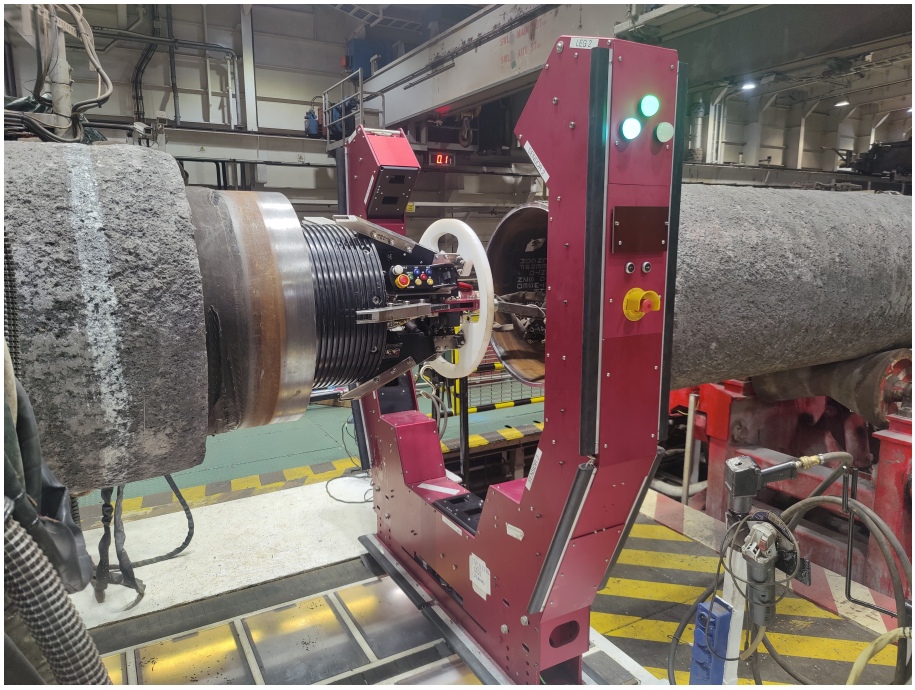
WI5118: AM Internship (cohort 2023)

Line Up Automation

Philip Soliman (4945255)

December 1, 2024



University supervisor:
Dr. Ir. L. Rens
Delft University of Technology
Faculty of Electrical Engineering, Mathematics and Computer Science

Company supervisor:
J. Ramlakhan
Senior R&D Engineer
Research & Development

Contents

1 Company description

Allseas Engineering B.V. is a Dutch company that specializes in offshore pipeline installation and subsea construction. The company was founded in 1985 by Edward Heerema and is headquartered in Delft, the Netherlands. Allseas is known for its innovative engineering solutions and state-of-the-art vessels, such as the Pioneering Spirit, which is the largest construction vessel in the world. The company has a global presence with offices in the Netherlands, Switzerland, and the United Kingdom, and employs over 2,500 people worldwide.

Allseas' core business is the installation of subsea pipelines and the construction of offshore structures for the oil and gas industry. The company's vessels are equipped with advanced technology and equipment to perform a wide range of offshore operations. [1]

Allseas' organizational structure is divided into several departments, which are shown in Figure 1.

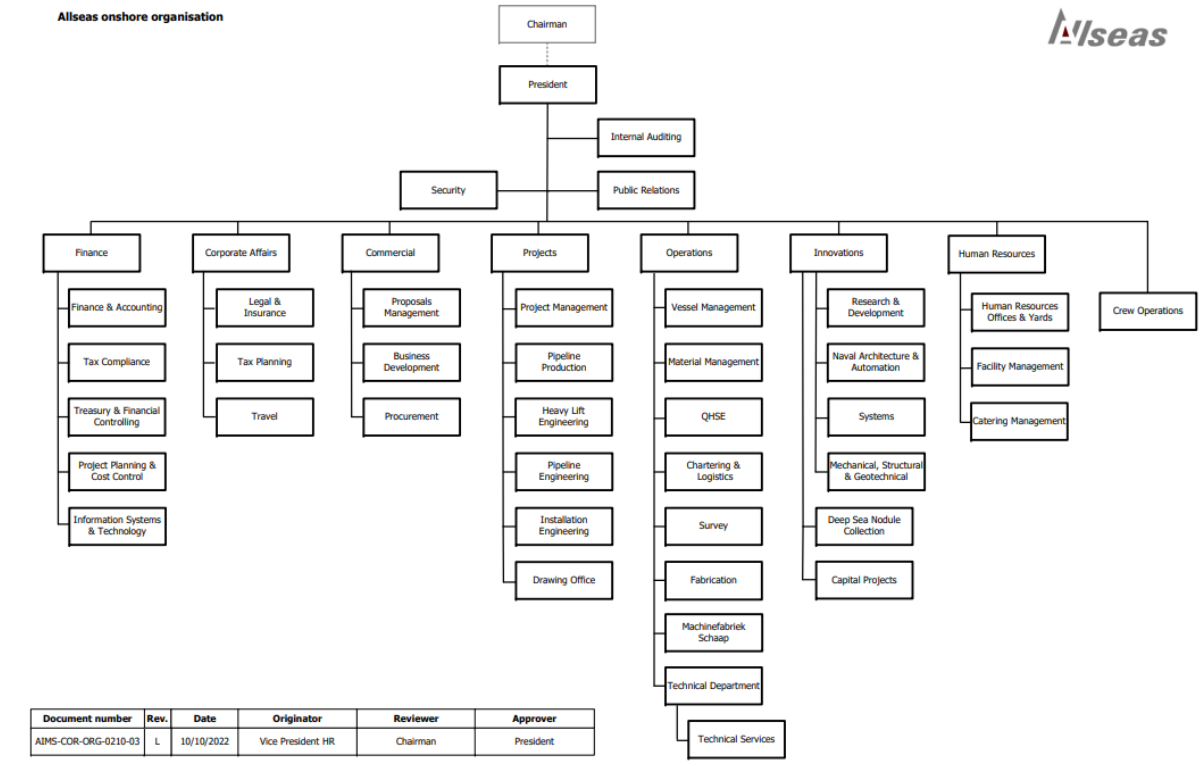


Figure 1: Allseas organizational structure.

The company's innovations department is responsible for the design and development of new technologies and solutions to improve the efficiency and safety of offshore operations. It is in this department that the internship project is situated. Figure 2 shows the organizational structure of the innovations department. In particular, the project is performed in the sub-department of Research & Development Delft, which is led by the department manager (Dominik Bujakiewicz). The project is supervised by the senior R&D engineer (J. Ramlakhan) and performed by the intern (Philip Soliman).

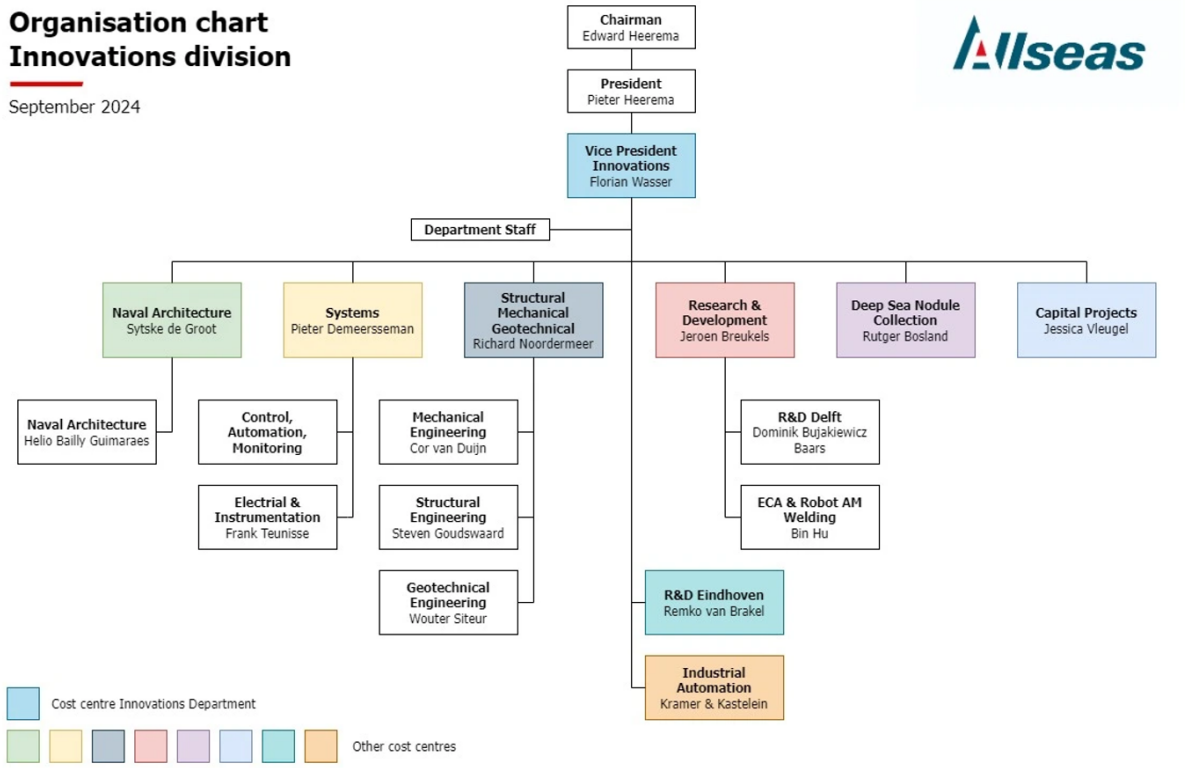


Figure 2: Allseas innovations department organizational structure.

2 Problem description

During pipelay every few minutes a new pipe (new joint) section must be fitted onto the end of the existing pipeline (main line). For decades in the offshore industry this has been a manual process. Allseas is working on a method to automate this.

2.1 Main requirements for automated line-up

Several projects have been initiated to automate the line-up process starting as early as 2008. However, These have all been unsuccessful. The latest project was started in 2017 and is still ongoing. The main requirements are:

1. Reducing line-up cycle times in the beadstall; This will result in significant cost reduction by decreasing vessel production time.
2. Increasing accuracy of line-ups (reduce amount of root pass weld errors); This will also reduce costs due to decreasing vessel production time.
3. Providing a safer work environment for the people working in the beadstall.

2.2 Current situation

The latest iteration of automated line-up is a system that uses 5 laser line scanners to measure the pipe ends' position and orientation (see figure 3).

