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 applications need to employ, to a lesser or greater extent, geometric constraints that condition the geometric models being produced. As an example, consider that of a line that goes through a point and is parallel to another line, or constructing a circle from three points. The specification (and solving) of geometric constraints allows the creation of geometric shapes that, otherwise, would require substantially more complex descriptions. The inclusion, in a programming language, of primitive operations for creating geometric constraints augments the expressiveness of the language and frees the programmer from the specification of otherwise apparently irrelevant details. Said primitive operations could include the definition of incidence relations, parallelism or perpendicularity, distances, angles, etc., to which the various parts in a geometric model must obey. The focus of this work is the creation and implementation, of geometric constraint primitives that facilitate the specification of geometric forms.

Keywords

Algorithmic Design; Geometric Constraints; Geometric Constraint Solving; Parametric CAD; Programming Language.

Resumo

Aplicações modernas de projecto assistido por computador precisam de empregar, em menor ou maior grau, restrições geométricas que condicionam os modelos geométricos produzidos. Como exemplos, considerem-se o de uma linha que atravessa um ponto e é paralela a outra linha, ou o da construção de um círculo recorrendo a três pontos. A especificação (e resolução) de restrições geométricas permite a criação de formas geométricas que, de outro modo, exigiriam descrições substancialmente mais complexas. A inclusão, numa linguagem de programação, de operações primitivas capazes de criar restrições geométricas aumenta a expressividade da linguagem e liberta o programador de especificar detalhes aparentemente irrelevantes. Essas operações podem incluir a definição de relações de incidência, paralelismo ou perpendicularidade, distâncias, e ângulos; às quais o modelo geométrico deve obedecer. O foco deste trabalho consiste na criação e implementação de primitivas de restrições geométricas que facilitam a especificação de formas geométricas.

AML: Não está muito convidativo. Precisa de ser rees-crito.

Palavras Chave

Design Algorítmico; Restrições Geométricas; Resolução de Restrições Geométricas; CAD Paramétrico; Linguagem de Programação.

Contents

List of Figures

List of Tables

List of Listings

Acronyms and Abbreviations

AD Algorithmic Design

AEC Architecture, Engineering, and Construction

API Application Programming Interface

B-Rep Boundary Representation

BIM Building Information Modeling

CAD Computer-Aided Design

CGAL Computational Geometry Algorithms Library

CS Computer Science

CSG Constructive Solid Geometry

CSP Constraint Satisfaction Problem

DSL Domain-specific Language

EPS Encapsulated PostScript

FFI Foreign Function Interface

GC Geometric Constraint

GCS Geometric Constraint Solving

GUI Graphical User Interface

IDE Integrated Development Environment

LEDA Library for Efficient Data Types and Algorithms

PGF Portable Graphics Format

RGC Rhythmic Gymnastics Center

SDK Software Development Kit

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 (AEC) industry.

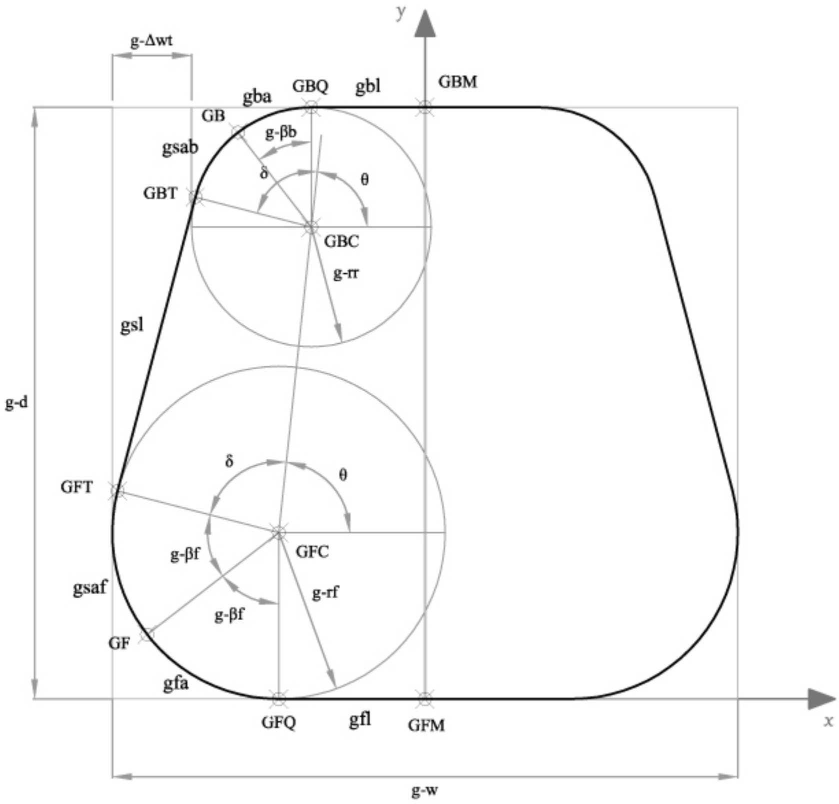
Parametric modeling is used to design with constraints, whereby users express a set of parameters and interdependent operations, establishing restrictions between geometric entities. The resulting geometry can be controlled from input parameters using two computational mechanisms: (1) parametric operations, 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 [2], 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 [3], 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 [4] where the chair's overall shape is controllable by specifying the values for a set of input parameters.

The seat's corners are defined by circles whose respective front and rear radius' length, $g-rf, g-rr$,



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 ($g - \Delta wt$), (4) front radius ($g - rf$), and (5) rear radius ($g - rr$).

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 0 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 some 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 detailed solution proposal, including an explanation of how its implementation can be capable of efficiently handling the specification of GC problems, in chapter 3.

