MASARYK UNIVERSITY

FACULTY OF INFORMATICS

Web Application for Parametric Modelling of 2D Shapes Using Geometric Constraints

Master's Thesis

BC. MIROSLAV ŠERÝ

Brno, Fall 2023

MASARYK
UNIVERSITY

FACULTY OF INFORMATICS

Web Application for Parametric Modelling of 2D Shapes Using Geometric Constraints

Master's Thesis

BC. MIROSLAV ŠERÝ

Advisor: RNDr. Katarína Furmanová, Ph.D.
Consultant: Ing. Jiří Hon, Ph.D.

Department of Visual Computing

Brno, Fall 2023

FACULTAS ARTIS INFORMATICÆ • UNIVERSITAS MASARYKIANA •

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Bc. Miroslav Šerý

Advisor: RNDr. Katarína Furmanová, Ph.D.
Consultant: Ing. Jiří Hon, Ph.D.

Acknowledgements

I want to express thanks to my advisor, RNDr. Katarína Furmanová, Ph.D., for her guidance and constructive feedback throughout the writing of this thesis. My sincere appreciation goes to Ing. Jiří Hon, Ph.D., for introducing me to this topic and for his great support and motivation. Moreover, I am thankful to Ing. Jan Hon for his valuable insights and support. Last but not least, my thanks go to my wife for her motivation and rubber duck debugging, as well as to my family, friends, and *Thee, who art in Heaven*.

Abstract

Parametric modeling in Computer-Aided Design (CAD) systems involves creating models through a series of parameter-driven operations. A key part of this process is working with parametric 2D sketches. Sketches consist of geometries and geometric constraints managed by a geometric solver. This thesis discusses the development of Sketcher – a new web application for parametric 2D sketching. Sketcher provides a graphical user interface to work with geometries and constraints. An open-source C++ solver, planegcs from FreeCAD, was adapted for the web platform and published as an open-source wrapper library to solve the geometric constraints in Sketcher. In addition, several automatic constraint optimizations were implemented for better performance and user experience. Sketcher extends Designer, an existing furniture modeling application, for which it provides the necessary functionality to model curved shapes. Sketcher can project and reference spatial geometries from the 3D scene of Designer, greatly simplifying the modeling. Sketcher also increases reusability because multiple variations of a single parametric template sketch can be used in a design. A user study was conducted to test the Sketcher user interface and identify areas for further improvement. The study confirmed the overall good usability of Sketcher, as all users completed the assigned tasks with little or no intervention. On the other hand, the study pointed out several features that could still be implemented to achieve better ergonomics.

Keywords

geometric solving, Computer-aided Design, parametric modeling, parametric sketch, FreeCAD, feature-based modeling, WebAssembly, emscripten, JavaScript, TypeScript, Svelte

Contents

List of Tables

List of Figures

1 Introduction

In the area of Computer-aided Design (CAD), there are two main paradigms for 3D modeling – *direct* and *parametric*. With direct modeling, the user interacts directly with the 3D model in a simple and easily predictable way. For example, pushing or pulling the model surfaces is common in the popular direct modeling tool SketchUp[1]. In contrast, the parametric paradigm is mostly associated with history-based editing. This approach was first used in the 1980s, and since then, it has been deployed in most mainstream CAD software [1, 2]. The principle is that the user builds the 3D model step-by-step through a series of recorded consecutive operations (as shown in Figure 1.1). These operations are called *features* and represent editable model creation history.

These features can be driven by parameters – dimensions, radii, patterns, or others. A feature can be viewed as a function with inputs (parameters) generating outputs (geometry). In this sense, parametric modeling is analogous to writing a computer program. The main advantage of parametric models over non-parametric ones is that the change in a parameter is propagated and influences the resulting model. Therefore, this parametricity is useful for products that vary in dimensions, because all their different configurations can be quickly generated from one parametric template. On the other hand, feature-based modeling can lead to unexpected results [1] and requires some skills related to computer programming (recognizing patterns, debugging, or breaking problems down into smaller steps).

An essential building block in parametric design is a 2D sketch consisting of points, lines, arcs, and other geometries. Combined with other operations such as extrusion, the sketch influences the shape of the resulting 3D model (see Figure 1.1). The parametricity of the sketch is implemented through geometric constraints (e.g., coincidence, tangency, or distance), which maintain the coherence and properties of the shape when the parameters change. The software component that enforces the constraints is called a *geometric solver*. A geometric solver is a key part of a user interface for parametric sketching. When the input parameters or the constraints change, the solver outputs a new

1. `https://www.sketchup.com`

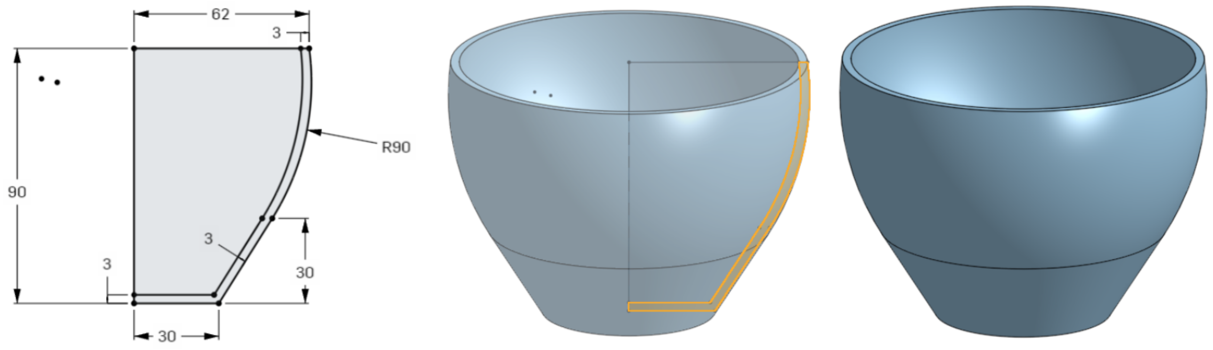


Figure 1.1: Parametric modeling of a cup using an initial sketch. First, the sketch was created using line segments, arcs, and geometric constraints (left picture). Second, a sketch area (highlighted) is used as an input for revolving along the vertical axis (middle picture). Third, the final 3D model was generated (right picture). The model was prepared in OnShape – web modeling software.

sketch state, which satisfies the constraints. The solver may fail when the constraints conflict or for various other reasons.

The main goal of this thesis is to develop a graphical web interface for creating and editing parametric 2D sketches – Sketcher. The work is motivated by its broad applicability in Designer, an existing web-based parametric furniture-modeling software. Sketcher should extend Designer's functionality with geometric solving capabilities and tools for sketching. The resulting sketches can be used for shaping/machining the 3D parts or defining trajectories for placing objects.

Designer aims to be a unified platform capable of computing all the geometries and realistic visualizations in real-time in a web browser. Furniture design has great potential for parametric modeling. One of the future goals is to develop an e-shop module for live visualizations of configurable desks, shelves, or beds that can be immediately transferred to automated production.

1.1 Requirements for Sketcher

The requirements for the Sketcher module were discussed with the client[2] that develops Designer. The requirements are based on the client's ideas and experience with existing (insufficient) software solutions for furniture design and production. This specific industry also limits the required functionality of Sketcher, as implementing all the features common in commercial CAD applications would far exceed the scope of a thesis.

R1 *Necessary 2D parametric modeling functionality.*

- Drawing tools – point, line, circle, arc, ellipse, and elliptical arc.
- Constraint tools – dimensional (distance, angle, ...) and non-dimensional (horizontal, perpendicular, ...).
- Modification tools – selection, dragging, and deleting.

R2 *Referencing variables and expressions.* In Designer, the user can define variables and use expressions. It is required that the expression can be used in parametric constraints in Sketcher.

R3 *Projection of spatial geometries.* Sketcher is required to support the projection of geometries from the 3D scene generated by other features to the currently active sketch plane.

R4 *Error handling, notification, and hints.* If an invalid state occurs, the user must be informed of the problem. Such problems can arise from the geometric solver or from the user interaction. The messages should explain the problem and be helpful for inexperienced users.

It is expected that Designer technology and its APIs will be used in Sketcher. The following requirements are based on the technology decisions made for the Designer application:

2. `http://www.salusoft89.cz`

R5 *No server-side computation for sketch evaluation.* The parametric sketch solution must be computed directly in the browser to ensure high interactivity and offload the computation from server to client. This supposes using an open-source solver with a license that allows free use in commercial software.

R6 *Render sketches to the 3D scene of Designer.* The module needs to use the interface provided by the rendering engine to correctly display the geometries in the 3D scene and properly handle user interaction.

R7 *JSON data structure for sketches.* All sketch data should be represented in JSON structure that captures the geometric relationships and can be stored in the Designer's storage.

1.2 Outline of the thesis

An overview of selected methods for solving parametric sketches is discussed in Chapter 2. Chapter 3 describes the *planegcs* solver from the open-source desktop CAD application FreeCAD and adapting it to the web by creating a wrapper library. This provides the necessary functionality for the Sketcher module that is described in the follow-up Chapter 4. Finally, Chapter 5 reports on user testing of Sketcher and discusses the observations. The project is summarized in Chapter 6, including proposals for future improvements.

2 Background and related work

A key component that enables working with parametric sketches is a geometric solver. Generally, the solver searches for a configuration of the geometries that satisfies the required constraints. Moreover, the input from the user is not always well constructed and may contain redundant constraints or conflicts that can cause issues when solving directly. The solver should also be able to detect and report these inconsistencies. To achieve its functionality, it usually performs the following steps: (i) translation of the sketch to an internal data structure, such as a graph or a system of algebraic equations; (ii) analysis and decomposition of the system; (iii) finding solutions of the decomposed subsystems; and (iv) assembling the solution [2].

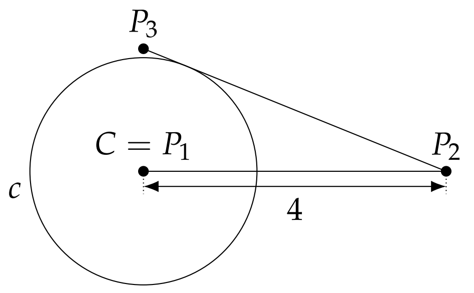
The sketch can be regarded as a *Geometric Constraint System* (GCS), which is characterized in Section 2.1 together with an overview of common approaches for solving GCS – most notably the numerical and graph-based methods. The topic is too broad to provide a comprehensive review within the scope of this thesis. The following review is limited to the scope of this thesis – the 2D Euclidian space and constraints that use *equality* in their definition, focusing on numerical solving methods.

Section 2.2 presents a survey of open-source libraries that implement constraint-solving techniques. Their suitability for a web-based Sketcher is discussed.

2.1 2D Geometric constraint system

A geometric constraint system (GCS) in 2D consists of elementary geometric primitives in the Euclidean space, such as points, lines, circles, or arcs, and constraints that encode geometric relations between them or relations to values of numeric parameters. The constraints can be defined as *non-dimensional* – e.g., perpendicularity, coincidence, or concentricity, or *dimensional* – represented by distance, radius, or angle. Constraints that describe relations between the parameters are called *algebraic*.

A solution of GCS corresponds to a valid evaluation of input parameters, such that all the constraints are satisfied. One way of translating



C_1: Fix$(P_1, (0,0))$
C_2: Coincident(P_1, C)
C_3: Horizontal(P_1, P_2)
C_4: HDistance$(P_1, P_2) = 4$
C_5: Vertical(P_1, P_3)
C_6: Tangent(c, P_2P_3)

$$\begin{aligned}
C_1 &\to & x_1 = y_1 &= 0 \\
C_2 &\to & x_c - x_1 = y_c - y_1 &= 0 \\
C_3 &\to & y_2 &= y_1 \\
C_4 &\to & x_2 - x_1 &= 4 \\
C_5 &\to & y_3 &= y_1 \\
C_6 &\to & \frac{|(x_3 - x_2)(y_2 - y_c) - (x_2 - x_c)(y_3 - y_2)|}{\sqrt{(x_3 - x_2)^2 + (y_3 - y_2)^2}} &= r_c
\end{aligned}$$

Figure 2.1: Example of a geometric constraint system. It is defined by the geometries visualized here (two line segments and a circle c with radius r_c) and a list of constraints on the right. The constraints are written in their analytical form below. The coordinates (x_i, y_i) are for points P_i and (x_c, y_c) for the point C. Note that the tangency constraint C_6 is an equation for the distance between point C and line P_2P_3.

GCS into a solvable system is the analytical approach, where the system of equations represents both the geometries and constraints [3]. Figure 2.1 illustrates a particular instance of a geometric system and its corresponding constraint equations.

2.1.1 Characterization of a constraint system

The equation system can be in four possible states [4] concerning its set of solutions S (see Figure 2.2):

1. *under-constrained* when S is infinite,
2. *inconsistently over-constrained* when S is empty (the system has no solutions),

3. *consistently over-constrained* when S is finite, and there is a subset of constraint equations that can be removed without changing S (i.e., increasing $|S|$),

4. *well-constrained* when S is finite and there is no such subset.

A significant value related to GCS is *Degree of Freedom* (DOF). In terms of linear algebra, DOF is complementary to the *rank* (i.e., the number of linearly independent equations) of the matrix defining the GCS with n parameters:

$$n = \text{DOF} + \text{rank}. \tag{2.1}$$

Adding an unconstrained geometric object to the sketch increases the total DOF by the number of independent variables (coordinates) used to represent that object. For example, a sketch with two points has DOF $= 4$, corresponding to 4 variables x_1, y_1, x_2, and y_2 that can be set arbitrarily. When a constraint is added, such as $x_1 = x_2$, the DOF of the system is either decreased (in this case to DOF $= 3$) or remains constant, which means the new constraint is redundant (linearly dependent). If the system has DOF $= 0$ and does not contain any redundant constraints, then it is well-constrained. Each system with DOF > 0 is either under-constrained or inconsistent.

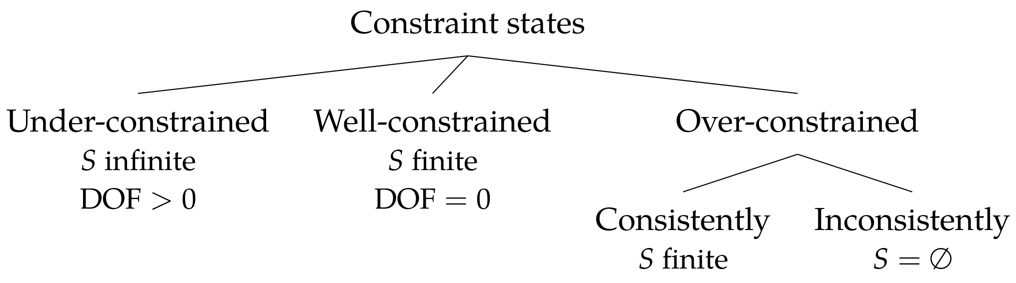


Figure 2.2: Illustration of constraint system states.

2.1.2 Numerical solving

Numerical methods are one of the first approaches to solving GCS. They are iterative and mostly based on Newton's method (also known as *Newton-Raphson method*) [5].

