TÉCNICO LISBOA

Geometric Constraints in Algorithmic Design

Rui Guilherme Cruz Ventura

Thesis to obtain the Master of Science Degree in

Computer Science and Engineering

Supervisor: Prof. Dr. António Menezes Leitão

July 2021

Acknowledgments

To anyone and everyone who supported and bore with me: thank you.

Abstract

Modern Computer-Aided Design (CAD) applications need to employ, to a lesser or greater extent, Geometric Constraints (GCs) that condition the geometric models being produced. However, these applications prove insufficient for the production of complex and sophisticated designs. In response to this limitation, a new paradigm named Algorithmic Design (AD) emerged. It comprehends the creation of designs through algorithmic specifications, enabling the automation of repetitive tasks. Alas, it is not yet as widespread as more traditional methods, partially due to the added time, effort, and expertise required to specify relations between objects. This can be mitigated through the incorporation of GC functionality in AD tools to help bridge the gap between AD and more traditional paradigms. The focus of this work is the creation, and implementation, of primitive GC functionality, supported by a mature geometric computation library, that facilitates the specification of geometric forms. We benchmark our solution's performance, as well as test it with four different constraint-ridden shapes inspired by existing designs, highlighting two different approaches: an analytic approach, naturally used in programming, and a constructive approach, the one our solution is based on. Additionally, we explore beneficial side effects of our implementation regarding the repurposing of more complex functionality with very little extra effort. We conclude our solution's approach proves more comprehensible and intuitive for practitioners.

Keywords

Parametric CAD; Geometric Constraints; Algorithmic Design; Exact Computation; Constructive Geometry.

Resumo

Aplicações modernas de projecto assisto por computador precisam de empregar, em menor ou maior grau, restrições geométricas que condicionam os modelos produzidos. Todavia, estas aplicações provam ser insuficientes para a produção de modelos complexos e sofisticados. Como resposta a esta limitação, surgiu um novo paradigma nomeado Algorithmic Design (AD). O paradigma consiste na criação de modelos provenientes de descrições algorítmicas, permitindo a automatização de tarefas repetitivas. Infelizmente, não é um paradigma tão difundido como métodos mais tradicionais, parcialmente devido ao tempo, esforço, e conhecimento adicional necessário para especificar relações entre objectos. Isto pode ser mitigado através da incorporação de funcionalidade de restrição geométrica em ferramentas de AD para ajudar preencher a lacuna entre AD e paradigmas mais tradicionais. O foco deste trabalho assenta sobre a criação, e implementação, de funcionalidade primitiva de restrição geométrica, suportada por uma biblioteca de computação geométrica exacta, que facilita a especificação de formas geométricas. Produzimos testes de referência de desempenho da nossa solução, também têmo-la testado com quatro formas geométricas dominadas por restrições inspiradas por projectos existentes, salientado duas abordagens diferentes: uma abordagem analítica, naturalmente usada em programação, e uma abordagem constructiva, em que se baseia a nossa solução. Adicionalmente, exploramos efeitos secundários benéficos da nossa implementação relativamente ao reaproveitamento de funcionalidade mais complexa com muito pequeno esforço extra. Concluímos que a abordagem empregue pela nossa solução prova ser mais compreensível e intuitiva para praticantes.

Palavras Chave

CAD Paramétrico; Restrições Geométricas; Algorithmic Design; Computação Exacta; Geometria Construtiva.

Contents

List of Figures

List of Tables

List of Equations

List of Listings

Acronyms and Abbreviations

AD	Algorithmic Design
AMD	Advanced Micro Devices
API	Application Programming Interface
B-Rep	Boundary Representation
BIM	Building Information Modeling
CAD	Computer-Aided Design
CAS	Computer Algebra System
CGAL	the Computational Geometry Algorithms Library
CPU	Central Processing Unit
CS	Computer Science
CSG	Constructive Solid Geometry
CSP	Constraint Satisfaction Problem
DDR	Double Data Rate
EPS	Encapsulated PostScript
FFI	Foreign Function Interface
GC	Geometric Constraint
GCS	Geometric Constraint Solving
GFL	Geometry Functions Library
GUI	Graphical User Interface
IDE	Integrated Development Environment
LEDA	the Library for Efficient Data types and Algorithms
PGF	Portable Graphics Format

RAM	Random Access Memory
RGC	Rhythmic Gymnastics Center
SDK	Software Development Kit
SO-DIMM	Small Outline Dual In-line Memory Module
TikZ	TikZ ist *kein* Zeichenprogramm
TPL	Textual Programming Language
VBA	Visual Basic for Applications
VPL	Visual Programming Language

1

Introduction

Contents

Modern Computer-Aided Design (CAD) tools include substantial support for parametric operations and Geometric Constraint Solving (GCS). These mechanisms have been developed over the past few decades [1] and are heavily and ubiquitously used across the Architecture, Engineering, and Construction industry.

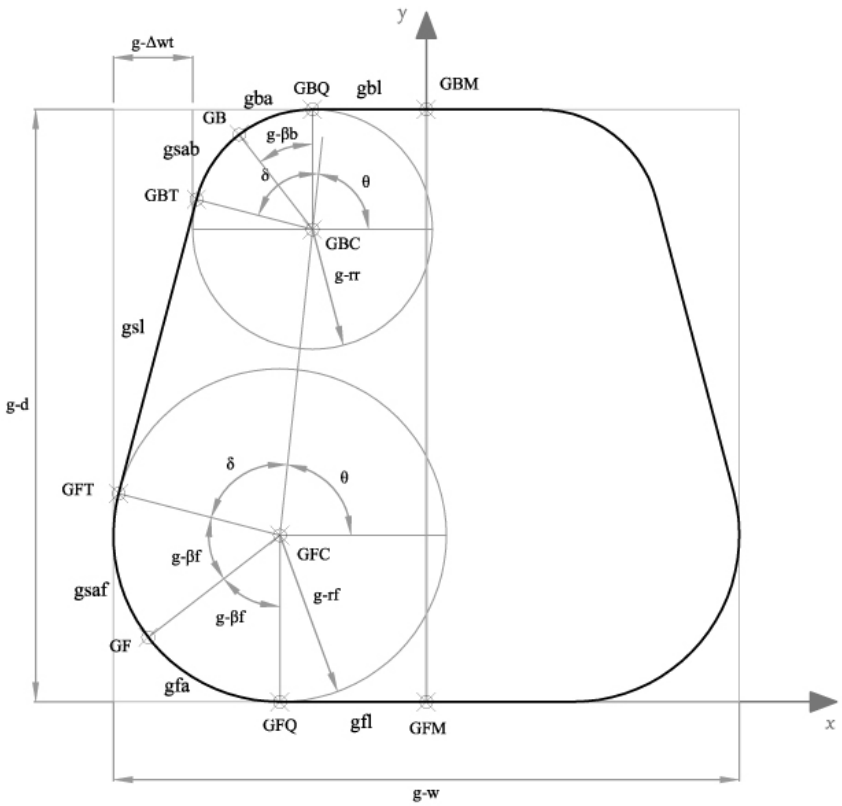
Parametric modeling is used to design with constraints, whereby users express a set of parameters and interdependent operations, establishing restrictions between geometric entities. The resulting geometry can be controlled from input parameters using two computational mechanisms: (1) parametric operations, which build geometry that implicitly abides by constraints imposed when the user selects the operation and its inputs, and (2) GCS, which finds the positions of geometric entities that satisfy a set of constraints explicitly imposed by the user.

Nowadays, parametric operations in CAD software are mostly accessible through intuitive, robust, and easy-to-use direct-manipulation interfaces, offering a wide variety of different operations. These operations are created when a designer uses solid modeling operations, such as face extrusions or shape unions; and recorded into a user-controlled sequential history of construction steps that can be replayed in the event of changes, updating the modeled geometry. Alas, dependency propagation direction is fixed, forcing users to plan their model's features beforehand. In constraint solving, by contrast, dependency propagation direction isn't fixed. Instead, users introduce a set of parameters and geometric entities followed by specifying the constraints that relate these objects. Naturally, GCS fits in CAD software, having been the target of considerable research and development to implement efficient approaches and methodologies capable of solving Geometric Constraint (GC) problems. So much so that it has become standard in major CAD software, such as AutoCAD[1] from Autodesk®, which supports the ability to constrain objects in a variety of ways, e.g., point coincidence, line perpendicularity, and tangencies, among other kinds of constraints.

However, traditional interactive methods for parametric modeling suffer from the disadvantage that they do not scale properly when designing more complex ideas. In recent years, a novel approach to design named Algorithmic Design (AD) has emerged, allowing the specification of sketches and models through algorithms [2], leading to the creation and integration of AD tools into CAD software as well. Some use Visual Programming Languages (VPLs), others Textual Programming Languages (TPLs), or even a mixture of both. The latter overcomes a fundamental issue with VPLs which is the frequently disproportionate complexity between the program and the respective resulting model.

Dealing with GCs, regardless of the approach, can still prove to be an arduous task. Take as an example the sketch of a chair seat's outer frame, as seen in fig. 1.1, from a multi-purpose chair generation tool [3] where the chair's overall shape is controllable by specifying the values for a set of input parameters.

[1] https://autodesk.com/autocad



Source: Project source code, publicly unavailable (Jan 2019)

Figure 1.1: Sketch of a chair seat's outer frame, defined by 5 input parameters: (1) Width `g-w`, (2) depth `g-d`, (3) taper width `g-Δwt`, (4) front radius `g-rf`, and (5) rear radius `g-rr`.

The seat's corners are defined by circles whose respective front and rear radius' length, `g-rf`, `g-rr`, is obtained by computing distances, from which the circles' centers, `GFC` and `GBC`, can be obtained. The circles are then connected through outer tangent lines, `gsl`, forming the outer frame of the chair's seat. Some of these operations, such as the radius computation, *tangency*, and *circumcenter*, depend on operations that query if a point is at a certain distance from an object, or if two points are coincident. Such operations must be handled carefully due to numerical robustness issues that may arise when performing fixed-precision arithmetic. As such, on top of the design process itself, the user must identify the GCs, resorting to trigonometry analysis, perform tolerance-based comparisons to determine point distance or if two points are coincident, among other techniques the user most likely is not aware he must rely upon to circumvent these issues, particularly, when we take into consideration that most AD practitioners are architects and designers without an extensive background in Computer Science (CS).

To overcome the limitations exposed above, this report proposes the implementation of GC primitives with specialized efficient solutions for different combinations of input objects. We additionally focus our work around TPLs, further making them more attractive, and easier to both adopt and use.

1.1 Document Structure

The present document is structured in 5 different chapters, namely:

Introduction Broken into several sections, including this one, presents: (1) a brief historical overview of the development of parametric operations in CAD software in section 1.2, (2) the main approaches to GCS in CAD, in section 1.3, (3) two simple algebraically formulated examples of GC problems and respective solutions along with code examples, in section 1.4, and (4) a section dedicated to further elaborating on AD and the benefits and drawbacks it introduces to the design process, in section 1.5.

Related Work An exposition of the related work in the form of (1) a comprehensive discussion about numerical robustness in computational processes, showcasing a set of software tools capable of handling these issues in the context of geometric computation, in section 2.1, (2) an overview of GC tools, presenting some of their benefits and drawbacks, in section 2.2, and (3) an overview of algorithmic design tools, similarly comparing them and addressing positive and negative points, in section 2.3.

Solution A solution proposal, followed by how it was implemented and how it is capable of efficiently handling the specification of GC problems, in chapter 3, going over its core components: (1) an *Exact Geometric Computation Library*, discussed in section 3.1.1, which provides the basic geometric entities and functions to work it, (2) an interoperability *Wrapper Code* layer that allows the target platform to repurpose the previous component's foreign constructs, detailed in section 3.1.2, and (3) *Geometric Constraint Primitives*, showcased in section 3.1.3 capable of solving a class of GC problems employing a Euclidean constructive approach to producing geometry.

Evaluation The methodology used to evaluate the proposed solution in chapter 4, involving (1) a performance comparison in section 4.1 with a similar project with a shared goal, (2) an extensive evaluation comparing two different approaches at solving GC problems present in four case studies inspired by existing designs, and (3) a performance, correctness, and effort estimation analysis, in section 4.3, comparing an implementation of a complex geometric algorithm in the target platform vs. a repurposed mature implementation.

Conclusion Concluding remarks that summarize our work, in chapter 5, accompanied by candidate future work.

1.2 Parametric Operations in CAD

Ivan Sutherland introduced the world to Sketchpad [4] in 1963, an interactive 2D CAD program. Despite never using the word *parametric* in writing, Sutherland's Sketchpad was capable of establishing atomic constraints between objects which had all the essential properties of parametric equations, being the first

of its kind and the prime ancestor of modern CAD programs. The earliest 3D system [5] dates from the 1970s. It used a Constructive Solid Geometry (CSG) [6, 7] binary tree, and Boundary Representation (B-Rep) [8] for representing solid objects. This system's parametric nature rested in the CSG tree, which acted as a rudimentary construction step history. The user could make modifications to the controlling parameters' values of a certain operation in the tree, reapply the modified history, and generate the newly updated model. Surfacing nearly a decade later, Pro/ENGINEER[2] [9] was the first system to be acknowledged as a parametric system. It enabled the establishment of relations between the objects' sizes and positions such that a change in a dimension between objects would automatically change affected objects accordingly. Unlike Sketchpad, it supported 3D geometry and changes would propagate over different drawings made by different users. This amid a sudden flux of activity and interest, GCS soon became standard in drawings by the early 1990s [10–12]. Efforts to expand the benefits of constraint solving beyond simple sketches were made, some systems having implemented constraint solving in 3D. Improvements from then on focused mostly on robustness and operation variety.

In recent decades, emphasis shifted to making parametric CAD software more interactive and user-friendly. The intent was to make it as simple as dragging a face of an object to where it should be instead of scrolling through a construction history in attempts to locate a specific operation, and hopefully changing the correct controlling parameter's value within that operation. This in itself is a tedious and error-prone process that can lead to undesired side effects instead of producing the intended changes. A variety of systems have been developed to mitigate this rigidity [13–15], but not without drawbacks, since direct-manipulation operations were just added to the construction history as transformation operations, oblivious to parent operations the new ones might depend on. Further limitations are discussed in [16], along with a proposal for future design software exempt of parametric operations. Nonetheless, parametric operations will still see continued usage for the foreseeable future.

1.3 Constraints in CAD

We have briefly seen how parametric operations in CAD software have evolved. These operations allow the user to create geometric objects that satisfy certain constraints *implicitly* imposed on the objects when the user selects the operations they want to use along with the respective operation's inputs. Naturally, GCS fits well in CAD applications. GCs allow the repositioning and scaling of geometric objects so that they satisfy constraints *explicitly* imposed on them by the user.

Constraint Satisfaction Problems (CSPs) are a well-known subject of research both in mathematics and in the CS field. GCS is a subclass of CSPs. More specifically, it is a CSP in a computational geometry setting. The abstract problem of GCS is often described as follows [1, pp. 6]:

[2] https://www.ptc.com/en/products/creo/pro-engineer

> *Given a set of geometric objects, such as points, lines, and circles; a set of geometric and dimensional constraints, such as distance, tangency, and perpendicularity; and an ambient space, usually the Euclidean plane; assign coordinates to the geometric objects such that the constraints are satisfied, or report that no such assignment has been found.*

One of the important features of a solver is its *competence*, which is related to the capability of reporting unsolvability: if in fact no solution for the problem at hand exists and the solver is capable of reporting unsolvability in that case, the solver is deemed fully competent. Since constraint solving is mostly an exponentially complex problem [17], partial competence suffices as long as decent solutions can be found in affordable time and space.

There are multiple approaches to constraint solving, but the most relevant ones are graph-based, logic-based, algebraic, and theorem prover-based, of which the first is the predominant one. It is important for these approaches that the GC system does not have too few or too many constraints. Summarily, a system can either be (1) under-constrained if the number of solutions is unbound due to lack of constraint coverage over the entities involved, (2) over-constrained if there are no solutions because of constraint contradictions, or (3) well-constrained if the number of solutions is bound to a finite positive number.

Some of the subjects approached here are briefed in [18]. The following sections present and briefly discuss the aforementioned approaches to constraint solving.

1.3.1 Graph-Based Approaches

In graph-based approaches, the problem is translated into a labeled *constraint graph*, where vertices are constrained geometric objects, and edges the constraints themselves. This approach is split into three main branches:

Constructive Approaches The graph is decomposed and recombined to extract basic construction steps that must be solved. In a subsequent phase, this is elaborated upon by employing algebraic and/or numerical methods. This has become the dominant approach to GCS, also becoming the target of considerable research and development.

Degrees of Freedom Analysis The graph's vertices are labeled with the degrees of freedom of the represented object. Each edge is labeled by the degrees of freedom the constraint cancels out. This graph is then analyzed for a solution strategy.

A symbolic solution method is derived using rules with geometric meaning, a method proved to be correct in [19]. It is further extended by using it along with numerical methods as a fallback if geometric reasoning fails [20].

Latham and Middleditch [21] decompose the graph into minimal connected components they call *balanced sets* that are solved by a geometric construction, falling back to a numerical solution attempt. This method can deal with symbolic constraints and identifies under- and overconstrained problems, where the latter kind is approached by prioritizing the given constraints.

Propagation Approaches The graph's vertices represent variables and equations, and the edges are labeled with occurrences of the variables in equations. The goal is to orient the graph such that all incident edges to an equation vertex but one are incoming edges. If so, the equation system has been triangularized. Orientation algorithms include degree-of-freedom propagation and propagation of known values [22, 23] which can fail in the presence of orientation loops, but such situations are addressed [23] and they may resort to numerical solvers.

1.3.2 Logic-Based Approaches

Using logic-based approaches, the constraint problem is translated into a set of geometric assertions and axioms which is then transformed in such a way that specific solution steps are made explicit by applying geometric reasoning. The solver then takes a set of construction steps and assigns coordinate values to the geometric entities.

A geometric locus[3] at which constrained elements must be is obtained using first order logic to derive geometric information, applying a set of axioms from Hilbert's geometry [24–26]. Two different types of constraints are further considered [27, 28]: (1) sets of points placed with respect to a local coordinate frame, and (2) sets of straight line segments whose directions are fixed. The reasoning is performed by applying a rewriting system on the sets of constraints. Once every geometric element is in a unique set, the problem is solved.

1.3.3 Algebraic Approaches

In the case of an algebraic approach, the problem is translated into a system of equations where the variables are coordinates of geometric elements and the equations, which are generally nonlinear, express the constraints upon them. This approach's main advantage is its completeness and dimension independence. However, it is difficult to decompose the equation system into subproblems, and a general, complete solution of algebraic equations is inefficient. Nonetheless, small algebraic systems tend to appear in the other approaches and are routinely solved.

[3] In mathematics, a locus is a set of points that satisfy some condition. In layman's terms, a location or place.

1.3.4 Symbolic Methods

Symbolic methods rely on general equation solvers which employ techniques to triangularize equation systems [29, 30] that emerge from employing an algebraic approach. A solver built on top of the Buchberger's algorithm is described in [31]; Kondo [32] further reports on a symbolic algebraic method.

These methods can produce generic solutions which can be evaluated for a different set of constraint assignments, then producing parameterized solutions. However, solvers are very slow and computation demands a lot of space, usually requiring exponential running time [33].

1.3.5 Numerical Methods

Among the oldest approaches to constraint solving, numerical methods solve large systems of equations iteratively. Methods like Newton iteration work properly if a good approximation of the intended solution can be supplied and the system is not ill-conditioned. Take, for example, a sketch of a model the user drew. If the starting point comes from said sketch, then it should follow that the result be close to what is intended. Alas, such methods may find only one solution, even in cases where there are many, and may not allow the user to select the one they are interested in. Such methods are called local methods, as opposed to global methods, exploring the problem space for every possible solution.

Relaxation methods [4, 34, 35] can be employed in attempts to partly minimize global error by perturbing the values assigned to the variables. However, in general, convergence to a solution is slow.

The Newton-Raphson iteration method, the most widely used one, is a local method and converges much faster than relaxation, but does not apply to over-constrained systems of equations unless expanded upon [36].

Global and guaranteed convergence can be had resorting to the *Homotopy continuation* family of methods [37]. Despite usage in GCS [33, 38], these are far less efficient than the Newton-Raphson method due to the latter's exhaustive nature.

1.3.6 Theorem Proving

GCS can be seen as a subproblem of geometric theorem proving, but the latter requires general techniques, therefore requiring much more complex methods than those required by the former.

Wen-Tsün Wu's method [39, 40] is an algebraic-based method that can be used to automatically find necessary conditions to obtain non-degenerated solutions. It can be used to prove novel geometric theorems [29]. Chou et al. [41, 42] develop on automatic geometric theorem proving, allowing the interpretation of the computed proof.

1.3.7 Other Areas

The following are briefly described key advances made during the past two decades that interface with other areas or that cannot be readily integrated into graph-constructive solvers. These techniques also constitute examples of further attempts to broaden the scope of GCS, proving that it is a strong field of research with many applications beyond CAD.

Deformations When restrictions are placed on the type of deformation, these problems can be seen as constraint solving. For example, Ahn et al. [43], Bao et al. [44], Moll and Kavraki [45] consider deformations that minimize strain energy; Xu et al. [46] entail surface deformation under area constraints. However, such techniques are rarely integrated with other GCs such as point distance or perpendicularity.

Dynamic Geometry The addition of constraints to a given under-constrained system can make it well-constrained, and such constraints can be seen as parameters when they are dimensional. Varying their values, different solutions arise, which can be wholly understood as a dynamic geometric configuration. Systems akin to Cinderella [47] can deal with these problems. Further literature exists on these problems from a constraint solving perspective [48].

Evolutionary Methods Consist of re-interpreting the problem as an optimization problem, attacking it using genetic, particle-swarm or other evolutionary methods [49, 50].

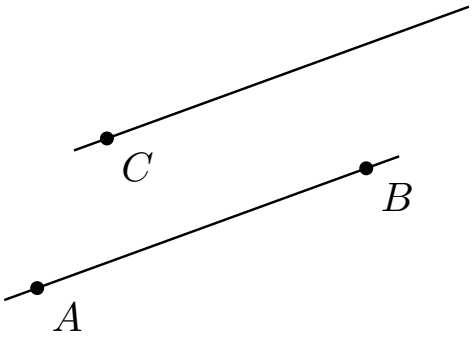
1.4 Geometric Constraint Problem Examples

This section presents two simple examples of geometric models that are defined through the specification of GCs, and the respective solutions using intuitive algebraic formulation, accompanied by programmatic solutions using TikZ [51] and Eukleides [52]. Depictions of the aforementioned models can be seen in fig. 1.2. The examples are limited to the two-dimensional Euclidean plane over real numbers, $\mathbb{R}^2$. Solutions for analogous problems in three-dimensional Euclidean space, $\mathbb{R}^3$, exist as well.

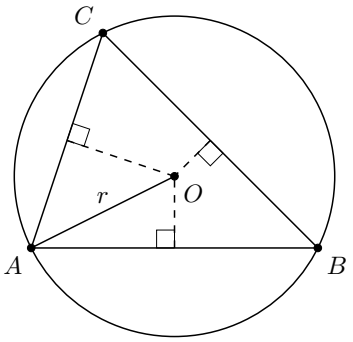
The first problem is that of a parallelism constraint: specifying a line that goes through a given point while also being strictly parallel to another already defined line. The second problem is a circumscription constraint: defining a circle that tightly wraps around a triangle, i.e., the circle's circumference goes through three given non-collinear points.

1.4.1 Parallel lines

Let A, B, $C \in \mathbb{R}^2$ such that C is a point in the line which is strictly parallel to the line $\overleftrightarrow{AB}$ (see fig. 1.2a).



(a) Line that goes through C, strictly parallel to $\overleftrightarrow{AB}$.



(b) $\odot O_r$ circumscribed about $\triangle ABC$.

Figure 1.2: Geometric models defined using GC relations: (a) showcases line parallelism, and (b) showcases a circle circumscription about a triangle.

A line in $\mathbb{R}^2$ can be described by the parametric equation

$$P_Q = Q + \lambda\vec{u} \Rightarrow \begin{cases} x = x_Q + \lambda u_x \\ y = y_Q + \lambda u_y \end{cases}, \lambda \in \mathbb{R} \tag{1.1}$$

where $Q = (x_Q, y_Q)$ is a point on the line that goes through $P_Q = (x, y)$, and $\vec{u} = (u_x, u_y)$ is the vector that drives the line. To then describe the line that goes through C and is parallel to $\overleftrightarrow{AB}$, one must compute the base point Q, trivially C, and the directional vector $\vec{u}$, which can be obtained from $\overleftrightarrow{AB}$. Let $Q = C$, and $\vec{u} = B - A$, such that

$$P_C = C + \lambda\vec{u},\ \lambda \in \mathbb{R}.$$

Listing 1.1 shows the code used to produce the example shown in fig. 1.2a using TikZ with the `tkz-euclide`[4] LaTeX package, using `tkzDefLine`, which takes two points A, B, with the `parallel` transformation option. This option takes the point C the resulting line goes through. The result is a point $D = C + \vec{u}$, which can be obtained using `tkzGetPoint` to later draw the line.

```
1 \begin{tikzpicture}[rotate=20]
2   \tkzDefPoints{0/0/A,3/0/B,1/1/C}
3   \tkzDefLine[parallel=through C](A,B) \tkzGetPoint{D}
4   \tkzDrawLines[add=.1 and .1](A,B C,D)
5   \tkzDrawPoints(A,B,C)
6   \tkzLabelPoints(A,B,C)
7 \end{tikzpicture}
```

Listing 1.1: Parallel lines example from fig. 1.2a using `tkz-euclide`. The highlighted line shows how to define the line L_C parallel to $\overleftrightarrow{AB}$.

Listing 1.2 shows the code used to produce an identical figure using Eukleides. In Eukleides, the parallel line L_C can be obtained through the `parallel` function, which takes the line $\overleftrightarrow{AB}$ it is parallel to

[4] https://ctan.org/pkg/tkz-euclide

and the point C it goes through.

```
A B C triangle 3, pi/4 rad, pi/6 rad, 20 deg
AB = line(A, B)
lC = parallel(AB, C)
draw
  AB; lC
  A; B; C
end
label
  A -pi/4 rad
  B -pi/4 rad
  C -pi/4 rad
end
```

Listing 1.2: Parallel lines example from fig. 1.2a using Eukleides. The highlighted line shows how to define the line L_C parallel to $\overleftrightarrow{AB}$.

1.4.2 Circumcenter

Let A, B, C, $O \in \mathbb{R}^2$ be points such that O is the center point of a circle of radius r, $\odot O_r$, that is circumscribed about the triangle $\triangle ABC$ (see fig. 1.2b).

A precondition for this computation is that $\triangle ABC$ is not degenerate, i.e., its vertices are non-collinear. That can be verified by computing the cross product of any two distinct vectors that drive $\triangle ABC$'s edges and verifying it does not equate to zero.

To draw $\odot O_r$, we must compute both its center and radius. Its radius r can be trivially defined as the distance of the center O to any of the $\triangle ABC$'s vertices, i.e., $r = \overline{OA} = \overline{OB} = \overline{OC}$. To determine O, one must compute the intersection of the perpendicular bisectors of the triangle's edges. Said bisectors are the mediators between an edge's vertices, which can be described by eq. (1.1), where P is the midpoint between the vertices, and $\vec{u}$ is a vector normal to the edge. The midpoint $M_{P_1P_2}$ of two points $P_1, P_2 \in \mathbb{R}$ is given by

$$M_{P_1P_2} = \frac{P_1 + P_2}{2} = (\frac{x_1 + x_2}{2}, \frac{y_1 + y_2}{2}). \tag{1.2}$$

Further, the scalar product of two vectors $\vec{u}, \vec{v} \in \mathbb{R}^2$ is given by

$$\vec{u} \cdot \vec{v} = (u_x, u_y) \cdot (v_x, v_y) = u_x v_x + u_y v_y. \tag{1.3}$$

The normal vector $\vec{n}$ is such that, for some vector $\vec{u}$,

$$\vec{u} \cdot \vec{n} = 0.$$

A vector $\vec{n} \in \mathbb{R}^2$ normal to another vector $\vec{u}$ can be easily obtained by swapping the components of $\vec{u}$

while negating one of them, a property easily verified by applying eq. (1.3).

Computing the edges' midpoints and respective normal vectors, we can then describe the mediators. Let M_{AB}, M_{AC}, $M_{BC} \in \mathbb{R}^2$ be the midpoints, by eq. (1.2), of the respective edges, and $\vec{u_1}$, $\vec{u_2}$, $\vec{u_3}$ the edges' normal vectors, such that

$$\begin{aligned} P_{M_{AB}} &= M_{AB} + \lambda_1 \vec{u_1} \\ P_{M_{AC}} &= M_{AC} + \lambda_2 \vec{u_2},\ \lambda_i \in \mathbb{R}. \\ P_{M_{BC}} &= M_{BC} + \lambda_3 \vec{u_3} \end{aligned}$$

This problem can be further simplified by eliminating one of the redundant bisectors. Since the intersection of two lines already yields a single point, we can eliminate one of the equations. Say we discard the mediator of line $\overleftrightarrow{BC}$. We then require that

$$P_{M_{AB}} = P_{M_{AC}} \overset{(1.1)}{\Rightarrow} \begin{cases} x_{M_{AB}} + \lambda_1 u_{1x} = x_{M_{AC}} + \lambda_2 u_{2x} \\ y_{M_{AB}} + \lambda_1 u_{1y} = y_{M_{AC}} + \lambda_2 u_{2y} \end{cases} .$$

Every variable is known except for λ_1 and λ_2, but the equation system can be solved in order to assign values to both of them since we have exactly two equations that relate them. Finally, we can define O using one of the equations with the respectively found λ, i.e., using $L_{M_{AB}}$, for instance, we have

$$O = M_{AB} + \lambda_1 \vec{u}.$$

Listing 1.3 shows the code used to produce the example in Figure 1.2b using TikZ with the `tkz-euclide` LaTeX package. To compute the center point of $\odot O_r$, one can use `tkzCircumCenter`, which takes three points A, B, C, and generates the result O, obtainable using `tkzGetPoint`.

Listing 1.4 shows the code that produces an identical figure using Eukleides. In Eukleides, one can use the `circle` function, which similarly takes three points A, B, C, and generates the circle $\odot O_r$ circumscribed about $\triangle ABC$, while O can be obtained using the `center` function.

