600)
7  node2 = Node([1400,1900])
8  node3 = Node([6600,1900])
9  node4 = Node([0,4100])
10 node5 = Node([2667,4100])
11 node6 = Node([5334,4100])
12 node7 = Node([8001,4100])
13 node8 = Node([10001,4100])
14 node9 = Node([12001,5900],853262,400,3*np.pi/2)
15 node10 = Node([10001,5900])
16 node11 = Node([5334,5900])
17 node12 = Node([2667,5900])
18 node13 = Node([0,5900],1706737,400,3*np.pi/2)
19
20 system.addNode(node1)
21 system.addNode(node2)
22 system.addNode(node3)
23 system.addNode(node4)
24 system.addNode(node5)
25 system.addNode(node6)
26 system.addNode(node7)
27 system.addNode(node8)
28 system.addNode(node9)
29 system.addNode(node10)
30 system.addNode(node11)
31 system.addNode(node12)
32 system.addNode(node13)
33
34 system.addElementWithNodes(node2, node1)
35 system.addElementWithNodes(node1, node3)
36 system.addElementWithNodes(node3, node2)
37 system.addElementWithNodes(node2, node4)
38 system.addElementWithNodes(node3, node7)
39 system.addElementWithNodes(node4, node5)
40 system.addElementWithNodes(node5, node6)
41 system.addElementWithNodes(node6, node7)
42 system.addElementWithNodes(node7, node8)
43 system.addElementWithNodes(node8, node9)
44 system.addElementWithNodes(node9, node10)
45 system.addElementWithNodes(node10, node11)
46 system.addElementWithNodes(node12, node13)
47 system.addElementWithNodes(node13, node5)
48 system.addElementWithNodes(node12, node6)
49 system.addElementWithNodes(node11, node7)
50 system.addElementWithNodes(node7, node10)
51 system.addElementWithNodes(node13, node4)
52 system.addElementWithNodes(node12, node5)
53 system.addElementWithNodes(node11, node6)
54 system.addElementWithNodes(node10, node8)
55 system.addElementWithNodes(node11, node12)
56
57 e=ElementForce(system)
58
59 Checks(system, 305, 55, 200)
60
61 Plot(system)

```

Figure 63 STM code setup for example 3

6.4.3 Results

The resulting plot from the program is in Figure 65, and the resulting plot from the example is in Figure 66. The printed output lines from the program are in Figure 64.

```
The capacity of node 1 is OK!
Node 2 is a smeared node, no check needed
Node 3 is a smeared node, no check needed
Node 4 is a smeared node, no check needed
The capacity of node 5 is OK!
The capacity of node 6 is OK!
The capacity of node 6 is OK!
Node 7 is a smeared node, no check needed
The capacity of node 8 is OK!
The capacity of node 9 is OK!
The capacity of node 10 is OK!
The capacity of node 11 is OK!
The capacity of node 12 is OK!
The capacity of node 13 is OK!
The capacity of node 13 is OK!

The capacity of strut 1 is not enough.
The capacity of strut 2 is not enough.
Required reinforcement area in tie 3 is 2159mm^2
The capacity of strut 4 is OK!
The capacity of strut 5 is OK!
Required reinforcement area in tie 6 is 1878mm^2
Required reinforcement area in tie 7 is 421mm^2
The capacity of strut 8 is OK!
The capacity of strut 9 is OK!
The capacity of strut 10 is OK!
Required reinforcement area in tie 11 is 2185mm^2
Required reinforcement area in tie 12 is 4370mm^2
Required reinforcement area in tie 13 is 1457mm^2
The capacity of strut 14 is OK!
The capacity of strut 15 is OK!
The capacity of strut 16 is OK!
The capacity of strut 17 is OK!
The capacity of strut 18 is OK!
Required reinforcement area in tie 19 is 983mm^2
Required reinforcement area in tie 20 is 983mm^2
Required reinforcement area in tie 21 is 1967mm^2
Required reinforcement area in tie 22 is 2913mm^2

The capacity of the system is NOT enough!!!

