Geometric Constraints in Algorithmic Design

Rui Ventura
rui.ventura@tecnico.ulisboa.pt
Instituto Superior Técnico, Universidade de Lisboa
Lisbon, Portugal

ABSTRACT

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

KEYWORDS

Parametric CAD, Geometric Constraints, Algorithmic Design, Exact Computation, Constructive Geometry

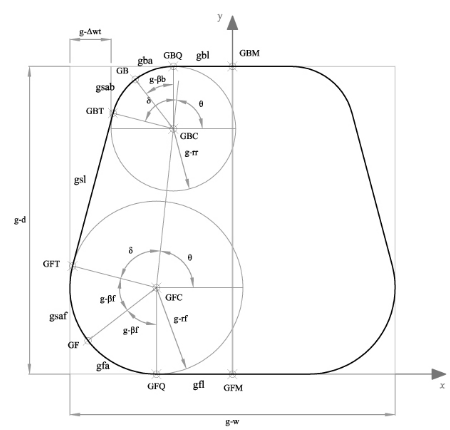
1 INTRODUCTION

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 [3] and are heavily and ubiquitously used across the Architecture, Engineering, and Construction industry.

Parametric modeling is used to design with constraints. Users express a set of parameters and operations, establishing restrictions between geometric entities. The resulting geometry can be controlled from input parameters using two computational mechanisms: (1) parametric operations, and (2) GCS.

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 [29] using either a Textual Programming Language (TPL), a Visual Programming Languages (VPLs), or even a mixture of both.

Dealing with Geometric Constraints (GCs) can still prove to be an arduous task. Take as an example the sketch of a chair seat's outer frame, as seen in figure 1, from a multi-purpose chair generation tool [16] where the chair's overall shape is controllable by specifying the values for a set of input parameters.



Source: Project source code, publicly unavailable.

Figure 1: Sketch of a chair seat's outer frame, defined by several input parameters.

The seat's corners are defined by circles whose respective front and rear radii are obtained by computing distances from which the circles' centers can be obtained. The circles are then connected through outer tangent lines, forming the outer frame of the chair's seat. Some of these operations, such as point distance and tangency, must be handled carefully due to numerical robustness issues that may arise when performing fixed-precision arithmetic computations. This means that, on top of the design process itself, the user must spend additional time identifying and carefully researching how to robustly implement solutions to GCs.

To overcome this problem, 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 Parametric Operations in CAD

Ivan Sutherland introduced the world to Sketchpad [40] in 1963, an interactive 2D CAD program. 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 [35] dates from the 1970s. This system's parametric nature rested in a Constructive Solid Geometry (CSG) [15, 34] tree, which acted as a rudimentary construction step history. The user could modify an operation's parameters' values, reapply the modified history, and generate the newly updated model. Nearly a decade later, Pro/ENGINEER[1] [19] surfaced. It enabled the establishment of relations between the objects' sizes and positions such that a change in a dimension between objects would automatically change affected objects accordingly. GCS soon became standard in drawings by the early 1990s [6, 12, 32]. Efforts to expand the benefits of constraint solving beyond simple sketches were made, some systems having implemented constraint solving in 3D. Improvements from then on focused mostly on robustness and operation variety.

In recent decades, emphasis shifted towards 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 an operation and changing the correct parameter's value. This is a tedious and error-prone process that can lead to undesired side effects. A variety of systems have been developed to mitigate this rigidity [13, 37, 45], but not without drawbacks. Nonetheless, parametric operations will still see continued usage for the foreseeable future.

1.2 Constraints in CAD

Parametric operations allow the user to create geometric objects that satisfy certain constraints *implicitly* imposed on the objects when the user selects the operation they want. GCs, on the other hand, allow the repositioning and scaling of objects so that they satisfy constraints the user *explicitly* imposed on them.

The abstract problem of GCS consists of assigning coordinates to constrained geometric objects such that the constraints they are subject to are satisfied. Otherwise, the solver should report no such assignment can be found.

One of the important features of a solver is its *competence*, which is related to the capability of reporting unsolvability: if no solution for the problem exists and the solver is capable of reporting unsolvability, the solver is deemed fully competent. Since constraint solving is mostly an exponentially complex problem [36], partial competence suffices as long as decent solutions can be found in affordable time and space.

In the context of GCS, it is also important 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; (2) over-constrained, if there are no solutions because of contradictions; or (3) well-constrained, if the number of solutions is finite.

Some of the subjects approached here are briefed in [17]. The following sections present and briefly discuss the most relevant approaches to constraint solving.

1.2.1 Graph-Based Approaches. The problem is translated into a labeled *constraint graph*, where vertices are constrained geometric objects, and edges the constraints themselves. These became the dominant GCS approaches.

1.2.2 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.

1.2.3 Algebraic Approaches. The problem is translated into a system of equations, which is generally nonlinear. 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.

1.2.4 Symbolic Methods. Symbolic methods rely on general equation solvers that employ techniques to triangularize equation systems [9, 11] that emerge from employing an algebraic approach. These methods can produce generic solutions, but solvers are very slow and computation demands a lot of space, usually requiring exponential running time [14].

1.2.5 Numerical Methods. Among the oldest approaches to constraint solving, numerical methods solve large systems of equations iteratively. Methods like Newton iteration work properly if a good approximation of the intended solution can be supplied and the system is not ill-conditioned. 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.

1.2.6 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.

1.3 Geometric Constraint Problem Examples

This section presents two simple examples of geometric models that are defined through GCs, and the respective solutions using algebraic formulation, accompanied by programmatic solutions using TikZ [41]. Depictions of the models can be seen in figure 2. The first problem is that of a parallelism constraint, while the second problem is a circumscription constraint.

1.3.1 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 figure 2a).

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

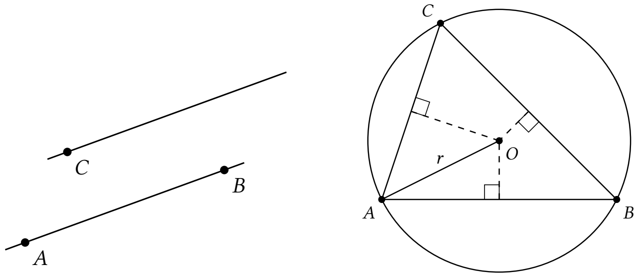
$$P_Q = Q + \lambda\vec{u} \Rightarrow \begin{cases} x = x_Q + \lambda u_x \\ y = y_Q + \lambda u_y \end{cases}, \lambda \in \mathbb{R}. \tag{1}$$

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 vector $\vec{u}$, which can be obtained from $\overleftrightarrow{AB}$.

Listing 1 shows the code used to produce the example shown in figure 2a using TikZ with the `tkz-euclide`[2] LaTeX package.

