Design and Implementation of a White-Box Geometric Constraint Solver

Dávid Kasznár[1]

[1]Independent Researcher, kasznardavid8@gmail.com

Corresponding author: Dávid Kasznár, kasznardavid8@gmail.com

Abstract. Geometric constraint solvers are a fundamental component of parametric CAD systems, yet most production implementations are large, opaque, and tightly coupled to specialized libraries. This paper presents a white-box reference implementation of a two-dimensional geometric constraint solver written entirely in Go using only the standard library. The system represents constraints as a set of nonlinear algebraic equations over point coordinates and solves them using multidimensional Newton's method. Derivatives required for the Jacobian matrix are computed symbolically via recursive traversal of expression trees, avoiding both numerical finite differences and external computer-algebra dependencies. The complete implementation spans approximately 2000 lines of code across four packages. Each mathematical and algorithmic step is explicitly encoded and can be traced, modified, or extended without specialist tooling. The solver is validated on two-dimensional distance constraints, from which further geometric relationships can be derived. This work is intended as a pedagogical and research artifact that exposes the core architecture underlying parametric CAD sketching, providing a concrete foundation for teaching, experimentation, and future extension to three-dimensional geometry.

Keywords: geometric constraint solver, parametric modeling, symbolic differentiation, Newton's method, abstract syntax tree, reference implementation, CAD
DOI: https://doi.org/10.14733/cadaps.2027.55-67

1 INTRODUCTION

Parametric CAD systems allow designers to define geometry through relationships: distances, angles, parallelism, tangency, rather than fixed coordinates. A geometric constraint solver is the engine that enforces these relationships: given a set of constraints and an initial sketch configuration, it finds a configuration of geometric entities that satisfies all constraints simultaneously.

Despite their central importance, geometric constraint solvers are rarely studied from first principles. Production implementations such as those found in commercial CAD kernels are built on large software stacks, rely on proprietary numerical libraries, and expose no internal structure to the user. Even open-source systems

© 2027 U-turn Press LLC, http://www.cad-journal.net

such as SolveSpace [11] are sophisticated enough that their solving pipeline is difficult to trace end-to-end. As a result, constraint solving is frequently treated as a black box by developers and researchers alike, which hinders both pedagogy and experimentation.

This paper describes a white-box reference implementation of a two-dimensional geometric constraint solver. The system is written in Go using only the standard library and is structured so that every mathematical and algorithmic step is explicit and traceable. The implementation covers the complete pipeline: from the representation of geometric constraints as algebraic equations, through symbolic computation of partial derivatives, to the iterative Newton solver and the underlying linear algebra. With approximately 2000 lines of code, the entire system can be read and understood in a single session.

The primary contribution is not a new solving algorithm. Rather, it is a readable, dependency-free artifact that makes the internals of constraint-based modeling accessible to students, educators, and developers. The source code is publicly available at [7].

The remainder of the paper is organized as follows. Section 2 surveys related work. Section 3 describes the system architecture. Sections 4 and 5 present the equation representation and symbolic differentiation in detail. Section 6 covers the nonlinear solver. Section 7 demonstrates the system, and Section 8 discusses limitations. Section 9 concludes with directions for future work.

2 RELATED WORK

2.1 Constraint Solving Approaches

Surveys of geometric constraint solving [1, 4] identify three broad families of methods. Algebraic methods reduce constraints to polynomial systems and apply exact symbolic procedures such as Gröbner bases or resultant elimination to find all solutions. While theoretically complete, these methods are exponential in time and space [2], making them impractical for interactive CAD use. Constructive and rule-based methods work geometrically, sequentially placing entities according to precomputed construction plans or rewrite rules. Numerical methods formulate constraints as equations and solve the system iteratively; they are the most general and the approach taken here.

2.2 Constructive and Rule-Based Methods

Constructive solvers [6, 3] work by decomposing a sketch into an ordered sequence of construction steps, each step placing one geometric entity using ruler-and-compass constructions. Rule-based solvers [5] formalize this as a rewrite system: a library of inference rules is applied in sequence, each rule consuming known values to derive unknown ones. Both families share the same fundamental limitation: they are restricted to ruler-and-compass constructible configurations. When constraints are mutually dependent and there is no valid sequential construction order, these methods fail [8]. Lee and Kim [8] note that "due to the limitations of the numerical approach . . . most parametric design systems adopt the constructive approach," but also acknowledge that constructive methods cannot handle configurations that are "ruler-and-compass non-constructible." Their proposed hybrid (DOF-based graph reduction) combines both paradigms precisely to overcome this gap.