For the introduction of Newton's method, let's consider its univariate version for simplicity. Given a twice differentiable function $f(x)$ and an initial guess x_0, Newton's method iteratively finds a sequence $\{x_n\}$ that converges to some x^* for which $f'(x^*) = 0$ (i.e., x^* is a stationary point). In each iteration step, a quadratic model $L(\Delta x)$ of the surroundings is obtained by the Taylor expansion. For the first iteration, the model is following:

$$L(\Delta x) = f(x_0 + \Delta x) = f(x_0) + f'(x_0)\Delta x + \frac{1}{2}f''(x_0)\Delta x^2 \tag{2.2}$$

The desired step is obtained by searching for the model's stationary point:

$$\frac{d}{d\Delta x}L(\Delta x) = 0 \tag{2.3}$$

$$\frac{d}{d\Delta x}\left(f(x_0) + f'(x_0)\Delta x + \frac{1}{2}f''(x_0)\Delta x^2\right) = 0 \tag{2.4}$$

$$f'(x_0) + f''(x_0)\Delta x = 0 \tag{2.5}$$

$$\Delta x = -\frac{f'(x_0)}{f''(x_0)} \tag{2.6}$$

$$x1 = x0 - \frac{f'(x_0)}{f''(x_0)} \tag{2.7}$$

In the generalized multivariate case, the first derivative corresponds to the *gradient vector* $\nabla f(\boldsymbol{x_0})$ and the second derivative is represented by the *Hessian matrix* $H_f(\boldsymbol{x_0})$. The equivalent of Equation 2.7 for n variables is:

$$\boldsymbol{x_1} = \boldsymbol{x_0} - H_f(\boldsymbol{x_0})^{-1}\nabla f(\boldsymbol{x_0}), \tag{2.8}$$

where $\nabla f = \left[\frac{\delta f}{\delta x_1}, \ldots, \frac{\delta f}{\delta x_n}\right]$ and $(H_f)_{i,j} = \frac{\delta^2 f}{\delta x_i \delta x_j}$ for $\forall\, i, j \in 1..n$.

The step in Newton's method is not guaranteed to be a *descent* towards a minimum because it is generally taken towards the stationary point in the model L, which can be a minimum, maximum, or an inflection (saddle) point. The sufficient condition for the step to be a descent in the multivariate case is when the Hessian matrix is *positive definite*.

To apply the multivariate Newton's method to solve the GCS, its definition (e.g., the equations in Figure 2.1) is turned into the *least squares optimization problem*:

$$\min_x F(\boldsymbol{x}) = \frac{1}{2}\sum_{i=1}^{m} f_i(\boldsymbol{x})^2 = \frac{1}{2}||\boldsymbol{f}(\boldsymbol{x})||^2, \tag{2.9}$$

where $\boldsymbol{x}$ is a vector of all parameters and $\boldsymbol{f}$ is a vector of GCS equations formulated as $f_i(\mathrm{x}) = \mathbf{0}$ (when the geometric constraint is satisfied). $F(\boldsymbol{x})$ is also called the *error* or cost function. Intuitively, given a guess $\boldsymbol{x_k}$ in the k-th iteration step, the error tells us how acceptable the guess is. Consequently, if an algorithm finds such $\boldsymbol{x_k}$ that the error is zero, it finds a solution to the GCS equations f_i.

The update step from Newton's method is not directly used here, because computing the Hessian can be expensive and sometimes inappropriate. Instead, the *Gauss-Newton* method [6] applies the chain rule of differentiation and simplifies several terms. As a result, the gradient for the least squares problem can be expressed as:

$$\nabla F = J_f^T \boldsymbol{f}(\boldsymbol{x_0}), \tag{2.10}$$

where J_f is the *Jacobian matrix* of the function vector $\boldsymbol{f}$:

$$(J_f)_{i,j} = \frac{\delta f_i}{\delta x_j}, \tag{2.11}$$

and the Hessian is approximated by the following term:

$$H_f \approx J_f^T J_f. \tag{2.12}$$

If the Jacobian matrix J_f is linearly independent, this approximation is guaranteed to be positive definite [6]. This results in taking descent steps, which in effect increases the convergence success of Gauss-Newton.

Using these terms to replace those in the original Newton's update step (eq. 2.8) leads to the following equation:

$$\boldsymbol{x_1} = \boldsymbol{x_0} - \left[J_f(\boldsymbol{x_0})^T J_f(\boldsymbol{x_0})\right]^{-1} J_f(\boldsymbol{x_0})^T \boldsymbol{f}(\boldsymbol{x_0}) \tag{2.13}$$

To recapitulate, every iterative approximation algorithm requires two values: the *current state* of the iteration and the *next step*. For the least squares problem, into which the GCS can be formulated, the state is given by the error function – a sum of individual errors of f_i (also called *residuals*). One of the approaches to obtain the next step is the Gauss-Newton method. Gauss-Newton cannot be applied to general functions, but only to the least squares problem.

There are two main strategies to improve the convergence of iterative optimization algorithms – *line search* and *trust region*. In the line search, after the optimal direction of improvement is computed using the model L within the iteration step, a coefficient $\alpha \leq 1$ is determined to limit the step size. In the trust region, the model L is trusted only within a certain given region (corresponding to maximal step size). Within this limited region, the optimal step is computed [7].

The extensions of Gauss-Newton generally increase its robustness, meaning they find a solution even if the initial guess is far off the final solution. Two examples are the *Levenberg-Marquardt* or the Powell's *"dog leg"* algorithms. They use the trust region approach and interpolate between Gauss-Newton and gradient descent steps, each in a different manner [6, 7]. The gradient descent step is a standard technique, which uses the negative of the gradient vector as the step direction. The idea behind these hybrid methods is that near minimum, the Gauss-Newton converges faster, while further away it is better to use the gradient descent steps.

Broyden–Fletcher–Goldfarb–Shanno algorithm (BFGS) is based on Newton's method and therefore usable for general functions. It uses a different approximation of Hessian, continuously updated in each iteration [7].

The abovementioned algorithms are designed for optimization problems formulated as *unconstrained*. Although this term might seem confusing in this context, it points to the problem *definition*, where a single function $f(\boldsymbol{x})$ is minimized without further restrictions. Complementary to that, there are methods for the *constrained* problems, which are formulated as:

$$\begin{aligned} \min_{x} \quad & f(\boldsymbol{x}) \\ \text{subject to} \quad & g_i(\boldsymbol{x}) = c_i \quad \text{for } \ i = 1..m \end{aligned}$$

An example of a method for solving the constrained problems is *Sequential quadratic programming* (SQP) [7]. The GCS least squares formulation in Equation 2.9 can be turned into the constrained definition by separating m constraints from f_i into g_i. This can be used as a technique to prioritize the g_i constraints.

A distinctive family of methods is called *homotopy continuation* [8]. In contrast to methods based on Newton's method, they theoretically guarantee a globally optimum solution and behave better in finding the closest one. In practice, however, they are slower than the local methods [2].

2.1.3 Graph-based methods

Apart from numerical approximation, another mainstream approach in GCS solving is based on the graph methods. They play an essential role in solving sketches that involve hundreds or thousands of constraints by performing a qualitative analysis of the system in polynomial time. The analysis also provides useful explanations for the users about the treatment of problems in the sketch, which is not provided by the numeric solvers alone [2].

Graph-based methods use various types of graphs to represent the constrained system. For example, an *equation graph* can be created from the algebraic equations, or the direct representation of geometric objects and constraints can be put into the *constraint graph*. A common procedure performed in both cases is decomposing into smaller subsystems that are easier to solve, or separating well-, under-, and over-constrained parts.

The constraint graph regards the geometric objects as vertices and the constraints between them as edges. The strategy is to analyze this graph to detect subproblems that can be solved separately by predefined construction rules or numerical methods [4].

The decomposition of the constraint graph can be carried out in various ways – *top-down* [9] or *bottom-up* [10], where the former works by iteratively splitting the GCS into components and is more suitable for under-constrained sketches while the bottom-up approach is more suitable for over-constrained systems and operates via component merging. Some solutions combine both styles [5]. Technically, these algorithms compute *maximum flows* [11], *maximum matching* [12], or *k-connected components* [13].

2.1.4 Other solving approaches

Along with the techniques from the numerical optimization and graph theory, there are various other approaches for solving GCS [14, 15]:

- In the *logic-based* approach [16], a set of axioms and assertions is used to describe the system. This is then transformed by logical reasoning to obtain the solution steps.
- The *symbolic approach* [17] processes the system of (nonlinear) equations and rewrites it using specific algorithms with the use of so-called *Gröbner bases* or *Wu-Ritt decomposition*.

These approaches have an exponential (or double exponential) time complexity, so they are not widely used in the community. Still, they can be useful in various sub-tasks of solving GCS – e.g., symbolic solving of predetermined patterns in the subsystems can be combined with numerical optimization methods [15].

2.2 Existing software solutions of geometric solvers

Geometric solvers have been part of commercial CAD systems since the early 1980s [14]. Nowadays, it is common for software vendors not to develop a solution on their own. Instead, it is preferable to license an existing library. One of the most prominent examples of a commercial geometric solver is *D-Cubed DCM*[1]. D-Cubed was formerly a company founded by John Owen, who pioneered the top-down approach for GCS graph decomposition [18]. D-Cubed was later acquired by SIEMENS and became a mark under which solvers for many popular applications are provided. The list includes Autodesk Inventor, IronCAD, OnShape, SolidWorks, and others [19]. As an alternative to SIEMENS, there are solvers developed by *LEDAS*[2]. These mentioned

1. `https://www.plm.automation.siemens.com/global/en/products/plm-components/d-cubed.html`
2. `https://ledas.com/en/expertise/3d-modeling`

solvers are closed-source and offer a wide range of capabilities for reasoning about (ill-constrained) sketches and other automatic tooling via their API. However, their support for usage in web browsers is limited.

Implementing a fully competent geometric solver from scratch is a challenging task. Even with the help of libraries implementing general techniques from numerical optimization, it is still necessary to handle the over- and under-constrained cases, detection of redundancies, and other parts of the whole solving process.

Instead of licensing a commercial solver or developing one from scratch, the open-source contributions in this field were explored, and a suitable implementation was integrated, based on the client's requirements presented in Section 1.1. The conditions for an ideal candidate were three-fold: (i) A license that permits free use in closed-source commercial projects; (ii) the ability to use the library in a TypeScript project (to be executed in the web browser); and (iii) good maintenance status.

Table 2.1 presents the result of a survey of open-source libraries, including their license, programming language, and maintenance status. Of nine surveyed solvers, only three were maintained – SolveSpace, planegcs, and jsketcher. Though jsketcher is programmed in JavaScript and thus could be easily integrated into a web project, the license fee is too high [20]. Similarly, SolveSpace is licensed under the GPL license, which is unsuitable for a commercial project since it requires the whole product to be open-sourced under the GPL license. Therefore, I chose planegcs for its LGPL license, which permits free use in a closed-source commercial project. Moreover, a community has actively developed it since 2002 and still has good maintenance status [21]. Despite being part of a larger software, it is well separated from the rest of the source code, which makes it usable as a standalone library. Nevertheless, modifications need to be made to make it work on the web platform.

Table 2.1: Overview of open-source GCS solver implementations.

Name	Licence	Used methods	Language	Status
SolveSpace	GPL	numerical	C++[a]	maintained
planegcs	LGPL	numerical	C++	maintained
jsketcher	custom[b]	numerical	JavaScript	maintained
pygeosolve	GPL	numerical	Python	unmaintained
stutterCAD	–	numerical	JavaScript	unmaintained
NoteCAD	GPL	numerical	C$\sharp$	unmaintained
cluster	GPL	graph	Python	unmaintained
constraint-solver	–	graph	Python	experimental, unmaintained
Protractr	–	numerical	TypeScript	experimental, unmaintained

[a] with a binding to JavaScript
[b] paid commercial use

3 Planegcs wrapper library

The survey of the open-source libraries for solving geometric constraint systems shows that there is a part of FreeCAD's source code – *planegcs* – that best meets the project's requirements. To use it on the web platform, a JavaScript binding is necessary. Since none was found during the research, this part of the thesis focuses on its implementation via the available tooling.

This chapter starts with the introduction of sketching in FreeCAD in Section 3.1 followed by an overview of the solver implementation in Section 3.2. Then, the process of packaging the C++ solver code into a library that could be used from JavaScript environments is described in Section 3.3. As a result of this work, an open-source wrapper was created and published on GitHub[1] and NPM[2].

3.1 2D sketches in FreeCAD

FreeCAD is a multi-platform, open-source parametric 3D CAD modeler focused on product design and mechanical engineering. It is based on scientific open-source libraries, such as *Open Cascade Technology*[3]. FreeCAD is developed mainly in C++ with Python bindings, providing scripting functionality to experienced users. A community of developers has maintained it since the early 2000s, most of them being volunteers. The discussions about the project are held at a forum[4] or within its GitHub repository[5] [22]. For specific internals of FreeCAD, the forum is usually the only source of relevant information.

FreeCAD is designed to be an extensible software with a core system and additional modules. The module that facilitates parametric 2D sketching is called *Sketcher*[6]. The user interface of Sketcher is shown in Figure 3.1. The source code of this module contains a purposely

1. https://github.com/Salusoft89/planegcs
2. https://www.npmjs.com/package/@salusoft89/planegcs
3. https://dev.opencascade.org/
4. https://forum.freecad.org/
5. https://github.com/FreeCAD/FreeCAD
6. https://github.com/FreeCAD/FreeCAD/tree/main/src/Mod/Sketcher

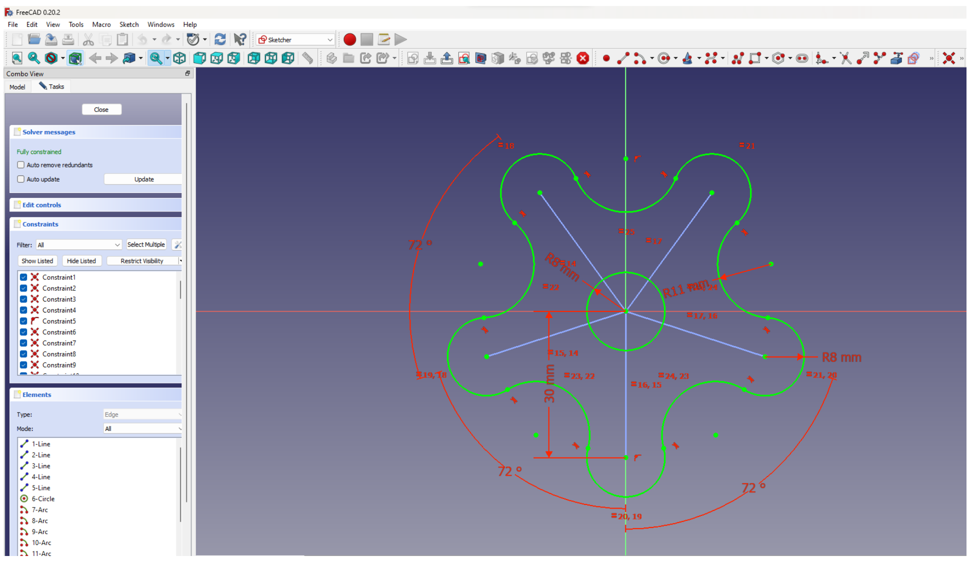


Figure 3.1: User interface of FreeCAD's Sketcher.

isolated part – the *solver*.[7] Each of these two abstraction levels uses its own representations of the constraints and geometries. While the Sketcher uses high-level classes and is interconnected with the rest of the application and GUI, the classes in the solver correspond to the algebraic representation used for numeric solving. Furthermore, one (complex) sketcher constraint might be mapped onto one or more solver constraints. The downside of this separation is that these two parts must be interconnected by an interface code [23]. On the other hand, it makes the independent solver part usable in other projects, such as within this thesis, or in the *SALOME*[8] project (*Shaper* module).

