# Verifiable Origami Folding

*G. Ramseyer*

**Abstract**: *Lines on a computationally-generated crease pattern often run between points not naively identifiable on a blank piece of paper. As such, origamists must make a series of reference folds in order to exactly identify various points and lines. Mistakes in this process can result in folding failures manifesting many hours later. We present a system for automatically verifying geometric properties of a candidate reference fold sequence. Our system may also aid in analyzing a crease pattern's tolerance for adjustment or error.*

## 1 Introduction

Over the last few decades, the variety and complexity in origami model designs have exploded. While the development can be attributed to many factors, much of the increase in complexity has been enabled by the development of rigorous mathematical techniques and associated computational tools. For example, tools like Lang's Treemaker [Lang 06] have facilitated the systematic construction of complex model bases.

Tools for model design typically give the user two-dimensional crease patterns with vertices provided numerically, rather than a complete set of folding instructions. This is not a limitation for the designer, as a designer will often want to refine the model given by the computer anyway, especially for the delicate shaping process.

However, computer-generated crease patterns can easily contain lines folded between numerically-specified points that are not physically identifiable on a blank sheet of paper. To fold them, the folder must perform a sequence of reference folds to identify the point. Such fold sequences can be worked out manually, or found through the aid of other computational tools like ReferenceFinder [Lang 17].

Any mistake in these reference folds will necessarily propagate throughout the folded model. This can, in the worst case, result in models with drastically altered proportions, perhaps with "flat" parts that do not fold flat, or even in missing appendages. Given that many complex models, especially those not published in books, are specified only by a few pictures and their crease patterns, folders sometimes must construct their own reference fold sequence. When a model takes many hours to fold, the cost of a mistake in the reference folds to the folder can be quite high. An individual might benefit, then, from the ability to validate that a reference fold sequence actually gives the desired reference points.

Furthermore, a folder, perhaps a designer folding someone else's design, might like to know how much flexibility they have in moving some of the reference points around without distorting other parts of the fold geometry. Doing such an analysis manually would require proving some arbitrary geometric predicate by hand, given some geometric construction. For example, one might like to know how much one can alter the size of a tail without compromising the topological integrity of the model's legs. This might require proving that certain angles or ratios are maintained given a small amount of variability.

This geometric analysis problem is what I address in this paper. In particular, given a sequence of geometric folds, whose definitions start based only on the corners of the paper and perhaps some variable points constrained in some manner, we would like to automate the generation of a proof of some geometric predicate about the folds. In essence, we can represent a fold by the lines and points that it uses for reference, a line by two points on the line, a point by two real variables, and any geometric constraint on folds, lines, and points by constraints on the corresponding variables. A proof that a geometric constraint holds, then, requires manipulating first-order constraints involving (multivariate) polynomials over real variables. For these manipulations, we can draw on a class of computer programs known collectively as SMT Solvers.

## 2 Preliminaries

### 2.1 Fold Sequences

In order to prove a geometric predicate about a sequence of folds, we must first find a suitable definition for a fold. In physical terms, a fold is an operation that, using lines and points already marked in the paper as references, creates one or more folds in the paper. Since we are working only in a reference geometry setting, and not computing with regard to a 3D model (as explained below, SMT solvers are not well suited to sequential folds or 3D folds), we will consider only the model in which each fold operation is immediately followed by the corresponding unfold operation. Hence, a fold is the definition of a line or lines in the paper by some origami means.

Origami folds are typically specified by some instruction involving existing points and lines. For example, one might "fold point A onto line B such that the resulting line passes through point C". Let $P$ be the set of all points in the paper and $L$ be the set of all lines. We can formalize a fold starting from a flat square paper as an operation that takes in a set of points $P^n$ and a set of lines $L^m$ and combines them in some fixed way to produce some new set of lines $L^{m'}$. We will call $P^n \times L^m$ reference objects.

Observe that, as in the above example, a fold's specification consists of a fixed operation on variable reference points. One fold of a paper, then, consists of choosing some method, or fold type, by which to combine reference objects, and then instantiating the fold type with the appropriate numbers of reference points and lines. Observe also that sometimes an instruction might not be foldable for a given reference object set. As in the above example, if the distance between $A$ and $C$ in the paper is less than the distance between $C$ and the closest point of line $B$, then one cannot fold point $A$ onto line $B$ with the fold line passing through line $C$. Thus, we must allow our definition of a fold operation to include unfoldable configurations. We will say that a fold operation will return $\perp$ in these unfoldable configurations. For convenience, we will also say that the operation defining a point as the intersection of two lines as a “fold” operation that produces either a point or $\perp$, and thus our “fold” operations will also be allowed to produce points.

Let us say, then, that a fold is an operation $f : P^n \times L^m \rightarrow \{L^{n'} \times P^{m'}\} \cup \{\perp\}$.

Prior to folding, the only reference objects identifiable on a square piece of paper are the corners and edges. For simplicity, we will look at only the corners, as the lines making up the edges are easily constructible symbolically from the corners. We will also allow the use of a set of unspecified “free” points and numerically specified points. Denote this set of points $P_0$.

A free variable will be some point in the paper that is not necessarily one of the corners of the paper, and thus not necessarily precisely identifiable to the folder. Nevertheless, some origami models involve simply picking a point on the paper, and folding based on that, and we might like to know how much flexibility we have in such choices. Oftentimes, one picks the point from a specific set of points - say, a random point on a line. Hence, we need to be able to assert various geometric conditions about our free variables. We will implicitly assert that all free points lie within the bounds of the paper.