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

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



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

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

Figure 2: Geometric models defined using GC relations: (a) line parallelism, and (b) circle circumscription.

```
\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: Parallel lines example from figure 2a using `tkz-euclide`.

1.3.2 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 figure 2b).

To draw $\odot O_r$, we need its center O, which results from intersecting the perpendicular bisectors of the triangle's edges; and its radius r, which is the distance from O to any of $\triangle ABC$'s vertices. Said bisectors can be described by equation (1), where P is the midpoint between the vertices, and $\vec{u}$ is a vector normal to the edge. The normal vector $\vec{n}$ is such that, for some vector $\vec{u}$,

$$\vec{u} \cdot \vec{n} = (u_x, u_y) \cdot (v_x, v_y) = u_x v_x + u_y v_y = 0.$$

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.

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. Say we discard the mediator of line $\overleftrightarrow{BC}$. We then require that

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

Every variable is known except for λ_1 and λ_2, but the equation system can be solved in order to determine their values. Finally, we can define O using one of the mediator line equations. Using $P_{M_{AB}}$, for instance, we have

$$O = M_{AB} + \lambda_1 \vec{u_1}.$$

Listing 2 shows the code used to produce the example in Figure 2b using TikZ with the `tkz-euclide` LaTeX package.

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

Listing 2: Circumcenter example from figure 2b using TikZ alongside `tkz-euclide`.

The language used to produce the examples' solutions provides a sensible set of constraint primitives. However, the syntax required for describing the models is outdated, rigid, and may cause confusion. Nonetheless, the underlying ideas can be repurposed and adapted, implementing them in a modern and more expressive language.

1.4 Algorithmic Design

In spite of the improved usability and pervasiveness of parametric features in modern CAD applications, approaches reliant on these tools tend to not scale well with design complexity. Applying changes to existing models becomes cumbersome and users end up wasting time and effort tweaking parameter values, which is an error-prone process.

AD consists in the generation of models through the specification of algorithmic descriptions [29]. The parametric nature of algorithmic specifications already implicitly constrains the model since dependencies within the description change if an ancestor parameter's value is changed.

Such an approach also lead to the creation and integration of programming tools into existing CAD and Building Information Modeling (BIM) software such as Grasshopper[3] for Rhinoceros[4] or Dynamo[5] for Revit[6]. Some tools, like Rosetta [26], offer a distinctly portable solution, enabling the generation of several identical models for a variety of different tools from a single specification [10].

Despite the benefits that come with the integration of AD tools in CAD and BIM software, it is key that these tools also provide an expressive platform to boost user productivity. This means these tools should provide capabilities that make them easier to create

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

complex models and designs [23]. The more expressive the platform is, the better it is with respect to usage, making it easier to learn. This becomes all the more important when generating a constrained geometric model. Thus, the inclusion of GC concepts in such tools would make working with constraints easier, in turn mitigating error propagation throughout the algorithm, increasing the tool's expressive power.

2 RELATED WORK

In this section, we discuss numerical accuracy issues that arise when performing computations with fixed-precision arithmetic. We then proceed to naming some precautions and highlighting alternative methods of obtaining practical solutions.

That is followed by a comparative analysis of a set of GCS-capable programming tools along different dimensions, such as supported language paradigm, native GCS capabilities, 2D and 3D support.

Similarly, we analyze a variety of AD tools. Some of them are integrated within CAD applications while others are standalone applications. These tools and their capabilities are summarized in table 2.

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

2.1 Robustness

The correctness proofs of most all geometric algorithms presented in theoretical papers assumes exact computation with real numbers [7]. However, floating-point numbers are represented with fixed precision in computers, which leads to inaccurate representations of real number. Typically, comparisons must be performed relying on tolerances, i.e., two floating-point numbers a and b are considered *the same* if $|a - b| \leq \epsilon$ for a given ϵ.

When used without care, fixed-precision arithmetic almost always leads to unwanted results due to marginal error accumulation caused by rounding (*roundoff*). 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 allow arbitrary-precision arithmetic with the drawback that operations are slower, however mitigating precision issues.

Several libraries already strive to implement robust geometric computation. One such example is the Computational Geometry Algorithms Library (CGAL) [42]. Moreover, other libraries, such as the Library for Efficient Data types and Algorithms (LEDA) [30], and CORE [20] and its successor [46], can also be leveraged to deal with robustness problems in 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 1.2. Some of them also support an associative programming model, allowing for the propagation of changes made to a variable to others that depended on the former. Table 1 succinctly analyzes tools capable of solving geometric constraints.

Table 1: Table of tools and languages with GCS capabilities.

Tool	TPL	VPL	Assoc†	Decl‡	Imp*	2D	3D
DesignScript [1]	✓	✗	✓	✗	✓	✓	✓
Eukleides [31]	✓	✗	✗	✓	✓	✓	✗
GeoGebra [18]	✓	✓	✗	✗	✓	✓	✓
GeoSolver [43]	✓	✓	✗	✗	✓	✓	✓
Kaleidoscope¶ [27]	✓	✗	✓	✗	✓	≈	≈
ThingLab [5]	✗	✓	✓	✓	✗	✓	✓
TikZ & PGF [41]	✓	✗	✗	✗	✓	✓	✗

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

2.3 Algorithmic Design Tools

As discussed in section 1.4, AD tools have been integrated into several modern CAD and BIM applications, using TPLs, VPLs, or even a mixture of both approaches.

Other tools, like JSCAD[7] and ImplicitCAD[8], are standalone CAD software hosted on the web. Alas, being relatively new, they lack features in comparison to the immense feature-set of applications such as AutoCAD.

Table 2 succinctly summarizes a list of CAD software that supports the usage of a programming language, as well as other AD tools that live detached from existing CAD software.

Table 2: CAD/BIM software with programmatic capabilities and AD software/tools.

Application	Tool	TPL	VPL	Note
AutoCAD	.NET API	✓	✗	Powerful, but very verbose; C# & VB.NET
	ActiveX Automation	✓	✗	Deprecated, bundled separately; VBA
	Visual LISP	✓	✗	IDE; AutoLISP extension
Dynamo Studio Revit	Dynamo	✓	✓	Data flow paradigm; Associative programming support through DesignScript
Archicad Rhinoceros3D	Grasshopper	✓	✓	Data flow paradigm; Rhino SDK access, C# & VB.NET
Rhinoceros3D	Python Scripting	✓	✗	Simple language; Create custom Grasshopper components
	RhinoScript	✓	✗	VBScript based
`Standalone`†	ImplicitCAD	✓	✗	Web hosted; OpenSCAD inspired
	JSCAD	✓	✗	Web hosted; JavaScript
	OpenSCAD	✓	✗	Solid 3D models; Simple domain language
	Rosetta [26]	✓	✗	Portable tool; Multiple front- and back-end support

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

2.4 Visual Programming Scalability

VPL-based approaches suffer from the disproportionate scalability between the code and the respective model's complexity [25]. Sophisticated modeling workflows tend to become difficult to create,

[7]https://openjscad.xyz
[8]https://implicitcad.org

and harder for a human to understand when compared to a textual approach.

As an example, consider the Irina Viner-Usmanova RGC. Figure 3 depicts an outside view of the RGC and its prominent overarching roof covering, the latter conceived using Grasshopper.

Developing a roof covering with such a contour lends itself well to AD. Such complex AD projects tend to require complex AD programs that become overly difficult to develop and understand with VPLs. This disadvantage, however, is mitigated by TPL alternatives which, despite project complexity, scale relatively better than VPLs on account of abstraction mechanisms.

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

Figure 3: Irina Viner-Usmanova RGC, Moscow, Russia.

3 SOLUTION

Despite strides in enhancing performance and efficiency of geometric constraint solving approaches, the core issue lies in the generality of geometric constraint solvers. Instead of delegating the problem to a solver, a more efficient approach would be to provide a specialized solution for said problem.

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. Choosing to implement these in a TPL further avoids VPLs' poor scalability with increasing code complexity.

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 robustness issues. The user can then resort to the implemented primitives, and, by composing them, elegantly specify the set of geometric constraints necessary to produce the idealized model.

Figure 4 shows the typical AD workflow and how the proposed solution could be integrated with the AD tool. The encapsulated modules in the figure represent our solution's components.

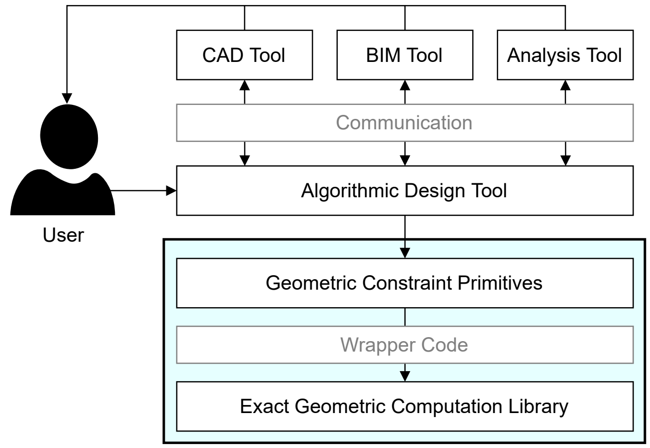


Figure 4: General overview of the solution's architecture integrated in the typical AD workflow.

3.1 Implementation

This section details implementation choices regarding the chosen platforms for realizing the proposed solution. We expand specifically on the concrete components corresponding to the ones within figure 4's light blue rectangle.

The AD tool we have chosen was Khepri[9] [24], a text-based tool written in the Julia programming language [4]. It follows that the *Geometric Constraint Primitives* were implemented in the Julia language as well, supported by an *Exact Geometric Computation Library*. The library chosen for the effect was CGAL [42], a highly performant library written in the C++ programming language [39].

We need to make CGAL available to the Julia language. Fortunately, the Julia language already possesses Foreign Function Interface (FFI) facilities that allow it to invoke functionality within C [22] or Fortran [2] libraries. This mechanism can be built upon to interface with C++.

Overcoming the language interoperability hurdle, we can focus on the *Geometric Constraint Primitives*. These primitives build on top of the functionality available in CGAL, some of which is directly inherited from it. 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`.

3.1.1 Computational Geometry Algorithms Library. CGAL is a library that provides easy access to efficient and reliable geometric algorithms as a C++ library. It is considered the industry's *de facto* standard geometric library. We chose CGAL because of its comprehensiveness and decades of work.

CGAL offers a multitude of data structures and algorithms, such as triangulations, Voronoi diagrams, and convex hull algorithms, to name a few. The library is broken up into three parts [8]: (1) The kernel, which consists of geometric primitive objects and operations on these objects, (2) basic geometric data structures and algorithms, and (3) non-geometric support facilities for debugging and for interfacing CGAL to various visualization tools.

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

Listing 3 showcases an example of a very simple CGAL program, demonstrating the construction of some points and a segment, and performing some basic operations on them.