Planegcs was not present in FreeCAD from the beginning. In the first stage, third-party solvers were included, later replaced by *FreeCGS* – a custom implementation by the community members in 2012. When some active members stopped contributing, it was rewritten and renamed to *planegcs* in 2019 and keeps being developed further [21].

7. The naming and distinction between a *sketcher* and *solver* that is used in this thesis is directly inspired by this separation in FreeCAD.
8. `https://www.salome-platform.org/`

3.2 Overview of *planegcs* solver

To introduce the usage of the planegcs solver and its data representations, let's see an example with a simple geometric system (two points and one constraint). The code in Listing 3.1 instantiates two classes from planegcs – `Point`, effectively being a container for the point's coordinates, and `System`, which serves as the main entry point for interacting with the solver:

```
Point p1;
p1.x = new double(0);
p1.y = new double(0);

Point p2;
p2.x = new double(1);
p2.y = new double(1);

double *distance = new double(10);

auto sys = new System();
sys->addConstraintDifference(p1.x, p2.x, distance);

std::vector<double *> p_parameters;
p_parameters.push_back(p1.x);
p_parameters.push_back(p1.y);
p_parameters.push_back(p2.x);
p_parameters.push_back(p2.y);
// distance is not pushed to p_parameters => remains
    fixed

sys->solve(parameters);
```

Listing 3.1: Basic usage of planegcs.

An important concept present in the example is the *p-parameters* vector of pointers to double (`double *`). The term p-parameters is used to denote this particular vector to avoid ambiguity of the word parameter. In this example, p-parameters are passed to the solve function, which enables planegcs to mutate their values during the solving process. Conversely, if a pointer is not present in `p_parameters` (such as `distance`), its value remains fixed.

The `addConstraint(...)` methods are convenience functions for instantiating constraint classes and adding them within the system.

Importantly, each constraint class contains an implementation of a common interface:

```
virtual double error();
virtual double grad(double *);
```

These methods are later utilized for building the system of equations and subsequently by the solver's algorithms for numerical optimization. Namely, the errors of all constraints are summed up to determine, whether the current approximation has a sufficiently low error and can be accepted as a valid solution.

When adding a constraint, an optional extra argument can be passed – the so-called *tag*. Primarily, it serves as an id of the constraint. However, when `-1` is passed as a tag, the constraint is referred to as a *temporary constraint* and is treated differently by the solver – among other things, its error does not contribute to the total error. Therefore, the solution may be returned as valid, even though the temporary constraints are not fully satisfied (i.e. have a non-trivial error). A typical case for using such constraints is when the user is dragging an already constrained sketch geometry that can't always exactly follow the mouse position.

Another parameter, which can be passed to change the solver's behavior is *scale*. Scale is simply a multiplication factor for the values of `error` and `grad` functions. The effect of scaling is that constraints with bigger values are more prioritized in solving.

Each constraint can be either *driving* (which is the default) or *reference*. The difference is that reference constraints never influence the resulting sketch geometry – reference constraints are used solely for measurements. Their behavior differs from the temporary constraints, which do influence the sketch geometry but only to an extent that does not cause conflicts.

Solving process

The complexity of the solving process is hidden behind the `solve` function. Here is a short description of the steps carried out by the planegcs and algorithms involved:

1. *Diagnosis* – First, the Jacobian matrix of the equation system is constructed from the constraints' gradients. A matrix decompo-

sition is performed to obtain its rank (and thus the degrees of freedom) and to detect redundant or conflicting constraints. For the decomposition, one of the two algorithms from *Eigen3*[9] linear algebra library is invoked – SparseQR or DenseQR. Though DenseQR is slower, it can yield more information about the system.

2. *Optimizations* – The equation system is partitioned into decoupled subsystems using the *Boost Graph Library* (BGL)[10]. Also, the p-parameters vector is reduced using the equality constraints, and temporary constraints are separated into their own subsystem.

3. *Solving* – Planegcs mainly uses numerical optimization – namely DogLeg, Levenberg-Marquardt, and BFGS (Broyden-Fletcher-Goldfarb-Shanno), which are implemented in the corresponsding functions `solve_DL`, `solve_LM`, and `solve_BFGS`. Alternatively, a different algorithm (SQP) for solving two sub-systems is used when the system has temporary constraints. The solving process might be altered by setting various parameters, such as the maximum number of iterations, convergence threshold, etc.

Planegcs is structured into multiple C++ files, which correspond to different components of the code. These are the key parts:

- `GCS.cpp` – Here is the main class (entry point) for working with planegcs. The solving process described above is implemented here.

- `Constraints.cpp` – This file contains all the constraints available. Each constraint implements methods for computing its gradient and error.

- `Geo.cpp` – Here, classes for the inner representation of geometries are defined, together with some utility classes for working with vectors and their derivatives. The following geometries are currently present in planegcs: Point, Line, Circle, Arc, Ellipse,

9. `https://eigen.tuxfamily.org`
10. `https://www.boost.org/doc/libs/1_83_0/libs/graph/doc/index.html`

ArcOfEllipse, Hyperbola, ArcOfHyperbola, Parabola, ArcOfParabola, and BSpline.

- `SubSystem.cpp` – SubSystem is a class for representing a system of equations and calculating its Jacobian, gradients and other values.

3.3 Implementation of wrapper library

Since the planegcs solver is part of FreeCAD – a desktop application written in C++, its code needs to be adapted for web browsers. The description of this process starts with an overview of the technologies used for this transition.

Technology stack

WebAssembly (wasm)[11] is a low-level assembly-like language executable in the web browser. It enables porting code from various languages to the web platform. The standard toolkit for compiling an existing C/C++ codebase into wasm is *Emscripten*[12]. Wasm modules can be called from JavaScript via an API.

TypeScript (TS)[13] is a superset of JavaScript that enhances its syntax with a flexible type system. TS also consists of a static analyzer and supports rich code completion. Code written in TS must be *transpiled* to JavaScript to run in browsers.

Docker[14] is a container-based virtualization platform that is used for running software in an isolated context. Docker containers can ease development processes and the deployment of the application in the cloud by providing a consistent environment with the necessary dependencies.

11. `https://webassembly.org/`
12. `https://emscripten.org/`
13. `https://www.typescriptlang.org/`
14. `https://www.docker.com/`

Architecture considerations

Several requirements were considered for the implementation of the wrapper library for planegcs. The library should: (i) cover all geometries and constraints available in planegcs, (ii) contain type annotations for convenience and type-level safety, (iii) work with JSON data (based on Requirement R7), and finally, (iv) it should stay synchronized with the up-to-date implementation from the FreeCAD repository. All of these requirements were taken into account while creating the library.

Generally, compiling code into a wasm module is not a trivial process. Usually, workarounds are necessary – either changes in the source code itself or writing some extra code (so-called *binding* code). A further limitation is that the current state-of-the-art toolkit for the task – Emscripten – generates JavaScript only, without any type annotations. Therefore, to take advantage of TypeScript's static analysis, the type annotations have to be created separately.

Moreover, there are many functions for adding various constraints in planegcs, and it would be cumbersome and error-prone to manually write the binding code and types for all of these different methods. Instead, it is preferable to have scripts that generate the desired outputs by parsing the C++ code. In the wrapper library, multiple scripts constitute the build process. Its output can be regarded as a lower-level module, which is then invoked by the main entry point of the library – `GcsWrapper`. This class provides useful abstractions and type safety for creating and solving sketches.

3.3.1 Build process

The build process of the wrapper library is depicted in Figure 3.2. It consists of multiple steps – In Step 1, the current version of the source code is downloaded from the git repository, and several patches are applied. This step is optional. Then, the C++ code is parsed, and type annotations and the binding code are generated via a templating engine (Step 2). The bindings and the patched planegcs code are then used in Step 3 as the input for emscripten. Each step is carried by a script invokable by `npm` – a CLI tool and a dependency manager for JavaScript.

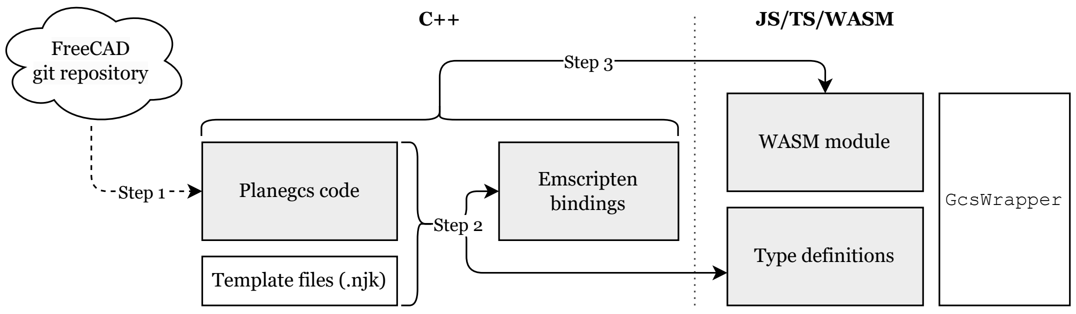


Figure 3.2: The build process of the planegcs wrapper library. The process starts with Step 1 by pulling the source code from FreeCAD's repository. Then (Step 2), bindings and type definitions are generated by parsing the source code. Finally (Step 3), the C++ code is compiled into WebAssembly. The white boxes represent manually written files, whereas the gray ones are those generated by a script. The diagram is divided into a C++ part and a JavaScript-related part. Only the latter is used at runtime. The main entry point, `GcsWrapper` is located on the right.

Step 1: Updating and patching planegcs code

The FreeCAD's codebase (including the solver implementation) keeps evolving as a group of volunteers actively maintains it. To keep the planegcs code up-to-date, the library includes a script that automatically downloads the newest files from the main branch of the git repository. Because planegcs does not have a dedicated repository, the update script uses a sparse[15] version of git checkout/clone commands that downloads only the relevant files and folders.

After downloading the code, `planegcs/commit.txt` is updated with the current commit hash, and patches necessary for a successful emscripten build are applied. These patches fix several `#include` statements for C++ header files and some other minor issues that mostly don't change the code functionality.

The update script is located in `planegcs/update_freecad.sh` and can be invoked via `npm run update-freecad`.

15. `https://git-scm.com/docs/git-sparse-checkout`

Step 2: C++ code parsing

The goal of C++ code parsing is to obtain the data necessary for generating the type annotations and the binding code. The *treesitter*[16] library was chosen as the parsing backend. Treesitter is a mature project with support for many programming languages, and it supports querying the abstract syntax tree (AST) for the target data. Using a full parser instead of simpler methods (such as regular expressions) requires more domain-specific code, but it is more robust for future code changes in FreeCAD.

During this step, the planegcs source code is queried for (i) specific enums and their values, (ii) geometry classes, and (iii) functions for adding constraints. For generating output files, the *nunjucks*[17] template library is utilized.

Multiple file types are generated in this step: (i) `bindings.cpp`, which includes the binding code required for compilation to WebAssembly via emscripten, and (ii) TypeScript files, which contain type annotations for code generated in the next step. All of the parsing-related code is located in the `build-bindings` folder of the wrapper and can be executed by the command `npm run build:bindings`.

Step 3: Compilation to WebAssembly

As mentioned, planegcs depends on two external libraries, *Eigen3* and *Boost*. These external libraries, including the emscripten compiler, must be available during the build. The process of installing and setting up these dependencies is captured in a `Dockerfile` used by the Docker tool to prepare the build environment.

In addition to files generated in the previous steps, the build folder contains a manually written `CMakeLists.txt`. This file instructs the `emcmake` compiler to link the external dependencies, and it alters the build output by setting various flags. Some of these flags are identical to those for compiling C++ applications (such as the level of optimizations), and others are specific for emscripten.

In JavaScript, there are historically multiple ways of importing dependencies. The modern way is to use ES6 module syntax with

16. `https://tree-sitter.github.io/tree-sitter/`
17. `https://mozilla.github.io/nunjucks/`

`import` and `export` keywords [24]. To support this style, the build must be explicitly configured:

```
    --closure 1 -s EXPORT_ES6=1 -s MODULARIZE=1
```

Further flags are used to instruct the compiler that some features are unnecessary, which can positively influence the size of the output files. The following optimization flags are used for the *release* version:

```
    -O3 -s WASM_BIGINT=0 -s ASSERTIONS=0
    -s ERROR_ON_UNDEFINED_SYMBOLS=0
    -s ELIMINATE_DUPLICATE_FUNCTIONS=1
    -s MALLOC=emmalloc -s NO_FILESYSTEM=1
```

This build step can be executed by the command `npm run build:wasm`, which executes the release build or `npm run build:wasm:debug` for debugging. These commands require the presence of the building Docker image in the host system, which can be created by running `npm run build:docker`.

3.3.2 Emscripten limitations

Generally, C++ codebase adaptation for usage from JavaScript requires workarounds for certain issues. These are mainly caused by the different nature of the C++ language versus JavaScript or the limitation of the tooling. The most prominent ones are presented in the following text.

Pointers and references

In their arguments, the `addConstraint` functions from planegcs contain references to C++ geometry classes and pointers to double values (i.e. p-parameters). While emscripten supports interchanging instances of classes by reference, the situation gets more complicated with the pointers to primitive data types, such as `double *`.

The binding file must explicitly state classes that should be passed to and from JavaScript. Exporting enums and specific data types to JavaScript can be configured similarly. However, JavaScript does not have a concept of pointers naturally occurring in the C++ code. Consequently, neither emscripten provides any workarounds for this incompatibility to our knowledge.

To overcome this limitation, a `std::vector` of p-parameters is created and used internally within the binding code, whereas only the numeric indices are used by the external API. Therefore, all planegcs methods are adapted to using these indices instead of pointers during Step 2. This solution requires adding a few helper methods to manage the p-parameters and instantiate the geometry classes.

The above solution is currently implemented in the wrapper library. Alternatively, a `MemoryBuffer` object could be used to manage p-parameters. MemoryBuffer is a native object, that allows sharing the same piece of memory between JavaScript and WebAssembly. The advantage is faster data exchange since it does not require data copying. Nonetheless, it requires a more low-level approach, which increases the complexity of the implementation.

Overloading functions

Another concept missing in JavaScript is *function overloading*. Overloading is used within planegcs because there are multiple possibilities for defining constraints. For example, the method for adding a point-on-line constraint has two overloads: one that accepts `Point` and `Line` and the second one that accepts three `Point` objects.

The solution that was implemented in the parsing script (Step 2) is to distinguish these overloads in the bindings code by a function name suffix, which corresponds to the arguments (such as `_pl` or `_ppp`, respectively).