Evaluation A brief plan of the methodology used to evaluate the proposed solution in chapter 4.

Conclusion Concluding remarks that summarize the document, in chapter 5.

1.2 Parametric Operations in CAD

Ivan Sutherland introduced the world to Sketchpad [5] 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 [6] dates from the 1970s. It used a Constructive Solid Geometry (CSG) [7, 8] binary tree, and Boundary Representation (B-Rep) [9] 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. Nearly a decade later, the first system to be acknowledged as a parametric system surfaced [10, 11], enabling 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 lead to the appearance of dimensions and GCs in parametric operations, having GCS in drawings become standard by the early 1990s [12, 13, 14]. Efforts to expand the benefits of constraint solving beyond simple sketches were made, having the majority of some systems 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 [15, 16, 17], 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 [18], 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've 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]:

> 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 [19], 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 [20]. 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, where a subsequent phase elaborates on this, employing algebraic and/or numerical methods. This has become the dominant approach to GCS, also becoming the target of considerable research and development [1].

Degrees of Freedom Analysis The graph's vertices are labeled with represented object's degrees of freedom. 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 [21]. It is further extended by using it along with numerical methods as a fallback if geometric reasoning fails [22].

Latham and Middleditch [23] 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 [24, 25] which can fail in the presence of orientation loops, but such situations are addressed [25] 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[1] 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 [26, 27, 28]. Two different types of constraints are further considered [29, 30]: (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.

1.3.4 Symbolic Methods

Symbolic methods rely on general equation solvers which employ symbolic techniques to triangularize equation systems [31, 32] that emerge from emplying an algebraic approach. A solver built on top of the Buchberger's algorithm is described in [33]; Kondo [34] further reports on a symbolic algebraic method.

These methods are powerful since they can produce generic solutions if constraints are used symbolically, which can be evaluated for a different set of constraint assignments, then producing parameterized solutions. However, solvers are very slow and computations demand a lot of space, usually requiring exponential running time [35].

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

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

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 [5, 36, 37] 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 [38].

Global and guaranteed convergence can be had resorting to the *Homotopy continuation* family of methods [39]. Despite usage in GCS [40, 35], 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 Wu-Ritt method [41, 42] 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 [31]. Chou et al. [43, 44] 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. [45], Bao et al. [46], Moll and Kavraki [47] consider deformations that minimize strain energy; Xu et al. [48] entail surface deformation under area constraints. However, such techniques are rarely integrated with other GCs such as point distance or perpendicularity.

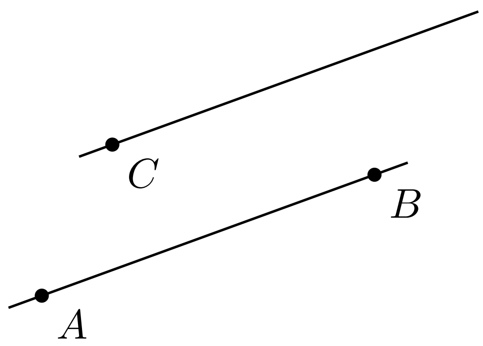
Dynamic Geometry The addition of constraints to a given under-constrained system can make it well-constrained, and such constraints can be seen as parameters when they are dimensional.

Varying their values, different solutions arise, which can be wholly understood as a dynamic geometric configuration. Systems akin to Cinderella [49] can deal with these problems. Further literature exists on these problems from a constraint solving perspective [50].

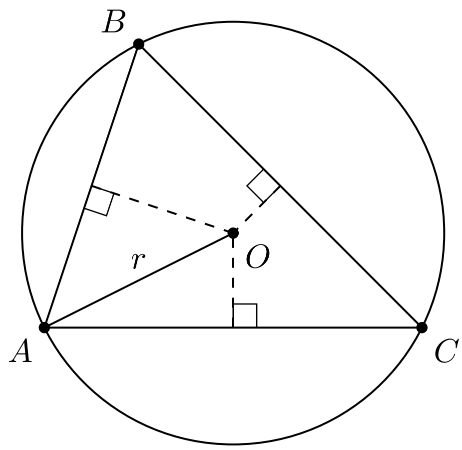
Evolutionary Methods Consist of re-interpreting the problem as an optimization problem, attacking it using genetic, particle-swarm or other evolutionary methods [51, 52].

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. Depictions of the aforementioned models can be seen in Figure 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.



(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.

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 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 Figure 1.2a using TikZ [53] with the `tkz-euclide` 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.

```
\begin{tikzpicture}[rotate=20]
  \tkzDefPoints{0/0/A,3/0/B,1/1/C}
  \tkzDefLine[parallel=through C](A,B) \tkzGetPoint{D}
  \tkzDrawLines[add=.1 and .1](A,B C,D)
  \tkzDrawPoints(A,B,C)
  \tkzLabelPoints(A,B,C)
\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 [54]. In Eukleides, the parallel line L_C can be obtained through the `parallel` function, which takes the line $\overleftrightarrow{AB}$ it is parallel to and the point C it goes through.

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

```
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 $\overleftrightarrow{AB}$.

are the mediators between an edge's vertices, which can be described by (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.

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 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

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

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`.

```
\begin{tikzpicture}
  \tkzDefPoints{0/0/A,1/3/B,4/0/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[above left](B)
  \tkzLabelPoints[below right](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 $\triangle ABC$.

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

```
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 $\triangle ABC$.

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.

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, these approaches 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.

Dubbed AD, this approach consists in the generation of CAD and Building Information Modeling (BIM) models through the specification of algorithmic descriptions [3], 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 model in a much more effortless and efficient manner when compared to direct-manipulation methods [55]. 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.