Both languages used to produce the examples' solutions provide a sensible set of constraint primitives. However, in the particular case of `tkz-euclide`, the syntax required for describing the models is outdated, rigid, and may cause confusion. For example, in listings 1.1 and 1.3, command results can not be used directly as inputs to other commands and must instead be obtained using another command to create a permanent symbol associated with the resulting value. By contrast, functions and expressions' results in modern languages can be used directly as well as stored by using a far friendlier assignment syntax. Nonetheless, the underlying ideas can be repurposed and adapted, implementing them in a modern and more expressive language.

```
\begin{tikzpicture}
  \tkzDefPoints{0/0/A,4/0/B,1/3/C}
  \tkzCircumCenter(A,B,C) \tkzGetPoint{O}
  \tkzDefMidPoint(A,B) \tkzGetPoint{AB}
  \tkzDefMidPoint(A,C) \tkzGetPoint{AC}
  \tkzDefMidPoint(B,C) \tkzGetPoint{BC}
  \tkzDrawSegments[style=dashed](AB,O AC,O BC,O)
  \tkzMarkRightAngles(A,AB,O B,BC,O C,AC,O)
  \tkzDrawPolygon(A,B,C)
  \tkzDrawCircle(O,A)
  \tkzDrawSegment(O,A)
  \tkzDrawPoints(A,B,C,O)
  \tkzLabelLine[above](O,A){$r$}
  \tkzLabelPoints[below left](A)
  \tkzLabelPoints[below right](B)
  \tkzLabelPoints[above left](C)
  \tkzLabelPoints(O)
\end{tikzpicture}
```

Listing 1.3: Circumcenter example from fig. 1.2b using TikZ alongside `tkz-euclide`. The highlighted line shows how to obtain the center of $\odot O_r$ via the non-degenerate triangle $\triangle ABC$.

1.5 Algorithmic Design

In spite of the improved usability and pervasiveness of parametric features in modern CAD applications, along with the immense strides made in the area of GCS, approaches reliant on these tools tend to not scale well with design complexity. Correctly applying modifications to existing models becomes cumbersome when experimenting with generating different variants of a model or adapting it to new requirements. Users have to spend most of their time and effort unnecessarily tweaking and changing their design's parameters' values, which can be, as mentioned, an error-prone process, hindering their capability to efficiently produce novel designs.

AD consists in the generation of CAD and Building Information Modeling (BIM) models through the specification of algorithmic descriptions [2], opposed to more classical approaches in which users directly interact with the geometric model being produced. Furthermore, the algorithms used to describe the idealized models are naturally parametric, which allows for the generation of multiple variants of said model by adjusting the algorithm parameters' values, enabling users to make changes to their models in a much more effortless and efficient manner when compared to direct-manipulation methods [53]. The parametric nature of the algorithmic specifications implicitly imposes constraints on the model since dependencies within the description are changed if an ancestor parameter's value changes upon re-execution, propagating the updates in a downwards fashion. This is advantageous since users can easily create more complex designs, hence also deeming AD a more scalable alternative to traditional approaches.

```
A.B.C = point(0, 0).point(1, 3).point(4, 0)
Or = circle(A, B, C)
O = center(Or)
AB.AC.BC = midpoint(A.B).midpoint(A.C).midpoint(B.C)
draw
  AB.O dashed
  AC.O dashed
  BC.O dashed
  (A.B.C); Or; O.A
  A; B; C; O
end
label
  A -3*pi/4 rad
  B  3*pi/4 rad
  C   -pi/4 rad
  O   -pi/4 rad
  A, AB, O right
  B, BC, O right
  C, AC, O right
end
```

Listing 1.4: Circumcenter example from Figure 1.2b using Eukleides. The highlighted line shows how to obtain the center of $\odot O_r$ via the non-degenerate triangle $\triangle ABC$.

Such an approach also lead to the creation and integration of programming tools into existing CAD and BIM software such as Grasshopper[5] for Rhinoceros[6] or Dynamo[7] for Revit[8]. Some tools, like Rosetta [54], offer a distinctly portable solution in contrast to the likes of the aforementioned ones, enabling the generation of several identical models for a variety of different CAD and BIM applications through a single specification [55] while also giving users room to experiment with a series of different available programming languages.

Despite the benefits that come with the integration of AD tools in CAD and BIM software, it is key that these tools also provide a highly expressive platform to further boost user productivity. This means these tools should provide a variety of primitive constructs, abstraction mechanisms, high-level concepts, among other capabilities, making it easier for users to create sophisticated models and designs [56]. Generally, the more expressive the platform is, the better it is with respect to usage, also making it easier to learn, a crucial point when migrating from traditional direct-manipulation user interfaces. This quality becomes all the more important when generating a geometric model riddled with constraints users have to manually specify and figure out, potentially introducing calculation or logical errors during the process. Thus, the inclusion of GC concepts in such tools would make working with constraints easier, in turn mitigating error propagation throughout the algorithm, and increase the tool's expressive power.

[5] https://www.grasshopper3d.com
[6] https://www.rhino3d.com
[7] https://dynamobim.org
[8] https://autodesk.com/revit

2

Related Work

Contents

In this chapter, we expose and discuss numerical accuracy issues that arise when performing computations with fixed-precision arithmetic. We then proceed to naming some precautions and steps in order to obtain practical solutions, followed by a brief mention of some software libraries dedicated to overcoming these issues through a series of exact algorithms and data structures.

We follow that by a comparative analysis of a set of GCS-capable programming tools along different dimensions, such as supported language paradigm, native GCS capabilities, 2D and 3D support. Of those tools, Eukleides [52], GeoSolver [57], and TikZ & PGF [51] are extensively discussed.

Similarly, we analyze AD tools. Some of them are integrated within CAD applications while others are standalone applications. These tools and their capabilities are summarized in table 2.2. Furthermore, Dynamo and Grasshopper are expanded upon.

The chapter closes with small remarks on VPLs' poorer scalability with increasing project complexity when compared to TPLs, showcasing the Rhythmic Gymnastics Center (RGC) as an example.

2.1 Robustness

The correctness proofs of nearly all geometric algorithms presented in theoretical papers assumes exact computation with real numbers [58]. However, floating-point numbers are represented with fixed precision in computers, making them inexact, which leads to inaccurate representations of the conceptual real number counterparts. For example, the rational number one-tenth ($\frac{1}{10}$) cannot be accurately represented as a floating-point number, nor is it guaranteed to be truly equal to another seemingly identical number. Such comparisons must be performed relying on tolerances, i.e., if a and b are two floating-point numbers, they are considered *the same* if $|a - b| \leq \epsilon$ for a given tolerance ϵ.

As an example, consider the problem of finding the closest of two points to the origin. The distance between two points $P, Q \in \mathbb{R}^2$ can be expressed by

$$d(P, Q) = \sqrt{(x_Q - x_P)^2 + (y_Q - y_P)^2}. \tag{2.1}$$

Let $A, B \in \mathbb{R}^2$ be two arbitrary points, and $O \in \mathbb{R}^2$ the origin. To determine which point, A or B, is closest to the origin O, we compare the former's distances to the latter's. That is, if

$$d(A, O) < d(B, O)$$

holds, A is the closest to the origin. Otherwise, they are either equidistant or B is closer. However, applying the square root operation in the distance computation is a step that will introduce errors. Given that we are only interested in comparing distances, and not use their actual value, we can, instead, compare the squared distances. As such, we avoid the square root, thus improving robustness, and

speeding up the process because the square root is a computationally heavy operation. Mei et al. [59] further discuss the issues with numerical robustness in geometric computation, namely how they arise, and propose practical solutions.

When used without care, fixed-precision arithmetic almost always leads to unwanted results due to marginal error accumulation caused by rounding (*roundoff*), propagated throughout a series of calculations. As seen above, careful observations must be made before proceeding with computations as simple as distance calculation. To help solve this problem, more robust numerical constructs and concepts can be used. In particular, exact numbers, such as rational numbers or arbitrary precision numbers. The latter, also known as *bignums*, allow arbitrary-precision arithmetic, capable of representing numbers with virtually infinite precision with the drawback that arithmetic operations are slower, however mitigating precision issues, providing more accurate constructs and improving code robustness.

Several libraries already strive to implement robust geometric computation. One such example is the Computational Geometry Algorithms Library (CGAL) [60]. CGAL is a comprehensive library that employs an exact computation paradigm [61], producing correct results despite roundoff errors and properly handling *degenerate* situations (e.g., 3D points on a 2D plane), relying on numbers with arbitrary precision to do so. Moreover, other libraries, such as the Library for Efficient Data types and Algorithms (LEDA) [62], and CORE [63] and its successor [64], also deal with robustness problems in geometric computation, offering simpler interfaces when compared to CGAL. However, CGAL arguably remains the *de facto* standard library for robust exact geometric computation.

2.2 Geometric Constraint Tools

Constraint-based programming comes in a wide variety of ways, following a diverse set of programming paradigms, using different approaches to problem solving briefly detailed in section 1.3. Some of them also support an associative programming model, such as DesignScript [65], further discussed in section 2.3.1, allowing for the propagation of changes made to a variable to others that depended on the former.

Table 2.1 succinctly analyzes tools capable of solving geometric constraints. From this table, Eukleides, GeoSolver, and the TikZ & PGF system are further discussed: Eukleides for its elegant declarative language, similar to some of the languages outlined in table 2.2; GeoSolver for its helpful analysis Graphical User Interface (GUI), along with the fact it is implemented in Python, a well established and easy to use language, already used in some competence in CAD software (see table 2.2); and TikZ for its wide support, development, usage, and collection of packages that extend it, enabling the specification of graphics and geometry in a variety of simple distinct ways.

Table 2.1: Table of tools and languages with GCS capabilities.

Tool	TPL	VPL	Assoc†	Decl‡	Imp*	2D	3D
DesignScript [65]	✓	✗	✓	✗	✓	✓	✓
Eukleides [52]	✓	✗	✗	✓	✓	✓	✗
GeoGebra [66]	✓	✓	✗	✗	✓	✓	✓
GeoSolver [57]	✓	✓	✗	✗	✓	✓	✓
Kaleidoscope¶ [67]	✓	✗	✓	✗	✓	≈	≈
ThingLab [35]	✗	✓	✓	✓	✗	✓	✓
TikZ & PGF [51]	✓	✗	✗	✗	✓	✓	✗

¶ — Doesn't natively support GCS, but can be extended to solve this class of constraint problems. † — Associative model / *change-propagation* mechanism; ‡ — Declarative paradigm; ∗ — Imperative paradigm

2.2.1 Eukleides

Devoted to elementary plane geometry, Eukleides is a simple, full-featured, and mainly declarative programming language, capable of handling basic data types, such as numbers and strings, and most importantly, geometric data types, such as points, vectors, lines, and circles. Like most languages, it provides control flow structures, allows user functions and module definitions, making it easily extendable.

Eukleides provides a wide variety of functions and constructions that easily allow the user to specify geometric constraints between objects, as demonstrated by listings 1.2 and 1.4. Among the listed ones, it includes functions to build parallel and perpendicular lines with respect to another line or segment, determine a line's bisector, tangent lines to a circle, shape intersection, and so on. It can generate Encapsulated PostScript (EPS) files or produce macros, enabling the embedding of Eukleides figures in LaTeX documents.

Alas, the lack of 3D geometry support arguably constitutes Eukleides' primary disadvantage. It is also a TPL, which means that, while being a very simple language, it is less intuitive than a VPL. The first version of Eukleides included a GUI, `xeukleides`, but one is not yet available for the current version.

Additionally, it has not seen any development for the last decade, while TikZ is still being actively maintained. The latter still dominates diagram and graphic production in LaTeX documents, some people[1] going as far as suggesting opting for it instead of using the former.

2.2.2 GeoSolver

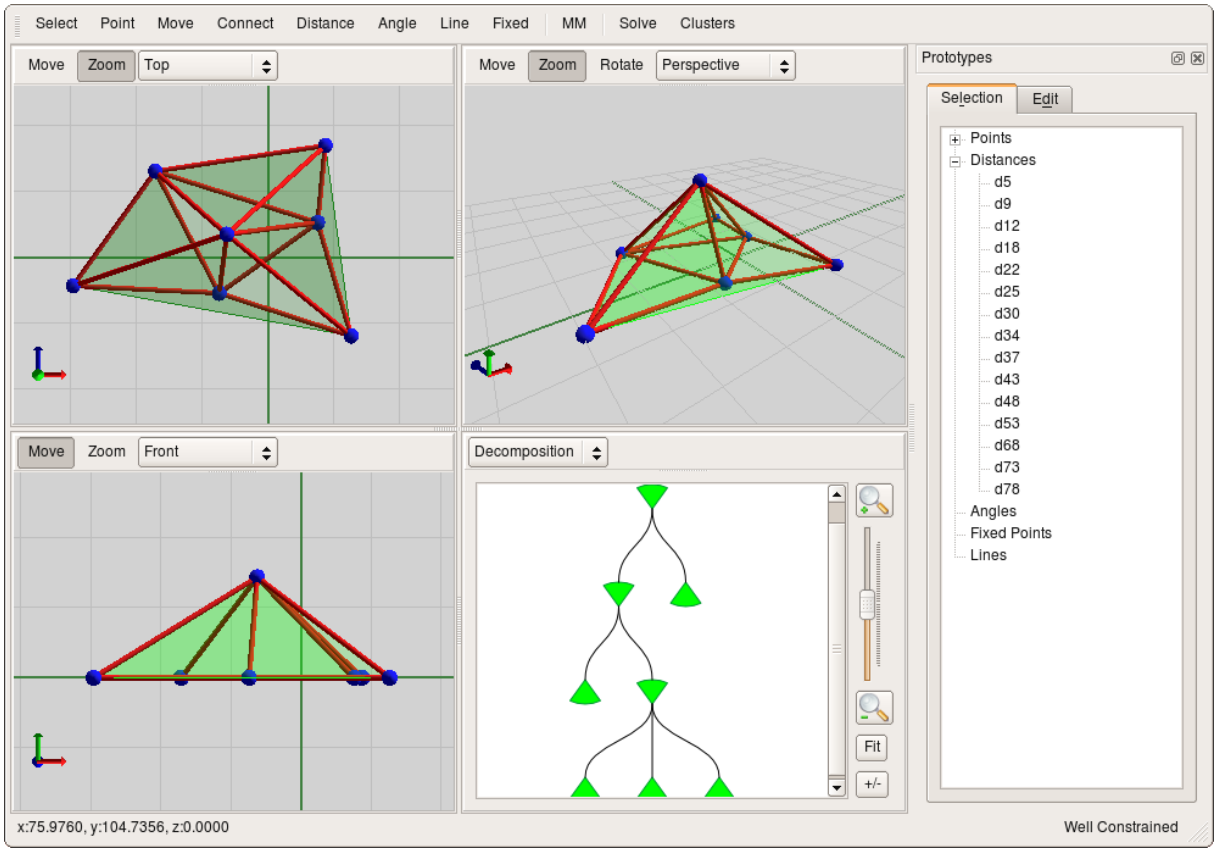
GeoSolver is an open-source Python package that provides classes and functions for specifying, analyzing, and solving geometric constraint problems. It features a set of 3D geometric constraint problems

[1] https://tex.stackexchange.com/a/208412/178614

consisting of point variables, two-point distance and three-point angle constraints. Problems with other geometric variables can be mapped to these basic constraints on point variables.

The solutions found by GeoSolver are generic and parametric, and can be used to derive specific solutions. Since generic solutions are exponentially hard to find, GeoSolver also allows different ways of reducing the number of solutions that would be generated, consequently reducing computation time. In order to efficiently find a solution, GeoSolver employs a cluster rewriting-based approach described in [57], capable of handling non-rigid clusters contrasting with typical graph constructive-based approaches.

A GUI interactive tool called GCS Workbench [68] (see fig. 2.1) is distributed along with the GeoSolver package. With it, the user can easily edit, analyze and solve geometric constraint problems. The latter features are obviously supported by GeoSolver, and 3D interactivity is supported via additional libraries, such as pyQt and pyOpenGL. Although an excellent tool for understanding how a geometric constraint problem is decomposed in GeoSolver, GCS Workbench is not efficient for complex design tasks when compared with its programmatic supporting package.



Source: http://geosolver.sourceforge.net (Jan 2019)

Figure 2.1: Depiction of the GCS Workbench's GUI with two separate panes: (1) showcasing different perspectives of the model and the constraint problem's decomposition, and (2) a prototyping pane, destined for constraint analysis and edition.

2.2.3 TikZ & PGF

Originally a small LaTeX style created by Till Tantau for his Ph.D. thesis, TikZ [51], along with its underlying lower-level Portable Graphics Format (PGF) system, is a fully featured graphics language, basically consisting of a series of TeX commands that draw graphics. TikZ stands for "TikZ ist *kein* Zeichenprogramm", a recursive acronym, which translates to "TikZ is no drawing program". As mentioned, the user instead programmatically describes their drawings.

On its own, TikZ already includes a series of commands capable of handling geometric constraints, such as tangency, perpendicularity, intersection; but may appear daunting to the user in its raw form. Several packages have been built on top of it to facilitate the generation of drawings using a simpler syntax, such as `tkz-2d` and its successor `tkz-euclide`[2]. The package `tkz-euclide` was designed for easy access to the programming of Euclidean geometry using a Cartesian coordinate system with TikZ. It was used to produce fig. 1.2 with the respective code listed in listings 1.1 and 1.3.

Like Eukleides, an obvious limitation they share is the lack for 3D modeling support. Unlike it, a plethora of resources and usage examples exist, along with an immense amount of packages that layer on top of it for a panoply of diverse use cases. It still undoubtedly remains the go-to graphics system within the TeX typesetting community. However, again comparing it to Eukleides, even using something as `tkz-euclide`, it can look syntactically appalling, even for the adept TeX user, instead of following a simpler and established familiar syntax akin to other declarative or imperative programming languages.

2.3 Algorithmic Design Tools

As discussed in section 1.5, AD tools have been integrated into several modern CAD and BIM applications, using TPLs, VPLs, or even a mixture of both approaches.

Other tools, like JSCAD[3] and ImplicitCAD[4], are standalone CAD software hosted on the web. Being cloud-based is advantageous in many fronts: it is inherently portable and removes the additional typical installation steps required for desktop applications. Alas, being relatively new, they lack features in comparison to the immense feature-set of applications such as AutoCAD.

Table 2.2 succinctly summarizes a list of CAD software that supports the usage of a programming language, as well as other AD tools that live detached from existing CAD software. From there, Dynamo and Grasshopper are further comparatively discussed, being relatively similar tools integrated within CAD/BIM software. Moreover, both include TPL and VPL support in different forms.

[2] https://ctan.org/pkg/tkz-euclide
[3] https://openjscad.xyz
[4] https://implicitcad.org

Table 2.2: CAD/BIM software with programmatic capabilities and AD software/tools. Added notes per tool shortly outline deemed significant characteristics.

Application	Tool	TPL	VPL	Note
AutoCAD	.NET API	✓	✗	Powerful, but very verbose; C# & VB.NET
	ActiveX Automation	✓	✗	Deprecated, bundled separately; VBA
	Visual LISP	✓	✗	IDE; AutoLISP extension
Dynamo Studio Revit	Dynamo	✓	✓	Data flow paradigm; Associative programming support through DesignScript
Archicad Rhinoceros3D	Grasshopper	✓	✓	Data flow paradigm; Rhino SDK access, C# & VB.NET
	Python Scripting	✓	✗	Simple language; Create custom Grasshopper components
	RhinoScript	✓	✗	VBScript based
Standalone†	ImplicitCAD	✓	✗	Web hosted; OpenSCAD inspired
	JSCAD	✓	✗	Web hosted; JavaScript
	OpenSCAD	✓	✗	Solid 3D models; Simple domain language
	Rosetta [54]	✓	✗	Portable tool; Multiple front- and back-end support

†These tools are standalone software, i.e., not directly integrated into any specific CAD application.

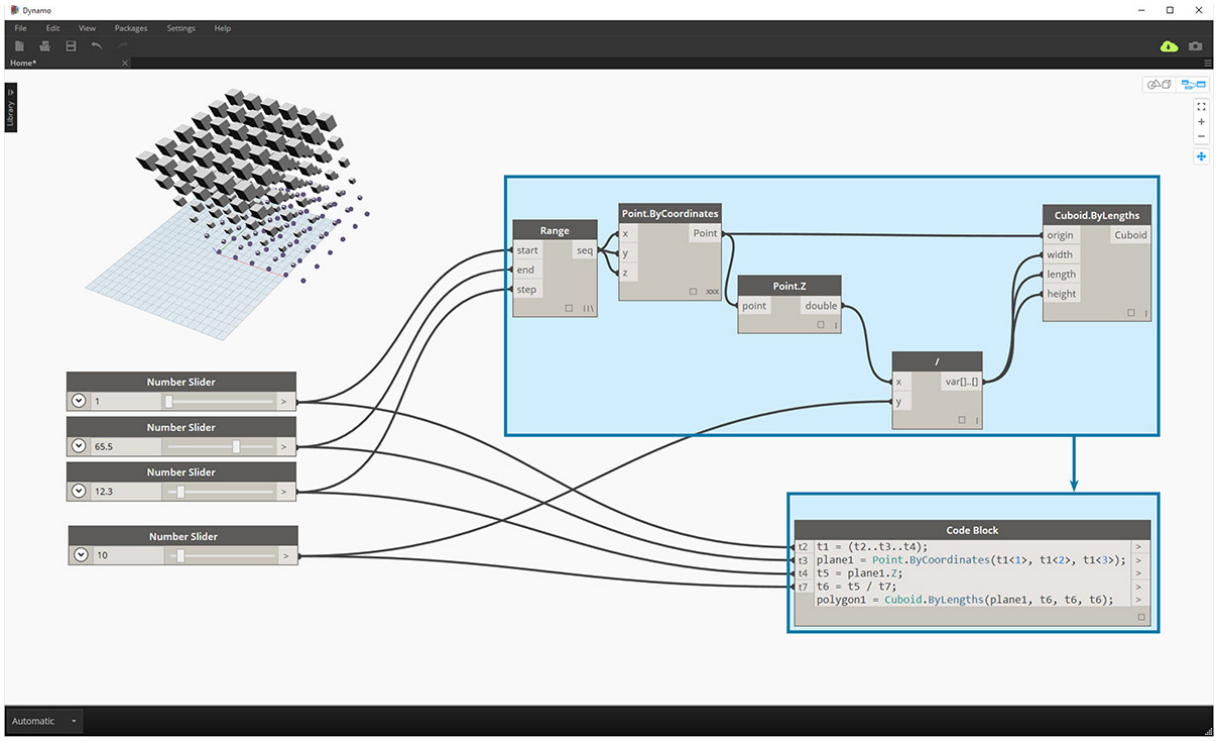
2.3.1 Dynamo

An open source AD tool available as a plug-in for Revit or by itself within Dynamo Studio, Dynamo extends BIM with the data and logic environment of a graphical algorithm editor. Dynamo can be used through both a VPL and a TPL, showcased in fig. 2.2.

In its visual form, Dynamo offers a wide variety of functions, called nodes, most of them capable of generating an even wider variety of geometry through node combination, wiring one's outputs to another's inputs, and resorting to predefined mutable parameters which can serve as some of the nodes' initial inputs. The workflow itself is the final product: a visual program, usually designed to execute a specific task. Dynamo further allows extension through the creation of custom nodes which can be shared as packages.

One of the nodes in Dynamo, aptly named code block, allows the usage of a TPL; a language called DesignScript.[5] Originally developed by Robert Aish [65], DesignScript is a multi-paradigm domain-specific language and is the programming language at the core of Dynamo itself. So much so that entire workflows can be reduced to one code block (see fig. 2.2).

[5] https://primer.dynamobim.org/10_Custom-Nodes/10-4_Python.html — Dynamo is also capable of interpreting Python code.



Source: http://primer.dynamobim.org/en/07_Code-Block/7-2_Design-Script-syntax.html (Jan 2019)

Figure 2.2: Showcase of Dynamo's visual interface containing a workflow that produces the model on the top left. The figure also shows Dynamo's capability of converting a workflow into a single DesignScript code block.

DesignScript is an associative language, which maintains a graph of dependencies with variables. Executing a script will effectively propagate the variables' values accordingly. By default, code blocks in Dynamo follow an associative paradigm. The user can, however, switch to an imperative paradigm approach instead effortlessly if needed.

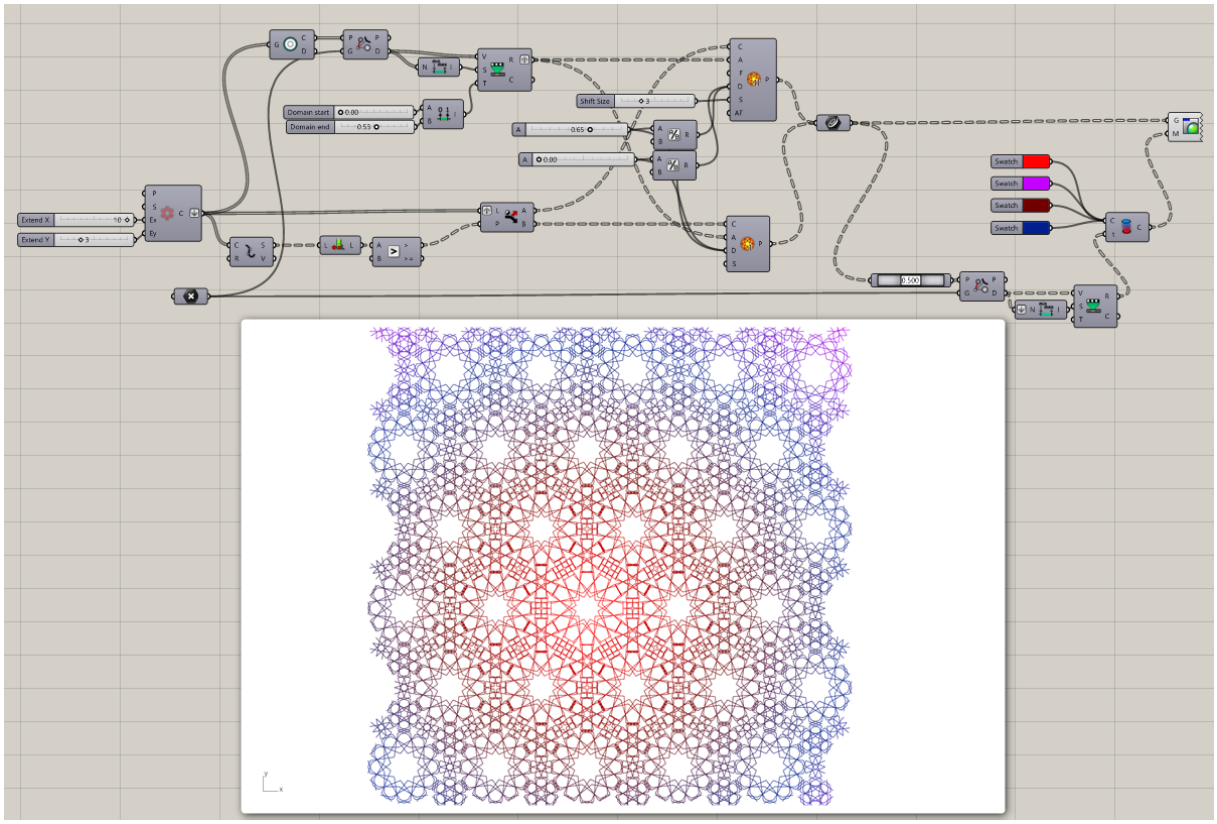
This *change-propagation* mechanism in DesignScript, consequently present in Dynamo, makes Dynamo a great tool for dealing with constraints. However, most users might not fully exercise DesignScript's associative capabilities and instead approach the problem with the mindset of an imperative programming paradigm given its overwhelming presence in and adoption by major well-known TPLs.

2.3.2 Grasshopper

Grasshopper is a graphical algorithm editor tightly integrated with Rhinoceros3D, destined for designers who are exploring generative algorithms. In spite of tight integration with Rhino, a CAD application, it is possible to use Grasshopper along with Archicad®[6], a BIM tool, as well as Revit, through Hummingbird[7]

[6] https://graphisoft.com/solutions/archicad
[7] https://www.food4rhino.com/en/app/hummingbird

or Lyrebird[8]. Figure 2.3 shows a simple example of a Grasshopper workflow.



Source: https://www.grasshopper3d.com/photo/islamic-pattern-parakeet (Jan 2019)

Figure 2.3: Islamic Pattern, by Esmaeil Mottaghi. On top is the Grasshopper workflow to produce the pattern below it, aided by Parakeet[9].

It is a closed-source product, designed by David Rutten and developed by McNeel and Associates, Rhino's developers. Its VPL is simple to use, which is crucial for users who are not familiar with programming using TPLs. Nonetheless, it offers a TPL alternative by way of custom programmatic components. Using C# or VB.NET, the user can create custom code components with access to Rhino's Software Development Kit (SDK) and openNURBS[10] within Rhino. Alternatively, through GhPython[11], the user can also write Python code. Unlike Dynamo's DesignScript, Python and the .NET languages don't support an associative programming model.

Functions in Grasshopper are called components and work just like Dynamo's nodes; a wide variety of them exist, most of them capable of producing geometry, and are composable.

[8] https://www.food4rhino.com/en/app/lyrebird
[9] https://www.food4rhino.com/app/parakeet
[10] https://developer.rhino3d.com/guides/opennurbs/what-is-opennurbs
[11] https://www.food4rhino.com/app/ghpython

2.3.3 Visual Programming Scalability

Both Dynamo and Grasshopper's visual approach suffer from the disproportionate scalability between the code and the respective model's complexity [53]. Sophisticated modeling workflows tend to become difficult to create, and harder for a human to understand when compared to a textual approach.

As an example, consider the Irina Viner-Usmanova RGC, a project developed by TPO Pride[12]. The RGC, built in Moscow's Luzhniki Stadium, is an arena that houses training sessions and competitions while also comprising several other premises. Figure 2.4 depicts an outside view of the RGC and its prominent overarching roof covering. The roof covering was designed using a combination of Rhino3D and Grasshopper. Grasshopper was used from a conceptual stage all the way through to production of construction drawings.