We also will allow numerical specification of some points. This allows the user, when folding, to construct various points approximately when actually folding the model, but to reason about the points as though they had been constructed exactly. This also could allow the user to refactor a given fold sequence into a simpler sequence, which may increase the likelihood that an SMT solver will succeed.

Hence, we will define a well-defined sequence of folds as a sequence $(f_1, P_1, L_1)$, ..., $(f_t, P_t, L_t)$ and associated starting points $P_0$ such that for each $f_i$, $(P_i \cup L_i) \subset P_0 \cup (\cup_{j<i} f_j(P_j, L_j))$. In other words, each fold in the sequence is instantiated with appropriate numbers of reference objects and these reference objects are visible on the paper at the time of the fold. We will say the sequence is foldable if none of the operations returns $\perp$.

Our program, then, needs to process a sequence of folds, validate that the sequence is well-defined, determine whether or not it can actually be folded (no operation returns $\perp$) and verify some geometric predicate about the resulting points and lines. For this, we must compile the sequence of folds into constraints about which a computer can reason. In this case, we will use constraints involving polynomials over real variables, and we will reason about them using a class of computer programs known as Satisfiability Modulo Theories (SMT) Solvers.

## 2.2 Intermediate Representation

Observe that a line in Euclidean space can be represented by two points, and that in $\mathbb{R}^2$, a point is defined by two real values. This is not, perhaps, the most mathematically elegant method of representing a line via real variables, but it plays very nicely with SMT systems. Hence, for each fold operation and for each line created by the fold operation, we must be able to symbolically compute two distinct points on the new line. Because we have defined folds as fixed operations that act on input reference objects, we can specify a fold type via some symbolic computation, substituting in input parameters as necessary.

In short, an SMT solver determines the satisfiability of formulas in first-order logic. In this context, all formulas will be quantifier-free. As an imprecise overview, for our purposes a formula consists of a combination of (possibly negated) atomic formulas, connected via boolean connectives like $\wedge$ and $\vee$. An atomic formula consists of an assertion of some relation, like equality, $<$, $\leq$, etc, between two terms. And a term, in this context, is a polynomial over real variables. For example, a term might be $x$ or 2 or $2x^3 + y * z$, an atomic formula might be $x = y$, and a formula might be $(z = 2) \wedge \neg(y * y \leq 4)$.

Observe that consistently substituting a term for a variable in a formula yields another formula. For a formula $\varphi(x)$, we will denote $\varphi[x/t]$ as the formula generated by replacing all instances of $x$ with a term $t$. Observe that $\varphi[x/t]$ may have no free variables (if $t$ is a constant term) or multiple free variables (if $t$ is a multivariate polynomial). We will say a formula $\varphi(x)$ is satisfiable if there exists some real value $c$ such that $\varphi(c)$ is true. These definitions naturally extend to formulas over multiple variables. The SMT solver's job, then, is to, given a particular formula $\varphi(x_1,...,x_n)$, decide whether or not $\exists y_1,...,y_n \varphi(y_1,...,y_n)$, and if such $(y_1,...,y_n)$ exist, return them. We will call $(y_1,...,y_n)$ a model that satisfies $\varphi$.

## 3 SMT Overview

The boolean satisfiability (SAT) problem is the problem in which, given a conjunction of clauses of (possibly negated) Boolean variables, one must determine whether or not there exists an assignment of the Boolean variables that makes the entire conjunction evaluate to true. The 3-SAT problem, in which each clause is the disjunction of exactly three variables, is one of the earliest problems to be shown to be NP-complete. It is generally believed that no polynomial-time algorithm exists that can solve this problem, and yet, starting in the 1960s, researchers began developing software to heuristically solve many instances of the problem. These solvers are now quite powerful, solving most instances of the problem in practical situations.

The satisfiability modulo theories (SMT) problem is an extension of the SAT problem to various first-order logic fragments. Instead of purely Boolean clauses, an instance of a problem might contain some simple arithmetic. For example, one might ask $\exists x,y,z(x+y=z) \wedge (3*x<2) \wedge (z*y=x)$. A satisfying solution would then consist of assignments of variables to values of some sensible type. In this case, a satisfying solution, or model, is $x=0, y=0, z=0$. Modern SMT solvers support many different logical fragments, like linear integer arithmetic, vectors, and unintepreted functions. The systems of polynomial constraints over real variables that we use in our verification is commonly referred to as QF_NRA for Quantifier-Free Nonlinear Real Arithmetic.

In 1951, Tarski published a decision procedure for deciding the truth of sentences of the elementary algebra of real numbers - and hence also of elementary geometry [Tarski 51]. In other words, Tarski proved that what we have set out to do is, in fact, computable. However, the algorithm he gave is far too inefficient to implement computationally. In 1975, Collins gave an algorithm for quantifier elimination in first-order statements on real arithmetic using Cylindrical Algebraic Decomposition (CAD), which runs much faster, albeit still in doubly-exponential time [Collins 75]. Nevertheless, this algorithm has proved critical to SMT solvers working with polynomial real arithmetic.

Classically, most SMT solvers work by mapping an input first-order formula, which consists of variables, Boolean connectives, and symbols of some other language *L*, into a Boolean formula, where every atomic formula (every top-level term using symbols other than the Boolean connectives) is mapped to a new Boolean variable. This Boolean formula is sent to a SAT solver, which gives a mapping of atomic formulae to Boolean values. These values can be checked for consistency by some procedure with knowledge of the interpretation of the symbols in *L*. In the event of an inconsistency being found (which was invisible to the SAT solver), the procedure sends back some additional conflict clause to the SAT solver to ensure the SAT solver will not repeat this mistake, at which point the procedure repeats until a satisfying assignment or a proof of unsatisfiability is found. Of note within this context is that the process of generating suitable conflict clauses is in some respects similar to removing a quantifier from a formula.