```cpp
#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: An example CGAL program illustrating object construction and some basic operations.

It is worth noting that floating point-based computation can lead to surprising results. CGAL offers easily interchangeable kernels that provide exact predicates and exact constructions.

However, CGAL is a terribly complex library under the hood, presenting many challenges when it comes to mapping it to the Julia language. Firstly, wrapping C++ with Julia requires additional steps. Secondly, both languages use differing memory management mechanisms. Finally, CGAL abuses C++ templates, making it cumbersome to transparently map functionality.

Fortunately, there are methods and libraries that can help us overcome some of those issues. We demonstrate how we overcame said issues, demonstrating it by reproducing the example in listing 3 in Julia.

3.1.2 From C++ to Julia. Having established CGAL as our *Exact Geometric Computation Library* of choice, we must now overcome the language barrier between Julia and C++. Fortunately, the former possesses FFI mechanisms that can aid us in resolving this issue. Julia provides a special construct named `ccall` that is capable of efficiently calling C and Fortran functions. It is similar to a function call that requires the target function's signature and arguments.

However, we cannot wrap C++ functions directly because C++ compilers mangle function names. We must inhibit the compiler from doing so by preceding the target functions with `extern "C"`. Besides primitive types, it is possible to map Julia `struct`s to C `struct`s to facilitate data transfer.

This strategy, however, does not scale. Though we could incrementally build on this approach, C++ is leaps and bounds more complex than C, which is enough to justify exploring a different approach.

Fortunately, there is a Julia package destined to wrapping C++ code named `CxxWrap.jl`[10]. `CxxWrap.jl` adopts an approach where the user writes the code for the Julia wrapper in C++ and initializes the library on the Julia side with little more than a single instruction. Listing 8 shows the code that wraps the functionality required to reproduce the program program in listing 3 in Julia.

After compiling the wrapper code, we can load it on the Julia side resorting to `CxxWrap.jl`. Listing 9 shows an example bare-bones CGAL Julia module.