Source: https://www.grasshopper3d.com/photo/rhythmic-gymnastics-center-moscow-russia-5 (Jul 2021)

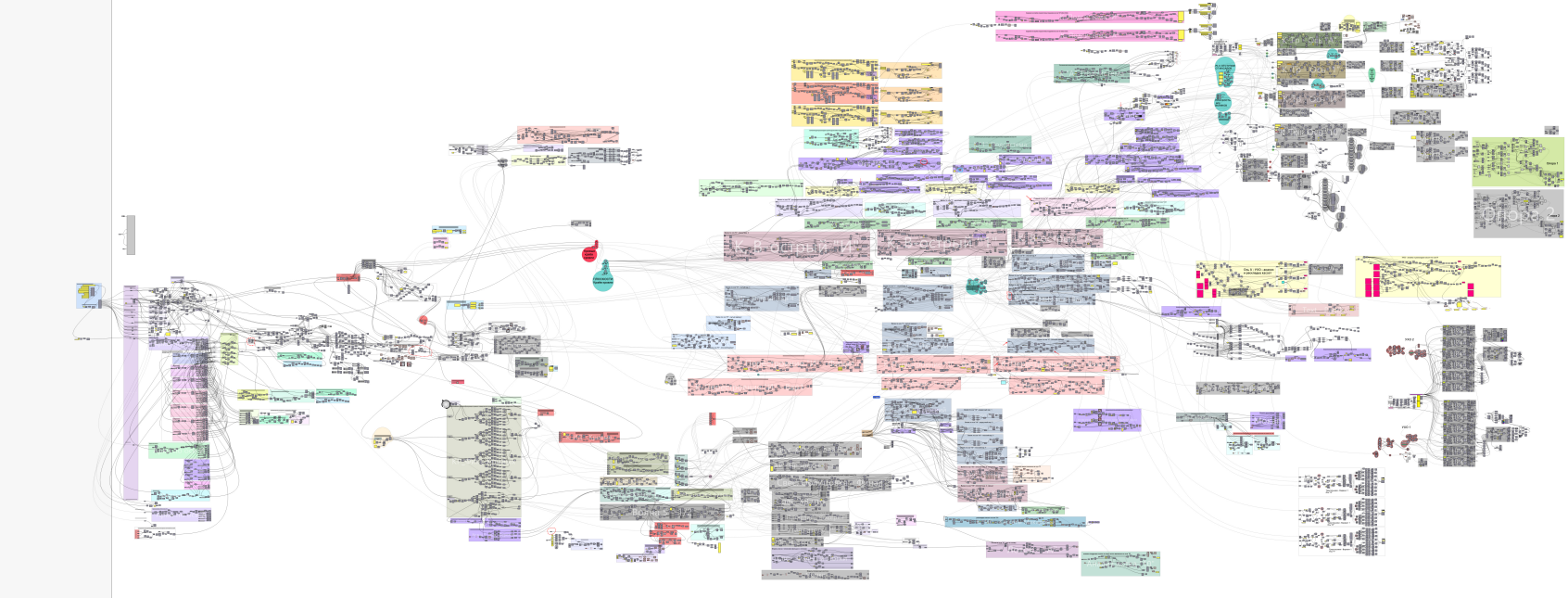
Figure 2.4: Irina Viner-Usmanova RGC in the Luzhniki Complex, Moscow, Russia. The roof covering was designed using Rhino/Grasshopper.

Developing a roof covering with such a contour lends itself well to AD since it resembles a sine wave whose amplitude is progressively dampened along the length of the building. Such a shape can be easily described through a relatively simple mathematical function to obtain the general wave's shape.

[12] http://prideproject.pro (July 2, 2021)

With parametrization in mind, one could then easily fluctuate input variables in order to achieve different variations of the roof covering's shape, e.g., varying the wave's frequency, amplitude, or dampening rate.

Such complex AD projects tend to require complex AD programs that become overly difficult to develop and understand with VPLs. The final Grasshopper definition of the RGC roof's covering can be seen in fig. 2.5. This further reinforces the statement that complex workflows become exponentially difficult to comprehend due to the added dimensionality of the constructs used in VPLs. This disadvantage, however, is mitigated with their respective TPL alternatives which, despite project complexity, scale relatively better than VPLs on account of abstraction mechanisms.



Source: https://www.grasshopper3d.com/photo/final-definition (Jul 2021)

Figure 2.5: Final Grasshopper definition of the Irina Viner-Usmanova RGC roof covering.

3

Solution

Contents

Despite strides in enhancing performance and efficiency of geometric constraint solving approaches, briefly discussed in section 1.3, the core issue lies in the generality of geometric constraint solvers. Although several approaches employ efficient methods to find a solution, they resort to solving potentially well-known problems generically when computationally lighter solutions exist. Instead of delegating the problem to a solver, a more efficient approach would be to pinpoint the type of geometric constraint itself, specializing a solution for several applicable entities. Take the tangency constraint as an example: positioning two circles tangent to each other or a line tangent to an ellipse. Depending on the inputs, these constraints might have particularly efficient solutions for each case, in kind making the computation more efficient.

Classical numerical methods constitute alluring alternatives to the predominant graph-based approaches. Having been studied for several decades, even if the provided solution does not encompass all the possible values within the problem's domain, they can be used to target specific problems efficiently. Nonetheless, these suffer from the robustness issues discussed in section 2.1, effectively yielding inaccurate solutions if precautions are not taken. A similar argument can be made about algebraic methods.

This work aims to implement a series of geometric constraint primitives in an already expressive TPL to overcome the need for the specification of unnecessary details when modeling geometrically constrained entities, promoting an easier and more efficient usage. Choosing to implement these in a TPL further avoids the poor scalability with increasing code complexity that arises from what could be analogous specifications in a VPL, a subject previously discussed in section 1.5.

Moreover, by relying on an exact geometry computation library, one of the core features of this solution lies in the capability of transparently dealing with plenty of the previously addressed robustness issues. The user can then resort to the implemented primitives, and, by composing them, elegantly specify the set of geometric constraints necessary to produce the idealized model. Since the available primitives will implement specialized solutions for a finite set of shapes, the user can utilize in whichever combination possible during the design process, the solution will be exempt of a generic solver component, potentially boosting performance of design generation.

Figure 3.1 shows the typical AD workflow and how the proposed solution could be integrated with the AD tool. The encapsulated modules in the figure represent the underlying computation library as an external component, featuring the geometric constraint primitives library and the code wrapping the computation library.

The following sections go over the components in fig. 3.1, namely the *Exact Geometric Computation Library*, the *Wrapper Code*, and the *Geometric Constraint Primitives*. Additionally, we discuss a few trade-offs from tackling the problem in this fashion as opposed to potential alternative routes, describing advantages and disadvantages of our approach.

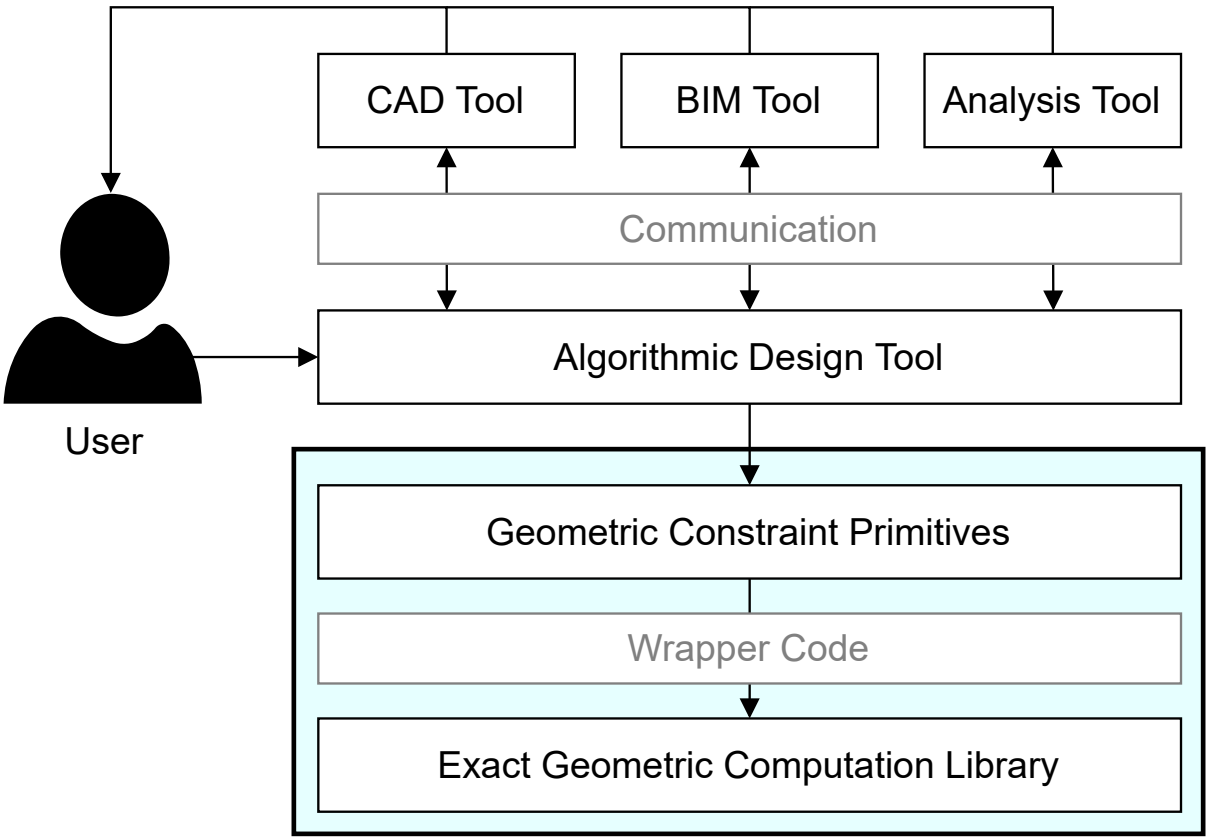


Figure 3.1: General overview of the solution's architecture encapsulated within the blue colored box beneath a depiction of the typical AD workflow.

3.1 Implementation

This section details implementation choices with regard to the chosen platforms for realizing the initially proposed general solution architecture, previously illustrated in fig. 3.1. Following a brief analysis, we expand specifically on the concrete components corresponding to the ones within fig. 3.1's light blue rectangle.

Examining the AD workflow portion of fig. 3.1, there are depictions of CAD, BIM, and analysis tools, of which examples could be Rhinoceros3D, Autodesk's Revit, and Radiance, respectively, with no particular focus on any of them. Digging a layer deeper, we find the AD tool, which, by means made available by the tools above it, produces models specific to those tools from a description provided by the user. The AD tool we've chosen was Khepri[1] [69], a text-based tool written in the Julia programming language [70]. Khepri is the successor of another text-based AD tool named Rosetta [54], a tool written in the Racket programming language [71]. It follows that the *Geometric Constraint Primitives* were implemented in the Julia language as well, supported by an *Exact Geometric Computation Library*. The library chosen for the

[1] https://github.com/aptmcl/Khepri.jl

effect was CGAL [60], a highly performant and robust geometric library written in the C++ programming language [72].

This language disparity between the *Geometric Constraint Primitives* module and the *Exact Geometric Computation Library* requires a solution for language interoperation. In other words, we need to make CGAL available to the Julia language. Fortunately, the Julia language already possesses facilities that allow it to invoke functionality within libraries written in the C [73] or the Fortran [74] programming languages. This interfacing mechanism is commonly known as Foreign Function Interface (FFI). It allows for the repurposing of mature software libraries in foreign languages without the need for a complete rewrite or adaptation.[2] This mechanism can also in turn be leveraged and built upon to interface with other programming languages, e.g., Java, Python, MATLAB, and, C++.[3]

Overcoming the language interoperability hurdle, we can now start focusing on the implementation of the *Geometric Constraint Primitives*. These primitives build on top of the functionality available in CGAL, some of which is directly inherited from it, substantially helping us in the process, e.g., intersections. We further enriched the pool with a few more functions, illustrating a constructive approach to GCS, similar and inspired by the approach of `tkz-euclide`, mentioned in section 2.2.3. By creating an abstraction over more primitive functionality, we aimed to provide an easy to understand and utilize set of tools so that users can avoid reimplementing it themselves, which is an error-prone process. Furthermore, we level the playing field by working at a conceptual level which is more familiar to and understood by traditional CAD software users rather than falling back to the more analytic approach programming languages naturally offer.

The following sections will elaborate further on the components emphasized in the previous paragraphs, adopting a bottom-up-like approach. We will discuss CGAL and what constructs and functionality it can provide to aid our goal, as well as some added benefits of building on top of a very mature and comprehensive library. That will be followed by a section detailing how it was possible to map said functionality to the Julia language, of which the result was a Julia package aptly named `CGAL.jl`[4] [75]. Finally, we showcase how we leveraged `CGAL.jl` to build the aforementioned *Geometric Constraint Primitives*, a set of functionalities that implements specialized yet comprehensible constructive approach solutions to GC problems.

[2]The decision to include such a mechanism at the language's core by the language designers makes it so that the language can rapidly evolve. As such, one can avoid reimplementing several facilities and software libraries in, but not limited to, scientific and numerical computing areas. Arguably, it may be one of *the* fundamental features that made the language as popular as it is and kept it afloat, unlike other similar historical examples that might have lacked such a mechanism.

[3]There is an entire GitHub organization with projects dedicated to foreign language interoperation at `https://github.com/JuliaInterop` (July 8, 2021)

[4]Packages in the Julia ecosystem are conventionally terminated with a `.jl` suffix, the extension used for Julia files. This is reminiscent of a familiar convention followed in the Java ecosystem where libraries and tools are usually prefixed with the letter `J`, e.g., `JUnit`, `JMeter`, `JDeveloper`, among others.

3.1.1 Computational Geometry Algorithms Library

CGAL is a software project that provides easy access to efficient and reliable geometric algorithms in the form of a C++ library. It is considered the industry's *de facto* standard geometric library, used in well-known projects such as OpenSCAD. It is also a very mature software library with decades of Ph.D.-grade research results, still being actively maintained to this day. Being an open-source project, one can easily contribute to it by reporting issues in the software as well as directly submitting patches.[5]

These factors, among others, justify our choice for our solution's *Exact Geometric Computation Library* component. We chose CGAL because of its comprehensiveness and decades of work instead of relying on less mature software, as well as the critical mass of maintainers behind it. That is not to say less mature software cannot be used in its stead, though it is unlikely they can match CGAL, be it in terms of performance, quality, or breadth.

CGAL offers a multitude of data structures and algorithms, such as triangulations, Voronoi diagrams, and convex hull algorithms, to name a few. The library is broken up into three parts [76]:

1. The kernel, which consists of geometric primitive objects and operations on these objects. The objects are represented both as (a) stand-alone classes parameterized by a representation class that specifies the underlying number types used for computation, and as (b) members of the kernel classes, which allows for more flexibility and adaptability of the kernel;

2. Basic geometric data structures and algorithms, parameterized by traits classes that define the interface between the data structure or algorithm and the primitives they use;

3. Non-geometric support facilities, such as circulators, random sources, and I/O support for debugging and for interfacing CGAL to various visualization tools.

Listing 3.1 showcases an example of a very simple CGAL program, demonstrating the construction of some points and a segment, and performing some basic operations on them.

As mentioned, geometric primitive types are defined in the kernel. The chosen kernel in the example uses `double` precision floating point numbers for the Cartesian coordinates of the point.

We can also see some predicates, such as testing the orientation of three points, and constructions, like the distance[6] and midpoint computation. Predicates typically produce a boolean logical value or one of a discrete set of possible results, whereas constructions produce either a number or another geometric entity.

It is worth noting that floating point-based geometric computation can lead to surprising results since we are relying on inexact constructions. If one must ensure that the numbers get interpreted at their

[5] The library's source is hosted on GitHub at `https://github.com/CGAL/cgal`. To illustrate the ease with which one can contribute to the project, here is a pull request the author submitted: `https://github.com/CGAL/cgal/pull/4705`.

[6] It is worth noting CGAL does `not` compute the absolute distance, offering instead to compute the squared distance, avoiding the additional square root computation. This preserves exactness and avoids a potentially unnecessary heavy computation.

Listing 3.1: An example CGAL program illustrating how to construct some points and a line segment, and perform some basic operations on them. It uses `double` precision floating point numbers for Cartesian coordinates.

```
#include <iostream>
#include <CGAL/Simple_cartesian.h>

typedef CGAL::Simple_cartesian<double> Kernel;
typedef Kernel::Point_2 Point_2;
typedef Kernel::Segment_2 Segment_2;

int main()
{
  Point_2 p(1,1), q(10,10);

  std::cout << "p = " << p << std::endl;
  std::cout << "q = " << q.x() << " " << q.y() << std::endl;

  std::cout << "sqdist(p,q) = "
            << CGAL::squared_distance(p,q) << std::endl;

  Segment_2 s(p,q);
  Point_2 m(5, 9);

  std::cout << "m = " << m << std::endl;
  std::cout << "sqdist(Segment_2(p,q), m) = "
            << CGAL::squared_distance(s,m) << std::endl;

  std::cout << "p, q, and m ";
  switch (CGAL::orientation(p,q,m)) {
  case CGAL::COLLINEAR:
    std::cout << "are collinear\n";
    break;
  case CGAL::LEFT_TURN:
    std::cout << "make a left turn\n";
    break;
  case CGAL::RIGHT_TURN:
    std::cout << "make a right turn\n";
    break;
  }

  std::cout << " midpoint(p,q) = " << CGAL::midpoint(p,q) << std::endl;
  return 0;
}
```

full precision, CGAL offers kernels that perform exact predicates and exact constructions. Revisiting listing 3.1, it is as simple as switching the `Simple_cartesian` kernel with one the provides exact constructions, e.g., `Exact_predicates_exact_constructions_kernel` or `Epeck` for short.

However, CGAL is a terribly complex library under the hood, presenting many challenges when it comes to mapping it to the Julia language. Firstly, it is a C++ library. Despite Julia's native capabilities for interoperating with C, there's additional work to be done to reach C++ code. Secondly, is problem of memory management, which differs between C/C++ and Julia, potentially leading to memory leaks and other related issues if not properly tended to. Finally, CGAL makes extensive use of C++ templates, proving sometimes difficult to transparently map some of its constructs.

Fortunately, there are both methods and additional libraries that aid us by transparently overcoming some of those issues. In the next section, we go over how we overcame these issues, demonstrating it by reproducing the example in listing 3.1 in Julia.

3.1.2 From C++ to Julia

Having established CGAL as our *Exact Geometric Computation Library* of choice, we must now overcome the quite literal language barrier between Julia and C++. Fortunately, the former possesses FFI mechanisms that can aid us in resolving this issue. Here is an excerpt from the language's manual about Julia's C and Fortran FFI capabilities:[7]

> *To allow easy use of (…) existing code, Julia makes it simple and efficient to call C and Fortran functions. (…) This is accomplished just by making an appropriate call with* **`ccall`** *syntax, which looks like an ordinary function call.*

To illustrate how this **`ccall`** construct can be used in the context of our problem, we created a small wrapper around CGAL exposing the `squared_distance` function. The original function we are interested in takes two instances of 3D points. To facilitate the wrapping, we create the points on the C++ side of things, instead passing primitive **`double`** values representing the points' coordinates. The C++ wrapper library is shown in listing 3.2.

To invoke our wrapper function, we must precede it with **`extern "C"`** as is illustrated. This is important because C++ compilers mangle function names. This technique is employed to solve a series of problems in order to support features like function overloading. By preventing this from happening, we can then refer to our wrapper function by its declared name, just like a C function.

After compiling the library, we can invoke the newly wrapped function from Julia using **`ccall`**, as showcased in listing 3.3.

[7] From https://docs.julialang.org/en/v1/manual/calling-c-and-fortran-code/

```cpp
#include <CGAL/Simple_cartesian.h>
#include <CGAL/squared_distance_3.h> // squared_distance

extern "C" // C function to be invoked in Julia using `ccall`
double squared_distance(double x1, double y1, double z1,
                        double x2, double y2, double z2) {
  typedef CGAL::Simple_cartesian<double>::Point_3 Point_3;
  Point_3 p(x1, y1, z1), q(x2, y2, z2);
  return CGAL::squared_distance(p, q);
}
```

Listing 3.2: Example C library code that wraps CGAL's `squared_distance` global function. The original function takes in instances of `Point_3` classes, so we instantiate them from our `double` coordinate inputs.

```julia
const lib = joinpath(@__DIR__, "libsqdist") # path to the compiled library

squared_distance(x₁::Real, y₂::Real, z₁::Real
                , x₂::Real, y₂::Real, z₂::Real) =
    ccall((:squared_distance, lib)    # qualified function name
        ,  Float64                    # return type
        , (Float64, Float64, Float64
        ,  Float64, Float64, Float64) # parameter types
        ,  x₁, y₁, z₁, x₂, y₂, z₂)    # arguments

let p = (0, 0, 0), q = (3, 4, 0)
    @info "Squared distance" p q squared_distance(p..., q...) # = 25.0
end
```

Listing 3.3: Example Julia program that invokes the functionality from the library whose source is listed in listing 3.2. Julia's `ccall` construct converts the input arguments' types to the types specified in the native C function's parameter types.

Though this looks like a good start, the number passing strategy soon shows its limitations. For example, think of the combinatorial explosion problem that may arise when a function requires a number M of N-dimensional points. We would then have to create a wrapper that takes $N \cdot M$ coordinates. Such an approach does not scale. It is possible to overcome this limitation by mirroring C `struct`s in Julia.

As an example, we consider the `circumcenter` function from CGAL, a function that takes three points as its parameters, returning the circumcenter about the given points, under the assumption the points are not collinear. We could try and directly mirror CGAL's `Point_3` type, but that would require that we know its layout. Even then, we would be breaking an abstraction barrier that could prove detrimental if CGAL's developers decide to change the type's internal representation. To circumvent this completely, we can just create a `struct` that opaquely wraps around CGAL's type. The C++ wrapper code for this example is listed in listing 3.4.

The wrapper code looks very similar to the previous example in listing 3.2. This time, however, we introduced a new `struct Point` that we will mirror in Julia so that we can seamlessly pass instances of

```cpp
#include <CGAL/Simple_cartesian.h>
#include <CGAL/Kernel/global_functions.h> // circumcenter

struct Point { double x, y, z; }; // C struct wrapping Point_3

extern "C" // C function to be invoked in Julia using `ccall`
Point circumcenter(Point p, Point q, Point r) {
  typedef CGAL::Simple_cartesian<double>::Point_3 Point_3;
  Point_3 _p(p.x, p.y, p.z), _q(q.x, q.y, q.z), _r(r.x, r.y, r.z)
        , _s = CGAL::circumcenter(_p, _q, _r);
  return Point{_s.x(), _s.y(), _s.z()};
}
```

Listing 3.4: Example C shared library source code that wraps CGAL's circumcenter global function. In this instance, we use an additional struct to wrap around CGAL's `Point_3` class to facilitate data transfer.

it to our externalized C++ function.[8] All the wrapper function does is take in our `Points` and create new `Point_3` objects, using them to invoke CGAL's `circumcenter` function. The returned `Point_3` is then used to create a `Point`, which is later sent back upstream to Julia.

On the Julia side of things, the process is much the same as before with the addition of a new `Point` type that contains three `Float64` fields. The previous example showed the latter Julia type's correspondence to the C/C++ `double` type. The Julia code for this example is listed in listing 3.5.

```julia
const lib = joinpath(@__DIR__, "libcirc") # path to the compiled library

struct Point # julia equivalent struct
    x::Float64
    y::Float64
    z::Float64
end

circumcenter(p₁::Point, p₂::Point, p₃::Point) =
    ccall((:circumcenter, lib)  # qualified function name
        ,  Point                # return type
        , (Point, Point, Point) # parameter types
        ,  p₁, p₂, p₃)          # argument types

let p = Point(1,2,3), q = Point(1,1,1), r = Point(0,1,2)
    @info "Circumcenter" p q r circumcenter(p,q,r) # = Point(1.0, 1.5, 2.0)
end
```

Listing 3.5: Example Julia program that invokes the functionality from the library listed in listing 3.4. We use an additional Julia struct that is equivalent to the one specified in C to facilitate data transfer.

We are once again met with a very familiar snippet of code. Much like with its respective wrapper library, there is a new `struct Point` that mirrors the one we created in C++. From then on, the process

[8] Note that, unlike the function, the `struct` was not externalized. This is not necessary because we do not need to refer it by name. We need only to match its field layout.

is exactly the same.

So far, we were able to extract relatively useful functions from CGAL. In fact, the latter already solves the problem exemplified in a previous chapter in section 1.4.2. Although we could incrementally build upon this approach, not only does it become cumbersome, but it proves impractical, given CGAL's complexity.

Fortunately, the Julia community has explored methods of interoperating with many other languages, one of them being C++. That exploration resulted in packages like `CxxWrap.jl`[9]. `CxxWrap.jl` adopts an approach to language interoperation similar to that of BOOST.Python [77] or pybind11[10]. The user writes the code for the Julia wrapper in C++, and then simply issue an instruction on the Julia side to initialize the library, making it available there. The Julia package is supported by a C++ component known as `libcxxwrap-julia`, or the friendlier name JlCxx. This component is what the C++ wrapper code depends on. Listing 3.6 constitutes the code to wrap necessary functionality to reproduce the example CGAL program in listing 3.1.

We direct our focus to the lines that are highlighted in listing 3.6. We define a function that is later invoked by `CxxWrap.jl`. In it, we start registering the required types and functions that we need in a declarative fashion[11], reminiscent of the builder design pattern. Notice the `JLCXX_MODULE` symbol preceding the function definition. That symbol takes care of properly externalizing the function regardless of the system the library is built for.[12]

After compiling the wrapper code, we can load it on the Julia side resorting to `CxxWrap.jl`. This process is reminiscent of and analogous to the ones illustrated earlier with simpler examples using **`ccall`**. Listing 3.7 shows a bare-bones CGAL Julia module.

All that is necessary to make the functionality we wrapped on the C++ side available is (1) tell `CxxWrap.jl` where the library is, (2) tell `CxxWrap.jl` to initialize itself when the module is loaded, and (3) export the mapped functionality. The rest of the non-highlighted code, both in this example and the previous only serves the purpose of obtaining a human-readable representation of the C++ objects in Julia.

Finally, we reach a point where we are able to reproduce the example listed in listing 3.1 in the Julia language, having mapped all the necessary functionality to do so. Listing 3.8 shows the example, translated from C++.

We illustrated how to repurpose some core functionality on which we can continue to incrementally build upon following a similar approach to that shown in listing 3.6. Doing so, with relatively low effort, we can obtain the primitive objects and functionality on which our *Geometric Constraint Primitives* will

[9] https://github.com/JuliaInterop/CxxWrap.jl
[10] https://github.com/pybind/pybind11
[11] Order with which types and functions are registered matters. This means we cannot add a function that depends on a type we did not yet register.
[12] Think similar to **`extern`** `"C"`, but slightly more robust.

Listing 3.6: C++ wrapper code powered by JlCxx that maps the types and functions needed from CGAL to reproduce the example shown in listing 3.1 in Julia.

```cpp
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h> // Epick
#include <CGAL/enum.h> // Orientation, alias of Sign
#include <CGAL/IO/io.h> // set_pretty_mode

#include <jlcxx/jlcxx.hpp>

// helper for generating CGAL global function wrappers
#define WRAPPER(F) \
  template<typename ...TS> \
  inline auto F(const TS&... ts) { return CGAL::F(ts...); }

WRAPPER(midpoint)
WRAPPER(orientation)
WRAPPER(squared_distance)

template<typename T> // used in julia to pretty print types
std::string to_string(const T& t) {
  std::ostringstream oss("");
  CGAL::set_pretty_mode(oss);
  oss << t;
  return oss.str();
}

JLCXX_MODULE define_julia_module(jlcxx::Module& m) {
  typedef CGAL::Epick       Kernel;
  typedef Kernel::Point_2   Point_2;
  typedef Kernel::Segment_2 Segment_2;

  // types
  m.add_type<Point_2>("Point2")
   .constructor<double, double>()
   .method("x", &Point_2::x)
   .method("y", &Point_2::y)
   .method("_tostring", &to_string<Point_2>);

  m.add_type<Segment_2>("Segment2")
   .constructor<const Point_2&, const Point_2&>()
   .method("_tostring", &to_string<Segment_2>);

  m.add_bits<CGAL::Orientation>("Orientation", jlcxx::julia_type("CppEnum"));
  m.set_const("COLLINEAR",  CGAL::COLLINEAR);
  m.set_const("LEFT_TURN",  CGAL::LEFT_TURN);
  m.set_const("RIGHT_TURN", CGAL::RIGHT_TURN);

  // functions
  m.method("midpoint", &midpoint<Point_2,Point_2>);
  m.method("orientation", &orientation<Point_2,Point_2,Point_2>);
  m.method("squared_distance", &squared_distance<Point_2,Point_2>);
  m.method("squared_distance", &squared_distance<Segment_2,Point_2>);
}
```

```julia
module CGAL

using CxxWrap
export Point2, Segment2,
       COLLINEAR, LEFT_TURN, RIGHT_TURN,
       x, y, midpoint, orientation, squared_distance

@wrapmodule joinpath(@__DIR__, "libcgal_julia") # path to shared library
__init__() = @initcxx # initialize CxxWrap

Base.show(io::IO, x::CxxWrap.CxxBaseRef{<:Real}) = print(io, x[])
for m ∈ methods(CGAL._tostring) # for pretty printing
    @eval Base.show(io::IO, x::$(m.sig.parameters[2])) = print(io, _tostring(x))
end

end # CGAL
```

Listing 3.7: An example Julia module that mimics `CGAL.jl`, wrapping the library produced from listing 3.6. It initializes the library and exports the mapped functionality.

```julia
using CGAL

p, q = Point2(1,1), Point2(10,10)

println("p = $p")
println("q = $(x(q)) $(y(q))")

println("sqdist(p,q) = $(squared_distance(p,q))")

s = Segment2(p,q)
m = Point2(5, 9)

println("m = $m")
println("sqdist(Segment2(p,q), m) = $(squared_distance(s, m))")

print("p, q, and m ")
let o = orientation(p,q,m)
    if     o == COLLINEAR  println("are collinear")
    elseif o == LEFT_TURN  println("make a left turn")
    elseif o == RIGHT_TURN println("make a right turn")
    end
end

println(" midpoint(p,q) = $(midpoint(p,q))")
```

Listing 3.8: The example program as seen in listing 3.1 written in the Julia programming language using `CGAL.jl`. The kernel instantiation is hidden away in the C++ layer of the wrapper code.

be supported. The following section goes over how we effectively used the functionality from `CGAL.jl` to implement constructive solutions for GC problems.

3.1.3 Geometric Constraint Primitives

Having overcome the language barrier between C++ and Julia, we can build our GC primitives on top of `CGAL.jl`. Our implementation of the *Geometric Constraint Primitives* is loosely inspired on tools like `tkz-euclide` and Eukleides. Both are based on Euclidean geometry, a constructive system where the production of geometry can be done solely resorting to a straightedge and a compass. This makes programs described using these constructs easier to understand and manually reproduce.

The following sections include a revisiting of our initially formulated example problems, described in section 1.4, and another set of primitives that will be the driving force behind the case studies we go over later in chapter 4.