In 2013, de Moura and Jovanović published the Model Constructing Satisfiability (MCSAT) calculus, a new SMT framework that has been primarily applied to polynomial real arithmetic [De Moura and Jovanović 13]. Instead of a SAT solver assigning Boolean values to atomic formulae, the solver computes (and when necessary, guesses) real values for real variables. This decision calculus, combined with CAD, has provided a dramatic increase in recent years of the power of SMT solvers to solve some classes of formulae involving polynomial arithmetic [Jovanović and De Moura 13], and in fact provides the basis for two of the leading solvers of polynomial real arithmetic, Z3 [z3 ] and Yices2 [yic ]. Without these developments, the kind of reasoning about origami that we desire would be extremely difficult.

As mentioned above, SMT solvers can be used for 2D reference geometry calculations, but seem to be not suited well for 3D settings or even layered 2D fold settings (that is, settings where we fold the paper and then fold the paper again without unfolding the first fold). In a modeling system like mathematica, given some set of points numerically specified on a paper, after a fold operation, the program can easily identify on which side of the resulting fold line each point lies. Computation on these points later on, then, can be easily performed. But in an SMT setting, where computation is not performed fold by fold, if the paper is not unfolded after each operation, then an expression for where any given point lies in 2D space after one fold must be conditional on which side of the fold line it lies on. This leads to a combinatorial explosion in expression complexity.

We use two points on a line to define the line, as opposed to notions of angle, because SMT solvers do not generally work well with trigonometric functions and their inverses. We could equivalently use a (point, vector) notation for a line, as SMT solvers perform linear arithmetic quite easily, and one can transform a (point, point) definition to a (point, vector) definition via simple subtraction. However, normalizing the length of a vector for a mathematically cleaner representation using unit vectors or normal vectors requires computing a square root, which, if the root is irrational, can dramatically increase computation time (as all computations must be exact).

## 4 Verification Algorithm

Formally, our algorithm will, given a sequence of folds and a set of free variable declarations, produce a sequence of constraints (first order formulas) over real variables that is satisfiable if and only if the sequence of folds is foldable. In fact, it will do more. For each line and point folded in the paper, there are 4 (respectively, 2) real variables whose values, if one found a solution to the constraint system, would naturally define the line (respectively, point) in $\mathbb{R}^2$ that would be generated by performing the fold sequence. Our algorithm first validates that the input sequence of folds is well-defined. Then, for each fold in sequence, it performs the following operations:

- Declare enough new real variables to specify the new line (or point)
- Fetch the symbolic variables used to define the reference objects
- Look up the formula $\varphi(\mathbf{x})$ corresponding to the type of the fold
- Substitute in the above variables in the appropriate positions of the constraints, producing $\psi = \varphi[\mathbf{x}/\mathbf{t}]$
- Output $\psi$.

It is easy to see that, if all of the symbolic constraints accurately define the results of a fold operation and the algorithm performs the correct substitutions, then the resulting formulas will accurately capture the geometric behavior of the folds. It remains, then, to define a set of fold operations and to show, for each fold operation, how to symbolically compute the resulting fold based on its reference objects.

For our work, we consider only fold operations that produce a single fold. Justin gave a complete list of all possible single-fold operations in 1989 [Justin 89], which was independently discovered by Huzita and Hatori. We will denote them as $O1$ through $O7$, following the nomenclature of [Alperin and Lang 09]. Although some of the axioms are special cases of each other, to meet the community standard, we will provide the ability to process each axiom. This in fact can save much in performance, as the 5th axiom and especially the 6th axiom are, in the general case, quite difficult for SMT solvers to handle.

Proving a geometric predicate, however, requires a more complicated approach. First, as above, we will translate a predicate into a first-order formula $\varphi$ on some set of variables $V$. Likewise, we will translate the folding sequence on which the predicate should hold into a first-order formula $\psi$ on variables in $V$. We would like to show that $\forall \mathbf{v}(\varphi(v) \rightarrow \psi(v))$ - in other words, every way of realizing the fold pattern satisfies the geometric predicate. Of course, when the fold sequence uses as reference points only the edges and corners of the paper, then there is at most one way of folding the sequence. But if we allow some parameters in the origami construction to be variable, then this problem becomes nontrivial. As an aside, note that constraints on what the initial variables can be a part of $\varphi$, not $\psi$, as we are reasoning about the origami construction, and constraints on the free variables are part of the construction's definition.

SMT solvers determine whether or not a formula $\theta$ is satisfiable. But this is equivalent to asking whether or not $\exists x \theta(x)$ is true. Hence, we can accomplish the above goal by asking the solver whether or not $\exists \mathbf{v} \neg(\varphi(\mathbf{v}) \rightarrow \psi(\mathbf{v}))$. In actuality, we simplify the constraint systems and ask $\exists \mathbf{v} \varphi(\mathbf{v}) \wedge \neg \psi(\mathbf{v})$. This vacuously holds, however, if $\varphi$ is unsatisfiable. Thus, our verification consists of two steps - first, validating that $\varphi$ is satisfiable, and second, validating that $\varphi \wedge \neg \psi$ is unsatisfiable. Only when the first query returns satisfiable and the second returns unsatisfiable do we know that the geometric predicate always holds on the fold sequence.