A further practical drawback of constructive and rule-based approaches is the need for a specialized knowledge base: every new constraint type requires a corresponding construction rule or inference rule to be added to the library. This couples the solver's capability directly to its rule set, making it harder to extend and inspect [4].

2.3 Numerical Equation-System Approach

In the numerical approach each constraint is expressed as an equation $f_i(\mathbf{x}) = 0$ over the point coordinates, yielding a system of nonlinear equations solved iteratively. The representation is uniform: any well-constrained

© 2027 U-turn Press LLC, http://www.cad-journal.net

problem expressible as equations can in principle be solved, with no restriction to constructible configurations. Adding a new constraint type requires only writing its equation; no rule library needs to be extended.

The cost is an $O(n^3)$ linear solve per Newton step [2] and dependence on initial conditions, issues that constructive methods avoid. For the small, pedagogically motivated sketches targeted here, these costs are immaterial, and the uniformity and simplicity of the equation-system representation are decisive advantages.

2.4 Open Implementations: SolveSpace and SketchFlat

The design of the solver presented here was informed by SolveSpace [11] and SketchFlat [10], both of which use Newton's method on a system of constraint equations. The present implementation was developed independently from scratch, with the specific goal of reducing the system to its minimal, readable core.

2.5 Differentiation Strategies

The Jacobian required by Newton's method can be obtained in three ways. Numerical differentiation approximates derivatives via finite differences; it introduces truncation error and doubles the number of function evaluations per column. Automatic differentiation propagates exact derivative values alongside function values, typically via operator overloading or source transformation; it is exact, but recomputes derivatives from scratch on every evaluation pass. Symbolic differentiation produces a new expression tree for each derivative. Because that tree is reusable, it is built once before the Newton loop and evaluated at every iteration against the updated parameter values, separating the one-time cost of differentiation from the per-iteration cost of evaluation. This work adopts the symbolic approach.

3 SYSTEM ARCHITECTURE

The implementation is organized into four packages, each with a clearly defined responsibility (Figure 1). A fifth package provides the user interface.

cmd (interactive UI, Ebiten)
sketch (points, constraints $\rightarrow$ equations)
solver / expr (AST, symbolic diff., Newton iteration)
math (Vector, Matrix, Gaussian elimination)

Figure 1: Package layering. Each layer depends only on the one below it.

math. Provides `Vector` (a typed `[]float64` slice) and `Matrix` (a `[]Vector`) together with arithmetic operations, row-swapping, and augmented-matrix support for Gaussian elimination.
solver/expr. Defines the expression AST and implements two tree-traversal algorithms: numerical evaluation (`Eval`) and symbolic partial differentiation (`PartialDiff`).
solver. Manages named scalar parameters, assembles the Jacobian from symbolic derivatives, evaluates the equation system, solves the linear update step via Gaussian elimination, and drives the Newton iteration loop.
sketch. Provides the geometric API: `Point`, `Line`, and `Sketch`. It translates user-level constraints (e.g., "distance between A and B equals 5") into algebraic equations over coordinate parameters.
cmd. A minimal interactive UI built on the Ebiten game library. It allows users to place points, add distance constraints, and trigger the solver. The UI is strictly separated from the mathematical core; the solver packages have no dependency on it.

© 2027 U-turn Press LLC, http://www.cad-journal.net

4 EQUATION REPRESENTATION

4.1 Expression Trees

Every constraint in the sketch is reduced to a scalar equation of the form $f(\mathbf{x}) = 0$ [10], where $\mathbf{x}$ is the vector of free point coordinates. Each such equation is represented as an abstract syntax tree (AST) whose nodes are instances of the Expr struct:

```
type ExprType string

const (
    CONSTANT  ExprType = "CONSTANT"
    PARAMETER ExprType = "PARAMETER"
    ADD       ExprType = "ADD"
    SUBTRACT  ExprType = "SUBTRACT"
    MULTIPLY  ExprType = "MULTIPLY"
    SQUARE    ExprType = "SQUARE"
    NEGATE    ExprType = "NEGATE"
)

type Expr struct {
    Type  ExprType
    Left  *Expr
    Right *Expr
    Value float64
    Name  string
}
```

Listing 1: Core expression node.