3.1.3.A Parallel lines

Revisiting our earlier examples, we now showcase implementations for those problems using our solution, accompanied by visualizations resorting to the Khepri AD tool. Listing 3.9 shows a solution to the "parallel lines" problem introduced in section 1.4.1.

```
using Khepri
import CGAL: Point2, Segment2, to_vector
# implementation
parallel(l::Segment2, p::Point2) = Segment2(p, p + to_vector(l))
# conversion
parallel(l, p) = convert(Line, parallel(convert(Segment2, l)
                                        , convert(Point2, p)))

begin
    backend(autocad); delete_all_shapes()

    with(current_cs, cs_from_o_phi(u0(), deg2rad(20))) do
        A, B, C = u0(), x(3), xy(1,1)
        v = .1(B - A) # small offset
        AB = line(A - v, B + v)
        parallel(AB, C - v)
        surface_circle.((A, B, C), 3e-2)
        text("A", add_pol(in_world(A), .3, -π/3), .2)
        text("B", add_pol(in_world(B), .3, -π/3), .2)
        text("C", add_pol(in_world(C), .3, -π/3), .2)
    end
end
```

Listing 3.9: Implementation of the parallel lines example illustrated in fig. 1.2a using Khepri alongside our solution.

The first line highlighted in the program consists of the implementation of the primitive construct. The `parallel` function takes a line segment `l`, the segment the resulting line is meant to be parallel to; and a point `p` which the resulting line passes through. We then build a new line segment whose starting point is the given point `p`, and the end point is the result of adding the difference between `l`'s endpoints to `p`, obtained using `CGAL.jl`'s `to_vector` function.

We then need to tell Khepri how to create objects our implementation can understand by converting both the inputs and the resulting output. `CGAL.jl` already contains a primitive shim of conversions between some objects that facilitate this effort.[13]

Finally, we can use our primitive as seen highlighted in the listing, reproducing the same result. Figure 3.2 illustrates the formulated program's output in AutoCAD, one of the visualization tools Khepri supports.

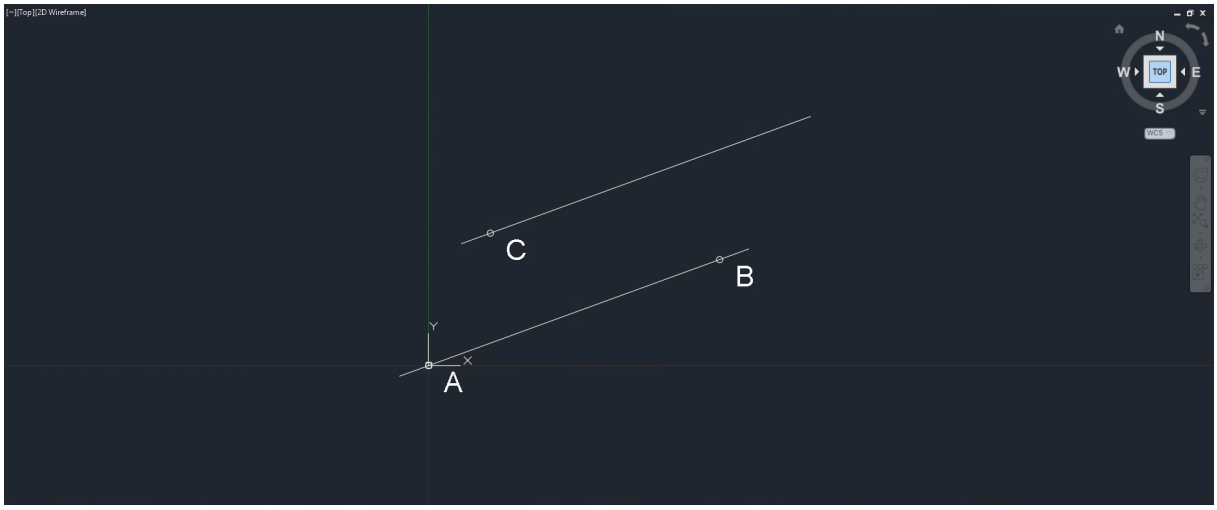


Figure 3.2: Parallel lines example from section 1.4.1 revisited using our solution's `parallel` primitive construct. Output visualized in AutoCAD.

3.1.3.B Circumcenter

Regarding our circumcenter problem, introduced in section 1.4.2, we initially solved it by intersecting two of the three triangle sides' bisectors. We can still approach the problem that way, defining a `circumcenter` function similar to the one in listing 3.10.

```
1  circumcenter(a, b, c) = intersection(bisector(a, b), bisector(b, c))
```

Listing 3.10: Initial implementation of the `circumcenter` function.

However, this is one occasion where this functionality is already present in CGAL. This is a simple

[13] Ideally, the target AD tool would integrate our solution's constructs in a tighter more seamless fashion.

yet perfect demonstration of one of our approach's benefits with regard to repurposing a comprehensive library with plenty of functionality, allowing us near-direct reuse without re-implementing it.

Listing 3.11 illustrates a solution to the "circumcenter" problem using CGAL's `circumcenter` function. The program's output can be seen in fig. 3.3.

```
using Khepri
import CGAL: Point2, circumcenter
# conversion
circumcenter(p, q, r) =
    convert(Loc, circumcenter(convert.(Point2, (p, q, r))...))

right_angle(p, q, r; scale=.2) =
    with(current_cs, cs_from_o_vx_vy(q, p - q, r - q)) do
        line(y(scale), xy(scale, scale), x(scale))
    end
right_angle(ps; kws...) = right_angle(ps...; kws...)

begin
    backend(autocad); delete_all_shapes()

    A, B, C = u0(), xy(1, 3), x(4)
    O  = circumcenter(A, B, C)
    AB = intermediate_loc(A, B)
    AC = intermediate_loc(A, C)
    BC = intermediate_loc(B, C)
    line.((AB, AC, BC), O)
    foreach(right_angle, ((A,AB,O), (B,BC,O), (C,AC,O)))
    polygon(A, B, C)
    circle(O, distance(O, A))
    line(O, A)
    surface_circle.((A, B, C, O), 3e-2)
    text("r", intermediate_loc(O, A) + vy(.1), .2)
    text("A", A + .2vxy(-1, -1), .2)
    text("B", B + .1vxy(-2,  1), .2)
    text("C", C + .1vxy( 1, -2), .2)
    text("O", O + .1vxy( 1, -2), .2)
end
```

Listing 3.11: Implementation of the circumcenter example illustrated in fig. 1.2b using Khepri alongside our solution.

3.1.3.C Circle tangent to a line

Moving on to a set of problems that cannot as directly be solved by CGAL is that of tangencies. Generally speaking, there are specific constructs available in CGAL that solve these problems. However, we can once again repurpose the ones we have and build on top of them.

This problem involves computing a circle that is tangent to an already existing line. Given the circle's center `o` and the line `l` we want the circle to be tangent to, the tangent circle is such that its radius is the

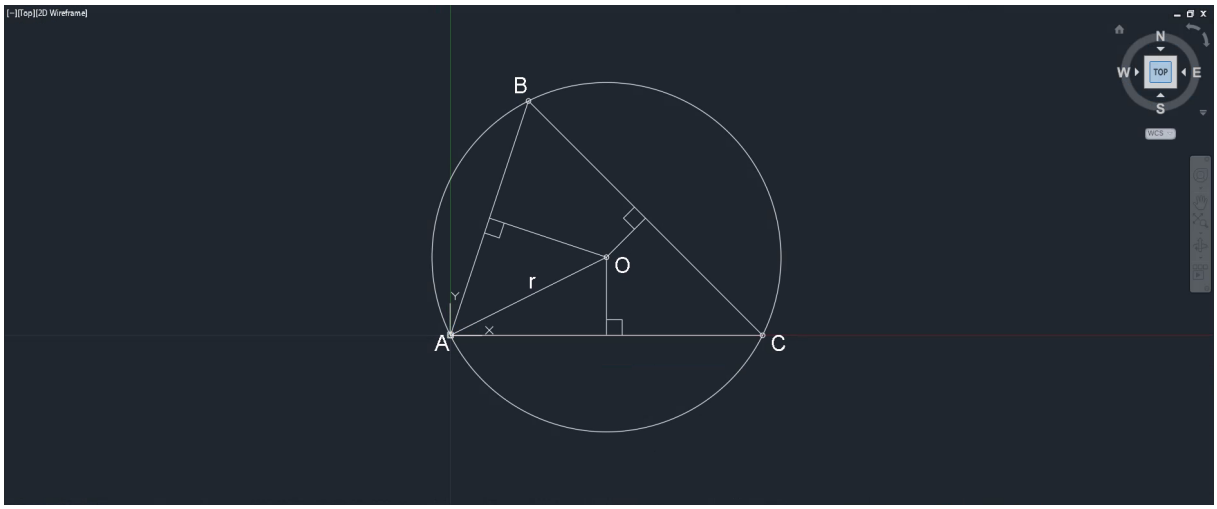


Figure 3.3: Circumcenter example revisited using our solution's `circumcenter` primitive construct. Output visualized in AutoCAD.

minimal distance between its center point `o` and the line `l`. The implementation of the problem's solution can be seen in listing 3.12.

```
1 tangent_circle(o::Point2, l::Line2) = Circle2(o, squared_distance(o, l))
```

Listing 3.12: Implementation of the "Circle tangent to a line" problem.

Notice this solution considers we are working with infinite lines, not necessarily line *segments* in general. Were we to apply the same simple approach to a line segment, the circle would end up tangent to the line segment as well. However, the resulting circle could be tangent to one of the segment's endpoints instead of tangent to a point along the segment. In some cases, the latter behavior may be preferred, for which adjustments to the solution must be made, such as indicating that there is no solution for the given center point.

We did not implement such a scenario regarding line segments, because much like the other primitives, it was created to quench a specific use case's thirst. A potential solution to that specific problem could be obtained reusing the `tangent_circle` function, however. We would then need to verify if the circle would fall somewhere between the segment's endpoints. Otherwise, there would not be a solution available.

3.1.3.D Tangent circles

Another instance of tangency is that of two circles tangent to each other. This proves to be a relatively more complex problem due to how the circles may be positioned.

Given two circles c_1 and c_2, and a vector `v` used to indicate the point of tangency on c_1, we draw a

circle centered on said point with the same radius as c_2 and intersect it with a ray `r'` originating from the former circle's center, giving us a point `f`. This ray may be directed in one of two ways, determining if the resulting circle contains both circles or not. We then compute the bisector `b` between `f` and c_2. We test if we are in a situation where the resulting circle would instead turn out to be a line. If that is the case, we produce an error. Otherwise, we construct the tangent circle.

Listing 3.13 contains the solution to the problem, assisted by our own definition of the intersection function between a circle and a ray, listed in listing 3.14.

```
11 tangent_circle(c₁::Circle2, c₂::Circle2, v::Vector2; inclusive=true) =
12     let r₁² = squared_radius(c₁),
13         r₂² = squared_radius(c₂),
14         r   = Ray2(center(c₁), v),
15         l   = supporting_line(r),
16         p   = intersection(c₁, r),
17         r′  = Ray2(p, inclusive ? -v : v),
18         f   = intersection(Circle2(p, r₂²), r′),
19         b   = bisector(center(c₂), f)
20
21         @assert !parallel(b, l) "infinite circle (tangent line)"
22
23         o = intersection(b, l)
24         Circle2(o, squared_distance(o, p))
25     end
```

Listing 3.13: Implementation of the "Tangent circles" problem.

3.1.3.E Tangent lines between circles

Another problem that surfaced in a case study earlier illustrated in fig. 1.1 is that of computing tangent lines between two circles. It is a slightly more complex problem for which there is no obvious solution. Fortunately, a constructive solution to this problem already exists[14], an approach our implementation is based on.

[14] https://en.wikipedia.org/wiki/Tangent_lines_to_circles#Synthetic_geometry

```
1 intersection(c::Circle2, r::Ray2) =
2     let res = intersection(c, supporting_line(r))
3         isnothing(res) && return res
4         isa(res, Vector) ? let res2 = filter(p->collinear_has_on(r, p), res)
5             isempty(res2) ? nothing :
6             length(res2) == 1 ? res2[1] :
7             res
8         end : res
9     end
```

Listing 3.14: Implementation of the intersection between a circle and a ray.

This problem's complexity lead us to break it into two different problems. First, we must determine the lines tangent to a circle that pass through a given point (listing 3.15). Then, repurposing the solution to that problem, we solve the overarching problem of determining the lines tangent between circles (listing 3.16). This is another example of how we can compose a smaller set of our primitives to solve a larger problem.

```
2  @cxxdereference tangent_lines(p::Point2, c::Circle2) =
3      let o  = center(c),
4          r² = squared_radius(c),
5          dₚ² = squared_distance(p, o)
6
7          if p == o || dₚ² ≤ r²
8              []
9          elseif r² == 0
10             [Segment2(o, p)]
11         else
12             ps = intersection(c, Circle2(midpoint(p, o), dₚ² / 4))
13             y(o) ≤ y(p) && reverse!(ps)
14             Segment2.(ps, Ref(p))
15         end
16     end
```

Listing 3.15: Implementation of the "Tangent lines to a circle" sub-problem.

Our implementation returns the results in a deterministic way, facilitating result choice. The line segments are oriented from c_1 to c_2, with the internal tangents surrounded by the external tangents, sorted in a counterclockwise orientation.

3.2 Trade-offs

Since virtually nothing comes without trade-offs and compromises, it is paramount we address our implementation's qualities, negative and positive.

Relying on a library such as CGAL proves to be as great as it can be daunting. As mentioned in section 3.1.1, CGAL is a very comprehensive and mature software library, arguably even far exceeding our solution's needs, yet fitting it perfectly.

It is, however, an external component, and with every such component, we do not hold as much control over it as if it was internal instead. For example, in the advent a bug is found within CGAL, one cannot *immediately* fix it by altering its source code and use this fixed version. Important emphasis on bugs not being *immediately* fixable lest we forget CGAL is still an open-source project arguably anyone can contribute to. Alas, said contribution deployments are still out of our control. In hindsight, however, it can be considered as just an inconvenience since it is a project that is actively maintained by very knowledgeable in the computer graphics and mathematics fields.

Listing 3.16: Implementation of the “Tangent lines between circles” problem by way of composition with the solution of “Tangent lines to a circle”.

```
@cxxdereference function tangent_lines(c₁::Circle2, c₂::Circle2)
    r₁², r₂² = squared_radius.((c₁, c₂))

    if r₂² > r₁² # swap arguments
        ss = tangent_lines(c₂, c₁)
        isempty(ss) && return ss
        ss[1], ss[end] = ss[end], ss[1]
        return opposite.(ss)
    end

    o₁, r₁  = center(c₁), √r₁²
    o₂, r₂  = center(c₂), √r₂²

    # Auxiliary
    translation(v) = AffTransformation2(TRANSLATION, v)
    translate(s, v) = transform(s, translation(v))
    vec(s, r) = (source(s) - o₁) * r₂ / r

    # External Tangents
    rₑ = r₁ - r₂
    ssₑ = map(tangent_lines(o₂, Circle2(o₁, rₑ^2))) do s
        iszero(rₑ) && return s
        translate(s, vec(s, rₑ))
    end

    isempty(ssₑ) && return ssₑ # no external tangents ⇒ no tangents
    if length(ssₑ) == 1 # r₁ == r₂
        s   = first(ssₑ)
        vₛ  = vector(direction(s))
        lₛ² = squared_length(s)
        vᵧ  = √(r₂² / lₛ²) * perpendicular(vₛ, orientation(c₁))
        ssₑ  = translate.(Ref(s), (vᵧ, -vᵧ))
    end

    # Internal Tangents
    rᵢ = r₁ + r₂
    ssᵢ = map(s -> translate(s, -vec(s, rᵢ)),
              tangent_lines(o₂, Circle2(o₁, rᵢ^2)))

    # Result
    [ssₑ[1], ssᵢ..., ssₑ[end]]
end
```

CGAL is also a highly generic library, making use and further abusing C++ templates. Although its design makes usage an elegant experience (as elegant as C++ can be), the same cannot be said when trying to wrap its constructs to another language, especially a language with different memory management paradigm which could lead to some nasty low-level ordeals. Luckily, `CxxWrap.jl` helps us solve most, if not all, of those issues.

Regarding our wrapper code, there is one small problem we still technically did not solve. We are still mapping CGAL types in an opaque fashion, fixing the kernel on the C++ side.[15] Alternative methods of supplying a different kernel were explored, including deploying a different library compiled with a different kernel. Despite not entirely solving the problem, it offered the user the choice of using exact constructions. However, this alternative became unmaintainable and virtually impossible to successfully employ. In part due to Julia's package pre-compilation, it is no longer viable to attempt and use the other library.

Ideally, the geometric objects would be parametric, and the kernel (or kernels) mapped to Julia as well, but we were met with complications that would make it unfeasible for us to implement our solution. As a compromise, `CGAL.jl` fixed a kernel that provides exact predicates with inexact constructions, favoring performance over some robustness loss. Nonetheless, in practical terms, it suffices for our case.

As an alternative to wrapping CGAL, we could have explored other options in the still growing Julia ecosystem. Some work seems promising,[16] but not only are some libraries still catching up, it is also highly unlikely they will ever meet the quality of CGAL.

Lastly, some of our GC primitives employ some computation that can lead to robustness loss as well. Primarily the ones that involve circles, non-linear geometric objects, typically require the computation of a square root to work with absolute distances. We tend to typically avoid those computations, postponing them as much as possible until they are absolutely necessary. As an example, in the "Tangent lines to a circle" implementation in listing 3.15, we first compare squared distances before inevitably requiring their absolute value.

Be that as it may, due to having fixed a kernel with inexact constructions for CGAL, we are bound with some robustness loss regardless. At this point, it can only be mitigated by also ensuring the quality of the inputs given to our primitives, and that is reliant on the user. If the user decides to provide "garbage" inputs, they will most likely be met with "garbage" outputs, a concern, however, that falls out of the scope of this work.

[15] There are other projects that attempt to make CGAL available in other language that resort to this same trick. See https://github.com/scikit-geometry/scikit-geometry and https://github.com/CGAL/cgal-swig-bindings.
[16] https://github.com/JuliaGeometry

4

Evaluation

Contents

In this chapter, we evaluate our solution by measuring the qualities of the approach we took to tackle GCS.

Firstly, we aim to benchmark our solution's performance by comparing it to a similar project called ConstraintGM [78]. We were able to reproduce the project's original benchmark tests and create an analogous test suite for our solution so we can compare both projects.

Secondly, we showcase four different case studies inspired by existing designs. This section of the evaluation focuses on comparing different approaches to solving GC problems present in each case study by adopting two different approaches: (1) an analytic approach, one programming naturally begs for, and (2) a constructive approach, essentially adding a conceptual abstraction layer over the former. We aim to show that, by following the latter approach, resulting programs become both easier to understand and reproducible as the set of instructions can be used to recreate the resulting geometry by hand, using a ruler and a compass.

Finally, as an added bonus, we explore our approach's potential regarding repurposing more complex geometric algorithms, contrasting it with re-implementing a version of said algorithms from scratch. Specifically, we set out to repurpose CGAL's 2D Delaunay Triangulation and Voronoi Diagram algorithms and further compare the resulting mapping's performance and correctness when compared to a native Julia implementation of the algorithms as provided by the package `VoronoiDelaunay.jl`[1], that, in the past, was once benchmarked against CGAL[2]. Additionally, we set out to estimate the effort it took to develop `VoronoiDelaunay.jl` and compare that to the effort it took to extract an analogous algorithm from CGAL.

Benchmarks were performed on a Lenovo® ThinkPad® E595 laptop computer with the following system specifications:

- AMD Ryzen™ 5 3500U CPU @ 2.1GHz[3];
- 1×16GB SO-DIMM of DDR4 RAM @ 2400MT/s.
- Arch Linux™[4] x86 64-bit, Linux® Kernel 5.12.15-zen1[5]

[1] https://github.com/JuliaGeometry/VoronoiDelaunay.jl
[2] https://gist.github.com/skariel/3d2018f9341a058e00fc
[3] Base clock frequency. Can boost up to 3.7GHz.
[4] https://archlinux.org
[5] https://github.com/zen-kernel/zen-kernel/tree/v5.12.15-zen1

4.1 ConstraintGM

ConstraintGM is a domain-specific language developed with the shared goal of tackling GC problem specification using a TPL (Racket, to be precise). This solution heavily and blindly relied on Maxima [79], a generic Computer Algebra System (CAS) to solve GC problems. Geometric entities were serialized into their algebraic equation representation coupled with the problem's constraints and sent to Maxima. Maxima would then attempt to solve the system of equations and return a result that was parsed and sent back to Racket.

This approach came at a severe performance cost, the reason being two-fold: (1) the communication between ConstraintGM and Maxima was slow, and (2) Maxima is a *generic* solver and could not take advantage of the geometric characteristics of the problem at hand. The considerable performance penalty of this approach is hard to justify in the case of simple geometric problems for which there are well-known efficient solutions. This lead to an *impromptu* implementation of some GC problem solutions, creating the Geometry Functions Library (GFL). As expected, the latter approach revealed that contextually specialized solutions had much better performance than relying on a generic solver.

It is worth noting that relying on Maxima meant ConstraintGM was intrinsically exact and robust since symbolic GCS methods, used by Maxima, automatically provide those features. The GFL, when compared to the Maxima-based approach, is more fragile in that regard because it will depend on the underlying constructs' number type. If an exact arbitrary precision number type is used, exactness and robustness will be preserved, but at a performance cost. Relying on inexact number types can eventually lead to erroneous results, but, by contrast, offers better performance. In the end, it is a matter of making a compromise and evaluating which fits the case at end the best.

The project's benchmark involve three different GC problems focused around object intersection, namely (1) line-line intersection, (2) circle-line intersection, and (3) circle-circle intersection. These problems expanded into thirteen different scenarios that consisted of rearranging the geometric entities' disposition in order to evaluate the intersection operation's results and measure each individual scenario's performance.

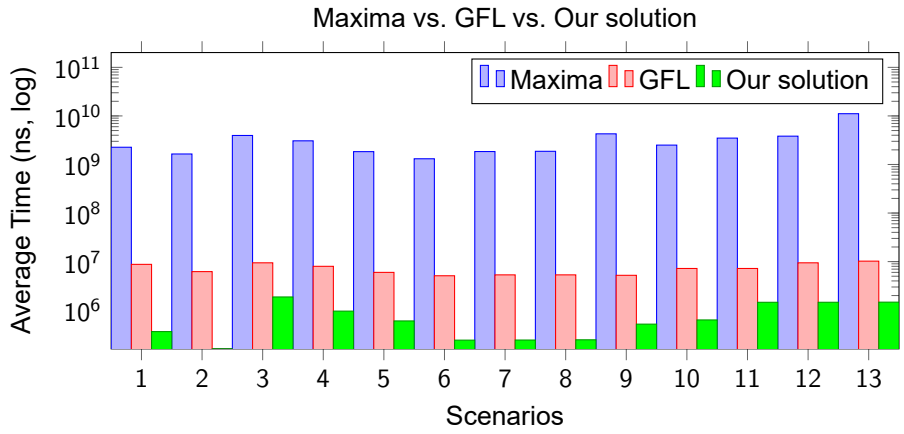
We measured real execution time instead of CPU time both for ConstraintGM and for our solution. We ran ConstraintGM's benchmarks a total of nine times to obtain a relatively decent sample of results, while our solution's benchmarks were aided by `BenchmarkTools.jl` [80], a package that considerably facilitates benchmarking Julia code. The source code used to benchmark both ConstraintGM and our solution is listed in listings A.1 and A.2 respectively. The benchmark's results are gathered in table 4.1. To facilitate comprehension, these are also plotted in fig. 4.1.

Observing the results, we can see the disparity between the approach reliant on Maxima when compared to both the GFL and our solution, which was to be expected. Even so, amidst relatively consistent results, scenario 13 made Maxima slug more than usual. That scenario consists of two circles inter-

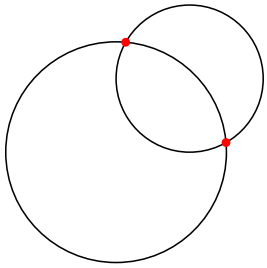
Table 4.1: Performance comparison between ConstraintGM's solutions, both using Maxima and GFL, and our solution.

Scenario	Execution time (mean ± σ)			GC problem
	Maxima	GFL	Our solution	
1	2.265 s ± 142.344 ms	8.778 ms ± 1.641 ms	362.191 µs ± 4.841 ms	Line ∩ Line
2	1.645 s ± 197.824 ms	6.222 ms ± 440.959 µs	159.567 µs ± 4.634 µs	
3	3.958 s ± 295.673 ms	9.444 ms ± 1.333 ms	1.871 ms ± 4.892 ms	Circle ∩ Line
4	3.070 s ± 243.768 ms	8.000 ms ± 0.000 ns	955.815 µs ± 3.710 ms	
5	1.839 s ± 84.460 ms	6.000 ms ± 0.000 ns	600.386 µs ± 19.807 µs	
6	1.311 s ± 59.139 ms	5.111 ms ± 333.333 µs	241.531 µs ± 12.504 µs	Circle ∩ Circle
7	1.847 s ± 50.466 ms	5.333 ms ± 500.000 µs	242.780 µs ± 5.692 µs	
8	1.868 s ± 37.650 ms	5.333 ms ± 500.000 µs	245.425 µs ± 11.840 µs	
9	4.277 s ± 37.053 ms	5.222 ms ± 440.959 µs	515.829 µs ± 4.881 ms	
10	2.506 s ± 238.696 ms	7.222 ms ± 440.959 µs	629.341 µs ± 5.360 ms	
11	3.493 s ± 258.715 ms	7.222 ms ± 440.959 µs	1.453 ms ± 5.070 ms	
12	3.830 s ± 150.402 ms	9.444 ms ± 527.046 µs	1.455 ms ± 5.088 ms	
13	11.111 s ± 81.302 ms	10.222 ms ± 1.093 ms	1.463 ms ± 5.055 ms	

secting at two different points, illustrated in fig. 4.1b. It is not the only scenario of the set that involves two circles that intersect. However, this is the one that produces relatively more complex results, which could justify why Maxima took relatively longer to compute the solution than it did for the scenarios that preceded this one.



(a) Y-bar plot of every approaches' benchmark results. N.B.: The Y-axis is logarithmic.



(b) Scenario 13 from the benchmark suite.

Figure 4.1: ConstraintGM benchmark results collected from table 4.1 in a Y-bar plot (a) beside the depiction of scenario 13 (b) from the benchmark suite.

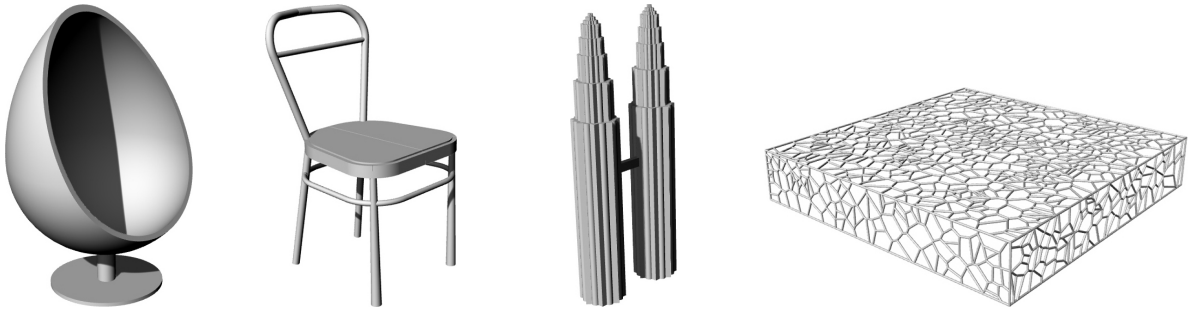
Analyzing the results further, we can see our solution also outdoes ConstraintGM's GFL by a significant margin. This is most likely due to the fact that we are relying on CGAL, a library implemented in C++. The latter is notoriously known for being a high-performance language, considerably outperforming Racket in a series of benchmarks. Nevertheless, despite some overhead in the case of Julia, the results are still positive.

In conclusion, our solution proves capable and performant, having surpassed ConstraintGM's GFL

by an entire order of magnitude on average.

4.2 Case Studies

In this section, we aim to demonstrate our solution when applied to four different case studies, each presenting a parametric geometric shape: an egg, a rounded trapezoid, a star with semicircles, and a Voronoi diagram. Each case, illustrated in fig. 4.2, was inspired by an existing design: (1) Eero Aarnio's Egg chair, (2) Thonet 214 chair seat, (3) César Pelli's Petronas tower section, and (4) PTW Architects' Beijing National Aquatics Center. These cases present a set of GC problems involving circles and lines, such as *tangent line to two circles* and *circle-line intersection*. These problems were solved employing both an *analytic* approach, an approach TPLs naturally demand, and a *constructive* approach, the one made possible by relying on our solution.



(a) Egg chair. **(b)** Simple 4-leg chair. **(c)** Islamic towers. **(d)** Voronoi cage.

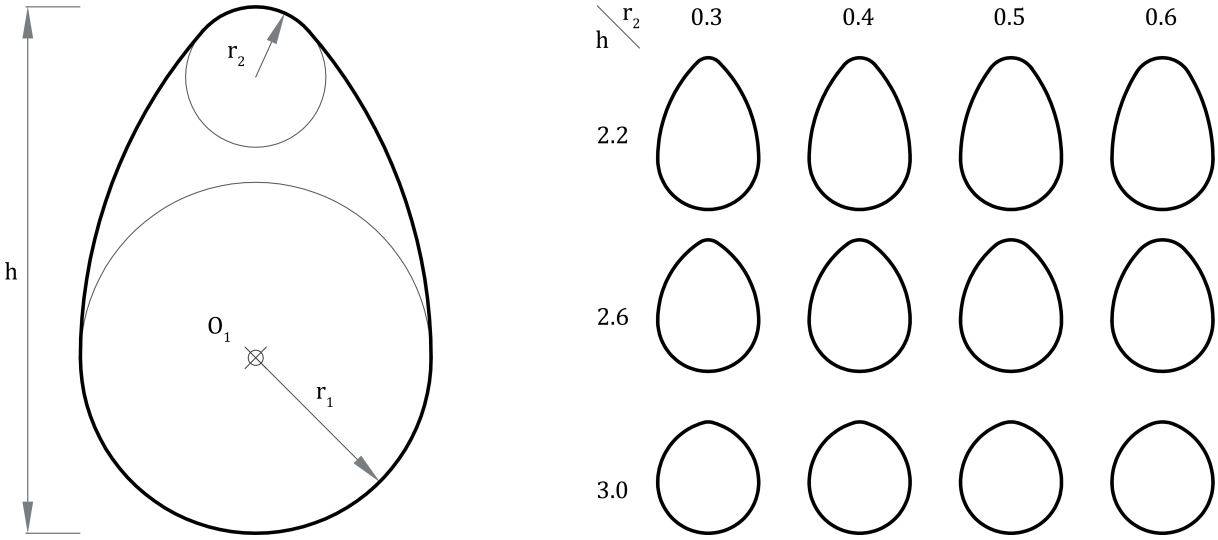
Figure 4.2: Case study designs inspired by the Eero Aarnio's Egg chair (a), Thonet 214 chair (b), César Pelli's Petronas Twin Towers (c), and PTW Architects' Beijing National Aquatics Center (d).

