Title:
Design and Implementation of a Transparent Geometric Constraint Solver

Authors:
Dávid Kasznár, kasznardavid8@gmail.com, Independent Researcher

Keywords:
geometric constraint solver, parametric modeling, symbolic differentiation, Newton's method, abstract syntax tree, reference implementation, CAD

DOI: 10.14733/cadconfP.2026.39-43

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 are built on large software stacks, rely on proprietary numerical libraries, and expose no internal structure. Even open-source systems such as SolveSpace [1] 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, which hinders both pedagogy and experimentation.

This work presents a transparent reference implementation of a two-dimensional geometric constraint solver written in Go using only the standard library. The system represents constraints as nonlinear algebraic equations over point coordinates and solves them using multidimensional Newton's method. Derivatives required for the Jacobian 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. Every mathematical and algorithmic step is explicit and can be traced, modified, or extended without specialist tooling. The source code is publicly available at [10].

Related Work:

Surveys of geometric constraint solving [3, 4] identify three broad families. *Algebraic methods* reduce constraints to polynomial systems and apply exact symbolic procedures such as Gröbner bases; they are theoretically complete but exponential in practice [5]. *Constructive and rule-based methods* [6, 7] work geometrically, sequentially placing entities by ruler-and-compass constructions. Their fundamental limitation is that they cannot handle configurations which are ruler-and-compass non-constructible; each new constraint type also requires a new inference rule [8]. *Numerical methods* formulate constraints as equations $f_i(\mathbf{x}) = 0$ and solve iteratively. Any constraint expressible as an equation can in principle be solved, with no restriction to constructible configurations. Adding a new constraint type requires only writing its equation. This uniformity makes the numerical approach the most general, and it is the approach taken here.

The design of the present solver was informed by SolveSpace [1] and SketchFlat [2], both of which apply Newton's method to 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.

The Jacobian required by Newton's method can be obtained by numerical finite differences, automatic differentiation, or symbolic differentiation. 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 updated parameter values, separating the one-time cost of differentiation from the per-iteration cost of evaluation. This work adopts the symbolic approach [2].

System Architecture:

The implementation is organized into four packages, each with a clearly defined responsibility (Fig. 1).

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

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

math provides `Vector` and `Matrix` types together with arithmetic operations and 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, and drives the Newton iteration loop. **sketch** provides the geometric API (`Point`, `Line`, `Sketch`) and translates user-level constraints into algebraic equations over coordinate parameters. A fifth package, **cmd**, provides a minimal interactive UI built on the Ebiten library; it is the only component that depends on a non-standard library.

Equation Representation:

Every constraint is reduced to a scalar equation $f(\mathbf{x}) = 0$, where $\mathbf{x}$ is the vector of free point coordinates. Each 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 constants (`CONSTANT`) or named parameters (`PARAMETER`). Internal nodes are binary or unary operators. A fluent builder API constructs trees compositionally (`e.Add(r)`, `e.Subtract(r)`,

`e.Square()`, etc.).

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

$$f(A_x, A_y, B_x, B_y) = (A_x - B_x)^2 + (A_y - B_y)^2 - d^2 = 0. \tag{2.1}$$

The resulting expression tree is:

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

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

Symbolic Differentiation:

Newton's method requires the Jacobian matrix $J_{ij} = \partial f_i / \partial x_j$. The `PartialDiff(by string)` method returns a new `*Expr` representing the partial derivative of the receiver with respect to the named parameter. It applies standard calculus rules by structural recursion:

$$\frac{\partial c}{\partial x} = 0, \quad \frac{\partial x}{\partial x} = 1, \quad \frac{\partial y}{\partial x} = 0 \ \ (y \neq x) \tag{2.2}$$

$$\frac{\partial (f \pm g)}{\partial x} = \frac{\partial f}{\partial x} \pm \frac{\partial g}{\partial x} \tag{2.3}$$

$$\frac{\partial (f \cdot g)}{\partial x} = \frac{\partial f}{\partial x} \cdot g + f \cdot \frac{\partial g}{\partial x} \quad \text{(product rule)} \tag{2.4}$$

$$\frac{\partial (f^2)}{\partial x} = 2 \cdot f \cdot \frac{\partial f}{\partial x} \quad \text{(chain rule for square)} \tag{2.5}$$

```go
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 2: Symbolic partial differentiation by AST traversal.

Applying `PartialDiff("Ax")` to the distance constraint (2.1) yields $\partial f/\partial A_x = 2(A_x - B_x)$. The produced derivative is itself an `Expr` tree — `MULTIPLY(Number(2), SUBTRACT(Param("Ax"), Param("Bx")))` — which is evaluated numerically in each Newton iteration.

Newton Iteration and Linear Solve:

Let the sketch have n free scalar parameters $\mathbf{x} = [x_1, \ldots, x_n]^T$ and m constraints $f_1, \ldots, f_m$. The solver targets the well-constrained case $m = n$. Given an initial guess $\mathbf{x}^{(0)}$ (the positions as placed by the user), Newton's method iterates

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - J^{-1}(\mathbf{x}^{(k)})\,\mathbf{F}(\mathbf{x}^{(k)}), \tag{2.6}$$

where $J_{ij} = \partial f_i/\partial x_j$. In practice the linear system $J(\mathbf{x}^{(k)})\,\mathbf{d}^{(k)} = \mathbf{F}(\mathbf{x}^{(k)})$ is solved for the update step $\mathbf{d}^{(k)}$ by Gaussian elimination with partial pivoting [9], and parameters are updated as $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \mathbf{d}^{(k)}$.

The Jacobian is first built symbolically — once, before the iteration loop — by calling `PartialDiff` for every (equation, parameter) pair. At each iteration the symbolic Jacobian is evaluated numerically. This separation means differentiation is a one-time cost; only fast floating-point evaluation recurs each step.