Leaf nodes are either constants (CONSTANT, carrying a float64 value) or named parameters (PARAMETER, carrying a string key that maps to the current value in a global lookup table). Internal nodes are binary or unary operators. A fluent builder API constructs trees compositionally:

```
func Number(value float64) *Expr { ... }
func Param(name string)    *Expr { ... }

func (e *Expr) Add(right *Expr)      *Expr { ... }
func (e *Expr) Subtract(right *Expr) *Expr { ... }
func (e *Expr) Multiply(right *Expr) *Expr { ... }
func (e *Expr) Square()              *Expr { ... }
func (e *Expr) Negate()              *Expr { ... }
```

Listing 2: Builder methods for composing expression trees.

4.2 Distance Constraint Example

The Euclidean distance constraint between points $A = (A_x, A_y)$ and $B = (B_x, B_y)$ requires

$$f(A_x, A_y, B_x, B_y) = (A_x - B_x)^2 + (A_y - B_y)^2 - d^2 = 0. \tag{1}$$

This is constructed in the sketch package as:

```
func (s *Sketch) SetDistance(A string, B string, d float64) {
    a := s.points[A]
    b := s.points[B]
    e := a.X.Subtract(b.X).Square().
        Add(a.Y.Subtract(b.Y).Square()).
        Subtract(Number(d).Square())
    s.system = append(s.system, e)
}
```

Listing 3: Building the distance constraint equation tree.

The resulting tree for Equation (1) is shown schematically in Figure 2.

© 2027 U-turn Press LLC, http://www.cad-journal.net

```
SUBTRACT
  ADD
    SQUARE( SUBTRACT(Param("Ax"), Param("Bx")) )
    SQUARE( SUBTRACT(Param("Ay"), Param("By")) )
  SQUARE( Number(d) )
```

Figure 2: Expression tree for the distance constraint $(A_x - B_x)^2 + (A_y - B_y)^2 - d^2$.

4.3 Numerical Evaluation

The `Eval` method traverses the tree recursively. `PARAMETER` nodes read their current value from a global `map[string]float64` that is populated with the current parameter vector at the start of each Newton iteration. This design decouples the tree structure from the parameter values, so the same tree is evaluated repeatedly as parameters are updated.

```
func (e *Expr) Eval() float64 {
    switch e.Type {
    case CONSTANT:  return e.Value
    case PARAMETER: return parameters[e.Name]
    case ADD:       return e.Left.Eval() + e.Right.Eval()
    case SUBTRACT:  return e.Left.Eval() - e.Right.Eval()
    case MULTIPLY:  return e.Left.Eval() * e.Right.Eval()
    case SQUARE:    v := e.Left.Eval(); return v * v
    case NEGATE:    return -e.Left.Eval()
    }
    panic("unknown node type")
}
```

Listing 4: Recursive evaluation of an expression tree.

5 SYMBOLIC DIFFERENTIATION

5.1 Motivation

Newton's method requires the Jacobian matrix $J_{ij} = \partial f_i / \partial x_j$. Computing Jacobian entries numerically via finite differences introduces $O(h)$ truncation error (for step size h), requires two evaluations per entry, and can fail near singularities. Automatic differentiation produces exact derivatives, but recomputes them from scratch on every evaluation pass. The approach taken here is symbolic differentiation [10]: the AST is traversed once to produce a new AST representing the partial derivative. That derivative tree is built before the Newton loop begins and reused at every iteration, evaluating the same expression structure against updated parameter values each time.

5.2 Differentiation Rules

The `PartialDiff(by string)` method returns a new `*Expr` representing the partial derivative of the receiver with respect to the named parameter `by`. It applies standard calculus rules by structural recursion:

$$\frac{\partial c}{\partial x} = 0, \qquad \frac{\partial x}{\partial x} = 1, \qquad \frac{\partial y}{\partial x} = 0 \;\; (y \neq x) \tag{2}$$

$$\frac{\partial (f+g)}{\partial x} = \frac{\partial f}{\partial x} + \frac{\partial g}{\partial x}, \qquad \frac{\partial (f-g)}{\partial x} = \frac{\partial f}{\partial x} - \frac{\partial g}{\partial x} \tag{3}$$