High residual in the calculation of forces, please check equilibrium!!
```

Figure 64 Printed output for example 3

Element	Program		Example		[kN]
	[kN]		Okey		
1 Strut	2169	No	2167	Okay	Difference
2 Strut	2169	No	2167	No	2
3 Tie	-937	Yes	-951	No	2
4 Strut	1517	Yes	1512	Yes	14
5 Strut	1517	Yes	1512	Yes	5
6 Tie	-815	Yes	-813	Yes	5
7 Tie	-183	Yes	-193	Yes	-2
8 Strut	449	Yes	238	Yes	10
9 Strut	948	Yes	238	Yes	211
10 Strut	1276	Yes	1273	Yes	710
11 Tie	-948	Yes	-950	Yes	3
12 Tie	-1896	Yes	-1893	Yes	2
13 Tie	-632	Yes	-632	Yes	-3
14 Strut	762	Yes	762	Yes	0
15 Strut	762	Yes	762	Yes	0
16 Strut	762	Yes	762	Yes	0
17 Strut	1276	Yes	1273	Yes	0
18 Strut	1280	Yes	1278	Yes	3
19 Tie	-427	Yes	-426	Yes	2
20 Tie	-427	Yes	426	Yes	-1
21 Tie	-853	Yes	853	Yes	-1
22 Tie	-1264	Yes	1276	Yes	0

Table 3 Comparing results for example 3

7 Study

7.1 General

There will be done two studies in this section. These studies are created to test the capabilities and value of the program developed in this thesis. Both studies are based on calculated examples (Goodchild et al., 2015). The goal is to optimize the concrete structures by changing the geometry of the strut and tie model and changing the structure's height. This study section is a separate segment from the rest of the report and will be followed by a discussion and conclusion.

As mentioned in section 2.5, the "best" strut and tie model is the model that minimizes the total strain energy in the STM. The reinforcement is what defines the strain. The program calculates the reinforcement area needed in a tie using the steel yield strength equation (11). The strain energy is calculated for the minimum amount of reinforcement in any given tie using equation (14). Models with nodes along the "main" tie can lead to changes in the reinforcement areas. Changing reinforcement areas is technically possible, but the practice is to use the same reinforcement on the whole tie, which would give a different result to the total strain energy in the STM. So, this study is based on the technical solutions, with the possibility of changing reinforcement areas, and not necessarily on the most practical solutions.

The studies are based on existing calculated problems. Four different STMs with a changing number of vertical ties will be run through the program with seven different concrete structure heights based on the calculated problems. These four STMs will be referred to as models. Both studies will be run two times. In the first run, the concrete specification will be unrealistically high, so the strain energy of the model will be given. The concrete specifications must be high since the program only gives output for the strain energy when the STM does not fail. The results will indicate how the "best" STM might look, but when more realistic values for the concrete are added in the second run, other factors may result in failure.

7.2 Study one: Symmetrical deep beam

7.2.1 General

Study one uses a slightly modified problem from Example 1: two-pile cap (Goodchild et al., 2015). The concrete structure is shown in Figure 67.

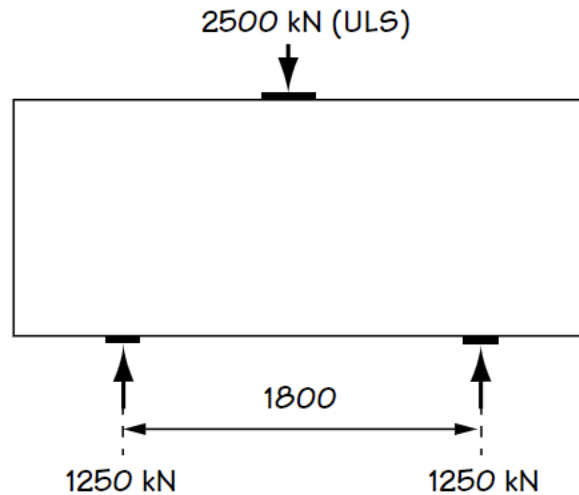


Figure 67 Geometry of problem in study one. Modified figure from (Goodchild et al., 2015)

The fixed parameters in the study are as follows:

- Point load, 2 500 kN
 - Support forces: 1 250 kN each support
- Breadth, 900 mm
- Width of the bearing plates, $b_p = 500$ mm and $b_{s1} = b_{s2} = 320$ mm
- The concrete class, $f_{ck} = 30$ MPa
- $c = 100$ mm
- Distance between supports: 1800 mm

The main difference between the models is the number of vertical ties used. The four models to test are: no vertical ties, two vertical ties, four vertical ties, and six vertical ties. The spacing of the vertical ties is uniform, dividing the half elements into equally large parts. The four models are presented in Figure 68.

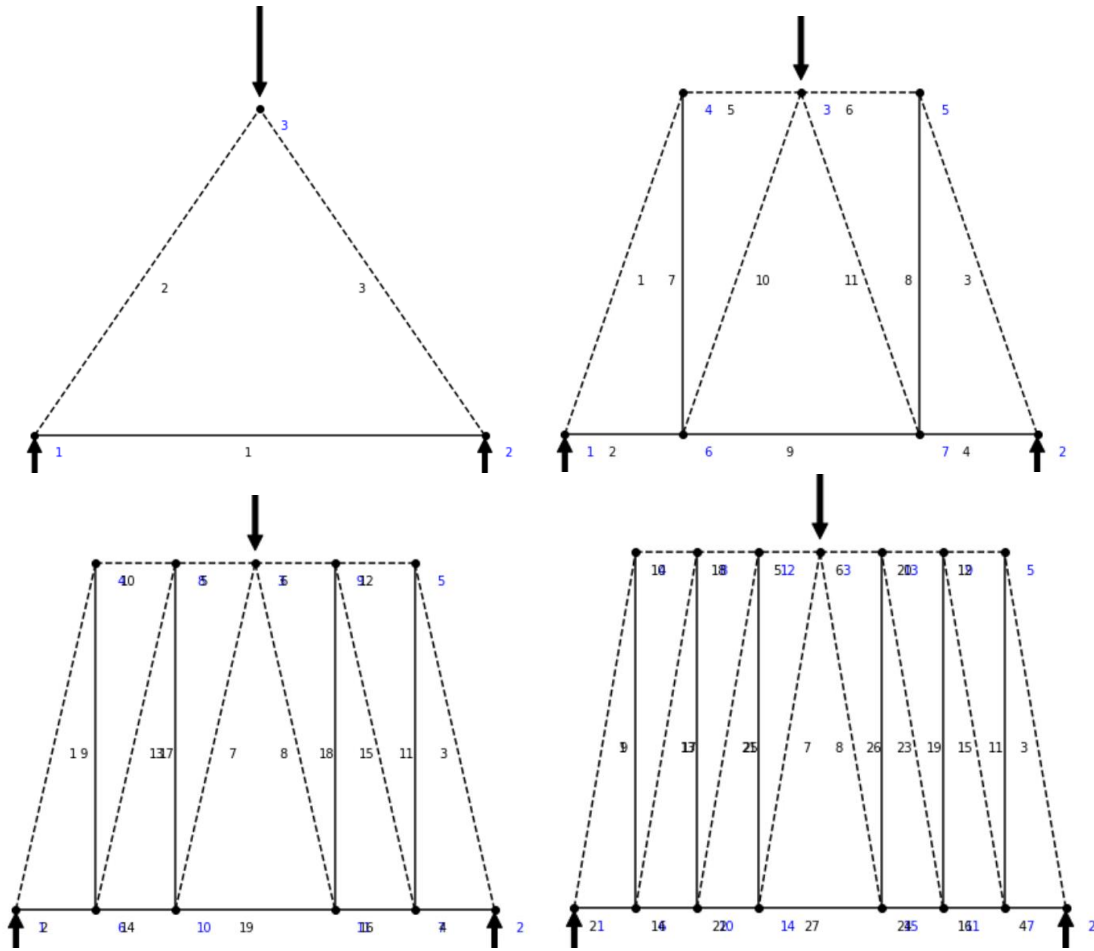


Figure 68 The four different models. Drawn by the program

7.2.2 First run

As mentioned, the first run will have unrealistically high concrete specifications to get the strain energy for all cases. The strain energy from all cases of each model can be seen in Table 4.

	300	500	700	900	1100	1300	1500
0	14647	8788	6277	4882	3994	3380	2929
2	12613	9303	8505	8544	8963	9587	10334
4	13020	11284	11780	13019	14598	16358	18228
6	14037	13630	15315	17669	20399	23270	26243

Table 4 The strain energy in J from the first run in study one, rows are the number of vertical ties, and columns are height in mm

7.2.3 Second run

Now the same models are rerun through the program, only this time with the correct concrete specifications given in the list in section 7.2.1. Table 5 shows the resulting strain energies with the correct concrete specifications. The cases where the models fail the checks are left blank, and only those who pass the checks have given strain energy in Table 5.

	300	500	700	900	1100	1300	1500
0			6277	4882	3994	3380	2929
2		9303	8505	8544	8963	9587	10334
4		11284	11780	13019			
6		13630	15315				

Table 5 The stain energy in J from the second run in study one, rows are the number of vertical ties, and columns are height in mm

7.2.4 Results

The results from Table 4 and Table 5 are represented in Figure 69. Each line represents each model, with a different number of vertical ties. The transparent line is the idealized strain energy from the first run. Where the models do not fail on the second run, the colors are opaque.

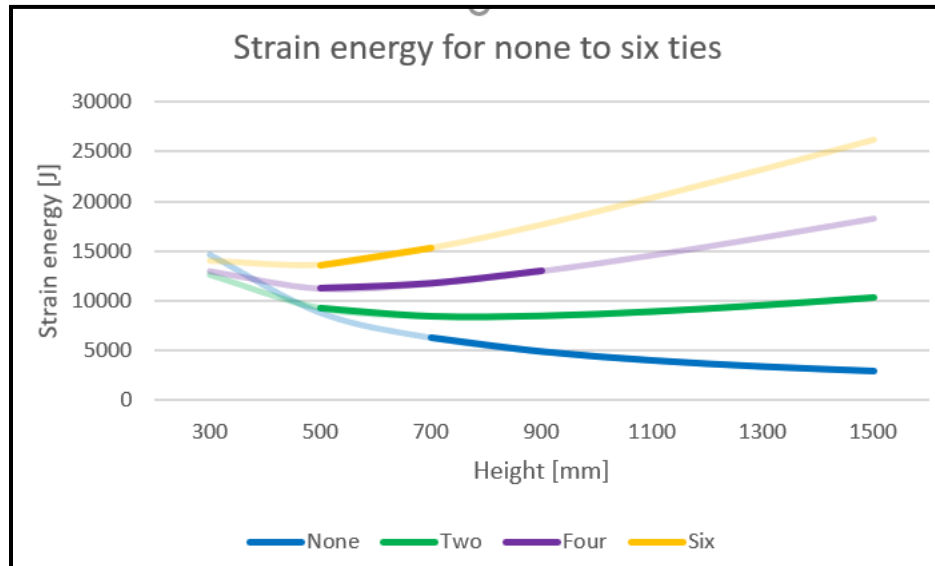


Figure 69 Plotted data from study one

Figure 69 illustrates that STMs with fewer vertical ties lead to lower strain energy. The model with no vertical ties has the lowest strain energy, except for the cases with a low height. At these heights, the rules for the construction of STM are not met, as the angle between the strut and the tie is too small.

Further, the figures and data show that the models with vertical ties have a specific height where the strain energy is minimized. This height becomes smaller as the number of vertical ties increases. However, the model without vertical ties never reaches a minimum strain energy. The strain energy of the model with no vertical ties converges towards zero as the height increase.

With the given parameters for the element, it is clear that the original model with no vertical ties is the optimal solution as long as the height is above 700 mm. If the model's height is lower, vertical reinforcement would be needed even though none of the models manage a height lower than 500 mm.

7.3 Study two: Deep beam with a non-centered point load

7.3.1 General

This problem is a modified problem from (Goodchild et al., 2015). This study case is not too dissimilar from the first study. The main difference is the non-symmetric nature of the STM, the scale of the element, as well as some concrete parameters. The concrete structure is shown in Figure 70.

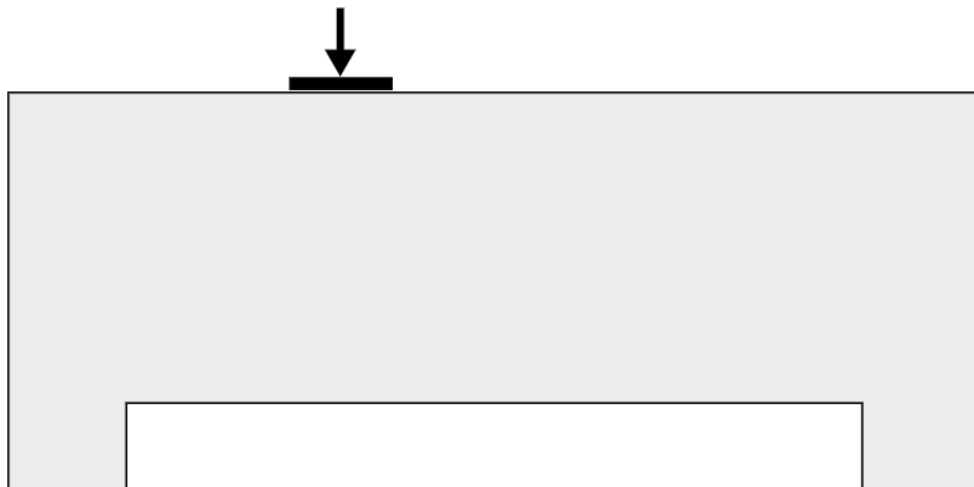


Figure 70 Geometry of problem in study two. Modified figure from (Goodchild et al., 2015)

The fixed parameters are as follows:

- Point load, 2 529 kN
- Distance to point load from support: 1250 mm
 - Support forces: $V_1 = 1\,794$ kN and $V_2 = 735$ kN
- Breadth, 500 mm
- Width of the bearing plates, $b_p = 500$ mm and $b_{s1} = b_{s2} = 500$ mm
- The concrete class, $f_{ck} = 35$ MPa
- $c = 100$ mm
- Distance between supports: 4300 mm

The main difference between the models is the number of vertical ties used. Since the point load is not centered on the structure and the STM is non-symmetric, the number of vertical ties is placed on the larger side of the point load. The largest side of the point load is divided into equally large sections. The four STMs to test are no vertical ties, one vertical, two vertical, and three vertical ties. The four models are presented in Figure 71.

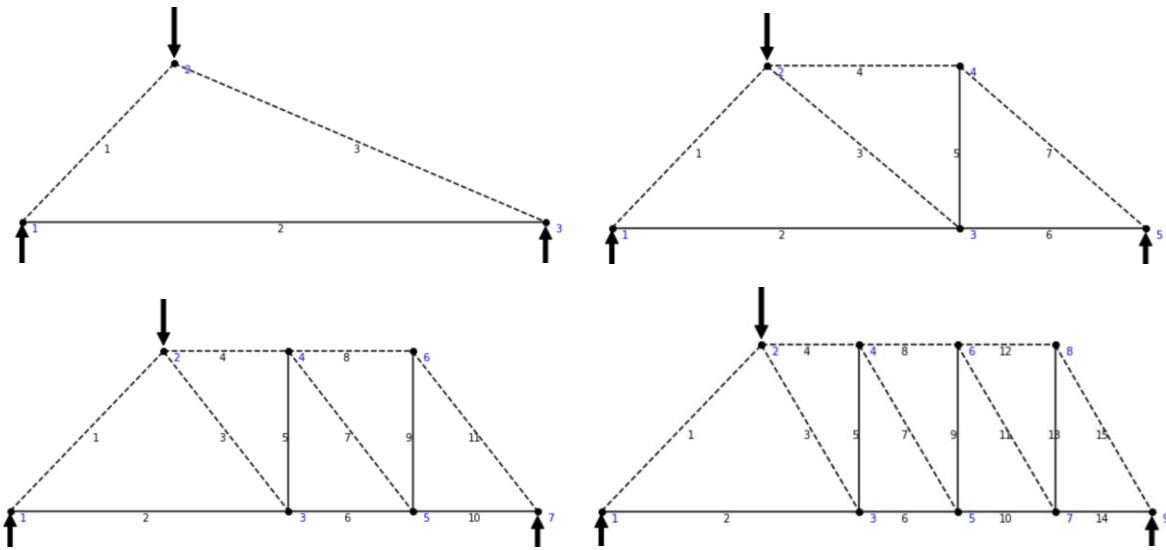


Figure 71 The four different models. Drawn by the program

7.3.2 First run

As mentioned, the first run will have unrealistically high concrete specifications to get the strain energy for all cases. The results can be found in Table 6 below.

	900	1300	1700	2100	2500	2900	3300
None	23247	16094	12307	9963	8369	7214	6340
One	20561	15315	12837	11546	10873	10561	10480
Two	20622	16437	14821	14307	14366	14760	15369
Three	21371	18035	17169	17363	18106	19173	20445

Table 6 The strain energy in J from the first run in study two, rows are the number of vertical ties, and columns are height in mm

7.3.3 Second run

Now the same models are rerun through the program, only this time with the correct concrete specifications given in the list in 7.3.1. Table 7 shows the resulting strain energies with the correct concrete specifications. The models that fail the checks are removed, and only those who pass the checks are left.

	900	1300	1700	2100	2500	2900	3300
None				9963	8369	7214	6340
One			12837	11546	10873	10561	10480
Two			14821	14307	14366	14760	15369
Three			17169	17363	18106	19173	20445

Table 7 The stain energy in J from the second run in study two, rows are the number of vertical ties, and columns are height in mm

7.3.4 Results

The results from Table 6 and Table 7 are represented in Figure 72. Each line represents each model, with a different number of vertical ties. The transparent line is the idealized strain energy from the first run. Where the models do not fail on the second run, the colors are opaque.

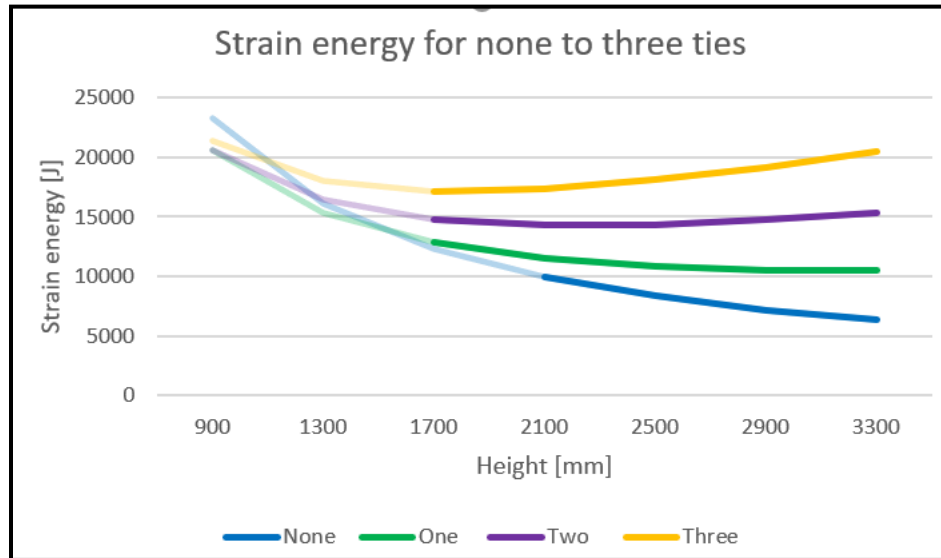


Figure 72 Plotted data from study two