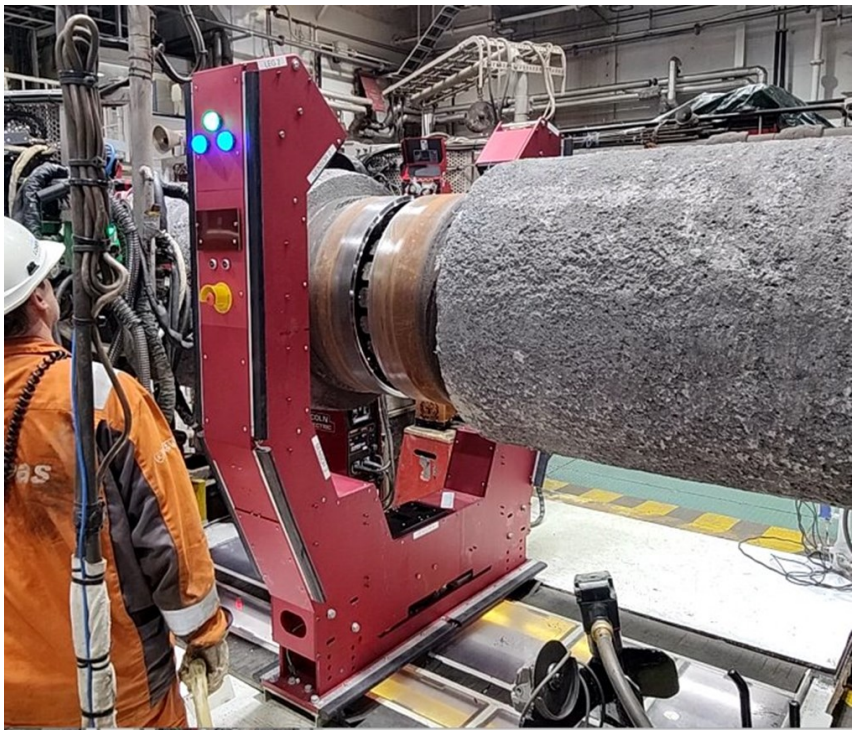


Figure 3: The portal or 'U-Frame'. The mainline is held in place in the background, while the new joint is lined up using instructions from automated line-up. The white stickers on the side of the frame indicate the position of the laser line scanners. Three on the side from which the picture is taken and two on the other side of the pipe. The U-Frame is mounted on the interface frame, which in turn is connected to the subframe. The subframe has motorized rollers and linear guides that allow the U-Frame to move with the new joint. This way the U-Frame ensures optimal visibility of the pipe ends for the laser line scanners. Additionally, the U-Frame can be moved out of way of other equipment, personel or operations when not in use.

The laser line scanners generate 4192 data points in 5 different planes with a frequency of $60 \sim 70$ Hz. Initially, a line finding algorithm derived from a Hough transform [4] is used to recover approximately straight sections from the laser's projection on the pipe. This allows for the recognition of the pipe ends or 'bevels' from the data.

Subsequently, the found pipe ends of the 5 sensors are combined to form a 3D representation of the pipe face. This is done using using a combination of Newton-Raphson optimization and a least squares fit.

2.3 Internship goals

The internship aims to improve the current automated line-up system by:

a **Decreasing code execution time** by searching for and replacing slow or redundant operations;

b **Writing maintainable code** by refactoring the current codebase;

c **Increasing machine accuracy** by comparing different pipe fitting algorithms.

3 Approach

In this section the approach to the problem is described. The approach is divided into several steps. The steps are ordered in a way that makes sense for the development of the project. The steps are as follows:

3.1 Understand automated line-up

Before any improvements can be made to the system, it is important to understand how the system works. This includes understanding the hardware, the software and the mathematical model that is used to fit the pipe ends. Next to that, getting to know the engineers that work on the project and the test set-up at the yard is also important. All of this provides a wholisitc view of the system, which in turn highlights the signifigance of the improvements described in Section 2.3.

📖 section sections 4.1 to 4.3

3.2 Familiarize with the code base

Read the code starting form the lowest level functions reading in the data received rom the laser line scanners to the highest level functions that combine the processed data to fit the pipe ends.

📖 section 4.4

3.3 Create a profiler for the code

The best way to improve code speed is to know where the bottlenecks are. A profiler will help to identify these bottlenecks. Focusing on the most time consuming parts of the code prevents wasting resources on speeding up parts of the code that are not used often.

✅ internship goal 2.3 **a**
📖 section 5

3.4 Optimize the code

Once the major bottlenecks are identified, the next step is to find more efficient ways to perform the operations in question. Adjustments may involve reducing memory allocation, re-ordering operations, correcting or updating algorithms, implenting precomputation of constant values or optimising data locality.

✅ internship goal 2.3 **a**
📖 section 6

3.5 Identify unreadable code

Halfway during the development between 2017 and the time of this internship a new standard for clean, readable code has been introduced within Allseas. Parts of the old code have also been improved, but quite some code still needs a bit of restructuring. This step serves to identify the pieces of code that are least readable and need to be updated to the new Allseas coding standard.

✅ internship goal 2.3 **b**
📖 sections 6 and 7, listings 1 and 2

3.6 Determine and improve upon current accuracy

Run new fitting algorithms on generated data to test the accuracy of the new algorithms. This will help to determine if the new algorithms are more accurate than the old ones. Do the same with recorded data from the test set-up to see if the new algorithms are also more reliable. Finally, test the algorithms on the real-time data from the test set-up to see if the new algorithms are also faster.

✅ internship goal 2.3 **c**
📖 section 7

3.7 Expand the fitting model to recognise ILUC

During line up of the new join with the main line, there is more equipment present in and around the latter. One major component present in the main line is the 'Internal Line Up Clamp' (ILUC) and is shown in Figure .

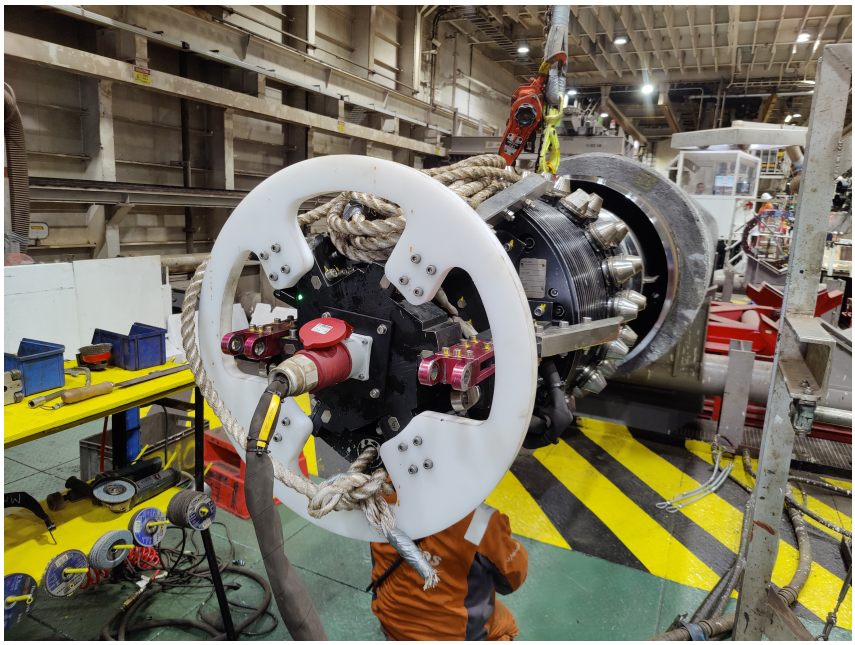


Figure 4: The Internal Line Up Clamp (ILUC). The ILUC sits in the main line, while the new join is shifted over it during line up. Its purpose is to steady the main line and new joint pipe ends after line up and during welding. Using a set of pneumatic knobs portruding from its exterior it pushes both pipe ends outward, ensuring a circular fit. Part of the ILUC is a black ribbed cylinder.

Recognising the ILUC gives a good indication of where the main line is before it can be measured, seeing as the U-Frame passes over the ILUC before it reaches the main line. Early estimation of the positon and orientation of the main line benefits the overall speed of the process, because less corrections have to be made in the final line up stage.

Therefore, the final step of the internship is the development of a new part of the recognition software finds the a specific part of the ILUC; its ribbed cylinder. As before, accuracy and reliability can be tested using the test set-up in the yard. Furthermore, a combination of different kinds of line, plane and circle fitting algorithms based on minimization of geometric distances can be used for this. Lastly, Where pipe ends tend to have deformations due to manufacturing, pre-heating or bevelling, the ILUC is a near-perfect cylinder with a defined radius. This can be used to improve the accuracy of the fitting algorithms.

✅ internship goal 2.3 **c**
📖 section 8

4 Understanding line up automation

This section describes basic concepts of the line up automation (LUA) system. The automated line-up system is a system that aligns two pipes together. The system consists of a U-Frame, laser line scanners, Line up Car Assemblies (LCAs) and a sub-frame. The U-Frame houses the laser line scanners and is mounted on the sub-frame. The sub-frame is responsible for the U-Frame's movements. The LCAs are devices that control the new joint's position and orientation. The laser line scanners are sensors that detect the new joint's position.

4.1 The U-frame

Figure 5 shows the U-Frame. The U-frame is a part of the automated line-up system. It is precisely machined to ensure that the positions of the laser line scanners are known as accurately as possible.

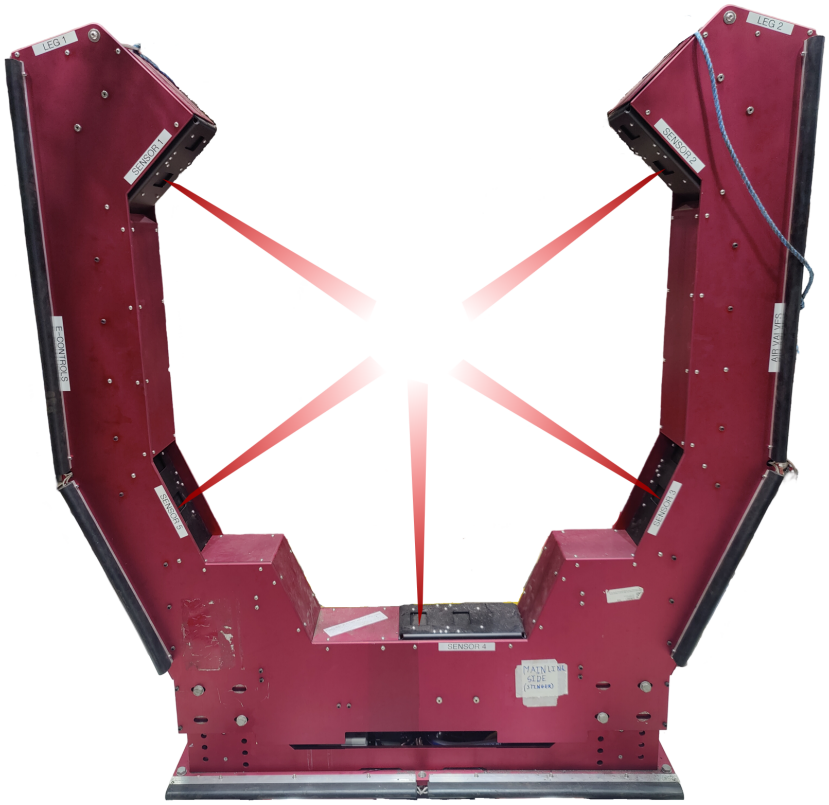


Figure 5: The U-Frame. Most importantly, the U-frame houses five laser line scanners that are used to detect pipes and other objects like the ILUC. The U-Frame is mounted on an interface frame, which in turn is mounted on the sub-frame (not visibile here). The sub-frame is responsible for the U-Frame's movements.