Such an approach also lead to the creation and integration of programming tools into existing CAD and BIM software such as Grasshopper [56] for Rhino3D [57] or Dynamo [58] for Revit [59]. Some tools, like Rosetta [60], 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 [61] 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 [62]. 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 (ideally nullifying) error propagation throughout the algorithm, and increase the tool's expressive power.

2

Related Work

Contents

In this chapter, we start by exposing and discussing numerical accuracy issues that arise when performing computations with fixed-precision arithmetic in section 2.1. 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. To that end, said libraries provide a series of exact algorithms and data structures.

Secondly, we comparatively analyze a set of GCS-capable programming tools' qualities, such as supported language paradigm, native GCS capabilities, 2D and 3D support; summarized in table 2.1. Of those tools, Eukleides, GeoSolver, and TikZ & PGF are extensively discussed.

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

Finally, we close off with small remarks on VPLs' poorer scalability with increasing project complexity when compared to TPLs, showcasing the Rhythmic Gymnastics Center (RGC) as an example, a building whose roof covering was conceptualized and designed using Grasshopper.

2.1 Robustness

The correctness proofs of nearly all geometric algorithms presented in theoretical papers assumes exact computation with real numbers [63]. 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, given two points $P, Q \in \mathbb{R}^2$, finding the closest point to the origin. The distance between two points can be expressed by

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

To determine the closest point, we compare both points' distance to the origin O. That is, if

$$d(P, O) < d(Q, O)$$

holds, P is the closest to the origin. Otherwise, they are either equidistant or Q is closer. However, applying the square root operation is a step that will further introduce errors in the computation process. 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, despite the limitations with fixed precision arithmetic, and speeding up the process because the square root is a computationally heavy operation. Mei et al. [64] 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) [65]. CGAL is a comprehensive library that employs an exact computation paradigm [66], 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 LEDA [67, 68], and CORE [69] and its successor [70], also deal with robustness problems in geometric computation, offering simpler interfaces when compared to CGAL. However, CGAL arguably remains the *de facto* 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 [71], 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.

Tool	TPL	VPL	Assoc†	Decl‡	Imp*	2D	3D
DesignScript [71]	✓	✗	✓	✗	✓	✓	✓
Eukleides [54]	✓	✗	✗	✓	✓	✓	✗
GeoGebra [72]	✓	✓	✗	✗	✓	✓	✓
GeoSolver [73, 74]	✓	✓	✗	✗	✓	✓	✓
Kaleidoscope¶ [75]	✓	✗	✓	✗	✓	≈	≈
ThingLab [37]	✗	✓	✓	✓	✗	✓	✓
TikZ & PGF [53]	✓	✗	✗	✗	✓	✓	✗

¶ — 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

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

2.2.1 Eukleides

A computer language devoted to elementary plane geometry [54], 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, proving for easy extensibility.

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 [54]. Mainly provisioned with two interpreters with the capability of generating Encapsulated PostScript (EPS) files or produce macros, enabling the embedding of Eukleides figures in LaTeX documents.

Despite heavy support for 2D geometric constructs, as mentioned, it is a tool that focuses on 2D geometry alone, which arguably leads to its primary disadvantage: the lack of 3D geometry support. It is also a TPL, which means that, although 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.

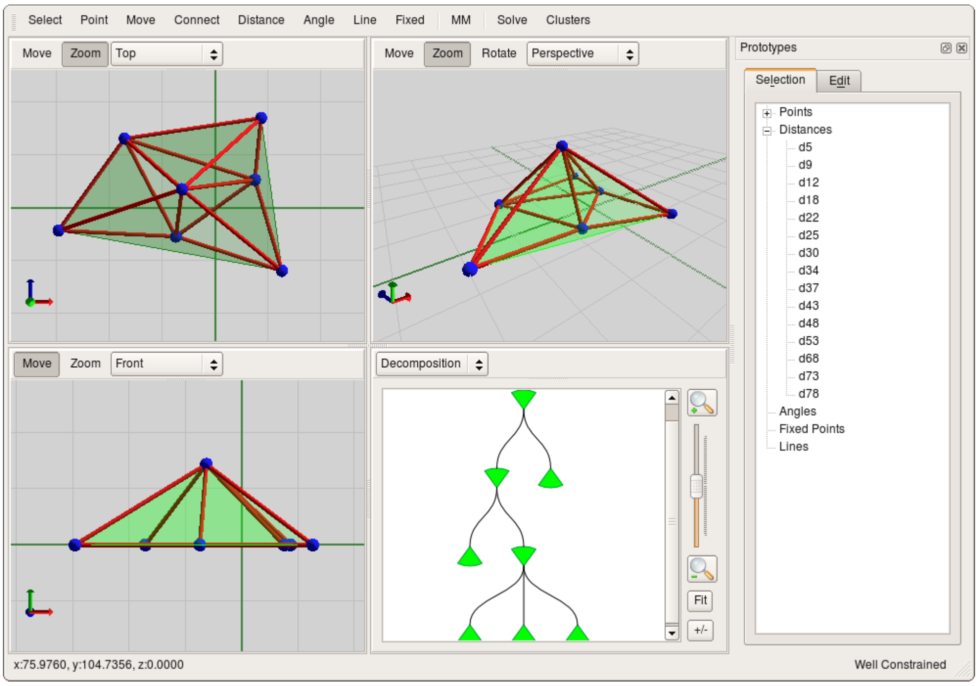
Eukleides has not been actively developed for a couple of years, while TikZ still is. The latter still dominates diagram and graphic production in LaTeX documents, some people going as far as suggesting opting for it instead of using the former [76].