4.2.1 Egg

The first case study is an egg shape (which can be considered a particular case of an oval shape). Examples of applications of this shape in design include the Eero Aarnio's Egg chair, James Law's Cybertecture Egg building, and RAU Architects and Ro&Ad Architects' Tij observatory.

The egg shape is a broad concept, as it can refer to a wide range of curves resembling an egg [81]. In this case, our shape is bilaterally symmetric and is composed of four arcs, the side arcs being tangent to the bottom and top arcs.

Our parametrization of the egg defines four parameters: the bottom arc's center O_1, the bottom and top arcs' radii r_1 and r_2 respectively, and the egg's height h (fig. 4.3a). Different egg shapes, more or less elongated, can be obtained by varying the parameters' values (fig. 4.3b). One of those variations, where $r_2 = (2 - \sqrt{2})r_1$ and $h = 2r_1 + r_2$, corresponds to the Moss' Egg (fig. 4.3, the egg with $h = 2.6$ and $r_2 = 0.6$).



(a) Egg parametrization: center O_1, bottom and top arcs' radii r_1 and r_2, and height h.

(b) Egg shape variations: variations change the values of the radius r_3 and the height h.

Figure 4.3: Egg problem: (a) shows our parametrization of the egg which can be used to generate shape variations, some of them shown in (b).

In this case, the major problem was to define the side arc A_3, which is given by the center O_3, radius r_3, and amplitude α (fig. 4.4a). The *analytic solution* is described below. The angle α and the radius r_3 are devised from the triangle $\triangle O_1O_2O_3$ (fig. 4.4b), and the center O_3 is given by a translation of O_1 through the unit vector $\vec{e}_x$, parallel to the X-axis, scaled by the factor $r_1 - r_3$. It is important to note that, despite their simplicity, the derivations needed to arrive at the following formulas are not obvious.

1. $\alpha = 2 \arctan \frac{r_1 - r_2}{h - r_1 - r_2}$
2. $r_3 = \frac{r_1 - r_2 \cos \alpha}{1 - \cos \alpha}$
3. $O_3 = O_1 + (r_1 - r_2)\, \vec{e}_x$

The *constructive solution* is based on the geometric problem of determining the *tangent line to two circles*. In fact, the side arc A_3 is tangent to two circles, C_1 and C_2, and passes through point P_1. Two GC primitives from our solution were employed in this solution, namely *intersection* and *bisector*. The solution can be computed according to the following procedure (fig. 4.4b):

1. $C_2' = \text{circle}\,(P_1, r_2)$
2. $I = \text{intersection}\left(C_2', \overline{O_1P_2}\right)$
3. $B = \text{bisector}\left(\overline{IO_2}\right)$
4. $O_3 = \text{intersection}\,(B, O_1P_1)$

5. $r_3 = \text{length}\left(\overline{O_3 P_1}\right)$

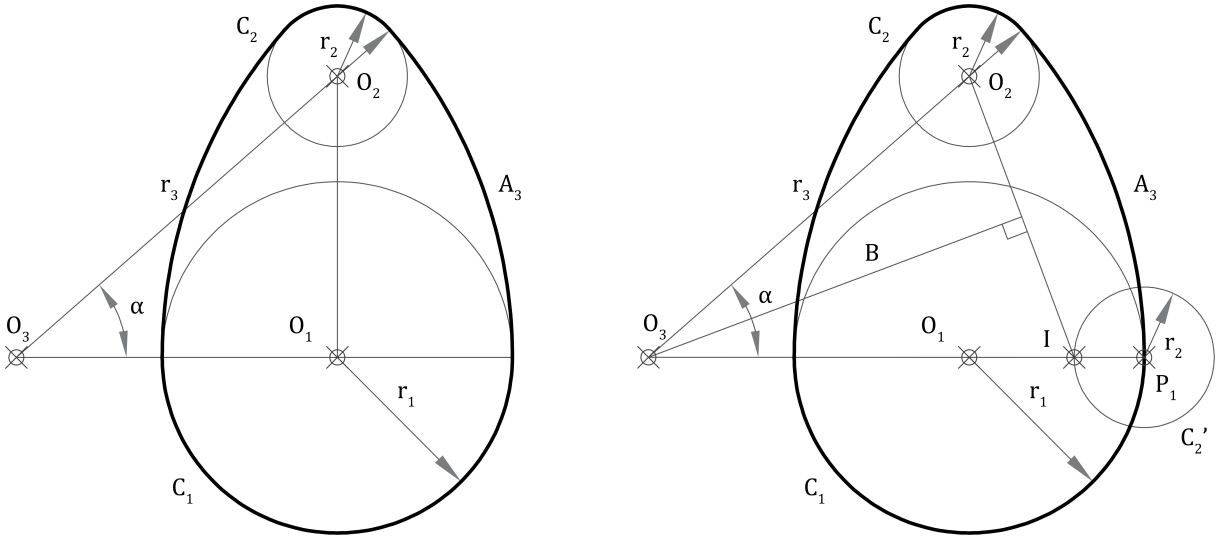
6. $\alpha = \text{angle}\left(O_2, O_3, O_1\right)$

We can further increase the expressive level of the solution by defining the $\text{tangent}_{\text{circle}}$ functionality. Given two circles, C_1 and C_2, and a point P_1 on C_1, $\text{tangent}_{\text{circle}}$ produces the circle tangent to both C_1 and C_2 that goes through P_1. This way, the solution can be simplified by replacing steps 1 through 5 by using the $\text{tangent}_{\text{circle}}$ functionality as follows:

1. $O_3, r_3 = \text{tangent}_{\text{circles}}\left(C_1, C_2, P_2\right)$

2. $\alpha = \text{angle}\left(O_2, O_3, O_1\right)$

In the egg's case, C_1 must be larger than C_2, and P_1 is one of the intersection points between the horizontal line that passes through O_1 and the circle C_1.



(a) Solution using the analytic approach.

(b) Solution using the constructive approach.

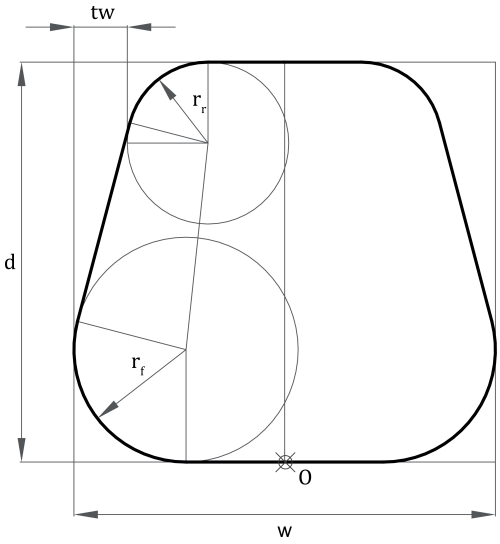
Figure 4.4: Solutions to the Egg problem using two different approaches.

Achieving the equations in the *analytic solution* is not a straightforward task for designers, since it involves considerable logical reasoning and the use of non-trivial formulas (such as trigonometric half-angle identities). It is also unclear how those equations were derived. By contrast, in the *constructive solution*, all the steps are clearly externalized, which makes it much more comprehensible.

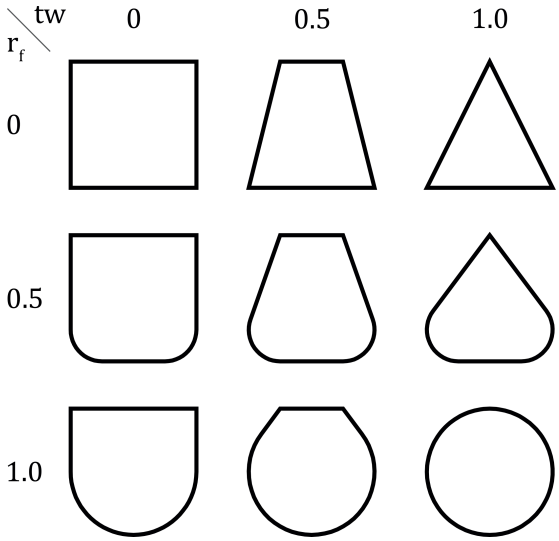
4.2.2 Rounded Trapezoid

The second case study is a rounded trapezoid shape. This parametric shapes is used in the contour of a variety of different chair seats, such as the Thonet 214 and the Zig Zag chairs [82]. This shape is

bilaterally symmetric and is defined by six parameters: width w, depth d, taper width tw, front radius r_f, rear radius r_r, and origin O (fig. 4.5a). The taper width and front and rear radii are ratios of the width and the depth. By varying these parameters' values, one can obtain different shapes, such as a square, a trapezoid, a triangle, a rounded rectangle, a drop-like shape, and a circle, among others (fig. 4.5b).



(a) Trapezoid parametrization: width w, depth d, taper width w, front and rear radii r_f and r_r respectively, and origin O.



(b) Rounded trapezoid shape variations: variations change the values of the front radius r_f and the taper width tw.

Figure 4.5: Rounded trapezoid problem: (a) shows our parametrization of the trapezoid which can be used to generate shape variations, some of them shown in (b).

One side of the chair is obtained by two circles and the *tangent line to two circles*, while the other side is a reflection of the former side. Given two circles C_2 and C_2, centered on O_1 and O_2 with radii r_1 and r_2 respectively (fig. 4.6), two solutions can be delineated that can accurately reproduce the tangent line $\overline{T_1T_2}$.

The *analytic solution* is achieved by calculating the angle δ between the line segment $\overline{O_1O_2}$ and the line perpendicular to the tangent line between the two circles passing through O_1. The angle θ is given by the slope of $\overline{O_1O_2}$. The point T_1 is given by a translation of O_1 through a vector $(r_1, \angle\delta + \theta)$ where r_1 is its length and $\angle\delta + \theta$ is its polar angle. A similar approach is used to obtain T_2.

1. $\delta = \arccos \frac{r_1 - r_2}{\|O_2 - O_1\|}$
2. $\theta = \angle\overrightarrow{O_2 - O_1}, \vec{e}_x$
3. $T_2 = O_1 + (r_1, \angle\delta + \theta)$
4. $T_2 = O_2 + (r_1, \angle\delta + \theta)$

The *constructive solution* follows a sequence of steps that reflects the geometric characteristics of

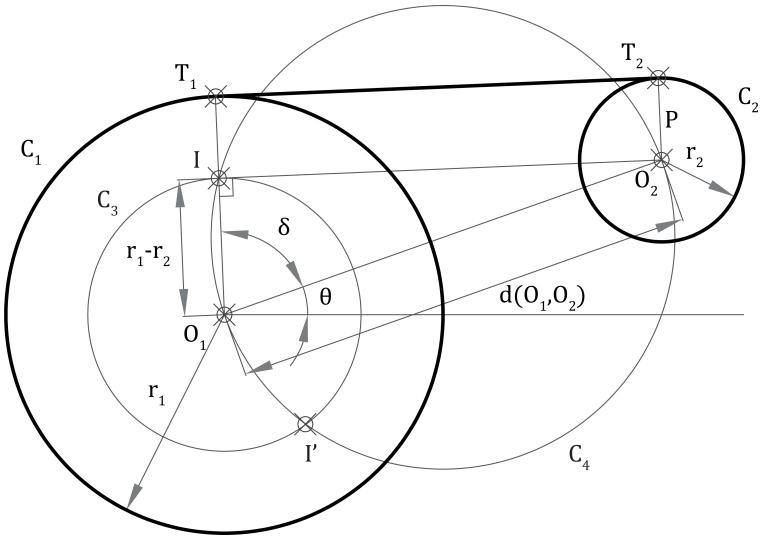


Figure 4.6: Both analytic and constructive solutions to the rounded trapezoid problem.

the problem (fig. 4.6). Four GC primitives were employed in this solution, namely *midpoint*, *intersection*, and *parallel lines*.

1. $C_3 = \text{circle}\,(O_1, r_1 - r_2)$
2. $M = \text{midpoint}\,(O_1, O_2)$
3. $d = \text{length}\,\left(\overline{O_1 O_2}\right)$
4. $C_4 = \text{circle}\,\left(M, \frac{d}{2}\right)$
5. $I, I' = \text{intersection}\,(C_3, C_4)$
6. $P = \text{parallel}\,\left(\overline{IO_1, O_2}\right)$
7. $T_1 = \text{intersection}\,(IO_1, C_1)$
8. $T_2 = \text{intersection}\,(P, C_2)$

Despite the adequacy of the *analytic solution* for the design problem at hand, it only produces the external tangent needed to solve this specific problem, which works well if the circle's center is on the 1st and 2nd quadrants: however, in the remaining quadrants, it produces an unintended tangent. In contrast, our *constructive approach* is capable of producing a solution independently of how the circles are arranged.

The advantage of using well-established functions from CGAL is that they deal with degenerate cases better than re-implementations of the same functionality from scratch. For instance, in the case of concentric circles, the intersection between those circles does not produce an error; instead, it produces an

empty result. In the case of the proposed *analytic solution*, the distance between the circles' centers is zero, which leads to a division-by-zero error.

Architects and designers are used to manipulating GCs, such as *tangency*, *parallelism*, and *intersection*, and less inclined to deal with the mathematical intricacies of analytic geometry, which makes the second approach more appealing to them. To that end, we can introduce the $\mathrm{tangent}_{\mathrm{lines}}$ functionality, which, given two circles C_1 and C_2, produces a sequence of tangent lines between the circles, allowing the user to select the lines that best suit the problem. This allows us to reduce the solution to the single step

$$\overline{T_1T_2}, \ldots = \mathrm{tangent}_{\mathrm{lines}}\,(C_1, C_2)$$

where $\overline{T_1T_2}$ is one of the lines of the sequence.

Depending on how the circles are arranged, there can be multiple solutions: (1) no segments, if one circle contains the other, (2) two segments, if the circles intersect, or (3) four segments, if the circles are disjoint. The advantage of having the $\mathrm{tangent}_{\mathrm{lines}}$ functionality is that it is capable of finding every solution to the generic *tangent line to two circles* problem. Hence, the user can reutilize this functionality each time a different problem of a similar nature arises.

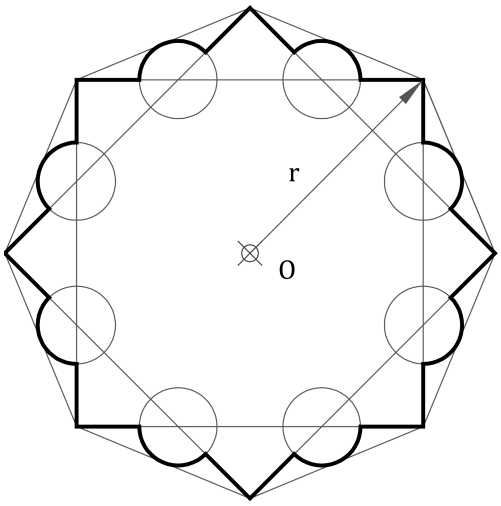
4.2.3 Star with Semicircles

The third case study is a star shape with semicircles. This shape was inspired by César Pelli's Petronas tower floor plan, which in turn mimics Islamic patterns. The contour of the Petronas tower floor plan is formed by two overlapping congruent squares, forming an octagram known as Star of Lakshmi, and by eight circles centered on each of the eight intersection points and tangent to the bounding octagon. This shape can be generalized to a parametric shape defined by three parameters: origin O, radius r, and number of vertices n (fig. 4.7a). Note that the number of vertices cannot be less than five. Five variations, comprising stars with 5 to 9 vertices, are illustrated in fig. 4.7.

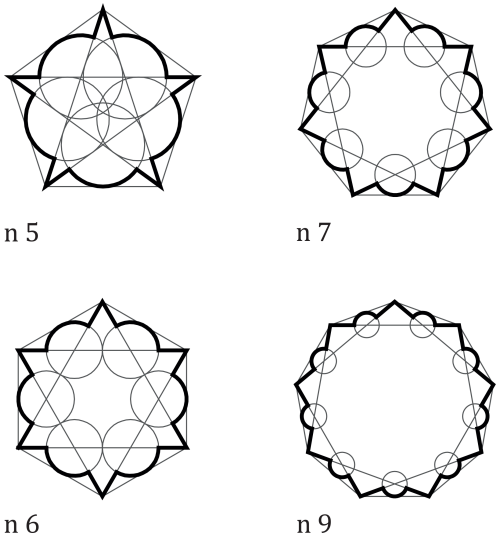
Both analytic and constructive solutions are based on computing one side of the star, composed of the line segment $\overline{V_1I_1}$, the arc centered on O_1 from I_1 to I_2, with radius r_1, and the line segment $\overline{I_2V_2}$ (fig. 4.8). The remaining sides result from the recursive application of the preceding side with a rotation transformation around the center.

The *analytic solution* is described below. The radius r_1 of the circle C_1 is calculated by step 1, the radius r_2 is defined by step 2, and its center O_1 is given by step 3. The intersection points I_1 and I_2 are given by a rotation around O_1 (fig. 4.8).

1. $r_1 = r\frac{\sin^2\frac{\pi}{n}}{\cos\frac{\pi}{n}}$
2. $r_2 = r\cos\frac{\pi}{n} - r_1$



(a) Star parametrization: origin O, radius r, and number of vertices n.



(b) Star shape variations: variations change the values of the number of vertices n.

Figure 4.7: Star with semicircles problem: (a) shows our parametrization of the star which can be used to generate shape variations, some of them shown in (b).

3. $O_1 = O + \left(r_2, \angle\frac{\pi}{n}\right)$
4. $I_1 = O_1 + \left(r_1, \angle\frac{2\pi}{n} - \frac{\pi}{2}\right)$
5. $I_2 = O_1 + \left(r_1, \angle\frac{\pi}{2}\right)$

The *construction solution* is based on finding the position and size of the circle C_1 (see fig. 4.8). Two GC primitives were employed in this solution, namely *intersection*, and *tangent circle to one line*.

1. $O_1 = \text{intersection}\left(\overline{V_1V_3}, \overline{V_2V_n}\right)$
2. $C_1 = \text{tangent}_{\text{circle}}\left(O_1, \overline{V_1V_2}\right)$
3. $P, r_1 = C_1$
4. $I_1 = \text{intersection}\left(\overline{V_1V_3}, C_1\right)$
5. $I_2 = \text{intersection}\left(\overline{V_2V_n}, C_1\right)$

As seen in the other cases, the constructive solution is more understandable, as one can easily reproduce it step-by-step by hand, using a ruler and a compass.

4.2.4 Voronoi Diagram

Our fourth case study is that of Voronoi diagrams, which are used in a variety of design fields. For instance, several facade designs exhibit a Voronoi appearance, such as PTW Architects' Beijing National

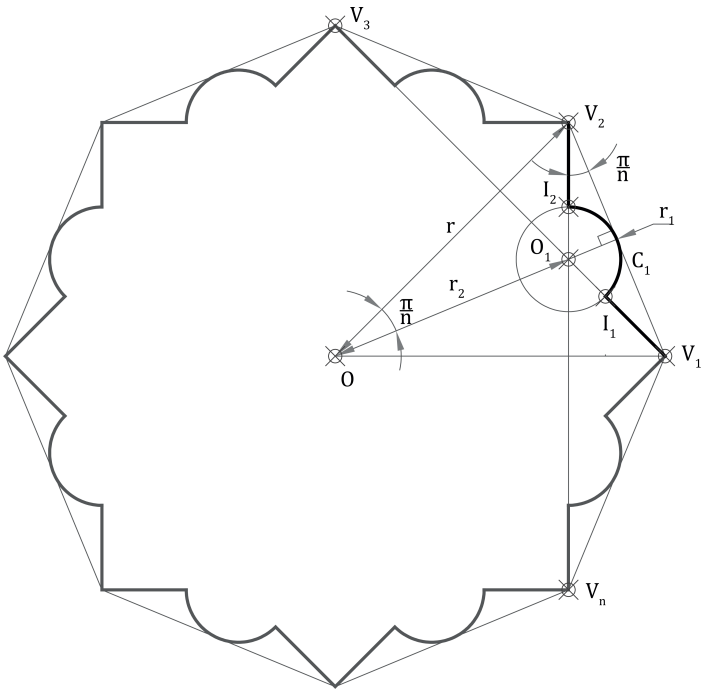


Figure 4.8: Both analytic and construction solutions to the star with semicircles problem.

Aquatics Center, ARM[6] Architecture's Melbourne Recital Centre, and Hassell's Alibaba Headquarters.

In its simplest version, a Voronoi diagram consists of partitioning a plane into regions from a set of points, called *sites*; for every site, each region contains every point in the plane closer to that site than to any other site. The sites are typically randomly distributed points, although they can follow other distributions. Figure 4.9 shows three Voronoi diagrams generated from entirely randomly distributed points (fig. 4.9a), from random points with one attractor point in the bottom-left corner (fig. 4.9b), and from random points with one attractor line at the bottom edge (fig. 4.9c). An attractor object is responsible for controlling the density of random points based on the distance to it.

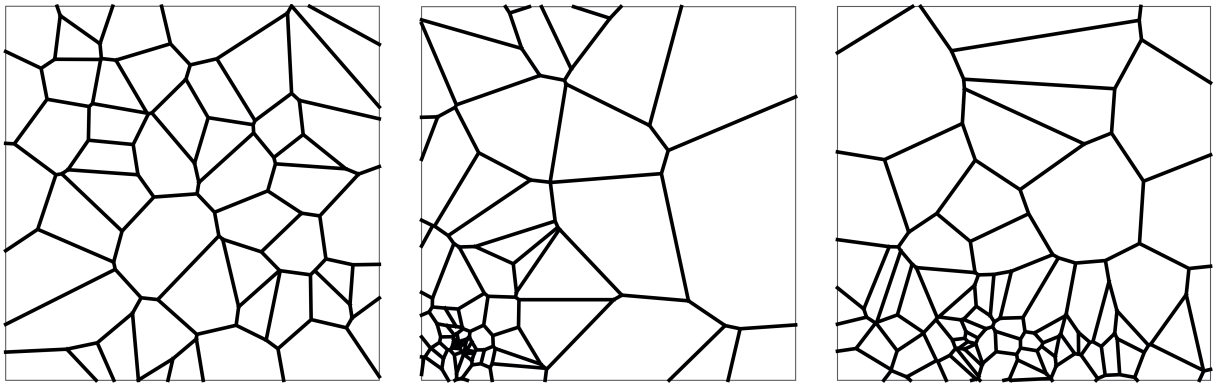
Both the analytic and constructive methods focus on computation of a vertex relies on the computation of the *circumcenter* of a triangle, for instance, triangle $\triangle P_1P_2P_3$ (fig. 4.10).

There are several possible *analytic solution* to compute a triangle's circumcenter. One of them is based on the *circumradius* formula (step 4), where a, b, and c are the lengths of the triangle's sides (step 1) and A is the triangle's area (step 3). The circumcenter C can then be easily computed by a translation (step 6) from P_1 following the angle α (step 5).

1. $a, b, c = \|P_2 - P_1\|, \|P_3 - P_1\|, \|P_3 - P_2\|$

2. $s = \frac{a+b+c}{2}$

[6] https://armarchitecture.com.au/projects/melbourne-recital-centre/. Not to be mistaken with the ARM family of computing architectures for computer processors.



(a) Randomly distributed points.

(b) Randomly distributed points with one attractor point in the bottom-left.

(c) Randomly distributed points with an attractor line at the bottom.

Figure 4.9: Voronoi diagram problem: depicted are three Voronoi diagrams variations which are mostly generated at random. (a) is entirely random, (b) expands on the latter with an attractor point, and (c) goes even further by using an entire attractor line.

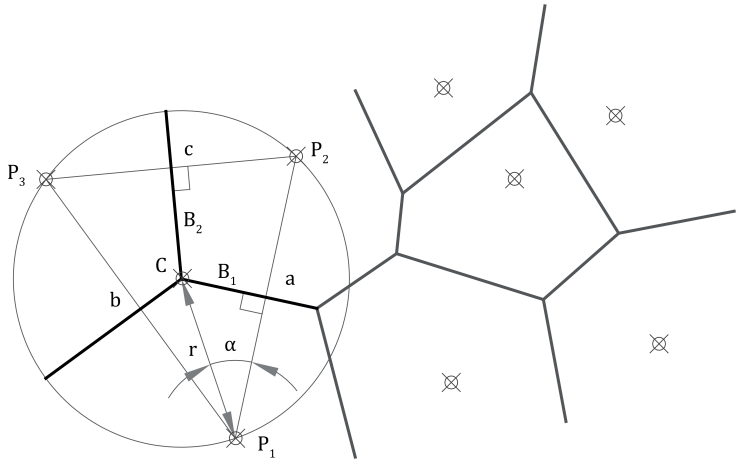


Figure 4.10: Analytic and constructive approaches to computing a Voronoi vertex via computing the circumcenter of every triangle in the Delaunay triangulation whose vertices are the Voronoi diagram's sites.

3. $A = \sqrt{s(s-a)(s-b)(s-c)}$
4. $r = \frac{abc}{4A}$
5. $\alpha = \arccos \frac{a}{2r}$
6. $C = P_1 + (r, \angle\alpha)$

The *constructive solution* computes the circumcenter, given by the intersection of the edges' perpendicular bisectors. In fact, this has been exemplified in chapter 1. Two GC primitives were employed in this solution, namely *bisector* and *intersection*.

1. $B_1 = \text{bisector}\left(\overline{P_1 P_2}\right)$

2. $B_2 = \text{bisector}\left(\overline{P_2P_3}\right)$

3. $C = \text{intersection}\left(B_1, B_2\right)$

We can increase the abstraction level of the solution by using the *circumcenter* functionality implemented in our solution, which, in reality, is implemented in CGAL:

$$C = \text{circumcenter}\left(P_1, P_2, P_3\right)$$

The circumcenter problem is only a small part of the generation of a Voronoi diagram, since we first need to build a Delaunay triangulation. We can then apply the circumcenter functionality to find the Voronoi vertices and draw the diagram's edges. This set of steps can be abstracted away in a functionality called *voronoi* that, given a set of points P_S, produces a 2D Euclidean Voronoi diagram:

$$V = \text{voronoi}\left(P_S\right)$$

Implementing this functionality from scratch is a demanding and error-prone task. Fortunately, CGAL already has an algorithm that produces robust 2D Euclidean Voronoi diagrams that can handle degenerate cases, such as dealing with three or more collinear points. This algorithm, much like the *circumcenter* functionality, was entirely repurposed, and made available in `CGAL.jl`, and, thus, it is also available in our solution. The final section of the evaluation goes over how we can repurpose this algorithm as a side effect of integrating such a comprehensive library as CGAL.

4.3 Voronoi Diagrams Extended

In the previous section, we left the problem of Voronoi Diagrams partially unresolved, only tackling a small sub-problem required for computing the diagram's vertices. This section expands on it by showing how we can repurpose CGAL's version of the Voronoi Diagram algorithm [83] as a beneficial side effect of integrating with the library. Additionally, we compare said version with a native Julia implementation of the algorithm described in [84] available in the `VoronoiDelaunay.jl`[7] package. Measurements involved obtaining an estimate of the effort required to obtain either implementation, measuring Delaunay Triangulation construction performance, and subsequent output triangulation and diagram comparison.

Both implementations adopt a similar incremental approach and are both robust. The version from `VoronoiDelaunay.jl`, however, uses floating point filtering [84], requiring all point coordinates to be in the interval $(1, 2) \times (1, 2) \subset \mathbb{R}^2$; a restriction CGAL's implementation does not have since it uses a dynamic floating point precision approach[8] [85]. The *limitation* in the former arguably constitutes a minor inconvenience since the diagram may be produced in that limited range and then scaled afterwards to

[7] https://github.com/JuliaGeometry/VoronoiDelaunay.jl
[8] Briefed at https://www.cs.cmu.edu/~quake/robust.html

meet the use case's needs. But it is worth noting the issue is not present in CGAL's implementation. This is relevant if the input coordinates cannot be altered to meet those needs.

To illustrate how we obtained the 2D Delaunay Triangulation and Voronoi Diagram algorithms from CGAL, listing B.1 shows a minimal working example of the wrapper library code supported by JlCxx required to make the necessary constructs and functionality available in Julia. It follows a familiar approach, the same used to obtain the core constructs supporting our *Geometric Constraint Primitives* solution. Listing 4.1 shows a bare-bones Julia module encapsulating and exposing the mapped functionality to the Julia language. `CGAL.jl` contains a version of these mappings that are, however, parameterized, and

```
module Voronoi

using CxxWrap
export Point2, Segment2, DelaunayTriangulation2, VoronoiDiagram2,
       x, y, point, source, target, segment, edges

@wrapmodule joinpath(@__DIR__, "libvoronoi")
__init__() = @initcxx

@cxxdereference Base.insert!(dt::DelaunayTriangulation2, ps::AbstractArray) =
    insert!(dt, StdVector(ps))

end # Voronoi
```

Listing 4.1: Bare-bones Julia module wrapping CGAL's 2D Delaunay Triangulation and Voronoi Diagrams, supported by the JlCxx wrapper in listing B.1.

adds methods for querying, insertion, and further manipulation. For the purposes of testing, we used the slimmer `Voronoi` module (listing 4.1).

The process for obtaining these code bindings is a relatively simple one that requires bare minimal C++ knowledge and following reference documentation as if it were a recipe book, looking up the necessary ingredients and adding them to the mix. At most, accounting for trial and error in case a minute detail was overlooked, to make this functionality available, it would take no more than a full day. Having that, the algorithms are ready and available for use.