The U-Frame is given a right-handed cartesian coordinate system. The origin of the coordinate system is at the center of the U-Frame. The x-axis points towards the new joint, the y-axis points left (seen from the main line) and the z-axis points upwards.

4.2 Laser line scanners

The laserline scanners measure in 2 dimensions, width (x-direction) and depth (z-direction). The scanners each generate 4192 data points in any one measurement. The frequency of the measurements in only limited by how fast the laser line scanner runs (see section 4.4). The maximum scanning frequency of the lasers lies anywhere between 600 and 9000 Hz [8].

The lasers measure in their respective coordinate systems. These measurements are then transformed to the U-Frame's coordinate system. To perfrom this transformation, the laser line scanners angles with respect to the U-Frame's coordinate system are needed. These angles are

$$\phi_1 = \frac{\pi}{4}, \tag{1}$$

$$\phi_2 = \frac{3\pi}{4}, \tag{2}$$

$$\phi_3 = \frac{9\pi}{8}, \tag{3}$$

$$\phi_4 = \frac{15\pi}{8}, \tag{4}$$

$$\phi_5 = \pi, \tag{5}$$

where the angles are measured from the z-axis in the y-z plane with positive rotation defined by the right hand rule applied to the x-axis. In this simple case the angles are the same as the angles of the laser line scanners in the U-frame.

An important aspect of the laser line scanners is their Field of View (FOV). The FOV is the area in which the laser line scanners can detect objects. Figure 6 gives the FOV of the laser line scanners.

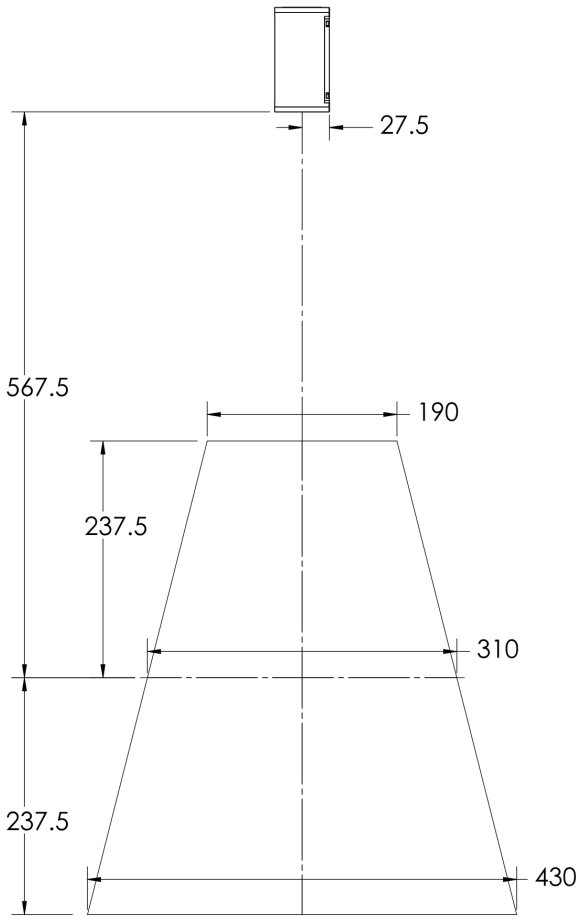


Figure 6: Field of View (FOV) of the laser line scanners. Measurements are in millimeters [9].

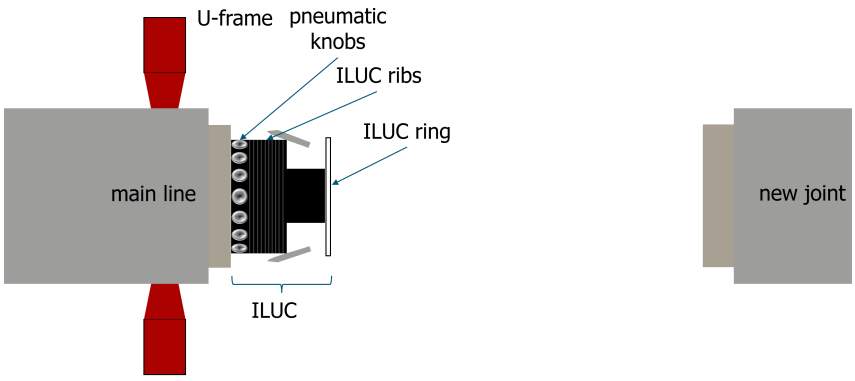
From figure 6 it can be seen that the FOV of the laser line scanners is limited. In particular, the width of the frame itself is $\sim$ 20 cm, which means the laser line scanners can see $\sim$ 10 cm beyond either side of the frame. This is why it is necessary for the U-Frame to move.

Depending on the distance of an object to a laser line scanner, the accuracy its measurement can vary. For an object close to the laser line scanners the resolution is $47\mu m$ in the x-direction. For an object further away from the laser line scanners the resolution is $104\mu m$ in the x-direction [8].

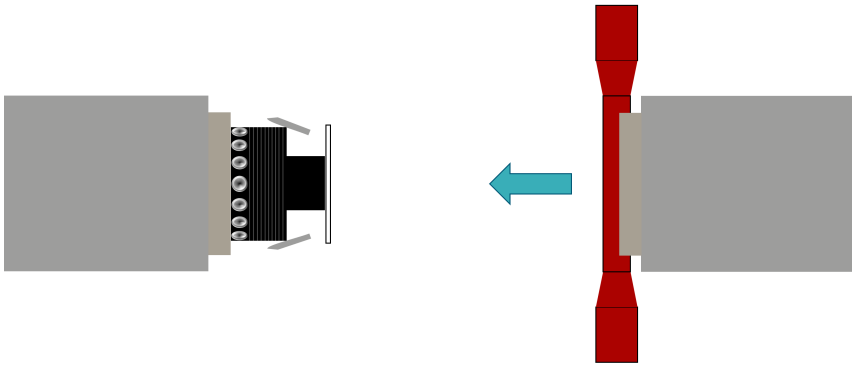
In terms of the z-direction, an indication of accuracy is given by the repeatability. This value is determined by using a flat target and calculating the 95% confidence variation of the average height over 4096 measurement frames. z values are averaged over the full FOV. The repeatability comes out to be $2\mu m$ [8].

4.3 Order of operations automated line-up

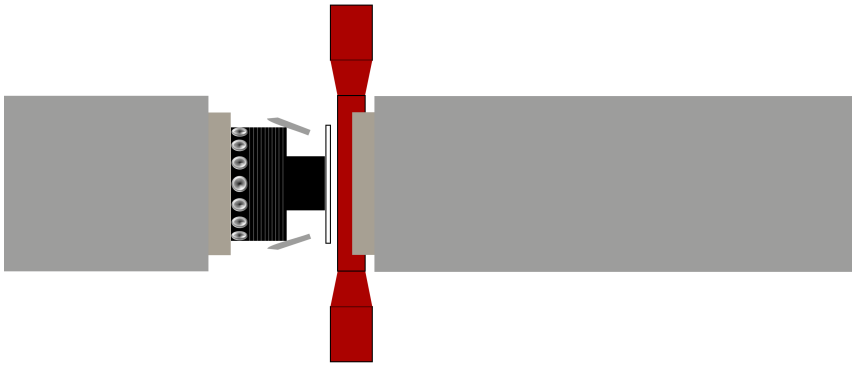
Figure 7 shows the order of operations during automated line-up.



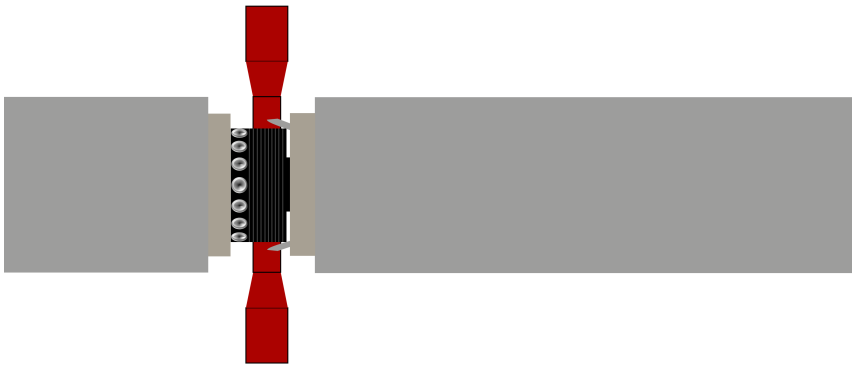
(a) Initial state of automated line up. The main line is depicted on the left, while the new joint is depicted on the right. The U-Frame is in its starting position. The ILUC sits in the main line. Visible parts of the ILUC are its pneumatic knobs, black ribbed cylinder (ILUC ribs) and white ring (ILUC ring).



(b) The U-Frame moves to the right until it encounters the new joint, which is moved towards the main line. The U-Frame reverses direction and tracks the new joint upon its recognition by the laser line scanners.



(c) The laser line scanners detect the ILUC ring. The first adjustments are made to the new joint's position. This is done to ensure that the new joint is shifted over the ILUC ring and other parts of the ILUC.



(d) The laser line scanners detect the ILUC ribs. This can be used to further adjust the new joint's position, matching more closely the orientation of the main line.

(e) Both pipe ends are fitted together during final line up. Adjustments are made to the new joint's position until a certain minimum gap tolerance is reached.

(f) U-Frame moves back to its starting position. The new joint is now fitted to the main line. The pipelines are now ready for welding.

Figure 7: Simplified order of operations during automated line-up. This procedure is repeated for each new joint that is added to the main line. After final line up in 7e is completed, the new joint is welded to the main line. Then, the whole pipeline is moved forward and the process is repeated for the next new joint.

Figure 7 offers only simplified overview of the automated line-up process. Not shown in the figure are the;

- **sub-frame**: system that controls the U-Frame's movements;
- **laser line scanners**: sensors that detect the new joint's position;
- **Line up Car Assemblies (LCAs)**: devices that precisely control the new joint's position and orientation;

4.4 Flow of line up automation software

The code flow of the laser line scanner software is shown in figure 8. This is an high-level overview of the software's operation.