2.2.2 GeoSolver

GeoSolver is an open-source Python package that provides classes and functions for specifying, analyzing, and solving geometric constraint problems [73]. It features a set of 3D geometric constraint problems 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 solution found by GeoSolver is generic and parametric. It can be used to derive a specific solution. Since generic solutions are exponentially hard to find, GeoSolver also allows two different ways of selecting a solution, 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 [74], capable of handling non-rigid clusters contrasting with typical graph constructive-based approaches.

A GUI interactive tool called GCS Workbench [77] (see fig. 2.1) is distributed along with the GeoSolver package. The user can easily edit, analyze and solve geometric constraint problems. The latter features are obviously supported by GeoSolver, and 3D interactivity support comes in the form of pyQt and pyOpenGL. An excellent tool for understanding how a geometric constraint problem is decomposed in GeoSolver, but 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 PhD thesis, TikZ [53], 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 not a 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`, a package superseded by `tkz-euclide` [78]. 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, for example, 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; tools that use TPLs, VPLs, or even a mixture of both approaches.

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

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

Application	Tool	TPL	VPL	Note
AutoCAD [2]	.NET API	✓	✗	Powerful, but very verbose; C# & VB.NET
	ActiveX Automation	✓	✗	Deprecated, bundled separately; VBA
	Visual LISP	✓	✗	IDE; AutoLISP extension
Dynamo Studio Revit [59]	Dynamo [58]	✓	✓	Data flow paradigm; Associative programming support through DesignScript
ArchiCAD [79] Rhinoceros3D [57]	Grasshopper [56]	✓	✓	Data flow paradigm; Rhino SDK access, C# & VB.NET
Rhinoceros3D [57]	Python Scripting	✓	✗	Simple language; Create custom Grasshopper components
	RhinoScript	✓	✗	VBScript based
`Standalone`†	ImplicitCAD [80]	✓	✗	Web hosted; OpenSCAD inspired
	OpenJSCAD [81]	✓	✗	Web hosted; JavaScript
	OpenSCAD [82]	✓	✗	Solid 3D models; Simple DSL
	Rosetta [60]	✓	✗	Portable tool; Multiple front- and back-end support

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

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

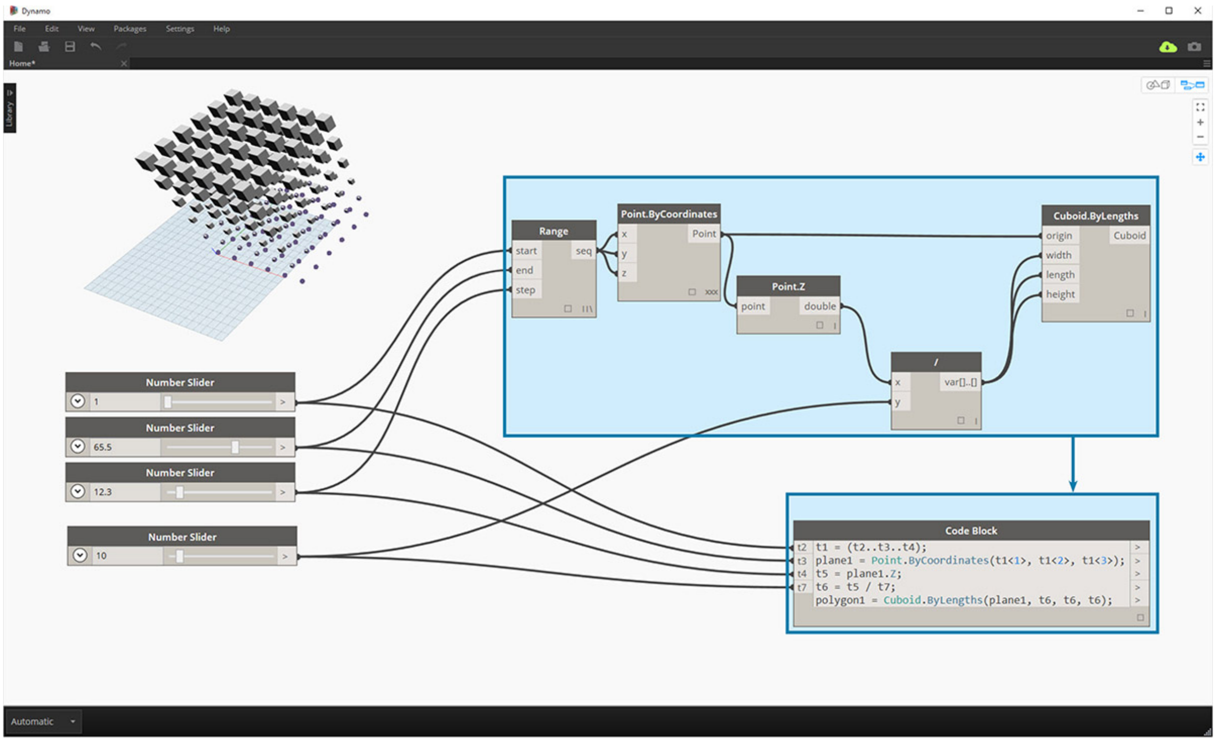
2.3.1 Dynamo

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 [58]. 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. Originally developed my Robert Aish [71], 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).

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



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.