The data shown in Figure 72 indicates that the models with less vertical ties have lower strain energy, except for cases with lower heights. At these heights, the model without vertical ties begins to experience more strain energy than the other models. However, these situations would not occur, as the general rules for angles between strut and tie will not be satisfied, and the model will fail.

In this study, the figures and data show that the models with vertical ties have a given height where the minimum strain energy occurs. With an increasing number of vertical ties, the height where the minimum strain occurs decreases. The strain energy goes toward zero as the height increase in the model with no vertical ties.

As for the element with the correct concrete parameters, the model without vertical reinforcement is the best solution, as long as the height is 2100 mm or more. Any lower than this, and there is a need for more ties. However, below 1700 mm, none of the models pass the design checks.

7.4 Discussion of the studies

The precalculated problem from the first study has a height of 1300 mm with no vertical ties. Choosing the model with no vertical reinforcement is also the best option from the study, with a height of 1300 mm. So, in the first study, the program agrees with the precalculated example.

The precalculated example from the second study has a height of 1300 mm and two vertical ties. All the models with a height of 1300 mm fail in the program. The failure is due to the slight modifications done to the original problem. However, even though the slightly modified version of the problem fails in this test, the strain energy calculations disagree with the choice of STM by the example, as the strain energy for one vertical tie is less than for two vertical ties, given that the models do not fail.

As for a comparison of the two studies, some patterns emerge. The “best” models, in general, have fewer vertical ties, as this is the case for both studies. This implies that if there is even a use for vertical ties to get a valid STM, the fewer, the better.

The data for both tests also show that the models with vertical ties have an optimum height for minimizing strain energy. Except for the models without vertical ties. This is because the higher the model becomes, the less force is needed to be carried by the tie, hence a need for less reinforcement and, therefore, less strain energy. It is clear that the more vertical ties are involved, the lower this optimum height becomes. Given that the two deep beams have different sizes, the easiest way to compare them is to use a height-to-length ratio. These ratios for study one are 0.44, 0.28, and 0.22 for two, four, and six vertical ties, respectively. Likewise, the ratios for study two are 0.77, 0.51, and 0.42 for one, two, and three vertical ties, respectively. From these ratios, it is evident that the height to length ratio for the optimum height reduces significantly as the number of ties goes up. It is also clear by the only direct comparison here that the nature of a tie on each side of the force gives a much smaller optimum height to length ratio than on just the one, as $0.44 < 0.77$.

7.5 Conclusion of the studies

The two studies show that the best STMs are the ones with the least number of vertical ties for all height to length ratios, except for low ones where rules for the angle between struts and ties would not be met. It is also clear that a given number of vertical ties have an optimum height-to-length ratio to get the least strain energy. The more vertical ties, the lower this ratio is for the minimum strain energy in the STM. However, inside realistic ratios, fewer vertical ties are always better.

8 Discussion

8.1 General

While developing the program, there were some challenges. The challenges that arose during development and the choices that were made will be discussed in this section. The limitations of the program will also be stated here.

8.2 The process

This thesis was meant to continue a previously done project work. This project had a script for varying all the possible parameters of a simply supported deep beam with a central point load, similar to the problem in example 1 in section 6.2. The only thing brought further from this project was the functions for the design checks of the nodes and trusses. Since these were already written, the rest of the script was developed around these functions.

The development started small and was built step by step outwards from there. Each step adding some difficulty and limitations to the program. The most challenging part of the development was calculating the internal forces of the elements in the system. Many trial programs were made based on iteration like a human would have done it. However, this turned out to be much more difficult than anticipated since humans have a more extensive understanding of the whole system and can make informed choices on what order the calculations should be done. This resulted in a need to develop new methods, and the system implemented was, despite its drawbacks, a good choice for developing the program.

The GUI part of the program may, was developed because of a desire to have some interactable user interface and the opportunity to learn about coding GUI in Python. So, even though it has its flaws, the GUI works, but the most important part of its development was the learning experience.

During the development process, some assumptions and simplifications were made. The most notable one in the program is the choice of width of horizontal struts. This was set to be the same as $2*c$ (2 times distance from surface to center of tie) as this was an input variable that could easily be changed without affecting the system too much. This value has proven to be one of the most important values in the program, as most failures encountered during testing are because of this value being too low. Maybe it should have been a separate variable for the user to decide, since arbitrarily increasing the c variable, also increase the strength in the CCT- and CTT-nodes.

Another assumption is that all the struts in the system are designed as if they are prismatic struts experiencing transverse tensile stress. This was chosen as it is the most conservative way of designing struts, and there would be great difficulty in scripting the program to differentiate different types of struts. So, if a strut fails in the program, there might be other design methods for struts that can be proven okay for the same values. Adding bursting reinforcement might help even more.

Other assumptions made are that other design rules for STM generation are not checked (like the angle between strut and tie), all the reinforcement are assumed to have sufficient anchorage, and requirements for minimum reinforcement are met or checked by the user.

8.3 Limitation

8.3.1 General

Even though the program was written to be as all-encompassing as possible, the finished product has some limitations. These limitations will be discussed in detail in this section. These are the limitation found in this thesis, but there are probably more that will be uncovered by further use of the program.

8.3.2 Geometry of the concrete structure

One of the most significant limitations may be the lack of options in choosing different geometries of the concrete structures. It would seem like the element only can be square, which is somewhat true. It is only the truss system itself that is calculated and drawn. The user defines this system, and as long as the geometry of an implied concrete element can encompass the truss system, the limitation of different geometries diminishes, as can be seen by the non-square nature of example 3. However, one must be careful about the forces around corners and holes in the element, as these have restrictive geometry and might not have room for the given nodes or struts. So, caution is advised if these kinds of concrete structures are to be calculated using this program.

8.3.3 Forces on the concrete structure

Another limitation is that the program does not have a capability for other types of external forces than point loads. So, other types of forces like distributed loading are impossible. If the concrete structure has distributed loading, the program user must decompose these loads into point loads, as is the norm when constructing an STM (see example 2 for an example of this).

Further, having a force anywhere other than directly on a node is impossible. The force can only be at nodes because the strut and tie model system is based on where the forces are applied. Thus, there will always be a node where an external force is applied.

In the script version of the program, there is a possibility to add forces with other angles than the verticals. Calculating the forces in the system with different angles should be fine, but some of the checks are not particularly good at handling forces with angles, and errors may occur.

All the external forces must be in equilibrium. The solve function from the NumPy package can run even though the concrete structure is not in equilibrium. However, the residual after this calculation is high if the equilibrium condition is not met. A check for a high residual exists, but it is best practice to ensure equilibrium before making the system. Sometimes this residual is high even though the forces are in equilibrium. Often when this is the case, the