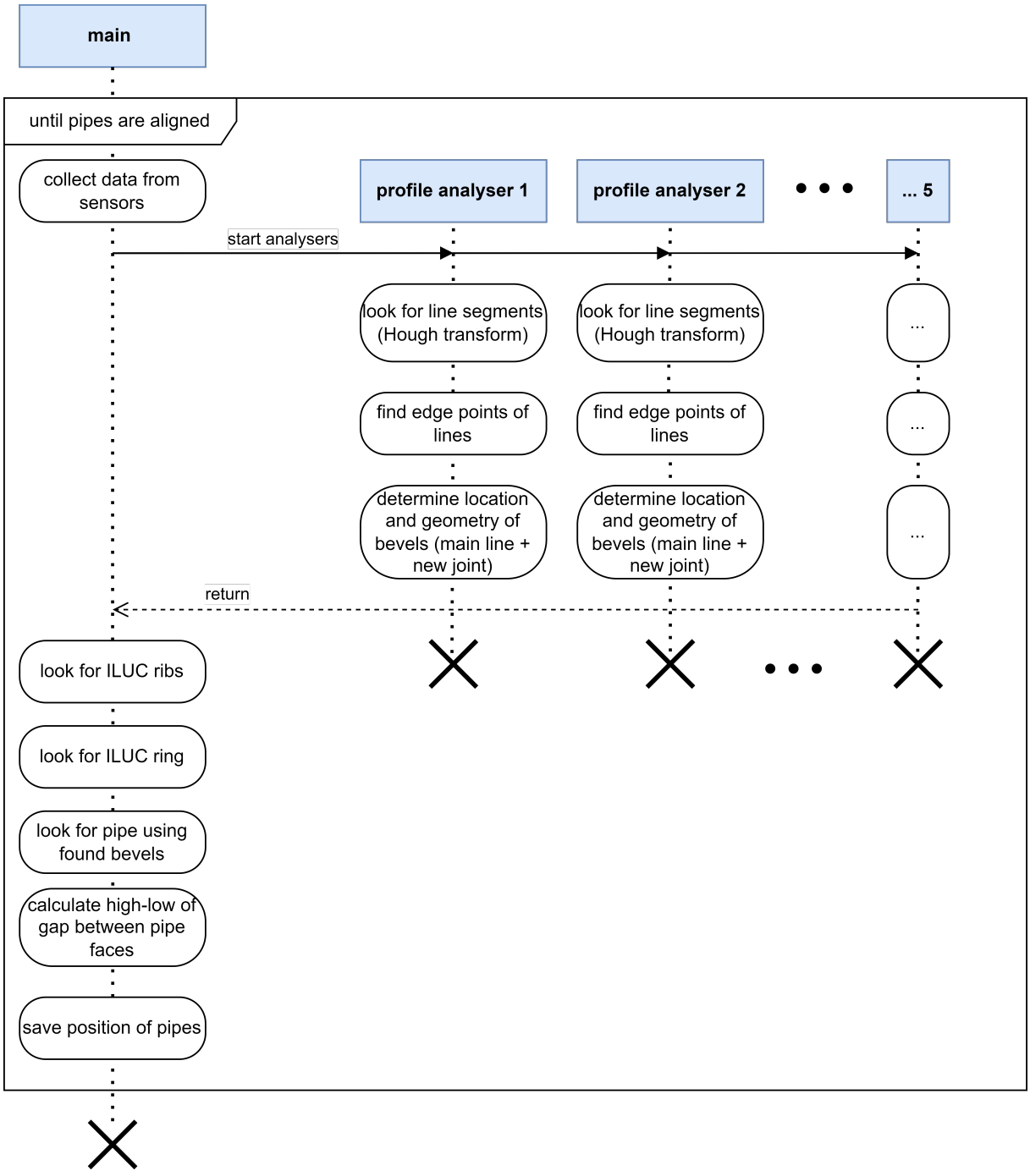


Figure 8: Flow of the laser line scanner software. The software is a multithreaded application. The blue boxes with bold text indicate the birth of these threads. The dotted lines are the 'life-lines' of the threads, which always terminate in an '×'. The frame symbolises the main loop of the software. The top-left compartment in the frame specifies the loop-condition; 'until pipes are aligned'. The rounded boxes on the life-lines indicate functions that are called. Note that only the main thread can call functions. The main thread is responsible for the initialisation of the software and the creation of the other threads.

As can be seen in figure 8, the laser line scanner software assigns a seperate thread for the analysis of the data coming from each of the five laser line scanners. These analyser

threads are responsible for finding line-segments, edge points and pipe bevels in their respective 2-dimensional data sets. Once this preliminary, disjoint analysis is done, the main thread combines the processed data into a 3-dimensional model of the ILUC ribs, -ring and/or pipes. Specifically, the models for the pipes are used to calculate the high-low values of the gap between the new joint and the main line. The high-low values determine if the pipes are lined-up correctly. If the gap is within a certain tolerance, the software will signal that the pipes are aligned and it terminates the main loop.

Comparing Figures 7 and 8, it becomes clear that the relative activity of the various parts of the software change depending on the stage of the automated line-up process. For example, determining the location and geometry of a pipe's bevel is only necessary as soon as the new joint comes into Field of View (FOV) of the laser line scanners (step 7b). Then, the activity roughly doubles as soon as the main line comes into FOV of the laser line scanners during final line up (step 7e). Furthermore, the finding of lines and their edges in each of the profile analyser threads becomes more computationally expensive for the ILUC ribs. This is further explored in section 5.

5 Profiler

This section describes the use of a profiler to identify bottlenecks in the LUA software. The profiler is used to identify functions that consume a lot of CPU time, and thus are candidates for optimization.

Microsoft Visual Studio (VS) comes with an installation of VSDiagnostics; a command-line profiling tool [7]. VSDiagnostics generates *.diagsession files, which VS interprets and visualizes. For this reason, it is more convenient to call VSDiagnostics indirectly from within Visual Studio.

To do this, debug and symbol flags need to be added to the compilation commands of object files and final linking of the main executable. Adding these flags ensures the compiler generates symbol (for Windows .pdb) files, which are required by any profiler.

The VS profiler discerns between two categories of CPU time: self-time and total time. Self-time is the time spent in a function, excluding time spent in subroutines/-functions. Total time is the time spent in a function, including time spent in subroutines/-functions [6]. The self time is expressed as

$$\text{self-time} = \frac{\text{CPU time spent in the function}}{\text{total execution time}} \times 100\%. \tag{6}$$

5.1 Profiling test setup

Profiling measurements are done in a controlled environment, where the LUA software is run with one laser line scanner. The scanner is placed in front of a test setup consisting of a lone ILUC ring, a mockup for the ILUC ribs and new joint.

(a) ILUC ring.

(b) ILUC ribs.

(c) Full setup. Note, laser is not connected

Figure 9: The experimental setup for profiling measurements.

These objects are moved in a controlled manner to simulate the phased of LUA. The experimental setup is described in Table 1.

Phase	Description	Experimental Setup
7b Joint recognition	LUA software detects joint as it moves towards mainline. U-Frame moves with joint after (successful) detection.	Flat surface resembling the joint (back of the ILUC-ribs mockup) is quickly brought into frame and subsequently kept still.
7c ILUC ring recognition	Joint and U-Frame move until ILUC ring is detected, after which partial line up takes place.	Next to the joint mockup, the side of the ILUC ring is brought into frame representing the ILUC-ring.
7d ILUC ribs recognition	The ILUC-ring moves into the joint during the aforementioned partial line-up, after which the ILUC-ribs are detected. The joint is moved over the ribs towards the mainline.	A joint mockup (a side of the ILUC-ring mockup) is still in frame. The ILUC ribs mockup is slowly moved into frame, revealing all its ribs.
7e Final line up	The mainline gets detected and the joint is lined up with it	The joint mockup and a second flat surface (mainline mockup) are slowly moved together.

Table 1: Profiling experiments simulating essentials phases of LUA. Each of the experiments are performed with one laser line scanner and done on a standard company PC (HP Z4 G4).

Every profiling test is approximately 20 seconds in length, in keeping with the profiling guidelines. Every test is preceded by a 20 second settling period during which the LUA software runs, but no profiling or setup movement takes place. The profiling sampling rate is >=1000 Hz.

5.2 Profiling results

Shown in Table 2 are the most time consuming functions per LUA phase.

Phase	Top function	self-time (%)
Joint recognition	`LineFinder::calculateNrPointsInEachBin`	13.94
ILUC ring recognition	`LineFinder::calculateNrPointsInEachBin`	17.89
ILUC ribs recognition	`LineFinder::calculateNrPointsInEachBin`	18.53
Final line up	`LineFinder::calculateNrPointsInEachBin`	15.87

Table 2: Profiling results of experiments described in Table 1.

The results show that the function `LineFinder::calculateNrPointsInEachBin` is the most time consuming function in all phases. This function is a bottleneck and is a target for optimization.

As mentioned in section 4.4, recognition of the ILUC ribs is the most time consuming phase. This can be attributed to the ILUC ribs consisting of many smaller line segments. The Hough transform function executed in each of the profile analyser threads in figure 8 is ran for each of these line segments. This in turn results in many calls to the function `LineFinder::calculateNrPointsInEachBin`, which is a part of the Hough transform.

Figure 10 shows profiling results of parts of the bottleneck function

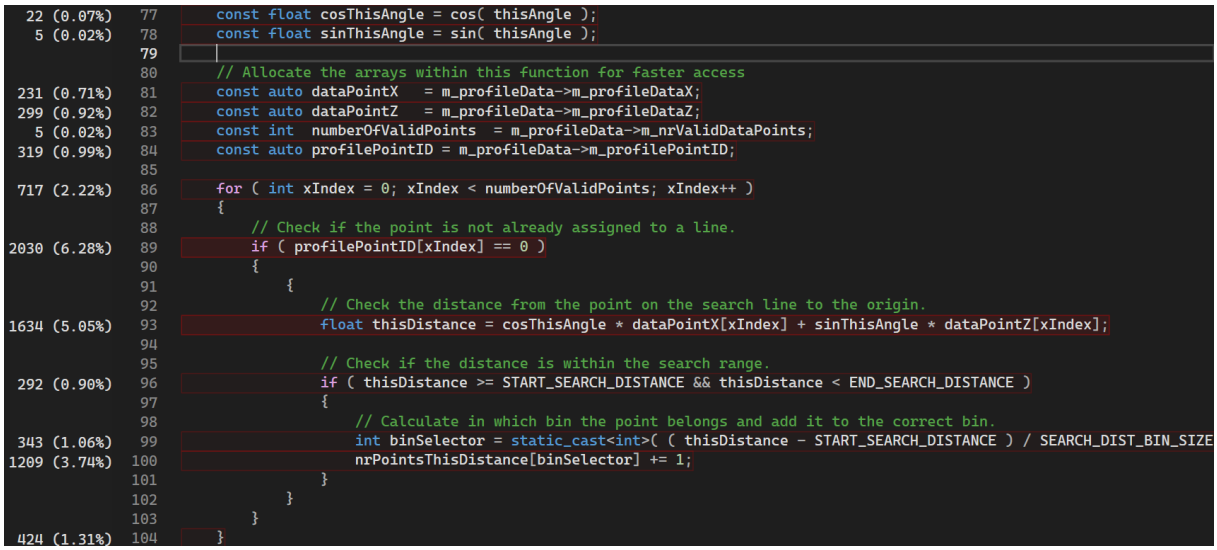


Figure 10: Profiling results of the function `LineFinder::calculateNrPointsInEachBin` during the ILUC ribs recognition phase.