Multithreading

The original planegcs implementation uses multithreading only for parallel execution of two different QR decompositions – one for detecting redundant/conflicting constraints and the second for identification of dependent p-parameters.

Parallelism support is disabled by default in emscripten and must be explicitly configured by the `-pthread` flag. However, as of the current state of the wrapper library, the multithreading is patched into single-thread execution. Although it creates a performance penalty for complex sketches, it simplifies using the library code in web projects.

Multithreading is a complication because the JavaScript code is executed in a single thread by default. In JavaScript environments, the support for executing code in other thread(s) is provided by Web Workers API [25]. Emscripten uses web workers and an implementation of shared memory (`SharedArrayBuffer`) to emulate the POSIX-standard threads. The shared memory buffer is supported in current browsers, but for security reasons, it is required to set special browser headers [26, 27].

In future releases of the wrapper library, there could be a configurable option for using the parallel execution of the QR algorithms to increase the solving speed.

3.3.3 Main API of the library

The main entry point of the wrapper library is the `GcsWrapper` class. This class offers some abstraction over the lower-level calls of planegcs methods and the management of the p-parameters vector present in the planegcs API.

Three different types of objects can be contained in a sketch: geometries, constraints (these two are together referred to as *primitives*), and sketch parameters. While the primitives have numeric IDs, the sketch parameters are identified by names.

GcsWrapper is a container for a *single* sketch. It provides a higher-level API for the definition of the sketch by JSON objects representing the sketch primitives:

```
gcs_wrapper.push_primitive({
    type: 'point',
    id: 1,
    x: 0,
    y: 0
});
// ...
gcs_wrapper.push_primitive({
    type: 'p2p_distance',
    id: 3,
    p1_id: 1,
    p2_id: 2,
    distance: 100,
});
```

Here, the advantage of TypeScript becomes apparent, because these JSON objects can be type-checked using the type annotations exported in Step 2 of the build process. Nevertheless, two important things happen upon calling the `push_primitive` function for adding a *geometry* primitive:

- the method `push_p_param` from the planegcs binding code is called (both for x and y in case of Point), which alters the C++ vector of p-parameters, and
- an internal JavaScript Map `p_param_index` in the GcsWrapper is updated to keep the mapping of object IDs to indices of p-parameters.

Pushing a constraint is a slightly more complex process as there are different types of constraints. For the sake of simplicity, the addition of the `p2p_distance` constraint is explained:

- the helper method `make_point` from the binding code is invoked (for both p1 and p2 in the example), which brings to JavaScript a reference to the `Point` C++ class instance,
- the method `push_p_param` is called with the value of `distance`,
- the planegcs method `addConstraintP2PDistance` is called with the correct arguments, and
- the geometry classes are destroyed to avoid memory leaks.

The addition of sketch parameters is realized by `push_sketch_param` method of the GcsWrapper. After the parameter is pushed, it can be referenced inside the constraints, as shown in the Listing 3.2:

```
gcs_wrapper.push_sketch_param('my_distance', 100);
gcs_wrapper.push_primitive({
    type: 'p2p_distance',
    id: 3,
    p1_id: 1,
    p2_id: 2,
    distance: 'my_distance',
});
```

Listing 3.2: Adding a constraint solver primitive with a referenced parameter.

After the sketch has been defined, the result of the planegcs solving is obtained, and if it is successful, the sketch stored in GcsWrapper can be updated by the `apply_solution` method. This pulls the solved p-parameters from planegcs and updates the sketch primitives' positions accordingly.

```
const status = gcs_wrapper.solve_system();
if (status === SolveStatus.Success) {
   gcs_wrapper.apply_solution();
}
```

3.3.4 Testing

The wrapper library includes multiple unit tests – (i) for parsing functionality within the build process, then (ii) for testing of the `GcsWrapper` logic using a mocked version of the WebAssembly module, and finally (iii) tests for planegcs that check whether the binding code is working properly. Also, given the limited sources of information about planegcs, it may serve as the documentation, too. The tests are written using the *Vitest*[18] testing framework and can be invoked by `npm test`.

Tests that use mocks can be executed without the WebAssembly module. Nevertheless, for running the tests for planegcs bindings, the WebAssembly file must be present and executable from within the testing framework. While I tried another testing framework – *Jest*, it had issues executing WebAssembly. Therefore, I chose Vitest for its native support of WebAssembly and TypeScript.

3.3.5 Limitations and future work

The abstract data model of the wrapper library currently has some restrictions. One such limitation is that there is no way for a constraint to reference a value of another constraint, which is otherwise possible in the C++ code. Therefore, constraints can include only parameters, whose value is *known before solving*. This prevents for example direct adding a constraint on equality of two angles.

From the performance perspective, the parallelization of the QR decomposition algorithms could be turned on for WebAssembly, to-

18. `https://vitest.dev`

gether with the possibility of changing the way the p-parameters are transferred from JavaScript to WebAssembly and back.

Another source of limitation comes from the numeric solvers used in planegcs, where it is advised by the FreeCAD users not to create bigger sketches having more than a hundred constraints, otherwise, the solver becomes ineffective. Some graph-based decomposition methods could be implemented on top of the numeric solver to obtain good results even for more complex sketches.

4 Sketcher module

This chapter presents Sketcher – a user interface for parametric sketching. It is based on the planegcs solver and the wrapper library described in the previous chapters. Sketcher makes it easy for end users to use the solver's low-level functionality through visualization and sketch manipulation tools.

Sketcher is implemented as a module extending the functionality of Designer, a web-based application for parametric furniture modeling. Sketcher gives the user the ability to work in two modes: (i) sketching on an arbitrary face of a 3D model or (ii) creating a reusable contour. If well-defined, the resulting sketch can act as a flexible shape and serve as an input for other Designer functions, such as extruding the 2D profile into a 3D object. Essentially, this and many other functionalities would not be possible without 2D parametric sketching abilities.

The first Section 4.1 introduces Designer, its technology stack, and APIs relevant to Sketcher. This is followed by a description of the sketch data model and the interface between Sketcher and planegcs in Section 4.2. Finally, the user interface development details are presented in Section 4.3.

4.1 Designer – parametric furniture modeler

Designer is a web application written in TypeScript. It aims to perform all the computations in the client's browser and only communicate with the server to interchange data with the database. My contribution to the Designer is the development of the Sketcher module and adding or improving some Designer's APIs, which are mentioned later. The technologies used for the development were mostly predetermined by the existing Designer's codebase. They are shortly introduced in the following text.

Technology stack

Svelte[1] – an open-source front-end framework. The code is structured into `.svelte` components, which combine JavaScript, HTML, and CSS. Svelte is not only a library but also a compiler, which makes it unique among other JavaScript frameworks. The recommended way to build the projects is using *SvelteKit*[2].

Three.js[3] – an open-source JavaScript library for 3D computer graphics based on a low-level API for graphics in browsers – WebGL[4]. Three.js includes rendering algorithms and functions for 3D vector and matrix operations.

Threlte[5] – a Svelte wrapper for Three.js, which exposes its API via the declarative syntax of the Svelte components.

Reactivity and state management in Svelte

The main concept of each frontend framework is *reactivity* – synchronization of state between JavaScript and HTML. Inside the Svelte components, variables defined in the top-level scope are reactive by default [28]. Generally, they are used for managing the local state of the component instance. Sharing the state among components can be done either by component *props* (i.e., reactive properties passed from a parent to a child) or by so-called *stores*. Stores are isolated data containers that implement the Observer pattern [29]. The Svelte compiler provides a syntactic sugar that simplifies working with the reactive value of the store by prefixing it with \$ [30].

In simple applications, stores are used as global state containers. More complex scenarios can be isolated in a specific Svelte *context*. The Svelte function `setContext` accepts an arbitrary object and makes

1. `https://svelte.dev`
2. `https://kit.svelte.dev`
3. `https://threejs.org`
4. `https://get.webgl.org`
5. `https://threlte.xyz`

it accessible to all descendants of the current component. This allows simple dependency injection scoped for a specific component subtree.

Feature-based evaluation

Designer is based on feature-based modeling, a traditional approach to parametric CAD modeling. In this approach, each product is defined by a sequence of operations called *features* – a "recipe" for building the product. The dependency between features can only be in the forward direction. In other words, the output of earlier features may influence the following ones, not vice versa. In Designer, every product definition starts with an *initial feature* – an empty 3D space of given dimensions. Subsequent features are added and configured by the user.

Each feature is evaluated to obtain the final result, and its output is stored in an object called *feature evaluation context* (also simply evaluation context; not to be confused with Svelte contexts). As an example, Designer has the *Variable* feature that accepts two fields: *name* and *value*. Evaluating this feature results in the name/value pair being added to the evaluation context, which makes it accessible to features that follow.

Scene items, hooks, slots

The objects rendered in a 3D scene are called *scene items*. They can be of three basic types: vertices, edges, or faces. If a feature evaluation generates any scene items, they are added to the evaluation context. Subsequently, they are rendered to the 3D scene using Threlte-specific components.

Each scene item is uniquely identified by a composite ID object. These ID objects are obtained during the selection mechanism, which uses utilities from Three.js for *raycasting*[6]. Raycasting is the process of projecting an imaginary line, or "ray", from the camera through the mouse cursor's position to identify intersecting objects. The result of raycasting is an array of ID objects.

I contributed to new Designer APIs, because of specific Sketcher requirements, such as overriding the default behavior of the editor,

6. `https://threejs.org/docs/#api/en/core/Raycaster`

showing tool panels, etc. This includes *Hooks*, which are a way of adding custom TypeScript handlers for pre-defined spots in code execution. They are used for keyboard shortcuts, mouse event handling, etc. Similarly, *Slots* are used to inject Svelte components to predefined places in the DOM.

4.1.1 Sketch and Contour features

The Sketcher module – its user interface and evaluation logic – is implemented by two features – *Sketch* and *Contour*. To override the default behavior of the Designer, these features register custom hooks and slots.

In the Sketch feature, the user first selects the face of an existing model. From this face, a coordinate basis and a transformation matrix are determined for rendering the sketch in 3D. A helper rectangle indicates the position of the active sketching face (see Figure 4.1).

On the contrary, the purpose of the Contour feature is to create a reusable sketch that could be applied within the configuration of subsequent features. Because a contour is independent of any previously generated object, the scene is modified to hide all elements from other features. The camera position and rotation are fixed so that the 3D effect disappears and the scene simulates a 2D canvas. A unique aspect of the Contour feature is a set of four *reference points* to which the geometries can be constrained. The position of these points is influenced by configuring the width or height in the contour configuration.

The code for these two features is in `src/lib/feature` in their respective folders and is exclusively my work as opposed to the rest of the codebase. Most code from the Sketch feature is also used in Contour. Similarly, both features use the same state object – a Svelte context called `sketcher_context`.

4.2 Sketch data model

As the sketching functionality is based on a 2D solver, all the sketch objects are internally stored in 2D as well. The data model must nonetheless reflect what entities the user sees and interacts with because the

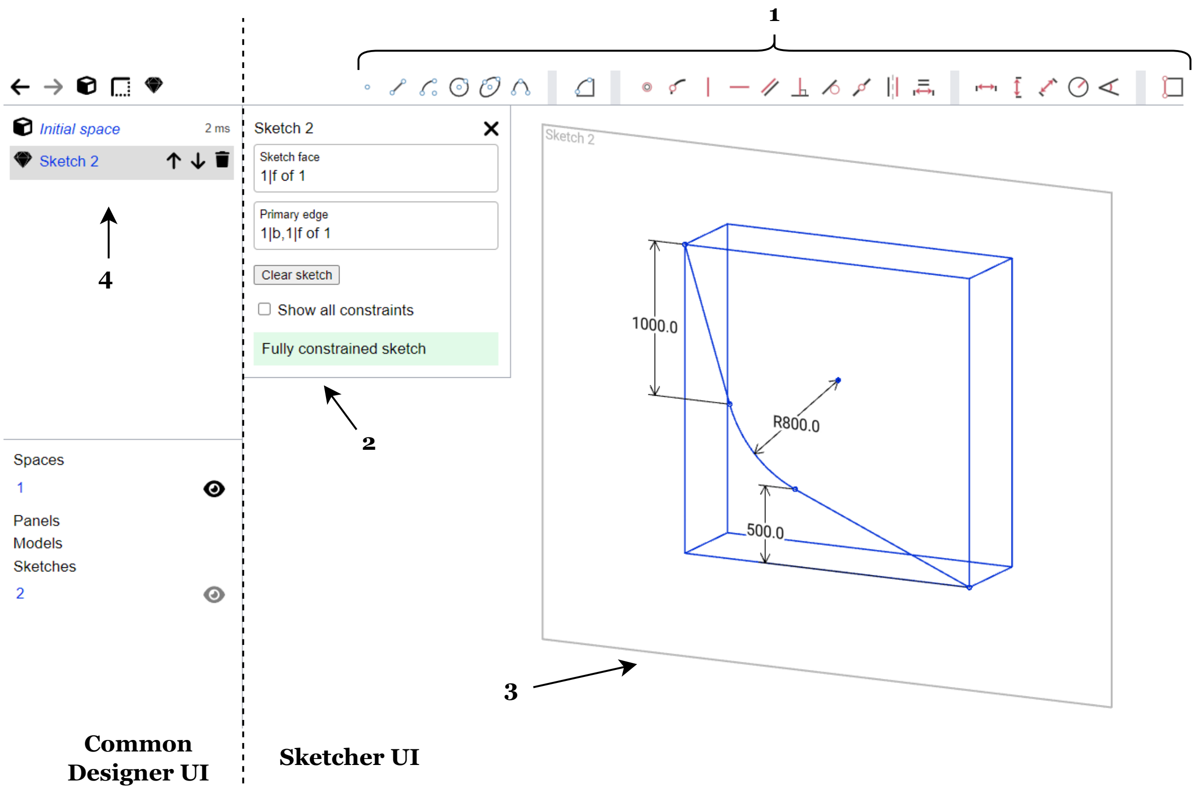


Figure 4.1: Sketch feature of the Sketcher user interface. This mode renders the sketch on top of an existing geometry generated by preceding features. While the left part is common for all Designer features, the right part of this image is specific for Sketcher. At the top (1) is a panel of tools for sketch modification (drawing and constraint). The options box (2) includes the configuration of the sketch location and basic controls. It also includes info related to geometric solving. The position of the sketch plane is visualized using a gray rectangle (3). The Sketch feature is defined along with other features visible in (4). The sketch shown in the picture contains one arc and two line segments. The arc is constrained to have a fixed radius and to be tangent to both lines, and their endpoints coincide. This combination of constraints is also called "smooth join". The line segments' other endpoints coincide with the corner vertices of the initial space. The vertical distances of the lines' endpoints are set, too. This makes the sketch fully constrained, as shown in (2).

solver primitives are too low-level. In my implementation, I aimed to create only a simple abstraction that would reuse the solver's model. Importantly, I did not create a completely new representation that would have to be transformed from and into the data for the solver.

I chose this approach mainly because the Sketcher was the first real testing ground for the planegcs solver and this usage shaped the wrapper library along the way. The disadvantage is that the higher-level data model is bound to this specific solver. However, I realized that the solver specificities influence the behavior of the user interface, which makes it bound to the solver anyway. For example, planegcs regards all angles as oriented, which differs from the observed behavior of other (commercial) solvers.

