Using an Analytical Computational-Geometry Library to Model Nonoverlap and Boundary-Distance Constraints and their Application to Packing Poly-Bézier Shapes

Paul B. Morton (pmorton@monarchip.com)

Abstract

In this paper we will show how to model nonoverlap as well as uniform and nonuniform boundary-distance constraints between poly-Bézier shapes using an analytical computational-geometry library. We then use this capability to develop, implement and analyze analytical-optimization solutions to minimum-area rectangular-boundary packing-problems as well as minimum-area one- and two-dimensional puzzle-piece packing-problems. In the process, we will demonstrate the ease and efficiency with which analytical-optimization solutions to complex packing-problems can be formulated, implemented, solved, and analyzed using these models.

Key words: Modeling Nonoverlap Constraints, Modeling Boundary-distance Constraints, Uniform Boundary-Distance Constraint, Nonuniform Boundary-Distance Constraint, Nonlinear Optimization, Shape Packing, Poly-Bézier Shapes, Puzzle-Piece Packing, Cylindrical Packing, Analytical Computational-Geometry Library, ACGL, IPOPT.

1 Introduction

Shape-packing is a ubiquitous and important problem across a wide range of disciplines, industries, and applications. Traditionally packing problems have been solved using specialized combinatorial algorithms [1]. Unfortunately, without help from a small group of domain experts, it is difficult to apply these algorithms beyond the narrow range of packing problems each algorithm was originally designed to solve. At the same time, it has long been understood that, in theory, a wide range of complex packing-problems could be formulated as analytical optimizations and solved using “off the shelf” commercial and open source nonlinear-optimization software [2] [3] [4] [5]. Additionally, formulating a packing problem as an analytical optimization allows the packing problem to be easily integrated into the analytical design-processes commonly used in industrial product design and engineering. To date, however, we have been unable to realize most of the benefits of this theoretical capability due primarily to the lack of mathematical-models and methods which are needed to easily and reliably construct efficient nonoverlap and boundary-distance constraints for a wide range of shapes.

Over the past twenty years concerted efforts have been made to alleviate this constraint-modeling bottle-neck [6] [7] [8] [9] [10] [11] [12] [13] [14]. Unfortunately, the underlying mathematics and specialized optimization-methods used in these approaches have proven to be difficult to extend for use beyond relatively simple constraints between elliptical shapes [15]. Further, the resulting models lack the

software ecosystem necessary for use by a broad range of users who may not be highly-skilled software-engineers and/or may have a limited understanding of the underlying mathematics and optimization methods which form the foundation of the models. As a consequence, these models are, again, only suitable for use by a small group of domain experts.

More recently, an analytical computational-geometry library (ACGL™) [16] [17] has been developed and implemented in C++. Using a new mathematical foundation having greater flexibility and expressiveness, ACGL seamlessly integrates the constructive or compositional elements of combinatorial computational-geometry [18] with the curved shapes and analytical elements of traditional analytical-geometry as well as the more recent geometric-modeling techniques [19] of numerical computational-geometry. This library provides the ecosystem necessary for a wide range of user, with only a modest amount of knowledge in software engineering and mathematical modeling, to easily and reliably formulate and construct complex nonoverlap and boundary-distance constraints between a wide variety of shapes. Specifically, ACGL provides a wide range of fundamental shapes, both curved and polygon, along with a set of composition and transformation operations from which constraints can be constructed. This is done using a simple and flexible nested-instantiation syntax which has been designed to be a natural and intuitive extension of C++ native syntax. The shapes and operations are designed to work at a high level of abstraction. This abstraction eliminates the need to keep track of the myriad low-level details and interactions, which would otherwise make the formulation and construction of complex constraints for use by an analytical-optimizer tedious, alarmingly error-prone, and prohibitively time-consuming.

In addition, ACGL provides extensive automated parameterization capabilities allowing for selective parameterization of the shapes, compositions, and transformations used in constraint construction. The parameterization process is further augmented by binding each parameter to a C++ variable during constraint construction. These constraints can automatically calculate the derivatives with respect to their selected parameters. In turn, the derivatives can be retrieved using a natural and intuitive extension of the C++ array syntax. These automated parameterization capabilities dramatically simplify the Jacobian construction process for complex constraints.

In this paper we will show how nonoverlap and boundary-distance constraints can be easily formulated and constructed using this library. We will then use them to develop and implement simple, reliable and efficient analytical-optimization solutions to a sequence of successively more complex packing problems. These packing problems include:

- Packing a set of variable-orientation poly-Bézier shapes into a minimum-area rectangular packing-boundary using nonoverlap constraints.
- Packing a set of variable-orientation poly-Bézier shapes into a minimum-area rectangular packing-boundary using uniform and nonuniform boundary-distance constraints.
- Packing a set of variable-orientation poly-Bézier shapes to form a minimum-area one-dimensional fixed-height puzzle-piece packing using nonoverlap and uniform boundary-distance constraints.
- Packing a set of variable-orientation poly-Bézier shapes to form a minimum-area two-dimensional puzzle-piece packing using nonoverlap constraints

Figure 2 through *Figure 5* give a preview of the packings that result when each of these four solutions are applied to the collection of 25 poly-Bézier shapes shown in *Figure 1*.

Figure 2 shows the packing results when the 25 shapes are packed into a minimum-area rectangular-boundary using nonoverlap constraints while allowing the shapes to rotate freely.

Figure 3 shows the packing results when the 25 shapes are packed into a minimum-area rectangular-boundary using uniform boundary-distance constraints which enforce a minimum spacing between adjacent shapes while allowing the shapes to rotate freely.

Figure 4 shows the packing results when the 25 freely rotating shapes are packed to form a minimum-area one-dimensional fixed-height puzzle-piece packing using nonoverlap constraints. The puzzle-piece packing is shown in blue and bracketed to the left and right by copies of its self, shown in red.

Figure 5 shows the packing results when the 25 freely rotating shapes are packed to form a minimum-area two-dimensional puzzle-piece packing using nonoverlap constraints. The puzzle-piece packing is shown in blue and surrounded by eight copies of its self, shown in red and black.

Figure 1 shows the initial positions and orientations of these 25 poly-Bézier shapes. Due to the number of shapes and the high degree of overlap between shapes, the shape numbers have been omitted.

The remainder of this paper is broken down into two main parts. In Part 1, consisting of *Sections* 2 and 3, we provide a high-level description of the ACGL concepts and functionality that will be needed to develop and implement the solutions developed in Part 2. In Part 2, consisting of *Sections* 4 through 7, we develop, implement, demonstrate, and analyze the analytical-optimization solutions for the four shape-packing problems outlined above.

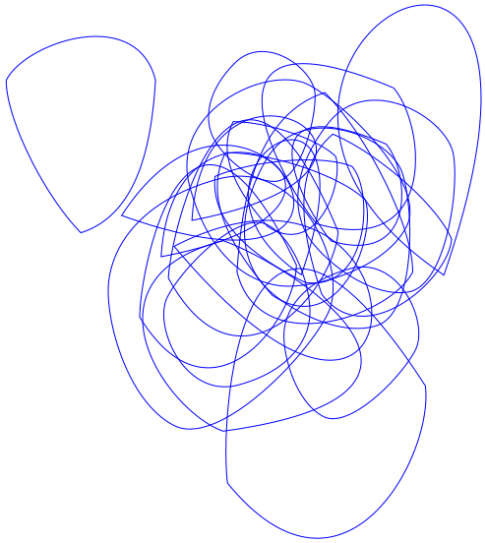

Figure 1: Initial positions and orientations of the 25 poly-Bézier shapes used in the packings of *Figure 2* through *Figure 5*.

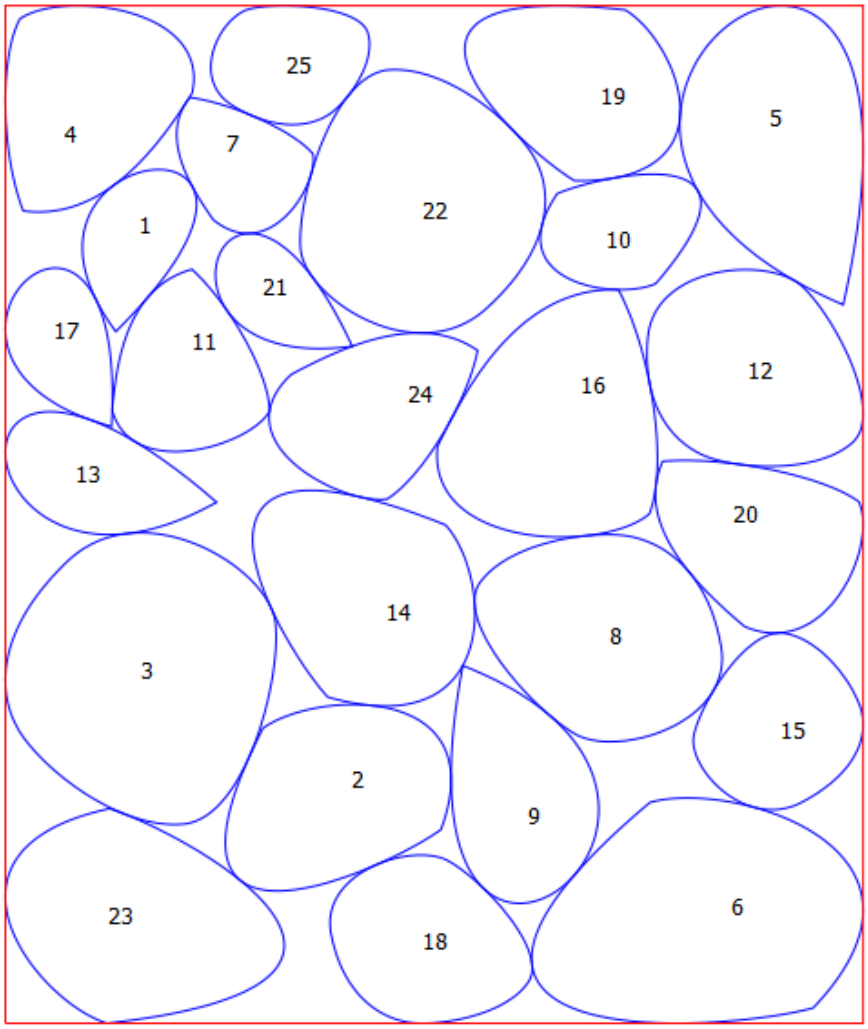

Figure 2: Minimum-area rectangular packing of 25 poly-Bézier shapes using nonoverlap constraints.

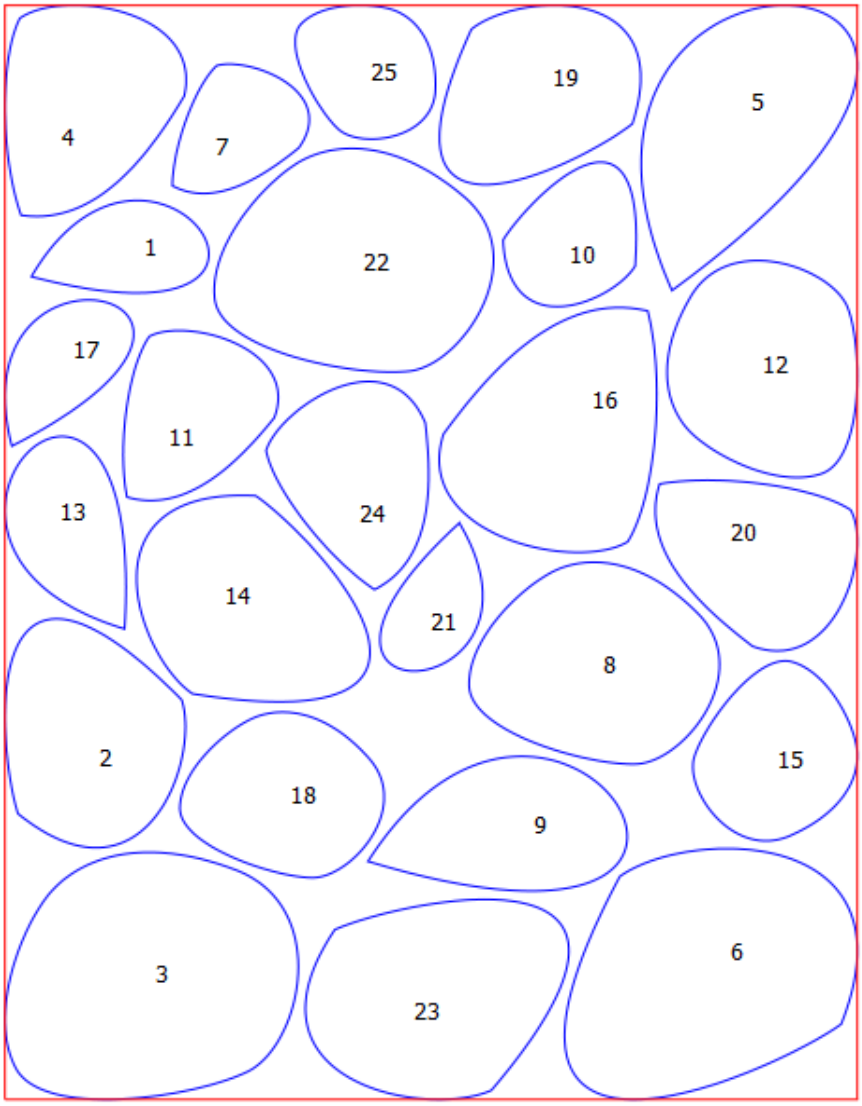

Figure 3: Minimum-area rectangular packing of 25 poly-Bézier shapes using uniform boundary-distance constraints.

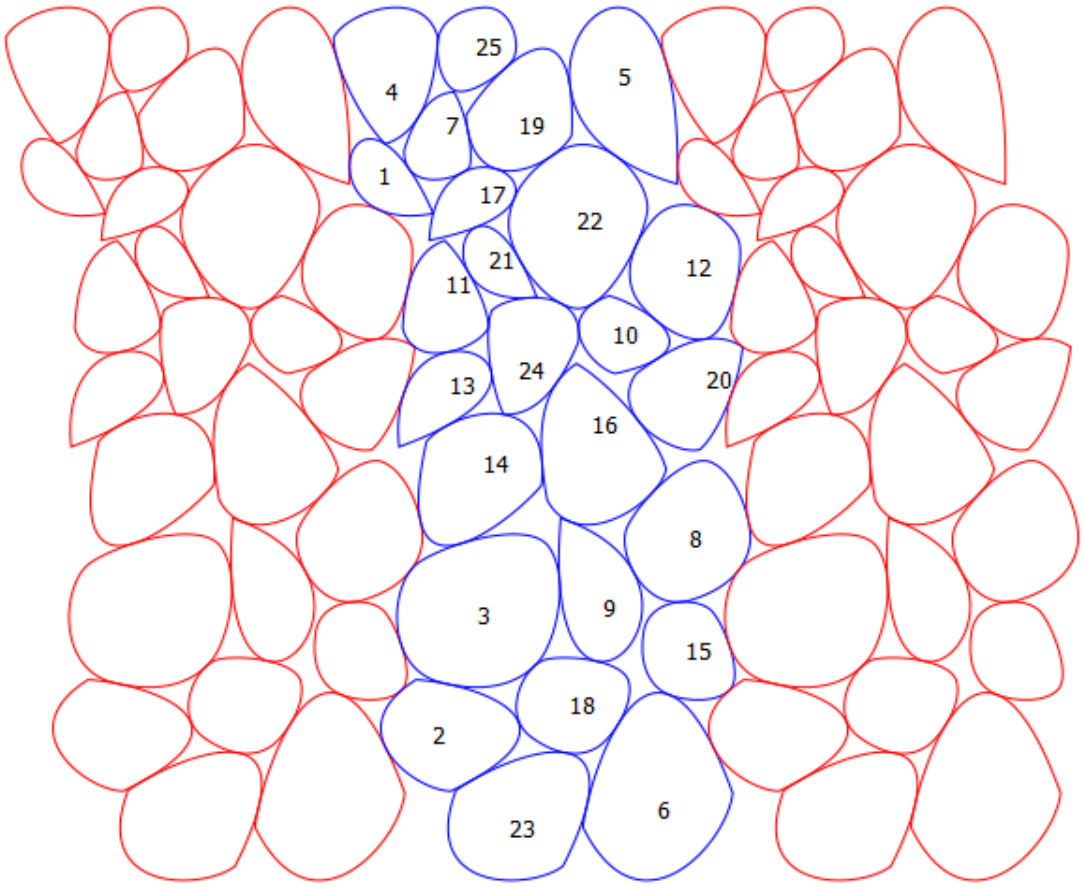

Figure 4: One-dimensional fixed-height puzzle-piece packing of 25 poly-Bézier shapes (shown in blue) bracketed by copies of its self (shown in red) to its left and right.

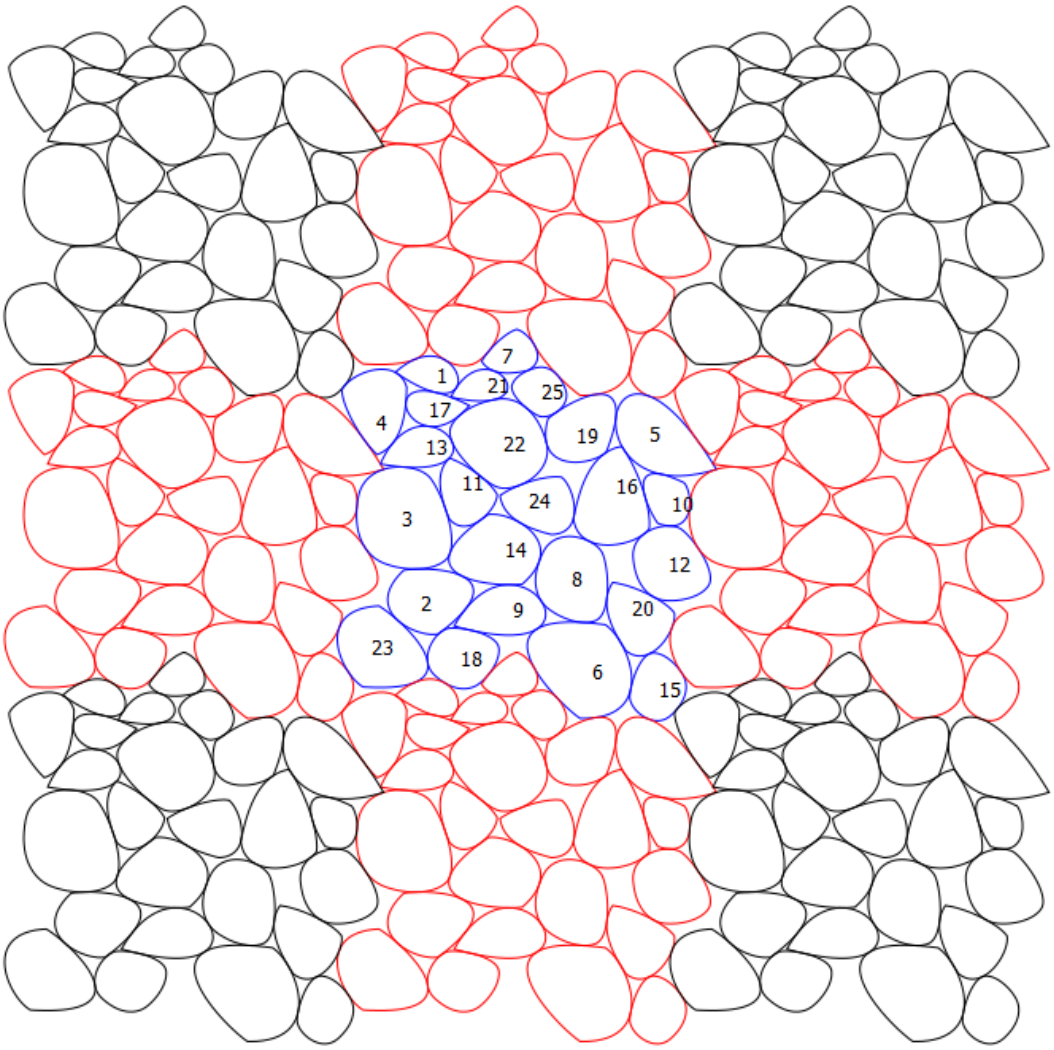

Figure 5: Two-dimensional puzzle-piece packing of 25 poly-Bézier shapes (shown in blue) surrounded by eight copies of its self (shown in red and black).

Contents

2 Shape-boundary

ACGL's basic building-block is a conceptually simple object called a "shape-boundary". Specifically, for a shape, *S*, as illustrated in *Figure 6*, its shape-boundary, SB_S, is a "description" of the geometry of the boundary of S, *Boundary*(*S*), relative to its origin, *Origin*(*S*). More specifically, a shape-boundary ONLY contains information on the geometry of a shape's boundary relative to its origin and contains NO information on the location of the shape's origin in space (the need for this distinction will become apparent in *Section 2.4*).

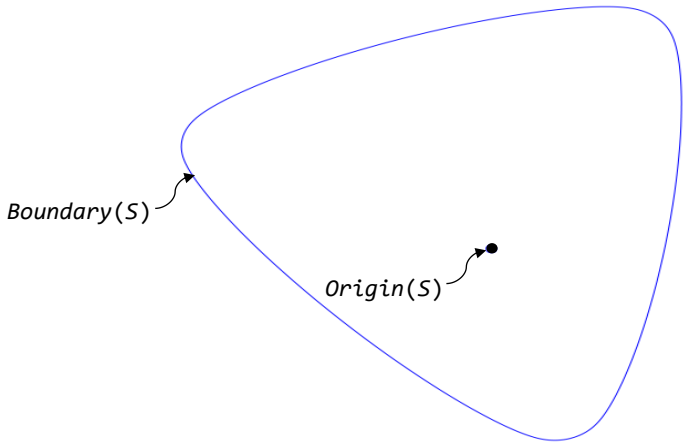

Figure 6

Further, shape-boundaries can be formed in three ways: directly from predefined basic-shapes; using transformations; or through compositions.

The basic shape-boundaries are a set of predefined shapes from which all other shape-boundaries will be derived. This set consists of a group of analytic shapes including: point, line, square, rectangle, and parallelogram as well as: circle, ellipse, superellipse, and generalized-superellipse (the superelliptical equivalent of a parallelogram). Recently this set of basic shapes has been expanded to include a prototype implementation for representing convex poly-Bézier shapes. This new basic shape makes it easier to work with more organic shapes which are not easily represented using compositions of the basic analytic-shapes and will be introduced in *Section 2.2*.

The transformations are formed by taking an existing shape-boundary and transforming it into a new shape-boundary through the use of operations such as: rotation, scaling, and offset.

The compositions are formed by taking two or more existing shape-boundaries and composing them into a new shape-boundary. Compositions include: convex hull, boundary-surface of closes-approach, and piece-wise composition.

2.1 Analytic Shape-boundaries

While ACGL provides many analytic shape-boundaries, we will only describe the circle, ellipse and square boundaries, as these will be the only analytic shape-boundaries used in this paper. In particular, they will be used for illustrative purposes and in constructing the boundary-distance constraints for the packing-problem solutions developed in *Sections 5* and *6*.

2.1.1 Circle

The *CircleBoundary* creates, as shown in *Figure 7*, circular boundaries. The *CircleBoundary* has one *double* parameter, its *Diameter*.