It remains to show, then, precisely how to translate individual fold operations and geometric predicates into first-order formulae.

## 5 Fold Constraints

The first-order formulae used to specify the Huzita-Justin Axioms can be described as follows. We transcribe the first few formulas explicitly, but the longer folds, for convenience, are summarized in natural language. Some of the formulas have additional free variables used in intermediate computations.

O1
- Construct a line between two points.
- References: $p_1 = (x_1, y_1)$, $p_2 = (x_2, y_2)$
- Constructs: $l = (a_1, b_1, a_2, b_2)$
- Constraint: $(a_1 = x_1) \wedge (b_1 = y_1) \wedge (a_2 = x_2) \wedge (b_2 = a_2) \wedge ((x_1 \neq x_2) \vee (y_1 \neq y_2))$
- The formula above defines the line by the two input points, and asserts that the two input points are not equal.

O2
- Construct a line whose points are equidistant from two points.
- References: $p_1 = (x_1, y_1)$, $p_2 = (x_2, y_2)$
- Constructs: $l = (a_1, b_1, a_2, b_2)$
- Constraint: $(a_1 = (x_1 + x_2)/2) \wedge (b_1 = (y_1 + y_2)/2) \wedge (a_2 = a_1 + (-(y_2 - y_1))) \wedge (b_2 = b_1 + (x_2 - x_1)) \wedge ((x_1 \neq x_2) \vee (y_1 \neq y_2))$
- The formula above specifies two points. The first is the midpoint between the two input points. The second is the midpoint plus a vector perpendicular to the vector running from one input point to the other. These two points together define the fold line. The formula also asserts that the input points are not equal.

O3 • Construct a line by folding one line onto another. This fold has two options. For specificity, we will choose the fold that results in a line $l$ where the angle from $l_1$ to $l$ and the angle from $l$ to $l_2$ are positive.
• References: $l_1 = (x_1, y_1, x_2, y_2)$, $l_2 = (z_1, w_1, z_2, w_2)$
• Constructs: $l = (a_1, b_1, a_2, b_2)$
• Constraint: Let $parallel(l_1, l_2)$ be a formula that holds true if and only if $l_1$ and $l_2$ are parallel, where $l_i$ is shorthand for the corresponding tuple of real variables. Call the term $c$ the term that computes the cross product of the vectors $v_1 = (x_2, y_2) - (x_1, y_1)$ and $v_2 = (z_2, w_2) - (z_1, w_1)$.

If $l_1$ and $l_2$ are parallel, we want to assert that $(a_1, b_1)$ is the midpoint of $(x_1, y_1)$ and $(z_1, w_1)$ and that $(a_2, b_2)$ is the midpoint of $(x_1, y_1)$ and $(z_2, w_2)$ (which are guaranteed to be distinct). Let $\varphi(l_1, l_2, l)$ be the formula that makes this assertion .

If $l_1$ and $l_2$ are not parallel, we want to assert that $(a_1, b_1)$ is the intersection point of $l_1$ and $l_2$ and that $(a_2, b_2)$ is equal to $(a_1, b_1) + v_1/|v_1| \pm v_2/|v_2|$, where the plus or minus depends on whether or not the cross product of $v_1$ and $v_2$ is positive or negative (the cross product conveniently tells us how $v_1$ and $v_2$ are oriented relative to each other, which lets us correctly identify $l$ from the two possible options). Let $\psi(l_1, l_2, l)$ be the formula that makes this assertion. We will also assert that $(a_1, b_1)$ lies within the bounds of the paper.

Then our desired formula is $\theta(l_1, l_2, l, c) = (parallel(l_1, l_2, l) \wedge \varphi(l_1, l_2, l)) \vee (\neg(parallel(l_1, l_2, l)) \wedge \psi(l_1, l_2, l))$.

A careful observer might note that symbolically computing the intersection point of two lines can require dividing by a factor that, when the two lines are parallel, is 0. This would kill a naive numerical computation, but the standard for SMT solvers over the reals is to extend division to a total function, with $(1/0)$ being assigned to some value as necessary (so $(1/0) = x$) is a satisfiable formula but $(1/0) = x \wedge (1/0) = x + 1)$ is not) [Tinelli 17]. Interpreting such results would be nonsensical, but whenever a divide by 0 would occur in $\psi$, $\neg parallel$ evaluates to False and so that subformula is ignored. Hence, this technicality does not impede verification.

O4 • Construct a line perpendicular to a line $l$ and running through point $p$.
• References: $l_1 = (x_1, y_1, x_2, y_2)$, $p = (z, w)$
• Constructs: $l = (a_1, b_1, a_2, b_2)$.
• Constraint: The constraint here is simple. Assert that $(a_1, b_1) = (z, w)$ and that $(a_2, b_2) = (a_1, b_1) + v$ where $v$ is a vector perpendicular to the vector $((x_2, y_2) - (x_1, y_1))$.

O5 • Construct a line passing through a point $p_1$ that places some other point $p_2$ onto a line $l_1$.

Clearly, one point on $l$ will be $p_1$. Finding the other point is equivalent, however, to finding the intersection point of $l$ with a circle centered at $p_1$ with radius $|p_2 - p_1|$ (the second point on $l$ is $p_1$ plus a vector perpendicular to a vector running from $p_2$ to the intersection point). This has potentially two solutions. Arbitrarily, we declare that the first is the one reached first if one started at $p_2$ and walked clockwise around the circle. For this fold, the user must specify which of the solutions to use (equivalently, there are two fold operations $f_1$ and $f_2$ that each have at most one solution).