From figure 10 it becomes clear what the most time consuming operations are in the calculation of the Hough transform. In order of most time consuming to least, these are:

B.N. 1 Checking if a data point is not already associated to a line (**line 89 in figure 10**);

B.N. 2 Calculating the distance between a data point and the laser line scanner's origin (**line 93 in figure 10**);

B.N. 3 Determining in which distance bin a data point falls (**line 100 in figure 10**).

6 Optimize & refactor code

In section 5.2 it is shown that the function `LineFinder::calculateNrPointsInEachBin` is the most time consuming function in all phases of the LUA software.

Before optimization can take place it is essential to understand the structure of this part of the LUA software. Algorithm 1 shows the pseudo code of the function `LineFinder::calculateNrPointsInEachBin`.

Algorithm 1 Pseudo code of the function `LineFinder::calculateNrPointsInEachBin`.

```
1:  function calculateNrPointsInEachBin(α)
2:      calculate sine and cosine of this angle α
3:      fetch X Z coordinates of the data points
4:      fetch the indices of the data points (used to determine line association)
5:      for each data point do
6:              check if data point is not already associated to a line
7:              calculate distance between data point and laser line scanner's origin
8:              determine in which distance bin a data point falls
9:      end for
10: end function
```

Algorithm 1 is nested in a loop that iterates over search angles specified for the Hough transform. This is shown in algorithm 2

Algorithm 2 Pseudo code of the function `LineFinder::calculateBinContainingMostPoints`.

```
1:  function calculateBinContainingMostPoints
2:      for each angle α in all search angles do
3:              calculateNrPointsInEachBin(α)
4:              for each distance bin in all search bins do
5:                      if bin contains more points than the current maximum then
6:                              update the current maximum
7:                      end if
8:              end for
9:      end for
10: end function
```

With regards to the order of for-loops, it is optimal to have a larger inner for-loop and a smaller outer for-loop [5, lecture 7, "temporal locality"]. Algorithms 1 and 2 show the nested for-loop structure that is used to perform the Hough transform. Note that the number of search angles is approximately 30, while the number of data points is approximately 4000. This means that the inner for-loop is much larger than the outer for-loop. Hence, the order of the for-loops is already optimal.

Closer inspection of algorithm 1 reveals that the fetching operations in lines 2 and 3 do not depend on the angle α. This means that these fetching operations can be moved outside the for-loop to the beginning of algorithm 2. This ultimately results in less memory calls happening in the inner for-loop. Furtermore, the calculation of the sine and cosine can be done once for all angles in the main thread of the LUA software before the main program loop (figure 8).

Figure 11 shows the profiling results of the implemented prefetching and precalculation optimizations.

```
                112  {
  9 (0.02%)     113      const int numberOfValidPoints = m_profileData->m_nrValidDataPoints;
                114
                115      // Create bins for all distances to keep track of how many datapoints fall into each interval
 19 (0.04%)     116      std::array<unsigned, NUM_ROUGH_RADII> nrPointsThisDistance = { 0 };
                117
965 (2.09%)     118      for ( int xIndex = 0; xIndex < numberOfValidPoints; xIndex++ )
                119      {
                120          // Check if the point is not already assigned to a line.
2780 (6.02%)    121          if ( profilePointID[xIndex] == 0 ) // line association check
                122          {
                123              {
                124                  // Check the distance from the point on the search line to the origin.
2552 (5.53%)    125                  float thisDistance = cosine * dataPointX[xIndex] + sine * dataPointZ[xIndex]; // distance calculation
                126
                127                  // Check if the distance is within the search range.
 370 (0.80%)    128                  if ( thisDistance >= START_SEARCH_DISTANCE && thisDistance < END_SEARCH_DISTANCE )
                129                  {
                130                      // Calculate in which bin the point belongs and add it to the correct bin.
 749 (1.62%)    131                      int binSelector = static_cast<int>( ( thisDistance - START_SEARCH_DISTANCE )
                132                                                          / SEARCH_DIST_BIN_SIZE ); // bin selection
1625 (3.52%)    133                      nrPointsThisDistance[binSelector] += 1;                       // bin association
                134                  }
                135              }
                136          }
 746 (1.62%)    137      }
 755 (1.64%)    138      return nrPointsThisDistance;
                139  }
```

(a) Profiling results of the function `LineFinder::calculateNrPointsInEachBin` (algorithm 1) after precalculation optimization.

```
                 51      // prefetch arrays & vectors before going into loop over angles
                 52      const std::array<float, NUM_ROUGH_ANGLES> angles  = m_analyserSettings->roughAngles;
                 53      const std::array<float, NUM_ROUGH_ANGLES> cosines = m_analyserSettings->cosines;
                 54      const std::array<float, NUM_ROUGH_ANGLES> sines   = m_analyserSettings->sines;
   34 (0.07%)    55      const std::vector<float> dataPointX( m_profileData->m_profileDataX.cbegin(), m_profileData->m_profileDataX.cend() );
   28 (0.06%)    56      const std::vector<float> dataPointZ( m_profileData->m_profileDataZ.cbegin(), m_profileData->m_profileDataZ.cend() );
   38 (0.08%)    57      const std::vector<int>   profilePointID( m_profileData->m_profilePointID.cbegin(),
                 58                                               m_profileData->m_profilePointID.cend() );
                 59
                 60      BestRoughLineFit bestFit;
                 61
                 62      for ( unsigned angleNr = 0; angleNr < NUM_ROUGH_ANGLES; angleNr++ )
                 63      {
16155 (35.00%)   64          auto tempBestFit = calculateDistanceIntervalContainingMostPoints(
                 65              angles[angleNr], cosines[angleNr], sines[angleNr], dataPointX, dataPointZ, profilePointID );
                 66
                 67          // Use the angle with the first best result
                 68          if ( tempBestFit.nrOfPointsFromCombinedBins > bestFit.nrOfPointsFromCombinedBins )
                 69          {
                 70              bestFit = tempBestFit;
                 71          }
                 72      }
                 73
   24 (0.05%)    74      return bestFit;
                 75  }
                 76
```

(b) Profiling results of the function `LineFinder::calculateBinContainingMostPoints` (algorithm 2) after prefetching optimization.

Figure 11: Profiling results of optimized bottleneck functions. Calculation of the sinusoidal functions is done during initialisation of the LUA software. Data arrays are prefetched before before the inner for-loop.

Comparing lines 55-57 from figure 11b with lines 81-84 from figure 10 shows that both the percentage of CPU time and the number of times these lines are sampled by the profiler are lower in the optimized case. This is a clear indication that the optimizations have been successful.

Further optimizations can be made by avoiding the most time consuming operation in the inner for-loop Bottle Neck 1. In general an if-statement in a for-loop causes a 'pipeline stall' [5, lecture 7, "Pipeline stall", slide 19]. This is because the CPU must wait until the if-statement is evaluated before it can continue executing the next instruction. This effect is made worse by the fact that the if-statement is in the inner most nested for-loop. The solution to this problem is to use a data structure that allows for fast look-up of data points that are already associated to a line. Perhaps this can be done by using a hash table. Alternatively, algorithmic adjustments can be made to avoid the if-statement entirely. In any case, this is left as a topic for future work.

7 Accuracy of circle fits

This section gives a mathematical estimate for the accuracy of circle fit algorithms that minimize geometric distances. This is useful knowlegde for LUA, since this kind of 'geometric circle fit' algorithm is used to form a 3-dimensional representation of the pipes, among other things (section 4.4). This discussion then ends with the implementation in the LUA software of a new kind of circle fit algrorithm that has a variance that is the same as the least squares method, but no bias in its estimation of the circle radius. This section builds on the methods described in the paper by Al-sharadqah and Chernov [2].

7.1 Statistical model

In order to estimate the accuracy of a circle fit, we need to define a statistical model for the error in the input. To that end, we assume that the data points (y_i, z_i) are noisy observations of the true points $(\tilde{y}_i, \tilde{z}_i)$, where the noise is i.i.d. normal random variables with bias 0 and variance s^2 [2, section 2] such that

$$y_i = \tilde{y}_i + \delta y_i, \quad z_i = \tilde{z}_i + \delta z_i, \quad \delta y_i, \delta z_i \sim N(0, s^2). \tag{7}$$

This gives that the true points must lie on a circle, satisfying the equation

$$(\tilde{y}_i - \tilde{a})^2 + (\tilde{z}_i - \tilde{b})^2 = \tilde{R}^2 \quad \text{for } i = 1, \ldots, n, \text{ midpoint } (\tilde{a}, \tilde{b}) \text{ and radius } \tilde{R}. \tag{8}$$

In the case of LUA the angles of the laser line scanners in the U-frame are known (section 4.2). This suggests normalisation of the data points to the unit circle, which is done by

$$u_i = \frac{\tilde{y}_i - \tilde{a}}{\tilde{R}} = \cos\phi_i, \quad v_i = \frac{\tilde{z}_i - \tilde{b}}{\tilde{R}} = \sin\phi_i, \tag{9}$$

where ϕ_i is the angle corresponding to the true point $(\tilde{y}_i, \tilde{z}_i)$ projected on the pipe (or ILUC ring/ -ribs) by the i^{th} laser line scanner. This leads to the normalised data matrix W with rows $W_i = (u_i, v_i, 1)^T$.

Define the vector of estimates $Q = (a, b, r)^T$ for the circle parameters. Then, let the first order terms in the Taylor expansion of Q in terms of $\delta y_i, \delta z_i$ be denoted by $\Delta_1 Q$. The accuracy of the circle fit can then be estimated by the variance of $\Delta_1 Q$ [2, section 2].