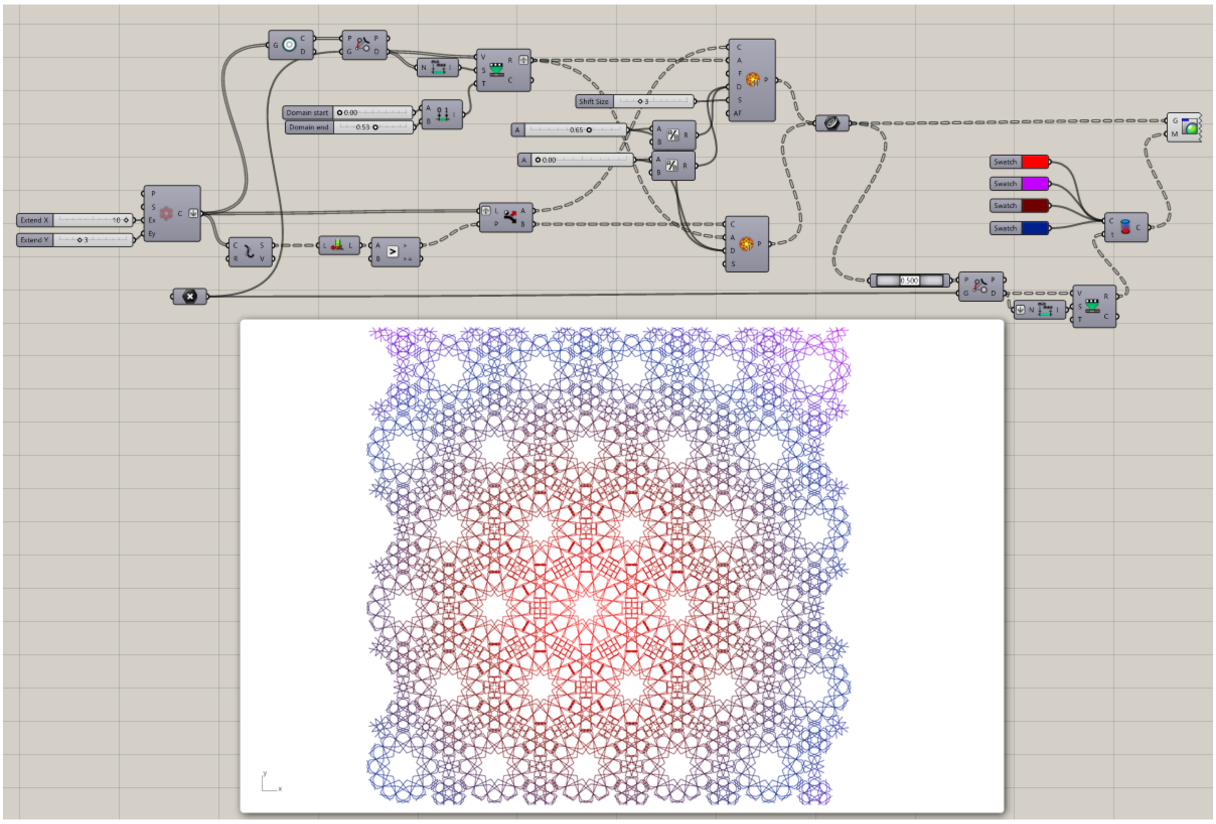
in Dynamo follow an associative paradigm. The user can, however, switch to an imperative paradigm approach instead effortlessly if needed.

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 [56]. In spite of tight integration with Rhino, a CAD application, it is possible to use Grasshopper along with ArchiCAD [79, 83], a BIM tool. Figure 2.3 shows a simple example of a Grasshopper workflow.

It is a closed-source product, designed by David Rutten and developed by McNeel and Associates, Rhino's developers. Its VPL is as simple to use as Dynamo's, which is crucial for users who are not familiar with programming using a TPL. Nonetheless, it offers a TPL alternative by way of custom pro-



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 [84].

grammatic components. Using C# or VB.NET, the user can create custom code components with access to Rhino's Software Development Kit (SDK) and openNURBS [85] within Rhino. Alternatively, through GhPython [86], the user can also write Python code. Unlike 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 they are composable, generating a workflow destined to accomplish a specific task.

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 [55]. Sophisticated modeling workflows tend to become difficult to properly represent, and harder for a human to efficiently interpret when compared to a textual approach.

As an example, consider the Irina Viner-Usmanova RGC, a project developed by TPO Pride[1] over the span of three years, from 2016 to 2019. 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.

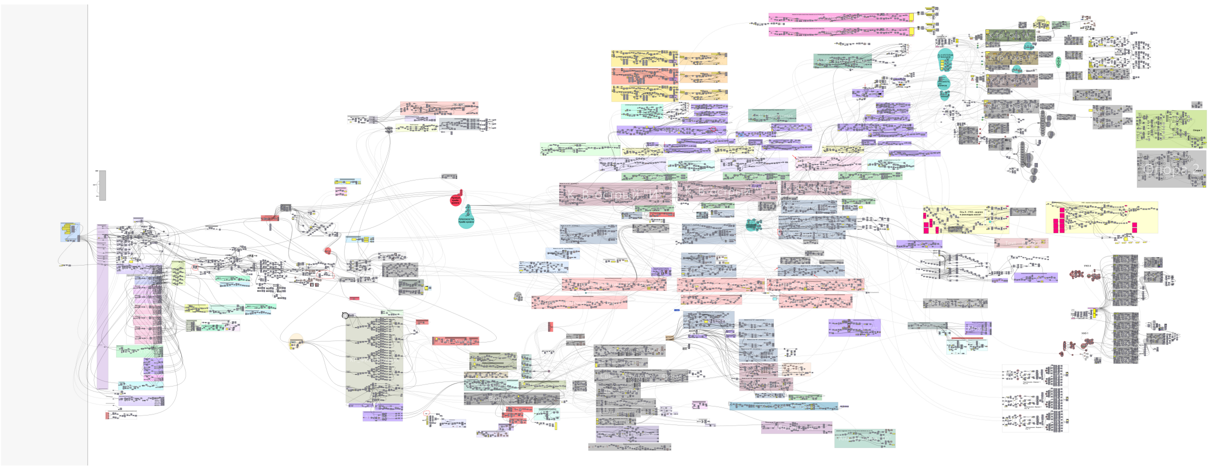
Source: https://www.grasshopper3d.com/photo/rhythmic-gymnastics-center-moscow-russia-5 (Jul 2021)

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. 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 underlying wave's frequency, amplitude, or dampening rate.