As a result, we can devise the program in Listing 4, which is a translation of C++ example in listing 3.

```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 4: The example program as seen in listing 3 written in the Julia programming language using `CGAL.jl`.

Our *Wrapper Code* component expands on this methodology to expose far more functionality, leading to the creation of the package `CGAL.jl` [44], containing the objects and functions we need to build our *Geometric Constraint Primitives*. The following section goes over how we effectively used `CGAL.jl` to implement constructive solutions for GC problems.

3.1.3 Geometric Constraint Primitives. Having overcome the language barrier between C++ and Julia, we can build our GC primitives on top of `CGAL.jl`. Our implementation follows a constructive

[10] https://github.com/JuliaInterop/CxxWrap.jl

approach where the production of geometry can be done solely resorting to a straightedge and a compass. This makes programs easier to understand and manually reproduce.

The following sections revisit of our initially formulated example problems from section 1.3.

Parallel lines. Revisiting our earlier examples, we now showcase implementations for those problems using our solution, accompanied by the Khepri AD tool. Listing 5 shows a solution to the "parallel lines" problem introduced in section 1.3.1.

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

begin
    backend(autocad); delete_all_shapes()

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

Listing 5: Implementation of the parallel lines example illustrated in figure 2a using Khepri alongside our solution.

The highlighted `parallel` function takes a line segment `l` and a point `p` and creates a new line segment starting at point `p` with the same length as `l` , obtained using `CGAL.jl`'s `to_vector` function. Figure 5 illustrates the program's output in AutoCAD.

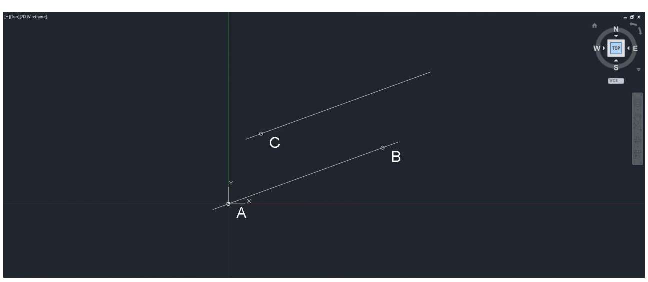


Figure 5: Parallel lines example using our solution, visualized in AutoCAD.

Circumcenter. We initially solved the circumcenter problem by intersecting triangle sides' bisectors. We can still approach the problem that way, defining a `circumcenter` function similar to the one in listing 6.

However, this functionality is already present in CGAL. This is a perfect demonstration of our approach's benefits regarding repurposing a comprehensive library with plenty of functionality.

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

Listing 6: Initial implementation of `circumcenter`.

Listing 7 illustrates a solution to the "circumcenter" problem using CGAL's `circumcenter` function. The program's output can be seen in figure 6.

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

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

begin
    backend(autocad); delete_all_shapes()

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

Listing 7: Implementation of the circumcenter example illustrated in figure 2b using Khepri alongside our solution.

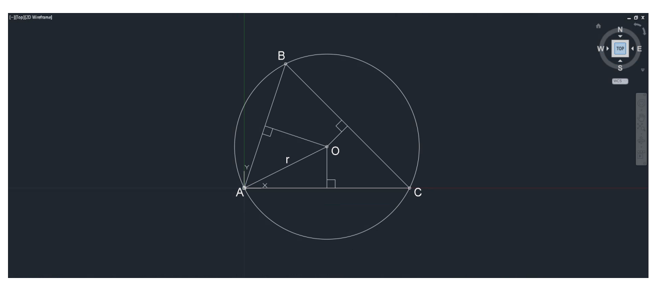


Figure 6: Circumcenter example using our solution, visualized in AutoCAD.

3.2 Trade-offs

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

CGAL is a highly generic library, abusing C++ templates. Although its design makes usage an elegant experience, the same cannot be said when trying to wrap its constructs to another language. Luckily, `CxxWrap.jl` helps us overcome this.

Regarding our wrapper code, we are still mapping CGAL types in an opaque fashion, fixing the kernel on the C++ side.[11] Ideally, objects would be parametric, and the kernel mapped to Julia as well. As a compromise, `CGAL.jl` fixed a kernel that provides exact predicates with inexact constructions, favoring performance over some robustness loss. Nonetheless, in practical terms, it suffices for our case.

As an alternative to wrapping CGAL, we could have explored other options in the still growing Julia ecosystem. Some work looks promising,[12] but not only are some libraries still catching up, it is also highly unlikely they will ever meet the quality of CGAL.

Lastly, some of our GC primitives employ some computation that can lead to robustness loss as well. We tend to typically avoid those computations, postponing them as much as possible. Be that as it may, we are bound with some robustness loss regardless. At this point, it can only be mitigated by also ensuring the quality of the inputs given to our primitives, and that is reliant on the user.

4 EVALUATION

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

Firstly, we benchmark our solution's performance by comparing it to a similar project called ConstraintGM [33].

Secondly, we showcase two different case studies, focusing on solving GC problems by adopting two different approaches: (1) an analytic approach, one programming naturally begs for, and (2) a constructive approach, adding an abstraction over the former. We aim to show the latter produces programs that are both easier to understand and to reproduce.

Finally, we explore our approach's potential for obtaining more complex geometric algorithms vs. re-implementing a version of said algorithms from scratch. Specifically, we repurpose CGAL's 2D Delaunay Triangulation and Voronoi Diagram algorithms, comparing them to a Julia implementation of the algorithms provided by the `VoronoiDelaunay.jl`[13], package. Additionally, we estimate the effort it took to extract the algorithms from CGAL and compare it to the effort it took to develop the ones in `VoronoiDelaunay.jl`.

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

- AMD Ryzen™ 5 3500U CPU @ 2.1GHz[14];
- 1×16GB SO-DIMM of DDR4 RAM @ 2400MT/s.