The algorithm present in `VoronoiDelaunay.jl` is the result of an immense body of research, detailed in the AREPO paper [84]. Julia is an expressive language that greatly benefits prototyping. Changes are easy to apply and test without the overhead of compilation, a downside of prototyping in C++, especially when compiling (against) a sizeable codebase. We can only infer that the development of the algorithm did not take long to draft. However, we can certainly say that it took more than a full day to obtain a robust implementation, far exceeding the effort taken by just repurposing an existing robust implementation in CGAL. It required interpretation and understanding of the approach described in the article since there is no explicit algorithm listed. This then requires the creation of a conceptual mapping of the algorithm's

entities and procedures, later translating them into corresponding, preferably efficient, data structures and functions in the Julia language, which, in fairness, is an easy platform to prototype in. These are among other trials and tribulations that we did not have to meet in the slightest. Our estimate is that developing the algorithm in `VoronoiDelaunay.jl` took far longer than obtaining one via our approach, the latter concisely, generously, fitting into the span of a full day at most.

Regarding both algorithms' performance, table 4.2 shows a series of results obtained by batch inserting various powers-of-ten sets of points into Delaunay Triangulation objects, effectively building the meshes. The results are plotted in fig. 4.11 for a better understanding of the tabulated results. The source code used to perform these tests is listed in listing B.3. More detailed data can be seen in listing B.4.

Table 4.2: Execution times for the construction of Delaunay Triangulations, comparing implementations of the algorithm: one from CGAL and one from `VoronoiDelaunay.jl`.

# Points	Execution time (mean ± σ)	
	`Voronoi.jl`	`VoronoiDelaunay.jl`
10^2	42.145 µs ± 5.638 µs	53.799 µs ± 39.811 µs
10^3	509.757 µs ± 1.258 ms	510.851 µs ± 113.523 µs
10^4	5.879 ms ± 3.874 ms	5.606 ms ± 382.603 µs
10^5	62.344 ms ± 11.980 ms	65.103 ms ± 2.287 ms
10^6	668.331 ms ± 44.458 ms	841.925 ms ± 51.394 ms

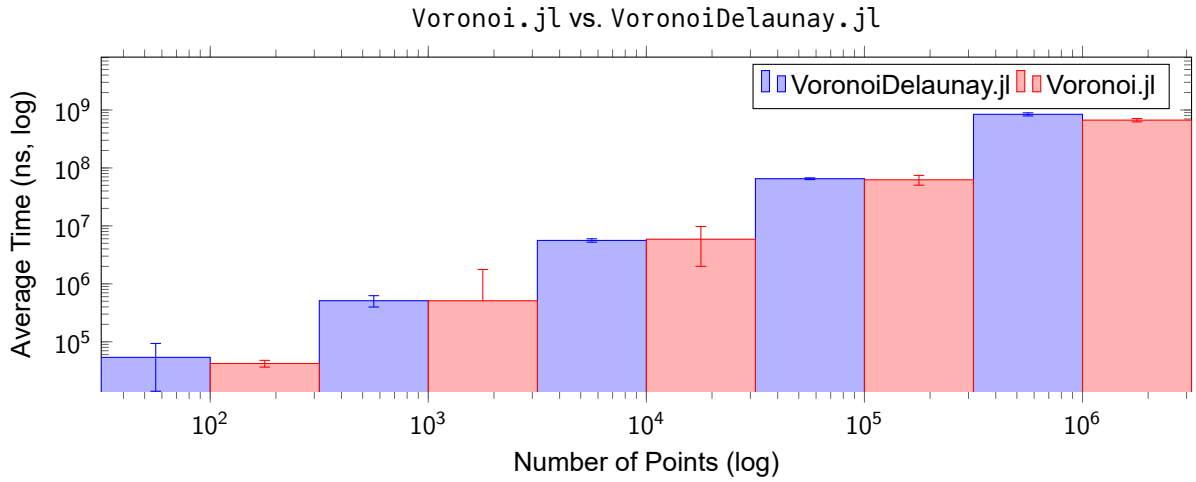


Figure 4.11: Delaunay Triangulation benchmark comparison results collected form table 4.2 in a Y-bar plot. Both axes follow a logarithmic scale.

The sets of points were randomly generated using an instance of the Mersenne Twister pseudo-random number generator, MT19937, with the seed `0xdeadbeef`. Tests for 10^n points used the same 10^n random points for every sample evaluated. Results are pretty identical, both variants trading blows. However, more often than not, we see CGAL's variant of the algorithm beating `VoronoiDelaunay.jl`'s by a relatively small margin, except the last result's difference is more than marginal. We see some more

variation from CGAL's algorithm, but looking at more detailed data (listing B.4), such cases seem to be punctual at best, proving negligible in practice.

One other interesting detail we can see represented in additional data is that of estimated memory usage. The algorithm in `VoronoiDelaunay.jl` appears to use far more memory than CGAL's does, differing by two entire orders of magnitude. We did not investigate any further, though additional profiling could help bring light to some of these results.

Finally, we take a look at the output Delaunay Triangulations and their respective dual Voronoi Diagrams, produced by both implementations. Figure 4.12 illustrates two plots of overlapping meshes: on the left, the Delaunay Triangulations, and on the right, the respective dual Voronoi Diagrams. Listing B.2 shows the Julia code used to produce the plots in the figures.

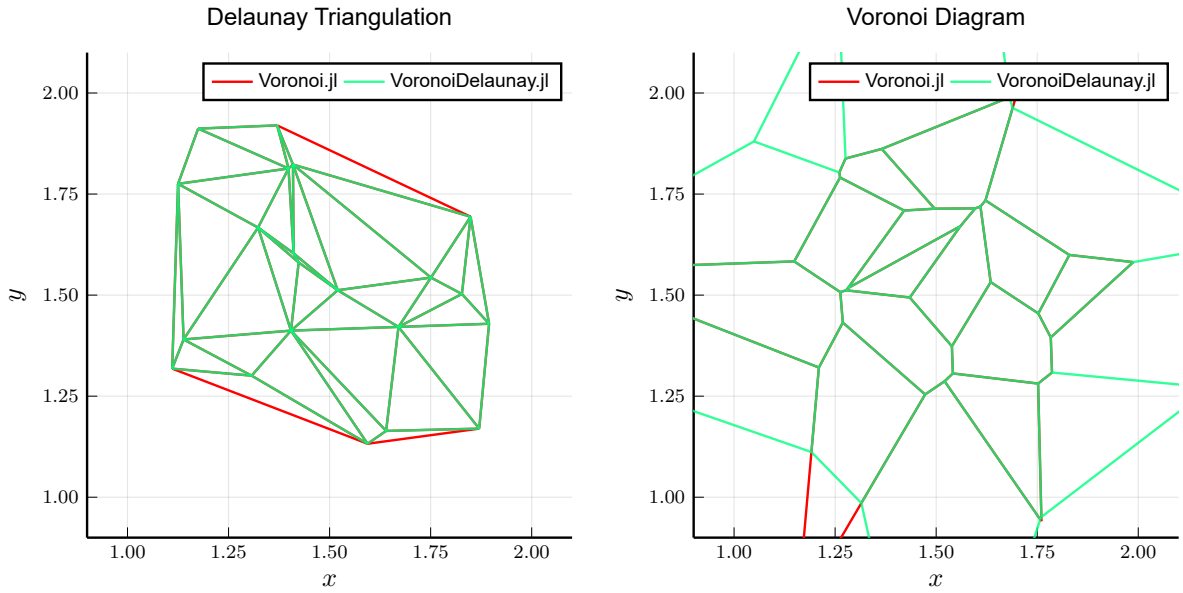


Figure 4.12: Delaunay Triangulations (on the left) and Voronoi Diagrams (on the right) produced both by `Voronoi.jl` and `VoronoiDelaunay.jl`. The mesh was produced from a 21-point cloud.

Surprisingly, the produced triangulations are not the same, which explains the divergence in the dependent Voronoi Diagrams. CGAL's implementation produces a few more edges, or perhaps it should be said the other way around: `VoronoiDelaunay.jl` does not produce every possible edge. Simpler point clouds illustrate this disparity further. If tested with a 3-point cloud, i.e., a triangle, CGAL will properly produce said triangle while `VoronoiDelaunay.jl` will not. Figure 4.13 shows this scenario blatantly. This example uses the same code in listing B.2 with a minor difference in the number of points changed from the default of 21 down to 3.

It is not clear which of the implementations is wrong. Though, given that CGAL is considered the *de facto* industry standard library for geometric computation, it is probably safe to assume there is something wrong with the implementation of the algorithm in `VoronoiDelaunay.jl`.

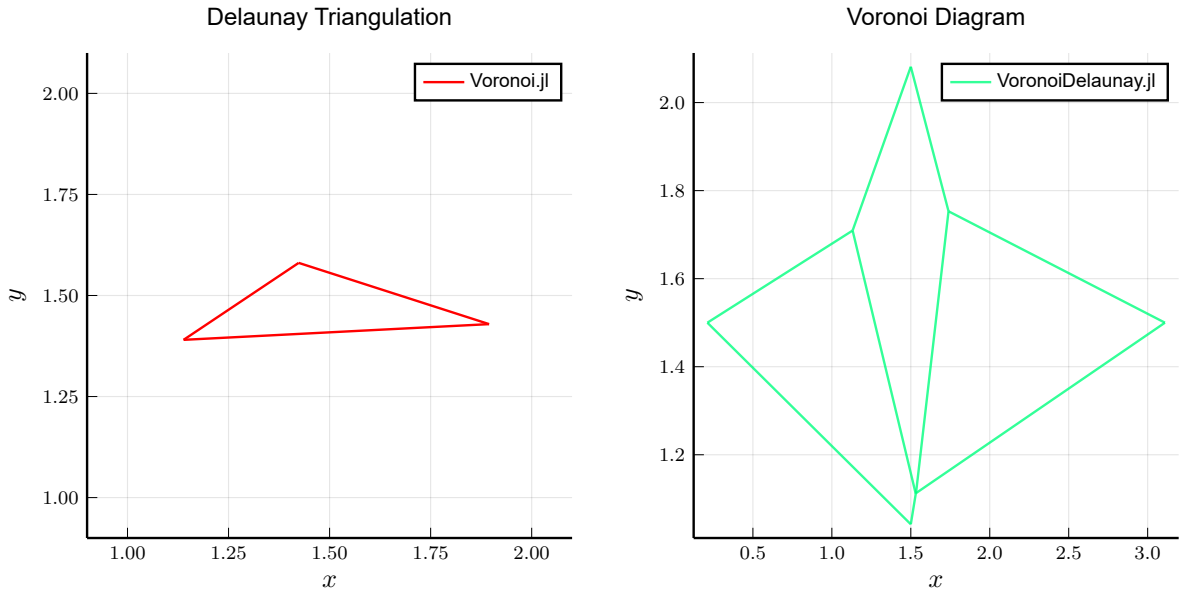


Figure 4.13: CGAL's algorithm produced a triangle (on the left) while `VoronoiDelaunay.jl` did not. Consequently, CGAL did not produce a (finite) Voronoi Diagram while `VoronoiDelaunay.jl` did (on the right).

Summarily, though we do not believe it to be entirely wrong to implement (especially complex) algorithms and functionality from scratch, we have seen how it can rapidly prove problematic, making it sometimes less preferable. From our benchmarks, there was no performance gain. As a matter of fact, the re-implementation was slower. The effort put into repurposing an already existing algorithm from what is considered to be a rather complex library is still far lower than the effort required to implement a solid, robust version of the algorithm from scratch. Differences between the outputs validate our argument here that new implementations will inevitably lead to an emergence of erroneous behavior, whereas more established and battle-tested software will probably have taken care of many of the issues that are still to become apparent.

5

Conclusion

The generation of highly constrained sophisticated designs is not viable through usage of interactive interfaces due to rigidity in the manipulation of existing models in order to generate multiple variants. Algorithmic Design (AD). Even then, Visual Programming Languages (VPLs) suffer from the disproportionate relation between the resulting workflow and respective design complexity. However, working with geometric constraints in Textual Programming Languages (TPLs) imposes a set of challenges, which can be overcome through the usage of Geometric Constraint Solving (GCS) approaches to solve complex systems of constraints. To achieve that goal, several methods can be employed, but they mostly resort to generic GCS algorithms. Alas, solvers, in general, have difficulties identifying specific underlying subproblems for which efficiently computable and robust solutions might be available.

Nonetheless, the prior analysis of the set of geometric constraints that must be dealt with requires certain background knowledge on numerical robustness to mitigate fixed-precision arithmetic issues, such as *roundoff* error accumulation throughout computation. Moreover, there is the added requirement of researching solutions to these specific constraint problems. The user will end up spending more time and effort in this process than in the design process itself.

Thus, in order to overcome these obstacles, an alternative approach is proposed in the form of the implementation of geometric constraint primitives in an expressive TPL supported by an exact geometric computation library. The latter provides a series of optimized geometric algorithms and exact data structures that allow transparent handling of robustness issues, lifting this concern from the user's shoulders with the goal of improving constrained geometry specification efficiency as well as consequently facilitating the design process.

However, the usage of exact data structures incurs a substantial performance hit that does not justify its pervasive use, only fitting very few scenarios in practice. We leveraged faster, albeit inexact, constructs, while still preserving exact predicate computation. Additionally, the implemented primitives still delay geometry construction as much as possible, remaining robust and preserving exactness up until that point. Misuse of resulting geometry might still lead to surprising erroneous situations, though in practical terms, these cases are few and far apart, meaning our solution still holds value.

Finally, we proved the approach employed by our solution is one that creates understandable programs that can be manually reproduced. By adopting a constructive approach to geometry specification, we externalize and clarify the steps required to build geometric objects. This is contrasted with the more natural analytical approach programming languages usually beg for. Following the latter approach is not only more cumbersome due to the solution derivation process, but it also produces incomprehensible programs, hiding the concrete geometry behind formulas. The former is preferred by AD practitioners, as well as industry professionals in general, but it proves alluring to novice users all the same. Said novice users might be starting to learn and adopt AD. With our work, we aim to bolster this adoption rate, driving more and more people from traditional means to novel design paradigms.

Future Work

Our solution certainly has some drawbacks and misfeatures that could be improved. Some were already discussed in section 3.2. To briefly reiterate a few, our wrapper around the underlying library is less transparent than desired. Constructs and functionality should be mapped as transparently as possible, fully parameterized, as to provide the user with more control and choice over the constructs they are using. Furthermore, the set of geometric primitives that were implemented was quite limited in size and could be further expanded on. However, let the ones showcased serve as an example for expanding the set of primitives even further.

This work focused exclusively on constrained geometry bound to the $\mathbb{R}^2$ Euclidean space, i.e., the 2D plane. There is still much work to be done researching problem solutions that encompass 3D space as well. Elevating a dimension means the solutions to problems once formulated in the 2D plane are no longer applicable in 3D space for some problems may now be under-constrained. As an example, our solution for the circumcenter problem, exemplified in section 1.4.2, would no longer work in 3D space. A line's bisector in 3D space is a plane, and noncoplanar nor parallel plane intersection results in a line. To obtain the actual circumcenter, one would additionally, for example, have to intersect the resulting line with the plane the circumscribed triangle sits on.

Bibliography

[1] B. Bettig and C. M. Hoffmann, “Geometric constraint solving in parametric CAD,” *Journal of Computing and Information Science in Engineering*, vol. 11, no. 2, p. 021001, Jun. 14, 2011. [Online]. Available: https://doi.org/10.1115/1.3593408

[2] J. McCormack, A. Dorin, and T. Innocent, “Generative design: A paradigm for design research,” in *Futureground — DRS International Conference 2004*, J. Redmond, D. Durling, and A. de Bono, Eds., Melbourne, Australia, 17–21 Nov. 2004. [Online]. Available: https://dl.designresearchsociety.org/drs-conference-papers/drs2004/researchpapers/171

[3] S. Garcia. (2012) ChairDNA. Accessed on 6 Jan 2019. [Online]. Available: https://chairdna.wordpress.com

[4] I. E. Sutherland, “Sketchpad: A man-machine graphical communication system,” *SIMULATION*, vol. 2, no. 5, pp. R–3–R–20, May 1, 1964. [Online]. Available: https://doi.org/10.1177/003754976400200514

[5] A. G. Requicha, “Representations for rigid solids: Theory, methods, and systems,” *ACM Computing Survey*, vol. 12, no. 4, pp. 437–464, Dec. 1980. [Online]. Available: http://doi.acm.org/10.1145/356827.356833

[6] A. A. G. Requicha and H. B. Voelcker, “Constructive solid geometry,” University of Rochester, Rochester, New York, Technical Report 25, Nov. 1977. [Online]. Available: http://hdl.handle.net/1802/26358

[7] J. D. Foley, F. D. Van, A. van Dam, S. K. Feiner, J. F. Hughes, E. Angel, and J. Hughes, *Computer Graphics: Principles and Practice*, ser. Addison-Wesley systems programming series, J. D. Foley, Ed. Addison-Wesley Professional, 1996, vol. 12110, accessed on 16 Jun 2021. [Online]. Available: https://books.google.pt/books?id=-4ngT05gmAQC

[8] I. Stroud, *Boundary Representation Modelling Techniques*, 1st ed. Springer, London, 2006. [Online]. Available: https://doi.org/10.1007/978-1-84628-616-2

[9] W. Jabi, *Parametric Design for Architecture*, 1st ed. Laurence King Publishing, London, Sep. 2013.

[10] J. Chung and M. Schussel, “Technical evaluation of variational and parametric design,” *Computers in Engineering*, vol. 1, pp. 289–298, 1990.

[11] J. C. Owen, “Algebraic solution for geometry from dimensional constraints,” in *Proceedings of the First ACM Symposium on Solid Modeling Foundations and CAD/CAM Applications*, ser. SMA ’91. Austin, Texas, USA: Association for Computing Machinery, New York, NY, USA, 1991, pp. 397–407. [Online]. Available: https://doi.org/10.1145/112515.112573

[12] W. Bouma, I. Fudos, C. M. Hoffmann, J. Cai, and R. Paige, “Geometric constraint solver,” *Computer-Aided Design*, vol. 27, no. 6, pp. 487–501, 1995. [Online]. Available: https://doi.org/10.1016/0010-4485(94)00013-4

[13] S. Samuel, “CAD package pumps up the parametrics,” *Machine Design*, vol. 78, no. 16, pp. 82–84, Aug. 24, 2006.

[14] N. Wu and H. T. Ilies, “Motion-based shape morphing of solid models,” in *Proceedings of the ASME 2007 International Design Engineering Technical Conferences and Computers in Engineering Conference*, ser. IDETC2007, vol. 6: 33rd Design Automation Conference, Parts A and B, Las Vegas, Nevada, USA, 4–7 Sep. 2007, pp. 525–535. [Online]. Available: https://doi.org/10.1115/DETC2007-34826

[15] C. Clarke, “Super models,” *Engineer*, vol. 284, pp. 36–38, 2009.

[16] B. Bettig, V. Bapat, and B. Bharadwaj, “Limitations of parametric operators for supporting systematic design,” in *Proceedings of the ASME 2005 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, vol. 5a: 17th International Conference on Design Theory and Methodology. Long Beach, California, USA: ASME, Sep. 2005, pp. 131–142. [Online]. Available: https://doi.org/10.1115/DETC2005-85165

[17] F. Rossi, P. van Beek, and T. Walsh, *Handbook of Constraint Programming*. Elsevier, Aug. 18, 2006.

[18] C. M. Hoffmann and R. Joan-Arinyo, “A brief on constraint solving,” *Computer-Aided Design and Applications*, vol. 2, no. 5, pp. 655–663, 2005. [Online]. Available: https://doi.org/10.1080/16864360.2005.10738330

[19] G. A. Kramer, “Solving geometric constraint systems,” *Association for the Advancement of Artificial Intelligence*, pp. 708–714, Jul. 1990. [Online]. Available: https://www.aaai.org/Library/AAAI/1990/aaai90-106.php

[20] C.-Y. Hsu and B. D. Brüderlin, *A Hybrid Constraint Solver using Exact and Iterative Geometric Constructions*. Springer, Berlin, Heidelberg, 1997, pp. 265–279. [Online]. Available: https://doi.org/10.1007/978-3-642-60718-9_19

[21] R. S. Latham and A. E. Middleditch, "Connectivity analysis: A tool for processing geometric constraints," *Computer-Aided Design*, vol. 28, no. 11, pp. 917–928, 1996. [Online]. Available: https://doi.org/10.1016/0010-4485(96)00023-1

[22] B. N. Freeman-Benson, J. Maloney, and A. Borning, "An incremental constraint solver," *Communications of the ACM*, vol. 33, no. 1, pp. 54–63, Jan. 1990. [Online]. Available: https://doi.org/10.1145/76372.77531

[23] R. C. Veltkamp and F. Arbab, "Geometric constraint propagation with quantum labels," in *Computer Graphics and Mathematics*, ser. Focus on Computer Graphics, B. Falcidieno, I. Herman, and C. Pienovi, Eds. Genoa, Italy: Springer, Berlin, Heidelberg, 1992, pp. 211–228. [Online]. Available: https://doi.org/10.1007/978-3-642-77586-4_14

[24] B. Aldefeld, "Variation of geometries based on a geometric-reasoning method," *Computer-Aided Design*, vol. 20, no. 3, pp. 117–126, 1988. [Online]. Available: https://doi.org/10.1016/0010-4485(88)90019-X

[25] W. Sohrt and B. D. Brüderlin, "Interaction with constraints in 3D modelling," in *Proceedings of the First ACM Symposium on Solid Modeling Foundations and CAD/CAM Applications*, ser. SMA '91. Austin, Texas, USA: Association for Computing Machinery, New York, NY, USA, May 1991, pp. 387–396. [Online]. Available: https://doi.org/10.1145/112515.112570

[26] B. Brüderlin, "Using geometric rewrite rules for solving geometric problems symbolically," *Theoretical Computer Science*, vol. 116, no. 2, pp. 291–303, 1993. [Online]. Available: https://doi.org/10.1016/0304-3975(93)90324-M

[27] G. Sunde, "A CAD system with declarative specification of shape," in *Intelligent CAD Systems I: Theoretical and Methodological Aspects*, ser. EurographicSeminars, Tutorials and Perspectives in Computer Graphics, P. J. W. ten Hagen and T. Tomiyama, Eds. Noordwijkerhout, Netherlands: Springer, Berlin, Heidelberg, 21–24 Apr. 1987, pp. 90–105. [Online]. Available: https://doi.org/10.1007/978-3-642-72945-4_6

[28] A. Verroust, F. Schonek, and D. Roller, "Rule-oriented method for parameterized computer-aided design," *Computer-Aided Design*, vol. 24, no. 10, pp. 531–540, Oct. 1992. [Online]. Available: https://doi.org/10.1016/0010-4485(92)90040-H

[29] S.-C. Chou, “An Introduction to Wu’s Method for Mechanical Theorem Proving in Geometry,” *Journal of Automated Reasoning*, vol. 4, no. 3, pp. 237–267, Sep. 1988. [Online]. Available: https://doi.org/10.1007/BF00244942

[30] B. Buchberger, “Gröbner bases: An algorithmic method in polynomial ideal theory,” *Multidimensional Systems Theory and Applications*, pp. 89–127, 1995. [Online]. Available: https://doi.org/10.1007/978-94-017-0275-1_4

[31] S. A. Buchanan and A. de Pennington, “Constraint definition system: A computer-algebra based approach to solving geometric-constraint problems,” *Computer-Aided Design*, vol. 25, no. 12, pp. 741–750, 1993. [Online]. Available: https://doi.org/10.1016/0010-4485(93)90101-S

[32] K. Kondo, “Algebraic method for manipulation of dimensional relationships in geometric models,” *Computer-Aided Design*, vol. 24, no. 3, pp. 141–147, 1992. [Online]. Available: https://doi.org/10.1016/0010-4485(92)90033-7

[33] C. B. Durand, “Symbolic and numerical techniques for constraint solving,” Ph.D. dissertation, Purdue University, 1998.

[34] R. C. Hillyard and I. C. Braid, “Characterizing non-ideal shapes in terms of dimensions and tolerances,” *SIGGRAPH Computer Graphics*, vol. 12, no. 3, pp. 234–238, Aug. 1978. [Online]. Available: https://doi.org/10.1145/965139.807396

[35] A. Borning, “The programming language aspects of ThingLab, a constraint-oriented simulation laboratory,” in *Readings in Artificial Intelligence and Databases*. San Francisco, California, USA: Morgan Kaufmann, 1989, pp. 480–496. [Online]. Available: https://doi.org/10.1016/B978-0-934613-53-8.50036-4

[36] J. P. Dedieu and M. Shub, “Newton’s method for overdetermined systems of equations,” *Mathematics of Computation*, vol. 69, no. 231, pp. 1099–1115, 2000. [Online]. Available: https://doi.org/10.1090/S0025-5718-99-01115-1

[37] E. L. Allgower and K. Georg, “Continuation and path following,” *Acta Numerica*, vol. 2, pp. 1–64, 1993. [Online]. Available: https://doi.org/10.1017/S0962492900002336

[38] H. Lamure and D. Michelucci, “Solving geometric constraints by homotopy,” *IEEE Transactions on Visualization and Computer Graphics*, vol. 2, no. 1, pp. 28–34, Mar. 1996. [Online]. Available: https://doi.org/10.1109/2945.489384

[39] W. Wen-Tsün, “Basic principles of mechanical theorem proving in elementary geometries,” *Journal of Automated Reasoning*, vol. 2, no. 3, pp. 221–252, Sep. 1986. [Online]. Available: https://doi.org/10.1007/BF02328447

[40] ——, *Mechanical Theorem Proving in Geometries: Basic Principles*, 1st ed., ser. Texts & Monographs in Symbolic Computation. Springer-Verlag, 1994. [Online]. Available: https://doi.org/10.1007/978-3-7091-6639-0

[41] S.-C. Chou, X.-S. Gao, and J.-Z. Zhang, "Automated generation of readable proofs with geometric invariants," *Journal of Automated Reasoning*, vol. 17, no. 3, pp. 325–347, Dec. 1996. [Online]. Available: https://doi.org/10.1007/BF00283133

[42] ——, "Automated generation of readable proofs with geometric invariants," *Journal of Automated Reasoning*, vol. 17, no. 3, pp. 349–370, Dec. 1996. [Online]. Available: https://doi.org/10.1007/BF00283134

[43] Y. J. Ahn, C. Hoffmann, and P. Rosen, "Geometric constraints on quadratic Bézier curves using minimal length and energy," *Journal of Computational and Applied Mathematics*, vol. 255, pp. 887–897, 2014. [Online]. Available: https://doi.org/10.1016/j.cam.2013.07.005

[44] F. Bao, Q. Sun, J. Pan, and Q. Duan, "A blending interpolator with value control and minimal strain energy," *Computers & Graphics*, vol. 34, no. 2, pp. 119–124, 2010. [Online]. Available: https://doi.org/10.1016/j.cag.2010.01.002

[45] M. Moll and L. E. Kavraki, "Path planning for deformable linear objects," *IEEE Transactions on Robotics*, vol. 22, no. 4, pp. 625–636, Aug. 2006. [Online]. Available: https://doi.org/10.1109/TRO.2006.878933

[46] Y. Xu, A. Joneja, and K. Tang, "Surface deformation under area constraints," *Computer-Aided Design and Applications*, vol. 6, no. 5, pp. 711–719, 2009. [Online]. Available: https://doi.org/10.3722/cadaps.2009.711-719

[47] J. Richter-Gebert and U. H. Kortenkamp, *The Cinderella.2 Manual: Working with The Interactive Geometry Software*, 1st ed. Springer, Berlin, Heidelberg, 2012. [Online]. Available: https://doi.org/10.1007/978-3-540-34926-6

[48] M. Freixas, R. Joan-Arinyo, and A. Soto-Riera, "A constraint-based dynamic geometry system," *Computer-Aided Design*, vol. 42, no. 2, pp. 151–161, 2010, ACM Symposium on Solid and Physical Modeling and Applications. [Online]. Available: https://doi.org/10.1016/j.cad.2009.02.016

[49] C. Chunhong, Z. Bin, W. Limin, and L. Wenhui, "The parametric design based on organizational evolutionary algorithm," in *PRICAI 2006: Trends in Artificial Intelligence*, ser. Lecture Notes in Computer Science, Q. Yang and G. Webb, Eds., vol. 4099. Guilin, China: Springer, Berlin, Heidelberg, 7–11 Aug. 2006, pp. 940–944. [Online]. Available: https://doi.org/10.1007/978-3-540-36668-3_110

[50] W. Li, M. Sun, H. Li, B. Fu, and H. Li, “Hierarchy and adaptive size particle swarm optimization algorithm for solving geometric constraint problems,” *Journal of Software*, vol. 7, no. 11, pp. 2567–2574, Nov. 2012.

[51] T. Tantau, *TikZ & PGF*, 3.1.9a ed., Jul. 2, 2021. [Online]. Available: https://github.com/pgf-tikz/pgf

[52] C. Obrecht, $E\Upsilon K\Lambda EI\Delta H\Sigma$ — *The Eukleides Manual*, 1.5.3 ed., 2010, accessed on 17 Jun 2019. [Online]. Available: http://www.eukleides.org/files/eukleides.pdf

[53] A. Leitão, R. Fernandes, and L. Santos, “Pushing the envelope: Stretching the limits of generative design,” in *SIGraDi 2013 — Knowledge-based Design, Proceedings of the 17th Conference of the Iberoamerican Society of Digital Graphics*, Departamento de Arquitectura de la Universidad Técnica Federico Santa María, Valparaíso, Chile, 20–22 Nov. 2013, pp. 235–238.

[54] J. Lopes and A. Leitão, “Portable generative design for CAD applications,” in *ACADIA 11 — Integration through Computation, Proceedings of the 31st Annual Conference of the Association for Computer Aided Design in Architecture (ACADIA)*, J. Taron, V. Parlac, B. Kolarevic, and J. Johnson, Eds., The University of Calgary, Banff, Canada, 13–16 Oct. 2011, pp. 196–203.

[55] R. Castelo-Branco and A. Leitão, “Integrated algorithmic design: A single-script approach for multiple design tasks,” in *ShoCK: Proceedings of the 35th Education and research in Computer Aided Architectural Design in Europe (eCAADe) Conference*, A. Fioravanti, S. Cursi, S. Elahmar, S. Gargar, G. Loffreda, Novembri, Gabriale, and A. Trent, Eds., vol. 1, Faculty of Civil and Industrial Engineering, Sapienza University of Rome, Rome, Italy, Sep. 2017, pp. 729–738.

[56] A. Leitão, “Improving generative design by combining abstract geometry and higher-order programming,” in *CAADRIA 2014 — Rethinking Comprehensive Design: Speculative Counterculture, Proceedings of the 19th International Conference on Computer-Aided Architectural Design Research in Asia (CAADRIA)*, N. Gu, S. Watanabe, H. Erhan, H. Haeusler, W. Huang, and R. Sosa, Eds., Kyoto Institute of Technology, Kyoto, Japan, 14–16 May 2014, pp. 575–584.