calculated forces in the system are roughly correct, but this would need to be investigated by the user.

8.3.4 Node

The program is mainly written for nodes to be the intersection between three trusses/forces, and some nodes that can handle more than three. Less than three forces typically give an error. Usually, no nodes experience just two forces, which only happens with poorly designed STM or when an external load is transported by one truss to a node. This can be overcome by placing the force directly on the internal nodes rather than at the edge, as shown in example 2.

When many things happen at a node, there are even more things to consider when designing: geometry, forces, node type, which forces to combine to perform design checks, among others. To be able to accommodate every type of node would be laborious. Thus, only the checks for the most common nodes have gotten their own code. Trying to design for an advanced node should not crash the program and still give results, but one should be aware that these might not be completely accurate and check these nodes' results.

8.3.5 Arrows

When plotting the systems, drawing the arrows for the forces is a significant contributor to error messages. As this part of the program was not prioritized the most, the arrows can be miscalculated, such that they become very large and sometimes too large in the plotted output. If this is the case, the user can turn off the arrows by editing the Boolean variable *arrows* from True to False in line 568 in the STM.py file.

8.4 Examples

Referring to the examples in section 6, they show that the results from the program are satisfactory compared to the results from the online example. There is more trouble in the complex example in section Example 3: Deep beam with large openings and recess. It shows that the results are satisfactory, but when a complex STM such as this should be established carefully. It is important to check that the internal forces are correct, especially if the residual number is high. The program has no trouble establishing and checking a simple STM such as Example 1: two-pile cap and Example 2: deep beam.

8.5 Study

The studies done in section 7 show how the program can be used to find the most optimal solution for an STM. It uses the same concrete structure and does some changes to compare the strain energy. Setting up the models and doing the iterations in Python were done manually. It might seem like a lot of work doing the iterations manually, but one of the program's main features is that it is easy to change variables manually. Hence the process of iteration did not take much time at all. Most of the human labor time was used to design the models and compare the results. This was just one example of studies that can be done with this program, and the options are almost limitless.

8.6 Using the program

The intention when developing the program was to create a program that would be intuitive to use by external users. It is hard to know whether it was succeeded or not before external users try using it on real structures. The authors discussed with friends and family to check if the program and the GUI were intuitive. There was some feedback, and after suggestions, some changes to the GUI were made. To have created a fully optimal user-friendly program, it should have been tested more by external users.

9 Conclusion

Strut and tie modeling is an effective tool when designing complex concrete structures where standard design rules cannot be applied. Even though the calculations are simple, there are often a lot of them, which will make the design process long and laborious.

Thus, this thesis aimed to develop an all-encompassing program in Python to design strut and tie models and use this program to perform a small study. Despite all the program's limitations, it can be a robust tool in designing a structure using STM, showcased by the examples in section 6. The precalculated examples and the results from the calculations done by the program coincide.

STM is a method based on iteration to find the most optimal solutions for a structure, which is easily done with the program. Even though the iteration needs to be done manually, it is designed so that changing variables, node placements, and changes to the STM should be relatively quick and easy. After performing the study in section 7, it is clear that it is easy to find the optimal solution by making small changes to an STM and concludes that the goals of making a functional Python program are fulfilled.

10 Future Work

The future of this program is dependent on its overall capabilities. While there are known limitations, there are also bound to be some unknown limitations. Hence, the program needs to be used more to uncover its overall value.

Since the program is publicly available, future work on this topic will be to build on the existing framework or use it as an inspiration for a new program with the intent of lowering the number of limitations to get an even more all-encompassing freely available program.

As for the program itself, the input method can be developed more. Either by having an easier way of setting up the system or a more capable and fleshed-out GUI.

References

- akrowne. (2013a, mars 22). *Linear least squares*. <https://planetmath.org/linearleastsquares>
- akrowne. (2013b, mars 22). *Overdetermined*. <https://planetmath.org/overdetermined>
- Chen, W. F., & El-Metwally, S. E. (2017). *Understanding Structural Engineering: From Science to Engineering*. 16.
- Colorito, A. B., Wilson, K. E., Bayrak, O., & Russo, F. M. (2017). *Strut-and-Tie Modeling (STM) for Concrete Structures*. Federal Highway Administration.
<https://www.fhwa.dot.gov/bridge/concrete/nhi17071.pdf>
- English, T. (2019, november 7). *Finite Element Analysis Is the Foundation of All Mechanical Engineering Simulation*. <https://interestingengineering.com/what-is-finite-element-analysis-and-how-does-it-work>
- Getting started with GitHub Desktop*. (2022). GitHub Docs. Hentet 26. april 2022, fra <http://ghdocs-prod.azurewebsites.net:80/en/desktop/installing-and-configuring-github-desktop/overview/getting-started-with-github-desktop>
- Ghoraba, A., El-Metwally, S., & El-Zoughiby, M. (2020). The strut-and-tie model and the finite element -good design companions. *Journal of Structural Engineering & Applied Mechanics*, 3, 244–275. <https://doi.org/10.31462/jseam.2020.04244275>
- Goodchild, C. H., Morrison, J., & Vollum, R. L. (2015). *Strut-and-tie Models: How to design concrete members using strut-and-tie models in accordance with Eurocode 2*.
- Hu, Q., Ley, M. T., & Russell, B. W. (2014). Determining Efficient Strut-and-Tie Models for Simply Supported Beams Using Minimum Strain Energy. *ACI Structural Journal*, 111(5), 1015–1025.
- Kuhlman, D. (2012, april 22). *A Python Book: Beginning Python, Advanced Python, and Python Exercises*.
https://web.archive.org/web/20120623165941/http://cutter.rexx.com/~dkuhlman/python_book_01.html

Make your first Git commit | GitLab. Hentet 27. april 2022, fra

https://docs.gitlab.com/ee/tutorials/make_your_first_git_commit.html

NumPy Developers. (2022). *numpy.linalg.lstsq—NumPy v1.22 Manual.*

<https://numpy.org/doc/stable/reference/generated/numpy.linalg.lstsq.html>

O’Grady, A. (2018). *GitLab Quick Start Guide: Migrate to GitLab for all your repository management solutions.* Packt Publishing Ltd.

Python, R. *Python GUI Programming With Tkinter – Real Python.* Hentet 23. mai 2022, fra

<https://realpython.com/python-gui-tkinter/>

Python Software Foundation. (2012, februar 24). *Why is Python a dynamic language and also a strongly typed language—Python Wiki.*

<https://wiki.python.org/moin/Why%20is%20Python%20a%20dynamic%20language%20and%20also%20a%20strongly%20typed%20language>

Python Software Foundation. (2022, april 26). *General Python FAQ — Python 3.10.4*

documentation. <https://docs.python.org/3/faq/general.html#why-was-python-created-in-the-first-place>

Raybaut, P. (2009, oktober 18). *[PyQt] [ANN] Spyder v1.0.0 released.*

<https://www.riverbankcomputing.com/pipermail/pyqt/2009-October/024764.html>

Schlaich, J., Schafer, K., & Jennewein, M. (1987). Toward a Consistent Design of Structural Concrete. *PCI Journal*, 32(3), 74–150.

<https://doi.org/10.15554/pcij.05011987.74.150>

Spinellis, D. (2012). Git. *IEEE Software*, 29(3), 100–101.

<https://doi.org/10.1109/MS.2012.61>

Spyder Doc Contributors. (2022). *Welcome to Spyder’s Documentation—Spyder 5*

documentation. <https://docs.spyder-ide.org/current/index.html>

Tae, K. M. (2021, april 28). *Strut-and-Tie Model: Part 1 - Basics.*

<https://www.midasbridge.com/en/blog/bridgeinsight/strut-and-tie-model-part-1-basics>

Tae, K. M. (2022, januar 19). *Strut-and-Tie Model: Part 2 - Determining STM*.

<https://www.midasbridge.com/en/blog/bridgeinsight/strut-and-tie-model-part-2-determining-stm>

Todisco, L. (2009). *Test evidence for applying Strut-and-Tie Models to deep beams*.

<https://doi.org/10.13140/RG.2.2.30462.38721>

Varsity Tutors. (2007). *Law of Sines*.

https://www.varsitytutors.com/hotmath/hotmath_help/topics/law-of-sines

walter. (2017, oktober 21). How to select the most appropriate strut-and-tie model in your design? *Wallingford Consultancy*.

<https://wallingford.com.my/index.php/2017/10/21/select-appropriate-strut-tie-model/>

Appendix

Appendix A: STM code

Appendix B: GUI code

A: STM code