7.2 Accuracy estimation

The geometric circle fit has both bias and (co-)variance. under the assumption that the number of data points approaches infinity $n \to \infty$, the bias is given by

$$\mathrm{E}[\Delta_1 Q] = \frac{2s^2}{\tilde{R}} \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \quad \text{[2, equation 6.14]}, \tag{10}$$

and therefore the bias exists only in the radius estimate as $\mathrm{E}[\Delta_1 R] = \frac{2s^2}{\tilde{R}}$. The covariance is given by

$$\mathrm{cov}[\Delta_1 Q] = s^2 (W^T W)^{-1} \quad \text{[2, equation 8.12]}. \tag{11}$$

Taking the diagonal elements of the covariance matrix gives the variance of the estimates for the circle parameters. Taking the square root of these variances gives the standard deviation $s(\cdot)$ of the estimates. The latter is good indicator of the accuracy of the circle fit.

It remains to estimate the noise level s in the data and the true angles ϕ_i for all i. From section 4.2 it is known that the resolution of the laser line scanners is $47\mu m$ to $104\mu m$ in the

x-direction and 2μ in the z-direction. This suggests that the noise level s is on the order of 10^{-4} m. This is a rough estimate, but it is sufficient for the purposes of this discussion. As stated in the previous section 7.1, the true angles correspond to the angle of the laser line scanner in the U-frame.

Using the above circle fit accuracy can be calculated for the idealised scenario where a 24-inch (= 0.5 m) pipe is exactly in the center of the U-frame with zero pitch or yaw. In this case, the angles of the true poins are the same as the angles of the laser line scanners in equation 5.

Substituting this into eqs. (10) and (11) above gives

$$\begin{aligned} \mathrm{E}[\Delta_1 R] &= 0.04\mu\mathrm{m}, \\ s[a] &= \sqrt{(\mathrm{cov}[\Delta_1 Q])_{11}} = 0.05 \text{ mm}, \\ s[b] &= \sqrt{(\mathrm{cov}[\Delta_1 Q])_{22}} = 0.09 \text{ mm}, \\ s[R] &= \sqrt{(\mathrm{cov}[\Delta_1 Q])_{33}} = 0.05 \text{ mm}. \end{aligned}$$

Hence, the accuracy of the geometric circle fit is given by

$$\epsilon_{\text{midpoint}} = \sqrt{s[a]^2 + s[b]^2} = 0.11 \text{ mm}, \quad \epsilon_{\text{radius}} = s[R] \approx 0.05 \text{ mm}. \tag{12}$$

Interestingly, the standard deviation of the midpoint estimate is larger in the z-direction than the y-direction. This can be attributed to the positioning of the laser line scanners in the U-frame; two on each side of the pipe near the y-plane and only one directly below in the z-plane. This suggests that the accuracy of the midpoint estimate can be improved upon by adding more laser line scanners in the z-direction.

7.3 Implementation of a new circle fit algorithm

The above circle fit algorithm has a bias in the radius estimate. This is not ideal for the LUA system, since the radius of the pipe is a critical estimate in determining sufficient line-up (section 4.3 and 4.4). Therefore, a new circle fit algorithm was implemented in the LUA software that has the same variance as the least squares method, but no bias in the radius estimate. This is done by using the method of Al-sharadqah and Chernov termed 'Hyper accurate algebraic circle fit' [2]. Additionally, the old code was refactored to improve readability and maintainability. The hyper accurate fit is implemented in code listing 1 and it is added to the function `PipeFaceFinder::findCircleNPoints` which is used in the LUA software to fit pipes in 3D, listing 2.

8 ILUC Recognition

This section describes a novel algorithm that allows the LUA software to recognize the ILUC. The algorithm combines the lines and edges coming from the profile analysers to form a 3-dimensional representation of the ILUC, similar to how the pipes are already found. The goal of the algorithm is to find the ILUC's direction, radius and centre. For convenience, these values together are termed the *ILUC face*.

The motivation to recognise the ILUC face lies in the ability to perform an accurate pre-line up of the new joint and main line. The new joint's position and orientation can be preemptively adjusted to the ILUC face during line up (see 7d). This ultimately leads to a more accurate and efficient line up of the new joint and main line, since the ILUC is always centered and fixed in the main line.

The ILUC consists of ten ribs (see figs. 4 and 7), each of which is recognised by the profile analysers as a line segment of a specific length (~ 1 cm). In turn, each of these line segments generates two edge points. Consequently, all five profile analysers together can generate up to

$$\begin{aligned} &5 \text{ profile analyser threads} \\ \times &10 \text{ ILUC ribs per profile analyser} \\ \times &2 \text{ ILUC rib edge point per ILUC rib} \\ = &100 \text{ ILUC rib edge points.} \end{aligned}$$

These edge points are used in algorithm 3 to obtain an estimate of the ILUC face.

Algorithm 3 Pseudo code for ILUC recognition.

```
function: multiSensorAnalyser_c::checkForILUCFace
Obtain ILUC rib edge points from the profile analysers
If there are more than six edge points then
    Run multiSensorAnalyser_c::analyseILUCRibs (see algorithm 4)
    Store the ILUC face
    If an ILUC face is found then
        Run multiSensorAnalyser_c::fitILUCCylinder (see algorithm 5)
        If the optimised ILUC face has a lower error then
            Overwrite the estimated ILUC face with the optimised ILUC face
        end if
    end if
end if
output: ILUC face
```

8.1 ILUC direction estimation

The problem of estimating the ILUC face can be solved by combininh circle fits on the ILUC rib edges. The midpoints of these ILUC rib edge circle fits can be used to estimate the ILUC direction. The algorithm is described in algorithm 4.

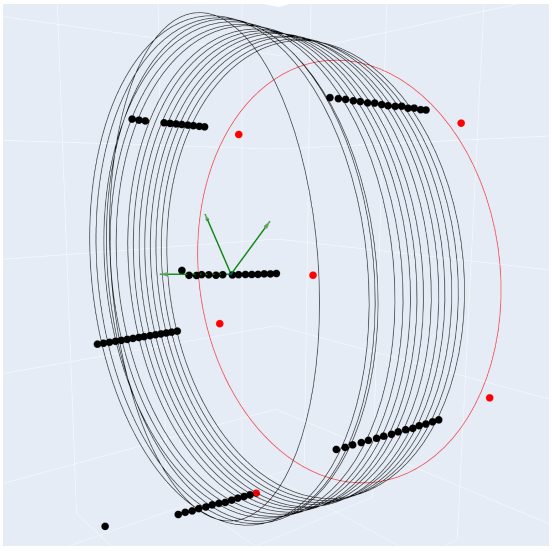
Algorithm 4 Pseudo code for ILUC face estimation.

1: **function** `multiSensorAnalyser_c::analyseILUCRibs`
2: **input** ILUC rib edge points per profile analyser
3: Naively group ILUC rib edge points into ILUC rib edges
4: Fit circles to the ILUC rib edges, yielding ILUC rib midpoints
5: Remove outliers from the set of ILUC rib midpoints
6: Get an approximate ILUC direction by fitting a line through ILUC rib edge midpoints
7: **If** an approximate ILUC direction is found **then**
8: Use approximate ILUC direction to correctly match ILUC rib edge points into edges
9: Fit circles to the ILUC rib edges, yielding ILUC rib midpoints
10: Remove outliers from the set of ILUC rib midpoints
11: Get a corrected ILUC direction by fitting a line through ILUC rib edge midpoints
12: Use corrected ILUC direction to perform a final circle fit, yielding the ILUC centre and radius and completing the ILUC face estimation
13: Calculate average error of the ILUC face
14: **end if**
15: **output:** ILUC face, ILUC rib edge points without outliers, average error

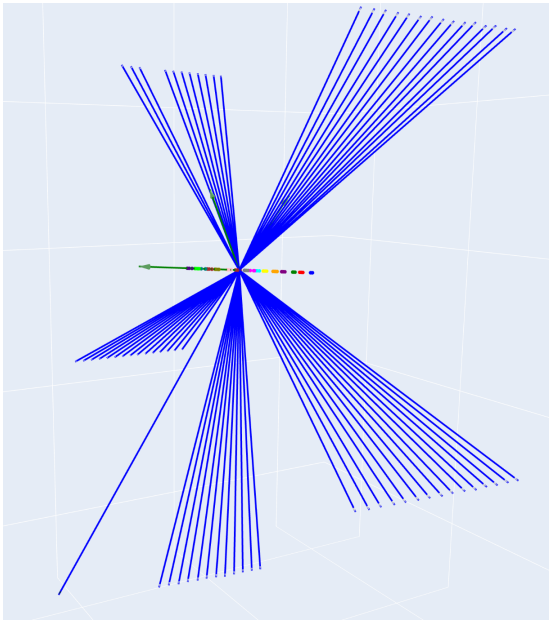
In line 3 of algorithm 4 the ILUC rib edge points are grouped into ILUC rib edges based on their index in each of the profile analyser threads. This approach is naive, because it does not consider the possibility of a missing edge point. Missing edge points can occur due to a profile analyser not detecting an ILUC rib edge. There are multiple reasons a profile analyser might miss an edge, which are left out of the scope of this discussion.

Mismatched ILUC rib edges result in slanted circles. The midpoints resulting from the possibly mismatched set of ILUC rib edge points are used to obtain a first estimate of the ILUC direction. This estimate is then used to correctly match the ILUC rib edge points into edges. This allows for the correction of the ILUC direction. Finally, a circle fit is performed on all ILUC rib edge points (without outliers). This yields the ILUC centre and radius, completing estimation of the ILUC face.

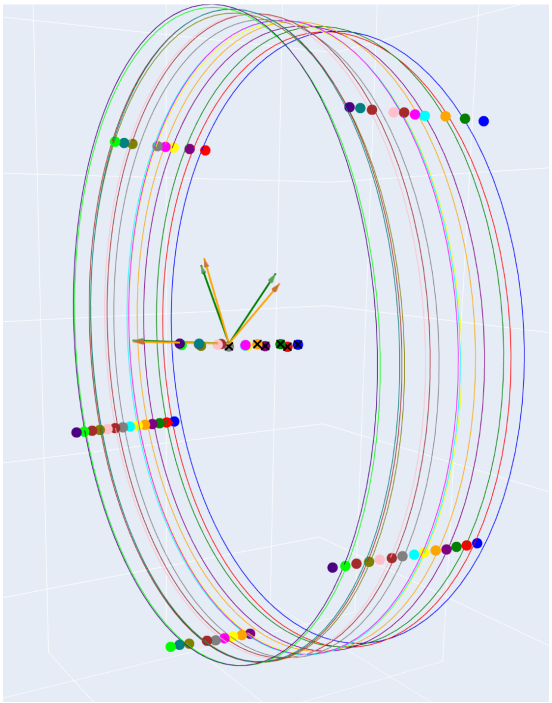
A visual representation of the steps of algorithm 4 is shown in figure 12.