The following line of C++ code creates a *CircleBoundary*, `C`, with a diameter of 1.5:

```
CircleBoundary C(1.5);
```

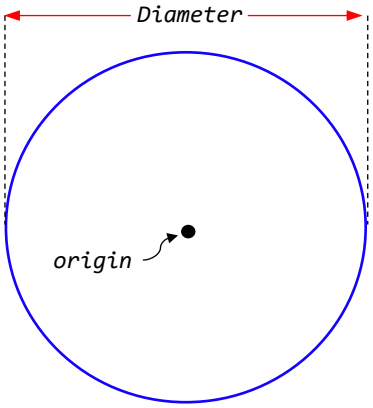

Figure 7

2.1.2 Ellipse

The *EllipseBoundary* creates axis-aligned elliptical boundaries. The *EllipseBoundary* has two *double* parameters: *Diameter1* and *Diameter2*. *Diameter1* is the X-axis diameter and *Diameter2* is the Y-axis diameter as shown in *Figure 8*.

The following line of C++ code creates an *EllipseBoundary*, `E`, with X-axis diameter of 1.5 and a Y-axis diameter of 1.0:

```
EllipseBoundary E(1.5, 1.0);
```

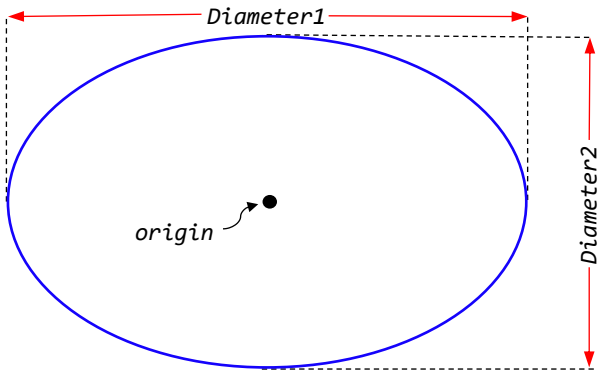

Figure 8

2.1.3 Square

The *SquareBoundary* creates, as shown in *Figure 9*, axis-aligned square boundaries. The *SquareBoundary* has one *double* parameter, its side *Length*.

The following line of C++ code creates a *SquareBoundary*, `S`, with a side length of `1.0`:

```
SquareBoundary S(1.0);
```

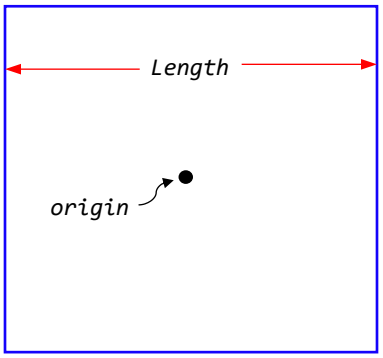

Figure 9

2.2 Poly-Bézier Shape-boundary

While ACGL provides a wide range of analytic shapes it can still be challenging to represent more organic shapes common in many industrial applications. To address this shortcoming a prototype capability has been developed to makes it much easier to represent shapes with more organic geometries. In particular, these shapes can now be represent using a sequence of cubic Bézier-curves, more commonly known as a composite-Bézier or poly-Bézier shape.

Bézier curves are a widely used and well understood means for efficiently capturing the geometry of a wide range of curves [20] [19] and have a history of use in industrial applications dating back over half a century [21] (for an excellent interactive introduction to Bézier curves, please see [22]). As a consequence, poly-Bézier shapes provide a compact representation of exact, or close approximations, for virtually any convex shape.

A cubic Bézier-curve is a parametric curve defined by a pair of cubic polynomials whose coefficients are derived from a list of four control points. As shown in *Figure 10* and *Figure 11*, the 1st and 4th control points (shown in red) define the end points of the curve. The 2nd and 3rd control points (shown in green), in combination with the end points, define the shape of a smooth curve traced out between the end points. In particular, the line through the 1st and 2nd points is tangent to the curve at the 1st point and the line through the 3rd and 4th point is tangent to the curve at the 4th point. Additionally, note that these two Bézier-curves having identical end points but dramatically different curves due to the difference in the position of the 2nd and 3rd control points for each curve.

The *PolyBezierBoundary* creates shape-boundaries which are composed of one or more connected Bézier-curves. Specifically, the end point of one curve is the starting point of the next curve, and the end point of the last curve is the starting point of the first curve.

Figure 12 through *Figure 15* each show an illustrative example of a *PolyBezierBoundary*:

- *Figure 12* shows a tear-drop shaped *PolyBezierBoundary* composed of a single Bézier-curve. Note that because this *PolyBezierBoundary* consists of a single Bézier-curve its 1st and 4th points are the same point.
- *Figure 13* and *Figure 14* show shape-boundaries composed of two and three Bézier-curves, respectively. Both of these shape-boundaries are non-smooth. This is due to the fact that the 3rd and 4th control points of each curve are not colinear with 1st and 2nd control points of the adjoining curve. Because of this, the end of one curve and the beginning of the next are not tangent to the same line and form a “kink” or an abrupt transition in the shape-boundary at the point where the two curves connect.
- Finally, *Figure 15* shows a smooth *PolyBezierBoundary* composed of four Bézier-curves. This shape-boundary is smooth due to the fact that the 3rd and 4th control points of each curve are colinear with 1st and 2nd control points of the adjoining curve. Because of this, the end of one curve and the beginning of the next are tangent to the same line and form a smooth transition between the two curves.

These four poly-Bézier shapes-boundaries, in combination with the scaling and rotation transformations, will be used in the construction of sets of shapes to be used in demonstrating the packing-problem solutions developed in *Sections 4* through *7*.

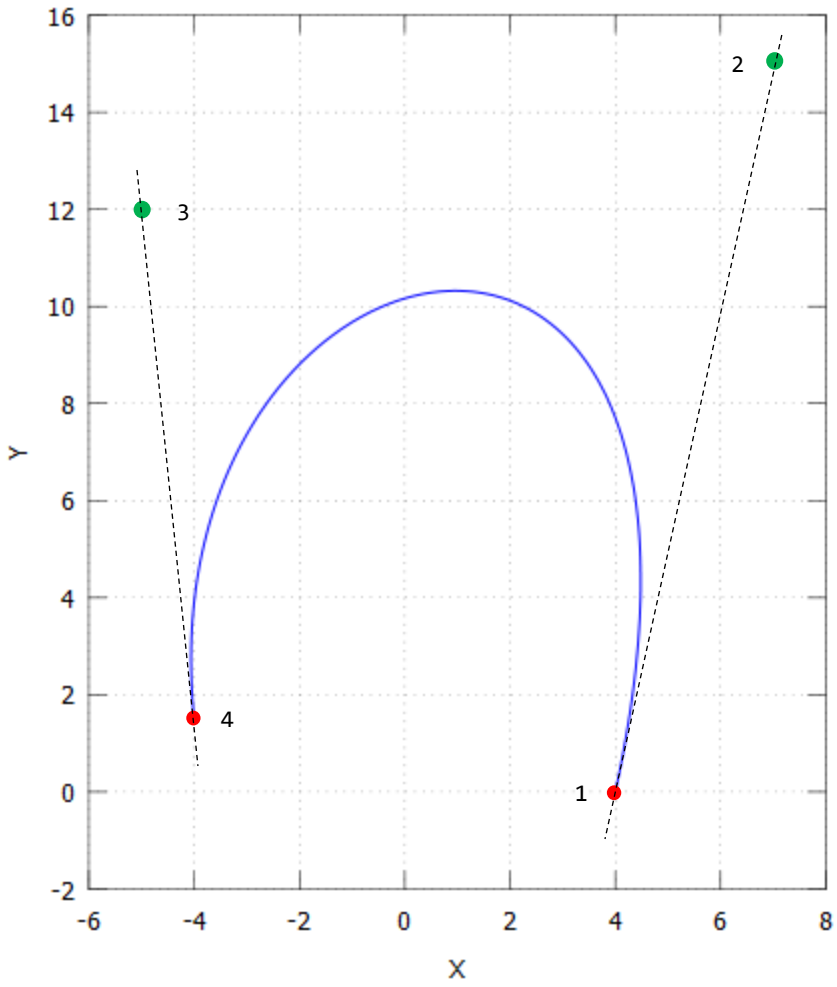

***Figure 10*:** Cubic Bézier-curve defined by the following control points:
(4.0, 0.0), **(7.0, 15.0)**, **(-5.0, 12.0)**, **(-4.0, 1.5)**

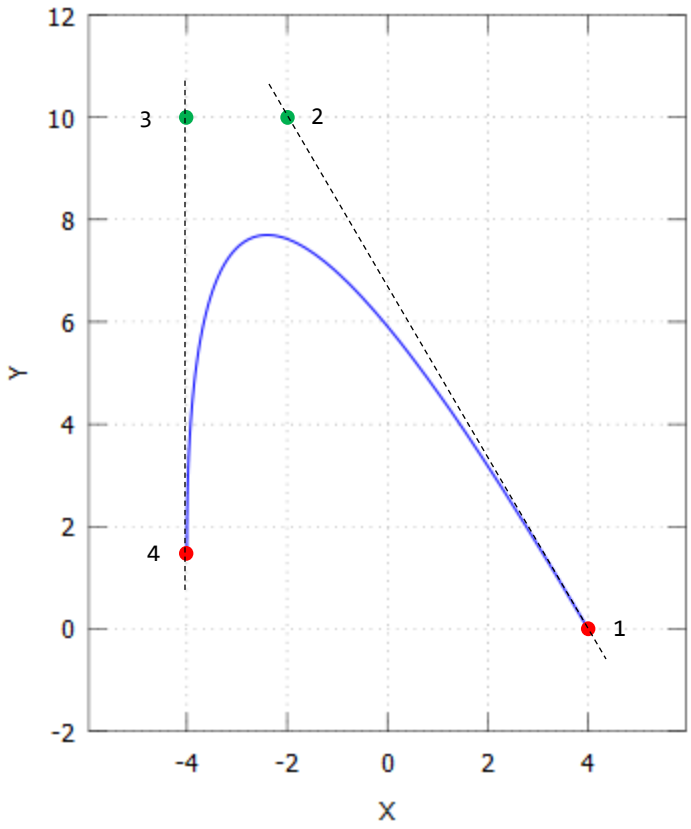

Figure 11: Cubic Bézier-curve defined by the following control points: **(4.0, 0.0)**, **(-2.0, 10.0)**, **(-4.0, 10.0)**, **(-4.0, 1.5)**

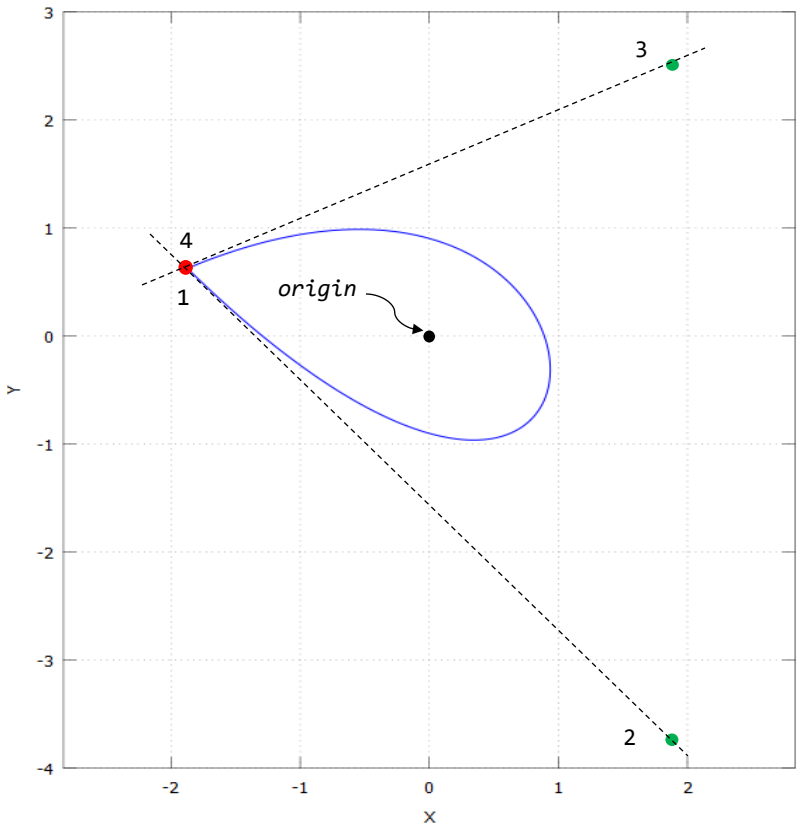

Figure 12: Tear-drop poly-Bézier shape-boundary composed of one curve defined by: `(-1.875, 0.625)`, `(1.875, -3.75)`, `(1.875, 2.5)`, `(-1.875, 0.625)`

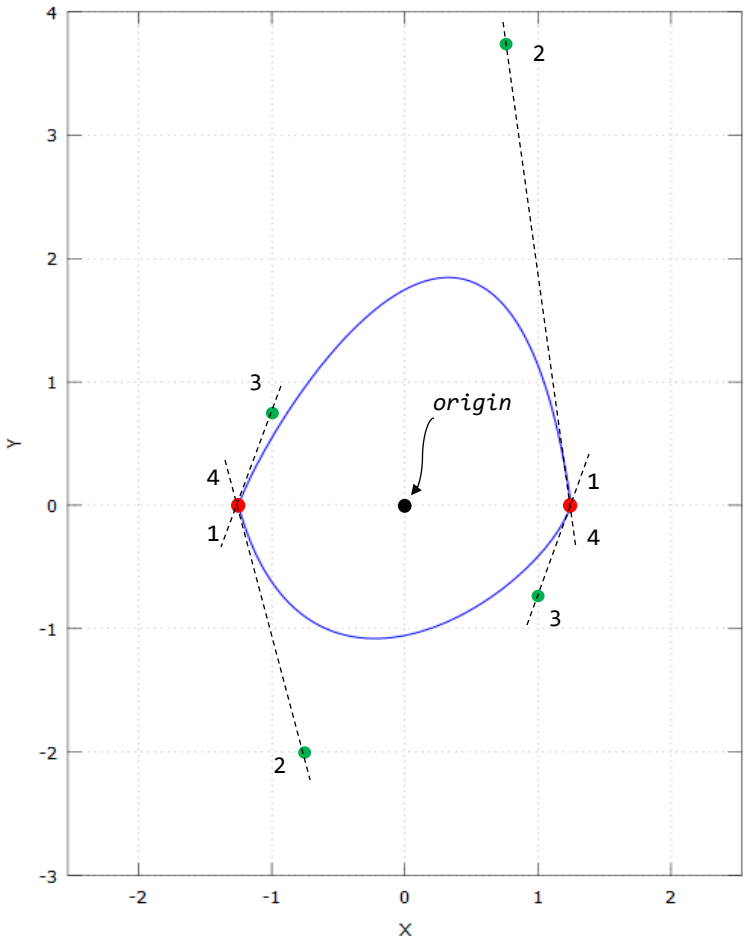

Figure 13: Non-smooth poly-Bézier shape-boundary composed of two curves defined by:
`(1.25, 0.0)`, `(0.75, 3.75)`, `(-1.0, 0.75)`, `(-1.25, 0.0)`
`(-1.25, 0.0)`, `(-0.75, -2.0)`, `(1.0, -0.75)`, `(1.25, 0.0)`

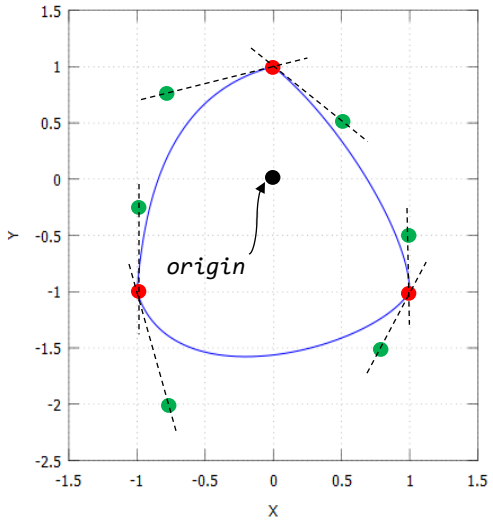

Figure 14: Non-smooth poly-Bézier shape-boundary composed of three curves defined by:
`(0.0, 1.0)`, `(-0.8, 0.75)`, `(-0.95, -0.25)`, `(-1.0, -1.0)`
`(-1.0, -1.0)`, `(-0.75, -2.0)`, `(0.75, -1.5)`, `(1.0, -1.0)`
`(1.0, -1.0)`, `(1.0, -0.5)`, `(0.5, 0.5)`, `(0.0, 1.0)`

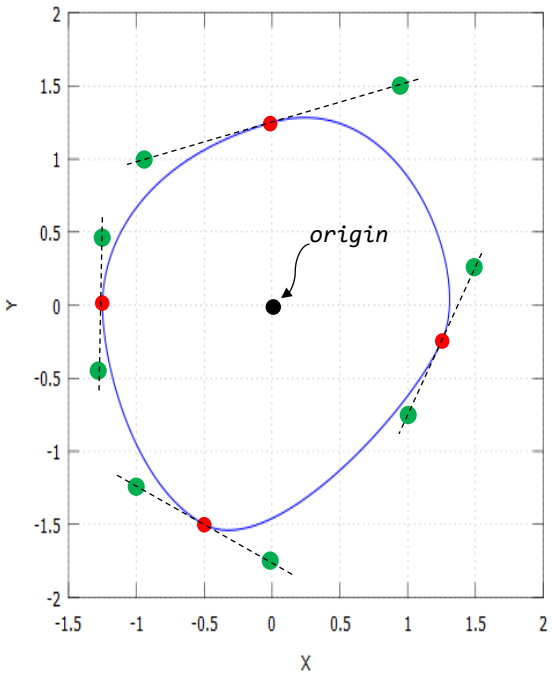

Figure 15: Smooth poly-Bézier shape-boundary composed of four curves defined by:
`(1.25, -0.25)`, `(1.5, 0.25)`, `(0.9375, 1.0)`, `(-1.25, 0.0)`
`(-1.25, 0.0)`, `(-0.9375, 1.0)`, `(-1.25, 0.4375)`, `(-1.25, 0.0)`
`(-1.25, 0.0)`, `(-1.25, -0.4375)`, `(-1.0, -1.25)`, `(-0.5, -1.5)`
`(-0.5, -1.5)`, `(0.0, -1.75)`, `(1.0, -0.7.5)`, `(1.25, -0.25)`

2.3 Shape-boundary Transformations

Transformations take an existing shape-boundary and transform it into a new shape-boundary through the use of parameterized operations. The two transformations we will focus on are the rotation and uniform-scaling transformations. These transformations will be used in the construction of sets of shapes to be used in demonstrating the packing-problem solutions developed in *Sections 4* through *7*.

2.3.1 Rotation

The *RotatedBoundary* creates a shape-boundary that is a rotated version of an existing shape-boundary. The *RotatedBoundary* has two parameters: *Shape* and *Angle*. *Shape* is an existing shape-boundary and *Angle* is the counterclockwise angle of rotation around the origin.

The following line of C++ code, as illustrated in *Figure 16*, creates a *RotatedBoundary*, `R`, containing the shape-boundary produced by rotating ellipse `E` by `alpha` radians in the counterclockwise direction:

```
RotatedBoundary R(E, alpha);
```

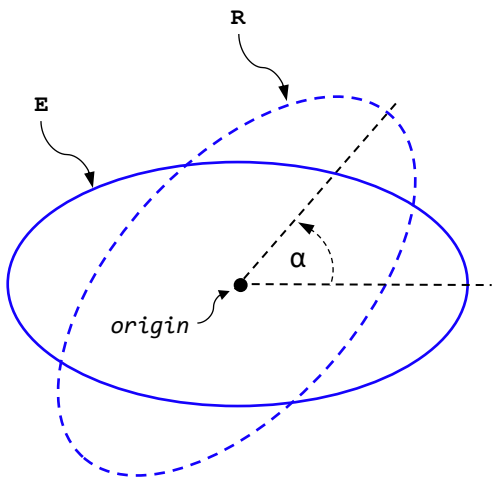

Figure 16

2.3.2 Scaling

The *ScaledBoundary* creates a shape-boundary that is a uniformly scaled version of an existing shape-boundary. The *ScaledBoundary* has two parameters: *Shape* and *Scale*. *Shape* is an existing shape-boundary, and *Scale* is a scaling factor.

The following line of C++ code, as illustrated in *Figure 17*, creates a *ScaledBoundary*, `S`, containing the shape-boundary produced by uniformly scaling ellipse `E` by a factor of 1.25:

```
ScaledBoundary S(E, 1.25);
```

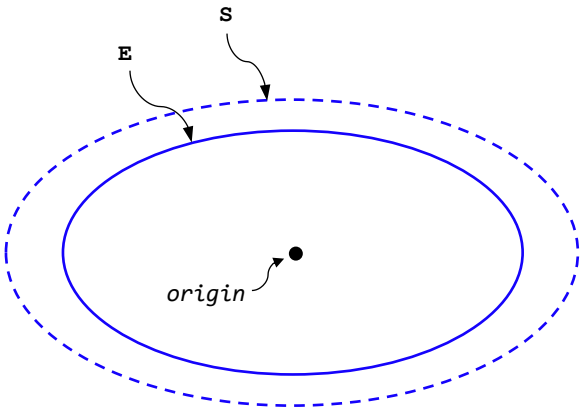

Figure 17

2.4 Closest-approach Composition

While ACGL provides variety composition operations, in this paper we will focus on the boundary-surface of closest-approach composition as it is the foundation on which nonoverlap and boundary-distance constraints will be constructed.

The *ClosestApproachBoundary* creates a shape-boundary that is the "boundary-surface of closest-approach" formed from two shape-boundaries. The *ClosestApproachBoundary* has two parameters: *Shape1* and *Shape2*. The boundary-surface of closest-approach formed from *Shape1* and *Shape2* is the surface mapped out, or inscribed by, the origin of *Shape2* as the boundary of *Shape2* is slid along the boundary of *Shape1* while maintaining the orientation of *Shape2*. The origin of the resulting boundary-surface of closest-approach corresponds to the location of the origin of *shape1* during the mapping process.

To illustrate this concept, consider the two shape-boundaries, `S` and `E`, shown in *Figure 18*. Let *Shape1* be the square, `S`, whose origin is indicated by the red dot in the center of the square. Let *Shape2* be the ellipse, `E`, whose origin is indicated by the green dot in the center of the ellipse. The resulting boundary-surface of closest-approach, shown in *Figure 19*, formed by sliding `E` around `S` is the yellow shape-boundary, `CA`, whose origin is at the center of the shape-boundary as indicated by the red dot.