© 2027 U-turn Press LLC, http://www.cad-journal.net

$$\frac{\partial(f \cdot g)}{\partial x} = \frac{\partial f}{\partial x} \cdot g + f \cdot \frac{\partial g}{\partial x} \quad \text{(product rule)} \tag{4}$$

$$\frac{\partial(f^2)}{\partial x} = 2 \cdot f \cdot \frac{\partial f}{\partial x} \quad \text{(chain rule for square)} \tag{5}$$

5.3 Implementation

```
func (e *Expr) PartialDiff(by string) *Expr {
    switch e.Type {
    case CONSTANT:
        return Number(0)
    case PARAMETER:
        if e.Name == by { return Number(1) }
        return Number(0)
    case ADD:
        return e.Left.PartialDiff(by).Add(e.Right.PartialDiff(by))
    case SUBTRACT:
        return e.Left.PartialDiff(by).Subtract(e.Right.PartialDiff(by))
    case MULTIPLY:
        dL := e.Left.PartialDiff(by)
        dR := e.Right.PartialDiff(by)
        return dL.Multiply(e.Right).Add(e.Left.Multiply(dR))
    case SQUARE:
        return Number(2).Multiply(e.Left).Multiply(e.Left.PartialDiff(by))
    }
    panic("unknown node type")
}
```

Listing 5: Symbolic partial differentiation by AST traversal.

5.4 Example: Derivative of the Distance Constraint

Applying `PartialDiff("Ax")` to the distance constraint $(A_x - B_x)^2 + (A_y - B_y)^2 - d^2$ yields, by the chain rule for `SQUARE` and the constant rule for the d^2 term:

$$\frac{\partial f}{\partial A_x} = 2(A_x - B_x) \cdot 1 + 2(A_y - B_y) \cdot 0 - 0 = 2(A_x - B_x). \tag{6}$$

Similarly, $\partial f / \partial B_x = -2(A_x - B_x)$, confirming that the signed displacement drives the update step. The produced derivative is itself an `Expr` tree: `MULTIPLY(Number(2), SUBTRACT(Param("Ax"), Param("Bx")))`, which is evaluated numerically in each Newton iteration.

6 SOLVING THE EQUATION SYSTEM

6.1 Problem Formulation

Let the sketch have n free scalar parameters $\mathbf{x} = [x_1, \ldots, x_n]^T$ (point coordinates) and m constraints $f_1, \ldots, f_m$. The solver seeks $\mathbf{x}^*$ such that

$$\mathbf{F}(\mathbf{x}) = \begin{bmatrix} f_1(x_1, \ldots, x_n) \\ \vdots \\ f_m(x_1, \ldots, x_n) \end{bmatrix} = \mathbf{0}. \tag{7}$$

The present implementation targets the well-constrained case ($m = n$), where a unique (or isolated) solution exists near the initial configuration.

© 2027 U-turn Press LLC, http://www.cad-journal.net

6.2 Newton's Method

Given an initial guess $\mathbf{x}^{(0)}$ (the positions as placed by the user [10]), Newton's method iterates

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - J^{-1}(\mathbf{x}^{(k)})\,\mathbf{F}(\mathbf{x}^{(k)}), \tag{8}$$

where $J(\mathbf{x})$ is the $m \times n$ Jacobian matrix with entries

$$J_{ij}(\mathbf{x}) = \frac{\partial f_i}{\partial x_j}(\mathbf{x}). \tag{9}$$

In practice, $J^{-1}\mathbf{F}$ is not computed by matrix inversion. Instead, the linear system

$$J(\mathbf{x}^{(k)})\,\mathbf{d}^{(k)} = \mathbf{F}(\mathbf{x}^{(k)}) \tag{10}$$

is solved for the update step $\mathbf{d}^{(k)}$, and the parameters are updated as $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \mathbf{d}^{(k)}$.

6.3 Parameter Management

The `SystemParameters` type maintains a list of named scalar parameters together with their current values. Before each Newton iteration, parameter values are synchronized to the global `map[string]float64` looked up by `Param.Eval()`. A separate ordered list (`paramList`) fixes the column ordering of the Jacobian:

```
func (sp *SystemParameters) save() {
    paramList = paramList[:0]
    for _, p := range sp.list {
        parameters[p.name] = p.value
        paramList = append(paramList, p.name)
    }
}
```

Listing 6: Parameter snapshot used by the evaluation layer.