```
import numpy as np
import math as mt
import numpy.linalg as la

class Node:
    def __init__(self,nodePosition_,forceMagnitude_= 0,forceWidth_= 0,
                 forceAngle_=np.pi/2):
        #The nodePosition must be a list that represent the coordinates to the node
        #in integer numbers. The rest of the arguments should be integer numbers.
        if(isinstance(nodePosition_, list)):
            if(len(nodePosition_)==2):
                if(isinstance(nodePosition_[0], int) and
                   isinstance(nodePosition_[1], int)):
                    self.nodePosition = nodePosition_
                    self.nodeX = nodePosition_[0]
                    self.nodeY = nodePosition_[1]
                else:
                    print("The node list needs to contain integer numbers")
            else:
                print("The node list has to be of length 2")
        else:
            print("Then nodePosition needs to be of type list")
        self.forceMagnitude = 0
        self.forceAngle = 0
        self.forceWidth = 0
        self.isOkay=True
        if(forceMagnitude_!=0):
            self.addForce(forceMagnitude_,forceWidth_,forceAngle_)
```

```

def addForce(self,forceMagnitude_,forceWidth_,forceAngle_=np.pi/2):
    #adds force to node
    if(isinstance(forceMagnitude_, int) and
        isinstance(forceAngle_, float) and
        isinstance(forceWidth_, int)):
        self.forceMagnitude = forceMagnitude_
        self.forceAngle = forceAngle_
        self.forceWidth = forceWidth_
        self.forceX = self.forceMagnitude*-mt.cos(forceAngle_)
        self.forceY = self.forceMagnitude*mt.sin(forceAngle_)
    else:
        print("The forceMagnitude and forceWidth needs to be int," +
            " and the forceAngle float")

```

```

def nodeType(self,system_): #determines node type
    system=system_
    elements=system.elementsFromNode(self)
    ties=0
    for elem in elements:
        if elem.forceMagnitude < 0:
            ties+=1
    if (len(elements) >= 3 and
        abs(self.forceMagnitude) >0) or len(elements)> 3 :
        #checks if more than 3 trusses at node
        if ties == 2:
            tieang=[]
            for elem in elements:
                if elem.forceMagnitude < 0:
                    ang=angle([elem.node2.nodePosition[0]-
                        elem.node1.nodePosition[0],

```

```

        elem.node2.nodePosition[1]-
        elem.node1.nodePosition[1]))
    tieang.append(ang)
    if tieang[0]+tieang[1] == 0 or tieang[0]+tieang[1] == np.pi:
        #if two ties are paralell, they count as one
        ties-=1
if ties==0:
    self.nodeType="CCC"
elif ties==1:
    self.nodeType="CCT"
else:
    self.nodeType="CTT"

```

class Element: #setup of element class, representing the trusses

```
def __init__(self,node1_,node2_):
```

```
    self.node1 = node1_
```

```
    self.node2 = node2_
```

```
    self.width = []
```

```
    self.isOkay=True
```

```
def returnOtherNode(self,node_): #return the other node of the element
```

```
    if(node_ == self.node1):
```

```
        return self.node2
```

```
    elif(node_ == self.node2):
```

```
        return self.node1
```

```
    else:
```

```
        print('not valid node')
```

```
def addForce(self,forceMagnitude_): #adds force to the element
```

```
    self.forceMagnitude = forceMagnitude_
```

```
def addWidth(self,width): #adds width to element in a list
    self.width.append(width)
```

```
class System: #setup for the STM system
```

```
def __init__(self):
    self.nodes = [] #The list of all nodes in the system
    self.elements = [] #The list of all elements in the system
```

```
def addNode(self,node_): #adds nodes to the system
    if(self.existingNode(node_.nodePosition) == False):
        self.nodes.append(node_)
    else:
        print("This node already exist!")
```

```
def addElementWithNodes(self,node1_,node2_): #adds elements to the system
    element = Element(node1_, node2_)
    if(self.existingNode(node1_.nodePosition) == False):
        print("This node1 does not exist. Creat node first")
    elif(self.existingNode(node2_.nodePosition) == False):
        print("This node2 does not exist. Creat node first")
    elif(self.existingElement(element)):
        print("This element already exist")
    else:
        self.elements.append(element)
```

```
def addElementWithElement(self,element_): #adds elements to the system
    node1 = element_.node1
    node2 = element_.node2
    self.addElementWithNodes(node1, node2)
```



```

def connectedNodes(self,node_):
    #returns which nodes are connected to another node by an element
    connectedNodes = []
    if(self.existingNode(node_.nodePosition) != False):
        for element in self.elements:
            if(element.node1.nodePosition == node_.nodePosition):
                connectedNodes.append(element.node2)
            elif(element.node2.nodePosition == node_.nodePosition):
                connectedNodes.append(element.node1)
        return connectedNodes
    else:
        print("This node is not part of the system")

```

```

def elementsFromNode(self,node_):
    #returns which elements are connected to a node
    connectedElements = []
    if(self.existingNode(node_.nodePosition)):
        for element in self.elements:
            if (node_.nodePosition == element.node1.nodePosition
                or node_.nodePosition == element.node2.nodePosition):
                connectedElements.append(element)
        return connectedElements
    else:
        print("This node is not part of the system")

```

```

def existingNode(self,nodePosition_):
    #checks if a node exist in the system
    for node in self.nodes:
        if(node.nodePosition == nodePosition_):
            return node

```

```
return False
```

```
def existingElement(self,element_):
```

```
    #checks if an element exist in the system
```

```
    for element in self.elements:
```

```
        if((element.node1.nodePosition==element_.node1.nodePosition and  
            element.node2.nodePosition==element_.node2.nodePosition) or  
            (element.node2.nodePosition==element_.node1.nodePosition and  
            element.node1.nodePosition==element_.node2.nodePosition)):
```

```
            return element
```

```
    return False
```

```
class ElementForce: #calculates the forces in the trusses
```

```
    def __init__(self,system_):
```

```
        self.system = system_
```

```
        self.solve()
```

```
        self.addTypes()
```

```
    def elementMatrix(self):
```

```
        #setup of matrix with the contributions from the trusses on the nodes
```

```
        nNodes = len(self.system.nodes)
```

```
        nElements = len(self.system.elements)
```

```
        elementMatrix = np.zeros((nNodes*2,nElements)) #creating the matrix
```

```
        for i in range(nNodes*2):
```

```
            if(i%2 == 0): #forces in x direction
```

```
                iNode = int(i/2)
```

```
                node = self.system.nodes[iNode]
```

```
                connectedElements = self.system.elementsFromNode(node)
```

```
                for iElement in range(nElements):
```

```
                    if(self.system.elements[iElement] in connectedElements):
```

```

        element = self.system.elements[iElement]
        elementMatrix[i][iElement] = self.xDirection(node,
element.returnOtherNode(node))

    elif(i%2 == 1): #forces in y direction
        iNode = int((i-1)/2)
        node = self.system.nodes[iNode]
        connectedElements = self.system.elementsFromNode(node)
        for iElement in range(nElements):
            if(self.system.elements[iElement] in connectedElements):
                element = self.system.elements[iElement]
                elementMatrix[i][iElement]=self.yDirection(node,
element.returnOtherNode(node))
        return elementMatrix

def xDirection (self,node1_,node2_): #calculates the contribution in x-direction
    x1=node1_.nodeX
    x2=node2_.nodeX
    y1=node1_.nodeY
    y2=node2_.nodeY
    L = mt.sqrt((x2-x1)**2+(y2-y1)**2)
    if L == 0: #failsafe
        return 0
    return (x2-x1)/L

def yDirection (self,node1_,node2_): #calculates contribution i y-direction
    x1=node1_.nodeX
    x2=node2_.nodeX
    y1=node1_.nodeY
    y2=node2_.nodeY
    L=mt.sqrt((x2-x1)**2+(y2-y1)**2)

```

```
if L == 0: #failsafe
    return 0
return (y2-y1)/L
```

```
def forceMatrix(self): #setup of matrix of external forces on the nodes
    forceMatrix = np.zeros(len(self.system.nodes)*2)
    i = 0
    for node in self.system.nodes:
        if(node.forceMagnitude):
            forceMatrix[i] = node.forceX
            forceMatrix[i+1] = node.forceY
        i+=2
    return forceMatrix
```