- References: $l_1 = (x_1, y_1, x_2, y_2)$, $p_1 = (z_1, w_1)$, $p_2 = (z_2, w_2)$.
- Constructs: $l = (a_1, b_1, a_2, b_2)$.
- Constraint: It is well known how to symbolically compute the intersection of a line and a circle. We construct a formula that does this, and in particular, contains two pairs of variables $s_1$ and $s_2$ whose values will represent both solutions to the line-circle intersection problem. We use the cross product of vectors running from $p_2$ to the solutions to determine which solution is the first or second solution.

  Let this chosen solution be $s$. Our formula can assert that $(a_1, b_1) = p_1$. If $p_2 = s$, then our formula can assert that $(a_2, b_2) = p_2$. Otherwise, it can assert that $(a_2, b_2)$ is equal to $p_1 + v$ where $v$ is a vector perpendicular to $p_2 - s$.

  Note that this fold has infinite solutions for $p_1 = p_2$. We do not explicitly rule out this input, but it should likely be avoided in order to obtain meaningful results.

O6
- Construct a line that maps point $p_1$ onto line $l_1$ and point $p_2$ onto line $l_2$.
- References: $p_1, l_1, p_2, l_2$
- Constructs: $l$
- Constraint: Computing a descriptor of such a line is equivalent to computing a line tangent to two parabolas. The two parabolas are defined with focus $p_i$ and directrix $l_i$. It is somewhat less well known how to compute this symbolically in a straightforward manner. For simplicity, we use the algorithm used in Lang's ReferenceFinder [Lang 17]. For more discussion of this fold, see [Lang 96].

  This kind of constraint has at most 3 solutions, so again, the user must choose one of the three solutions. We impose an ordering on the solutions, given by looking at the parabola defined by $p_1$ and $l_1$. Each solution's fold line will be tangent to this parabola at one point. We can project these points down to the directrix. If one orients one's view such that the directrix $l_1$ is the $x$-axis and the parabola points upwards, the $x$-coordinate of each point gives a natural ordering relation on the solutions, in which the first solution has the highest $x$-coordinate and the last solution has the lowest.

  We declare that if the user asks for the $i$th solution but there are not $i$ solutions in a particular instance of the problem, then the fold operation will fail (return $\perp$).

O7
- Construct a line that maps point $p$ onto line $l_1$ that is perpendicular to $l_2$.
- References: $p_1, l_1, l_2$
- Constructs: $l$
- Constraint: This constraint computes the intersection $p_1$ of a line parallel to $l_2$ running through $p$ with $l_1$. $l$ is then defined by $p_2 = (p + p_1)/2$ and by $p_2 + \mathbf{v}$ for some vector $v$ perpendicular to $l_2$.

INTERSECT
- Construct a point on the intersection of $l_1$ and $l_2$.

- References: $l_1, l_2$
- Constructs: $p$
- Constraint: We use well-known formulas to symbolically compute $p$ from $l_1$ and $l_2$. The constraint is necessarily unsatisfiable for any $l_1$ and $l_2$ that are parallel. We also assert that $p$ is within the bounds of the paper.

## 6 Predicates

At the moment, the predicates we assert are Boolean combinations of:

- $parallel(l_1, l_2)$ holds true if and only if $l_1 = (x_1, y_1, x_2, y_2)$ and $l_2 = (z_1, w_1, z_2, w_2)$ are parallel. $parallel$ can be written as $(x_2 - x_1) * (w_2 - w_1) = (z_2 - z_1) * (y_2 - y_1)$.
- $perpendicular(l_1, l_2)$ holds true if and only if $l_1$ and $l_2$ are perpendicular. This formula can be easily written in a manner similar to $parallel$.
- $colinear(l, p)$ holds if and only if $p$ lies on line $l$. This predicate can be easily reduced to $parallel$, in that if $l$ is defined by two points $p_1$ and $p_2$, then $p$ lies on $l$ if and only if $l$ is parallel to the vector from $p_1$ to $p$.

We also allow assertions of polynomials of distances between points. Denote the distance between points $a$ and $b$ as $d(a,b)$ The formula $\varphi(z, x_1, y_1, x_2, y_2) := (z^2 = (x_1 - x_2)^2 + (y_1 - y_2)^2) \wedge (x \geq 0)$ is satisfied only by $z = d((x_1, y_1), (x_2, y_2))$. The user can specify formulas like $(d(x,y) + d(y,z) * d(z,w) + 3) \leq (4 - d(x,w))$. Our translation algorithm creates new variables for each $d(a,b)$ mentioned in the user's constraints, asserts $\varphi$ with the appropriate subsitutions about the new variable $d_{ab}$ and then asserts the user-supplied constraint, with $d_{ab}$ substituted in for $d(a,b)$. To reduce redundant constraints, we use the same variable for $d(a,b)$ as $d(b,a)$. Distances are normalized such that the paper is a unit square.

## 7 Results

The project can be found at https://github.com/gramseyer/origami_smt, along with usage instructions and a formal description of the input fold specification language. The system includes the option to output constraints in the SMTLIB2 format [Barrett et al. 17], a format accepted by most modern SMT solvers. The project also contains limited tools for visualizing the a satisfying solution to a constraint. All tests were performed on a Core i7 7700-HQ quad-core processor running at 2.8 GHz. All times are averaged over 5 trials and had a 10 minute time out.