Such complex AD projects hide equally, if not more, complex corresponding programs. The final Grasshopper definition of the RGC roof's covering can be seen in fig. 2.5. This further reinforces the

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



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

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

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 as the former rely on text, which is one-dimensional, while the latter rely on boxes and interconnecting wires, which are two-dimensional.

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 robustness issues discussed in section 2.1, effectively yielding inaccurate solutions if precautions aren't 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 these primitives, and, by composing them, elegantly specify the set of geometric constraints necessary in order 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 Computation Geometric 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.

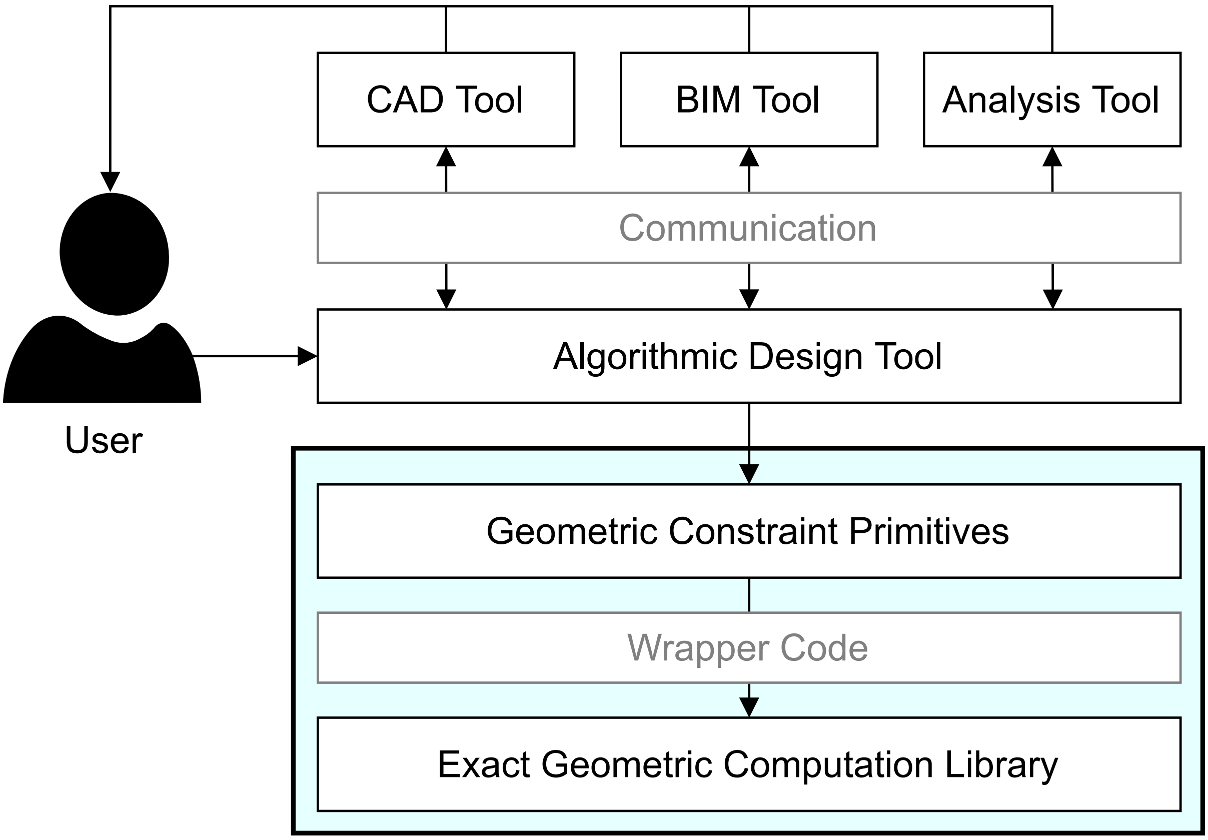


Figure 3.1: General overview of the solution's architecture encapsulated within the blue colored box beneath a depiction of the typical AD workflow.

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 the 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 [87, 88], a text-based tool written in the Julia programming language [89]. Khepri is the successor of another text-based AD tool named Rosetta [60], a tool written in the Racket programming language [90]. It follows that the *Geometric Constraint Primitives* were implemented in the Julia language as well, supported by an *Exact Computation Geometry Library*. The library chosen for the effect was the Computational Geometry Algorithms Library (CGAL) [65], a highly performant and robust geometric library written in the C++ programming language [91].

This language disparity between the *Geometric Constraint Primitives* module and the *Exact Computation Geometry 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 [92] or the Fortran [93] 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.[1] This mechanism can also in turn be leveraged and built upon to interface with other programming languages, e.g., Java, Python, MATLAB, and, the one needed for our particular use-case, C++.[2]

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 providing this abstraction over more primitive functionality, we aimed to provide an easy to understand and utilize set of tools so users can avoid reimplementing it themselves, which is an error-prone process. as levelling 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'll 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`[3] [94]. Finally, we showcase how we leveraged `CGAL.jl` to build the aforementioned *Geometric Constraint Primitives*, a set of functionality that implements specialized yet comprehensible constructive approach solutions to GC problems.

[1] The decision to include such a mechanism at the language's core by the language designers makes it so the language can rapidly evolve by avoiding 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've lacked such a mechanism.

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

[3] 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 Exact Computation Geometric Library

Introduce CGAL to the masses. Showcase example program using basic CGAL constructs and functionality. Not sure where to put notes about the library's complexity, but maybe include explanation of why it made sense to choose CGAL as our library of choice (mature library, tons of research resulting from several PhDs, open source $\implies$ open to contributions, lots of contributors, active development, *et cetera*).