6.4 Jacobian Assembly

The Jacobian is first built symbolically (once, before the iteration loop) by calling `PartialDiff` for every (equation, parameter) pair:

```
func createJacobian(equations []*Expr) [][]*Expr {
    J := make([][]*Expr, len(equations))
    for i, eq := range equations {
        J[i] = make([]*Expr, len(paramList))
        for j, p := range paramList {
            J[i][j] = eq.PartialDiff(p)
        }
    }
    return J
}
```

Listing 7: Symbolic Jacobian construction.

At each iteration, the symbolic Jacobian is evaluated numerically:

```
func evalJacobian(J [][]*Expr) Matrix {
    m := NewMatrix(len(J), len(J[0]))
    for i := range J {
        for j := range J[i] {
            m[i][j] = J[i][j].Eval()
        }
    }
    return m
}
```

Listing 8: Numerical evaluation of the symbolic Jacobian.

© 2027 U-turn Press LLC, http://www.cad-journal.net

6.5 Linear Solve: Gaussian Elimination

The linear system (10) is solved by Gaussian elimination with partial pivoting [9, 10]. The coefficient matrix is augmented with the right-hand-side vector to form $[J \mid \mathbf{F}]$, then reduced to upper-triangular form:

```
func gaussEliminate(A Matrix, n int) {
    for i := 0; i < n; i++ {
        pivot := i
        for j := i + 1; j < n; j++ {
            if math.Abs(A[j][i]) > math.Abs(A[pivot][i]) {
                pivot = j
            }
        }
        if pivot != i { A.SwapRows(pivot, i) }
        for j := i + 1; j < n; j++ {
            factor := A[j][i] / A[i][i]
            A[j] = A[j].Subtract(A[i].Multiply(factor))
        }
    }
}
```

Listing 9: Forward elimination with partial pivoting.

Back-substitution then recovers $\mathbf{d}^{(k)}$ from the triangular system.

6.6 Iteration Loop and Convergence

The full Newton loop is bounded at 100 iterations. Convergence is declared when every component of the update step $\mathbf{d}^{(k)}$ satisfies $|d_i| < 10^{-6}$:

```
func SolveSystem(system []*Expr, params *SystemParameters) {
    params.save()
    J_sym := createJacobian(system)
    for k := 0; k < 100; k++ {
        J_x := evalJacobian(J_sym)
        F_x := evalSystem(system)
        d   := SolveGauss(J_x, F_x)
        converged := true
        for _, v := range d {
            if math.Abs(v) > 1e-6 { converged = false; break }
        }
        if converged { break }
        x := params.getVec()
        params.saveVec(x.Subtract(d))
    }
}
```

Listing 10: Newton iteration loop in SolveSystem.

7 DEMONSTRATION

7.1 Interactive UI

The `cmd` package provides a minimal graphical interface backed by the Ebiten library. Users can place points by clicking on a canvas, enter distance constraints between point pairs, and trigger the solver. The canvas redraws after each solve step, showing the updated positions. The UI is the only component that depends on a non-standard library; the four core packages (`math`, `solver`, `sketch`, `utils`) use only the Go standard library.

7.2 Unit Tests

Each package carries unit tests that serve as executable specifications:

© 2027 U-turn Press LLC, http://www.cad-journal.net

- `expr_test.go`: verifies evaluation and symbolic differentiation for individual operators, including the product rule and chain rule.
- `solver_test.go`: constructs small equation systems and checks that Newton iteration converges to the correct solution within the expected number of steps.
- `sketch_test.go`: exercises distance constraints end-to-end, confirming that after `SatisfyConstraints()` the Euclidean distance between points matches the specified value to within numerical tolerance.
- `matrix_test.go` / `vector_test.go`: cover the linear algebra primitives and Gaussian elimination routine.

7.3 Geometric Examples

The following three examples are taken directly from the test suite in `sketch_test.go`. Each sketch is set up with fixed anchor points (shown as filled squares ■) and one free point (shown as a filled circle •). The open gray circle marks the initial guess supplied to the solver; the dashed arcs show the loci imposed by the distance constraints.