The following C++ code-fragment, as illustrated in *Figure 18* and *Figure 19*, creates a *ClosestApproachBoundary*, `CA`, formed from the *SquareBoundary*, `S`, and the *EllipseBoundary*, `E`:

```
ClosestApproachBoundary CA(S, E);
```

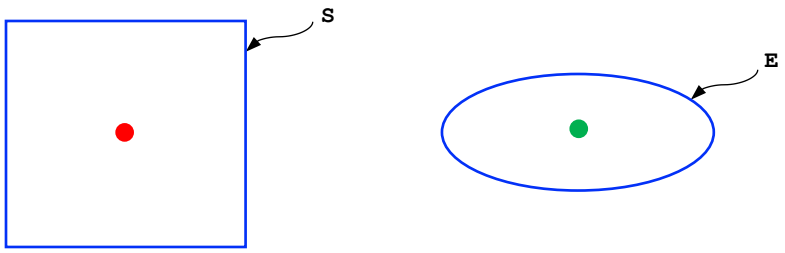

Figure 18

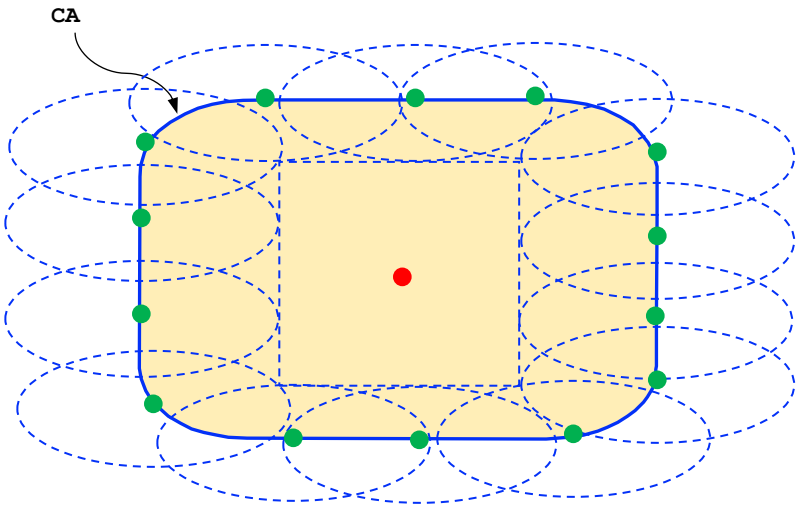

Figure 19

2.5 Shape-boundary Parameters

For the purposes of this paper a numerical shape-boundary parameter can take one of two forms: constant parameter or differentiable-variable parameter. A constant parameter is a parameter whose value is set when a shape-boundary is instantiated and which cannot be modified during the lifetime of the shape-boundary. A differentiable-variable parameter is a parameter whose value can be changed at will during the lifetime of the shape-boundary and, in addition, makes it possible to compute the derivative of the shape-boundary's tangent-distance method, *tanDist* (to be described in *Section 2.6*), with respect to that parameter.

In addition to these two parameter forms, a shape-boundary parameter can be further classified as either an explicit or implicit parameter. An explicit parameter is a parameter that is explicitly referenced and defined during the instantiation of the shape-boundary such as, for example, the diameter of a *CircleBoundary*. An implicit parameter is a parameter that is inherited from a shape-boundary that is used in the construction of another shape-boundary such as, for example, the diameter of a *CircleBoundary* that is used to construct a *ScaledBoundary*.

To select a constant-parameter form for a numerical shape-boundary parameter the parameter is passed to the shape-boundary using standard pass-by-value semantics. The

following C++ code-fragment illustrates the instantiation of two shape-boundaries, `C1` and `C2`, each having a single constant parameter:

```
CircleBoundary C1(1.5);
double d = 1.5;
CircleBoundary C2(d);
```

To select a differentiable-variable parameter-form for a numerical shape-boundary parameter, pass a variable preceded by the `(DiffVar)` prefix. This will bind the parameter to the passed variable. The following C++ code-fragment illustrates the instantiation of shape-boundary, `C1`, having one differentiable-variable parameter which is bound the variable `d`:

```
d = 1.5;
CircleBoundary C1((DiffVar)d);
```

2.6 Shape-boundary Methods

Associated with each shape-boundary is a set of methods for extracting information about the shape-boundary. In this paper we will primarily be interested in the *tanDist* (tangent distance) and *tanDistParamGrad* (tangent-distance parameter-gradient) methods as they form the foundation for constructing some or all of the containment-constraints for the packing-problem solutions developed in *Sections 4* through *7*.

2.6.1 Origin-to-tangent Distance

The *tanDist* method, as illustrated in *Figure 20*, returns the distance, *td*, along a line, oriented at an angle θ with respect to the X-axis, where the line passes through the *origin* and is perpendicular to a line, *T*, where *T* is tangent to the boundary of shape-boundary `S`.

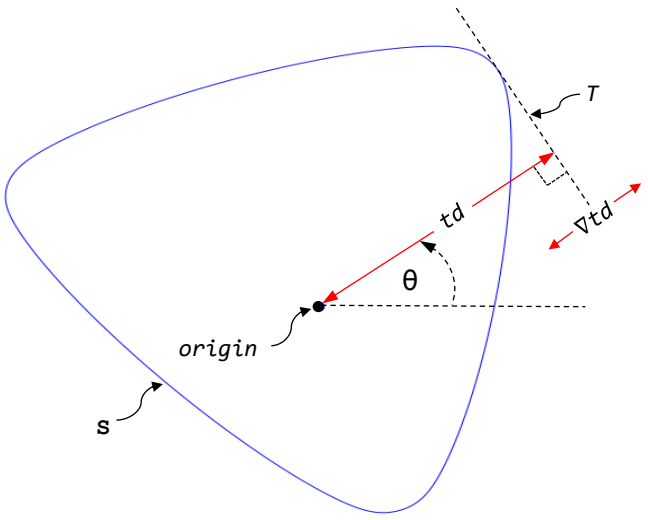

Figure 20

The following C++ code-fragment illustrates the use of the *tanDist* method on a *EllipseBoundary* to determine the origin-to-tangent distance for a tangent-distance angle θ = π/2:

```
EllipseBoundary E(1.0, 2.0);
td = E.tanDist(PI/2);
```

after executing the *tanDist* method the value of the variable `td` is set to 1.0, one half of `E`'s *diameter2* value.

2.6.2 Tangent-distance Gradient

The *tanDistParamGrad* method, as illustrated in *Figure 20*, returns the gradient, ∇td, of the tangent-distance, *td*. The *tanDistParamGrad* method returns the vector of first derivatives of *td* with respect to ALL of the shape-boundary's differentiable parameters both explicit AND implicit.

The following C++ code-fragment illustrates the use of the *tanDistParamGrad* method on a *EllipseBoundary* with two explicit differentiable-parameters:

```
EllipseBoundary E((DiffVar)xd, (DiffVar)yd);
xd = 1.0;
yd = 2.0;
GV = E.tanDistParamGrad(PI);
```

where the gradient vector, `GV`, contains the first derivative of `E`'s tangent-distance with respect to `E`'s X and Y diameters, `xd and yd`, evaluated at `xd` = 1.0, `yd` = 2.0, and θ = π.

The following C++ code-fragment illustrates the use of the *tanDistParamGrad* method on a *RotatedBoundary* with one explicit and two implicit differentiable-parameters:

```
EllipseBoundary E((DiffVar)xd, (DiffVar)yd);
RotatedBoundary R(E, (DiffVar)ang);
xd = 1.0;
yd = 2.0;
ang = PI/2;
GV = R.tanDistParamGrad(PI);
```

where the gradient vector, `GV`, contains the vector of first derivative of `R`'s tangent-distance with respect to `R`'s (explicit) rotation angle, `ang`, as well as `E`'s (implicit) X and Y diameters, `xd and yd`, evaluated at `ang` = π/2, `xd` = 1.0, `yd` = 2.0, and θ = π.

3 Shape-function

In order to construct nonoverlap and boundary-distance constraints we will need a "shape-function" whose value can tell us if a point, (x, y), is inside, outside or on the boundary of a shape-boundary when that shape-boundary has been placed such that its origin coincides with the origin of the Cartesian coordinate-system, $(0,0)$. Specifically, for a shape-boundary, B, the shape-function associated with B, $SF_B(x,y)$, is defined as a linear cone with a cross section shape defined by the geometry of B. The apex of the cone is coincident with the origin of the shape-boundary and is located

at the origin of the Cartesian coordinate-system. The value associated with the shape-function at the apex of the cone is defined to be zero while the value associated with each (x,y) point on the boundary of the shape-boundary is defined to be one.

To illustrate some of these concepts, consider the three shape-boundaries shown in *Figure 21* consisting of a circle, a cubic superellipse, and a square, where the origin of each shape-boundary is in the center of the boundary. The shape-functions corresponding to each of these shape-boundaries are the cones shown in *Figure* 22.

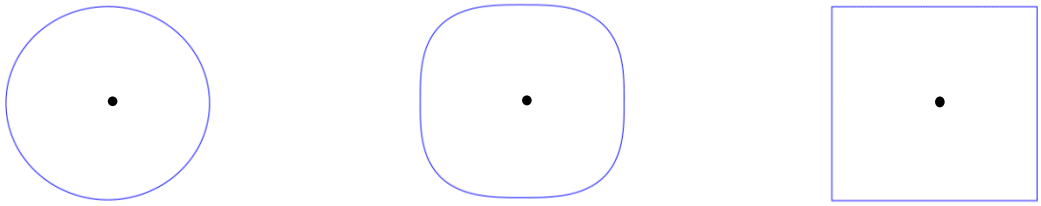

Figure 21

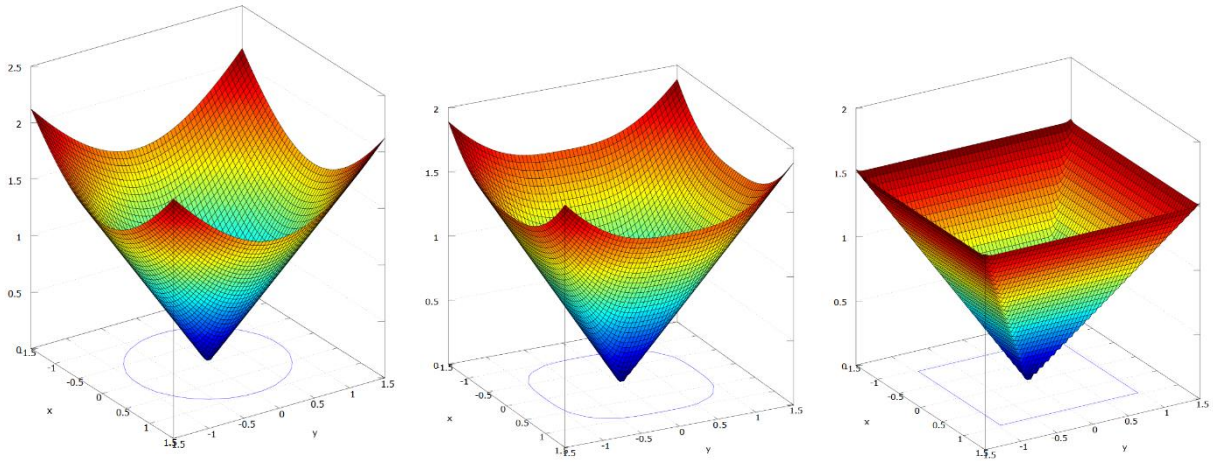

Figure 22

3.1 Overlap Detection

Based on the definition of a shape-function, we can now determine if a point is inside, outside, or on the boundary of a shape-boundary when that shape-boundary has its origin at the origin of the Cartesian coordinate-system. Specifically, a point (x, y) is outside the boundary of a shape-boundary, B, if $SF_B(x, y) > 1$. Similarly, if $SF_B(x, y) = 1$ then the point (x,y) is on the boundary of the shape-boundary. Finally, if $SF_B(x, y) < 1$ then the point (x,y) is inside the boundary of the shape-boundary.

Using this knowledge and the closest-approach composition for shape-boundaries, we can easily determine if two shape-boundaries overlap or abut once we are given the locations of their origins. Specifically, consider the two shape-boundaries, *shape1* and *shape2*, shown in *Figure 18* and the shape-boundary for their closest-approach composition, CA, shown in yellow in *Figure 19*. Assume that the location of the origin of *shape1* is represented as (x_1, y_1) and the location of the origin of *shape2* is represented as (x_2, y_2). If we further assume that $(x_1, y_1) = (0, 0)$, then we can check to see if *shape2* abuts *shape1* by forming the shape-function for CA, $SF_{CA}(x, y)$, and checking to see if $SF_{CA}(x_2, y_2) = 1$.

Similarly, we can check to see if *shape2* overlaps *shape1* by checking to see if $SF_{CA}(x_2, y_2) < 1$ and finally we can check to see if *shape2* does not abut or overlap *shape1* by checking to see if $SF_{CA}(x_2, y_2) > 1$. If we now remove the restriction that $(x_1, y_1) = (0,0)$, we can check to see if *shape2* abuts *shape1* by checking to see if $SF_{CA}(x_1, y_1, x_2, y_2) = 1$, where $SF_{CA}(x_1, y_1, x_2, y_2)$ is defined to be $SF_{CA}(x_2 - x_1, y_2 - y_1)$. Similarly, we can check for overlap and nonoverlap between *shape2* and *shape1* by checking to see if $SF_{CA}(x_1, y_1, x_2, y_2) < 1$ and $SF_{CA}(x_1, y_1, x_2, y_2) > 1$, respectively. Note that this approach to overlap detection is similar in spirit to the use of phi-functions for overlap detection [10] [1].

3.2 Bound Shape-function

The *BoundShapeFunction* creates a bound-shape-function. A bound-shape-function is a shape-function whose arguments are permanently bound to constants or variables transforming those arguments into numerical parameters of the bound-shape-function.

As was the case for numerical shape-boundary parameters described in *Section 2.5*, a numerical bound-shape-function parameter can take one of several forms including constant and differentiable-variable. Recall that a differentiable-variable parameter is a parameter whose value can be changed at will during the lifetime of the bound-shape-function and, in addition, makes it possible to compute the derivative of the bound-shape-function with respect to that parameter.

In addition to these parameter forms, a bound-shape-function parameter can be further classified as either an explicit or implicit parameter. An explicit parameter is a parameter that is explicitly referenced and defined during the instantiation of the bound-shape-function. An implicit parameter is a parameter that is inherited from the shape-function from which the bound-shape-function is created.

A bound-shape-function can be instantiated with five parameters: a shape-boundary and two coordinate pairs, (x_1, y_1) and (x_2, y_2), of numerical parameters. The two pairs of numerical parameters are internally combined as $(x_2 - x_1, y_2 - y_1)$ to form the coordinate-pair arguments for the shape-function.

To illustrate some of these concepts, the following C++ code-fragment instantiates a unit-circle shape-boundary, `C`, and then instantiates a bound-shape-function, `BC`, from `C`. This instantiation binds the variables `x1`, `y1`, `x2`, and `y2` as differentiable-variable parameters to be used in the calculation of the shape-function's (x,y) arguments:

```
CircleBoundary C(1.0);
BoundShapeFunction BC(C, (DiffVar)x1, (DiffVar)y1, (DiffVar)x2, (DiffVar)y2);
```

3.3 Bound Shape-function Methods

Associated with each bound shape-function is a set of methods for extracting information about the bound shape-function. In this paper we will primarily be interested in the *evaluateFunct* (evaluate function) and *evaluateGrad* (evaluate gradient) methods as they form the foundation for constructing the non-overlap and boundary-distance constraints for the packing-problem solutions developed in *Sections 4* through *7*.

3.3.1 Function Evaluation

The *evaluateFunct* method returns the value of the shape-function based on the current value of all of the shape-function's parameters and bound arguments.

The following C++ code-fragment illustrates the use of the *evaluateFunct* method on a bound-shape-function derived from an ellipse shape-boundary:

```
EllipseBoundary E((DiffVar)xd, (DiffVar)yd);
BoundShapeFunction BESF(E,(DiffVar)x1,(DiffVar)y1,(DiffVar)x2,(DiffVar)y2);
xd = 1.0;
yd = 2.0;
x1 = 3.0;
y1 = 4.0;
x2 = 5.0;
y2 = 6.0;
f = BESF.evaluateFunct();
```

In this example, `f` is assigned the value of the bound ellipse shape-function, `BESF`, evaluated at `xd` = 1.0, `yd` = 2.0, `x1` = 3.0, `y1` = 4.0, `x2` = 5.0, and `y2` = 6.0.

3.3.2 Gradient Evaluation

The *evaluateGrad* method returns the gradient of the shape-function based on the current value of all of the shape-function's parameters and bound arguments. The components of the gradient consist of the shape-function's first derivatives with respect to each of the differentiable parameters of the shape-boundary and each of the differentiable bound-arguments of the shape-function.

The following C++ code-fragment illustrates the use of the *evaluateGrad* method on a bound-shape-function derived from an ellipse shape-boundary:

```
EllipseBoundary E((DiffVar)xd, (DiffVar)yd);
BoundShapeFunction BESF(E, (DiffVar)x1,(DiffVar)y1,(DiffVar)x2,(DiffVar)y2);
xd = 1.0;
yd = 2.0;
x1 = 3.0;
y1 = 4.0;
x2 = 5.0;
y2 = 6.0;
GV = BESF.evaluateGrad();
```

In this example, the gradient vector, `GV`, is assigned the value of the vector of first derivatives of the bound ellipse shape-function, `BESF`, with respect to its four explicit differentiable parameters (`x1`, `y1`, `x2`, and `y2`) and its two implicit differentiable parameters (`xd` and `yd`) evaluated at `xd` = 1.0, `yd` = 2.0, `x1` = 3.0, `y1` = 4.0, `x2` = 5.0, and `y2` = 6.0.

4 Packing with Nonoverlap Constraints

In this section we will develop a solution for packing a set of variable-orientation poly-Bézier shapes into a minimum-area packing-boundary where the packing-boundary is an axis-aligned rectangle. That is, a rectangle where one pair of the rectangle's sides is parallel to the X-axis and the other pair of sides is parallel to the Y-axis.

A formal description of this packing-problem can be stated as follows:

> Given a set of variable-orientation ply-Bézier shapes, S, containing n shapes, determine the location of the origin, (x_i, y_i), and the rotation angle, α_i, of each shape, s_i, in S that minimize the area of an axis-aligned perimeter rectangle, $AAPR$, enclosing the shapes such that no two shapes overlap.

From this, we can formulate this packing problem as the following high-level optimization:

> Minimize the area of $AAPR$
> while satisfying:
> 1. Nonoverlap constraints between all pairs of shapes in S.
> 2. Containment constraints between each shape in S and $AAPR$.

4.1 Objective Function and Containment Constraints

In order to create a detailed representation of the objective function and containment constraints, we will need a representation of the $AAPR$. We will represent it using four optimization variables as illustrated in *Figure 23*. From this figure, we can see that UH represents the position of the upper horizontal side of the rectangle, LH represents the position of the lower horizontal side, UV represents the position of the upper vertical side, and LV represents the position of the lower vertical side.

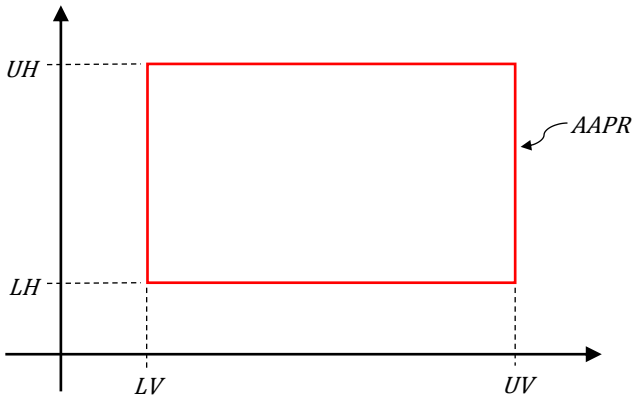

Figure 23

Using this representation of the $AAPR$, we can now formulate the optimization-problem's objective function as:

$$(UH - LH)\,(UV - LV) \qquad (1)$$

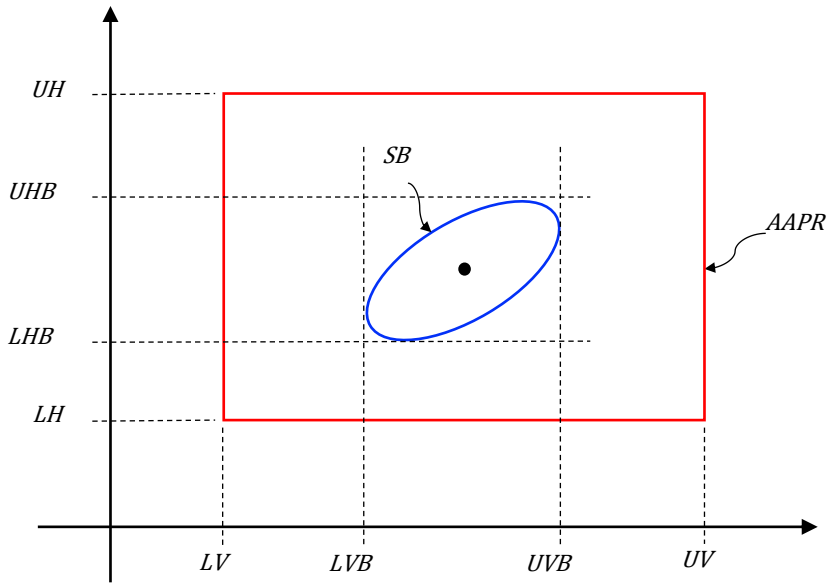

Figure 24

To formulate the containment-constraints, we begin by observing that, as shown in *Figure 24*, in order for a shape-boundary, *SB* (in this example, a rotated ellipse), to be contained inside the *AAPR*, the position of the shape's upper horizontal bound, *UHB*, must be below *UH*; the shape's lower horizontal bound, *LHB*, must be above *LH*; the shape's upper vertical bound, *UVB*, must be below *UV*; and the shape's lower vertical bound, *LVB*, must be above *LV*. These requirements can be concisely stated as:

$$UVB \leq UV \tag{2}$$

$$UHB \leq UH \tag{3}$$

$$LV \leq LVB \tag{4}$$

$$LH \leq LHB \tag{5}$$

As shown in *Figure 25*, we can compute *UHB*, *LHB*, *UVB*, and *LVB*, using the location of the origin of *SB*, (x,y), and the tangent-distance values associated with *SB* at the angles 0, π/2, π, and 3π/2. Specifically, we can compute *UVB*, *UHB*, *LVB*, and *LHB* as:

$$UVB = x + SB.tanDist(0) \tag{6}$$

$$UHB = y + SB.tanDist(\pi/2) \tag{7}$$

$$LVB = x - SB.tanDist(\pi) \tag{8}$$

$$LHB = y - SB.tanDist(3\pi/2) \tag{9}$$

which, in turn, allows us to formulate the following containment constraint inequalities:

$$x + SB.tanDist(0) \leq UV \tag{10}$$

$$y + SB.tanDist(\pi/2) \leq UH \tag{11}$$

$$LV \leq x - SB.tanDist(\pi) \tag{12}$$

$$LH \leq y - SB.tanDist(3\pi/2) \tag{13}$$

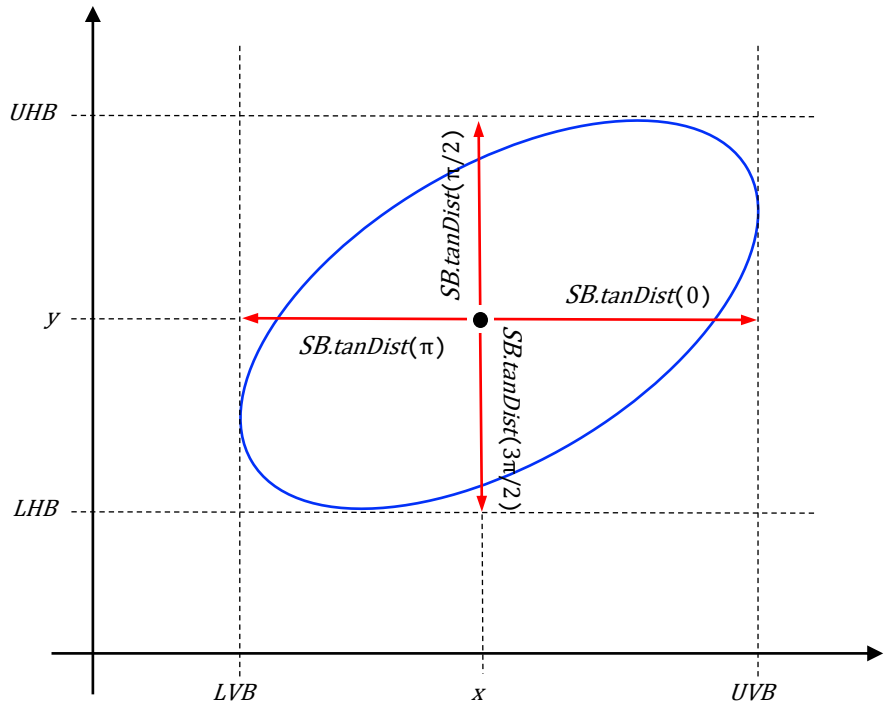

Figure 25

4.2 Nonoverlap Constraints

Conceptually, nonoverlap constraints must serve two purposes. First, they are used by the optimizer to determine if there is an overlap between a given pair of shapes. Second, if there is an overlap, the nonoverlap constraint is used to guide the optimizer in the process of eliminating that overlap.