3.1.2 Wrapper Code

Reiterate the language interoperability hurdle, maybe mention briefly there are complications especially when memory management models differ. Showcase Julia's native capabilities of invoking native C++ libraries coupled w/ an example using CGAL. Introduce a library that helps with C++ wrapping and showcase a slightly more complex example, maybe involving classes and the like. Consider demonstrating the ease with which it is possible to wrap things incrementally as, on demand, requests for new features may require algorithms from the library that weren't wrapped. Just like following a recipe, it's as simple as (1) looking at the docs, (2) mapping necessary types and functions, and (3) run Julia (Carefully not mapping *everything*, thought that is what I strived to do with CGAL.jl, but that's another discussion).

3.1.3 Geometric Constraint Primitives

With functionality available on the Julia side of things, call back to the previously formulated examples, potentially elaborating with yet another slightly more complex example. Showcase that some of the functionality that can be implemented with very little effort relying only on functionality already present in mature library alone like CGAL is. I confess I do not know what to discuss in this section, despite feeling like it is the most important one in some regard…

3.2 Trade-offs

Still not sure what to include here, but a section going over a couple of issues circling the monstruous complexities of wrapping a gargantuan library that CGAL proves to be a daunting task which, for simplicity's sake, required hiding and pre-setting a lot of things on the C++ side of things. Contrast this with the yet maturing Julia geometry ecosystem, which is proving to be going somewhere, but it is still relatively young compared to things like CGAL. However, also illustrate that there are geometric Julia packages that would be good candidates for replacing CGAL.

Additionally, explain why an approach using CxxWrap.jl was chosen, requiring an explicit C++ wrapper library to hook into, which requires manual-ish compilation and production, instead of using Cxx.jl, which can be used to inline C++ code within Julia. The former was chosen vs. the latter for what seemed like stability reasons at the time. The CxxWrap.jl approach seemed less complicated despite the extra step of producing a C++ code shim that can then be fed into CxxWrap.jl.

```
#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;
}
```

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.

```c
#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

# julia wrapper function around C function
squared_distance(x₁::Real, y₁::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

# alternative syntax using `@ccall`
squared_distance(x₁::Real, y₁::Real, z₁::Real,
                 x₂::Real, y₂::Real, z₂::Real) =
    @ccall lib.squared_distance(
        x₁::Float64, y₁::Float64, z₁::Float64
      , x₂::Float64, y₂::Float64, z₂::Float64)::Float64

let p = (x=0, y=0, z=0),
    q = (x=3, y=4, z=0)
    @info("Squared distance"
        , p, q
        , squared_distance(p.x, p.y, p.z
                         , q.x, q.y, q.z)) # = 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.

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

// C struct opaquely wrapping CGAL::Point_3<Kernel>
struct Point { double x, y, z; };

// C function to be invoked in Julia using `ccall`
extern "C"
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.

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

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

# julia wrapper function around C function
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

# alternative syntax using `@ccall`
circumcenter(p₁::Point, p₂::Point, p₃::Point) =
    @ccall lib.circumcenter(p₁::Point, p₂::Point, p₃::Point)::Point

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's equivalent to the one specified in C to facilitate data transfer.

```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 3.6: 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.

```julia
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 3.7: Implementation of the parallel lines example illustrated in fig. 1.2a using Khepri alongside our solution backed by `CGAL.jl`.

```
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

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(t -> right_angle(t...)
         , ((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 3.8: Implementation of the circumcenter example illustrated in fig. 1.2b using Khepri alongside our solution backed by `CGAL.jl`. In this particular case, we can leverage CGAL's facilities directly.

4

Evaluation

Contents

4.1 Performance & Correctness

Working title

4.1.1 ConstraintGM

Introduce Fábio's work with ConstraintGM, point out the mishap of feeling overconfident about maxima and how that prompted him, per suggestion, to implement specific/contextual solutions for geometric constraint problems, i.e., what we did, instead of solely relying on a generic algebraic solver. Recall his benchmarks and compare them with identical benchmarks with our solutions. Discuss the possibility that language differences might be the reason behind performance differences alone.

4.1.2 VoronoiDelaunay.jl

Maybe leave the "how easy it is to leverage our approach to get more, both in quantity and complexity, algorithsm from CGAL" discussion for this part and evaluate the potential time it takes to use Voronoi Diagrams from CGAL and either get more information on how long it took to implement VoronoiDelaunay.jl and compare its "correctness" vs. CGAL's version of the algorithm, assuming CGAL's results as a source of truth for correctness. Testing revealed diagrams differed slightly when it came to some edges. My suspicion was the Julia algorithm "gobble" some very very very small triangles, i.e., where the edges are very close together, but it only happens near the outside of the diagram, and, again, a more drastic diagram can be manufactured to showcase this. Maybe read up more on the underlying algorithm that was implemented in VoronoiDelaunay.jl

4.2 Case Studies

Nexus article evaluation, mostly. Maybe a bit more elaborate, maybe a bit less.

4.2.1 Egg

4.2.2 Rounded Trapezoid

4.2.3 Star with Semicircles

4.2.4 Voronoi Diagram

5

Conclusion

TODO: Review this after all is said and done.

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, or through VPLs because of the disproportionate relation between the resulting workflow and respective design complexity. However, working with geometric constraints in TPLs imposes a set of challenges, which can be overcome through the usage of 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, but solvers, in general, have difficulty in identifying specific underlying subproblems for which efficiently computable and robust solutions might be available.