Example 1: Symmetric two-circle intersection

Two anchors $O_1 = (0, 0)$ and $O_2 = (10, 0)$ are fixed. Point A is free, starting from the initial guess $(5, 3)$. Two distance constraints, $|O_1A| = 7$ and $|O_2A| = 7$, together require A to lie on the intersection of two circles of equal radius. The system is well-constrained ($m = n = 2$) and Newton's method converges to $A = (5,\ 4.90)$.

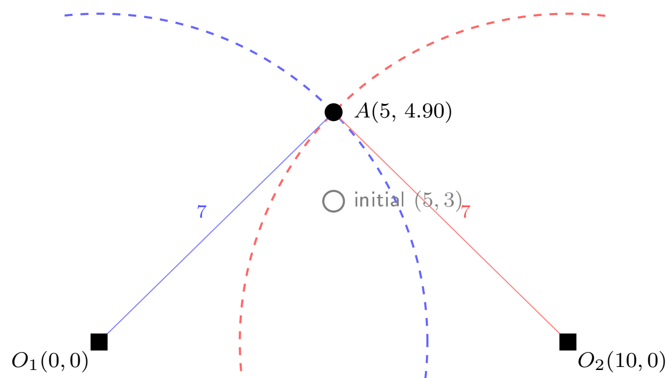

Figure 3: Example 1: symmetric distance constraints $|O_1A| = |O_2A| = 7$. The free point A (filled circle) converges from the initial guess (open circle) to the upper intersection of the two constraint circles (dashed arcs).

Example 2: Asymmetric distances

The same two anchors are used, but with unequal constraints: $|O_1A| = 5$ and $|O_2A| \approx 11.18$. Starting from the same initial guess $(5, 3)$, the solver converges to $A = (0,\ 5)$, the point that lies exactly on the circle of radius 5 around O_1 and at distance $\sqrt{125}$ from O_2.

© 2027 U-turn Press LLC, http://www.cad-journal.net

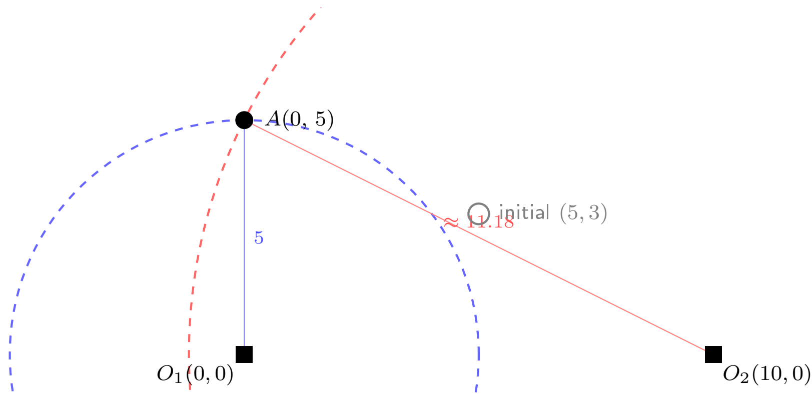

Figure 4: Example 2: asymmetric distance constraints $|O_1A| = 5$, $|O_2A| \approx 11.18$. The free point A converges to $(0, 5)$, lying on the small circle (blue, dashed) and at distance $\sqrt{125}$ from O_2 on the large circle (red, dashed).

Example 3: Equilateral triangle

Two anchors $O_1 = (0, 0)$ and $O_2 = (5, 0)$ define a fixed baseline of length 5. Point C is free, starting from the initial guess $(2, 1)$. The constraints $|O_1C| = 5$ and $|O_2C| = 5$ together with the fixed baseline form a well-constrained system ($m = n = 2$) whose solution is the apex of an equilateral triangle. The solver converges to $C = (2.5,\ 4.33)$.

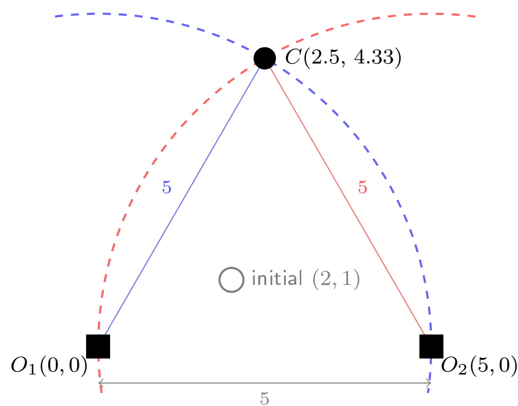

Figure 5: Example 3: equilateral triangle. Fixed baseline $O_1O_2 = 5$; free apex C is constrained by $|O_1C| = |O_2C| = 5$. After solving, C lies at the upper intersection of the two constraint circles (dashed arcs), forming an equilateral triangle with all sides equal to 5.