As a demonstration, we can prove that the commonly used sequence of folds in Figure 1 divides the top of the paper into three equal pieces. An encoding of the above folds can be found at “inputs/thirds.ori”.

The first step, showing that the construction is foldable, took 0.32 seconds, and the second step, proving that the top of the page is, in fact, divided into equal pieces took 0.19 seconds.

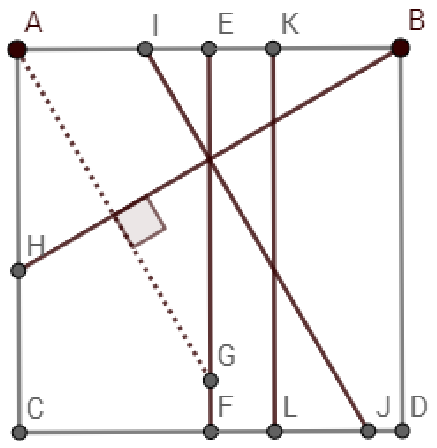

**Figure 1:**

1. *(O2) Fold line EF by folding point A onto point B.*
2. *(O5, solution 2) Fold line BH running through point B by moving point A onto line EF.*
3. *(Intersect) Find point H at the intersection of lines AC and BH.*
4. *(O2) Fold line IJ by folding point B onto point H.*
5. *(Intersect) Find point I at the intersection of lines IJ and AB.*
6. *(O2) Fold line KL by folding point B onto point I.*
7. *(Intersect) Find point K at the intersection of lines KL and AB.*
8. *Points I and K break line AB into 3 equal pieces.*

But we could have done the above simply by numerical computation. To make the example more interesting, we will prove that a slight adjustment to the above fold sequence will trisect any segment of the top of the page. In essence, we will replace every instance of "upper left corner" with a free variable "x," every instance of "upper right corner" with a free variable "y," and "the left edge of the paper" with "a vertical line through x" (folded with O4), and assert that x and y lie on the top of the page.

An encoding of the modified fold sequence can be found at "inputs/var2thirds.ori"

The foldability step took 0.77 seconds, and the assertion step took 0.23 seconds.

A note on the computation time - if one interpolates between the above two examples (only one free variable, and the assertion is about any segment starting at the upper left corner - see "inputs/varthirds.ori"), the proof steps take 0.25 seconds and 0.19 seconds. Observe that while in general, the computation time tends to increase with fold sequence complexity, the relationship can be obfuscated by the idiosyncracies of a given problem instance. This is a phenomenon observable in many other SMT applications.

However, if one attempts to look at applying this fold construction to any line segment, the foldability analysis step fails. As the variance and number of parameters to the construction increases, the size of the space in which the SMT solver much search for solutions increases. For the case of trisecting a segment, sometimes a segment cannot be trisected using this construction because various line intersections in the construction would occur off of the paper. This makes even the first step of the proof difficult for an SMT solver.

As a simple example, we can prove the triangle inequality by declaring three free variables and asserting the triangle inequality. A formalization can be found at ”inputs/triangleinequal.ori”. Without loss of generality, we can assume that one point lies on the top of the page. If we perform this construction, the first step of the proof took 0.18 seconds, while the second step took 10.5 seconds. However, if we let the SMT solver use the full search space (no constraint on one of the variables), then the solver times out.

Another use case for our algorithm could be in bounding the error in the points defined by some folds given a bound on error in the reference points. Suppose, for example,one folds both diagonals of a square, but the first fold is made slightly in error. If one accurately folds the second diagonal to be perpendicular to the first, what is the worst error that can be propagated to the second diagonal? Formalized, this would be the following:

Consider a point $x$ that is distance $d$ from $(1,0)$. If we fold a line $L_1$ running through $x$ and $(0,1)$ (O1), then fold a line $L_2$ perpendicular to $L_1$ running through $(1,1)$, how far from $(0,0)$ can $L_2$ be? The answer is $d$. A formalization of this proof can be found at ”inputs/error.ori”. The first step of the verification took 0.19 seconds, and the second step took 0.17 seconds.

## 8 Related Work

Automated deductions about geometric statements have been an open research area for the last several decades. Tarski in 1951 showed that decision problems about multivariate polynomials were decidable [Tarski 51]. As many polynomials can be encoded into planar geometry, this result opened the possibility for automated reasoning about geometry and the possibility of better algorithms. For example, in 1978 Wu published his own decision procedure for geometric statements and used it to prove a few theorems of planar geometry [Wu 78].

From the origami side of things, there has been much work in analysis and simulation of origami systems. Lang’s Tesselatica is a Mathematica package that provides tools for simulating tessellations [Lang 18]. In 2014, at *OSME*[6], many authors presented works involving reasoning about origami computationally. Xi and Lien, for example, demobstrated a tool for computationally determining the distinct shapes into which a crease pattern can fold [Xi and Lien ].

Most similar to this work is the E-Origami System (Eos), a mathematica-based project to model origami and perform automated reasoning about origami systems. This long running project represents origami folds as logical formulae, primarily through defining new relations on variables, and then translating these relations into a question of membership in an ideal in the polynomial ring. They solve this using Gröbner basis computation [Ida et al. 11].

The primary difference between our work and Eos is that proofs in Eos may require human management, whereas ours will be entirely automated. In particular, in Eos, the construction of an origami model within a computer is an interactive process. At each fold, Mathematica must solve a constraint satisfaction and manipulation problem, which can be done semi-automatically. And at the proof stage, sometimes users inject additional hypotheses into the system in order to aid the prover. By contrast, in our system, once the user writes down a sequence of folds, specified by reference points and lines and how these reference objects should be used, the user has no more input into the proof generation.