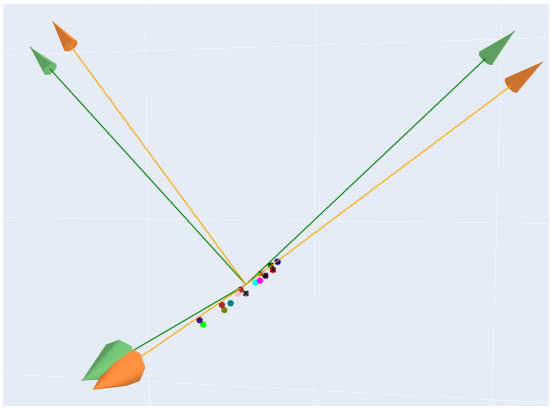
(a) Lines 3 to 6 of algorithm 4. Shown are the ILUC rib edge points grouped into possibly mismatched ILUC rib edges. The mismatched ILUC rib edges result in slanted circles. The midpoints of these circles are used to estimate the ILUC direction (green vector). One set of ILUC rib mid- and edge points are marked in red, since these points were determined to be outliers.



(b) Line 8 from algorithm 4. Using the approximate direction from figure 12a, the ILUC rib edge points are correctly matched into edges. The blue lines are vectors eminating from the centroid of the ILUC rib circle midpoints to each of the ILUC rib egde points. These vectors are projected onto the approximate ILUC direction. The ILUC rib edge points are then matched into ILUC edges based on the length of the projections. Each color represents a different ILUC rib edge.



(c) Line 9 from algorithm 4. Circles are fit to the correctly matched ILUC rib edges from figure 12b.



(d) Lines 10 and 11 from algorithm 4. The ILUC rib midpoints from figure 12c are filtered for outliers (marked with '$\times$' symbol) and used to obtain a corrected ILUC direction (orange vector). The difference between the corrected and approximate ILUC direction is approximately $0.7°$.

Figure 12: Visualisation of algorithm 4.

8.2 ILUC direction optimisation

The ILUC direction obtained from algorithm 4 can be further optimised by fitting a cylinder to the ILUC rib edge points. The algorithm is described in algorithm 5.

Algorithm 5 Pseudo code for ILUC face optimisation.

function `multiSensorAnalyser_c::fitILUCCylinder`
input ILUC face estimate, ILUC rib edge points without outliers
Use ILUC face estimate to get initial parameters for the cylinder fit y_c, z_c, β, γ, r
while a stopping criterion is not met **do**
    Calculate right hand side $\mathbf{\Delta y}$
    Calculate Jacobian $\mathbf{J}$
    Calculate $\mathbf{C} = \mathbf{J}^T\mathbf{J}$
    Calculate condition number $\kappa(\mathbf{C})$
    Solve $\mathbf{C\Delta p} = \mathbf{J}^T\mathbf{\Delta y}$ for $\mathbf{\Delta x} \leftarrow$ Use e.g. Cholesky decomposition, since C is a relatively small (5x5), symmetric matrix
    Update parameter vector $\mathbf{p} = \mathbf{p} - \mathbf{\Delta x}$
    Calculate loss l_{i+1} and error e of the updated cylinder
    Calculate convergence ratio $\rho = \frac{L_i - L_{i+1}}{L_i}$
    Update loss $L_i = L_{i+1}$
    Update iteration counter $i = i + 1$
end while
Convert cylinder parameters to ILUC face; direction, centre and radius
Calculate average error of the ILUC face
output: Optimised ILUC face, average error

The 'stopping criterion' in algorithm 5 refers to one of the following conditions:

- The maximum number of iterations is reached: $i > i_{max} = 100$.
- The error becomes smaller than a certain threshold: $e < e_{\text{min}} = 5 \times 10^{-4} = 0.5\text{mm}$.
- The perdentual change of the loss function is smaller than a certain threshold: $\rho < \rho_{\text{min}} = 10^{-4} = 0.01\%$.
- The loss and/or error becomes NaN.
- The condition number of the Jacobian matrix is too high: $\kappa > \kappa_{\text{max}} = 10^6$. This indicates that the routine has either converged to a minimum, is stuck in a local minimum or is stuck on a flat region of the loss function.

The values of the thresholds are chosen based on the expected accuracy of the ILUC face and the computational resources available. The thresholds are chosen such that the optimisation routine converges within a reasonable amount of time and the ILUC face is accurate enough for the purposes of the LUA software.

8.3 Cylinder fit model

The algorithm 5 is based on a non-linear least squares formulation of fitting a cylinder centered around the x-axis. The model equation for a cylinder of this kind is given by

$$f(\vec{x}, \vec{p}) = r^2 - Y^2 - Z^2, \tag{13}$$

where r is the radius of the cylinder, Y and Z are the coordinates of the ILUC rib edge points in the y and z direction respectively and $\vec{p} = (y_c, z_c, \beta, \gamma, r)$ are the parameters of the cylinder. The

parameters y_c and z_c are the coordinates of the cylinder centre in the y, z-plane, β and γ are the angles of the cylinder with respect to the y and z axes respectively. The model for the cylinder is shown in figure 13.

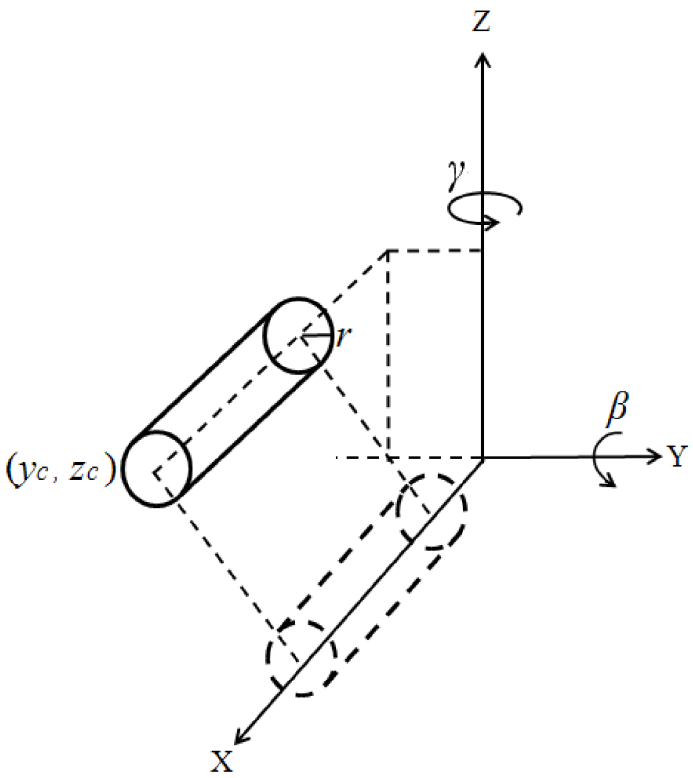


Figure 13: Model of the cylinder used in algorithm 5. The cylinder is centered around the x-axis and has radius r. The coordinates of the cylinder centre y_c and z_c in the y and z direction respectively. The cylinder is rotated by angles β and γ with respect to the y and z axes respectively [3, figure 2].

The variables Y, Z in equation 13 represent the coordinates of the ILUC rib edge points in the cylinder's frame of reference and are given by

$$\begin{pmatrix} X \\ Y \\ Z \end{pmatrix} = R_y(\beta)R_z(\gamma) \begin{pmatrix} x \\ y - y_c \\ z - z_c \end{pmatrix}, \tag{14}$$

where $R_y(\beta)$ and $R_z(\gamma)$ are rotation matrices around the y and z axes and $\vec{x} = (x, y, z)^T$ is the coordinate of an ILUC rib edge point respectively [3, equation 5]. The residual function of the model is given by

$$R(\vec{x}, \vec{p}) = -f(\vec{x}, \vec{p}) = -(r^2 - Y^2 - Z^2). \tag{15}$$

The loss function is defined as the sum of the squared residuals

$$L = \sum_{j=1}^{N} R_j^2, \tag{16}$$

where N is the number of ILUC rib edge points. The loss function is minimised by adjusting the parameters $\vec{p}$ of the cylinder. This is done by solving the normal equations

$$\mathbf{J}^T\mathbf{J}\boldsymbol{\Delta}\mathbf{p} = \mathbf{J}^T\boldsymbol{\Delta}\mathbf{y}, \tag{17}$$

where $\mathbf{J} \in \mathbb{R}^{(N\times 5)}$ is the Jacobian matrix of the residuals of the ILUC rib edge points with respect to the parameters $\vec{p}$ and $\mathbf{\Delta y} \in \mathbb{R}^N$ is the right hand side of the normal equations. The Jacobian matrix is given by

$$\mathbf{J}_{jk} = \frac{\partial R(\vec{x}_j, \vec{p})}{\partial p_k}, \tag{18}$$

where j is the index of the ILUC rib edge point and k is the index of the parameter. Then, the right hand side of the normal equations is given by

$$\mathbf{\Delta y}_j = R(\vec{x}_j, \vec{p}). \tag{19}$$

A major assumption of this model is that the cylinder is close to the x-axis. This is in fact the case for the ILUC in the coordinate system of the LUA software.

Furthemore, the initial parameters cannot be too far off from the minimizing parameters, because algorithm 5 is iterative. A bad initial guess can lead to slow (sub-quadratic) convergence, local minima or divergence. However, the initial parameters of the cylinder are obtained from the ILUC face estimate resulting from algorithm 4. Hence, the initial parameters are already close to the minimizing parameters.

8.4 ILUC face accuracy

Table 3 shows the accuracy of the ILUC face recognition algorithm for various data sets of the ILUC. 'stationary' refers to the ILUC being stationary, while 'oscillating (XY)' and 'oscillating (XZ)' refer to the ILUC oscillating in the y-z plane and x-z plane respectively. The table shows that the ILUC face recognition algorithm is able to accurately estimate the ILUC face for all data sets.

Data Set	**ILUC Face Recognition %**	**Average Error (mm)**	**Optimise Success %**	**Optimised Average Error (mm)**	**Improvement %**
stationary	100 %	0.699	38.8%	0.215	33.5%
oscillating (XY)	85.8%	0.806	34.4%	0.172	23.5%
oscillating (XZ)	87.7%	1.09	41.7%	0.238	28.3%

Table 3: ILUC Recognition Accuracy. **ILUC Face Recognition %** is the percentage of ILUC faces correctly recognised. **Average Error (mm)** is the average error of the ILUC face recognition. **Optimise Success %** is the percentage of ILUC faces successfully optimised. **Optimised Average Error (mm)** is the average error of the optimised ILUC face recognition. **Improvement %** is percentage of times the optimisation routine resulted in a smaller average error.