[57] H. A. van der Meiden and W. F. Bronsvoort, “A non-rigid cluster rewriting approach to solve systems of 3D geometric constraints,” *Computer-Aided Design*, vol. 42, no. 1, pp. 36–49, 2009. [Online]. Available: http://doi.org/10.1016/j.cad.2009.03.003

[58] H. Brönnimann, A. Fabri, G.-J. Giezeman, S. Hert, M. Hoffmann, L. Kettner, S. Pion, and S. Schirra, “2D and 3D linear geometry kernel,” in *CGAL User and Reference Manual*, 4.13 ed. CGAL Editorial Board, 2018. [Online]. Available: https://doc.cgal.org/4.13/Manual/packages.html#PkgKernel23Summary

[59] G. Mei, J. C. Tipper, and N. Xu, “Numerical robustness in geometric computation: An expository summary,” *Applied Mathematics & Information Sciences*, vol. 8, no. 6, pp. 2717–2727, Nov. 2014. [Online]. Available: https://doi.org/10.12785/amis/080607

[60] The CGAL Project, *CGAL User and Reference Manual*, 5.3 ed. CGAL Editorial Board, 2021. [Online]. Available: https://doc.cgal.org/5.3/Manual/packages.html

[61] C. Yap and T. Dubé, “The exact computation paradigm,” in *Computing in Euclidean Geometry*, ser. Lecture Notes Series on Computing. World Scientific, 1995, pp. 452–492. [Online]. Available: https://doi.org/10.1142/9789812831699_0011

[62] K. Mehlhorn and S. Näher, “LEDA: A library of efficient data types and algorithms,” in *Mathematical Foundations of Computer Science 1989*, ser. Lecture Notes in Computer Science, A. Kreczmar and G. Mirkowska, Eds., vol. 379. Kozubnik, Porąbka, Poland: Springer, Berlin, Heidelberg, Aug. 28 – Sep. 1, 1989, pp. 88–106. [Online]. Available: https://doi.org/10.1007/3-540-51486-4_58

[63] V. Karamcheti, C. Li, I. Pechtchanski, and C. Yap, “A core library for robust numeric and geometric computation,” in *Proceedings of the Fifteenth Annual Symposium on Computational Geometry*, ser. SCG ’99. Miami Beach, Florida, USA: Association for Computation Machinery, New York, NY, USA, 1999, pp. 351–359. [Online]. Available: https://doi.org/10.1145/304893.304989

[64] J. Yu, C. Yap, Z. Du, S. Pion, and H. Brönnimann, “The design of CORE 2: A library for exact numeric computation in geometry and algebra,” in *Mathematical Software — ICMS 2010*, ser. Lecture Notes in Computer Science, K. Fukuda, J. van der Hoeven, M. Joswig, and N. Takayama, Eds., vol. 6327. Kobe, Japan: Springer, Berlin, Heidelberg, 13–17 Sep. 2010, pp. 121–141. [Online]. Available: https://doi.org/10.1007/978-3-642-15582-6_24

[65] R. Aish, “DesignScript: Origins, explanation, illustration,” in *Computational Design Modelling*, C. Gengnagel, A. Kilian, N. Palz, and F. Scheurer, Eds. Berlin, Germany: Springer, Berlin, Heidelberg, 2011, pp. 1–8. [Online]. Available: https://doi.org/10.1007/978-3-642-23435-4_1

[66] M. Hohenwarter and K. Fuchs, “Combination of dynamic geometry, algebra and calculus in the software system GeoGebra,” in *Computer algebra systems and dynamic geometry systems in mathematics teaching conference*, 2004, pp. 1–6. [Online]. Available: https://www.researchgate.net/publication/228398347_Combination_of_dynamic_geometry_algebra_and_calculus_in_the_software_system_GeoGebra

[67] G. Lopez, B. Freeman-Benson, and A. Borning, “Kaleidoscope: A constraint imperative programming language,” in *Constraint Programming*, ser. NATO ASI F, B. Mayoh, E. Tyugu, and J. Penjam, Eds., vol. 131. Springer, Berlin, Heidelberg, 1994, pp. 313–329. [Online]. Available: https://doi.org/10.1007/978-3-642-85983-0_12

[68] R. de Regt, H. A. van der Meiden, and W. F. Bronsvoort, “A workbench for geometric constraint solving,” *Computer-Aided Design and Applications*, vol. 5, no. 1-4, pp. 471–482, 2008. [Online]. Available: http://doi.org/10.3722/cadaps.2008.471-482

[69] A. Leitão, R. Castelo-Branco, and G. Santos, “Game of renders: The use of game engines for architectural visualization,” in *Intelligent & Informed: Proceedings of the 24th CAADRIA Conference*, M. H. Haeusler, M. A. Schnabel, and T. Fukuda, Eds., vol. 1, Victoria University of Wellington, Wellington, New Zealand, 15–19 Aug. 2019, pp. 655–664.

[70] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, “Julia: A fresh approach to numerical computing,” *SIAM Review*, vol. 59, no. 1, pp. 65–98, 2017. [Online]. Available: https://doi.org/10.1137/141000671

[71] M. Flatt and PLT, “Reference: Racket,” PLT Design Inc., Tech. Rep. PLT-TR-2010-1, 2010. [Online]. Available: https://racket-lang.org/tr1

[72] B. Stroustrup, *The C++ Programming Language*, 4th ed. Addison-Wesley Professional, 2013.

[73] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, 2nd ed. Prentice Hall Professional Technical Reference, 1988.

[74] J. W. Backus, R. J. Beeber, S. Best, R. Goldberg, L. M. Haibt, H. L. Herrick, R. A. Nelson, D. Sayre, P. B. Sheridan, H. Stern, I. Ziller, R. A. Hughes, and R. Nutt, “The FORTRAN automatic coding system,” in *Western Joint Computer Conference: Techniques for Reliability*, ser. IRE-AIEE-ACM ’57 (Western). Los Angeles, California: Association for Computing Machinery, New York, NY, USA, 26–28 Feb. 1957, pp. 188–198. [Online]. Available: https://doi.org/10.1145/1455567.1455599

[75] R. Ventura. (2021, Jul. 15) CGAL.jl (Version v0.5.0). Zenodo. [Online]. Available: https://doi.org/10.5281/zenodo.5137358

[76] H. Brönnimann, A. Fabri, G.-J. Giezeman, S. Hert, M. Hoffmann, L. Kettner, S. Pion, and S. Schirra, “2D and 3D linear geometry kernel,” in *CGAL User and Reference Manual*, 5.3 ed. CGAL Editorial Board, 2021. [Online]. Available: https://doc.cgal.org/5.3/Manual/packages.html#PkgKernel23

[77] D. Abrahams and R. W. Grosse-Kunstleve, “Building hybrid systems with Boost.Python,” *C/C++ Users Journal*, vol. 21, no. LBNL-53142, 2003.

[78] F. Pinheiro, “Modelação Geométrica com Restrições,” Master’s thesis, Instituto Superior Técnico, Universidade de Lisboa, May 2016. [Online]. Available: https://fenix.tecnico.ulisboa.pt/cursos/meic-a/dissertacao/1691203502342199

[79] Maxima. (2021, Jun. 21,) Maxima, a computer algebra system. [Online]. Available: https://maxima.sourceforge.io

[80] J. Chen and J. Revels, "Robust benchmarking in noisy environments," *arXiv e-prints*, Aug. 2016, provided by the SAO/NASA Astrophysics Data System. [Online]. Available: https://ui.adsabs.harvard.edu/abs/2016arXiv160804295C

[81] R. A. Dixon, *Mathographics*. Mineola, New York: Dover Publications, Feb. 1991.

[82] S. Garcia and A. Leitão, "Shape grammars as design tools: An implementation of a multipurpose chair," *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, vol. 32, no. 2, pp. 240–255, 2018. [Online]. Available: https://doi.org/10.1017/S0890060417000610

[83] M. Karavelas, "2D voronoi diagram adaptor," in *CGAL User and Reference Manual*, 5.3 ed. CGAL Editorial Board, 2021. [Online]. Available: https://doc.cgal.org/5.3/Manual/packages.html#PkgVoronoiDiagram2

[84] V. Springel, "*E pur si muove*: Gallilean-invariant cosmological hydrodynamical simulations on a moving mesh," *Monthly Notices of the Royal Astronomical Society*, vol. 401, no. 2, pp. 791–851, Jan. 2010. [Online]. Available: https://doi.org/10.1111/j.1365-2966.2009.15715.x

[85] J. R. Shewchuk, "Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates," *Discrete & Computational Geometry*, vol. 18, no. 3, pp. 305–363, Oct. 1997.

ConstraintGM

Listing A.1: Source code from ConstraintGM's benchmarks for thirteen different scenarios involving the intersection of differently arranged geometric entities.

Adapted from: https://bitbucket.org/FabioPinheiro/geometric-constraints/src/master (July 2021)

```
#lang racket
(require "../core/entities-declaration.rkt"
         "../mathematical-module/geometric-restrictions.rkt"
         "../core/dispatcher.rkt"
         "../core/communication.rkt")

(define (benchmark-dispatcher e1 e2 [times 1000])
  (displayln "# new benchmark")
  (define seq (stream->list (in-range 0 times 1)))
  (collect-garbage)
  (time (for ([i seq]) (intersection-dispatcher e1 e2)))
  (collect-garbage)
  (time (for ([i seq]) (intersection e1 e2))))

(define l-l
  (list (cons (new-line (new-point 7 10) (new-point  7 -10))
              (new-line (new-point 0  5) (new-point 15  5)))
        (cons (new-line (new-point 1  0) (new-point  3  0))
```

```
19                (new-line (new-point 1  1) (new-point  3   1)))))
20  (define c-l
21    (list (cons (new-circle (new-point  0 0) 1)
22                (new-line   (new-point -5 0) (new-point 5 0)))
23          (cons (new-circle (new-point  0 0) 1)
24                (new-line   (new-point -5 1) (new-point 5 1)))
25          (cons (new-circle (new-point  0 0) 0.5)
26                (new-line   (new-point -5 1) (new-point 5 1)))))
27  (define c-c
28    (list (cons (new-circle (new-point   0  0)  1)
29                (new-circle (new-point   0  0)  2))
30          (cons (new-circle (new-point   0  0) 10)
31                (new-circle (new-point   1  0)  1.5))
32          (cons (new-circle (new-point 100  0)  1)
33                (new-circle (new-point  10  0)  2))
34          (cons (new-circle (new-point   1 -2)  3)
35                (new-circle (new-point   1 -2)  3))
36          (cons (new-circle (new-point   0  0)  1)
37                (new-circle (new-point   0  2)  1))
38          (cons (new-circle (new-point   0  0)  1.5)
39                (new-circle (new-point   1  0)  1.5))
40          (cons (new-circle (new-point   0  0)  1.5)
41                (new-circle (new-point   1  0)  1))
42          (cons (new-circle (new-point   0  0)  1.5)
43                (new-circle (new-point   1  1)  1))))
44
45  (displayln "### Time for start-maxima ###")
46  (time (start-maxima))
47  (displayln "### intersection-line-line ###")
48  (for ([i l-l]) (benchmark-dispatcher (car i) (cdr i)))
49  (displayln "### intersection-circle-line ###")
50  (for ([i c-l]) (benchmark-dispatcher (car i) (cdr i)))
51  (displayln "### intersection-circle-circle ###")
52  (for ([i c-c]) (benchmark-dispatcher (car i) (cdr i)))
```

Listing A.2: Benchmark code testing the same scenarios benchmarked by ConstraintGM using our solution.

```
1   import Base.Iterators: flatten
2
3   using BenchmarkTools
4   using CGAL
5
6   const ss = [
7       Segment2(Point2(7, 10), Point2( 7, -10)) =>
8       Segment2(Point2(0,  5), Point2(15,   5)),
9       Segment2(Point2(1,  0), Point2( 3,   0)) =>
10      Segment2(Point2(1,  1), Point2( 3,   1)),
11  ]
12  const cs = [
13      Circle2(Point2(0, 0), 1)     => Segment2(Point2(-5, 0), Point2(5, 0)),
14      Circle2(Point2(0, 0), 1)     => Segment2(Point2(-5, 1), Point2(5, 1)),
```

```
    Circle2(Point2(0, 0), 0.5^2) => Segment2(Point2(-5, 1), Point2(5, 1)),
]
const cc = [
    Circle2(Point2(  0,  0),  1)     => Circle2(Point2( 0,  0), 2^2),
    Circle2(Point2(  0,  0), 10^2)   => Circle2(Point2( 1,  0), 1.5^2),
    Circle2(Point2(100,  0),  1)     => Circle2(Point2(10,  0), 2^2),
    Circle2(Point2(  1, -2),  3^2)   => Circle2(Point2( 1, -2), 3^2),
    Circle2(Point2(  0,  0),  1)     => Circle2(Point2( 0,  2), 1),
    Circle2(Point2(  0,  0),  1.5^2) => Circle2(Point2( 1,  0), 1.5^2),
    Circle2(Point2(  0,  0),  1.5^2) => Circle2(Point2( 1,  0), 1),
    Circle2(Point2(  0,  0),  1.5^2) => Circle2(Point2( 1,  1), 1),
]

const scenarios = flatten([ss, cs, cc])

sample(a, b; times = 1000) = for i ∈ 1:times intersection(a, b) end

@info "========================= STARTING  BENCHMARKS ========================="
for (i, (a, b)) ∈ enumerate(scenarios)
    @info "Scenario $i" a b intersection(a, b)
    display(@benchmark sample($a, $b))
    println("\n")
end
@info "============================ BENCHMARKS  OVER ============================"
```

Listing A.3: Data produced through the execution of the code in listing A.1 a total of nine times to obtain a slightly bigger sample size. Includes post-processing that involves grouping and reducing results.

```
## Scenario 1
# Maxima
cpu time: 1231 real time:  2140 gc time: 17  cpu time:
cpu time: 1232 real time:  2185 gc time: 15   (min … max): 1.231 s …   1.526  s
cpu time: 1300 real time:  2226 gc time: 17   (median):    1.399 s
cpu time: 1361 real time:  2101 gc time: 17   (mean ± σ):  1.372 s ± 101.282 ms
cpu time: 1399 real time:  2125 gc time: 18  real time:
cpu time: 1405 real time:  2356 gc time: 18   (min … max): 2.101 s …   2.475  s
cpu time: 1444 real time:  2318 gc time: 17   (median):    2.226 s
cpu time: 1453 real time:  2475 gc time: 18   (mean ± σ):  2.265 s ± 142.334 ms
cpu time: 1526 real time:  2457 gc time: 19  gc time:
                                              (min … max): 15.000 ms … 19.000 ms
                                              (median):    17.000 ms
                                              (mean ± σ):  17.333 ms ±  1.118 ms
# GFL
cpu time:    7 real time:     7 gc time:  0   cpu time:
cpu time:    7 real time:     7 gc time:  0    (min … max): 7.000 ms … 12.000 ms
cpu time:    8 real time:     8 gc time:  0    (median):    8.000 ms
cpu time:    8 real time:     8 gc time:  0    (mean ± σ):  8.778 ms ±  1.641 ms
cpu time:    8 real time:     8 gc time:  0   real time:
cpu time:    9 real time:     9 gc time:  0    (min … max): 7.000 ms … 12.000 ms
cpu time:   10 real time:    10 gc time:  0    (median):    8.000 ms
cpu time:   10 real time:    10 gc time:  0    (mean ± σ):  8.778 ms ±  1.641 ms
cpu time:   12 real time:    12 gc time:  0   gc time:
                                              (min … max): 0.000 s … 0.000 s
```

```
                                                  (median):    0.000 s
                                                  (mean ± σ):  0.000 s ± 0.000 s

29  ## Scenario 2
    # Maxima
    cpu time:  569 real time:  1200 gc time:  9  cpu time:
    cpu time:  629 real time:  1492 gc time: 10   (min … max): 569.000 ms … 808.000 ms
    cpu time:  689 real time:  1637 gc time: 10   (median):    739.000 ms
    cpu time:  702 real time:  1599 gc time: 10   (mean ± σ):  722.889 ms ±  83.047 ms
    cpu time:  739 real time:  1723 gc time: 11  real time:
    cpu time:  783 real time:  1800 gc time: 12   (min … max): 1.200 s …   1.807  s
    cpu time:  792 real time:  1780 gc time: 12   (median):    1.723 s
    cpu time:  795 real time:  1807 gc time: 12   (mean ± σ):  1.645 s ± 197.524 ms
    cpu time:  808 real time:  1767 gc time: 15  gc time:
                                                  (min … max):  9.000 ms … 15.000 ms
                                                  (median):    11.000 ms
                                                  (mean ± σ):  11.222 ms ±  1.787 ms
43  # GFL
    cpu time:    6 real time:     6 gc time:  0  cpu time:
    cpu time:    6 real time:     6 gc time:  0   (min … max): 6.000 ms …   7.000 ms
    cpu time:    6 real time:     6 gc time:  0   (median):    6.000 ms
    cpu time:    6 real time:     6 gc time:  0   (mean ± σ):  6.222 ms ± 440.959 µs
    cpu time:    6 real time:     6 gc time:  0  real time:
    cpu time:    6 real time:     6 gc time:  0   (min … max): 6.000 ms …   7.000 ms
    cpu time:    6 real time:     6 gc time:  0   (median):    6.000 ms
    cpu time:    7 real time:     7 gc time:  0   (mean ± σ):  6.222 ms ± 440.959 µs
    cpu time:    7 real time:     7 gc time:  0  gc time:
                                                  (min … max): 0.000 s … 0.000 s
                                                  (median):    0.000 s
                                                  (mean ± σ):  0.000 s ± 0.000 s

57  ## Scenario 3
    # Maxima
    cpu time: 1597 real time:  3603 gc time: 14  cpu time:
    cpu time: 1771 real time:  3832 gc time: 16   (min … max): 1.597 s …   2.048  s
    cpu time: 1777 real time:  3637 gc time: 15   (median):    1.898 s
    cpu time: 1896 real time:  3821 gc time: 15   (mean ± σ):  1.892 s ± 151.672 ms
    cpu time: 1898 real time:  4149 gc time: 15  real time:
    cpu time: 2002 real time:  4271 gc time: 17   (min … max): 3.603 s …   4.306  s
    cpu time: 2011 real time:  4298 gc time: 16   (median):    3.832 s
    cpu time: 2028 real time:  3704 gc time: 15   (mean ± σ):  3.957 s ± 295.673 ms
    cpu time: 2048 real time:  4306 gc time: 16  gc time:
                                                  (min … max): 14.000 ms …  17.000 ms
                                                  (median):    15.000 ms
                                                  (mean ± σ):  15.444 ms ± 881.917 µs
71  # GFL
    cpu time:    9 real time:     9 gc time:  0   cpu time:
    cpu time:    9 real time:     9 gc time:  0    (min … max): 9.000 ms …  13.000 ms
    cpu time:    9 real time:     9 gc time:  0    (median):    9.000 ms
    cpu time:    9 real time:     9 gc time:  0    (mean ± σ):  9.444 ms ±   1.333 ms
    cpu time:    9 real time:     9 gc time:  0   real time:
    cpu time:    9 real time:     9 gc time:  0    (min … max): 9.000 ms …  13.000 ms
    cpu time:    9 real time:     9 gc time:  0    (median):    9.000 ms
    cpu time:    9 real time:     9 gc time:  0    (mean ± σ):  9.444 ms ±   1.333 ms
    cpu time:   13 real time:    13 gc time:  0   gc time:
                                                  (min … max): 0.000 s … 0.000 s
                                                  (median):    0.000 s
                                                  (mean ± σ):  0.000 s ± 0.000 s
```

```
 85  ## Scenario 4
     # Maxima
     cpu time: 1217 real time:  2703 gc time: 13  cpu time:
     cpu time: 1232 real time:  2710 gc time: 13   (min … max): 1.217 s …   1.482  s
     cpu time: 1296 real time:  2979 gc time: 14   (median):    1.343 s
     cpu time: 1341 real time:  3050 gc time: 14   (mean ± σ):  1.362 s ± 100.657 ms
     cpu time: 1343 real time:  3059 gc time: 15  real time:
     cpu time: 1429 real time:  3287 gc time: 13   (min … max): 2.703 s …   3.386  s
     cpu time: 1458 real time:  3172 gc time: 14   (median):    3.059 s
     cpu time: 1462 real time:  3386 gc time: 14   (mean ± σ):  3.069 s ± 243.768 ms
     cpu time: 1482 real time:  3282 gc time: 15  gc time:
                                                   (min … max): 13.000 ms …  15.000 ms
                                                   (median):    14.000 ms
                                                   (mean ± σ):  13.889 ms ± 781.736 µs
 99  # GFL
     cpu time:    8 real time:     8 gc time:  0  cpu time:
     cpu time:    8 real time:     8 gc time:  0   (min … max): 8.000 ms …  8.000 ms
     cpu time:    8 real time:     8 gc time:  0   (median):    8.000 ms
     cpu time:    8 real time:     8 gc time:  0   (mean ± σ):  8.000 ms ±  0.000  s
     cpu time:    8 real time:     8 gc time:  0  real time:
     cpu time:    8 real time:     8 gc time:  0   (min … max): 8.000 ms …  8.000 ms
     cpu time:    8 real time:     8 gc time:  0   (median):    8.000 ms
     cpu time:    8 real time:     8 gc time:  0   (mean ± σ):  8.000 ms ±  0.000  s
     cpu time:    8 real time:     8 gc time:  0  gc time:
                                                   (min … max): 0.000 s … 0.000 s
                                                   (median):    0.000 s
                                                   (mean ± σ):  0.000 s ± 0.000 s

113  ## Scenario 5
     # Maxima
     cpu time:  617 real time:  1775 gc time:  7  cpu time:
     cpu time:  622 real time:  1672 gc time:  8   (min … max): 617.000 ms … 673.000 ms
     cpu time:  633 real time:  1786 gc time:  8   (median):    656.000 ms
     cpu time:  648 real time:  1931 gc time:  8   (mean ± σ):  648.333 ms ±  19.862 ms
     cpu time:  656 real time:  1835 gc time:  7  real time:
     cpu time:  659 real time:  1866 gc time:  8   (min … max): 1.672 s …  1.944  s
     cpu time:  662 real time:  1880 gc time:  8   (median):    1.858 s
     cpu time:  665 real time:  1858 gc time:  8   (mean ± σ):  1.839 s ± 84.460 ms
     cpu time:  673 real time:  1944 gc time:  7  gc time:
                                                   (min … max): 7.000  ms …   8.000 ms
                                                   (median):    8.000  ms
                                                   (mean ± σ):  7.667  ms ± 500.000 µs
127  # GFL
     cpu time:     6 real time:     6 gc time:  0  cpu time:
     cpu time:     6 real time:     6 gc time:  0   (min … max): 6.000 ms …  6.000 ms
     cpu time:     6 real time:     6 gc time:  0   (median):    6.000 ms
     cpu time:     6 real time:     6 gc time:  0   (mean ± σ):  6.000 ms ±  0.000  s
     cpu time:     6 real time:     6 gc time:  0  real time:
     cpu time:     6 real time:     6 gc time:  0   (min … max): 6.000 ms …  6.000 ms
     cpu time:     6 real time:     6 gc time:  0   (median):    6.000 ms
     cpu time:     6 real time:     6 gc time:  0   (mean ± σ):  6.000 ms ±  0.000  s
     cpu time:     6 real time:     6 gc time:  0  gc time:
                                                    (min … max): 0.000 s … 0.000 s
                                                    (median):    0.000 s
                                                    (mean ± σ):  0.000 s ± 0.000 s

141  ## Scenario 6
```

```
     # Maxima
     cpu time:   530 real time:  1290 gc time:  4  cpu time:
     cpu time:   534 real time:  1213 gc time:  5   (min … max): 530.000 ms … 589.000 ms
     cpu time:   539 real time:  1304 gc time:  5   (median):    545.000 ms
     cpu time:   539 real time:  1249 gc time:  4   (mean ± σ):  547.889 ms ±  17.934 ms
     cpu time:   545 real time:  1353 gc time:  4  real time:
     cpu time:   546 real time:  1278 gc time:  4   (min … max): 1.213 s …  1.382  s
     cpu time:   547 real time:  1382 gc time:  5   (median):    1.304 s
     cpu time:   562 real time:  1359 gc time:  5   (mean ± σ):  1.311 s ± 59.139 ms
     cpu time:   589 real time:  1373 gc time:  6  gc time:
                                                        (min … max): 4.000  ms …   6.000 ms
                                                        (median):    5.000  ms
                                                        (mean ± σ):  4.667  ms ± 707.107 µs
155  # GFL
     cpu time:     5 real time:     5 gc time:  0  cpu time:
     cpu time:     5 real time:     5 gc time:  0   (min … max): 5.000 ms …   6.000 ms
     cpu time:     5 real time:     5 gc time:  0   (median):    5.000 ms
     cpu time:     5 real time:     5 gc time:  0   (mean ± σ):  5.111 ms ± 333.333 µs
     cpu time:     5 real time:     5 gc time:  0  real time:
     cpu time:     5 real time:     5 gc time:  0   (min … max): 5.000 ms …   6.000 ms
     cpu time:     5 real time:     5 gc time:  0   (median):    5.000 ms
     cpu time:     5 real time:     5 gc time:  0   (mean ± σ):  5.111 ms ± 333.333 µs
     cpu time:     6 real time:     6 gc time:  0  gc time:
                                                    (min … max): 0.000 s … 0.000 s
                                                    (median):    0.000 s
                                                    (mean ± σ):  0.000 s ± 0.000 s

169  ## Scenario 7
     # Maxima
     cpu time:   539 real time:  1761 gc time:  5  cpu time:
     cpu time:   548 real time:  1843 gc time:  4   (min … max): 539.000 ms … 577.000 ms
     cpu time:   554 real time:  1852 gc time:  4   (median):    558.000 ms
     cpu time:   556 real time:  1932 gc time:  4   (mean ± σ):  559.000 ms ±  11.758 ms
     cpu time:   558 real time:  1802 gc time:  4  real time:
     cpu time:   560 real time:  1885 gc time:  4   (min … max): 1.761 s …  1.932  s
     cpu time:   568 real time:  1855 gc time:  5   (median):    1.852 s
     cpu time:   571 real time:  1813 gc time:  4   (mean ± σ):  1.847 s ± 50.466 ms
     cpu time:   577 real time:  1879 gc time:  4  gc time:
                                                    (min … max): 4.000  ms …   5.000 ms
                                                    (median):    4.000  ms
                                                    (mean ± σ):  4.200  ms ± 440.959 µs
183  # GFL
     cpu time:     5 real time:     5 gc time:  0  cpu time:
     cpu time:     5 real time:     5 gc time:  0   (min … max): 5.000 ms …   6.000 ms
     cpu time:     5 real time:     5 gc time:  0   (median):    5.000 ms
     cpu time:     5 real time:     5 gc time:  0   (mean ± σ):  5.333 ms ± 500.000 µs
     cpu time:     5 real time:     5 gc time:  0  real time:
     cpu time:     5 real time:     5 gc time:  0   (min … max): 5.000 ms …   6.000 ms
     cpu time:     6 real time:     6 gc time:  0   (median):    5.000 ms
     cpu time:     6 real time:     6 gc time:  0   (mean ± σ):  5.333 ms ± 500.000 µs
     cpu time:     6 real time:     6 gc time:  0  gc time:
                                                    (min … max): 0.000 s … 0.000 s
                                                    (median):    0.000 s
                                                    (mean ± σ):  0.000 s ± 0.000 s

197  ## Scenario 8
     # Maxima
     cpu time:   515 real time:  1794 gc time:  4  cpu time:
```

```
      cpu time:   528 real time:  1829 gc time:  5   (min … max): 515.000 ms … 568.000 ms
      cpu time:   533 real time:  1869 gc time:  4   (median):    544.000 ms
      cpu time:   542 real time:  1865 gc time:  4   (mean ± σ):  542.333 ms ±  15.516 ms
      cpu time:   544 real time:  1893 gc time:  4  real time:
      cpu time:   548 real time:  1867 gc time:  4   (min … max): 1.794 s …  1.921  s
      cpu time:   548 real time:  1892 gc time:  5   (median):    1.869 s
      cpu time:   555 real time:  1886 gc time:  4   (mean ± σ):  1.868 s ± 37.650 ms
      cpu time:   568 real time:  1921 gc time:  4  gc time:
                                                     (min … max): 4.000  ms …   5.000 ms
                                                     (median):    4.000  ms
                                                     (mean ± σ):  4.200  ms ± 440.959 µs
211   # GFL
      cpu time:     5 real time:     5 gc time:  0  cpu time:
      cpu time:     5 real time:     5 gc time:  0   (min … max): 5.000 ms …   6.000 ms
      cpu time:     5 real time:     5 gc time:  0   (median):    5.000 ms
      cpu time:     5 real time:     5 gc time:  0   (mean ± σ):  5.333 ms ± 500.000 µs
      cpu time:     5 real time:     5 gc time:  0  real time:
      cpu time:     5 real time:     5 gc time:  0   (min … max): 5.000 ms …   6.000 ms
      cpu time:     6 real time:     6 gc time:  0   (median):    5.000 ms
      cpu time:     6 real time:     6 gc time:  0   (mean ± σ):  5.333 ms ± 500.000 µs
      cpu time:     6 real time:     6 gc time:  0  gc time:
                                                     (min … max): 0.000 s … 0.000 s
                                                     (median):    0.000 s
                                                     (mean ± σ):  0.000 s ± 0.000 s

225   ## Scenario 9
      # Maxima
      cpu time:   644 real time:  4223 gc time:  5  cpu time:
      cpu time:   662 real time:  4313 gc time:  4   (min … max): 644.000 ms … 702.000 ms
      cpu time:   665 real time:  4257 gc time:  5   (median):    669.000 ms
      cpu time:   666 real time:  4230 gc time:  5   (mean ± σ):  674.556 ms ±  18.789 ms
      cpu time:   669 real time:  4311 gc time:  5  real time:
      cpu time:   677 real time:  4287 gc time:  6   (min … max): 4.223 s …  4.313  s
      cpu time:   686 real time:  4248 gc time:  4   (median):    4.287 s
      cpu time:   700 real time:  4312 gc time:  5   (mean ± σ):  4.277 s ± 37.581 ms
      cpu time:   702 real time:  4311 gc time:  6  gc time:
                                                     (min … max): 4.000 ms …   5.000 ms
                                                     (median):    4.000 ms
                                                     (mean ± σ):  4.200 ms ± 440.959 µs
239   # GFL
      cpu time:     5 real time:     5 gc time:  0  cpu time:
      cpu time:     5 real time:     5 gc time:  0   (min … max): 5.000 ms …   6.000 ms
      cpu time:     5 real time:     5 gc time:  0   (median):    5.000 ms
      cpu time:     5 real time:     5 gc time:  0   (mean ± σ):  5.222 ms ± 440.959 µs
      cpu time:     5 real time:     5 gc time:  0  real time:
      cpu time:     5 real time:     5 gc time:  0   (min … max): 5.000 ms …   6.000 ms
      cpu time:     5 real time:     5 gc time:  0   (median):    5.000 ms
      cpu time:     6 real time:     6 gc time:  0   (mean ± σ):  5.222 ms ± 440.959 µs
      cpu time:     6 real time:     6 gc time:  0  gc time:
                                                     (min … max): 0.000 s … 0.000 s
                                                     (median):    0.000 s
                                                     (mean ± σ):  0.000 s ± 0.000 s

253   ## Scenario 10
      # Maxima
      cpu time:   990 real time:  2249 gc time:  8  cpu time:
      cpu time:  1011 real time:  2350 gc time:  9   (min … max): 990.000 ms …  1.197  s
      cpu time:  1027 real time:  2493 gc time:  9   (median):      1.045  s
```

```
      cpu time:  1044 real time:  2575 gc time:  8   (mean ± σ):    1.058  s ± 59.110 ms
      cpu time:  1045 real time:  2482 gc time:  9  real time:
      cpu time:  1060 real time:  2480 gc time:  9   (min … max): 2.249 s …   3.035  s
      cpu time:  1070 real time:  2256 gc time:  9   (median):    2.482 s
      cpu time:  1076 real time:  2636 gc time:  8   (mean ± σ):  2.506 s ± 238.696 ms
      cpu time:  1197 real time:  3035 gc time: 11  gc time:
                                                     (min … max): 8.000  ms …  11.000 ms
                                                     (median):    9.000  ms
                                                     (mean ± σ):  8.889  ms ± 927.961 µs
267   # GFL
      cpu time:     7 real time:     7 gc time:  0  cpu time:
      cpu time:     7 real time:     7 gc time:  0   (min … max): 7.000 ms …   8.000 ms
      cpu time:     7 real time:     7 gc time:  0   (median):    7.000 ms
      cpu time:     7 real time:     7 gc time:  0   (mean ± σ):  7.222 ms ± 440.959 µs
      cpu time:     7 real time:     7 gc time:  0  real time:
      cpu time:     7 real time:     7 gc time:  0   (min … max): 7.000 ms …   8.000 ms
      cpu time:     7 real time:     7 gc time:  0   (median):    7.000 ms
      cpu time:     8 real time:     8 gc time:  0   (mean ± σ):  7.222 ms ± 440.959 µs
      cpu time:     8 real time:     8 gc time:  0  gc time:
                                                     (min … max): 0.000 s … 0.000 s
                                                     (median):    0.000 s
                                                     (mean ± σ):  0.000 s ± 0.000 s

281   ## Scenario 11
      # Maxima
      cpu time:  1368 real time:  3237 gc time:  9  cpu time:
      cpu time:  1383 real time:  3326 gc time: 10   (min … max): 1.368 s …  1.605  s
      cpu time:  1408 real time:  3233 gc time: 10   (median):    1.458 s
      cpu time:  1416 real time:  3306 gc time:  9   (mean ± σ):  1.482 s ± 97.673 ms
      cpu time:  1458 real time:  3484 gc time:  9  real time:
      cpu time:  1501 real time:  3417 gc time: 10   (min … max): 3.233 s …   3.882  s
      cpu time:  1597 real time:  3882 gc time: 11   (median):    3.417 s
      cpu time:  1602 real time:  3669 gc time: 11   (mean ± σ):  3.493 s ± 258.715 ms
      cpu time:  1605 real time:  3882 gc time: 10  gc time:
                                                     (min … max): 9.000  ms …  11.000 ms
                                                     (median):   10.000  ms
                                                     (mean ± σ):  9.889  ms ± 781.736 µs
295   # GFL
      cpu time:     9 real time:     9 gc time:  0  cpu time:
      cpu time:     9 real time:     9 gc time:  0   (min … max): 7.000 ms …   8.000 ms
      cpu time:     9 real time:     9 gc time:  0   (median):    7.000 ms
      cpu time:     9 real time:     9 gc time:  0   (mean ± σ):  7.222 ms ± 440.959 µs
      cpu time:    10 real time:    10 gc time:  0  real time:
      cpu time:    10 real time:    10 gc time:  0   (min … max): 7.000 ms …   8.000 ms
      cpu time:    10 real time:    10 gc time:  0   (median):    7.000 ms
      cpu time:    10 real time:    10 gc time:  0   (mean ± σ):  7.222 ms ± 440.959 µs
      cpu time:    10 real time:    10 gc time:  0  gc time:
                                                     (min … max): 0.000 s … 0.000 s
                                                     (median):    0.000 s
                                                     (mean ± σ):  0.000 s ± 0.000 s

309   ## Scenario 12
      # Maxima
      cpu time:  1520 real time:  3614 gc time: 10  cpu time:
      cpu time:  1531 real time:  3652 gc time:  9   (min … max): 1.520 s …  1.753  s
      cpu time:  1551 real time:  3759 gc time: 11   (median):    1.621 s
      cpu time:  1607 real time:  3791 gc time: 11   (mean ± σ):  1.622 s ± 80.986 ms
      cpu time:  1621 real time:  3758 gc time: 10  real time:
```