Because our system is fully automated, the user need not have a deep mathematical understanding of the underlying algebra. It therefore is much easier to use by a nontechnical audience and its operation by any user requires less effort than an interactive system. In exchange for this ease of use, the user gives up the ability to help the prover. In the worst-case scenario where a proof fails, the user is unable to help the proof along by injecting additional information. However, the author's motivation for building this system involved annoyance at repeated mistakes in reasoning about reference fold geometry, so the author is disinclined to trust a proof that relies on additional user-provided hypotheses. That said, when an SMT solver cannot solve an instance of a problem, the user could refactor the problem to perhaps make it easier for the SMT solver.

We also reason about a slightly smaller class of origami models than it seems Eos can manipulate - after every fold, we perform an unfold operation. Of course, when folding reference points, this is a natural thing to do, but it does give up some generality. In [Ida et al. 11], the Eos proof is about an origami system where each fold is followed by an unfold operation, but this may not need hold true for Eos to operate in general.

In a different direction, in 2016 Ida, Fleuriot, and Ghourabi developed a new system of representing origami manipulations in 2 and 3 dimensions [Ida et al. 16]. In essence, it involves a new algebraic structure for representing origami. Using the proof assistant Isabelle, they validate that their structure behaves as they expect it to, allowing them to, with proof of correctness, translate origami folds into constraints involving polynomials. It would be an interesting experiment to see how these equations would perform with an SMT solver.

## 9 Future Work

The problem of determining satisfiability of formulas involving polynomials over real variables is in NP. SMT solvers, then, can only attempt to solve a subset of all possible instances of problems. Yet as solvers improve over time, and as computer hardware gets faster and faster, the reliability of the verification should improve.

However, the ability of an SMT solver to solve a problem can vary dramatically with the presentation of the problem. For example, if one rearranges the ordering of clauses in a large formula, sometimes the speed at which a solver terminates varies dramatically. Some problem instances that are solvable in seconds can, after rearrangement, remain unsolved after hours of computation. It remains to be seen whether or not there is some way of representing folds, lines, and points in a way that is more amenable to SMT solvers.

In particular, we chose the two-point representation of lines, instead of the perhaps more mathematically elegant definition involving normal unit vectors used in programs like Lang's ReferenceFinder, because taking the norm of a vector means taking a square root. If the resulting value is irrational, then computation becomes vastly more complicated (and slower). As such, we generated our formulas keeping in mind a desire to minimize the number of such complex operations.

Likewise, certain classes of formulas that are quite straightforward conceptually can be very difficult for a solver. For example, one early attempt at transcribing O3 (fold one line onto another line) involved, in essence, telling the solver to compute the intersection of the two input lines $l_1$ and $l_2$, and then pick two points $p_1$ and $p_2$ such that $p_i$ is on $l_i$ and $p_1$ and $p_2$ are equidistant from the intersection point. The resulting line would then run through the intersection point and the midpoint of $p_1$ and $p_2$. This failed often, however, likely because it involved making a nondeterministic step (choose $p_i$ such that...). Our current version of O3 improves on this by computing $p_i$ as a function of the points that define $l_i$, which works since any pair of $p_1, p_2$ satisfying the above predicate will do. Even though this involves normalizing a vector, solvers seem to reason about this formula quite easily.

The performance of O6 still leaves substantial room for improvement. In particular, the verification complexity comes from at least two sources. First, the fold is equivalent to solving a cubic equation. 3 is a small number, but when one considers that the coefficients of the cubic are also all symbolic variables, and often complex expressions, the axiom turns out to actually involve finding roots of a polynomial of much higher degree. And second, O6 works by computing all distinct solutions of the polynomial, and then returning the solution specified by the user. This involves, in essence, satisfying $(\exists x_1,x_2,x_3)\ \varphi(x_1) \wedge \varphi(x_2) \wedge \varphi(x_3) \wedge (x_1 \neq x_2) \wedge (x_1 \neq x_3) \wedge (x_2 \neq x_3) \wedge (x_1 < x_2) \wedge (x_2 < x_3)$, where $\varphi(x)$ holds if and only if $x$ is a root of the appropriate polynomial. SMT solvers take a local view of distinctness constraints, which means that finding sets of disequal variables can take an extremely long time. In the worst case, finding $n$ distinct variables in $[n]$ can require enumerating over $[n]^n$. This is made worse by the fact that when SMT solvers guess variable assignments to test satisfiability, they often use similar processes for each variable. That is to say, in the absence of other information (like bounds on the variable), the first guess might be 0, the second might be 1, and so on. Often, therefore, finding $x, y, z$ disequal can mean guessing $(0,0,0)$, realizing that $x_2 = x_3$ and backtracking, guessing $(0,0,1)$, realizing that $x_1 = x_2$ and backtracking, and then guessing $(0,1,0)$, etc. It remains to be seen if there is some other way of formulating O6 that circumvents these difficulties. The main blocking factor is the lack of a general non-trigonometric closed form solution to a cubic equation.

Under this constraint framework, one can easily plug in constraints that correspond to multi-fold operations. However, the number of possible 2-fold operations numbers in the hundreds [Alperin and Lang 09], and most folders do not use such complex operations very often, so we have not included them all here. A small subset of interest, though, could be added quite simply.