To construct a nonoverlap constraint that can serve both purposes, we will formulate each nonoverlap constraint using a bound-shape-function as the constraint function. As outlined in *Section 3.1*, two shapes do NOT overlap when the bound-shape-function of their closest-approach shape-boundary is greater than one.

Specifically, given two shape-boundaries, *SB1* and *SB2*, and the position of their origins, $(x1, y1)$ and $(x2, y2)$, we can form the bound-shape-function, *BSF*, of their closest-approach shape-boundary as:

$$BSF = BoundShapeFunction(ClosestApproachBoundary(S1, S2), (DiffVar)x1, (DiffVar)y1, (DiffVar)x2, (DiffVar)y2) \quad (14)$$

allowing us to formulate the nonoverlap-constratint between *SB1* and *SB2* as:

$$BSF.evaluteFunct() \geq 1 \quad (15)$$

where $x1$, $y1$, $x2$, and $y2$ are differentiable variables of the bound-shape-function.

This formulation gives the optimizer the ability to detect overlaps by checking to see if the value of *BSF.evaluateFunct*(), is less than one. To determine how to resolve an overlap, the optimizer uses the information contained in *BSF*'s gradient vector,

BSF.evaluateGrad( ). For the purposes of guiding the optimizer in the process of eliminating an overlap, the information contained in *BSF*'s gradient vector can best be viewed as a vector field representing a spreading force which is being applied between the origins of the overlapping shapes.

To illustrate the concept of interpreting the gradient of *BSF* as a spreading force, consider the two shapes shown in *Figure 26*. Assume, for the purposes of this illustration, that the first shape, *SB1*, is an immovable unit-square whose origin is fixed at $(0,0)$ and the second shape, *SB2*, is a movable unit-circle whose origin is initially located at $(0.2,0)$.

We can represent the bound-shape-function associated with the nonoverlap-constraint function, *BSF*, as the shape-function associated with the closest-approach shape-boundary formed from the square and the circle. Along with the green square, red circle, and their closest-approach shape-boundary (shown in blue) *Figure 27* illustrates the linear cone of this bound-shape-function and its gradient field (projected onto the XY plane) as a function of *x2* and *y2* (the location of the origin of the unit circle).

Looking at *Figure 28*, which is the XY plane of *Figure 27*, we can see that the vector field is pointing to the right at the location of the origin of the circle. This indicates that the optimizer will try and "push" the circle to the right in an effort to resolve the overlap between the circle and the square.

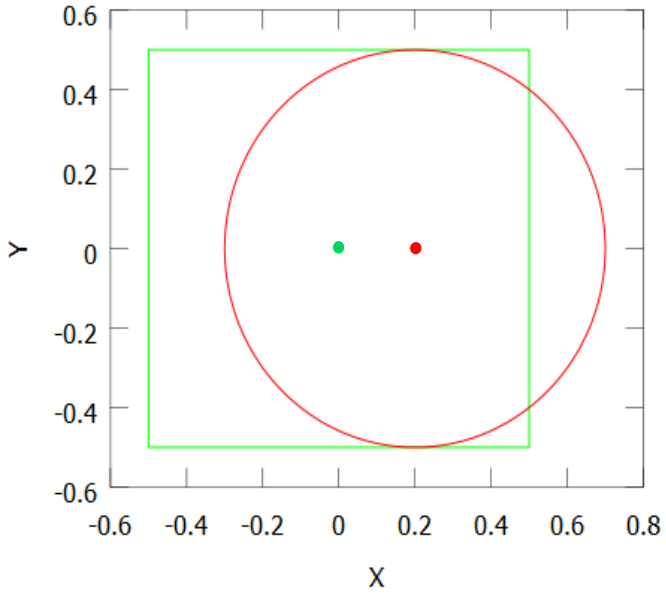

Figure 26

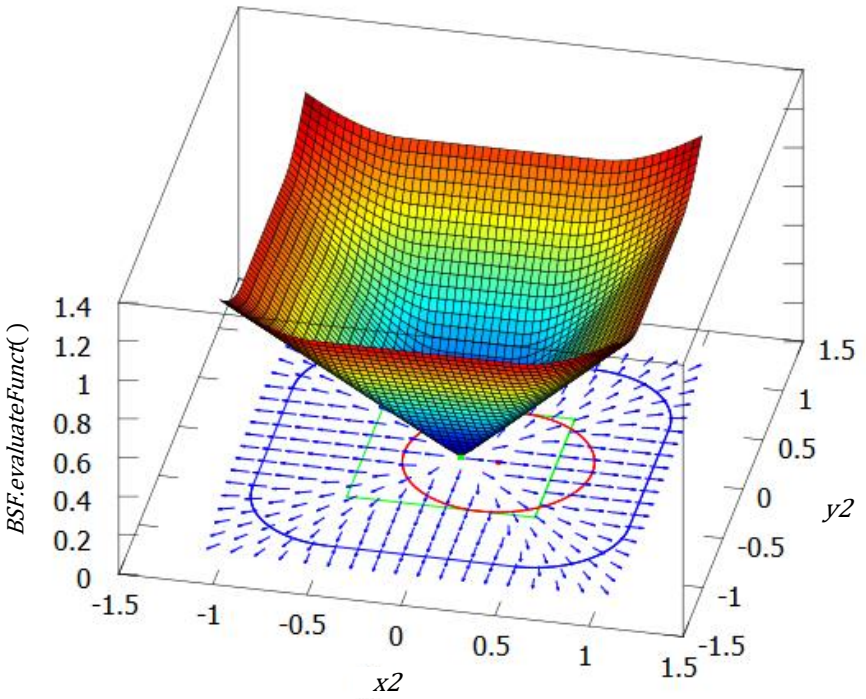

Figure 27

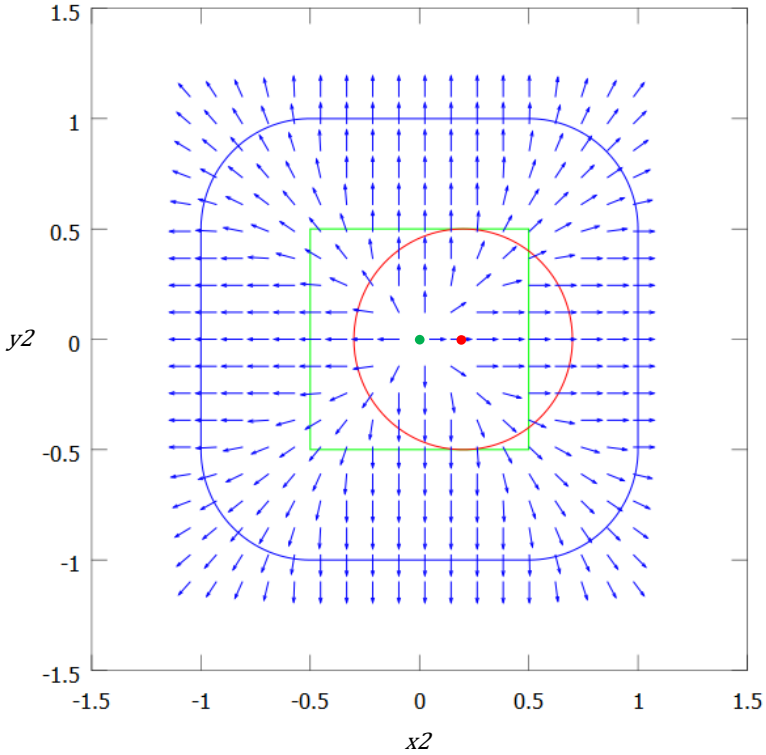

Figure 28

4.3 Implementation Details

We implemented the solution to this packing problem in C++ using IPOPT [23] (version 3.11.0) and its C interface.

For the purposes of this paper, we created the set of variable-orientation poly-Bézier shapes, *S*, using scaled versions of the four poly-Bézier shapes shown in *Figure 29* and developed in *Section 2.2*.

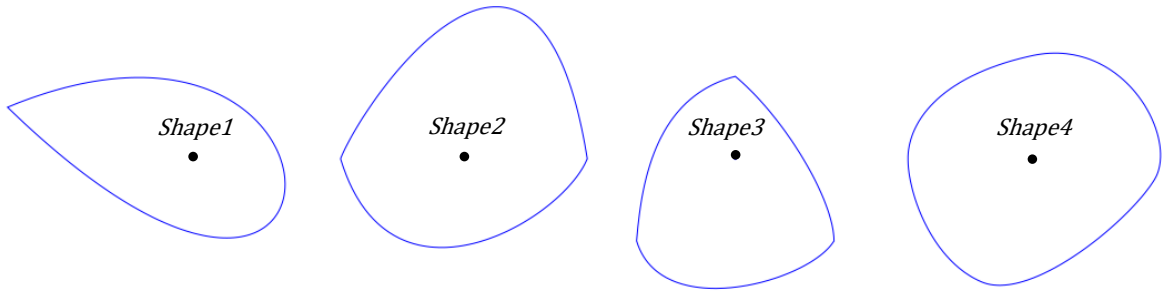

Figure 29

We created variable-orientation, scaled versions of these shapes using *RotatedBoundary*, described in *Section 2.3.1*, and *ScaledBoundary*, described in *Section 2.3.2*. For example, the following C++ code-fragment illustrates how to create a shape-boundary, *SB*, which is a variable-orientation, scaled, version of the *Shape1* shape-boundary with a constant scale-factor of 0.8 and `rotAng` as its rotation-angle optimization-variable:

```
SB = RotatedBoundary(ScaledBoundary(Shape1, 0.8),(DiffVar) rotAng);
```

Additionally, since the solution was implemented using IPOPT, we converted all the inequality constraints to the following IPOPT inequity-constraint form:

$$C_L \le C(x_1, x_2, \cdots x_n) \le C_U \tag{16}$$

where $C(x_1, x_2, \cdots x_n)$ is a constraint function with n optimization variables and C_L and C_U are the lower and upper bounds on $C(x_1, x_2, \cdots x_n)$. In particular, the four containment-constraints (Eq. (10) through (13)) for each shape were converted to the following set of four IPOPT style inequality-constraints:

$$0 \le UV - x - SB.tanDist(0) \le \infty \tag{17}$$

$$0 \le UH - y - SB.tanDist(\pi/2) \le \infty \tag{18}$$

$$0 \le x - SB.tanDist(\pi) - LV \le \infty \tag{19}$$

$$0 \le y - SB.tanDist(3\pi/2) - LH \le \infty \tag{20}$$

Similarly, each nonoverlap-constraint (Eq. (15)) was converted to the following IPOPT style inequality-constraint:

$$1 \le BSF.evaluteFunct() \le \infty \tag{21}$$

Constructing IPOPT's Jacobian matrix elements for the four containment-constraints associated with each shape-boundary was fairly easy as the containment constraints are linear in *UV*, *UH*, *LV*, *LH*, and the XY-position of each shape-boundary. Additionally, the derivative of the tangent-distance (for tangent-distance angles 0, $\pi/2$, π, and

$3\pi/2$) with respect each shape-boundary's rotation angle was determined using each shape-boundary's *tanDistParamGrad* method.

The six Jacobian elements for each nonoverlap-constraint (two for the position and one for the rotation angle of each shape-boundary) were determined using the values obtained from the *evaluateGrad* method for each nonoverlap-constraint's bound shape-function.

Additionally, we used IPOPT's Limited-memory Hessian-approximation feature to generate the Hessian matrix. We activated and configured this Hessian-approximation feature by setting the `hessian_approximation` option to `"limited-memory"` and the `limited_memory_max_history` option to 50.

Finally, we applied a 20x scaling factor to the objective by setting IPOPT's `obj_scaling_factor` option to 20.

4.4 Packing Results

Figure 30 shows the execution times for this implementation when packing sets of shapes ranging in size from 5 to 80 shapes. Execution times were generated from a 64-bit executable on a laptop with an Intel Core i7 CPU running at 2.8 GHz with 8 GB of memory running under Microsoft Windows.

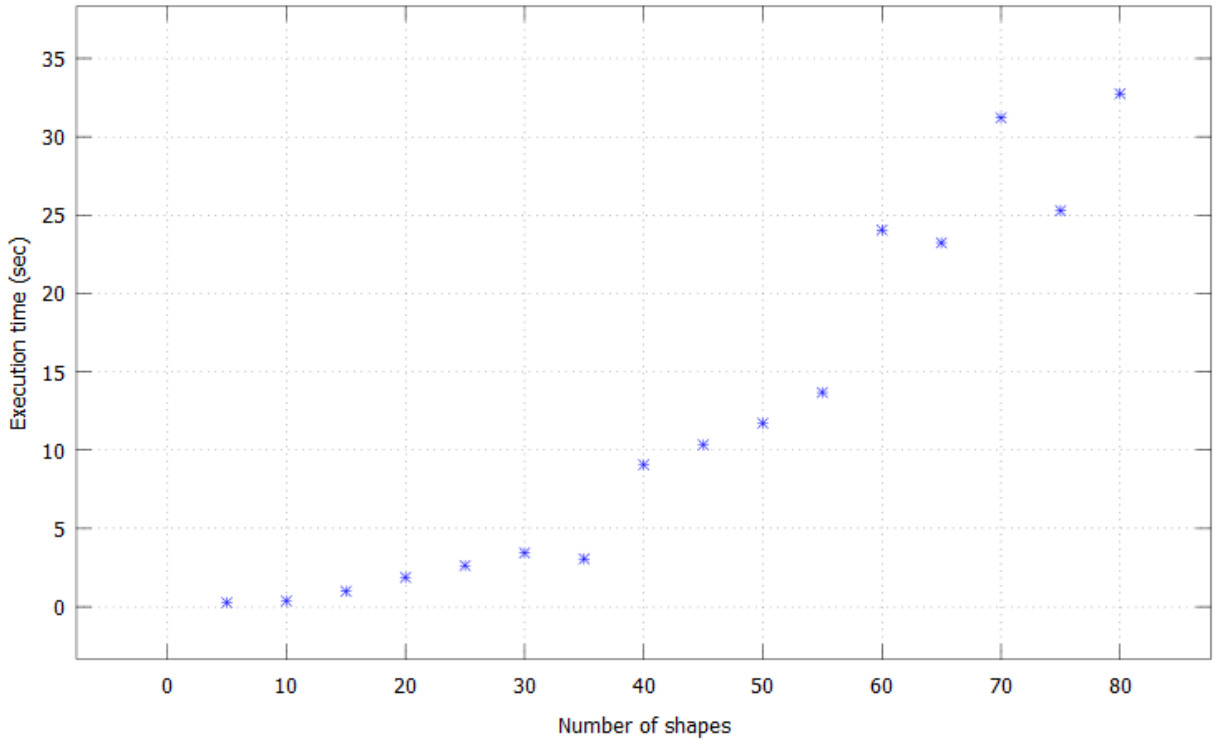

Figure 30: Execution times for minimum-area rectangular packing using nonoverlap constraints.

Figure 32 shows the packing results obtained when we pack a set of 70 variable-orientation poly-Bézier shapes, whose initial positions and orientations are shown in *Figure 31*, into a minimum-area axis-aligned perimeter-rectangle.

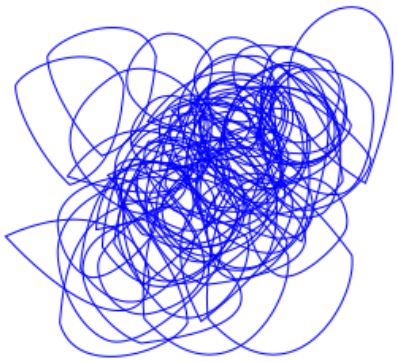

Figure 31: Initial position and orientation of the 70 poly-Bézier shapes used in the packing of *Figure 32*.

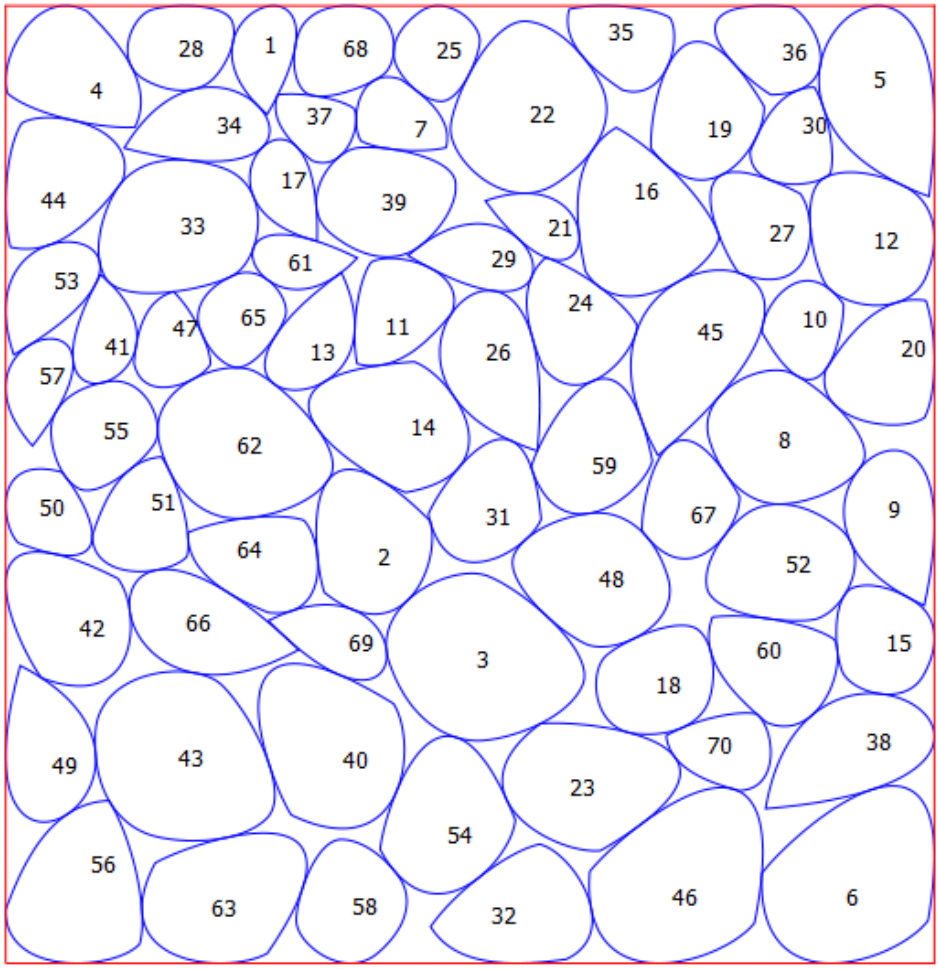

Figure 32: Minimum-area rectangular packing of 70 poly-Bézier shapes using nonoverlap constraints.

5 Packing with Boundary-distance Constraints

In this section we will develop a solution for packing a set of variable-orientation poly-Bézier shapes into a minimum-area packing-boundary where the packing-boundary is an axis-aligned rectangle while maintaining a minimum distance, d, between the shapes

A formal description of this packing-problem can be stated as follows:

> Given a set of variable-orientation poly-Bézier shapes, S, containing n shapes, determine the location of the origin, (x_i, y_i), and the rotation angle, α_i, of each shape, s_i, in S that minimizes the area of an axis-aligned perimeter rectangle, *AAPR*, enclosing the shapes such that the boundary distance between all pairs of shapes is no less than d.

From this, we can formulate this packing problem as the following high-level optimization:

> Minimize the area of *AAPR*
> while satisfying:
> 1. Boundary-distance constraints between all pairs of shapes in S.
> 2. Containment constraints between each shape in S and the *AAPR*.

which is just a simple modification of the high-level optimization-problem for packing with nonoverlap constraints outlined in *Section 4*. Specifically, we replaced the nonoverlap constraints between all pairs of shapes in S with boundary-distance constraints between all pairs of shapes in S.

5.1 Boundary-distance Constraints

To formulate a boundary-distance constraint between two shape-boundaries, *S1* and *S2*, we note that a boundary-distance constraint is equivalent to expanding the boundary of *S1*, as illustrated in *Figure 33*, by a distance d along a line perpendicular to the boundary of the shape at every point on its boundary and then enforcing a nonoverlap constraint between this expanded boundary, *ES*, and *S2*, as shown in *Figure 34*. This guarantees that *S1* can be no closer than a distance d to *S2*. Further, as shown in *Figure 35*, the expanded shape, *ES*, is a *ClosestApproachBoundary* formed from *S1* and a *CircleBoundary* of diameter $2d$:

$$ES = ClosestApporachBoundary(S1, CircleBoundary(2d)) \tag{22}$$

As such, this means we can formulate the bound-shape-function for the boundary-distance constraint between *S1* and *S2* as:

$$BSF = BoundShapeFunction(ClosestApproachBoundary(ES, S2), (DiffVar)x1, (DiffVar)y1, (DiffVar)x2, (DiffVar)y2) \tag{23}$$

where the origin of *S1* is located at $(x1, y1)$ and the origin *S2* is located at $(x2, y2)$.