4.2.1 Sketch structures and sketch definition

The key observation is that the solver primitives can be separated into disjoint groups called *sketch structures*. For example, the *line primitive* contains only IDs of its endpoints, making it a mere link of two points that need to be defined a priori. Therefore, the group [P1, P2, Line(P1, P2)] can be regarded as a single entity – the line *structure*. More complex geometry structures are created similarly; they simply contain more primitives (usually points and internal constraints specific for planegcs).

Each solver primitive belongs to exactly one sketch structure – no primitives are shared or reused. This approach brings a small amount of redundancy but suitably simplifies subsequent operations in the sketch. Here, I prioritized establishing consistency in the data model. Importantly, ad hoc optimizations can be later added to remove these redundancies and speed up the solving.

The sketch structure does not have its own distinctive ID. Instead, it is identified by the ID of the primitive at the zeroth position. This simple concept of grouping primitives has one advantage over other possible abstractions: converting from the sketch structures to planegcs primitives only requires flattening an array. No further data transformations are necessary. Moreover, since the sketch structures are JavaScript objects, they are easily serializable to JSON, as demanded by Requirement R7.

The collection of all sketch structures that together make up the content of a single sketch is called *sketch definition*. The primitives from the sketch definition are sent together with sketch parameters to the solver and retrieved after solving.

4.2.2 Solving process as feature evaluation

In Designer, each new instance of a sketch definition is bound to a single feature – Sketch or Contour. As a result of the user's interaction, the currently active feature data is changed, which triggers a reevaluation of the current and all subsequent features (which may depend on the output of the current one).

The solving step takes place in the (re)evaluation process of the respective feature. Based on Requirement R5, all the processing is performed client-side, without any communication with the server. Apart from solely executing planegcs, a series of actions are performed:

I. Analysis and optimization of the sketch

Several analysis methods were implemented, which are used in multiple places in the Sketcher codebase. One of them deals with point coincidences in the sketch. In the sketch definition, the information about point coincidences is scattered among the individual constraints. To quickly obtain all (transitive) coincidences of a chosen point, the *union-find* class (also known as disjoint-set) [31] is used. This class implements two operations: *union* for combining two sets, and *find* to get the representative of a set the element belongs to. One example of its usage is visualization of points – instead of rendering every single point in the scene, only one per coincident group is chosen.

Another part of the sketch analysis is building the "reverse index" of constraints. It is a simple mapping that returns an array of related constraints for a given ID of a sketch primitive. This is used e.g., in searching for unconstrained primitives or visualization of constraints as described in Section 4.3.5.

Along with the (read-only) analysis methods, Sketcher also uses an optimization that modifies the sketch definition before solving to improve the solver's convergence. One problem of planegcs is that tangency constraints are handled ineffectively, as proved experimentally, and also mentioned by the FreeCAD community. In certain situations, they can be replaced with better-performing alternatives. The optimization looks for so-called *smooth joints*. Smooth joint configurations (when two connected segments are constrained to be tangent) are common in parametric modeling. The optimization iterates through all tangency constraints from the sketch definition and using the union-find class, it detects whether any of the constrained segments' endpoints are coincident. The smooth-joint tangencies are then replaced with planegcs' `angle_via_point` constraints. Without this optimization, the user experience might get very poor. Therefore, detecting and fixing smooth joints is critical for Sketcher.

II. Adding sketch parameters

Sketch parameters are simple name-value pairs that can be passed to planegcs. When the parameter name is referenced within a constraint primitive (see Listing 3.2 for comparison), its value cannot change during solving.

This solver API was used to implement multiple functions of the Sketcher. The first use case deals with *variables* and *expressions* according to Requirement R2. Let's consider a situation where the user already defined a variable `r = 10` via the Designer's Variable feature, and they use the expression `r * 3` for constraining a circle radius. In this setting, the *whole expression* is used inside the constraint primitive. Since the solver itself does not include any API for evaluating expressions, it is necessary to evaluate it a priori and add it as a sketch parameter to the solver, so that it can look up its value by its name – in the above example, the name is the whole expression string `r * 3`, and the value is computed to be `30`.

The second use case is referencing *external* (*spatial*) *geometries* generated by features different from the one being evaluated. The external geometries are available as scene items in the evaluation context. The approach is to separate individual attributes of the external geometries and use them as parameters for the solver. For example, an external point is broken into its x and y coordinates. The names of these parameters are made unique by special prefixes. This is the basis for implementing the projection described later in Section 4.3.8.

III. Planegcs solving

The planegcs *solve* method can be invoked with an argument that changes the used algorithm. However, this argument is ignored when the sketch contains constraints with the temporary flag.

Therefore, the sketch definition is first analyzed for the presence of temporary constraints. If there are any, then the solve method is invoked only once. Otherwise, it is invoked with each of the algorithms until the solution is successful or all attempts fail. This way of using the solver API is inspired by the implementation of FreeCAD's Sketcher.

If the solution is accepted, it is used as a new sketch state. Otherwise, the sketch remains unchanged. Custom heuristics are applied besides the solver's internal mechanisms of detecting a failed solution. They improve the total ergonomics of the Sketcher by dismissing some solutions – for example, those that include any arc with a zero radius. Such arcs decrease the convergence success of the solving in the subsequent evaluations.

IV. Conversion to scene items

Through a series of actions, the two-dimensional geometries represented by the sketch structures are segmented (particularly the elliptical geometries), and closed paths are detected. Finally, they are converted into sets of 3D scene items – vertices, edges, and faces over the closed paths, which all participate in the rendering of the scene, as requested by Requirement R6. This conversion was developed in collaboration with the Designer's team.

V. Result of the solving process

After the Sketch or Contour feature is evaluated, its result is added to the evaluation context as the `SolvedSketch` interface. By that, it becomes accessible to the rest of the application. This object contains multiple data entries:

- the accepted version of the sketch definition,
- planegcs solver-related metadata (solve status, redundancies, conflicts, . . .),

- the outputs of the post-solving analysis,
- the 3D scene items, and
- the location and orientation of the sketch used for coordinate transformations.

4.3 User interface for sketching

The following section describes the implementation details of the Sketcher user interface. This interface is activated within Designer for two features of Designer (Sketch and Contour, with slight differences). It allows the user to visually interact with the sketch geometries and constraints as defined in Requirement R1.

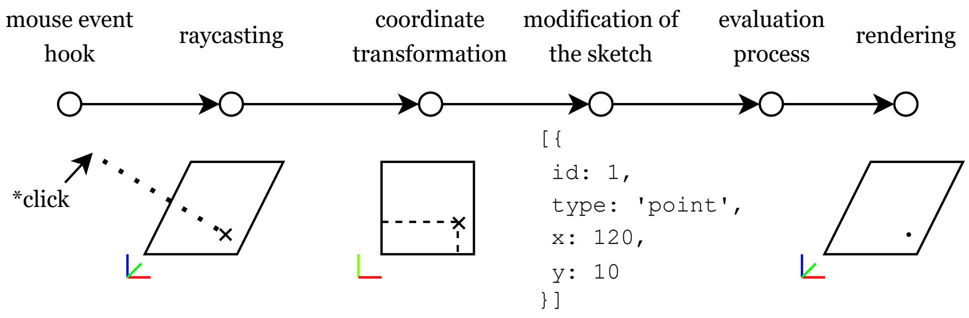
4.3.1 Drawing geometries

The first part of sketching is drawing geometries. A drawing tool can be activated by clicking its icon in the Sketcher toolbar (see Figure 4.1). While the drawing tool is active, the rotation of the 3D scene is fixed.

In the current implementation, *any* change in the sketch is realized by changing the sketch definition, which triggers reevaluation of the given feature and subsequent rendering of respective scene items. This exact mechanism also happens while drawing new geometries – it is a change of the sketch definition, which, in effect results in executing the solver. A consequence is that the code for drawing geometries must deal with the specificities of planegcs.

This design decision was motivated by using the drawing tools as a testing ground for the solver during the development of Sketcher. It leads to a unified way of visualizing the sketch geometries but also to a more complicated code of the drawing tools. This could be modified in the future so that the solver is executed only when necessary.

The Sketcher offers tools for drawing several geometries: Point, Line segment (Polyline), Circle, Circular arc, Ellipse, and Elliptical arc. The code for drawing arcs, ellipses, and elliptical arcs is more involved than the rest. This is caused by the internal representation of these geometries in planegcs. The arc is always oriented counterclockwise (given its start and end point), and this direction cannot be easily



```
[{
 id: 1,
 type: 'point',
 x: 120,
 y: 10
}]
```

Figure 4.2: A series of steps performed when drawing a point.

swapped when needed. A similar issue happens for the ellipse (and elliptical arc) and switching between its major and minor radius. All these swaps must be handled in the drawing tools' code.

The code of the drawing tools is located in the `src/lib/feature/sketch/drawing` folder as a series of TypeScript modules. Each module contains two functions: `click` and `mouse_move`, which are handlers for the respective mouse events. These methods are invoked by the mechanism of editor hooks. When a mouse event is observed, raycasting determines the 3D intersection point with the sketch plane. Consequently, this 3D point is transformed into the 2D sketch coordinates, and the handler of the drawing tool is invoked. The handler modifies the definition of the sketch, which causes the reevaluation and subsequent rendering. Figure 4.2 illustrates this process for the case of adding a point.

4.3.2 Adding constraints

To constrain an existing sketch geometry, the user must first select some geometries and click on an icon of a constraint tool. These are located in the tool panel, next to the icons for drawing (see Figure 4.1). After initial testing, the constraint icons were differentiated by a red color for better clarity.

Sketcher offers constraints that are common in parametric CAD systems: Coincidence, Point on Object, Vertical, Horizontal, Parallel, Perpendicular, Tangency, Midpoint, Symmetry, and Equality as the non-dimensional constraints, and Horizontal Distance, Vertical Distance, Length, Radius, and Angle as dimensional constraints. The behavior of these tools was inspired by a document called *Sketcher Lecture* [32], created by the members of the FreeCAD community.

The code of the constraint tools is located in the `src/lib/feature/sketch/constraint` folder. Similarly to the drawing tools, the constraint tools are each defined in a separate TypeScript module. However, their interface is different – they implement a single function `apply`, which is invoked upon activating the constraint tool. This function takes an array of objects that identify the selected scene items. These ID objects must be mapped to the sketch structures. This makes it possible to check that the selected geometries are correct given the tool type. If not, an error message is shown. If correct, the tool modifies the sketch definition by adding the respective constraint(s), based on the selection.

If a dimensional constraint is applied, then it is necessary to compute the value of the dimension based on the current state of the sketch. This ensures the sketch does not unexpectedly change after applying the constraint. In the current version of Sketcher, this is not yet handled for every possible combination of geometries and remains for future work.

The Angle constraint is an example of how the design of a geometrical solver influences some aspects of the user interface. In planegcs, the angle is always *oriented*. This must be respected in multiple places in Sketcher, for example when precomputing the angle or visualizing the dimension. Interestingly, other CAD solutions based on different solvers do not enforce the angle direction.

4.3.3 Fillet tool

The Fillet tool inserts a tangent arc (i.e. an arc with tangency constraints) between two segments, as shown in Figure 4.3. First, the coincidences related to the segments' endpoints must be removed (otherwise, the endpoints could not be moved from each other). Afterward, the tool moves the endpoints (L1, L2), connects them to the new arc's endpoints (A1, A2), and adds tangency constraints. When the segments' endpoints coincide with another point P, it is necessary to add constraints that keep the line segments aligned with P (as visualized by the dotted lines in Figure 4.3).

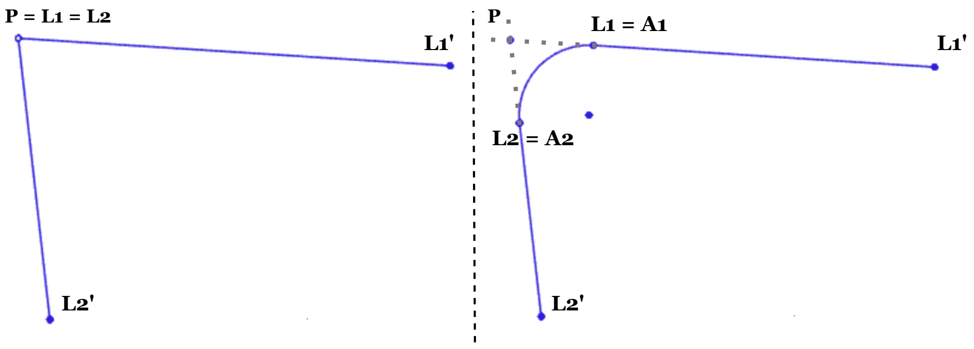


Figure 4.3: Effect of the Fillet tool. The picture shows both states – before (right) and after (left) applying the tool. On the right, there are two line segments L_1L_1', L_2L_2', and an (unrelated) point P, which coincides with L_1 and L_2. The tool moves L_1 and L_2 toward L_1' or L_2', respectively, and a tangent arc is added. Its endpoints A_1 and A_2 are set to coincide with L_1 and L_2. L_1' and L_2' are not affected by the tool. In this case, the tool constraints P to lie on lines L_1L_1' and L_2L_2', as marked by grey dotted lines.

The Fillet tool is an example of how a structural change in geometry also requires changes in the constraints. Moreover, multiple cases must be explicitly handled in the logic to keep the sketch consistent after the user interaction.

4.3.4 Mouse dragging tool

The tool for dragging has a unique interface with three functions: `on_drag_start`, `on_drag_move`, and `on_drag_end`. When the Sketch / Contour feature is activated, it automatically registers special hooks that override the default mouse behavior. Among other tasks, they handle the automatic activation of the dragging tool. This tool only activates when the mouse hovers over a particular sketch geometry. To give visual feedback to the user, this geometry is highlighted (i.e., colored differently).

The movement of the sketch geometries towards the mouse is realized by using constraint primitives with the *temporary* flag set to true. Without that, these constraints could conflict with existing constraints in the sketch, resulting in a solver error. The different behavior of the temporary constraints is optimal for this use case.

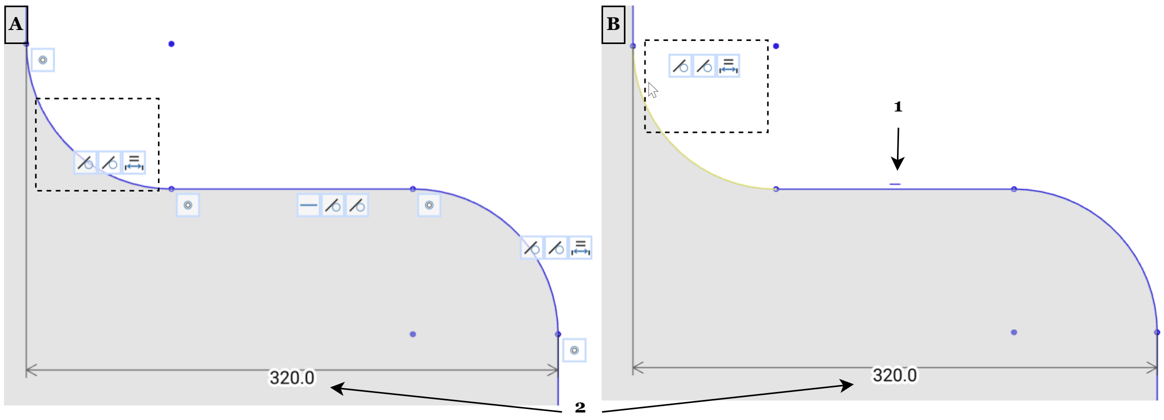


Figure 4.4: Detail of a sketch with constraint icons. Sketcher shows the constraint icons in two modes – (A) all icons and (B) only for the hovered geometry. In B, helper marks for horizontal and vertical constraints are displayed (1). Note the change of position of icons within the dotted rectangle – in A, they are near the center of the arc; in B, they are located near the mouse cursor. The dimensions are displayed in both modes (2).