- Arch Linux™[15] x86 64-bit, Linux® Kernel 5.12.15-zen1[16]

[11] There are other projects that attempt to make CGAL available in other language that resort to this same trick. See https://github.com/scikit-geometry/scikit-geometry and https://github.com/CGAL/cgal-swig-bindings.
[12] https://github.com/JuliaGeometry
[13] https://github.com/JuliaGeometry/VoronoiDelaunay.jl
[14] Base clock frequency. Can boost up to 3.7GHz.
[15] https://archlinux.org
[16] https://github.com/zen-kernel/zen-kernel/tree/v5.12.15-zen1

4.1 ConstraintGM

ConstraintGM is a domain-specific language developed with the goal of tackling GC problems using the Racket TPL. This solution blindly relied on Maxima [28].

This approach came at a grave performance cost for two reasons: (1) the communication between ConstraintGM and Maxima was slow, and (2) Maxima is a *generic* solver. The considerable performance penalty of this approach is hard to justify in the case of simple geometric problems. This lead to the implementation of some GC problem solutions, creating the Geometry Functions Library (GFL).

The project's benchmark involved three different GC problems focused around object intersection, namely (1) line-line intersection, (2) circle-line intersection, and (3) circle-circle intersection.

We measured real execution time instead of CPU time, both for ConstraintGM and for our solution, and plotted the results in figure 7. Observing the results, we can see the disparity between the approach reliant on Maxima when compared to both the GFL and our solution, which was to be expected.

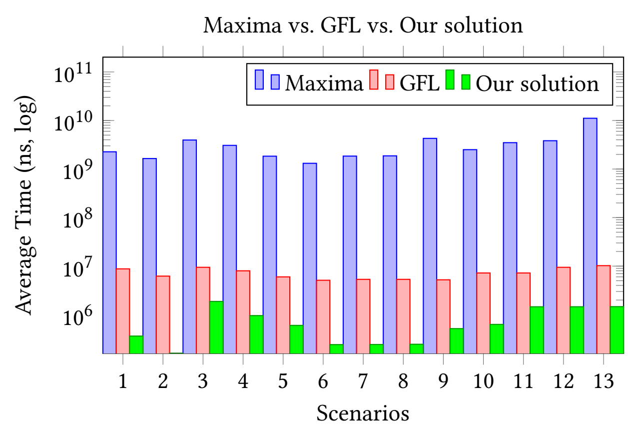


Figure 7: ConstraintGM benchmark results in a Y-bar plot.

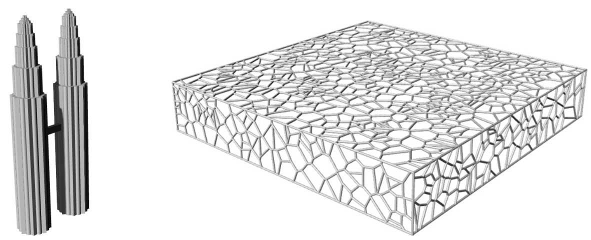
Furthermore, we can see our solution outdoes ConstraintGM's GFL. This is most likely due to the fact that we are relying on CGAL, a library implemented in C++. The latter is notoriously known for being a high-performance language, considerably outperforming Racket in a series of benchmarks. Nevertheless,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

In conclusion, our solution proves capable and performant, having surpassed ConstraintGM's GFL by an entire order of magnitude on average.

4.2 Case Studies

In this section, we aim to demonstrate our solution when applied to two different case studies, each presenting a parametric geometric shape: a star with semicircles, and a Voronoi diagram. Each case, illustrated in figure 8, was inspired by an existing design: (1) César Pelli's Petronas tower section, and (2) PTW Architects' Beijing National Aquatics Center. These problems were solved employing both an *analytic* approach, an approach TPLs naturally demand,

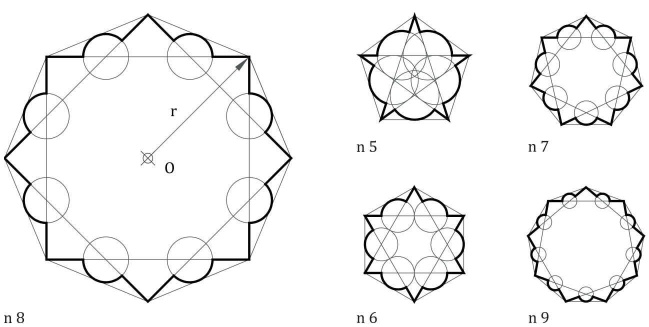
and a *constructive* approach, the one made possible by relying on our solution.



(a) Islamic towers. **(b) Voronoi cage.**

Figure 8: Case study designs inspired by César Pelli's Petronas Twin Towers (a), and PTW Architects' Beijing National Aquatics Center (b).

4.2.1 Star with Semicircles. The first case study is a star shape with semicircles, inspired by César Pelli's Petronas tower floor plan. The contour of the Petronas tower floor plan is formed by two overlapping congruent squares, forming an octagram, and by eight circles each centered on one of the eight intersection points and tangent to the bounding octagon. This shape can be generalized to a parametric shape, shown in (figure 9a). Variations are illustrated in figure 9.



(a) Parametrization. **(b) Shape variations.**

Figure 9: Star with semicircles problem: (a) shows our parametrization of the star which can be used to generate shape variations, some of them shown in (b).

Both analytic and constructive solutions are based on computing one side of the star, composed of the line segment $\overline{V_1 I_1}$, the arc centered on O_1 from I_1 to I_2, with radius r_1, and the line segment $\overline{I_2 V_2}$ (figure 10).

The *analytic solution* is described below.

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

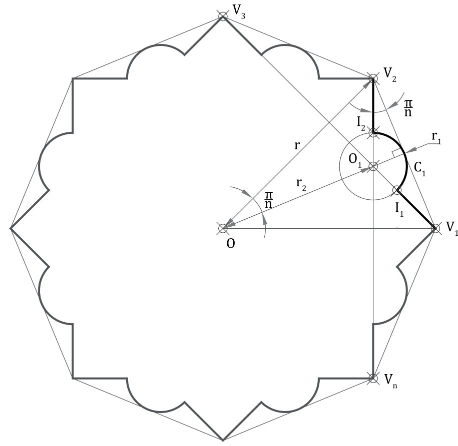


Figure 10: Analytic and constructive solutions to the star with semicircles problem.

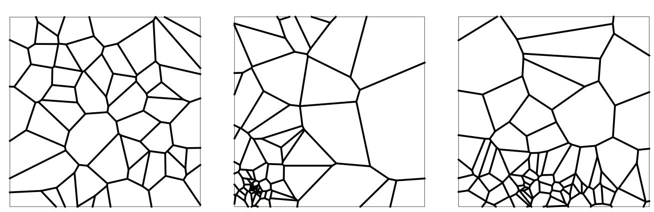
The *constructive solution* is described below. It uses two primitives from our solution, namely `intersection` and `tangent_circle`.

(1) $O_1 = \text{intersection}\left(\overline{V_1 V_3}, \overline{V_2 V_n}\right)$
(2) $C_1 = \text{tangent}_{\text{circle}}\left(O_1, \overline{V_1 V_2}\right)$
(3) $P, r_1 = C_1$
(4) $I_1 = \text{intersection}\left(\overline{V_1 V_3}, C_1\right)$
(5) $I_2 = \text{intersection}\left(\overline{V_2 V_n}, C_1\right)$

Achieving the equations in the *analytic solution* is not a straightforward task. It is also unclear how those equations were derived. By contrast, in the *constructive solution*, all the steps are clearly externalized, which makes it much more comprehensible.

4.2.2 Voronoi Diagram. Our second case study is that of Voronoi diagrams, which are used in a variety of design fields. For instance, several facade designs exhibit a Voronoi appearance, such as PTW Architects' Beijing National Aquatics Center, ARM Architecture's Melbourne Recital Centre, and Hassell's Alibaba Headquarters.

Figure 11 shows three Voronoi diagrams generated from entirely randomly distributed points, from random points with one attractor point, and from random points with one attractor line.



(a) **(b)** **(c)**

Figure 11: Voronoi diagram problem: (a) is entirely random, (b) adds an attractor point, and (c) adds an attractor line.

Both the analytic and constructive methods focus on computation of a vertex relies on the computation of the *circumcenter* of a triangle, for instance, triangle $\triangle P_1P_2P_3$ (figure 12).