```
def solve(self): #solves the matrix equation, and add the truss forces into the elements
    elementMatrix = self.elementMatrix()
    forceMatrix = self.forceMatrix()
    self.elementForce = la.lstsq(elementMatrix, forceMatrix , rcond=None)
    res=False
    for force in self.system.nodes:
        l=len(str(int(force.forceMagnitude)))
        if self.elementForce[1] > 10**(l-1) and l !=1:
            res=True
    if res: #checks if residual is too high
        self.system.residual="\nHigh residual in the calculation of forces, please check
equilibrium!!"
    else:
        self.system.residual=""
    i = 0
    for force in self.elementForce[0]:
        self.system.elements[i].addForce(force)
```

```
    i+=1
return self.elementForce
```

```
def addTypes(self): #adds the nodetypes to the node objects after the trussforces have
been calculated
```

```
    for node in self.system.nodes:
        node.nodeType(self.system)
```

```
class Checks:
```

```
def __init__(self,system_,thicness_,conClass_,c_):
    #performs design checks for all nodes and elements in the system
    self.system=system_
    self.thickness=thicness_
    self.conClass=conClass_*0.85/1.5
    self.v=1-self.conClass/250
    self.c=c_
    self.output_strings = []
    self.is_okay=True
    self.check()
    string=self.system.residual
    self.output_strings.append(string)
    print(string)
```

```
def check(self):
    num=1 #nodenumber
    for node in self.system.nodes:
        elements=self.system.elementsFromNode(node)
        morethan3=False
        if (len(elements) >= 3 and abs(node.forceMagnitude) >0) or len(elements)> 3 :
#checks if more than 3 forces act on the node
```

```

morethan3=True
if node.nodeType=="CCC":
    isflat=False
    for elem in elements: #checks if one of the struts are flat
        vector=[elem.node2.nodePosition[0]-
elem.node1.nodePosition[0],elem.node2.nodePosition[1]-elem.node1.nodePosition[1]]
        ang=angle(vector)

    if ang==0:
        isflat=True
        flat=elem
if node.forceMagnitude: #if external force exist
    F1=node.forceMagnitude
    width1=node.forceWidth
    if isflat:
        F2=flat.forceMagnitude
        width2=2*self.c      #assumes height of flat strut to be 2* cover
        flat.addWidth(width2)
        comp=[] #components
        ang=[] #angles
        for elem in elements:
            if elem != flat:
                comp.append(elem)
                node2=elem.returnOtherNode(node)
                vector=[node2.nodePosition[0]-
node.nodePosition[0],node2.nodePosition[1]-node.nodePosition[1]]
                ang.append(angle(vector))
        if morethan3: #calculates the resultant if more than 3 forces
            R=0
            vert=0
            hor=0

```

```

i=0
while i < len(comp):
    R+=comp[i].forceMagnitude**2
    vert+=comp[i].forceMagnitude*mt.sin(ang[i])
    hor+=comp[i].forceMagnitude*mt.cos(ang[i])
    i+=1
R=mt.sqrt(R)
ang=abs(mt.atan(vert/hor))
F3=R
width3=width1*mt.sin(ang)+width2*mt.cos(ang)
node.isOkay=self.CCC(F1, F2, F3, width1, width2, width3, self.thickness,
self.conClass, self.v, num)
else: #don't need a resultant
    F3=comp[0].forceMagnitude
    width3=abs(width1*mt.sin(ang[0]))+abs(width2*mt.cos(ang[0]))
    comp[0].addWidth(width3)
    node.isOkay=self.CCC(F1, F2, F3, width1, width2, width3, self.thickness,
self.conClass, self.v, num)
else: #isn't flat
    comp=[]
    ang=[]
    for elem in elements:
        comp.append(elem)
        node2=elem.returnOtherNode(node)
        vector=[node2.nodePosition[0]-
node.nodePosition[0],node2.nodePosition[1]-node.nodePosition[1]]
        angtemp=angle(vector)
        if angtemp > np.pi/2: angtemp=np.pi-angtemp
        ang.append(angtemp)
    F2=comp[0].forceMagnitude
    width2=width1*mt.sin(ang[0])/(mt.sin(np.pi-ang[0]-ang[1]))
    F3=comp[1].forceMagnitude

```

```

width3=width1*mt.sin(ang[1])/(mt.sin(np.pi-ang[0]-ang[1]))
comp[0].addWidth(width2)
comp[1].addWidth(width3)
node.isOkay=self.CCC(F1, F2, F3, width1, width2, width3, self.thickness,
self.conClass, self.v, num)
else: #no external force, internal CCC node is smeared
string = "Node "+str(num)+" is a smeered node, no check needed"
print(string)
self.output_strings.append(string)
elif node.nodeType=="CCT":
if node.forceMagnitude > 0: #if external force exist
F1=node.forceMagnitude
width1=node.forceWidth
if morethan3:
comp=[]
ang=[]
strut=0
tie=0
for elem in elements:
if elem.forceMagnitude < 0:
tie+=1
else:
strut+=1
comp.append(elem)
node2=elem.returnOtherNode(node)
vector=[node2.nodePosition[0]-
node.nodePosition[0],node2.nodePosition[1]-node.nodePosition[1]]
angtemp=angle(vector)
ang.append(angtemp)
if strut==2: #2 struts at CCT node with external force, checked twice,
once for each side

```



```

        width2_1=width1*np.sin(ang[0])+2*self.c*np.cos(ang[0]) #assumes
height of tie to be 2* cover
        width2_2=width1*np.sin(ang[1])+2*self.c*np.cos(ang[1])
        F2_1=comp[0].forceMagnitude
        F2_2=comp[1].forceMagnitude
        comp[0].addWidth(width2_1)
        comp[1].addWidth(width2_2)
        node.isOkay=self.CCT(F1, F2_1, self.thickness, width1, width2_1,
self.conClass, self.v, num)
        node.isOkay=self.CCT(F1, F2_2, self.thickness, width1, width2_2,
self.conClass, self.v, num)
    else:
        string = "please have a maximum of 2 struts at node " +str(num)
        print(string)
        self.output_strings.append(string)
else: #if not more than 3
    for elem in elements:
        if elem.forceMagnitude > 0:
            node2=elem.returnOtherNode(node)
            vector=[node2.nodePosition[0]-
node.nodePosition[0],node2.nodePosition[1]-node.nodePosition[1]]
            ang=angle(vector)
            F2=elem.forceMagnitude
            width2=width1*np.sin(ang)+2*self.c*np.cos(ang) #assumes heighth
of tie to be 2* cover
            elem.addWidth(width2)
            node.isOkay=self.CCT(F1, F2, self.thickness, width1, width2, self.conClass,
self.v, num)
    else: #no external force
        if morethan3:
            comp=[]
            ang=[]
            strut=0

```

```

isflat=False
for elem in elements:
    if elem.forceMagnitude > 0:
        strut+=1
        comp.append(elem)
        node2=elem.returnOtherNode(node)
        vector=[node2.nodePosition[0]-
node.nodePosition[0],node2.nodePosition[1]-node.nodePosition[1]]
        angtemp=angle(vector)
        ang.append(angtemp)
        if angtemp==0:
            isflat=True
            flat=elem
            comp.remove(flat)
if strut==3:
    if isflat:
        F1=flat.forceMagnitude
        width1=2*self.c
        flat.addWidth(width1)
        R=0
        vert=0
        hor=0
        i=0
        while i < len(comp):
            R+=comp[i].forceMagnitude**2
            vert+=comp[i].forceMagnitude*mt.sin(ang[i])
            hor+=comp[i].forceMagnitude*mt.cos(ang[i])
            i+=1
        R=mt.sqrt(R)
        ang=abs(mt.atan(vert/hor))
        F2=R

```

```

        width2=width1*mt.sin(ang)+width2*mt.cos(ang)
        node.isOkay=self.CCT(F1, F2, self.thickness, width1, width2,
self.conClass, self.v, num)
    else:
        string = "Node "+str(num)+" is a smeared node, no check needed"
        print(string)
        self.output_strings.append(string)
    else:
        string = "Too many struts at node "+str(num)
        print(string)
        self.output_strings.append(string)
else: #3 trusses
    comp=[]
    ang=[]
    isflat=False
    for elem in elements:
        if elem.forceMagnitude > 0:
            comp.append(elem)
            node2=elem.returnOtherNode(node)
            vector=[node2.nodePosition[0]-
node.nodePosition[0],node2.nodePosition[1]-node.nodePosition[1]]
            angtemp=angle(vector)
            ang.append(angtemp)
            if angtemp==0:
                isflat=True
                flat=elem
                comp.remove(flat)
    if isflat:
        F1=flat.forceMagnitude
        width1=2*self.c
        flat.addWidth(width1)

```