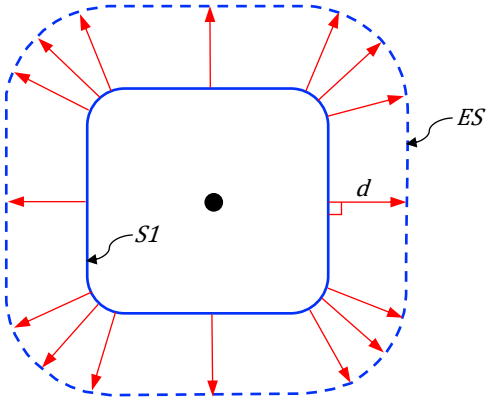

Figure 33

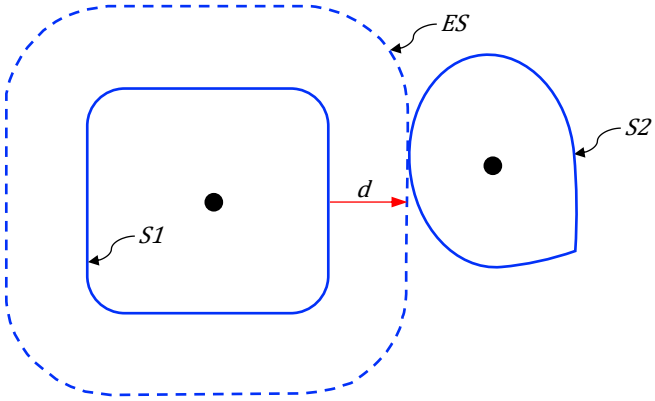

Figure 34

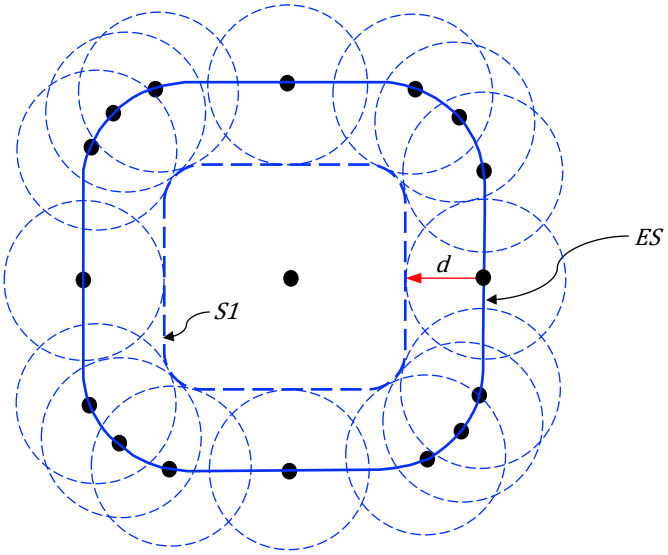

Figure 35

From this formulation it can be seen that this boundary-distance constraint is the simplest form of a more general boundary-distance constraint. That is, the distance between *S1* and *S2* is the same in all directions forming a "uniform" boundary-distance constraint. By choosing a shape-boundary other than a circle we can form a direction-dependent, or "nonuniform", boundary-distance constraint. As an example of this concept, consider the case shown in *Figure 36*. In this case we can form a nonuniform boundary-distance constraint which has a "horizontal spacing" *hd*, a "vertical spacing" *vd*, and an elliptical transition between its horizontal and vertical spacing.

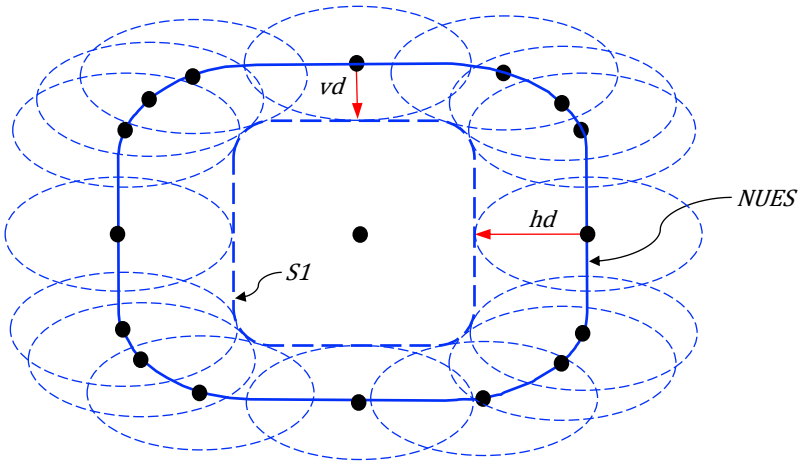

Figure 36

From this we can form the corresponding nonuniform boundary-distance constraint's bound-shape-function as follows:

$$NUES = ClosestApporachBoundary(S1, EllipseBoundary(2hd, 2vd)) \tag{24}$$

$$BSF = BoundShapeFunction(ClosestApproachBoundary(NUES, S2), (DiffVar)x1, (DiffVar)y1, (DiffVar)x2, (DiffVar)y2) \tag{25}$$

where, again, the origin of *S1* is located at (*x1*, *y1*) and the origin *S2* is located at (*x2*, *y2*).

5.2 Packing Results

Figure 37 shows the packing results obtained when we pack a set of 70 variable-orientation poly-Bézier shapes, whose initial positions and orientations are shown in *Figure 31*, into a minimum-area axis-aligned perimeter-rectangle with a uniform boundary-distance spacing, *d*, shown between shapes 4 and 33 in the upper-left corner of the figure.

Figure 38 shows the packing results obtained when we pack the same set of 70 variable-orientation poly-Bézier shapes into a minimum-area axis-aligned perimeter-rectangle with a nonuniform elliptic boundary-distance spacing where $hd = 2d$ and $vd = d/2$.

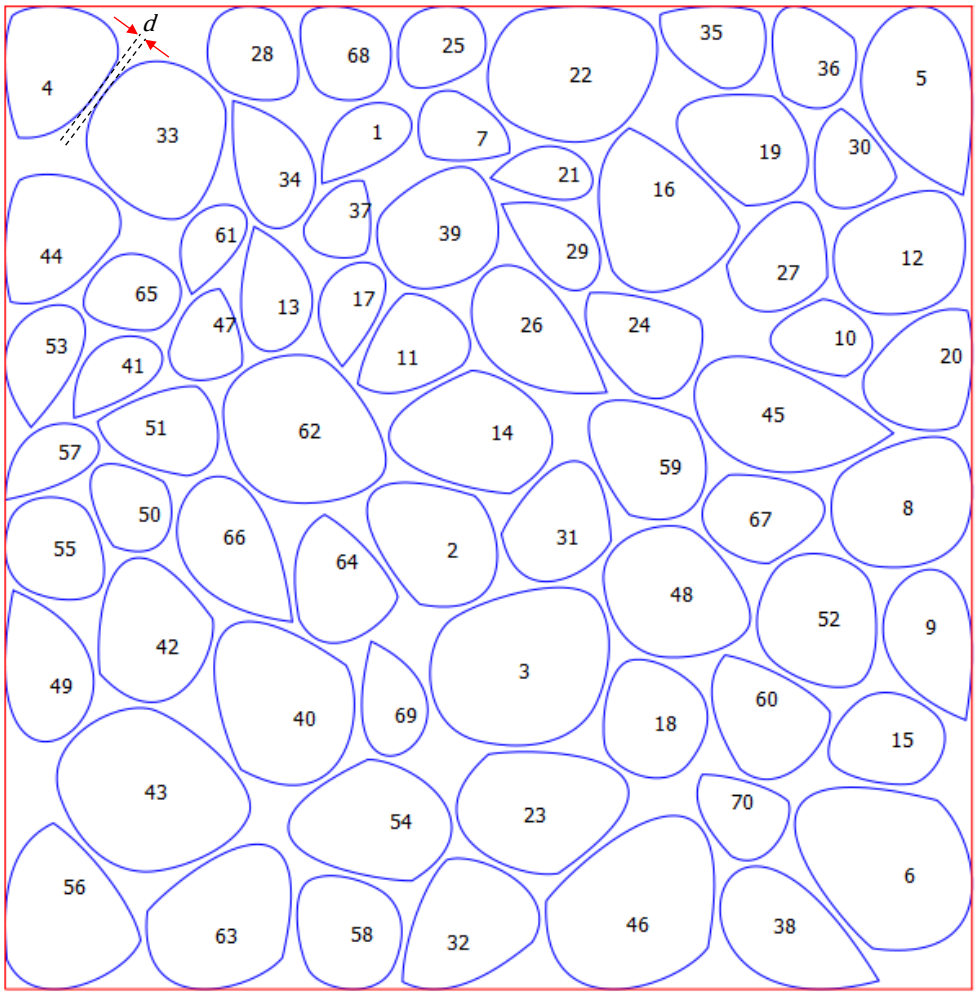

Figure 37: Minimum-area rectangular packing of 70 poly-Bézier shapes using uniform boundary-distance constraints.

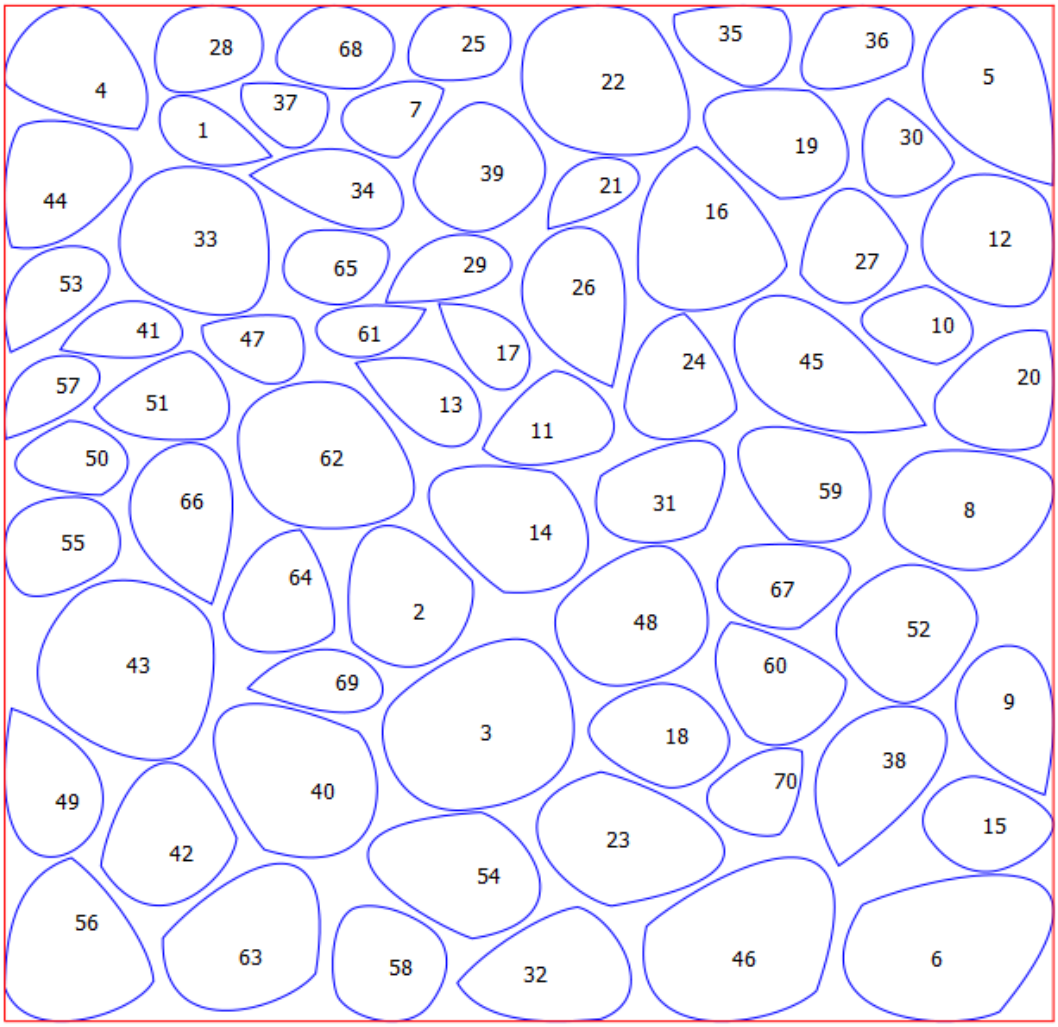

Figure 38: Minimum-area rectangular packing of 70 poly-Bézier shapes using nonuniform boundary-distance constraints.

6 One-dimensional Puzzle-piece Packing

In this section we will develop a solution for packing a set of variable-orientation poly-Bézier shapes into a minimum area where the packing area has a fixed height and where the left boundary of the packing can be wrapped around to fit snuggly into, or interleave with, the right boundary.

We refer to the resulting packing as a one-dimensional "puzzle-piece" packing since a sequence of these packings can be placed end to end where the left side of one element in the sequence interlocks tightly into the right side of the next element. These interlocking boundaries eliminate the "hard" straight-line vertical packing-boundaries giving the optimizer more freedom in how it uses the space within the packing boundary. In particular, it eliminates the wasted space at the boundary between adjacent elements that would have been present had the elements been formed by packing the shapes into a fixed-height rectangle.

Additionally, a solution to the one-dimensional puzzle-piece packing-problem is also a solution to the cylindrical packing-problem. That is, it is a solution to the problem of packing a set of shapes onto the lateral surface of a fixed-height minimum-circumference cylinder, or a "cylindrical packing". Further, using this equivalence between puzzle-piece and cylindrical packing we can define the area of a one-dimensional puzzle-piece packing as the lateral surface-area of the minimum-circumference fixed-height cylinder around which the puzzle-piece packing can be wrapped.

This type of packing is of particular value in a just-in-time continuous-manufacturing environment where the same set of parts needs to be "punched" or cut from a fixed-width rolled sheet-stock at regular time-intervals using a rotary die-cutter [24] [25]. In particular, the pattern that needs to be engraved on the rotary die-cutter is a boundary-distance constrained fixed-height cylindrical-packing of the set of parts to be punched from the sheet stock.

A formal description of the one-dimensional puzzle-piece packing-problem can be stated as follows:

Given:

1. a set of variable-orientation poly-Bézier shapes, $\mathcal{S}$, containing n shapes;
2. a fixed-height cylinder of circumference c;

determine the location of the origin, (x_i, y_i), and the rotation angle, α_i, of each shape, s_i, in $\mathcal{S}$ that minimizes the circumference of the cylinder such that:

1. the packing is connected;
2. no shape s_i in $\mathcal{S}$ overlaps its self or any other shape in $\mathcal{S}$ when the packing is wrapped around the lateral surface of the cylinder.

Solving this packing problem as a single optimization would require the use of a modulo or "saw tooth" function, such as that shown in *Figure 39*, to map all shapes onto a single revolution of the cylinder. As the saw-tooth function is discontinuous it does not work well, if at all, with commercial nonlinear optimization-software. Without it however, depending on the optimizer and each shape's initial position, the optimization process can easily spread the shapes across an unknown number of revolutions of the cylinder. This, in turn, would require that we check each of these revolutions against

every other revolution to ensure that no shape in one revolution overlaps any shape in another. This would produce an unmanageable optimization-problem. To avoid these difficulties, we instead develop and implement a two-stage successive-refinement strategy.

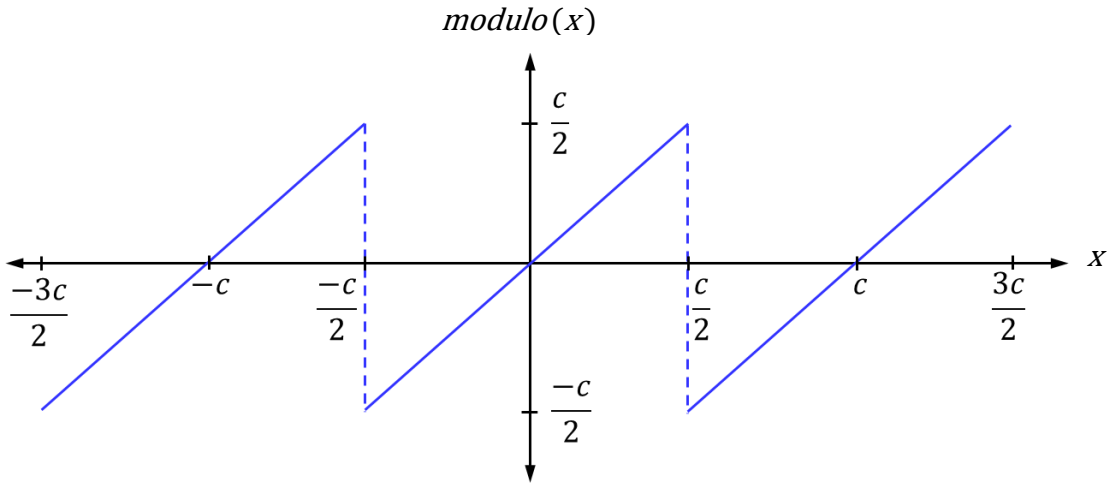

Figure 39: A symmetric modulo (or "saw-tooth") function which maps a shape's X-coordinate onto one revolution (from x=-c/2 to x=c/2) of a cylinder with a circumference c.

In the first stage we pack the shapes into a fixed-height minimum-width rectangle whose height is the same as the height of the cylinder. The width of this rectangle is the minimum circumference of the cylinder around which we can wrap this packing without the shapes on the left edge overlapping the shapes on the right edge. However, the two edges are not interleaved and thus this is not the minimum-circumference cylinder around which we can pack these shapes.

As the first stage packing is, in some sense, close to the minimum-circumference fixed-height cylindrical-packing, the second stage need only make incremental changes to the orientations and positions of the shapes in order to produce a minimum-circumference cylindrical-packing. This incremental movement of the shapes keeps them contained to no more than three successive revolutions of the cylinder. Because of this we can now easily identify a finite number of optimization constraints necessary to produce a minimum-circumference fixed-height cylindrical-packing.

Using this approach, we can formulate a solution to this packing problem as the following two-stage high-level optimization:

Stage 1:

Minimize the width of a fixed height *AAPR*
while satisfying:

1. Nonoverlap constraints between all pairs of shapes in S.
2. Containment constraints between each shape in S and the *AAPR*.

Stage 2:

Initialize the circumference of the cylinder, c, to the width of the *AAPR* determined in Stage-1. Initialize the position, (x_i, y_i), and orientation, α_i, of each shape to the position and orientation determined in Stage-1. Then:

Minimize c
while satisfying:

1. Nonoverlap constraints between all pairs of shapes in S.
2. Vertical containment-constraints for each shape in S.
3. Nonoverlap constraints between every shape in S and an offset copy of every shape in S where the offset copies are offset horizontally by a distance c in the positive x direction.

6.1 Stage-1 Optimization

The Stage-1 optimization is conceptually fairly easy as it is a simple variation on the optimization used to solve packing problem in *Section 4* (Packing Shapes into an Axis-Aligned Rectangular-Boundary using Nonoverlap Constraints) where the rectangular-boundary has a fixed height. In particular, *UV* and *LV* are now a pair of constants and can be eliminated as optimization variables from the optimization problem. This also allows us to simplify the objective developed in *Section* 4.1 (Eq. (1)) to the following:

$$UH - LH \tag{26}$$

as well as eliminate the *UV* and *LV* Jacobian-matrix entries associated with the vertical containment-constraints.

6.2 Stage-2 Optimization

Conceptually, the Stage-2 optimization is a modified version of the Stage-1 optimization. The objective is now the circumference, c, of the fixed height cylinder around which we imagine the packing being wrapped. Additionally, we replaced the stage-1 "hard" horizontal containment-constraints with a second set of nonoverlap constraints which act as a set of "soft" horizontal containment-constraints. This set of soft containment-constraints is designed to let the shapes on the right side of the packing "see" and "interact with" copies of the shapes on the left side of the packing. This allows the shapes on the right side of the packing to interleave with the shapes on the left side as the circumference, c, of the cylinder is minimized.

Note that for the first set of nonoverlap constraints we form one constraint between each pair of shapes in S. For example, if s_1 and s_2 are two shapes in S, we form a nonoverlap constraint between s_1 and s_2 but do not need to also form a nonoverlap constraint between s_2 and s_1 as it would be redundant. However, for the second set of nonoverlap constraints, the soft horizontal containment-constraints, we need to form one constraint between each shape in S and every one of S's offset-shape copies. Specifically, if s_1 and s_2 are two shapes in S and os_1 and os_2 are their offset copies, we form a nonoverlap constraint between s_1 and os_2, a constraint between s_2 and os_1, as well as a nonoverlap constraint between s_1 and os_1, and a constraint between s_2 and os_2. These last two nonoverlap constraints are necessary to guarantee that c remains large enough to prevent any shape from wrapping around the imaginary cylinder and overlapping its self.

Finally, note that for the soft horizontal containment-constraints we arbitrarily chose to offset the shape copies in the positive x direction. We could have just as easily chosen to offset the shape copies in the negative x direction and achieved the same result.

6.3 Implementation Details

To construct the bound-shape-functions for the Stage-2 soft horizontal containment-constraints we first create a set of auxiliary optimization-variables. These auxiliary optimization-variables contain the x positions of the offset shape copies. Specifically, for each s_i in S we need an auxiliary optimization variable, ox_i, to contain the x position of its offset copy. Additionally, we created an IPOPT equality constraint for each s_i to ensures that $ox_i = x_i + c$:

$$0 \leq ox_i - x_i - c \leq 0 \tag{27}$$

Further, since the Stage-2 optimization is intended to incrementally refine the solution obtained by the Stage-1 optimization we need to keep IPOPT from straying too far from the Stage-1 solution at the start of the Stage-2 optimization. To do this we used IPOPT's "warm-start" feature. This feature allows us to preset IPOPT's internal state at the beginning of the Stage-2 optimization with the internal state of the optimizer at the end of the Stage-1 optimization. IPOPT's internal state is captured in the values of the optimization variables and the three sets of Lagrange multipliers: the Lagrange multipliers for the optimization-problem constraints; the Lagrange multipliers for the optimization-variable lower-bound constraints; and the Lagrange multipliers for the optimization-variable upper-bound constraints. To execute the Stage-2 warm-start optimization, we set IPOPT's `warm_start_init_point` option to the value "`yes`" prior to calling the *IpoptSolve* function with the internal state captured from the results of the Stage-1 optimization.

In order for the IPOPT warm-start successive-refinement approach to work, the Stage-1 and Stage-2 optimizations must have the same basic structure. Specifically, each optimization must have the same number of optimization variables and the same number of constraints. With this requirement in mind, we constructed the Stage-1 and Stage-2 optimizations so that they have the same optimization variables, the same objective, and the same set of constraint functions.

To synchronize the objectives and optimization variables between the Stage-1 and Stage-2 optimizations we replaced all instances of the optimization variable c in the Stage-2 optimization with the difference $UH - LH$. That is, the circumference in the Stage-2 optimization will now be represented as the difference of the optimization variables UH and LH instead of the single optimization variable c. This allows us to simultaneously synchronize the objectives and eliminate c as an optimization variable in the Stage-2 optimization.

In order for both optimizations to have the same set of constraint functions, we formed a set of combined constraints which were used in the formation of both the Stage-1 and Stage-2 optimizations. That is, each optimization contained the constraints needed for the Stage-1 optimization and the constraints needed for the stage-2 optimization. In order to keep the hard horizontal containment-constraints (needed for the Stage-1 optimization) from conflicting with the soft horizontal containment-constraints (needed for the Stage-2 optimization) we "deactivate" the soft horizontal containment-constraints during the Stage-1 optimization and then deactivate the hard horizontal

containment-constraints during the Stage-2 optimization. Specifically, given a constraint:

$$C_L \leq C(x_1, x_2, \cdots x_n) \leq C_U \tag{28}$$

where $C(x_1, x_2, \cdots x_n)$ is the constraint function and C_L and C_U are the lower and upper bounds on the constraint, we can deactivate this constraint by setting C_L = -∞ and C_U = ∞. This deactivated constraint can now be included as part of an optimization while having no meaningful impact on the results.