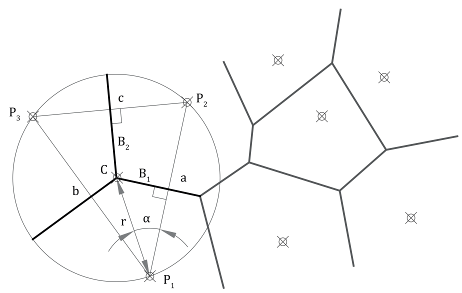


Figure 12: Analytic and constructive approaches to computing a Voronoi vertex.

One possible *analytic solution* is based on the *circumradius* formula. The circumcenter C can then be easily computed by a translation from P_1 following the angle α.

(1) $a, b, c = \|P_2 - P_1\|, \|P_3 - P_1\|, \|P_3 - P_2\|$
(2) $s = \frac{a+b+c}{2}$
(3) $A = \sqrt{s(s-a)(s-b)(s-c)}$
(4) $r = \frac{abc}{4A}$
(5) $\alpha = \arccos \frac{a}{2r}$
(6) $C = P_1 + (r, \angle\alpha)$

The *constructive solution* computes the circumcenter, directly provided by our *Geometric Constraint Primitives*.

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

The circumcenter is only a sub-problem of the generation of a Voronoi diagram. We first need to build a Delaunay triangulation. Then, we can apply the `circumcenter` to find the Voronoi vertices and draw the diagram's edges.

Implementing this functionality from scratch is a demanding and error-prone task. Fortunately, CGAL already has an algorithm that produces Voronoi diagrams. This algorithm was made available in `CGAL.jl`, and, thus, it is also available in our solution. The final section of the evaluation goes over how we can repurpose this algorithm as a side effect of integrating such a comprehensive library as CGAL.

4.3 Voronoi Diagrams Extended

In the previous section, we left the problem of Voronoi Diagrams partially unresolved. This section expands on it by repurposing CGAL's version of the Voronoi Diagram algorithm [21] and comparing it with a native Julia implementation of the algorithm described in [38], provided by the `VoronoiDelaunay.jl`[17] package. We estimate the effort required to obtain either implementation, measuring Delaunay Triangulation construction performance, and compare the outputs of both algorithms.

Wrapping CGAL's Voronoi diagram algorithm follows a similar process to that used by our solution's *Wrapper Code* component. Requiring bare minimal C++ knowledge and following reference documentation as if it were a recipe book, it may take no more than a full day to obtain the necessary functionality.

The algorithm present in `VoronoiDelaunay.jl` is the result of an immense body of research [38]. It is safe to say that it took more than a full day to obtain a robust implementation, requiring interpretation and understanding of the approach described in the [38] since there is no explicit algorithm listed.

Regarding both algorithms' performance, figure 13 shows the results of building Delaunay Triangulations by batch inserting several powers-of-ten sets of points.

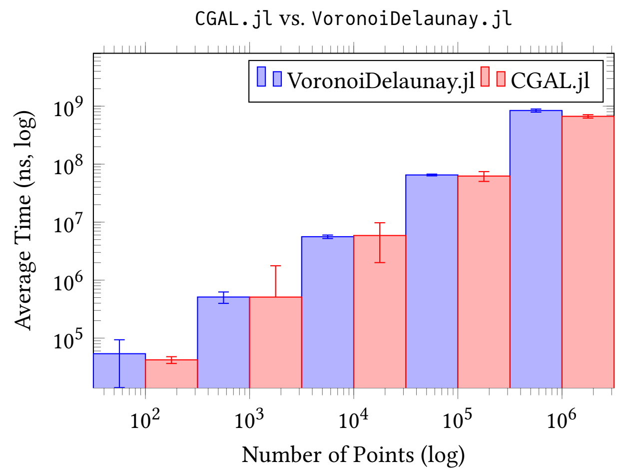


Figure 13: Delaunay Triangulation benchmark results.

Results are pretty identical. However, we see CGAL's variant of the algorithm beating `VoronoiDelaunay.jl`'s by a relatively small margin.

Finally, we take a look at the output Delaunay Triangulations and their respective Voronoi Diagrams, produced by both implementations. Figure 14 illustrates two plots of the output meshes: on the left, the Delaunay Triangulations, and on the right, the respective dual Voronoi Diagrams.

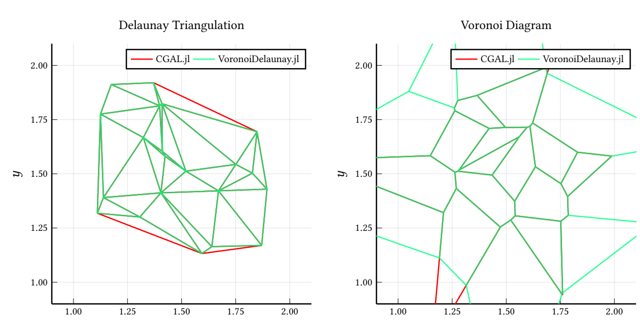


Figure 14: Delaunay Triangulations (on the left) and Voronoi Diagrams (on the right) produced both by `CGAL.jl` and `VoronoiDelaunay.jl`.

Surprisingly, the triangulations are not the same, explaining the divergence in the respective Voronoi Diagrams. Simpler point

[17] https://github.com/JuliaGeometry/VoronoiDelaunay.jl

distributions illustrate this disparity further. Figure 15 shows this scenario blatantly.

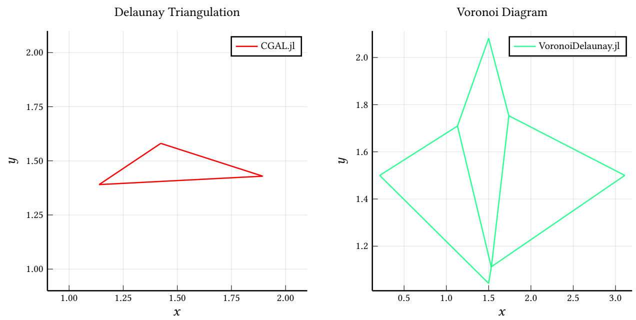


Figure 15: CGAL's algorithm produced a triangle (on the left) while `VoronoiDelaunay.jl` did not. Consequently, CGAL did not produce a (finite) Voronoi Diagram while `VoronoiDelaunay.jl` did (on the right).

It is not clear which of the implementations is wrong. Though, given that CGAL is a more mature library, it is probably safe to assume that the implementation in `VoronoiDelaunay.jl` has issues.

Summarily, re-implementations of complex algorithms from scratch are very likely to produce erroneous results, paling in comparison to more mature alternatives that may be repurposed. Though new implementations may show potential, it will be hard to compete with established battle-tested software.

5 CONCLUSION

The generation of highly constrained sophisticated designs is not viable through usage of interactive interfaces due to rigidity in the manipulation of existing models in order to generate multiple variants. AD. Even then, VPLs suffer from 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. Alas, solvers, in general, have difficulties identifying specific underlying subproblems for which efficiently computable and robust solutions might be available.

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

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

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

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

Future Work

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

This work focused exclusively on constrained geometry bound to the $\mathbb{R}^2$ Euclidean space, i.e., the 2D plane. There is still much work to be done researching problem solutions that encompass 3D space as well. Elevating a dimension means the solutions to problems once formulated in the 2D plane are no longer applicable in 3D space for some problems may now be under-constrained. As an example, our solution for the circumcenter problem, exemplified in section 1.3.2, would no longer work in 3D space. A line's bisector in 3D space is a plane, and noncoplanar nor parallel plane intersection results in a line. To obtain the actual circumcenter, one would additionally, for example, have to intersect the resulting line with the plane the circumscribed triangle sits on.