© 2027 U-turn Press LLC, http://www.cad-journal.net

8 DISCUSSION

8.1 Numerical Stability

The Gaussian elimination step uses partial pivoting, selecting the row with the largest absolute value in the pivot column before each elimination step. This reduces the growth of round-off errors and avoids division by near-zero values in well-posed systems. For the small, well-constrained sketches targeted by this implementation, the Jacobian is typically full-rank and well-conditioned, so partial pivoting is sufficient.

Newton's method exhibits quadratic convergence near the solution [9]: once the iterate is close enough to a root, the error decreases roughly as the square of the previous error. Convergence is not guaranteed when the initial configuration is far from a solution or when the Jacobian is near-singular.

8.2 Robustness for Complex Constraint Systems

Four challenges arise when scaling the equation-system approach to more complex sketches, each with well-established remedies that fit naturally into the existing framework.

Global convergence. Pure Newton iteration has quadratic local convergence but may diverge when the initial guess is far from the solution. The standard remedy is a damped Newton step

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \alpha\,\mathbf{d}^{(k)}, \tag{11}$$

where $\alpha \in (0, 1]$ is chosen by a backtracking line search (e.g., the Armijo condition [9]) to guarantee a sufficient decrease of $\|\mathbf{F}\|$ at each step. Once the iterate enters the region of quadratic convergence, $\alpha = 1$ is accepted and the full Newton rate is recovered. The only change to the existing solver loop is the addition of a step-length selection before the parameter update.

Under- and over-constraint detection. The rank of the Jacobian encodes the constraint status: full row rank means the system is well-constrained; a rank deficit signals redundant or inconsistent constraints. Placing the Jacobian in reduced row echelon form via Gauss-Jordan elimination, as described by Westhues [10], identifies which rows are linearly dependent (redundant constraints) and which columns lack a leading pivot (free parameters), enabling precise feedback to the user about which constraints to add or remove.

Performance for large models. Dense Gaussian elimination costs $O(n^3)$ per Newton step, which is negligible for sketches with tens of constraints. For larger models the Jacobian is structurally sparse: each equation involves only the few parameters of the geometric entities it constrains, so sparse direct solvers or iterative Krylov methods such as GMRES [9] reduce the per-iteration cost substantially while leaving the Newton framework, the symbolic Jacobian construction, and the convergence criterion unchanged.

8.3 Extension to Three Dimensions

The equation-system architecture adapts to 3D with minimal changes. Each `Point` gains a third scalar parameter z; a sketch with n free points then has $3n$ free parameters instead of $2n$. The 3D Euclidean distance constraint becomes

$$(A_x - B_x)^2 + (A_y - B_y)^2 + (A_z - B_z)^2 - d^2 = 0, \tag{12}$$

which adds one new `SQUARE` term compared to Equation (1). The symbolic differentiator handles it identically: `PartialDiff("Az")` traverses the tree and returns $2(A_z - B_z)$ by the same chain rule already in use.

Other 3D constraints follow the same pattern of expressing a geometric relationship as an algebraic equation. A point-on-plane constraint, given a fixed unit normal $\mathbf{n} = (n_x, n_y, n_z)$ and a reference point $\mathbf{p}_0$, is the linear equation

$$n_x(P_x - p_{0x}) + n_y(P_y - p_{0y}) + n_z(P_z - p_{0z}) = 0, \tag{13}$$

© 2027 U-turn Press LLC, http://www.cad-journal.net

built from MULTIPLY and ADD nodes over the point's parameters. An angle between two line directions uses a normalised dot product,

$$\frac{(B-A)\cdot(D-C)}{|B-A|\,|C-D|} - \cos\theta = 0, \tag{14}$$

which requires SQRT and DIVIDE nodes not currently in the AST but straightforwardly added with corresponding differentiation rules (quotient rule and chain rule for square root).

The solver and math packages require no modification: symbolic differentiation, Jacobian assembly, Gaussian elimination, and Newton iteration operate entirely on abstract parameter names and expression trees with no notion of spatial dimension. Only the sketch layer, the part that maps geometric relationships to equation trees, changes.