When a more complex geometry is dragged, adding some extra temporary constraints is beneficial to minimize unwanted changes in the sketch caused by the solver. For example, when dragging the center point of an arc, the arc's radius is constrained to remain unchanged. Similar constraints were chosen based on experimenting and observing other CAD software. These extra constraints are further prioritized via setting the *scale* property.

4.3.5 Visualization of constraints

As mentioned before, there are two types of constraints: *dimensional* and *non-dimensional*. As a result, these groups differ in their visual representation. The former are shown as technical dimensions. The latter are presented as small icons near the constrained geometries. By default, the icons are shown individually at hovered geometries. This can be changed by setting the flag "Show all constraint icons". Figure 4.4 shows one dimension and multiple non-dimensional constraint icons in both modes.

Non-dimensional constraints

The icons for the non-dimensional constraints are rendered using a special `<HTML>`[7] component from Threlte. This component acts as a wrapper for regular HTML content – in this case, a button. When the user clicks the button, a box appears with the constraint's name and a delete icon.

Threlte automatically applies 3D CSS transforms that make the content *look like* it is a part of the 3D scene. This relatively simple way of adding interactive elements has one limitation. The content is always rendered on top of the scene, therefore it cannot be (partially) occluded by other objects. However, this property is an advantage for the constraint icons, because they stay visible to the user.

A simple grouping for icons of constraints of (nearly) overlapping geometries is applied. After initial testing, the grouping distance threshold was set to a low value. As a result of grouping or when there are more constraints on a single geometry, the buttons are displayed next to each other.

The position of the (group of) constraint icons is based on the visual center of a sketch geometry. This is overridden in mode B (see Figure 4.4), where the position is set to be near the mouse pointer. This mode also contains the logic of showing and hiding the buttons, which ignores very quick movements and keeps the icons visible for a fixed amount of time when the mouse pointer leaves.

Dimensions

In the case of the dimensional constraints, the standard is to visualize them as technical *dimensions*. There are multiple types of dimensions: (A) the distance dimensions, which are rendered as a line (horizontal, vertical, or parallel to the constrained geometry) that is translated by a given offset; (B) the radius dimension has three versions based on the ratio of current zoom and the length of the radius; and (C) the angle dimension. When the numeric value of the dimension is clicked, an HTML editable expression input is shown, which allows the user to change the value. This input is a component from the Designer. Each dimension variant is depicted in Figure 4.5 including the editable state.

7. `https://threlte.xyz/docs/reference/extras/html`

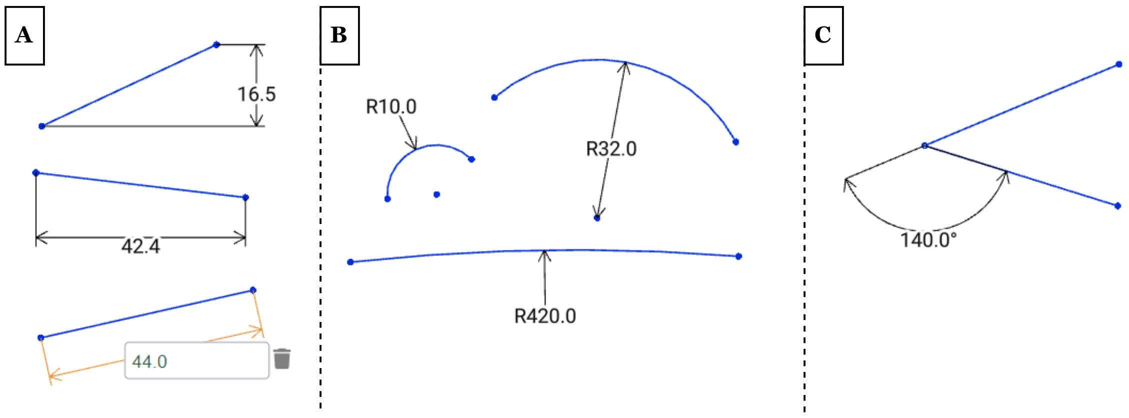


Figure 4.5: Multiple variants of the dimensions: (A) Vertical Distance, Horizontal Distance, and Distance constraints; (B) the Radius constraint – three variations; and (C) the Angle constraint. The editable state is shown for the Distance constraint in (A).

In the current implementation, the position of the dimensions is fixed – they are not draggable. Instead, their offset is updated when the zoom changes to look constant to the user. The radius dimension has three variations based on the amount of perceived space between the center and the arc.

4.3.6 Deleting geometries and constraints

After selection, deleting objects from the scene can be triggered by pressing the `Del` key. For the constraints, it is a trivial deletion of the given sketch structure from the sketch definition. In the case of geometries, however, multiple things must be handled.

First, the selected geometries are filtered based on whether they are defined as deletable. For example, the center point of a circle alone is not considered deletable. This behavior is inspired by other CAD applications. Second, constraints related to these objects must be deleted, because otherwise, planegcs would return an error of a missing primitive with a given ID.

4.3.7 Solver status messages

The output of the planegcs solver consists of multiple metadata – the number of Degrees of Freedom (DOF), a list of constraints considered redundant or failed, and the solving status. If the status is successful, the user is notified whether it is an under-constrained sketch and how many DOFs remain to get to the well-constrained state.

Based on Requirement R4, the user is informed about the solving status in a side box under the configuration of the Sketch or Contour feature (see Figure 4.1). If any problems are detected, such as redundant or conflicting constraints, a warning message box with general recommendations is shown. In addition, the respective constraint elements (icons and dimensions) are colored red to indicate that the user should pay attention to them.

4.3.8 Projection of spatial geometries in 3D

Explicit projection of spatial geometries is available only in the Sketch feature. Besides, there is also an implicit type of projection. It is automatically performed when the user applies a constraint to a selection that includes scene items that are not an output of the currently active feature. Explicitly projected geometries are shown in black, while the implicit ones are invisible to the user and are only used for the solver. In both cases, they become part of the sketch definition in the form of distinctive sketch structures bound to the spatial geometries. This binding is accomplished by adding equality constraints within the sketch structure for each of the geometry primitives' properties. For example, a line structure would contain 4 extra equality constraints (x and y for both endpoints). Each such constraint binds one property to a predefined parameter name. The values of the parameters are obtained by the orthographic projection of the geometries and added to the solver as described in Subsection 4.2.2.

The orthographic projection of every geometry type is based on projecting its points via methods available in Three.js. While the projection of point and line geometries is straightforward, the process is more involved for a conic (i.e., circle, arc, ellipse, or elliptical arc). First, four control points are determined by the intersection of the ellipse with its major and minor axis (in the case of a circle, the x and y axes are used). These points are projected to obtain the so-called *conjugated half-diameters*. From them, the resulting elliptical shape is reconstructed using Rytz's construction [33]. To reconstruct an arc, its start and end points are projected, too.

As an effect of the (orthographic) projection, geometries might change their types. For example, a circle can be projected as an ellipse, or a line, when the source and target planes are perpendicular. This conversion is detected when the projection tool is applied. Then, the final sketch structures are added to the sketch definition and sent to the solver. The main reason for the conversion is that the different types of sketch structures are mutually incompatible with respect to their applicable constraints.

A change in the orientation of the source or target sketch face can cause the original type of the projected geometry to be invalid. This subsequent conversion cannot be automatically applied for two main reasons: (i) the projected geometry can already be constrained, and the change would invalidate these constraints, and (ii) the evaluation of one feature could change the *definition* of another feature, which is an unwanted behavior. Another invalidation case is when the source geometry is deleted.

5 Evaluation

To evaluate the usability of the Sketcher module, identify its potential weaknesses, and get general feedback, I conducted several test sessions. They were held in Czech to minimize the possible language barrier. In all sessions, the users used Chrome or Firefox on Windows and Linux. First, informal testing of a Sketcher prototype was carried out with the client which showed several major shortcomings. After implementing the key missing features, I held an informal (group) session with the client's working professionals. The focus of this session was to present Sketcher, get feedback on the introduction, and help shape the final questionnaire.

The main part of the evaluation consisted of one-to-one think-aloud sessions with five participants working with the system for the first time. One person participated in the pilot study and four others in the final testing. The pilot study helped to find weak spots in the tasks' instructions, several bugs, and missing features. After fixing the reported issues, the final session was prepared. Its form is presented in the following text.

The testing focused exclusively on the Contour feature – the Sketch feature and other Designer features were not present in the test version of the application. This ensured the testing was targeted at the sketching functionality, not other specifics of Designer. Keeping a reasonable time limit for each session was also an important factor.

5.1 User study introduction

Each session in the final testing was started by introducing the study's purpose and the Sketcher's background. I showed the participant a few real-world sketches I prepared in advance with Sketcher using the Contour feature. The user was asked if they agreed with the screen and voice recording. Then I demonstrated the basic functionality of Sketcher, the mouse and keyboard controls, and explained the concept of the contour's reference points. The participant had 5-10 minutes to practice sketching independently and ask questions about its use. Subsequently, the rest of the constraint tools were shortly demonstrated.

During that, the principle of the value of Degrees of Freedom was explained.

5.2 User tasks

The second part of the study contained three simple introductory and five follow-up tasks. Each task was formulated by (i) the instructions, (ii) an image of a sketch (see Figure 5.1), and (iii) a target value of Degrees of Freedom so that the participant could distinguish the finished task. The tasks were given to the users in printed form and are attached in Appendix A.

The participants could ask questions freely about the interface during the introductory tasks. These tasks were included to ensure the users understood the instructions and recognized when a task was completed. They consisted of drawing simple lines and using the Fillet tool. No geometric reasoning was necessary for completing these tasks because they were formulated explicitly and contained icons of tools that should be used.

In contrast, the follow-up task instructions were slightly more high-level, mostly without explicitly stating which constraints should be used. As a result, the tasks could be achieved in various ways. The first follow-up task was to create an equilateral triangle. The rest were closed curves (lines, arcs) that were taken from examples used in furniture design. The users were instructed to ask for help only when stuck. Each task started from scratch by creating a new Contour feature. One exception was Task 4, which instructed modification of the previous output (see Figure 5.1).

5.2.1 Observations

Before the final testing, I chose several aspects to focus on during the individual session. I grouped them together into three groups, which are described below.

Intuitivity of Sketcher's interface and missing features

The users reported multiple features they missed from other CAD applications that they considered important for intuitiveness and us-

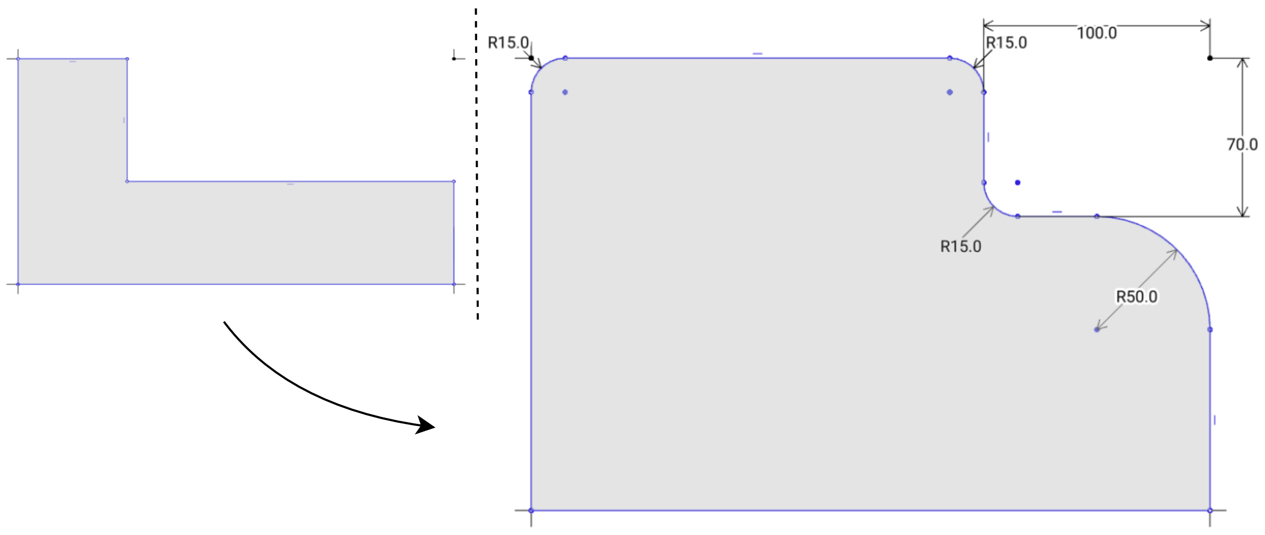


Figure 5.1: Example of two sketches from the user testing tasks. First, the participants were instructed to draw the left sketch (Task 3) and transform it into the right sketch (Task 4). Both are created in the Contour feature, which includes four reference points (black with corner markers).

ability. For example, most users mentioned the "orthogonal mode" to automatically add horizontal or vertical constraints while drawing or editing the geometries. Most participants would like to activate the constraint tool first and then apply it to the geometries (the current version requires first selecting the geometry and then applying the tool). One participant mentioned different coloring of the well-constrained parts of the sketch and the possibility of converting lines to arcs. Regarding the selection mechanism, one user would like to use a selection rectangle for a bulk selection.

Overall, one specific repeated behavior was observed very often. After drawing a closed shape, every participant started intuitively dragging the geometries, but the drawing tools were configured to remain active. This was noticed only after drawing an unwanted geometry. Although the users did not explicitly report it was an obstacle, I consider changing the drawing tools' default behavior, which could reduce the number of useless actions.

Confusing situations and concepts

The interaction with Sketcher could lead to situations where the solver failed for non-obvious reasons. In these cases, the users typically asked what went wrong. One such "dead end" was caused by tangency constraints involving line segments of zero length. This happened to one user, who created a fillet (arc with two tangent line segments), but then reduced it back to an arc. In this case, the user was instructed to delete the fillet and start over to finish the task.

Another user created a circle instead of an arc, leading to confusion, because Sketcher does not include any tool for changing a circle to an arc. Again, I instructed the user to delete the circle and restated there is a tool specifically for drawing arcs.

Several situations confusing for the participants were caused by bugs in the Sketcher interface. One was related to breaking the transitivity of coincidence constraints by deleting sketch geometries, which happened during the pilot test session. Another discovered bug was the improper selection mechanism for contour reference points.

User 4, otherwise very fluent in task solving, was confused by a case where the Fillet tool automatically inserted constraints and the solver immediately marked them as redundant. This situation could be handled automatically by testing whether any automatically added constraint is redundant before adding it.