ACKNOWLEDGMENTS

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

REFERENCES

[1] Robert Aish. 2011. DesignScript: Origins, Explanation, Illustration. In *Computational Design Modelling*, Christoph Gengnagel, Axel Kilian, Norbert Palz, and Fabian Scheurer (Eds.). Springer, Berlin, Heidelberg, Berlin, Germany, 1–8. https://doi.org/10.1007/978-3-642-23435-4_1

[2] 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. 1957. The FORTRAN Automatic Coding System. In *Western Joint Computer Conference: Techniques for Reliability (IRE-AIEE-ACM '57 (Western))*. Association for Computing Machinery, New York, NY, USA, Los Angeles, California, 188–198. https://doi.org/10.1145/1455567.1455599

[3] Bernhard Bettig and Christoph M. Hoffmann. 2011. Geometric Constraint Solving in Parametric CAD. *Journal of Computing and Information Science in Engineering* 11, 2 (June 14, 2011), 021001. https://doi.org/10.1115/1.3593408

[4] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. 2017. Julia: A fresh approach to numerical computing. *SIAM Review* 59, 1 (2017), 65–98. https://doi.org/10.1137/141000671

[5] Alan Borning. 1989. The Programming Language Aspects of ThingLab, a Constraint-oriented Simulation Laboratory. In *Readings in Artificial Intelligence and Databases*. Morgan Kaufmann, San Francisco, California, USA, 480–496. https://doi.org/10.1016/B978-0-934613-53-8.50036-4

[6] William Bouma, Ioannis Fudos, Christoph M. Hoffmann, Jiazhen Cai, and Robert Paige. 1995. Geometric Constraint Solver. *Computer-Aided Design* 27, 6 (1995), 487–501. https://doi.org/10.1016/0010-4485(94)00013-4

[7] Hervé Brönnimann, Andreas Fabri, Geert-Jan Giezeman, Susan Hert, Michael Hoffmann, Lutz Kettner, Sylvain Pion, and Stefan Schirra. 2018. 2D and 3D Linear Geometry Kernel. In *CGAL User and Reference Manual* (4.13 ed.). CGAL Editorial Board. https://doc.cgal.org/4.13/Manual/packages.html#PkgKernel23Summary

[8] Hervé Brönnimann, Andreas Fabri, Geert-Jan Giezeman, Susan Hert, Michael Hoffmann, Lutz Kettner, Sylvain Pion, and Stefan Schirra. 2021. 2D and 3D Linear Geometry Kernel. In *CGAL User and Reference Manual* (5.3 ed.). CGAL Editorial Board. https://doc.cgal.org/5.3/Manual/packages.html#PkgKernel23

[9] Bruno Buchberger. 1995. Gröbner Bases: An Algorithmic Method in Polynomial Ideal Theory. *Multidimensional Systems Theory and Applications* (1995), 89–127. https://doi.org/10.1007/978-94-017-0275-1_4

[10] Renata Castelo-Branco and António Leitão. 2017. 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*, Antonio Fioravanti, Stefano Cursi, Salma Elahmar, Silvia Gargar, Gianluigi Loffreda, Novembri, Gabriale, and Armando Trent (Eds.), Vol. 1. Faculty of Civil and Industrial Engineering, Sapienza University of Rome, Rome, Italy, 729–738.

[11] Shang-Ching Chou. 1988. An Introduction to Wu's Method for Mechanical Theorem Proving in Geometry. *Journal of Automated Reasoning* 4, 3 (Sept. 1988), 237–267. https://doi.org/10.1007/BF00244942

[12] J. Chung and M. Schussel. 1990. Technical Evaluation of Variational and Parametric Design. *Computers in Engineering* 1 (1990), 289–298.

[13] C Clarke. 2009. Super Models. *Engineer* 284 (2009), 36–38.

[14] Cassiano Berenguer Durand. 1998. *Symbolic and Numerical Techniques for Constraint Solving*. Ph.D. Dissertation. Purdue University.

[15] James D. Foley, Foley Dan Van, Andries van Dam, Steven K. Feiner, John F. Hughes, Edward Angel, and J. Hughes. 1996. *Computer Graphics: Principles and Practice*. Addison-Wesley systems programming series, Vol. 12110. Addison-Wesley Professional. https://books.google.pt/books?id=-4ngT05gmAQC accessed on 16 Jun 2021.

[16] Sara Garcia. 2012. *ChairDNA*. https://chairdna.wordpress.com accessed on 6 Jan 2019.

[17] Cristoph M. Hoffmann and Robert Joan-Arinyo. 2005. A Brief on Constraint Solving. *Computer-Aided Design and Applications* 2, 5 (2005), 655–663. https://doi.org/10.1080/16864360.2005.10738330

[18] Markus Hohenwarter and Karl Fuchs. 2004. Combination of Dynamic Geometry, Algebra and Calculus in the Software System GeoGebra. In *Computer algebra systems and dynamic geometry systems in mathematics teaching conference*. 1–6. https://www.researchgate.net/publication/228398347_Combination_of_dynamic_geometry_algebra_and_calculus_in_the_software_system_GeoGebra

[19] Wassim Jabi. 2013. *Parametric Design for Architecture* (1 ed.). Laurence King Publishing, London.

[20] Vijay Karamcheti, Chen Li, Igor Pechtchanski, and Chee Yap. 1999. A Core Library for Robust Numeric and Geometric Computation. In *Proceedings of the Fifteenth Annual Symposium on Computational Geometry (SCG '99)*. Association for Computation Machinery, New York, NY, USA, Miami Beach, Florida, USA, 351–359. https://doi.org/10.1145/304893.304989

[21] Menelaos Karavelas. 2021. 2D Voronoi Diagram Adaptor. In *CGAL User and Reference Manual* (5.3 ed.). CGAL Editorial Board. https://doc.cgal.org/5.3/Manual/packages.html#PkgVoronoiDiagram2

[22] Brian W. Kernighan and Dennis M. Ritchie. 1988. *The C Programming Language* (2 ed.). Prentice Hall Professional Technical Reference.

[23] António Leitão. 2014. 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)*, Ning Gu, Shun Watanabe, Halil Erhan, Hank Haeusler, Weixin Huang, and Ricardo Sosa (Eds.). Kyoto Institute of Technology, Kyoto, Japan, 575–584.

[24] António Leitão, Renata Castelo-Branco, and Guilherme Santos. 2019. Game of Renders: The Use of Game Engines for Architectural Visualization. In *Intelligent & Informed: Proceedings of the 24th CAADRIA Conference*, Matthias Hanks Haeusler, Marc Aurel Schnabel, and Tomohiro Fukuda (Eds.), Vol. 1. Victoria University of Wellington, Wellington, New Zealand, 655–664.