Similarly, additional geometric constraints would be quite easy to plug into the software architecture, once one finds a suitable formula. Predicates like these two angles are equal could be implemented feasibly using vector cross products and dot products, as these give us relationships between the sines and cosines of two angles and $(\sin(\theta_1) = \sin(\theta_2)) \wedge (\cos(\theta_1) = \cos(\theta_2)) \iff \theta_1 = \theta_2$.

Suggestions on improvements to the user interface and fold specification language would be much appreciated. Moreover, when the SMT solver returns a satisfactory result, the result comes in the form of real values assigned to variables. The variables are named to correspond with the lines and points they represent, so an experienced user can read off the numerical results without much difficulty, but it is not a terribly intuitive process. I built a simple result visualizer, but more work in this area could be a great improvement for usability.

I would also like to experiment with allowing approximate solutions to difficult folds like O6. However, maintaining sensible bounds on error propagating through the fold sequence is a difficult challenge. Bounding the distances between an exact point and its approximation by an error term is insufficient. For example, approximate the point $x = (0, \varepsilon)$ by $x' = (\varepsilon, \varepsilon)$. Then the line defined by $(0,0)$ and $x$ runs through $(0,1)$, while the line defined by $(0,0)$ and $x'$ runs through $(1,1)$. A more detailed solution is needed, but the more detailed solution must also not be too complex, or else the constraint system's complexity will not reduce.

## 10 Conclusion

In this work, I present a system for computationally verifying predicates about the geometries generated by origami folds. It can prove statements about many straightforward fold patterns, although it struggles to prove predicates when the search space is especially large, and it struggles to reason about computationally difficult folds like those generated by the O6.

As the space of all SMT problems is very large, and solving all instances is impossible unless P=NP, in most SMT literature, solvers measure their performance against classes of benchmarks submitted from their users, often from industrial applications [ben ]. However, no such set of benchmarks exists for origami fold sequences. I have generated some examples for testing the system, but these examples are likely not representative of all the things folders might like to do. To that end, I would like to solicit submissions of fold sequences and geometric predicates against which to measure the behavior of the solver. Without a representative sample of benchmarks, one cannot effectively evaluate the relative performance of a change in the verification process.

Some difficult constructions can be made easier for the solver by breaking the computation into steps. If, say, a difficult fold sequence depends on no input variables the user can validate the difficult part separately, and then replace its dependencies with numerical or algebraic specifications.

## References

[Alperin and Lang 09] Roger C Alperin and Robert J Lang. "One-, two-, and multi-fold origami axioms."

[Barrett et al. 17] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. "The SMT-LIB Standard: Version 2.6." Technical report, Department of Computer Science, The University of Iowa, 2017. Available at `www.SMT-LIB.org`.

[ben ] “SMT-LIB Benchmarks.” Available online (http://smtlib.cs.uiowa.edu/benchmarks.shtml).

[Collins 75] George E Collins. “Quantifier elimination for real closed fields by cylindrical algebraic decompostion.” In *Automata Theory and Formal Languages 2nd GI Conference Kaiserslautern, May 20–23, 1975*, pp. 134–183. Springer, 1975.

[De Moura and Jovanović 13] Leonardo De Moura and Dejan Jovanović. “A model-constructing satisfiability calculus.” In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pp. 1–12. Springer, 2013.

[Ida et al. 11] Tetsuo Ida, Asem Kasem, Fadoua Ghourabi, and Hidekazu Takahashi. “Morleys theorem revisited: Origami construction and automated proof.” *Journal of symbolic computation* 46:5 (2011), 571–583.

[Ida et al. 16] Tetsuo Ida, Jacques Fleuriot, and Fadoua Ghourabi. “A new formalization of origami in geometric algebra.” *ADG2016*, p. 117.

[Jovanović and De Moura 13] Dejan Jovanović and Leonardo De Moura. “Solving non-linear arithmetic.” *ACM Communications in Computer Algebra* 46:3/4 (2013), 104–105.

[Justin 89] Jacques Justin. “Résolution par le pliage de léquation du troisieme degré et applications géométriques.” In *Proceedings of the first international meeting of origami science and technology*, pp. 251–261. Ferrara, Italy, 1989.

[Lang 96] Robert J Lang. “Origami and geometric constructions.” *Self Published (1996 2003)*.

[Lang 06] Robert J Lang. “TreeMaker.”, 2004-2006. Available online (http://www.langorigami.com/article/treemaker).

[Lang 17] Robert J Lang. “ReferenceFinder.”, 2004-2017. Available online (http://www.langorigami.com/article/referencefinder).

[Lang 18] Robert J Lang. “Tessellatica.”, 2014-2018. Available online (http://www.langorigami.com/article/tessellatica).

[Tarski 51] Alfred Tarski. “A Decision Method for Elementary Algebra and Geometry.”

[Tinelli 17] Cesare Tinelli. “Reals.”, 2010-2017. Available online (http://smtlib.cs.uiowa.edu/theories-Reals.shtml).

[Wu 78] WEN-TSN Wu. “On the Decision Problem and the Mechanization of Theorem-Proving in Elementary Geometry.” *A ()* 2 (1978), 001.

[Xi and Lien ] Zhonghua Xi and Jyh-Ming Lien. “Determine distinct shapes of rigid origami.”

[yic ] “Yices2.” Available online (http://yices.csl.sri.com/).

[z3 ] “Z3.” Available online (https://github.com/Z3Prover/z3/wiki).

Geoffrey Ramseyer
Stanford University, 353 Serra Mall, Stanford, CA 94305, e-mail: geoff.ramseyer@cs.stanford.edu