Additionally, note that while auxiliary-variable equality-constraints are only needed in the Stage-2 optimization they were left active in the Stage-1 optimization. While leaving them active in the Stage-1 optimization has no meaningful impact on its results, it saved us the trouble of having to initialize the auxiliary variables at the start of the Stage-2 optimization.

Since we used IPOPT's Limited-memory Hessian-approximation feature to generate the Hessian matrix. We activated and configured this Hessian-approximation feature by setting the `hessian_approximation` option to "`limited-memory`" and the `limited_memory_max_history` option to 50.

Finally, we applied a 20x scaling factor to the objective by setting IPOPT's `obj_scaling_factor` option to 20.

6.4 Packing Results

Figure 40 and *Figure 41* show the Stage-1 and Stage-2 packing results, respectively, for a set of 16 poly-Bézier shapes. Similarly, *Figure 42* and *Figure 43* show the Stage-1 and Stage-2 packing results, respectively, for a set of 20 poly-Bézier shapes.

Figure 40 and *Figure 42* show the Stage-1 packing results for the 16 and 20 shape cases, respectively. In particular, they show three abutting copies of each fixed-height minimum-width *AAPR* packing. Additionally, the *AAPR* for each packing is shown surrounding the center copy. Finally, a pair of dashed lines indicates the positions of the upper and lower vertical-bounds of the fixed-height packings.

Figure 41 and *Figure 43* show the Stage-2 packing results for the 16 and 20 shape cases, respectively. In particular, they show three abutting copies of each puzzle-piece packing as well as a measure of each packing circumference, c, shown below the packing copies. Again, a pair of dashed lines indicates the positions of the upper and lower vertical-bounds of the fixed-height packings.

Comparing *Figure 40* to *Figure 41* and *Figure 42* to *Figure 43* we can see a significant reduction in area both along the boundary between packings as well as internal to the packings. In particular there is an 8% reduction in packing area for the 16-shape packing and an 8.5% reduction in packing area for the 20-shape packing.

More broadly, *Figure 44* shows the reduction in packing area between the Stage-1 and Stage-2 packing results for sets of shapes ranging in size from 5 to 80 shapes. Initially there is no reduction in packing area, due to the limitations imposed by the Stage-2 self-intersection constraints, then there is a sharp increase to the maximum reduction of 12% followed by an asymptotic decay back towards 0% as number of shapes increases.

To illustrate the reason for getting no reduction in the packing area for shape sets containing small numbers of shapes, consider the Stage-1 packing shown in *Figure 45*. This packing contains 8 shapes and its width is entirely determined by, and equal to, the minimum width of shape 6. Because of this, the Stage-2 packing problem is constrained by the self-intersection constraint for shape 6. That is, this constraint will prevent the circumference from being any smaller than the minimum width of shape 6 in order to prevent the left side of shape 6 from overlapping its right side. Note that shape 6 forms the limiting width for sets containing 6, 7, and 8 shapes, while the width of shape 3 forms the limiting width for sets containing 3, 4, and 5 shapes, and finally, the width of shape 2 is the limiting width for the set containing two shapes.

Finally, *Figure 46* and *Figure 47* show the Stage-1 and Stage-2 packing results for the set of 16 shapes when we use uniform boundary-distance constraints with the boundary-distance spacing, d, shown in upper-left corner of *Figure 46*.

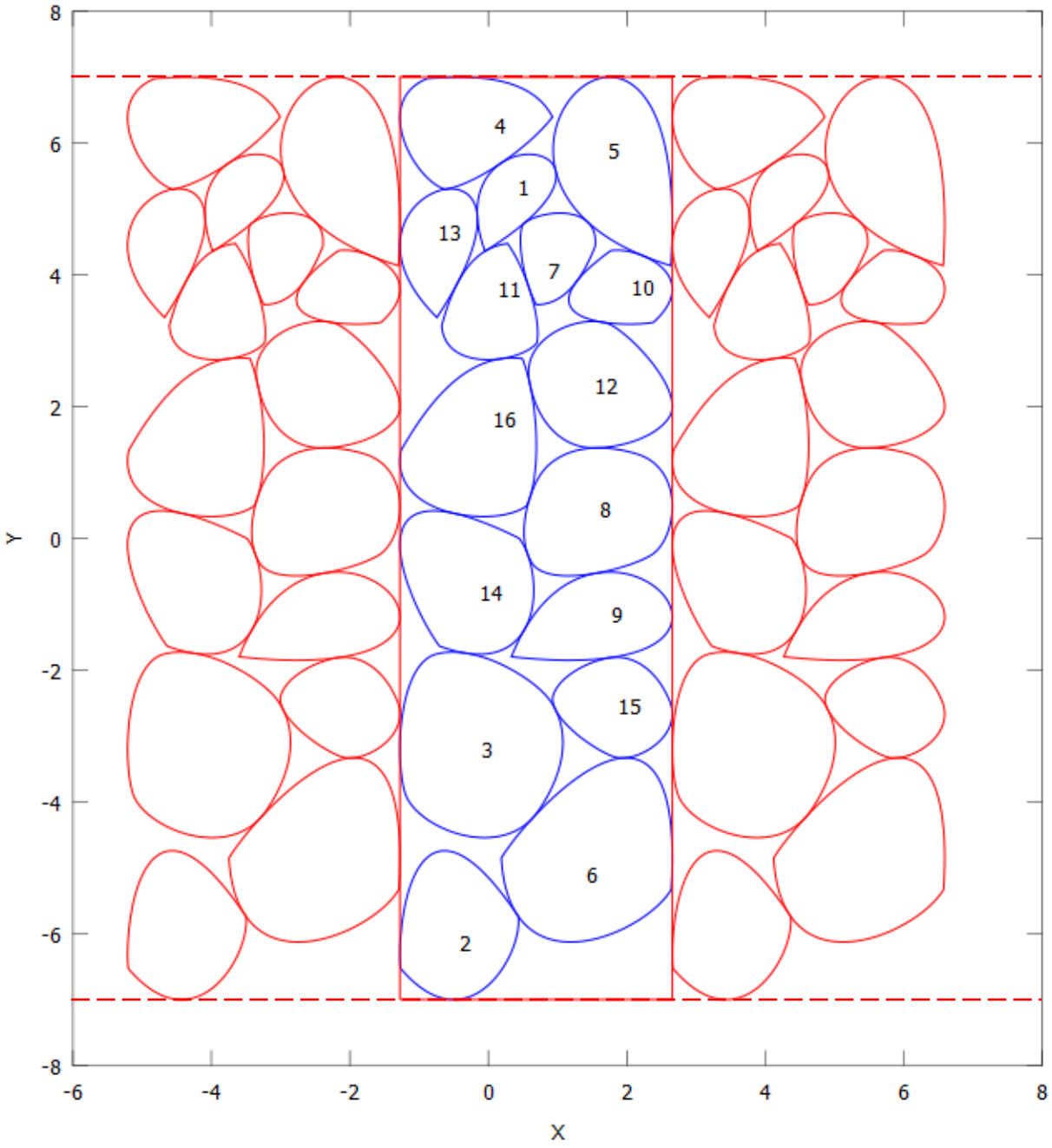

Figure 40: Stage-1 one-dimensional puzzle-piece packing of 16 poly-Bézier shapes (shown in blue) bracketed by its left and right neighbors (shown in red).

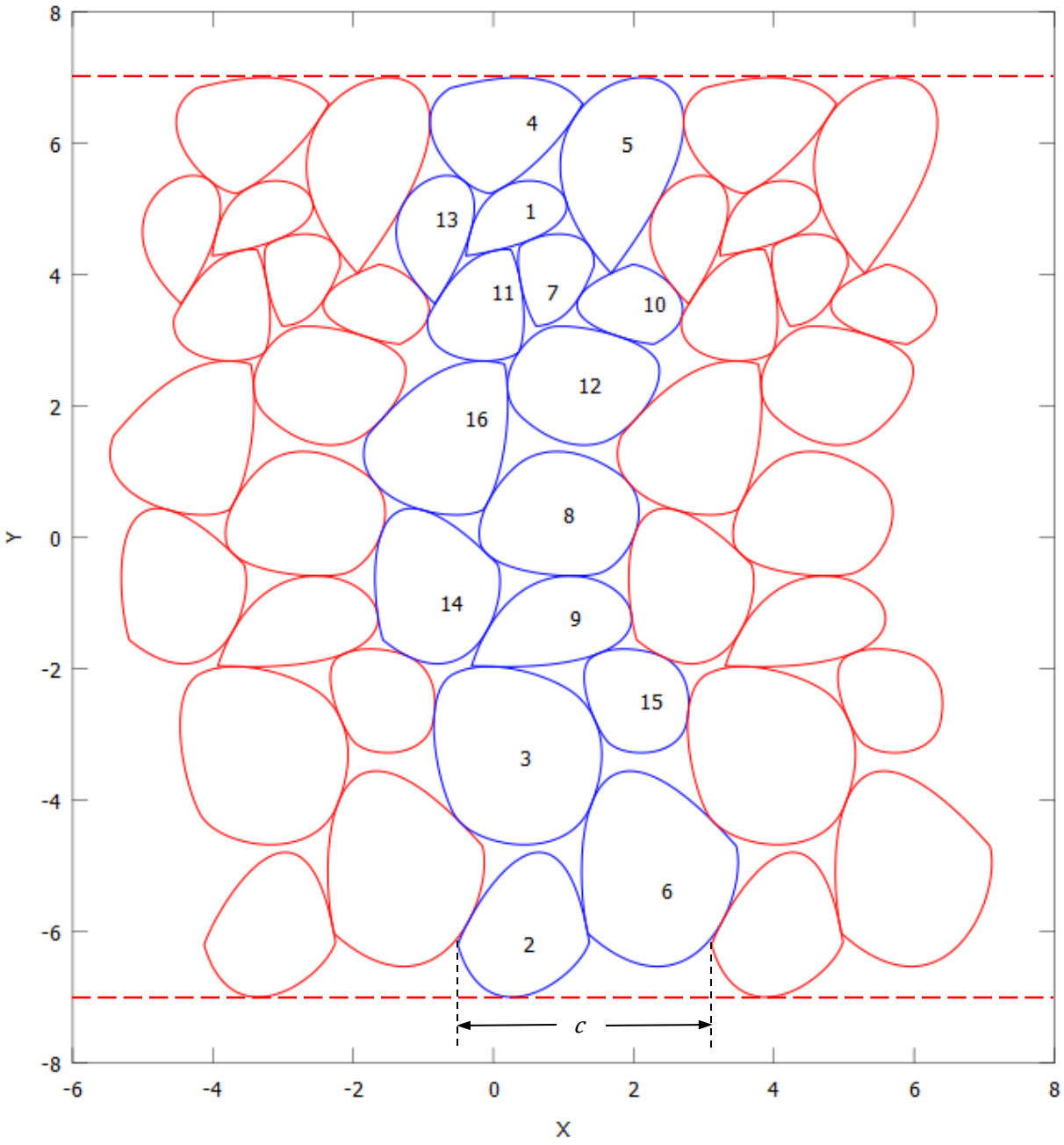

Figure 41: Stage-2 one-dimensional puzzle-piece packing of 16 poly-Bézier shapes (shown in blue) bracketed by its left and right neighbors (shown in red).

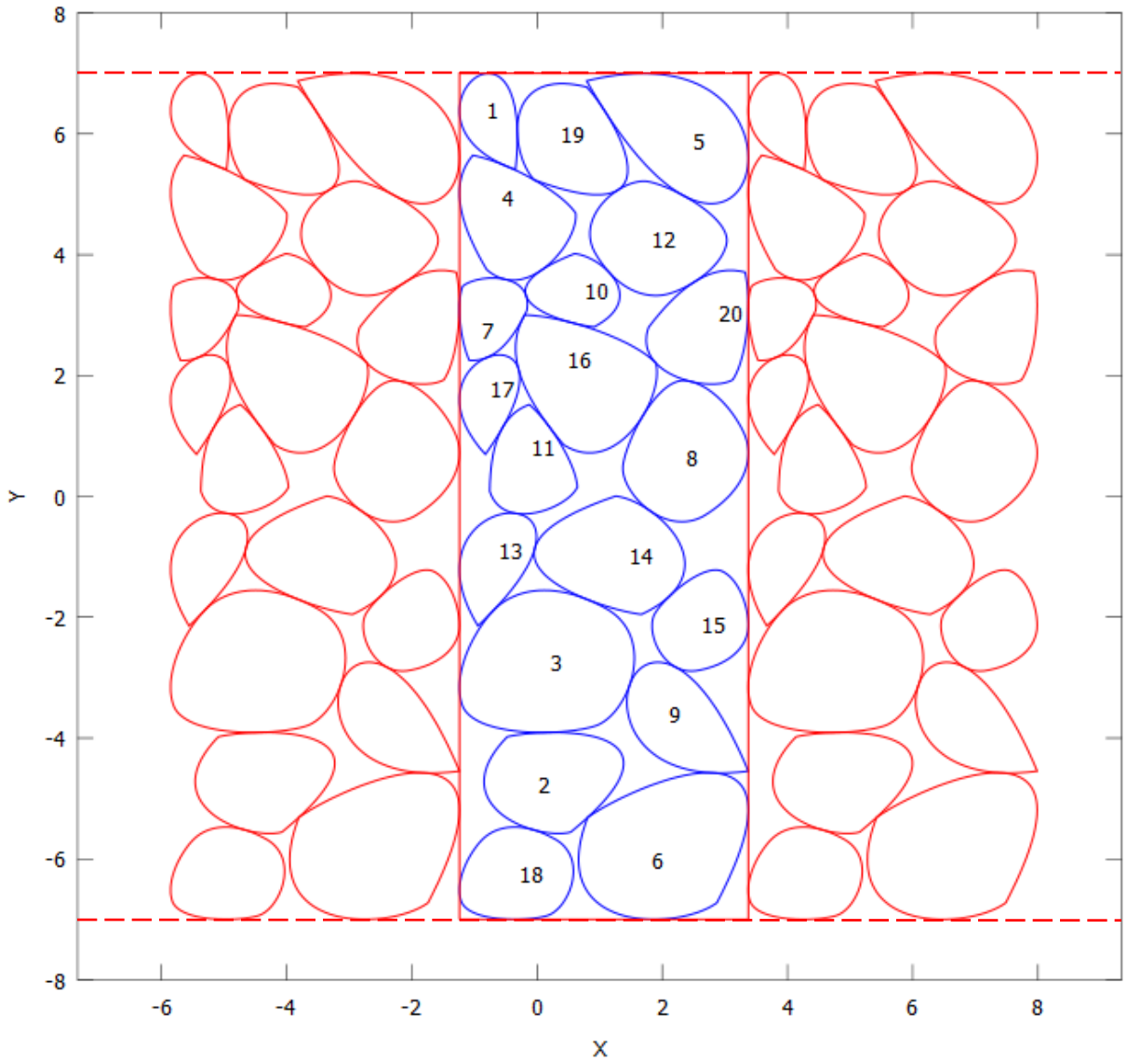

Figure 42: Stage-1 one-dimensional puzzle-piece packing of 20 poly-Bézier shapes.

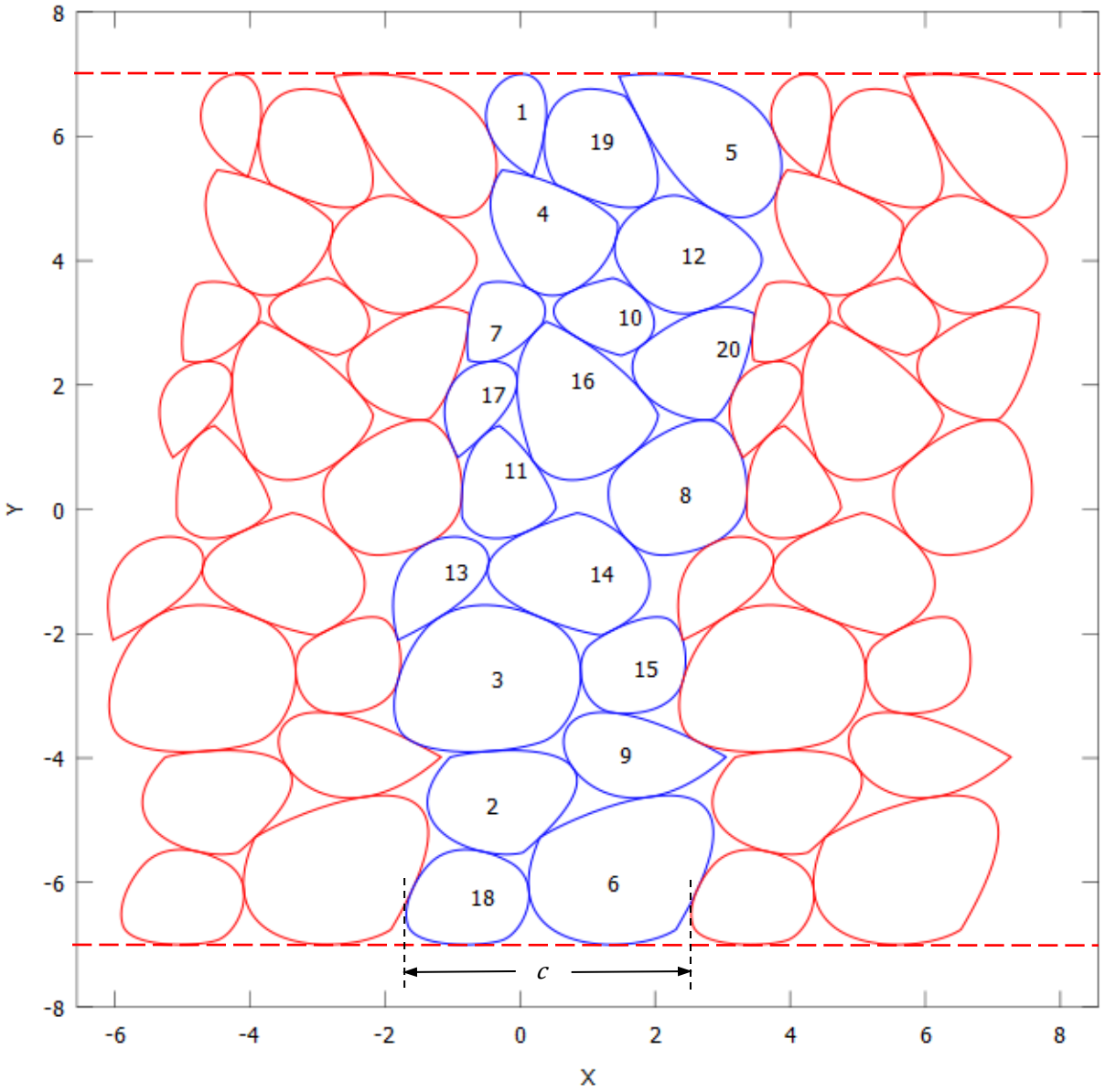

Figure 43: Stage-2 one-dimensional puzzle-piece packing of 20 poly-Bézier shapes.

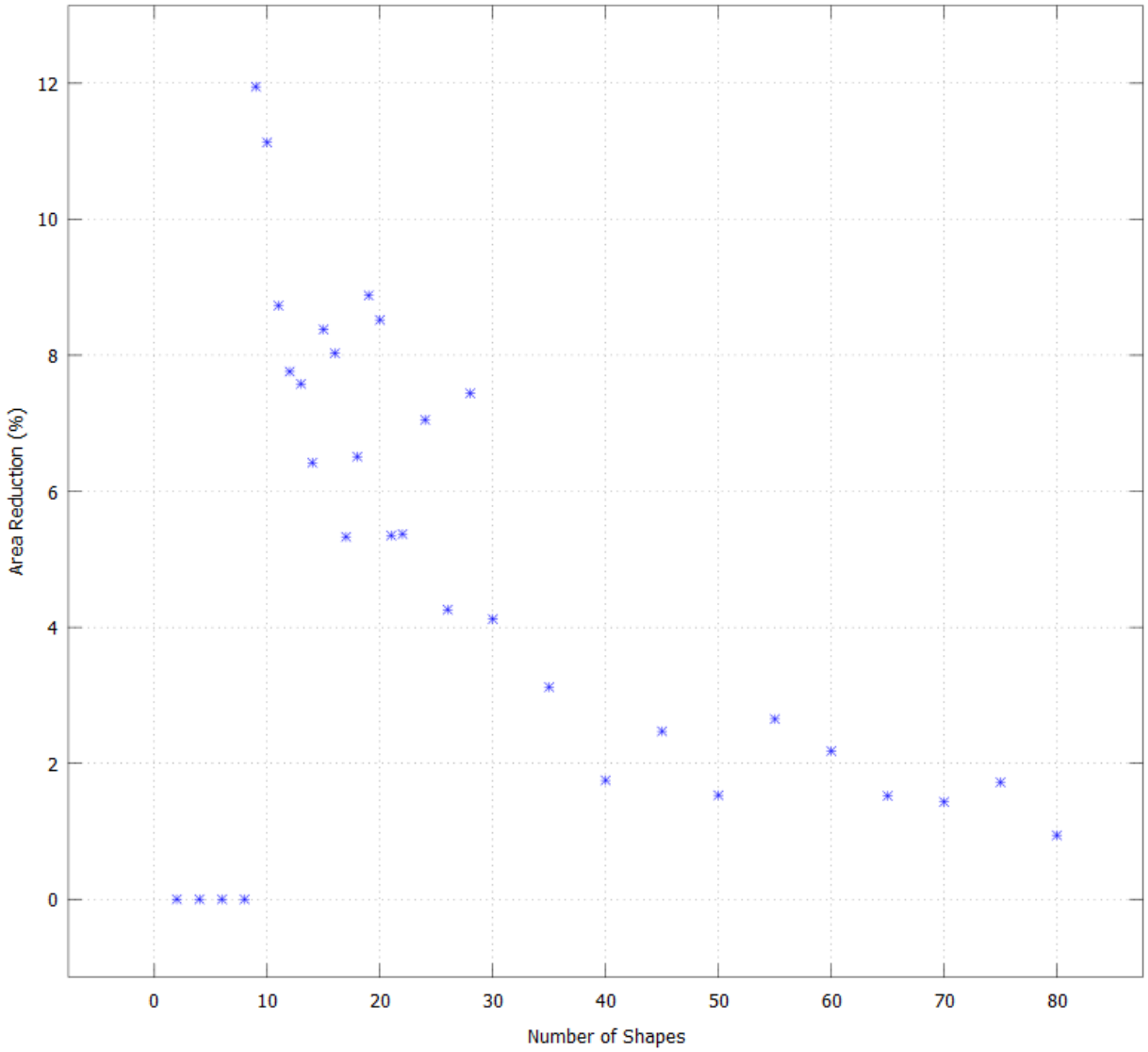

Figure 44: Reduction in packing area between Stage-1 and Stage-2 for one-dimensional puzzle-piece packing.