[25] António Leitão, Rita Fernandes, and Luís Santos. 2013. 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, 235–238.

[26] José Lopes and António Leitão. 2011. 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)*, Joshua Taron, Vera Parlac, Brank Kolarevic, and Jason Johnson (Eds.). The University of Calgary, Banff, Canada, 196–203.

[27] Gus Lopez, Bjorn Freeman-Benson, and Alan Borning. 1994. Kaleidoscope: A Constraint Imperative Programming Language. In *Constraint Programming (NATO ASI F, Vol. 131)*, Brian Mayoh, Enn Tyugu, and Jaan Penjam (Eds.). Springer, Berlin, Heidelberg, 313–329. https://doi.org/10.1007/978-3-642-85983-0_12

[28] Maxima. 2021. Maxima, a Computer Algebra System. https://maxima.sourceforge.io

[29] Jon McCormack, Alan Dorin, and Troy Innocent. 2004. Generative Design: A Paradigm for Design Research. In *Futureground — DRS International Conference 2004*, J. Redmond, D. Durling, and A. de Bono (Eds.). Melbourne, Australia. https://dl.designresearchsociety.org/drs-conference-papers/drs2004/researchpapers/171

[30] Kurt Mehlhorn and Stefan Näher. 1989. LEDA: A Library of Efficient Data Types and Algorithms. In *Mathematical Foundations of Computer Science 1989 (Lecture Notes in Computer Science, Vol. 379)*, Antoni Kreczmar and Grazyna Mirkowska (Eds.). Springer, Berlin, Heidelberg, Kozubnik, Porąbka, Poland, 88–106. https://doi.org/10.1007/3-540-51486-4_58

[31] Christian Obrecht. 2010. *ΕΥΚΛΕΙΔΗΣ — The Eukleides Manual* (1.5.3 ed.). http://www.eukleides.org/files/eukleides.pdf accessed on 17 Jun 2019.

[32] J. C. Owen. 1991. Algebraic Solution for Geometry from Dimensional Constraints. In *Proceedings of the First ACM Symposium on Solid Modeling Foundations and CAD/CAM Applications (SMA '91)*. Association for Computing Machinery, New York, NY, USA, Austin, Texas, USA, 397–407. https://doi.org/10.1145/112515.112573

[33] Fábio Pinheiro. 2016. *Modelação Geométrica com Restrições*. Master's thesis. Instituto Superior Técnico, Universidade de Lisboa. https://fenix.tecnico.ulisboa.pt/cursos/meic-a/dissertacao/1691203502342199

[34] Aristides A. G. Requicha and Herbert B. Voelcker. 1977. *Constructive Solid Geometry*. Technical Report 25. University of Rochester, Rochester, New York. http://hdl.handle.net/1802/26358

[35] Aristides G. Requicha. 1980. Representations for Rigid Solids: Theory, Methods, and Systems. *ACM Computing Survey* 12, 4 (Dec. 1980), 437–464. https://doi.org/10.1145/356827.356833

[36] Francesca Rossi, Peter van Beek, and Toby Walsh. 2006. *Handbook of Constraint Programming*. Elsevier.

[37] Stephen Samuel. 2006. CAD Package Pumps Up the Parametrics. *Machine Design* 78, 16 (Aug. 24, 2006), 82–84.

[38] Volker Springel. 2010. *E pur si muove*: Gallilean-invariant cosmological hydrodynamical simulations on a moving mesh. *Monthly Notices of the Royal Astronomical Society* 401, 2 (Jan. 2010), 791–851. https://doi.org/10.1111/j.1365-2966.2009.15715.x arXiv:https://academic.oup.com/mnras/article-pdf/401/2/791/3952227/mnras0401-0791.pdf

[39] Bjarne Stroustrup. 2013. *The C++ Programming Language* (4 ed.). Addison-Wesley Professional.

[40] Ivan E. Sutherland. 1964. Sketchpad: A Man-Machine Graphical Communication System. *SIMULATION* 2, 5 (May 1, 1964), R–3–R–20. https://doi.org/10.1177/003754976400200514

[41] Till Tantau. 2021. *TikZ & PGF* (3.1.9a ed.). https://github.com/pgf-tikz/pgf

[42] The CGAL Project. 2021. *CGAL User and Reference Manual* (5.3 ed.). CGAL Editorial Board. https://doc.cgal.org/5.3/Manual/packages.html

[43] Hilderick A. van der Meiden and Willem F. Bronsvoort. 2009. A Non-rigid Cluster Rewriting Approach to Solve Systems of 3D Geometric Constraints. *Computer-Aided Design* 42, 1 (2009), 36–49. https://doi.org/10.1016/j.cad.2009.03.003

[44] Rui Ventura. 2021. *CGAL.jl (Version v0.5.0)*. Zenodo. https://doi.org/10.5281/zenodo.5137358

[45] Nier Wu and Horea T. Ilies. 2007. Motion-based Shape Morphing of Solid Models. In *Proceedings of the ASME 2007 International Design Engineering Technical Conferences and Computers in Engineering Conference (IDETC2007, Vol. 6: 33rd*

Design Automation Conference, Parts A and B). Las Vegas, Nevada, USA, 525–535. https://doi.org/10.1115/DETC2007-34826

[46] Jihun Yu, Chee Yap, Zilin Du, Sylvain Pion, and Hervé Brönnimann. 2010. The Design of CORE 2: A Library for Exact Numeric Computation in Geometry and Algebra. In *Mathematical Software — ICMS 2010 (Lecture Notes in Computer Science, Vol. 6327)*, Komei Fukuda, Joris van der Hoeven, Michael Joswig, and Nobuki Takayama (Eds.). Springer, Berlin, Heidelberg, Kobe, Japan, 121–141. https://doi.org/10.1007/978-3-642-15582-6_24

A APPENDIX

Listing 8: C++ wrapper code that maps the functionality required to reproduce the example shown in listing 3 in Julia.

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

#include <jlcxx/jlcxx.hpp>

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

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

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

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

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

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

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

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

Listing 9: An example Julia module, wrapping the library produced from listing 8.

```julia
module CGAL

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

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

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

end # CGAL
```