The prior analysis of the set of geometric constraints that must be dealt with, nonetheless, requires certain background knowledge on numerical robustness to mitigate fixed-precision arithmetic issues, such as *roundoff* error accumulation throughout calculation, as well as investigation on how to solve these specific constraint problems. The user will end up having to spend 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 geometrical 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.

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] Autodesk Inc. (1982, Dec.) AutoCAD — CAD software to design anything. Accessed on 23 Dec 2018. [Online]. Available: https://www.autodesk.com/products/autocad

[3] 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

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

[5] 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

[6] 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

[7] A. A. G. Requicha and H. B. Voelcker, "Constructive solid geometry," *CumInCAD*, Nov. 1977.

[8] 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

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

[10] Parametric Technology Corp. (1980) Pro/ENGINEER. Accessed on 27 Nov 2018. [Online]. Available: https://www.ptc.com/en/products/cad/pro-engineer

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

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

[13] 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

[14] 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

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

[16] 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

[17] C. Clarke, "Super models," *Engineer*, vol. 284, pp. 36–38, 2009.

[18] 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

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

[20] 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

[21] 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

[22] 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

[23] 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

[24] 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

[25] 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

[26] 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

[27] 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

[28] 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

[29] 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

[30] 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

[31] 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

[32] 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

[33] 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

[34] 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

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

[36] 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

[37] 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

[38] 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

[39] 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

[40] 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

[41] 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

[42] ——, *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

[43] 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

[44] ——, “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

[45] 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

[46] 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

[47] 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

[48] 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

[49] 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

[50] 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

[51] 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

[52] 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.

[53] T. Tantau, *TikZ & PGF Manual*, 3.0.1 ed., Aug. 29, 2015, accessed on 2 Jan 2019. [Online]. Available: http://sourceforge.net/projects/pgf

[54] 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

[55] 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.

[56] D. Rutten and Robert McNeel and Associates. (2007, Sep.) Grasshopper — algorithmic modelling for rhino. Accessed on 23 Dec 2018. [Online]. Available: https://www.Grasshopper.com/

[57] Robert McNeel and Associates. (1998, Oct.) Rhinoceros 3D — design, model, present, analyze, realize. Accessed on 23 Dec 2018. [Online]. Available: https://www.rhino3d.com

[58] I. Keough and Autodesk Inc. (2012) Dynamo BIM. Accessed on 23 Dec 2018. [Online]. Available: http://dynamobim.org

[59] Revit Technology Corporation and Autodesk Inc. (2000, 2002) Revit — built for building information modelling. Accessed on 23 Dec 2018. [Online]. Available: https://www.autodesk.com/products/revit

[60] 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.

[61] 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.

[62] 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.

[63] 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, accessed on 1 Jan 2019. [Online]. Available: https://doc.cgal.org/4.13/Manual/packages.html#PkgKernel23Summary

[64] 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

[65] CGAL. (2018) Computational Geometry Algorithms Library. Accessed on Dec 31 2018. [Online]. Available: https://www.cgal.org

[66] 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

[67] LEDA. (2017, Apr.) Library for Efficient Data Types and Algorithms. Accessed on Dec 31 2018. [Online]. Available: https://algorithmic-solutions.com/leda

[68] 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

[69] 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

[70] 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

[71] 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

[72] 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

[73] R. Van der Meiden. (2009) GeoSolver. Accessed on 1 Jan 2019. [Online]. Available: https://sourceforge.net/projects/geosolver

[74] 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

[75] 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

[76] O. Christian. (2014, Oct.) Using Eukleides. Question answered by user `LaRiFaRi` on 22 Oct 2014. Accessed on 16 Jun 2021. [Online]. Available: https://tex.stackexchange.com/a/208412/178614

[77] 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

[78] A. Matthes, `tkz-euclide` *manual*, 3.06c ed., Mar. 18, 2020, accessed on 17 Jun 2021. [Online]. Available: https://github.com/tkz-sty/tkz-euclide/blob/f42be71d047861909a109bbf19f818cc076f2064/doc/TKZdoc-euclide.pdf

[79] Graphisoft. (2018, May) ArchiCAD — A 3D architectural BIM software for design & modelling. Accessed on 1 Jan 2019. [Online]. Available: https://www.graphisoft.com/archicad

[80] J. Longtin. (2018) ImplicitCAD. Accessed on 4 jan 2019. [Online]. Available: http://www.implicitcad.org

[81] R. K. Mueller, J. Gay, M. Moissette, and JSCAD Organization. (2019) OpenJSCAD. Accessed on 4 Jan 2019. [Online]. Available: https://openjscad.org

[82] M. Kintel. (2019) OpenSCAD. Accessed on 4 Jan 2019. [Online]. Available: http://www.openscad.org

[83] Graphisoft. (2018) Rhinoceros — Grasshopper Connection. Accessed on 1 Jan 2019. [Online]. Available: https://www.graphisoft.com/archicad/rhino-grasshopper

[84] E. Mottaghi. (2018, Nov.) Parakeet. Accessed on 2 Jan 2019. [Online]. Available: https://www.food4rhino.com/app/parakeet

[85] D. Lear. (2018, Dec.) What is openNURBS? Accessed on Dec 31 2018. [Online]. Available: https://developer.rhino3d.com/guides/opennurbs/what-is-opennurbs

[86] Giulio@mcneel.com. (2017, Feb.) GhPython. Accessed 1 Jan 2019. [Online]. Available: https://www.food4rhino.com/app/ghpython

[87] A. Leitão, "Khepri.jl," Sep. 5, 2018. [Online]. Available: https://github.com/aptmcl/Khepri.jl

[88] 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.

[89] 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

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

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

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

[93] 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

[94] R. Ventura, "CGAL.jl," Oct. 2019. [Online]. Available: https://github.com/rgcv/CGAL.jl