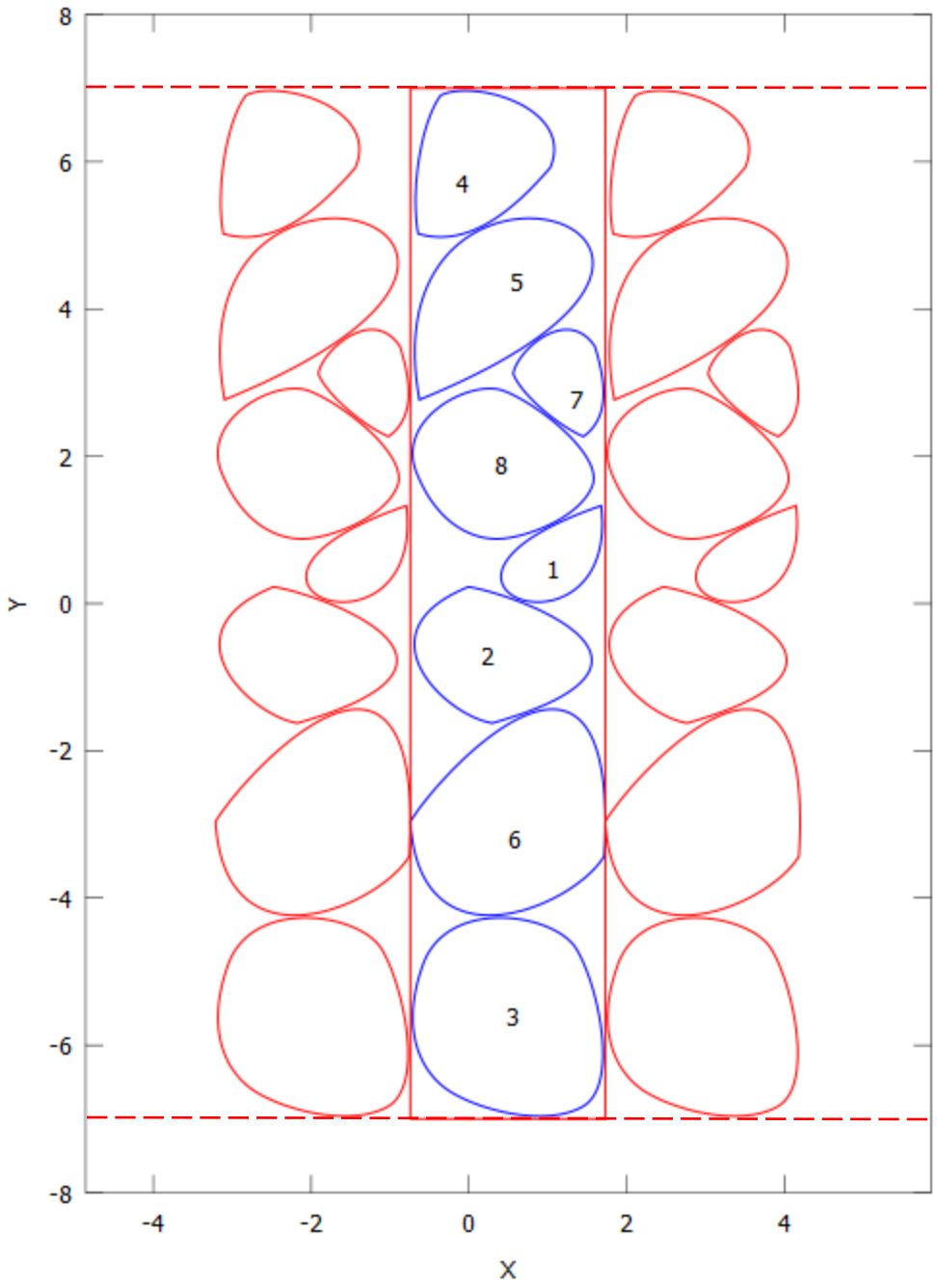

Figure 45: Stage-1 one-dimensional puzzle-piece packing of 8 poly-Bézier shapes where the packing width limited by the width of shape 6.

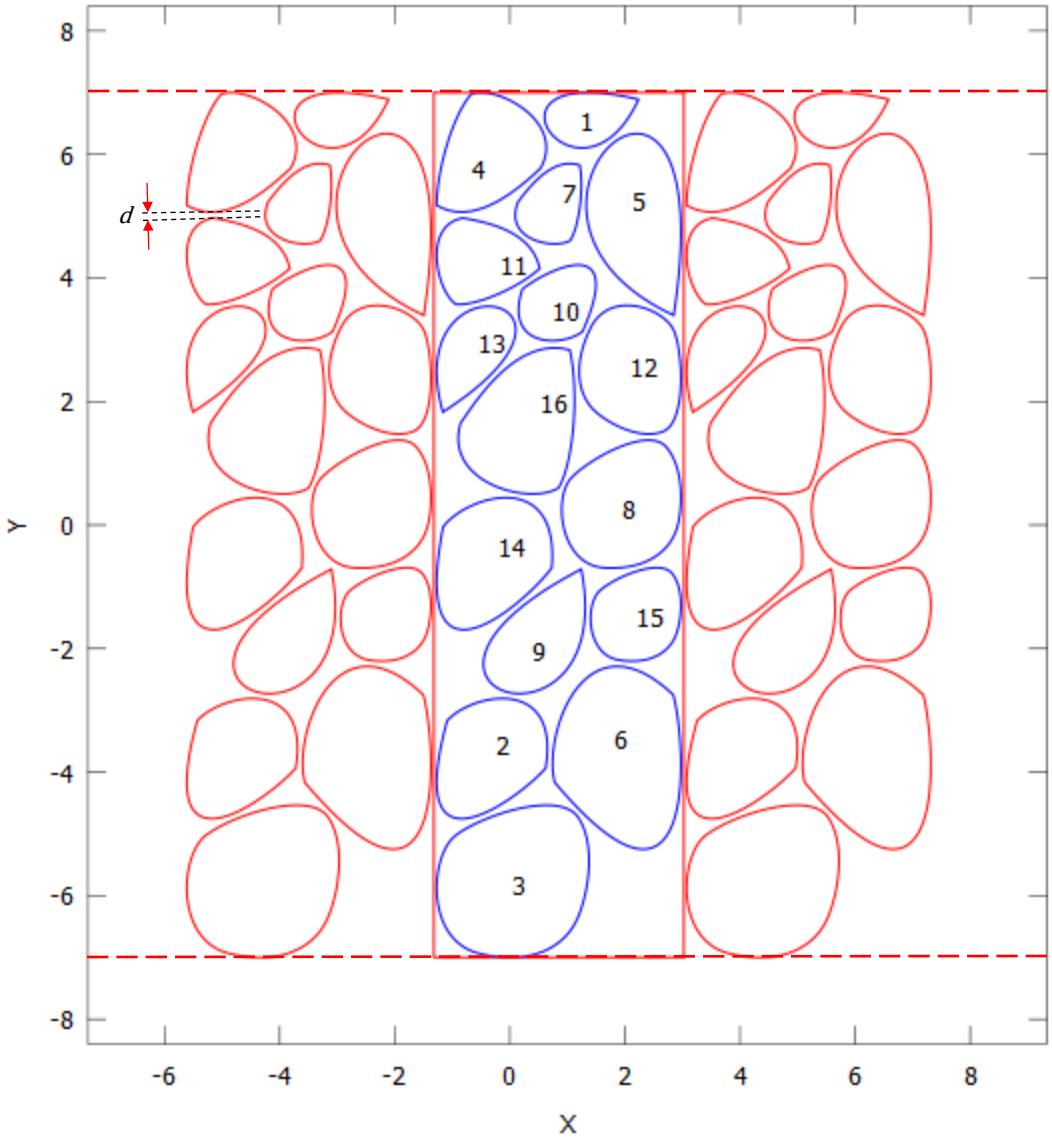

Figure 46: Stage-1 one-dimensional puzzle-piece packing of 16 poly-Bézier shapes using uniform boundary-distance constraints.

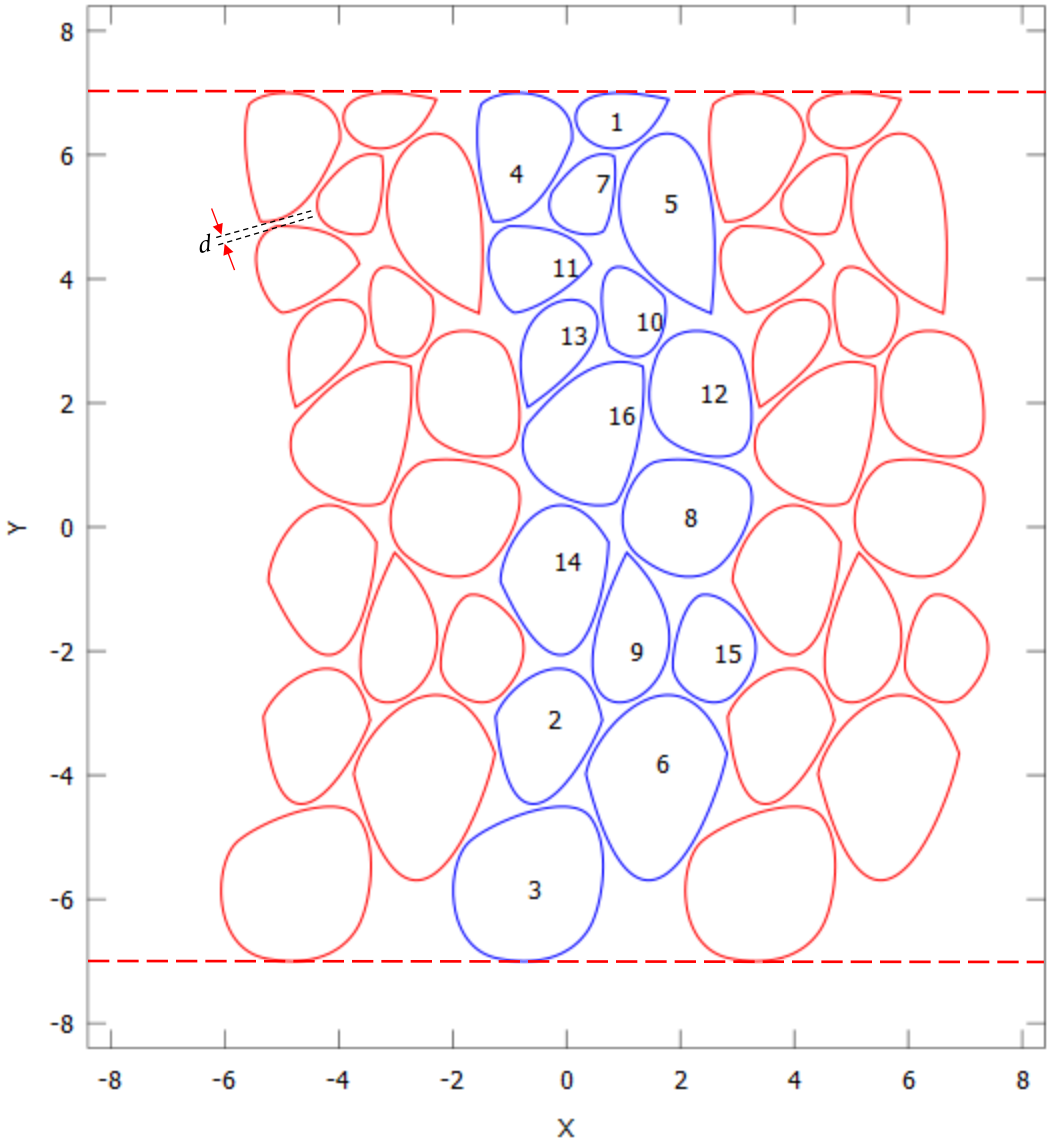

Figure 47: Stage-2 one-dimensional puzzle-piece packing of 16 poly-Bézier shapes using uniform boundary-distance constraints.

7 Two-dimensional Puzzle-piece Packing

In this section we will develop a solution for packing a set of variable-orientation poly-Bézier shapes into a minimum area where the left boundary of the packing can be wrapped around to fit snuggly into, or "interleave" with, the right boundary and, similarly, the top boundary of the packing can be wrapped around to fit snuggly into the bottom boundary. This packing is the simplest two-dimensional generalization of the one-dimensional puzzle-piece packing developed in the previous section.

We refer to the resulting packing as a two-dimensional rectangular puzzle-piece packing since it can be used to form a planar tessellation where every element in the tessellation interlocks tightly into its immediately adjacent horizontal and vertical neighbors. These interlocking boundaries eliminate the "hard" straight-line packing-boundaries giving the optimizer more freedom in how it uses the space within the packing boundary. In particular, it eliminates the wasted space at the boundary between adjacent elements that would have been present had the elements been formed by packing the shapes inside a rectangle.

A formal description of this two-dimensional puzzle-piece packing-problem can be stated as follows:

Given:

1. a set of variable-orientation poly-Bézier shapes, S, containing n shapes;
2. a vertical cylinder of circumference cv;
3. a horizontal cylinder of circumference ch;

determine the location of the origin, (x_i, y_i), and the rotation angle, α_i, of each shape, s_i, in S that minimizes $cv \cdot ch$ such that:

1. the packing is connected;
2. no shape s_i in S overlaps its self or any other shape in S when the packing is wrapped around the lateral surface of the vertical cylinder;
3. no shape s_i in S overlaps its self or any other shape in S when the packing is wrapped around the lateral surface of the horizontal cylinder.

Note that by minimizing the product $cv \cdot ch$ we are also minimizing the area of the boundary enclosing the shapes. We show this using a simple constructive geometric proof illustrated, without loss of generality, in *Figure 48*. In particular, EB is the boundary enclosing the shapes. EB has a horizontal circumference, ch, and vertical circumference, cv. Breaking EB into the three pieces (P1, P2, and P3) shown in *Figure 48* we see that

$$Area(\mathrm{EB}) = Area(\mathrm{P1}) + Area(\mathrm{P2}) + Area(\mathrm{P3}) \tag{29}$$

Further, from the upper-right red-square shown in *Figure 48*, we also see that

$$Area(\mathrm{P1}) + Area(\mathrm{P2}) + Area(\mathrm{P3}) = ch \cdot cv \tag{30}$$

and therefore

$$Area(\mathrm{EB}) = ch \cdot cv \tag{31}$$

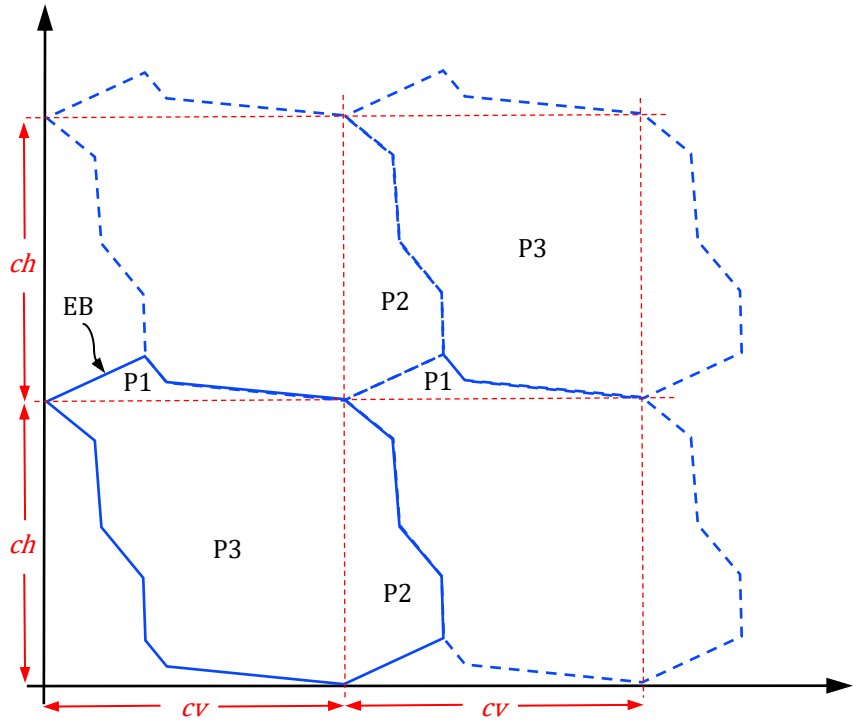

Figure 48

For the same reasons outlined in *section 6* for the one-dimensional puzzle-piece packing-problem, solving the two-dimensional rectangular puzzle-piece packing-problem as a single optimization is impractical as we cannot identify a finite set of constraints. As a result, we will again develop and implement the solution to the two-dimensional rectangular puzzle-piece packing-problem using a two-stage successive-refinement strategy.

In the first stage we pack the shapes into a minimum-area rectangle. The width of this rectangle is the minimum circumference of a vertical cylinder around which we can wrap this packing without the shapes on the left edge overlapping the shapes on the right edge. Similarly, the height of this rectangle is the minimum circumference of a horizontal cylinder around which we can wrap this packing without the shapes on the top edge overlapping the shapes on the bottom edge. However, neither set of edges is interleaved and thus the area of the packing has not been minimized.

As the first stage packing is, in some sense, close to the two-dimensional rectangular puzzle-piece packing, the second stage need only make incremental changes to the orientations and positions of the shapes in order to produce a two-dimensional rectangular puzzle-piece packing. This incremental movement of the shapes keeps them contained to no more than three successive revolutions of each of the horizontal and vertical cylinders. Because of this we can now easily identify a finite number of optimization constraints necessary to produce a two-dimensional rectangular puzzle-piece packing.

Using this approach, we can formulate a solution to the two-dimensional rectangular puzzle-piece packing-problem as the following two-stage high-level optimization:

Stage 1:

Minimize the area of an *AAPR*
while satisfying:

1. Nonoverlap constraints between all pairs of shapes in S.
2. Containment constraints between each shape in S and the *AAPR*.

Note that this stage-1 optimization is identical to the optimization used to solve the nonoverlap-constrained packing-problem in *Section 4*. As such, there is no need to elaborate any further on the stage-1 optimization.

Stage 2:

Initialize the circumference of the horizontal cylinder, ch, to the height of the *AAPR* determined in Stage-1. Initialize the circumference of the vertical cylinder, cv, to the width of the *AAPR* determined in Stage-1. Initialize the position, (x_i, y_i), and orientation, α_i, of each shape to the position and orientation determined in Stage-1. Then:

Minimize $ch \cdot cv$
while satisfying:

1. Nonoverlap constraints between all pairs of shapes in S.
2. Nonoverlap constraints between every shape in S and an offset copy of every shape in S where the offset copies are offset vertically by a distance ch in the positive y direction.
3. Nonoverlap constraints between every shape in S and an offset copy of every shape in S where the offset copies are offset horizontally by a distance cv in the positive x direction.
4. Nonoverlap constraints between every shape in S and an offset copy of every other shape in S where the offset copies are offset vertically by a distance ch in the positive y direction and horizontally a distance cv in the positive x direction .
5. Nonoverlap constraints between every shape in S and an offset copy of every other shape in S where the offset copies are offset vertically by a distance ch in the positive y direction and horizontally a distance cv in the negative x direction .

7.1 Stage-2 Optimization

Conceptually, the Stage-2 optimization is a highly modified version of the Stage-1 optimization. The objective is now the product of the circumferences, $ch \cdot cv$, of the horizontal and vertical cylinders around which we imagine the packing being wrapped instead of the product of the *AAPR* width and height. Additionally, we have replaced the "hard" *AAPR* containment-constraints with four sets of nonoverlap constraints which collectively act as a set of "soft" containment-constraints. The first set of these nonoverlap constraints is designed to let the shapes on the top side of the packing "see" and "interact with" copies of the shapes on the bottom side of the packing. Similarly, the second set of these nonoverlap constraints lets the shapes on the right side of the packing see and interact with copies of the shapes on the left side of the packing. The remaining two sets of nonoverlap constraints allow the shapes on the corners to see and interact with the shapes on the diagonally opposite corner. The combined effect of the four sets of soft containment-constraints is to allow the shapes on the top side of the packing to interleave with the shapes on the bottom side and

the shapes on the right side of the packing to interleave with the shapes on the left side as the product of the circumferences, $ch \cdot cv$, of the horizontal and vertical cylinders is minimized.

Note that for the first set of nonoverlap constraints we form one constraint between each pair of shapes in S. For example, if s_1 and s_2 are two shapes in S, we form a nonoverlap constraint between s_1 and s_2 but do not need to also form a nonoverlap constraint between s_2 and s_1 as it would be redundant. However, for the second and third set of nonoverlap constraints (the constraints forming the horizontal and vertical soft containment-constraints) we need to form one constraint between each shape in S and every one of S's offset shape copies. Specifically, if s_1 and s_2 are two shapes in S and os_1 and os_2 are their offset copies, we form a nonoverlap constraint between s_1 and os_2, a constraint between s_2 and os_1, as well as a nonoverlap constraint between s_1 and os_1, and a constraint between s_2 and os_2. These last two nonoverlap constraints are necessary to prevent any shape from wrapping around one of the imaginary cylinders and overlapping its self. For the fourth and fifth set of nonoverlap constraints (the nonoverlap constraints forming the diagonal soft containment-constraints) we need to form one constraint between each shape in S and every other one of S's offset shape copies. Specifically, if s_1 and s_2 are two shapes in S and os_1 and os_2 are their diagonally offset copies, we form a nonoverlap constraint between s_1 and os_2, and a constraint between s_2 and os_1. However, we do not need to form a nonoverlap constraint between s_1 and os_1, or a constraint between s_2 and os_2 as the horizontal and vertical soft containment-constraints that prevent self-intersection are already sufficient to prevent any shape from wrapping around and overlapping its self.

Finally, note that the choice of positive offsets for the horizontal and vertical soft containment-constraints was arbitrary. We could have just as easily chosen to offset either or both sets of shape copies in the opposite direction and achieved the same result. Similarly, the choice of direction for the diagonal offsets was also arbitrary. That is, for each set of diagonal soft containment-constraints we have a choice of two offsets and we will achieve the same result regardless of which of these two offsets we chose to use.

7.2 Implementation Details

To construct the bound-shape-functions for the Stage-2 soft containment-constraints we first create three sets of auxiliary optimization-variables. The first set of auxiliary optimization-variables contains the x positions of both the horizontally-offset shape-copies and the diagonally-offset shape-copies used in the first set of soft diagonal containment-constraints. Specifically, for each s_i in S we need an auxiliary optimization variable, $ox1_i$, to represent the x position of both the horizontally-offset copy used in the construction of the soft horizontal containment-constraints and the diagonally-offset copy used in the construction of first set of soft diagonal containment-constraints. The second set of auxiliary optimization-variables contain the y positions of its vertically-offset shape-copies and the y positions of both of its diagonally-offset shape-copies. Specifically, for each s_i in S we need an auxiliary optimization variable, oy_i, to represent the y position of both the vertically offset copy used in the construction of the soft vertical containment-constraints and the diagonally-offset copy used in the construction of both of the soft diagonal containment-constraints. The third set of auxiliary optimization-variables contain the

x positions of the diagonally-offset shape-copies used in the second set of soft diagonal containment-constraints. Specifically, for each s_i in S we need an auxiliary optimization variable, $ox2_i$, to represent the x position of its diagonally offset copy used in the construction of the second soft diagonal containment-constraints.