```
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 3: Newton iteration loop.

Convergence is declared when every component of the update step satisfies $|d_i| < 10^{-6}$, with a maximum of 100 iterations. Simple distance-constrained sketches converge in three to ten iterations from a nearby initial guess.

Discussion:

Numerical stability. Gaussian elimination with partial pivoting selects the row with the largest absolute value in each pivot column, reducing round-off growth and avoiding division by near-zero values. For the small, well-constrained sketches targeted here the Jacobian is typically full-rank and well-conditioned, so partial pivoting is sufficient. Newton's method exhibits quadratic convergence near the solution [9]; convergence is not guaranteed when the initial configuration is far from the solution or the Jacobian is near-singular.

Robustness for complex systems. Four challenges arise when scaling to larger sketches, each with well-established remedies that fit naturally into the existing framework. (i) *Global convergence*: a damped step $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \alpha\,\mathbf{d}^{(k)}$, with α chosen by a backtracking line search (Armijo condition [9]), guarantees sufficient decrease of $\|\mathbf{F}\|$ at each step and recovers the full Newton rate once the iterate enters the quadratic-convergence region. (ii) *Constraint status*: the rank of the Jacobian encodes whether the system is well-, under-, or over-constrained. Placing the Jacobian in reduced row echelon form via Gauss-Jordan elimination identifies linearly dependent rows (redundant constraints) and columns without a leading pivot (free parameters) [2], enabling precise feedback to the user. (iii) *Performance*: dense

Gaussian elimination costs $O(n^3)$ per Newton step, negligible for small sketches. For larger models the Jacobian is structurally sparse; sparse direct solvers or iterative Krylov methods such as GMRES reduce the per-iteration cost while leaving the Newton framework, symbolic Jacobian construction, and convergence criterion unchanged.

Extension to 3D. The architecture adapts to three dimensions with minimal changes: each `Point` gains a third parameter z, and the 3D distance constraint $(A_x - B_x)^2 + (A_y - B_y)^2 + (A_z - B_z)^2 - d^2 = 0$ adds one `SQUARE` term to Eq. (2.1). The symbolic differentiator handles it identically; `PartialDiff("Az")` returns $2(A_z - B_z)$ by the same chain rule already in use. Further constraints such as point-on-plane or angle between lines follow the same pattern of expressing a geometric relationship as an algebraic equation; the `solver` and `math` packages require no modification.

Conclusions:

This work presented a minimal, transparent geometric constraint solver implemented in Go without external dependencies. Geometric constraints are expressed as algebraic equations over point coordinates using an AST. Partial derivatives are computed symbolically by recursive AST traversal applying the product rule and chain rule. A multidimensional Newton iteration assembles the Jacobian from these symbolic derivatives, 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, intended as a pedagogical and research artifact that exposes every step of the constraint-solving pipeline in a form that can be read, traced, and modified without specialist tooling.

Dávid Kasznár, https://orcid.org/0009-0000-6120-5892

References:

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

[2] Westhues, J.: SketchFlat: A Constraint-Based Drawing Tool — Internals, https://cq.cx/dl/sketchflat-internals.pdf, 2007.

[3] Ait-Aoudia, S.; Bahriz, M.; Salhi, L.: 2D Geometric Constraint Solving: an Overview, Second International Conference in Visualisation (VIS), 2009, 136–141. https://doi.org/10.1109/VIZ.2009.29

[4] Joan-Arinyo, R.: Basics on Geometric Constraint Solving, Universitat Politècnica de Catalunya, Lecture Notes, 1999.

[5] Ait-Aoudia, S.; Jegou, R.; Michelucci, D.: Reduction of Constraint Systems, Compugraphics, 1993, 331–340.

[6] Joan-Arinyo, R.; Soto, A.: A Ruler-and-Compass Geometric Constraint Solver, IFIP Product Modeling for Computer Integrated Design and Manufacture, 1997, 384–393. https://doi.org/10.1007/978-0-387-35187-2_33

[7] Bouma, W.; Fudos, I.; Hoffmann, C.; Cai, J.; Paige, R.: A Geometric Constraint Solver, Computer-Aided Design, 27(6), 1995, 487–501. https://doi.org/10.1016/0010-4485(94)00013-4

[8] Lee, J. Y.; Kim, K.: A 2-D Geometric Constraint Solver Using DOF-Based Graph Reduction, Computer-Aided Design, 30(11), 1998, 883–896. https://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, 3rd ed., Cambridge University Press, 2007.

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