```

        F2=comp[0].forceMagnitude
        width2=width2=width1*mt.sin(ang[0])+width2*abs(mt.cos(ang[0]))
        comp[0].addWidth(width2)
        node.isOkay=self.CCT(F1, F2, self.thickness, width1, width2,
self.conClass, self.v, num)
    else: #isn't flat
        string = "Node "+str(num)+" is a smeared node, no check needed"
        print(string)
        self.output_strings.append(string)
else: #CTT node
    for elem in elements:
        if elem.forceMagnitude > 0:
            F1=elem.forceMagnitude
            node2=elem.returnOtherNode(node)
            vector=[node2.nodePosition[0]-
node.nodePosition[0],node2.nodePosition[1]-node.nodePosition[1]]
            ang=angle(vector)
            width1=self.c*np.sin(ang)+2*self.c*np.cos(ang)
            elem.addWidth(width1)
            node.isOkay=self.CTT(F1, self.thickness, width1, self.conClass, self.v, num)
        num+=1    #adds to the nodenumber
print("")
self.output_strings.append("")
num=1 #truss number
for elem in self.system.elements:
    if elem.forceMagnitude >0: #checks if strut or tie
        elem.isOkay=self.strut(elem, self.thickness, self.conClass, self.v, num)
    else:
        self.tie(elem.forceMagnitude,num)
    num+=1 #adds to trussnumber
if self.is_okay: #if the system doesnt fail

```

```

string="\nThe capacity of the system is enough"
print (string)
self.output_strings.append(string)
s=strainEnergy(self.system)      #calculate strain energy if the system holds
string='\nThe strain energy in the system is ' +str(int(s))+' J'
print (string)
self.output_strings.append(string)
else:
    string="\nThe capacity of the system is NOT enough!!!"
    print (string)
    self.output_strings.append(string)

def CCC(self, F1, F2, F3, width1, width2, width3, bredd, strength, v, num):
    #design check of CCC node
    sigma_1=(F1/(bredd*width1))    #stresses
    sigma_2=(F2)/(bredd*width2)
    sigma_3=F3/(bredd*width3)
    sigma_M=v*strength             #Ec2. formula(6.60) for ccc-node
    if sigma_1 > sigma_M or sigma_2 > sigma_M or sigma_3 > sigma_M:
        string = "The capacity of node " +str(num)+" is not enough."
        print (string)
        self.output_strings.append(string)
        self.is_okay=False
        return False
    else:
        string = "The capacity of node " +str(num)+" is OK!"
        print(string)
        self.output_strings.append(string)
        return True

```

```
def CCT(self, F1, F2, bredd, width1, width2, strength, v, num): #design check for CCT
node
```

```
    sigma_1=F1/(width1*bredd)      #stresses
```

```
    sigma_2=F2/(width2*bredd)
```

```
    sigma_M=0.85*v*strength        #Ec2. formula(6.61) for cct-node
```

```
    if sigma_1 > sigma_M or sigma_2 > sigma_M:
```

```
        string = "The capacity of node " +str(num)+" is not enough."
```

```
        print (string)
```

```
        self.output_strings.append(string)
```

```
        self.is_okay=False
```

```
        return False
```

```
    else:
```

```
        string="The capacity of node " +str(num)+" is OK!"
```

```
        print(string)
```

```
        self.output_strings.append(string)
```

```
        return True
```

```
def CTT(self, F1, bredd, width1, strength, v, num): #design check for CTT node
```

```
    sigma_1=F1/(width1*bredd)      #stress
```

```
    sigma_M=0.75*v*strength        #Ec2. formula(6.62) for ctt-node
```

```
    if sigma_1 > sigma_M:
```

```
        string = "The capacity of node " +str(num)+" is not enough."
```

```
        print (string)
```

```
        self.output_strings.append(string)
```

```
        self.is_okay=False
```

```
        return False
```

```
    else:
```

```
        string = "The capacity of node " +str(num)+" is OK!"
```

```
        print(string)
```

```
        self.output_strings.append(string)
```

```
        return True
```

```

def strut(self, element, thickness, strength, v, num):
    #design check of strut
    isokay=True
    sigma_M=0.6*v*strength # Ec2. formula(6.56) for strut with transverse tensile stress
    for i in element.width: #checking both ends of the strut
        sigma=element.forceMagnitude/(i*thickness)
        if sigma > sigma_M:
            isokay=False

    if isokay == False:
        string = "The capacity of strut " +str(num)+" is not enough."
        print (string)
        self.output_strings.append(string)
        self.is_okay=False
        return False
    else:
        string = "The capacity of strut " +str(num)+" is OK!"
        print(string)
        self.output_strings.append(string)
        return True

def tie(self,F,num): #calculate required reinforcement in the ties
    Area_s=round(abs(F)/434 +0.5)
    string = "Required reinforcement area in tie "+str(num)+" is "+str(Area_s)+"mm^2"
    print(string)
    self.output_strings.append(string)

class Plot: #plots the system
    def __init__(self,system_):

```

```
self.system=system_  
self.pl=self.draw()
```

```
def draw(self):  
    arrows=True  
  
    import matplotlib as mpl  
    import matplotlib.pyplot as plt  
  
    f = plt.figure()  
    num=1 #nodenumber  
    for node in self.system.nodes: #plotting the nodes  
        x=node.nodeX  
        y=-node.nodeY  
        cor="("+str(node.nodeX)+","+str(-node.nodeY)+")"  
        if node.isOkay:  
            plt.plot(x, y, 'ko')  
        else:  
            plt.plot(x,y,'ro')  
        plt.annotate(num, xy=(x, y), xytext=(x+80, y-80), color='blue')  
        plt.annotate(cor, xy=(x, y) , xytext=(x, y+150),color='blue')  
        if arrows:  
            if node.forceMagnitude !=0: #arrows, not fun might break  
                lolx=len(str(int(node.forceX)))  
                if abs(node.forceX) < 1:  
                    x1=x  
                    if node.forceX<0:  
                        x1=x-(node.forceX*node.forceMagnitude/(10**(lolx-3.2)))  
                    else:  
                        x1=x+(node.forceX*node.forceMagnitude/(10**(lolx-3.2)))  
                loly=len(str(int(node.forceY*node.forceMagnitude)))  
                if node.forceAngle == (3*np.pi/2):  
                    y1=y+(node.forceY*node.forceMagnitude/(10**(loly-4.2)))
```



```

else:
    y1=y+(node.forceY*node.forceMagnitude/(10**(loly-3.2)))
    plt.annotate(str(abs(node.forceMagnitude)), xy=(x, y), xytext=(x1, y1),
arrowprops=dict(facecolor='black', shrink=0.05))
    num+=1
num=1 #element number
for elem in self.system.elements:
    xmid=(elem.node1.nodeX+elem.node2.nodeX)/2
    ymid=(-elem.node1.nodeY+-elem.node2.nodeY)/2
    #angles of the plotted forces, and numbering of element
    if (elem.node1.nodeX>elem.node2.nodeX and elem.node1.nodeY <
elem.node2.nodeY) or (elem.node2.nodeX>elem.node1.nodeX and elem.node2.nodeY <
elem.node1.nodeY):
        ang=180-mt.degrees(angle([elem.node2.nodeX-elem.node1.nodeX,-
elem.node2.nodeY--elem.node1.nodeY]))
        plt.annotate(num, xy=(x, y), xytext=(xmid+50,ymid-80))
    else:
        ang=-mt.degrees(angle([elem.node2.nodeX-elem.node1.nodeX,-
elem.node2.nodeY--elem.node1.nodeY]))
        plt.annotate(num, xy=(x, y), xytext=(xmid-60,ymid-80))
    if elem.forceMagnitude<0:
        plt.plot([elem.node1.nodeX,elem.node2.nodeX],[-elem.node1.nodeY,-
elem.node2.nodeY],'k-')
        plt.annotate(round(elem.forceMagnitude), xy=(xmid,ymid), xytext=(xmid-110,
ymid+50), rotation=ang)
    else:
        if elem.isOkay:
            plt.plot([elem.node1.nodeX,elem.node2.nodeX],[-elem.node1.nodeY,-
elem.node2.nodeY],'k--')
            plt.annotate(round(elem.forceMagnitude), xy=(xmid,ymid), xytext=(xmid-80,
ymid+50), rotation=ang)
        else:
            plt.plot([elem.node1.nodeX,elem.node2.nodeX],[-elem.node1.nodeY,-
elem.node2.nodeY],'r--')

```

```

        plt.annotate(round(elem.forceMagnitude), xy=(xmid,ymid), xytext=(xmid-80,
ymid+50), rotation=ang)
        num+=1
    plt.axis('off') #turn off axissystem
    plt.gca().set_aspect('equal', adjustable='box')
    f.set_size_inches(10, 5.6, forward=True) #set size of the figure
    return f

```

```

def strainEnergy(system): #calculates the total strain energy in the system
    strain=0
    for i in system.elements:
        if i.forceMagnitude < 0:
            l=mt.sqrt((i.node1.nodeX-i.node2.nodeX)**2+(i.node1.nodeY-i.node2.nodeY)**2)
            strain+=abs(i.forceMagnitude)*l*434/200000*10**(-3)
    return strain

```

```

def angle(vector): #calculate angle from the horizontal to a vector
    vector_2=[mt.inf,0]
    ang1 = np.arctan2(*vector[::-1])
    ang2 = np.arctan2(*vector_2[::-1])
    ang=(ang2 - ang1) % (2 * np.pi)
    if ang >np.pi/2: ang=np.pi-ang
    return abs(ang)

```

B: GUI Code

```
import tkinter as tk
import STM as e
import numpy as np
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
from matplotlib.figure import Figure

nodes = []
elements = []
system = e.System()
thicness = 0
conClass = 0
c = 0
number_nodes = 0

#first window, to specify number of nodes:
window1 = tk.Tk()

window1.title('STM')

window1.geometry('500x200')

def submitNumberNodes():
    number=ent_number_nodes.get()
    if(number.isdigit()):
        global number_nodes
        number_nodes = int(number)
        window1.destroy()
```