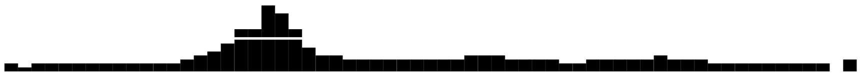
```
      cpu time:  1637 real time:  3961 gc time: 10   (min … max): 3.614 s …   3.016  s
      cpu time:  1668 real time:  4010 gc time: 10   (median):    3.791 s
      cpu time:  1716 real time:  3912 gc time: 11   (mean ± σ):  3.333 s ± 150.402 ms
      cpu time:  1753 real time:  4016 gc time: 11  gc time:
                                                     (min … max): 9.000  ms …  11.000 ms
                                                     (median):   10.000  ms
                                                     (mean ± σ): 10.333  ms ± 707.107 µs
323   # GFL
      cpu time:     9 real time:     9 gc time:  0  cpu time:
      cpu time:     9 real time:     9 gc time:  0   (min … max): 9.000 ms …  10.000 ms
      cpu time:     9 real time:     9 gc time:  0   (median):    9.000 ms
      cpu time:     9 real time:     9 gc time:  0   (mean ± σ):  9.444 ms ± 527.046 µs
      cpu time:     9 real time:     9 gc time:  0  real time:
      cpu time:    10 real time:    10 gc time:  0   (min … max): 9.000 ms …  10.000 ms
      cpu time:    10 real time:    10 gc time:  0   (median):    9.000 ms
      cpu time:    10 real time:    10 gc time:  0   (mean ± σ):  9.444 ms ± 527.046 µs
      cpu time:    10 real time:    10 gc time:  0  gc time:
                                                     (min … max): 0.000 s … 0.000 s
                                                     (median):    0.000 s
                                                     (mean ± σ):  0.000 s ± 0.000 s

337   ## Scenario 13
      # Maxima
      cpu time:  2373 real time: 10990 gc time: 15  cpu time:
      cpu time:  2397 real time: 10976 gc time: 15   (min … max): 2.373 s …   2.672  s
      cpu time:  2479 real time: 11129 gc time: 16   (median):    2.552 s
      cpu time:  2496 real time: 11109 gc time: 15   (mean ± σ):  2.531 s ± 103.464 ms
      cpu time:  2552 real time: 11087 gc time: 15  real time:
      cpu time:  2573 real time: 11144 gc time: 16   (min … max): 10.976 s … 11.197  s
      cpu time:  2602 real time: 11191 gc time: 16   (median):    11.129 s
      cpu time:  2639 real time: 11197 gc time: 15   (mean ± σ):  11.111 s ± 81.302 ms
      cpu time:  2672 real time: 11176 gc time: 17  gc time:
                                                     (min … max): 15.000  ms …  17.000 ms
                                                     (median):    15.000  ms
                                                     (mean ± σ):  15.556  ms ± 726.486 µs
351   # GFL
      cpu time:     9 real time:     9 gc time:  0  cpu time:
      cpu time:    10 real time:    10 gc time:  0   (min … max):  9.000 ms … 13.000 ms
      cpu time:    10 real time:    10 gc time:  0   (median):    10.000 ms
      cpu time:    10 real time:    10 gc time:  0   (mean ± σ):  10.222 ms ±  1.093 ms
      cpu time:    10 real time:    10 gc time:  0  real time:
      cpu time:    10 real time:    10 gc time:  0   (min … max):  9.000 ms … 13.000 ms
      cpu time:    10 real time:    10 gc time:  0   (median):    10.000 ms
      cpu time:    10 real time:    10 gc time:  0   (mean ± σ):  10.222 ms ±  1.093 ms
      cpu time:    13 real time:    13 gc time:  0  gc time:
                                                     (min … max): 0.000 s … 0.000 s
                                                     (median):    0.000 s
                                                     (mean ± σ):  0.000 s ± 0.000 s
```

Listing A.4: More detailed data from benchmarking our solution by submitting it to a test suite analogous to ConstraintGM's. This is the output produced by listing A.2.

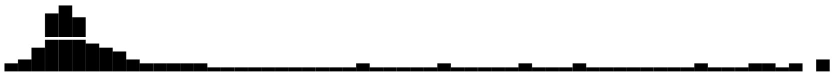
```
1   BenchmarkTools.Trial: 10000 samples with 1 evaluation.
     Range (min … max):  248.077 µs … 311.634 ms  ┊ GC (min … max):  0.00% … 45.83%
     Time  (median):     266.237 µs               ┊ GC (median):     0.00%
```

```
 Time  (mean ± σ):   362.191 μs ±   4.841 ms  ┊ GC (mean ± σ):  10.69% ±  1.26%
```

```
  248 μs           Histogram: frequency by time          393 μs <

 Memory estimate: 15.62 KiB, allocs estimate: 1000.

12  BenchmarkTools.Trial: 10000 samples with 1 evaluation.
 Range (min … max):  149.461 μs … 199.677 μs  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     157.493 μs               ┊ GC (median):    0.00%
 Time  (mean ± σ):   159.567 μs ±   4.634 μs  ┊ GC (mean ± σ):  0.00% ± 0.00%
```

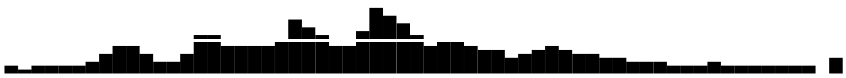


```
  149 μs           Histogram: frequency by time          173 μs <

 Memory estimate: 0 bytes, allocs estimate: 0.

23  BenchmarkTools.Trial: 2662 samples with 1 evaluation.
 Range (min … max):  1.520 ms … 101.968 ms  ┊ GC (min … max): 0.00% … 28.94%
 Time  (median):     1.564 ms               ┊ GC (median):    0.00%
 Time  (mean ± σ):   1.871 ms ±   4.892 ms  ┊ GC (mean ± σ):  4.85% ±  1.83%
```



```
  1.52 ms          Histogram: frequency by time          260 ms <

 Memory estimate: 140.62 KiB, allocs estimate: 4000.

34  BenchmarkTools.Trial: 5196 samples with 1 evaluation.
 Range (min … max):  873.160 μs … 265.349 ms  ┊ GC (min … max): 0.00% … 32.06%
 Time  (median):     893.135 μs               ┊ GC (median):    0.00%
 Time  (mean ± σ):   955.815 μs ±   3.710 ms  ┊ GC (mean ± σ):  2.35% ±  1.17%
```

```
  873 μs           Histogram: frequency by time          977 μs <

 Memory estimate: 15.62 KiB, allocs estimate: 1000.

45  BenchmarkTools.Trial: 8242 samples with 1 evaluation.
 Range (min … max):  576.334 μs …  1.000 ms  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     599.660 μs              ┊ GC (median):    0.00%
 Time  (mean ± σ):   600.386 μs ± 19.807 μs  ┊ GC (mean ± σ):  0.00% ± 0.00%
```



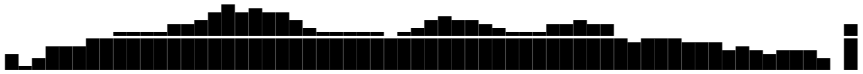
```
  576 μs          Histogram: frequency by time          629 μs <

 Memory estimate: 0 bytes, allocs estimate: 0.

56  BenchmarkTools.Trial: 10000 samples with 1 evaluation.
 Range (min … max):  228.941 μs … 499.228 μs  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     238.928 μs               ┊ GC (median):    0.00%
 Time  (mean ± σ):   241.531 μs ±  12.504 μs  ┊ GC (mean ± σ):  0.00% ± 0.00%
```

Memory estimate: 0 bytes, allocs estimate: 0.

```
 67  BenchmarkTools.Trial: 10000 samples with 1 evaluation.
      Range (min … max):  231.595 µs … 339.919 µs  ┊ GC (min … max): 0.00% … 0.00%
      Time  (median):     240.814 µs               ┊ GC (median):    0.00%
      Time  (mean ± σ):   242.780 µs ±   5.692 µs  ┊ GC (mean ± σ):  0.00% ± 0.00%
```

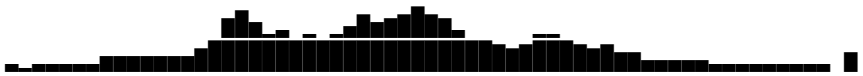
Memory estimate: 0 bytes, allocs estimate: 0.

```
 78  BenchmarkTools.Trial: 10000 samples with 1 evaluation.
      Range (min … max):  234.389 µs … 492.523 µs  ┊ GC (min … max): 0.00% … 0.00%
      Time  (median):     242.840 µs               ┊ GC (median):    0.00%
      Time  (mean ± σ):   245.425 µs ±  11.840 µs  ┊ GC (mean ± σ):  0.00% ± 0.00%
```



Memory estimate: 0 bytes, allocs estimate: 0.

```
 89  BenchmarkTools.Trial: 9576 samples with 1 evaluation.
      Range (min … max):  393.697 µs … 309.638 ms  ┊ GC (min … max): 0.00% … 48.23%
      Time  (median):     418.072 µs               ┊ GC (median):    0.00%
      Time  (mean ± σ):   515.829 µs ±   4.881 ms  ┊ GC (mean ± σ):  8.11% ±  1.29%
```



Memory estimate: 15.62 KiB, allocs estimate: 1000.

```
100  BenchmarkTools.Trial: 8297 samples with 1 evaluation.
      Range (min … max):  493.222 µs … 317.227 ms  ┊ GC (min … max): 0.00% … 47.42%
      Time  (median):     511.171 µs               ┊ GC (median):    0.00%
      Time  (mean ± σ):   629.341 µs ±   5.360 ms  ┊ GC (mean ± σ):  7.68% ±  1.42%
```

Memory estimate: 15.62 KiB, allocs estimate: 1000.

```
111  BenchmarkTools.Trial: 3425 samples with 1 evaluation.
      Range (min … max):  1.102 ms … 103.956 ms  ┊ GC (min … max): 0.00% … 28.37%
      Time  (median):     1.121 ms               ┊ GC (median):    0.00%
      Time  (mean ± σ):   1.453 ms ±   5.070 ms  ┊ GC (mean ± σ):  6.83% ±  1.93%
```

```
 1.1 ms          Histogram: frequency by time        1.51 ms <

 Memory estimate: 140.62 KiB, allocs estimate: 4000.
```

```
122  BenchmarkTools.Trial: 3422 samples with 1 evaluation.
 Range (min … max):  1.097 ms … 102.675 ms  ┊ GC (min … max): 0.00% … 28.67%
 Time  (median):     1.123 ms               ┊ GC (median):    0.00%
 Time  (mean ± σ):   1.455 ms ±   5.088 ms  ┊ GC (mean ± σ):  6.84% ±  1.93%
```

```
 1.1 ms          Histogram: frequency by time        1.55 ms <

 Memory estimate: 140.62 KiB, allocs estimate: 4000.
```

```
133  BenchmarkTools.Trial: 3403 samples with 1 evaluation.
 Range (min … max):  1.105 ms … 102.643 ms  ┊ GC (min … max): 0.00% … 28.69%
 Time  (median):     1.132 ms               ┊ GC (median):    0.00%
 Time  (mean ± σ):   1.463 ms ±   5.055 ms  ┊ GC (mean ± σ):  6.78% ±  1.94%
```

```
 1.11 ms          Histogram: frequency by time         1.6 ms <

 Memory estimate: 140.62 KiB, allocs estimate: 4000.
```

B

Voronoi Delaunay

Listing B.1: Wrapper code using JlCxx around the Computational Geometry Algorithms Library (CGAL)'s algorithms for producing 2D Delaunay Triangulations and Voronoi Diagrams.

```
#include <sstream>
#include <vector>

#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Delaunay_triangulation_2.h>
#include <CGAL/Delaunay_triangulation_adaptation_traits_2.h>
#include <CGAL/Delaunay_triangulation_adaptation_policies_2.h>
#include <CGAL/Voronoi_diagram_2.h>
#include <CGAL/IO/io.h>

#include <jlcxx/jlcxx.hpp>
#include <jlcxx/stl.hpp>

using Kernel = CGAL::Epick;

using DT = CGAL::Delaunay_triangulation_2<Kernel>;
using AT = CGAL::Delaunay_triangulation_adaptation_traits_2<DT>;
using AP = CGAL::Delaunay_triangulation_caching_degeneracy_removal_policy_2<DT>;
using VD = CGAL::Voronoi_diagram_2<DT, AT, AP>;
```

```
20
21  using Point_2   = VD::Point_2;
22  using Segment_2 = DT::Segment;
23
24  JLCXX_MODULE define_julia_module(jlcxx::Module &m) {
25    m.add_type<Point_2>("Point2")
26     .constructor<double, double>()
27     .method("x", &Point_2::x)
28     .method("y", &Point_2::y);
29    jlcxx::stl::apply_stl<Point_2>(m);
30
31    m.add_type<Segment_2>("Segment2")
32     .method("source", &Segment_2::source)
33     .method("target", &Segment_2::target);
34
35    // Delaunay Triangulation
36    std::string t_name = "DelaunayTriangulation2";
37    m.add_type<DT::Vertex>(t_name + "Vertex")
38     .method("point", [](const DT::Vertex& v){ return v.point(); });
39
40    m.add_type<DT::Edge>(t_name + "Edge");
41
42    auto dt = m.add_type<DT>(t_name)
43     .method("edges", [](const DT& dt) {
44       std::vector<DT::Edge> res(dt.edges_begin(), dt.edges_end());
45       return res;
46     })
47     .method("segment", [](const DT& dt, const DT::Edge& e) {
48       return dt.segment(e);
49     });
50    m.set_override_module(jl_base_module);
51    dt.method("insert!", [](DT& dt, std::vector<Point_2> ps) -> DT& {
52      dt.insert(ps.begin(), ps.end());
53      return dt;
54    });
55    m.unset_override_module();
56
57    // Voronoi diagram
58     std::string vd_name = "VoronoiDiagram2";
59     m.add_type<VD::Vertex>(vd_name + "Vertex")
60      .method("point", &VD::Vertex::point);
61
62     m.add_type<VD::Halfedge>(vd_name + "Halfedge")
63      .method("source", [](const VD::Halfedge& h) {
64        return h.has_source()
65          ? (jl_value_t*)jlcxx::box<VD::Vertex>(*(h.source()))
66          : jl_nothing;
67      })
68      .method("target", [](const VD::Halfedge& h) {
69        return h.has_target()
70          ? (jl_value_t*)jlcxx::box<VD::Vertex>(*(h.target()))
71          : jl_nothing;
72      });
```

```
73
74      m.add_type<VD>(vd_name)
75       .constructor<const DT&>()
76       .method("edges", [](const VD& vd) {
77         std::vector<VD::Halfedge> res(vd.edges_begin(), vd.edges_end());
78         return res;
79       });
80  }
```

Listing B.2: Julia code used to plot Delaunay Triangulations and Voronoi Diagrams using both CGAL's algorithm and `VoronoiDelaunay.jl`'s algorithm.

```
1   using VoronoiDelaunay
2   include("Voronoi.jl"); using .Voronoi
3
4   import RandomNumbers.MersenneTwisters: MT19937
5
6   randpoints(n=21, lo=nextfloat(1.), hi=prevfloat(2., 2); seed=0xdeadbeef) =
7       let rng = MT19937(seed)
8           map(1:n) do _
9               x = (rand(rng); rand(rng))
10              y = (rand(rng); rand(rng))
11              lo .+ (hi - lo) .* (x, y)
12          end
13      end
14
15  plotxy(dt::DelaunayTriangulation2) =
16      let xs = Float64[], ys = Float64[]
17          foreach(segment.(Ref(dt), edges(dt))) do seg
18              s, t = source(seg), target(seg)
19              push!(xs, x(s)[], x(t)[], NaN)
20              push!(ys, y(s)[], y(t)[], NaN)
21          end
22          xs, ys
23      end
24
25  plotxy(vd::VoronoiDiagram2) =
26      let xs = Float64[], ys = Float64[],
27          isbounded(hf) = all(!isnothing, (source(hf), target(hf)))
28          for hf ∈ filter(isbounded, edges(vd))
29              s, t = point.((source(hf), target(hf)))
30              push!(xs, x(s)[], x(t)[], NaN)
31              push!(ys, y(s)[], y(t)[], NaN)
32          end
33          xs, ys
34      end
35
36  using Plots; pgfplotsx()
37  let ps  = randpoints(),
38      cdt = DelaunayTriangulation2(),
```

```
    vdt = DelaunayTessellation()
    insert!(cdt, [Point2(p...) for p ∈ ps])
    push!(vdt,   [Point(p...)  for p ∈ ps])

    plot(plotxy(cdt)
       , lab = "Voronoi.jl"
       , lims = (.9, 2.1)
       , titlefontsize = 10
       , tickfontfamily = "monospace"
       , color = :red)
    plot!(getplotxy(delaunayedges(vdt))
        , lab = "VoronoiDelaunay.jl"
        , color = :springgreen
        , seriesalpha = .8
        , extra_kwargs = :subplot, legend_columns = -1)
    title!("Delaunay Triangulation")
    pl1 = current()

    plot(plotxy(VoronoiDiagram2(cdt))
       , lab = "Voronoi.jl"
       , lims = (.9, 2.1)
       , titlefontsize = 10
       , color = :red)
    plot!(getplotxy(voronoiedges(vdt))
        , lab = "VoronoiDelaunay.jl"
        , color = :springgreen
        , seriesalpha = .8
        , extra_kwargs = :subplot, legend_columns = -1)
    title!("Voronoi Diagram")
    pl2 = current()

    plot(pl1, pl2
       , xlabel = raw"$x$"
       , ylabel = raw"$y$"
       , size = (700,350)
       , legend = :topright
       , legendfonthalign = :left)
    savefig(joinpath(dirname(@__DIR__), "tikz", "voronoi-plots.tikz"))
    current()
end
```

Listing B.3: Julia code used to benchmark the creation of Delaunay Triangulations using both CGAL's algorithm and `VoronoiDelaunay.jl`'s algorithm.

```
using VoronoiDelaunay
include("Voronoi.jl"); using .Voronoi

import RandomNumbers.MersenneTwisters: MT19937

randpoints(n=21, lo=nextfloat(1.), hi=prevfloat(2., 2); seed=0xdeadbeef) =
    let rng = MT19937(seed)
```

```
        map(1:n) do _
            x = (rand(rng); rand(rng))
            y = (rand(rng); rand(rng))
            lo .+ (hi - lo) .* (x, y)
        end
    end

using BenchmarkTools
import Statistics: std
printhead(io::IO, s, w = 80; delim = "=") =
    println(io,
            delim^(w÷2 - floor(Int, length(s) / 2) - 1)
          * " $s "
          * delim^(w÷2 - ceil(Int, length(s) / 2) - 1))
printhead(args...; kwargs...) = printhead(stdout, args...; kwargs...)

cd(joinpath(dirname(@__DIR__)), "data")
open("voronoi-cgal.csv", "w") do cio
    open("voronoi-vdjl.csv", "w") do vio
        for n ∈ Int.(exp10.(2:6))
            ps = randpoints(n)
            ct = @benchmark insert!(DelaunayTriangulation2()
                                    , $([Point2(p...) for p ∈ ps]))
            vt = @benchmark push!(DelaunayTessellation()
                                  , $([Point(p...) for p ∈ ps]))
            printhead("$n points")
            printhead("CGAL.jl", delim="~")
            display(ct)
            println()
            printhead("VoronoiDelaunay.jl", delim="~")
            display(vt)
            println("\n")
            println(cio, "$n,$(time(mean(ct))),$(time(std(ct)))")
            println(vio, "$n,$(time(mean(vt))),$(time(std(vt)))")
        end
    end
end
```

Listing B.4: Detailed benchmark data from several batch point insertion tests using both CGAL's algorithm and `VoronoiDelaunay.jl`'s algorithm.

```
================================== 100 points ==================================
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ CGAL.jl ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
BenchmarkTools.Trial: 10000 samples with 1 evaluation.
 Range (min … max):  34.432 μs … 554.193 μs  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     41.416 μs               ┊ GC (median):    0.00%
 Time  (mean ± σ):   42.145 μs ±   5.638 μs  ┊ GC (mean ± σ):  0.00% ± 0.00%
```

```
  34.4 μs         Histogram: frequency by time          52 μs <

 Memory estimate: 928 bytes, allocs estimate: 3.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ VoronoiDelaunay.jl ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
BenchmarkTools.Trial: 10000 samples with 1 evaluation.
 Range (min … max):  43.931 μs …  1.489 ms  ┊ GC (min … max): 0.00% … 91.99%
 Time  (median):     51.264 μs              ┊ GC (median):    0.00%
 Time  (mean ± σ):   53.799 μs ± 39.811 μs  ┊ GC (mean ± σ):  2.01% ±  2.65%
```

```
  43.9 μs         Histogram: frequency by time       69.7 μs <

 Memory estimate: 39.41 KiB, allocs estimate: 215.

================================== 1000 points ==================================
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ CGAL.jl ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
BenchmarkTools.Trial: 9667 samples with 1 evaluation.
 Range (min … max):  478.624 μs … 124.182 ms  ┊ GC (min … max): 0.00% … 2.22%
 Time  (median):     495.596 μs               ┊ GC (median):    0.00%
 Time  (mean ± σ):   509.757 μs ±   1.258 ms  ┊ GC (mean ± σ):  0.06% ± 0.02%
```

```
  479 μs           Histogram: frequency by time          513 μs <

 Memory estimate: 7.97 KiB, allocs estimate: 3.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ VoronoiDelaunay.jl ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
BenchmarkTools.Trial: 9658 samples with 1 evaluation.
 Range (min … max):  470.034 μs …   1.939 ms  ┊ GC (min … max): 0.00% … 70.32%
 Time  (median):     499.508 μs               ┊ GC (median):    0.00%
 Time  (mean ± σ):   510.851 μs ± 113.523 μs  ┊ GC (mean ± σ):  1.76% ±  5.82%
```

```
  470 μs           Histogram: frequency by time          592 μs <

 Memory estimate: 308.34 KiB, allocs estimate: 2016.

================================= 10000 points ==================================
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ CGAL.jl ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
BenchmarkTools.Trial: 849 samples with 1 evaluation.
 Range (min … max):  5.697 ms … 118.578 ms  ┊ GC (min … max): 0.00% … 0.99%
 Time  (median):     5.728 ms               ┊ GC (median):    0.00%
 Time  (mean ± σ):   5.879 ms ±   3.874 ms  ┊ GC (mean ± σ):  0.02% ± 0.03%
```

```
  5.7 ms       Histogram: log(frequency) by time      5.98 ms <

 Memory estimate: 78.23 KiB, allocs estimate: 4.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ VoronoiDelaunay.jl ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
BenchmarkTools.Trial: 889 samples with 1 evaluation.
 Range (min … max):  5.378 ms …   7.643 ms  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     5.491 ms               ┊ GC (median):    0.00%
 Time  (mean ± σ):   5.606 ms ± 382.603 µs  ┊ GC (mean ± σ):  1.64% ± 5.01%
```

```
  5.38 ms      Histogram: log(frequency) by time      7.12 ms <

 Memory estimate: 2.91 MiB, allocs estimate: 20016.

73 ================================ 100000 points ================================
   ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ CGAL.jl ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
   BenchmarkTools.Trial: 81 samples with 1 evaluation.
    Range (min … max):  60.715 ms … 168.723 ms  ┊ GC (min … max): 0.00% … 0.56%
    Time  (median):     60.929 ms               ┊ GC (median):    0.00%
    Time  (mean ± σ):   62.344 ms ±  11.980 ms  ┊ GC (mean ± σ):  0.02% ± 0.06%
```

```
  60.7 ms         Histogram: frequency by time         64.5 ms <

 Memory estimate: 781.36 KiB, allocs estimate: 4.
85 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ VoronoiDelaunay.jl ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
   BenchmarkTools.Trial: 77 samples with 1 evaluation.
    Range (min … max):  61.386 ms … 75.334 ms  ┊ GC (min … max): 0.00% … 9.91%
    Time  (median):     64.489 ms              ┊ GC (median):    3.77%
    Time  (mean ± σ):   65.103 ms ±  2.287 ms  ┊ GC (mean ± σ):  4.14% ± 3.48%
```

```
  61.4 ms         Histogram: frequency by time        69.9 ms <

 Memory estimate: 29.00 MiB, allocs estimate: 200016.

97 ================================ 1000000 points ================================
   ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ CGAL.jl ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
   BenchmarkTools.Trial: 8 samples with 1 evaluation.
    Range (min … max):  648.159 ms … 777.243 ms  ┊ GC (min … max): 0.00% … 0.11%
    Time  (median):     650.569 ms               ┊ GC (median):    0.00%
    Time  (mean ± σ):   668.331 ms ±  44.458 ms  ┊ GC (mean ± σ):  0.02% ± 0.04%
```

```
  648 ms           Histogram: frequency by time          777 ms <

 Memory estimate: 7.63 MiB, allocs estimate: 4.
109 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ VoronoiDelaunay.jl ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
   BenchmarkTools.Trial: 6 samples with 1 evaluation.
    Range (min … max):  766.543 ms … 918.534 ms  ┊ GC (min … max):  7.65% … 22.32%
    Time  (median):     831.606 ms               ┊ GC (median):    14.46%
    Time  (mean ± σ):   841.925 ms ±  51.394 ms  ┊ GC (mean ± σ):  15.64% ±  5.09%
```


```
Memory estimate: 290.05 MiB, allocs estimate: 2006420.
```