Problematic geometric subtasks

I identified two situations, where the users struggled or asked for help and could possibly be avoided by a better user interface. The first case involved constraining a line segment to lie between two contour reference points without directly connecting its endpoints to any of them. Two participants asked for help with this task. They also suggested adding dotted reference lines between the reference points, which could be used to automatically snap the geometry.

In the second case, the users were supposed to create a dimensional constraint for a horizontal/vertical distance. After selecting a point and a line segment, an error was displayed that they must choose two points. This rather usual case could be automatically handled by a more robust logic of the tools.

ID	Statement
I	What is your experience with parametric modeling?
S1	The introduction with examples was overall understandable.
S2	I found navigating more complex sketches with more constraints easy.
S3	The drawing and constraint icons were confusing.
S4	The control of the drawing tools was intuitive.
S5	The control of the constraint tools was intuitive.
S6	I understood from the error messages what I should do differently.
S7	The system often behaved differently than I expected.
S8	I can imagine that I would get used to the system after longer practice.
S9	I found the system very cumbersome to use.
S10	I would like to use this system frequently.
S11	I found the system unnecessarily complex.
S12	I felt very confident while using the system.
S13	For usability, I missed features that I know from other applications.

ID	Type	User 1	User 2	User 3	User 4	Scores
I	7-expert	6	4	6	6	
S1	P	1	1	1	2	4, 4, 4, 3
S2	P	4	3	1	2	1, 2, 4, 3
S3	N	3	5	4	5	2, 4, 3, 4
S4	P	3	1	3	1	2, 4, 2, 4
S5	P	3	1	2	1	2, 4, 3, 4
S6	P	4	2	2	2	1, 3, 3, 3
S7	N	2	5	3	4	1, 4, 2, 3
S8	P	1	1	2	1	4, 4, 3, 4
S9	N	2	5	3	5	1, 4, 2, 4
S10	P	-	2	3	2	2, 3, 2, 3
S11	N	-	5	4	5	2, 4, 3, 4
S12	P	5	4	2	2	0, 1, 3, 3
S13	N	1	5	1	4	0, 4, 0, 3

Table 5.1: Results of the user questionnaire. The statements S1-S13 were either positive (P), which means 1 (agreement) has the highest score, or negative (N) scored reversely. The initial statement (I) about parametric modeling experience used a 7-point scale.

5.3 Final questionnaire

In the last part of the user study, the participants completed a questionnaire regarding their experience using the Sketcher interface. The original form of the statements and the answers can be found in Table 5.1. The participants self-evaluated their expertise in parametric modeling on a 7-point Likert scale in the initial statement (I). The rest of the statements used the 5-point scale to indicate agreement (1) or disagreement (5). I designed the statements for the Sketcher parts I wanted to focus on. Three of them (S9-S11) were taken from the standard System Usability Scale (SUS) [34]. I did not include more SUS statements since that would make the questionnaire too long. The participants could omit answering a question when it did not make sense. The score of each statement was normalized according to the SUS scoring – answers for positive statements were scored as $5 - a$ and answers for negative as $a - 1$, where a is the original score given by the person. Unanswered statements were assigned the middle choice. The statements are presented below in four groups based on their sum of scores among the participants, where the maximum sum is 16.

Participants' background

Three users (1, 3, and 4) evaluated themselves as advanced in parametric modeling. User 2 marked the middle option. Users 1 and 2 stated experience mainly with the Imos software, a tool for non-visual parametric modeling used internally in their company. Users 3 and 4 were experienced in Autodesk Inventor, which is a tool for professional mechanical design. User 1 also mentioned AutoCAD and User 4 worked with SolidWorks and Catia V5.

Very good score

The participants uniformly answered that the introduction was well understandable (S1). Also, all agreed that they would get used to the system after a longer time (S8). In particular, User 1 stated having too little time to familiarise himself with the system.

Statement	Scores	Sum
S1	4, 4, 4, 3	15
S8	4, 4, 3, 4	15

Table 5.2: Statements with a very good score.

Good score

Statements regarding the tools and their icons had a bit worse scoring. This is in accordance with the users expressing they would like to use the tools in multiple regimes. User 3 reported the drawing tools were slightly less intuitive (S4) than the constraint tools (S5). Other users scored them equally. User 1 reported confusion by the icons due to lack of time and being used to different applications. User 4 suggested moving the Equal constraint icon to another position. The users mostly did not consider the system unnecessarily complex (S11).

Statement	Scores	Sum
S4	2, 4, 2, 4	12
S5	2, 4, 3, 4	13
S11	2, 4, 3, 4	13

Table 5.3: Statements with a good score.

Medium score

Orientation in the sketch with multiple constraints (S2) was harder for Users 1 and 2. Nobody fully agreed that the error messages were always helpful (S6). This points to the situations where the users struggled to fix the issue independently as discussed in Section 5.2.1. These situations may explain the scoring of the other two statements – that the system often behaved differently than they expected (S7) and that it was cumbersome to use (S9). The users did not fully agree that they would like to use the system often (S10).

Statement	Scores	Sum
S2	1, 2, 3, 4	10
S6	1, 3, 3, 3	10
S7	1, 4, 2, 3	10
S9	1, 4, 2, 4	11
S10	2, 3, 2, 3	10

Table 5.4: Statements with a medium score.

Weak score

Two statements' scores can be regarded as weak: feeling confident while using the system (S12), which was reported as low by Users 1 and 2. However, these users also reported having too little time or being nervous, respectively. Another statement with a low score was the satisfaction with the Sketcher functionality compared to other CAD applications (S13). The missing features mentioned by the users are discussed in Section 5.2.1.

Statement	Scores	Sum
S12	0, 1, 3, 3	7
S13	0, 4, 0, 3	7

Table 5.5: Statements with a weak score.

5.4 Summary

Overall, I consider the user testing successful for two reasons. First, all users completed the tasks with little or no intervention from me. They only needed my help when there was a bug in the application or a solver-specific issue. Users who experienced problems were more likely to report that they had little time to familiarise themselves with the system. All users agreed they would get used to the system with more practice. Secondly, the observations and subsequent discussions brought up multiple ideas for improvements of Sketcher. The users also reported the functionality of other CAD systems that they missed in Sketcher and how they would use it during task solving. As the tasks contained real-world sketches, I consider such feedback valuable for the project's future. With the final questionnaire, I checked that the observations of the users interacting with Sketcher were aligned with their perception.

Apart from the Sketcher interface itself, the user testing also helped to shape the introduction to the system, which was evaluated as well-understandable in the final questionnaire. This introduction with examples might be later used as a basis for future training sessions.

6 Conclusion

The goal of the thesis was to develop a web-based user interface for modeling parametric 2D shapes using geometric constraints. To achieve that, it was necessary to adapt a geometric solver for the browser environment. I created a JavaScript binding for an open-source solver planegcs from FreeCAD. I prepared a build process that outputs a WebAssembly port of the solver including automatic generation of TypeScript annotations. This process includes a script for downloading the current version of planegcs from the FreeCAD official repository and applying patches to it. The code was published as an open-source wrapper library `@salusoft89/planegcs` to GitHub and npm. This library has great potential applicability, especially in commercial software thanks to its LGPL license inherited from FreeCAD.

In the second part, I implemented the Sketcher interface as a module for Designer – a 3D web application for furniture modeling, based on the client's requirements. I created data structures for the abstraction of the solver data model and integrated the solving mechanism into Designer. I extended the existing Designer's user interface with new tools for drawing geometries, constraining, editing the sketch, filleting, and projecting spatial geometries from the 3D scene. This required collaborating on multiple Designer API changes to override or extend the 3D scene rendering, state management, and event handling mechanisms. To improve the performance and user experience, I applied several automatic optimization methods. For example, the tangency constraints in so-called smooth joints are automatically replaced with their better-performing alternatives. Another example of optimization is discarding solver outputs that may cause subsequent errors. With Sketcher, I fulfilled all the requirements discussed with the client. The implemented functionality makes it possible to model parametric curved shapes and create reusable parametric template sketches in the furniture design.

I conducted a user study with participants, who were instructed to use Sketcher for redrawing given sketches. The evaluation results were overall positive. The participants mostly fulfilled the tasks without serious issues. Nonetheless, several bugs and confusing situations were observed during the testing. The participants also provided useful feedback about features they missed from similar CAD applications. Their suggestions were incorporated into the plan for future work on the system.

Future improvements

For better ergonomics, features from other CAD software could be incorporated. This includes multiple usage patterns of the constraint tools, an automatic orthogonal mode for drawing lines, draggable dimensions, a rectangular selection, and other features the users missed.

An important area for improvement is reasoning about the state of the sketch. The aim is to avoid "dead ends" when possible – situations that are hard to recover from. This can be provided by better methods for sketch analysis. They can be used to suggest the next action to the user or preprocess the sketch before sending it to the geometric solver. Some of these changes depend on improving the planegcs wrapper library for the solver or using other algorithms on top of it. An example is the detection of under-constrained parts, which requires further exploration of the planegcs solver internals or graph-based algorithms.

The principle of the Contour feature and its four reference points is uncommon in other CAD software. Since this may be a new concept for some users, it could be more intuitive and easier to understand if its width and height were visualized in the sketch as fixed dimensions or if helper reference lines were added between the reference points.

Currently, a rapid change in the width or height of the contour can cause poor solver convergence. This generally happens when the state before solving is too far from the desired solution. One potential way to address this issue is by pre-scaling the sketch geometries so that the solver has a better starting point, which requires fewer iterations. Further possible performance improvements include adding more sketch optimization techniques or implementing the support for multithreaded execution within the planegcs wrapper library.

Bibliography

1. SHAH, Jami J. Designing with Parametric CAD: Classification and comparison of construction techniques. In: *Geometric Modelling. GEO 1998. IFIP — The International Federation for Information Processing*. Boston, MA: Springer, 2001, vol. 75, pp. 53–68. ISBN 978-0-387-35490-3. Available from DOI: `10.1007/978-0-387-35490-3_4`.
2. MICHELUCCI, Dominique; FOUFOU, Sebti. Detecting All Dependences in Systems of Geometric Constraints Using the Witness Method. In: BOTANA, Francisco; RECIO, Tomas (eds.). *Automated Deduction in Geometry*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 98–112. ISBN 978-3-540-77356-6.
3. GE, Jian-Xin; CHOU, Shang-Ching; GAO, Xiao-Shan. Geometric constraint satisfaction using optimization methods. *Computer-Aided Design*. 1999, vol. 31, no. 14, pp. 867–879. ISSN 0010-4485. Available from DOI: `https://doi.org/10.1016/S0010-4485(99)00074-3`.
4. ZOU, Qiang; TANG, Zhihong; FENG, Hsi-Yung; GAO, Shuming; ZHOU, Chenchu; LIU, Yusheng. A review on geometric constraint solving. *arXiv preprint arXiv:2202.13795*. 2022.
5. BETTIG, Bernhard; M. HOFFMANN, Christoph. Geometric Constraint Solving in Parametric Computer-Aided Design. *Journal of Computing and Information Science in Engineering*. 2011, vol. 11, no. 2, p. 021001. ISSN 1530-9827. Available from DOI: `10.1115/1.3593408`.
6. MADSEN, Kaj; NIELSEN, Hans Bruun; TINGLEFF, Ole. *Methods for Non-Linear Least Squares Problems (2nd ed.)* 2004. Available also from: `https://orbit.dtu.dk/en/publications/methods-for-non-linear-least-squares-problems-2nd-ed`.
7. NOCEDAL, Jorge; WRIGHT, Stephen J. *Numerical optimization*. New York, NY, USA: Springer, 1999.

8. LAMURE, Hervé; MICHELUCCI, Dominique. Solving geometric constraints by homotopy. In: *Proceedings of the third ACM symposium on Solid modeling and applications*. 1995, pp. 263–269. Available also from: `https://dl.acm.org/doi/pdf/10.1145/218013.218071`.

9. OWEN, John C. Algebraic solution for geometry from dimensional constraints. In: *Proceedings of the first ACM symposium on Solid modeling foundations and CAD/CAM applications*. 1991, pp. 397–407. Available also from: `https://dl.acm.org/doi/pdf/10.1145/112515.112573`.

10. BOUMA, William; FUDOS, Ioannis; HOFFMANN, Christoph; CAI, Jiazhen; PAIGE, Robert. Geometric constraint solver. *Computer-Aided Design*. 1995, vol. 27, no. 6, pp. 487–501. ISSN 0010-4485. Available from DOI: `https://doi.org/10.1016/0010-4485(94)00013-4`.

11. SERRANO, David. Automatic dimensioning in design for manufacturing. In: *Proceedings of the first ACM symposium on Solid modeling foundations and CAD/CAM applications*. 1991, pp. 379–386.

12. HOFFMAN, Christoph M; LOMONOSOV, Andrew; SITHARAM, Meera. Decomposition plans for geometric constraint systems, part I: Performance measures for CAD. *Journal of Symbolic Computation*. 2001, vol. 31, no. 4, pp. 367–408.

13. LATHAM, Richard S; MIDDLEDITCH, Alan E. Connectivity analysis: a tool for processing geometric constraints. *Computer-Aided Design*. 1996, vol. 28, no. 11, pp. 917–928.

14. HOFFMANN, Christoph M. Summary of basic 2D constraint solving. *International Journal of Product Lifecycle Management*. 2006, vol. 1, no. 2, pp. 143–149.

15. MOUSSAOUI, Adel. *Geometric Constraint Solver*. 2016. Available also from: `https://hal.science/tel-01402691`. PhD thesis. Ecole nationale Supérieure d'Informatique (ex I.N.I), Alger.

16. ALDEFELD, Bernd. Variation of geometries based on a geometric-reasoning method. *Computer-Aided Design*. 1988, vol. 20, no. 3, pp. 117–126.

17. BUCHANAN, S Alasdair; PENNINGTON, Alan de. Constraint definition system: a computer-algebra based approach to solving geometric-constraint problems. *Computer-Aided Design*. 1993, vol. 25, no. 12, pp. 741–750.

18. GRABOWSKI, Ralph. All About D-Cubed's 2D DCM. *UpFront.eZine* [online]. 2010, no. 645 [visited on 2023-09-28]. Available from: `https://web.archive.org/web/20120226151432/http://www.upfrontezine.com/2010/upf-645.htm`.

19. RIMMER, Jon. *Geometric Constraint Solving #1: introduction* [online]. 2022. [visited on 2023-10-18]. Available from: `https://blogs.sw.siemens.com/plm-components/geometric-constraint-solving-1-introduction/`.

20. AUTODROP3D. *Parametric CAD (beta)* [online]. 2023. [visited on 2023-12-04]. Available from: `https://www.autodrop3d.com/parametric-cad-beta.html`.

21. FREECAD COMMUNITY. *History* [online]. 2023. [visited on 2023-10-23]. Available from: `https://wiki.freecad.org/History`.

22. FREECAD COMMUNITY. *About FreeCAD* [online]. 2023. [visited on 2023-10-23]. Available from: `https://wiki.freecad.org/About_FreeCAD`.