9 Conclusion

In conclusion, the proposed approach to improve the automated line-up (LUA) system has been successfully implemented and tested. The following key milestones were achieved:

- **Create a profiler for the code to search for slow operations:** A profiler was developed to identify bottlenecks in the LUA software. The profiler provided valuable insights into the performance of different parts of the code, allowing for comparison between different versions. This enabled the identification of the most time-consuming functions, such as `LineFinder::calculateNrPointsInEachBin`, as discussed in section 5 and shown in Figure 10.
- **Find more efficient ways to perform the needed operations:** By focusing on the identified bottlenecks, several optimizations were made to the code. These optimizations included prefetching data and precalculating values, which significantly increased the overall code speed. The profiler was used to quantify these improvements, as detailed in section 6 and illustrated in Figure 11.
- **Isolate parts of the mathematical model and test accuracy:** The accuracy of the mathematical model was tested both mathematically and experimentally. The new circle fit algorithm, which has no bias in the radius estimate, was implemented and tested. The results showed that the new algorithm met the 0.1 mm accuracy requirement, as described in section 7.
- **Expand the fitting model to increase the workability of the complete system:** The fitting model was expanded to recognize the ribbed cylinder of the ILUC. This allowed for better pre-line-up of the pipes, improving the overall efficiency of the system. The successful recognition of the ILUC ribbed cylinder is discussed in section 8 and shown in Figure 4.

Overall, the improvements made to the LUA system have resulted in increased accuracy, efficiency, and maintainability. The successful implementation of the proposed approach demonstrates the potential for further advancements in automated line-up technology.

10 Skills learned

Throughout the course of this project, several key skills were developed and honed. These skills not only contributed to the successful completion of the project but also provided valuable experience for future endeavors. The following sections outline the primary skills learned:

10.1 Profiling and performance optimization

One of the critical skills acquired was the ability to profile and optimize code. By developing and utilizing a profiler, I was able to identify bottlenecks in the LUA system. This process involved:

- Understanding how to use profiling tools to gather performance data.
- Analyzing the collected data to pinpoint slow operations.
- Implementing optimizations such as prefetching data and precalculating values to enhance performance.

These skills are essential for improving the efficiency of any software system.

10.2 Mathematical modeling and accuracy testing

Another significant skill developed was the ability to create and test mathematical models. This included:

- Isolating parts of the mathematical model to test their accuracy.
- Conducting both mathematical and experimental tests to ensure the model met the required accuracy standards.
- Implementing algorithms with no bias in estimates, such as the new circle fit algorithm.

These skills are crucial for ensuring the reliability and precision of computational models.

10.3 Advanced algorithm development

The project also provided experience in developing advanced algorithms. Key aspects included:

- Expanding the fitting model to recognize complex shapes, such as the ribbed cylinder of the ILUC.
- Ensuring the algorithms were efficient and could be integrated seamlessly into the existing system.
- Testing the algorithms to verify their effectiveness in real-world scenarios.

Developing these algorithms enhanced problem-solving abilities and technical expertise.

10.4 Project management and collaboration

Finally, the project required effective project management and collaboration skills. This involved:

- Planning and organizing tasks to ensure timely completion of milestones.
- Collaborating with team members to share knowledge and solve problems collectively.

- Documenting progress and results to maintain clear communication and facilitate future work.

These skills are vital for the successful execution of any complex project.

In summary, the skills learned during this project encompass a wide range of technical and soft skills, all of which are invaluable for future projects and professional growth.

Code

Listing 1: Implementation of the new circle fit algorithm in the LUA software.

```
Vector2f PipeFaceFinder::findCircleSVDHyper(
    const Eigen::VectorXf& X,
    const Eigen::VectorXf& Y,
    float& radius )
{
    Vector2f center;
    int     n = X.rows();
    VectorXf R = X.array() * X.array() + Y.array() * Y.array();
    MatrixXf D( n, 4 );
    D.col( 0 ) = R;
    D.col( 1 ) = X;
    D.col( 2 ) = Y;
    D.col( 3 ).setOnes();

    JacobiSVD<MatrixXf> svd( D, ComputeThinU | ComputeThinV );
    Vector4f         A;
    if ( svd.singularValues()( 3 ) / svd.singularValues()( 0 ) < 1e-12 ) // singular
    {
        A = svd.matrixV().col( 3 );
    }
    else // regular
    {
        DiagonalMatrix<float, 4> S( svd.singularValues() );
        Matrix4f             N, W, Eigvecs;
        Vector4f             mean, Eigvals, Astar;

        mean = D.colwise().mean();
        N << 8 * mean( 0 ), 4 * mean( 1 ), 4 * mean( 2 ), 2, //
            4 * mean( 1 ), 1, 0, 0,                   //
            4 * mean( 2 ), 0, 1, 0,                   //
            2, 0, 0, 0;
        W = svd.matrixV() * S * svd.matrixV().transpose();

        EigenSolver<MatrixXf> es;
        es.compute( W * N.inverse() * W );
        // taking the real part, since return type is complex
        Eigvals = es.eigenvalues().real();
        Eigvecs = es.eigenvectors().real();

        Vector4i index = Vector4i::LinSpaced( 0, 4 );
        std::stable_sort(
            index.begin(),
            index.end(),
            [&Eigvals]( size_t i1, size_t i2 ) { return Eigvals[i1] < Eigvals[i2]; }
        );

        Astar = Eigvecs.col( index( 1 ) );

        A = W.ldlt().solve( Astar );
    }

    center = -A( Eigen::seq( 1, 2 ) ) / ( 2 * A( 0 ) );

    radius = std::sqrt(
        A( 1 ) * A( 1 ) + A( 2 ) * A( 2 ) - 4 * A( 0 ) * A( 3 ) ) / ( 2 * std::abs( A( 0 ) )
    );

    return center;
```

```
    }
```

Listing 2: Refactored pipe face finder.

```
    pipeFace_s PipeFaceFinder::findCircleNPoints( const std::vector<Vector3f>& points,
                                        std::vector<float>&      pointsDisToPlane,
                                        std::vector<float>&      pointsDisToCircle )
{
    // ======================= //
    // FIND BEST FITTING PLANE //
    // ======================= //
    int nrOfPoints = points.size();

    // convert points to matrix
    MatrixXf pointsMatrix( nrOfPoints, 3 );
    pointsMatrix = Eigen::Map<const Eigen::Matrix<float, Eigen::Dynamic, 3, Eigen::RowMajor>>(
        reinterpret_cast<const float*>( points.data() ), nrOfPoints, 3 );

    // translate coordinates to centroid of data
    Vector3f centroid = calculateCentroid( pointsMatrix );
    pointsMatrix.rowwise() -= centroid.transpose();

    // plane fitting algorithm
    Vector3f normal;
    normal = fitPlaneNR( pointsMatrix );

    // ======================== //
    // PROJECT POINTS ONTO PLANE //
    // ======================== //
    MatrixXf projectedPoints( nrOfPoints, 3 );
    projectedPoints = projectPointsOntoPlane( pointsMatrix, normal );

    // calculate distance of points to the plane
    pointsDisToPlane.clear();
    MatrixXf pointsDisToPlaneVectors = projectedPoints - pointsMatrix;
    VectorXf pointsDisToPlaneVector = pointsDisToPlaneVectors.rowwise().norm();

    // transform to plane coordinates
    MatrixXf planeBasis = spanPlane( normal );
    MatrixXf projectedPointsInPlaneCoordinates( nrOfPoints, 2 );
    projectedPointsInPlaneCoordinates = translateCoordinates( projectedPoints, planeBasis );

    // ============= //
    // FIND CIRCLE //
    // ============= //
    pipeFace_s tempPipeFace;

    // circle finding algorithm
    Vector2f circleCentre;
    if ( nrOfPoints == 3 )
    {
        circleCentre = findCircleLSTSQ( projectedPointsInPlaneCoordinates.col( 0 ),
                                        projectedPointsInPlaneCoordinates.col( 1 ),
                                        tempPipeFace.radius );
    }
    else
    {
        circleCentre = findCircleSVDHyper( projectedPointsInPlaneCoordinates.col( 0 ),
                                           projectedPointsInPlaneCoordinates.col( 1 ),
                                           tempPipeFace.radius );
    }
```

```
    // calculate distance to the circle
    pointsDisToCircle.clear();
    MatrixXf pointsDisToCirclecCentreVectors( nrOfPoints, 2 );
    pointsDisToCirclecCentreVectors =
        projectedPointsInPlaneCoordinates.rowwise() - circleCentre.transpose();
    VectorXf pointsDisToCircleVector = pointsDisToCirclecCentreVectors.rowwise().norm();
    pointsDisToCircleVector.array() -= tempPipeFace.radius;

    // ================================ //
    // TRANSFORM BACK TO THE VESSEL //
    // ================================ //
    // translate back to lab frame
    Vector3f circleCentreGlobal =
        centroid + circleCentre( 0 ) * planeBasis.col( 0 )
        + circleCentre( 1 ) * planeBasis.col( 1 );

    // ============= //
    // STORE RESULTS //
    // ============= //
    std::copy(
        circleCentreGlobal.data(), circleCentreGlobal.data() + 3, begin( tempPipeFace.centrePos )
    );
    std::copy( normal.data(), normal.data() + 3, begin( tempPipeFace.centreDirVec ) );
    std::copy( pointsDisToPlaneVector.data(),
              pointsDisToPlaneVector.data() + nrOfPoints,
              std::back_inserter( pointsDisToPlane ) );
    std::copy( pointsDisToCircleVector.data(),
              pointsDisToCircleVector.data() + nrOfPoints,
              std::back_inserter( pointsDisToCircle ) );

    tempPipeFace.found = true;

    return tempPipeFace;
}
```

Bibliography

Allseas. (2024). *About allseas*. URL.

Al-sharadqah, A., & Chernov, N. (2009). Error analysis for circle fitting algorithms. *Electronic Journal of Statistics*, *3*. https://doi.org/10.1214/09-EJS419

Chan, T. O., & Lichti, D. D. (2012). Cylinder-based self-calibration of a panoramic terrestrial laser scanner. *The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, *XXXIX-B5*, 169–174. https://doi.org/10.5194/isprsarchives-XXXIX-B5-169-2012

Duda, R. O., & Hart, P. E. (1972). Use of the hough transformation to detect lines and curves in pictures. *Commun. ACM*, *15*(1), 11.15. https://doi.org/10.1145/361237.361242

Lin, D. I. H. (2024, April). Lecture notes in scientific programming (tw4260tu).

microsoft. (2024a). *Profiling cpu usage*. URL.

microsoft. (2024b). *Vs diagnostics*. URL.

Technologies, L. (2024a). *Gocator 2650 smart 3d laser line profile sensor: Datasheet*. URL.

Technologies, L. (2024b). *Gocator 2650 smart 3d laser line profile sensor: Field of view*. URL.