```
intro_string = ("Welcome! This program will help you check if your strut-and-tie model is
sufficient.\n" +
               "This program will show you four windows in sequence. Each window will\n" +
               "tell you what you need to know to fill in the required information. Good luck!")
```

```
lbl_intro = tk.Label(text =intro_string)
```

```
lbl_number_nodes = tk.Label(text="Enter number of nodes:")
```

```
ent_number_nodes = tk.Entry()
```

```
btn_submit = tk.Button(text="Submit", command = submitNumberNodes)
```

```
lbl_intro.pack()
```

```
lbl_number_nodes.pack()
```

```
ent_number_nodes.pack()
```

```
btn_submit.pack()
```

```
window1.mainloop()
```

```
#second window to setup the system
```

```
window2 = tk.Tk()
```

```
window2.title('System setup')
```

```
window2.geometry('500x200')
```

```
entry_x = [] #cordinates in x direction
```

```
entry_y = [] #cordinates in y direction
```

```
connected = [] #checkboxes
```

```
n=0
```

```
entry_force_magnitudes = [] #external forces
```

```
entry_force_widths = [] #external forces widths
```

```
error_text = []
```

```
error_labels = []
```

```
satisfied = True
```

```
def submitNodes():
```

```
    global error_text
```

```
    error_text = []
```

```
    global satisfied
```

```
    satisfied = True
```

```
    global nodes
```

```
    nodes.clear()
```

```
    global elements
```

```
    elements.clear()
```

```
    sum_force_magnitude = 0
```

```
    for i in range(number_nodes):
```

```
        force_magnitude = entry_force_magnitudes[i].get()
```

```
        force_width = entry_force_widths[i].get()
```

```
        if(force_magnitude==""):
```

```
            force_magnitude = 0
```

```
            force_width = 0
```

```
        elif(force_magnitude.isdigit() or (force_magnitude[0]=="-" and  
force_magnitude[1:len(force_magnitude)].isdigit())):
```

```
            if(force_width.isdigit()):
```

```
                force_magnitude = int(force_magnitude)
```

```
                force_width = int(force_width)
```

```
            else:
```

```
                error_text.append("Force width must be filled with integer number when force  
magnitude is applied.")
```

```

        print("Force width must be filled with integer number when force magnitude is
applied.")
        satisfied = False
        break
    else:
        error_text.append("The magnitude must be filled with integer number.")
        print("The magnitude must be filled with integer number.")
        satisfied = False
        break
    sum_force_magnitude += force_magnitude
    node_position = []
    x = entry_x[i].get()
    y = entry_y[i].get()
    val1 = True
    try:
        x = int(x)
        y = int(y)
    except:
        val1 = False
    if(val1):
        satisfied = True
        node_position.append(x)
        node_position.append(y*-1)
        if(force_magnitude<0):
            node =
e.Node(node_position,forceMagnitude_=abs(force_magnitude),forceWidth_=force_width)
            nodes.append(node)
        else:
            node =
e.Node(node_position,forceMagnitude_=force_magnitude,forceWidth_=force_width,forceAn
gle_=3*np.pi/2)
            nodes.append(node)

```

```
else:
```

```
    error_text.append("All positions need to be filled with integer number")
```

```
    print("All positions need to be filled with integer number")
```

```
    satisfied = False
```

```
    break
```

```
connected_index=0
```

```
if(sum_force_magnitude !=0):
```

```
    error_text.append("The structure needs to be at vertical equilibrium")
```

```
    satisfied = False
```

```
if(len(error_labels)>0):
```

```
    for l in error_labels:
```

```
        l.destroy()
```

```
    error_labels.clear()
```

```
if(satisfied==False):
```

```
    for t in error_text:
```

```
        label_error=tk.Label(text = t)
```

```
        label_error.pack()
```

```
        error_labels.append(label_error)
```

```
if(satisfied):
```

```
    for i in range(number_nodes-1):
```

```
        for j in range(number_nodes-1-i):
```

```
            element = []
```

```
            if(connected[connected_index].get()):
```

```
                element.append(nodes[i])
```

```
                element.append(nodes[j+i+1])
```

```
            elements.append(element)
```

```

        connected_index+=1

for node in nodes:
    global system
    system.addNode(node)

for element in elements:
    system.addElementWithNodes(element[0], element[1])
e.ElementForce(system) #calculating the internal forces of the elements
window2.destroy()

label = tk.Label( text="Input: Node cordinates, connected node(s), force at node.\n" +
                 "Right handed cordinate system, positiv x to the rigth and positive y upwards.")
label.pack(padx=1, pady=1)
grid_frame = tk.Frame(window2)

#creating the grid for the nodes:
for i in range(number_nodes+2):

    grid_frame.columnconfigure(i, weight=1)
    grid_frame.rowconfigure(i, weight=1)

for j in range(number_nodes+6):
    frame = tk.Frame(master=grid_frame)
    frame.grid(row=i, column=j,padx=1, pady=1)
    if(i==0):
        if(j==0):
            label = tk.Label(master=frame, text="Nodes")
            label.pack()
        elif(j==1):
            label = tk.Label(master=frame, text="X [mm]")

```



```

        label.pack()
elif(j==2):
    label = tk.Label(master=frame, text="Y [mm]")
    label.pack()
elif(j>2 and j<number_nodes+3):
    label = tk.Label(master=frame, text=j-2)
    label.pack()
elif(j==number_nodes+4):
    label = tk.Label(master=frame, text="Force magnitude [N]")
    label.pack()
elif(j==number_nodes+5):
    label = tk.Label(master=frame, text="Force width [mm]")
    label.pack()
if(j==0 and i>0 and i<number_nodes+1):
    label = tk.Label(master=frame, text=i)
    label.pack()
if(j==1 and i>0 and i<number_nodes+1):
    ent_number_x = tk.Entry(master=frame,width = 10)
    ent_number_x.pack()
    entry_x.append(ent_number_x)
if(j==2 and i>0 and i<number_nodes+1):
    ent_number_y = tk.Entry(master=frame,width = 10)
    ent_number_y.pack()
    entry_y.append(ent_number_y)
if(i>=1 and i<number_nodes+1 and j>i+2 and j<number_nodes+3):
    var = tk.IntVar()
    connected.append(var)
    check = tk.Checkbutton(master=frame,variable = connected[n])
    check.pack()
    n+=1
if(i > 0 and i<number_nodes+1 and j == number_nodes+4):

```

```

ent_force_magnitude = tk.Entry(master=frame,width = 18)
ent_force_magnitude.pack()
entry_force_magnitudes.append(ent_force_magnitude)
if(i > 0 and i<number_nodes+1 and j == number_nodes+5):
    ent_force_width = tk.Entry(master=frame,width = 18)
    ent_force_width.pack()
    entry_force_widths.append(ent_force_width)
if(i == number_nodes+1 and j == number_nodes+5):
    btn_submit_1 = tk.Button(master=frame,text="Submit", command =
submitNodes)
    btn_submit_1.pack()

grid_frame.pack( fill = tk.BOTH )

window2.mainloop()

#window three to specify the concrete
window3 = tk.Tk()

window3.title('Concrete specification')

window3.geometry('500x200')

def submitInputs():
    global thicness
    global conClass
    global c
    val = True

```

```
if(ent_concrete_class.get().isdigit()):
    conClass = int(ent_concrete_class.get())
else:
    val=False
if(ent_concrete_cover.get().isdigit()):
    c = int(ent_concrete_cover.get())
else:
    val=False
if(ent_thicness.get().isdigit()):
    thicness = int(ent_thicness.get())
else:
    val=False
if(val):
    window3.destroy()
```

```
lbl_instruction = tk.Label(text="All the numbers needs to be a integer number")
lbl_concrete_class = tk.Label(text="Enter the concrete class:")
ent_concrete_class = tk.Entry()
lbl_thicness = tk.Label(text="Enter the breadth of the concrete structure in mm:")
ent_thicness = tk.Entry()
lbl_concrete_cover = tk.Label(text="Enter the distance from concrete surface to center of tie in mm:")
ent_concrete_cover = tk.Entry()
btn_submit2 = tk.Button(text="Submit", command = submitInputs)
```

```
lbl_instruction.pack()
lbl_concrete_class.pack()
ent_concrete_class.pack()
lbl_thicness.pack()
ent_thicness.pack()
```

```
lbl_concrete_cover.pack()
```

```
ent_concrete_cover.pack()
```

```
btn_submit2.pack()
```

```
window3.mainloop()
```

```
checks = e.Checks(system, thicness, conClass, c) #do the checks for the stm
```

```
#window four to show results
```

```
window4 = tk.Tk()
```

```
window4.title('Results')
```

```
lbl_frame = tk.Frame(window4)
```

```
plt_frame = tk.Frame(window4)
```

```
output = checks.output_strings
```

```
for string in output:
```

```
    label=tk.Label(master = lbl_frame,text = string)
```

```
    label.pack()
```

```
plot = e.Plot(system)
```

```
fig = plot.draw()
```

```
canvas = FigureCanvasTkAgg(fig, master = plt_frame)
```

```
canvas.draw()
```

```
canvas.get_tk_widget().pack()
```

```
lbl_frame.pack(side=tk.LEFT)  
plt_frame.pack(side=tk.RIGHT)
```

```
window4.mainloop()
```