8.4 Design Trade-offs

The use of a global parameter map for Eval() is a deliberate simplicity choice: it avoids threading a parameter context through every recursive call. A production implementation would prefer an explicit parameter vector argument to enable safe concurrent evaluation. Likewise, the dedicated SQUARE node type avoids implementing a general power operator, which would require a more complex chain-rule case; this is sufficient for all constraints derived from the Euclidean distance formula.

9 CONCLUSION AND FUTURE WORK

This paper presented a minimal, white-box geometric constraint solver implemented in Go without external dependencies. Geometric constraints are expressed as algebraic equations over point coordinates using an abstract syntax tree. Partial derivatives are computed symbolically by recursive AST traversal, applying the product rule for multiplication and the chain rule for squaring. A multidimensional Newton iteration assembles the Jacobian from these symbolic derivatives, evaluates it numerically at each step, solves the resulting linear system by Gaussian elimination with partial pivoting, and updates the parameters until convergence. The complete pipeline is encoded in approximately 2000 lines of readable, testable Go code.

The solver is intended as a pedagogical and research artifact. It exposes every step of the constraint-solving pipeline in a form that can be read, traced, and modified without specialist tooling, providing a foundation for:

- Teaching numerical methods and constraint-based CAD in courses on computational geometry or CAD system development.
- Rapid prototyping of new constraint types: adding a constraint requires only constructing its equation tree and appending it to the system.
- Exploring alternative differentiation or solving strategies by replacing individual components.

The robustness extensions discussed in Section 8 (damped Newton with line search, constraint-rank analysis for under- and over-constraint detection, sparse linear solvers for large models, and 3D constraint formulations) are natural next steps, each of which slots into the existing layered architecture without requiring changes to the symbolic differentiation or Newton iteration core.

The source code is available at [7].

References

[1] Ait-Aoudia, S.; Bahriz, M.; Salhi, L.: 2D geometric constraint solving: an overview. In 2009 Second International Conference in Visualisation (VIS), 136–141, 2009. http://doi.org/10.1109/VIZ.2009.29.

© 2027 U-turn Press LLC, http://www.cad-journal.net

[2] Ait-Aoudia, S.; Jegou, R.; Michelucci, D.: Reduction of constraint systems. In Compugraphics, 331–340, 1993.

[3] Bouma, W.; Fudos, I.; Hoffmann, C.; Cai, J.; Paige, R.: A geometric constraint solver. Computer-Aided Design, 27(6), 487–501, 1995. http://doi.org/10.1016/0010-4485(94)00013-4.

[4] Joan-Arinyo, R.: Basics on geometric constraint solving. Tech. rep., Universitat Politècnica de Catalunya, 1999. Lecture notes, Departament de Llenguatges i Sistemes Informàtics.

[5] Joan-Arinyo, R.; Soto, A.: A correct rule-based geometric constraint solver. Computers & Graphics, 21(5), 599–609, 1997. http://doi.org/10.1016/S0097-8493(97)00038-1.

[6] Joan-Arinyo, R.; Soto, A.: A ruler-and-compass geometric constraint solver. In Product Modeling for Computer Integrated Design and Manufacture (IFIP), 384–393. Chapman & Hall, 1997. http://doi.org/10.1007/978-0-387-35187-2_33.

[7] Kasznár, D.: geometric-constraint-solver. https://github.com/kasznar/geometric-constraint-solver, 2024.

[8] Lee, J.Y.; Kim, K.: A 2-D geometric constraint solver using DOF-based graph reduction. Computer-Aided Design, 30(11), 883–896, 1998. http://doi.org/10.1016/S0010-4485(98)00045-1.

[9] Press, W.H.; Teukolsky, S.A.; Vetterling, W.T.; Flannery, B.P.: Numerical Recipes: The Art of Scientific Computing. Cambridge University Press, 3rd ed., 2007.

[10] Westhues, J.: SketchFlat: A constraint-based drawing tool — internals. https://cq.cx/dl/sketchflat-internals.pdf, 2007.

[11] Westhues, J.: SolveSpace – parametric 2d/3d CAD. https://solvespace.com/tech.pl, 2023. Accessed: 2024.

© 2027 U-turn Press LLC, http://www.cad-journal.net