23. ABDULLAH, Tahiri. *FreeCAD Sketcher Solver Architecture*. 2018. Available also from: `https://github.com/Salusoft89/planegcs/blob/main/doc/solver-abdullah.pdf`.

24. MDN CONTRIBUTORS. *JavaScript modules* [online]. 2023. [visited on 2023-11-03]. Available from: `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Modules`.

25. MDN CONTRIBUTORS. *Using Web Workers* [online]. 2023. [visited on 2023-11-04]. Available from: `https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers`.

26. EMSCRIPTEN CONTRIBUTORS. *Pthreads support* [online]. 2015. [visited on 2023-11-04]. Available from: `https://emscripten.org/docs/porting/pthreads.html`.

27. INGVAR, Stepanyan. *Using WebAssembly threads from C, C++ and Rust* [online]. 2021. [visited on 2023-11-03]. Available from: `https://web.dev/articles/webassembly-threads`.

28. SVELTE CONTRIBUTORS. *Svelte components* [online]. 2023. [visited on 2023-11-23]. Available from: `https://svelte.dev/docs/svelte-components`.

29. GAMMA, Erich; HELM, Richard; JOHNSON, Ralph; VLISSIDES, John M. *Design Patterns: Elements of Reusable Object-Oriented Software*. 1st ed. Addison-Wesley Professional, 1994. ISBN 0201633612.

30. SVELTE CONTRIBUTORS. *svelte/store* [online]. 2023. [visited on 2023-11-23]. Available from: `https://svelte.dev/docs/svelte-store`.

31. GALIL, Zvi; ITALIANO, Giuseppe F. Data Structures and Algorithms for Disjoint Set Union Problems. *ACM Comput. Surv.* 1991, vol. 23, no. 3, pp. 319–344. ISSN 0360-0300. Available from DOI: `10.1145/116873.116878`.

32. CHRISTOPH, Blaue. *A Sketcher Lecture*. 2021. Available also from: `https://github.com/Salusoft89/planegcs/blob/main/doc/sketcher-lecture.pdf`.

33. VAVŘÍKOVÁ, Eva. *Descriptive geometry*. Ostrava: VŠB – Technical University of Ostrava, 2005. ISBN 80-248-1006-9. Available also from: `http://mdg.vsb.cz/portal/en/DescriptiveGeometry.pdf`.

34. BROOKE, John. *"SUS-A quick and dirty usability scale." Usability evaluation in industry*. CRC Press, 1996. ISBN 9780748404605.

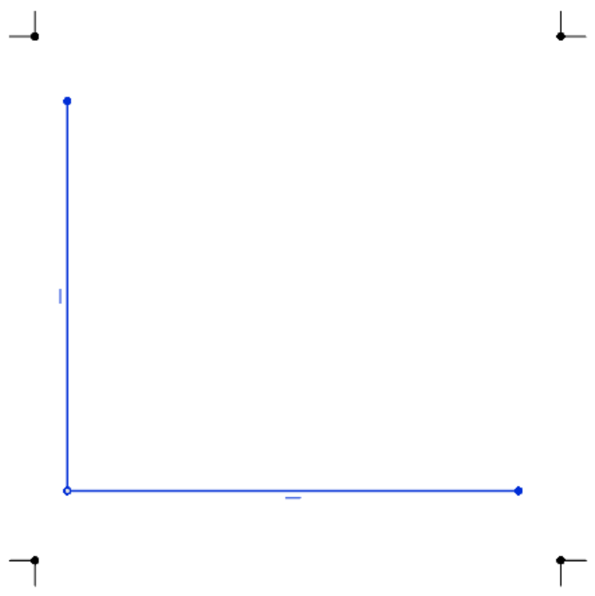
A Digital attachments

The thesis archive includes the following files:

- `user_study` – directory with documents in Czech, which are related to the user study:
  - `tasks.pdf` – tasks given to the participants, *also attached in this appendix below.*
  - `questionnaire.pdf` – questionnaire for the evaluation of the user study.
- `planegcs` – directory with the source code of the planegcs wrapper library.
  - `README.md` – usage manual and instruction for building.
- `designer` – directory with the source code of a limited version of Designer, which includes only the Sketch and Contour features (plus the basic functionality).
  - `README.md` – instructions for running the source code and specification of my contribution.
  - `src/lib/feature/sketch` – code of the Sketch feature.
  - `src/lib/feature/contour` – code of the Contour feature.

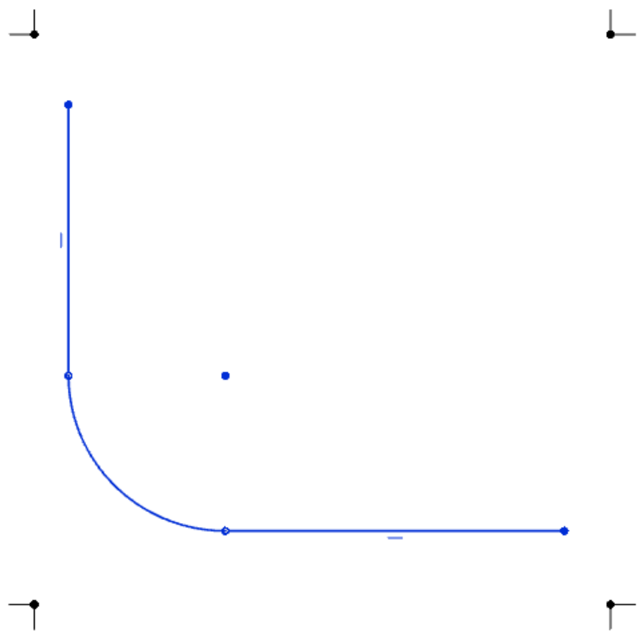
Úvodní úkoly

1. pomocí kreslícího nástroje pro čáru nakreslete dvě navazující úsečky, které nejsou omezeny na žádný referenční (černý) bod, na jednu nastavte vodorovné omezení —, na druhou svislost |



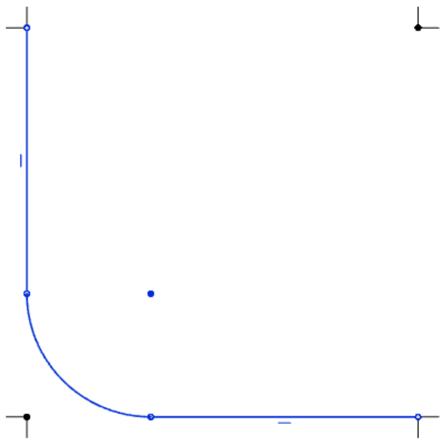
(4 volné parametry)

2. aplikujte na čáry nástroj Zaoblení (Fillet)



(5 volných parametrů)

3. Nyní uchyťte horní bod k levému hornímu referenčnímu bodu a podobně, pravý bod k pravému spodnímu referenčnímu bodu



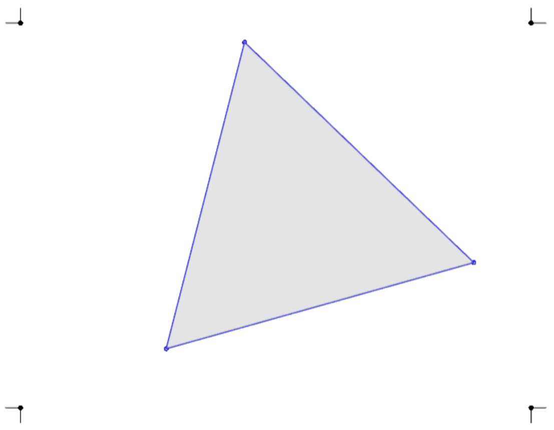
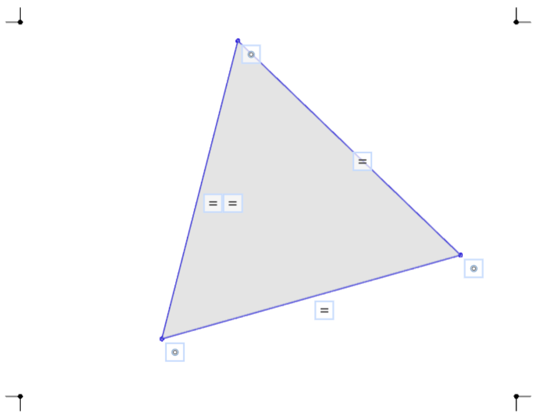
(1 volný parametr)

4. zkuste si zahýbat se slidery pro nastavení výšky a šířky (width, height) kontury, čáry by se měly samy roztahovat,

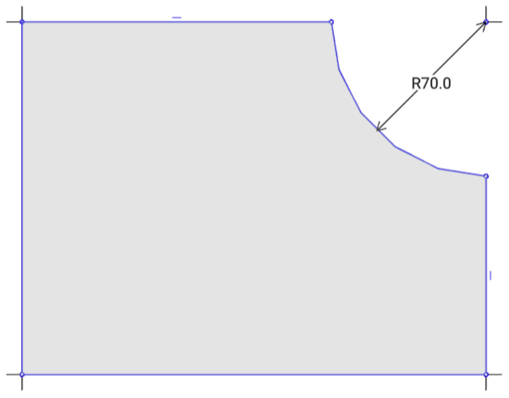
Následuje ukázka zbývajících omezení.

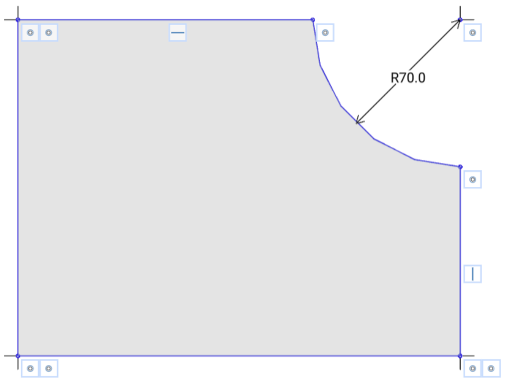
Navazující úkoly

1. Nakreslete rovnostranný trojúhelník **(4 volné parametry)**
    - po dokončení vytvořte novou konturu

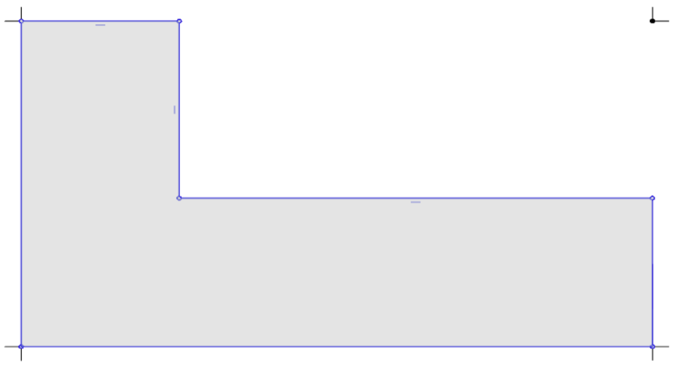



2. Nakreslete vykrojený obdélník, uchycený na referenční body kontury **(Plně omezená skeč)**
    - radius nastavte na nějakou fixní hodnotu
    - po dokončení vytvořte novou konturu

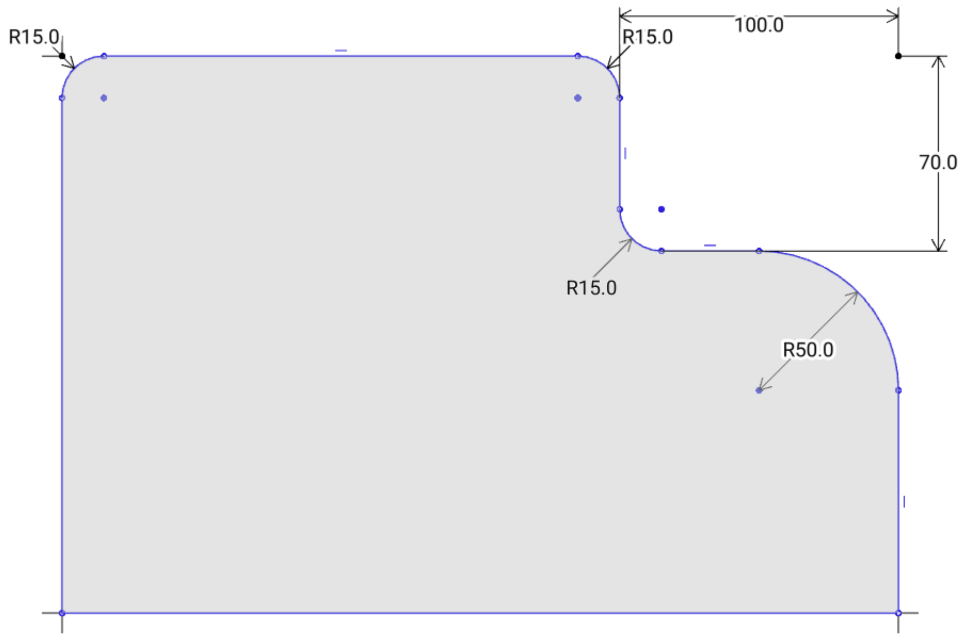




3. Nakreslete následující tvar **(3 volné parametry)**
    - nastavte rozměry kontury na `300x200` (šířka x výška)
    - tři rohy přichyťte na referenční body kontury, použijte vod. a svislé omezení
    - po dokončení **nemažte konturu**

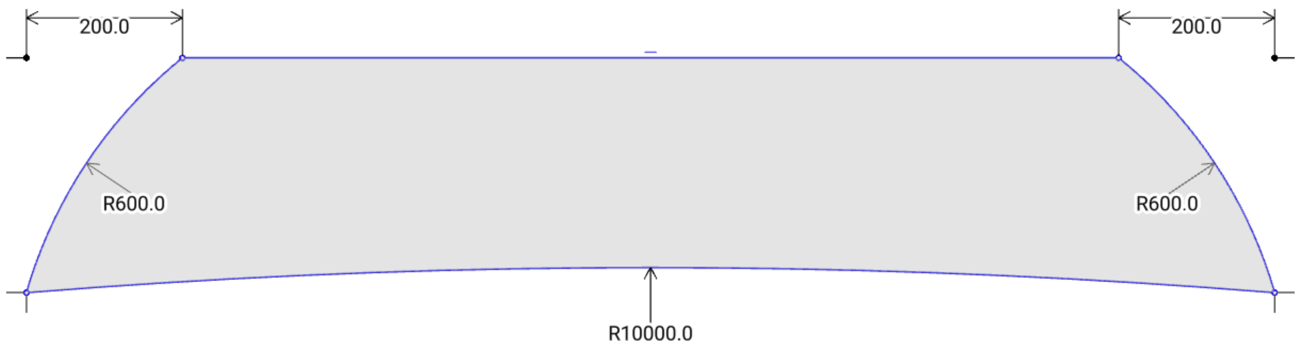


4. Převeďte existující tvar na následující pomocí relevantních nástrojů a přidejte rozměrová omezení podle obrázku **(Plně omezená skeč)**



- otestujte změnu rozměru skeče v nastavení kontury
- po dokoneční vytvořte novou konturu a nastavte rozměry na `1600x300`

5. Nakreslete následující tvar **(Plně omezená skeč)**
   - hodnota horizontálních vzdáleností (nahoře vpravo/vlevo) je dána výrazem: `width / 8`


200.0
200.0
R600.0
R600.0
R10000.0