Additionally, we created a set of IPOPT equality constraints for each s_i to ensure that $ox1_i = x_i + cv$, $oy_i = y_i + ch$, and $ox2_i = x_i - cv$:

$$0 \le ox1_i - x_i - cv \le 0 \tag{32}$$

$$0 \le oy_i - y_i - ch \le 0 \tag{33}$$

$$0 \le ox2_i - x_i + cv \le 0 \tag{34}$$

Further, since the Stage-2 optimization is intended to incrementally refine the solution obtained by the Stage-1 optimization we need to keep IPOPT from straying too far from the Stage-1 solution at the start of the Stage-2 optimization. To do this we used IPOPT's "warm-start" feature. This feature allows us to preset IPOPT's internal state at the beginning of the Stage-2 optimization with the internal state of the optimizer at the end of the Stage-1 optimization. IPOPT's internal state is captured in the values of the optimization variables and the three sets of Lagrange multipliers: the Lagrange multipliers for the optimization-problem constraints; the Lagrange multipliers for the optimization-variable lower-bound constraints; and the Lagrange multipliers for the optimization-variable upper-bound constraints. To execute the Stage-2 warm-start optimization, we set IPOPT's `warm_start_init_point` option to the value "`yes`" prior to calling the *IpoptSolve* function with the internal state captured from the results of the Stage-1 optimization.

In order for the IPOPT warm-start successive-refinement approach to work, the Stage-1 and Stage-2 optimizations must have the same basic structure. Specifically, each optimization must have the same number of optimization variables and the same number of constraints. With this requirement in mind, we constructed the Stage-1 and Stage-2 optimizations so that they have the same optimization variables, the same objective, and the same set of constraint functions.

To synchronize the objectives and optimization variables between the Stage-1 and Stage-2 optimizations we replaced all instances of the optimization variables cv and ch in the Stage-2 optimization with the differences $UH - LH$ and $UV - LV$ respectively. That is, the vertical circumference in the Stage-2 optimization will now be represented as the difference of the optimization variables UH and LH instead of the single optimization variable cv. Similarly, the horizontal circumference in the Stage-2 optimization will now be represented as the difference of the optimization variables UV and LV instead of the single optimization variable ch. This allows us to simultaneously synchronize the objectives and eliminate cv and ch as optimization variables in the Stage-2 optimization.

In order for both optimizations to have the same set of constraint functions, we formed a set of combined constraints which were used in the formation of both the Stage-1 and Stage-2 optimizations. That is, each optimization contained the constraints needed for the Stage-1 optimization and the constraints needed for the stage-2 optimization. In order to keep the hard containment-constraints (needed for the Stage-1 optimization) from conflicting with the soft containment-constraints (needed for the Stage-2 optimization) we "deactivate" the soft containment-constraints during the Stage-1

optimization and then deactivate the hard containment-constraints during the Stage-2 optimization. Specifically, given a constraint:

$$C_L \leq C(x_1, x_2, \cdots x_n) \leq C_U \tag{35}$$

where $C(x_1, x_2, \cdots x_n)$ is the constraint function and C_L and C_U are the lower and upper bounds on the constraint, we can deactivate this constraint by setting C_L = -∞ and C_U = ∞. This deactivated constraint can now be included as part of an optimization while having no meaningful impact on the results.

Additionally, note that while auxiliary-variable equality-constraints are only needed in the Stage-2 optimization they were left active in the Stage-1 optimization. While leaving them active in the Stage-1 optimization has no meaningful impact on its results, it saved us the trouble of having to initialize the auxiliary variables at the start of the Stage-2 optimization.

Since, we used IPOPT's Limited-memory Hessian-approximation feature to generate the Hessian matrix. We activated and configured this Hessian-approximation feature by setting the `hessian_approximation` option to "`limited-memory`" and the `limited_memory_max_history` option to 50.

Finally, we applied a 50x scaling factor to the objective by setting IPOPT's `obj_scaling_factor` option to 50.

7.3 Packing Results

Figure 49 shows the reduction in packing area between the Stage-1 and Stage-2 packing results for sets of shapes ranging in size from 5 to 80 shapes. From this figure we see a maximum reduction of 12% for packings containing a small number of shapes. This is followed by an asymptotic decay back towards 0% as number of shapes increases.

Figure 50 through *Figure 65* show the stage-1 and stage-2 packing results for sets of shapes containing 14, 24, 34, and 70 poly-Bézier shapes, respectively. There are four figures associated with each set of shapes. The first is the Stage-1 packing result shown in blue and surrounded by its eight nearest neighbors shown in red and black. The second is the Stage-2 packing result shown in blue and surrounded by its eight nearest neighbors shown in red and black. The third is an enlarged copy of the first figure showing the Stage-1 packing in greater detail. The fourth is an enlarged copy of the second figure showing the Stage-2 packing in greater detail.

For each of the packings for the four sets of shapes we can see that the soft containment-constraints used in the stage-2 optimization allows the optimizer to recover wasted space that resulted from the use of hard containment-constraints in the Stage-1 optimization. In particular, it should be noted that the recovered space came not only from the packing boundary but also from the interior of the packing. This effect on the interior of the packings is present in all four cases but can be most clearly seen when comparing the Stage-1 and Stage-2 packings for the set of 34 shapes shown in *Figure 60* and *Figure 61*, respectively.

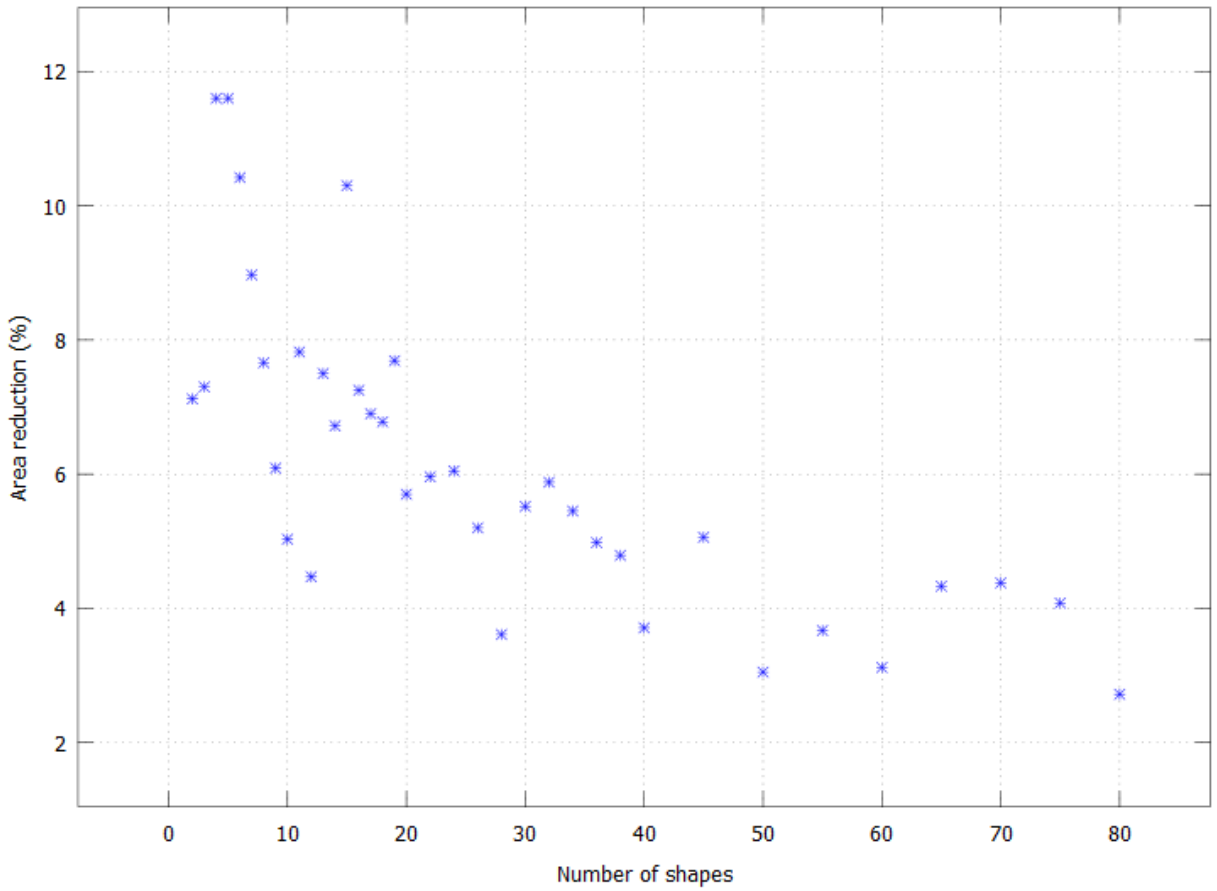

Figure 49: Reduction in packing area between Stage-1 and Stage-2 for two-dimensional puzzle-piece packing.

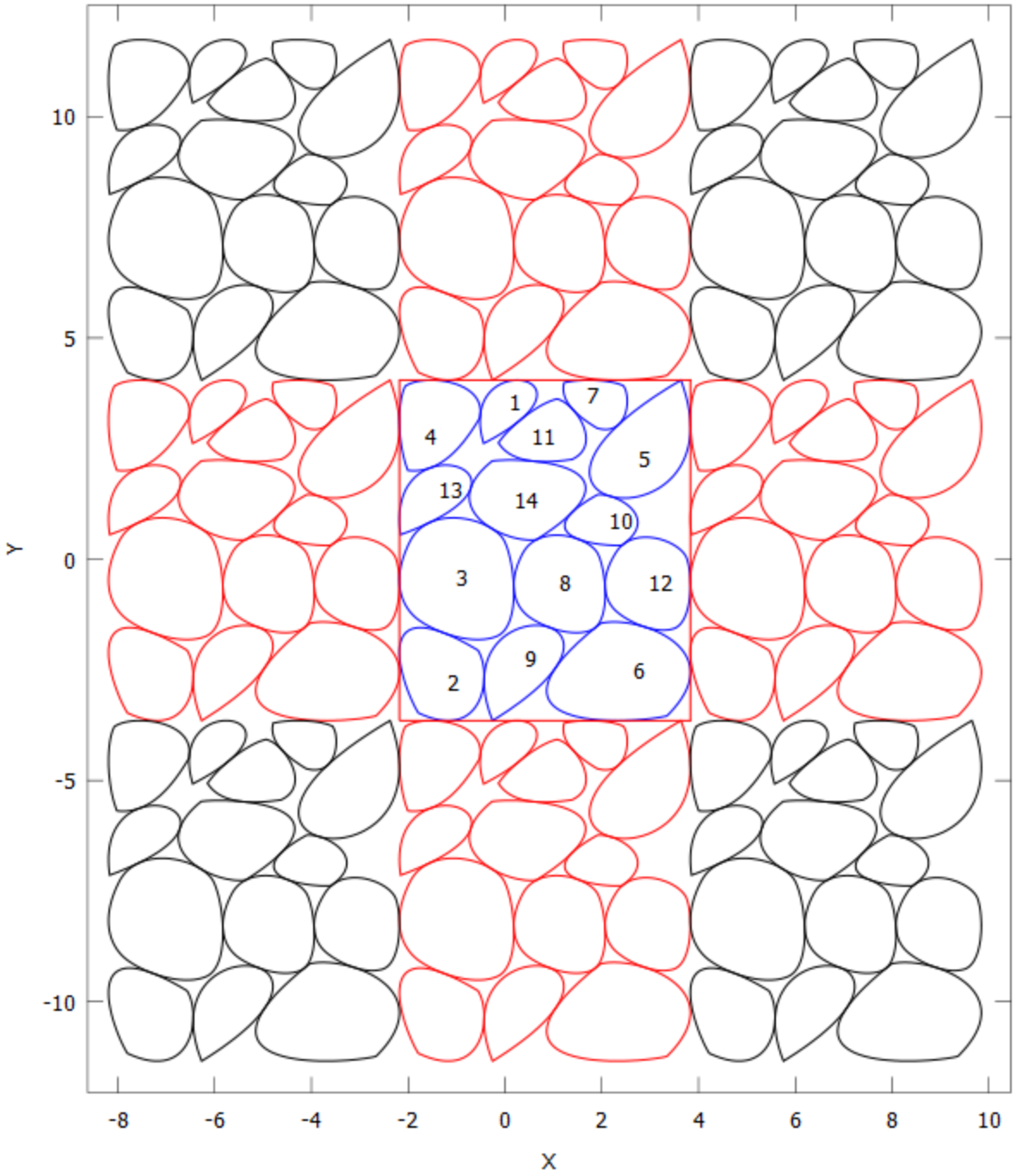

Figure 50: Stage-1 two-dimensional puzzle-piece packing of 14 poly-Bézier shapes (shown in blue) surrounded by its eight nearest neighbors (shown in red and black).

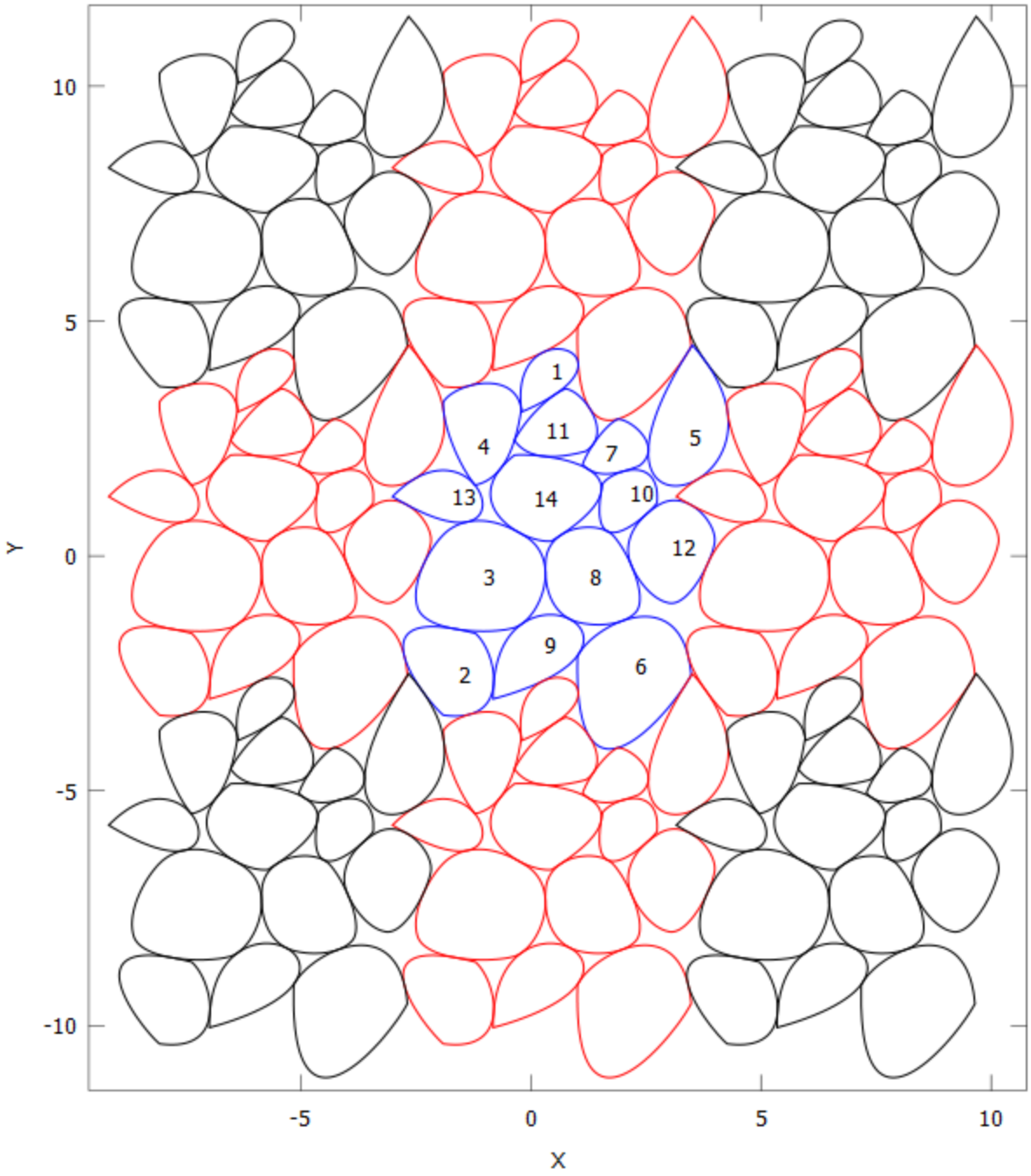

Figure 51: Stage-2 two-dimensional puzzle-piece packing of 14 poly-Bézier shapes (shown in blue) surrounded by its eight nearest neighbors (shown in red and black).

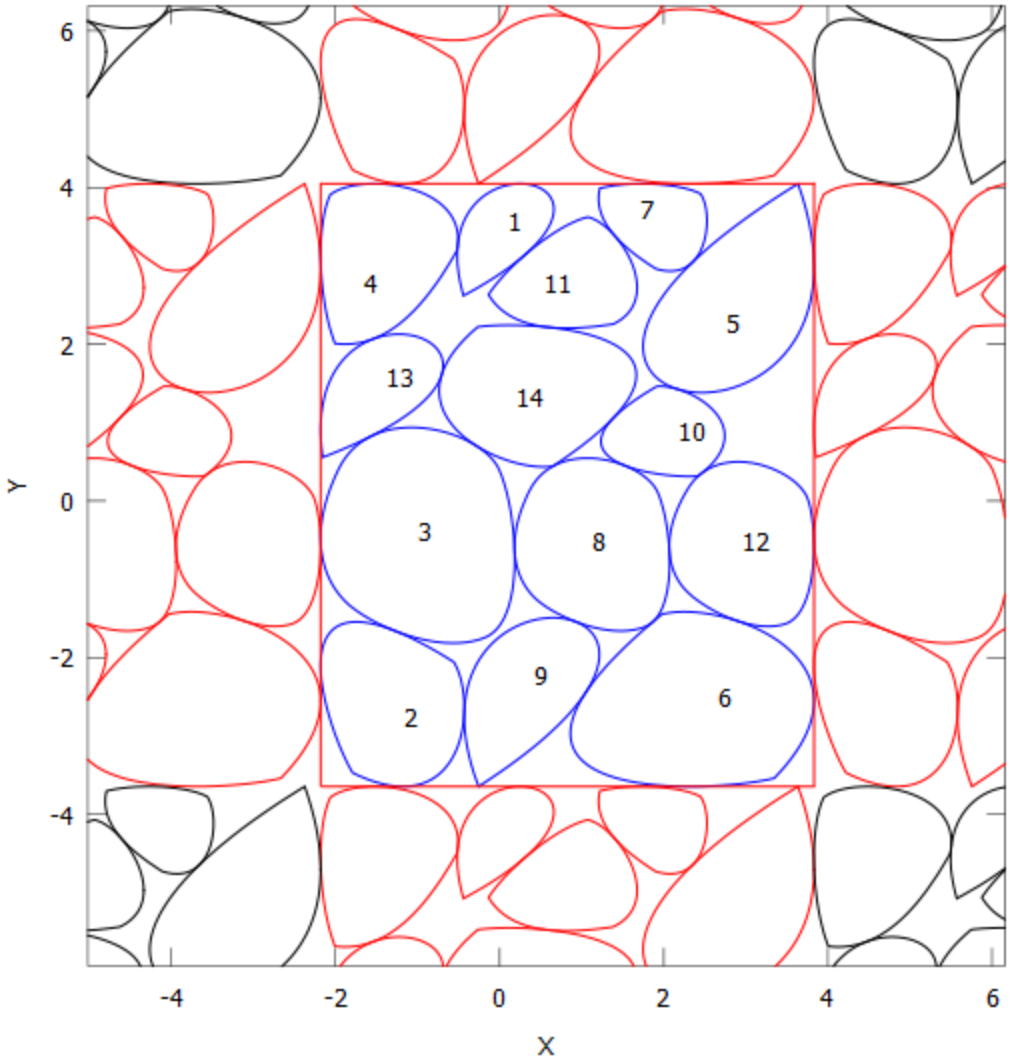

Figure 52: Enlarged view of the Stage-1 packing of 14 poly-Bézier shapes shown in *Figure 50*.

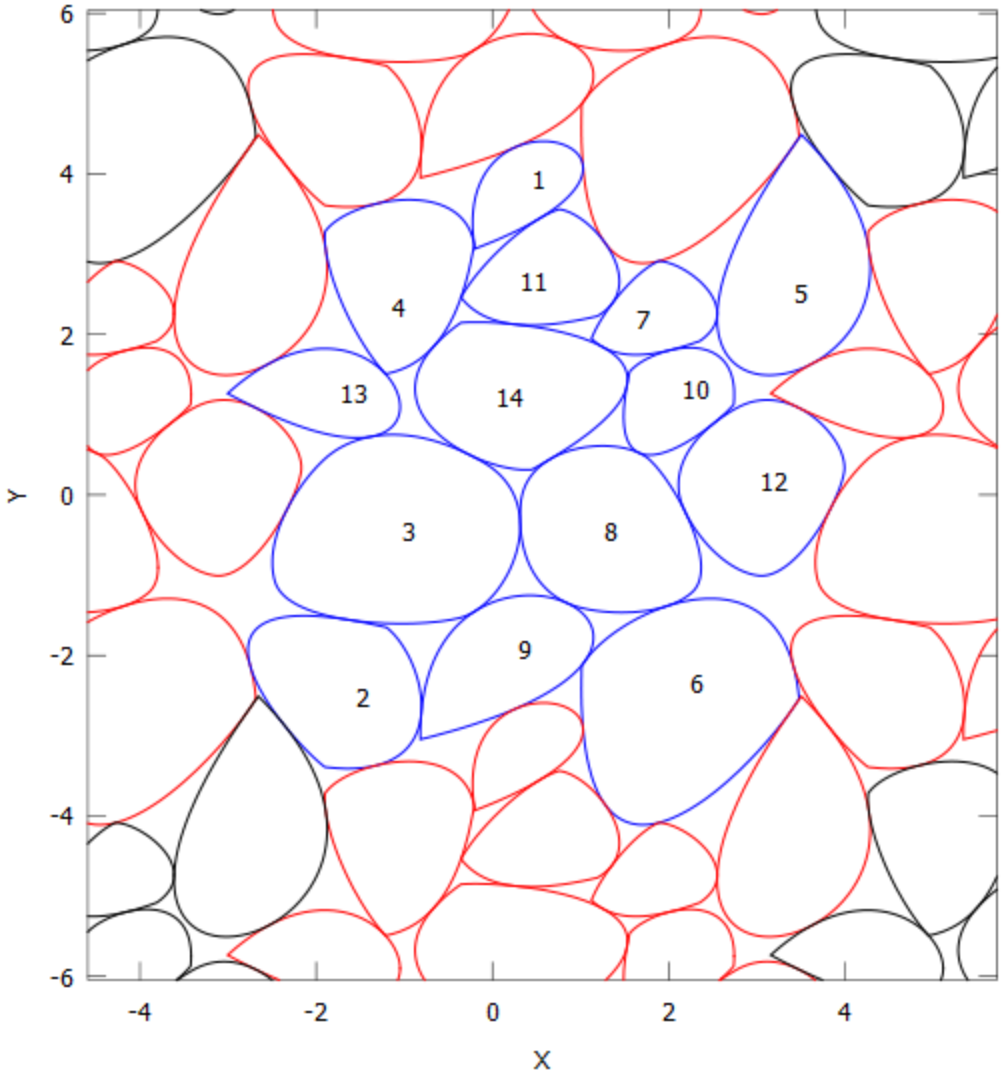

Figure 53: Enlarged view of the Stage-2 packing of 14 poly-Bézier shapes shown in *Figure 51*.

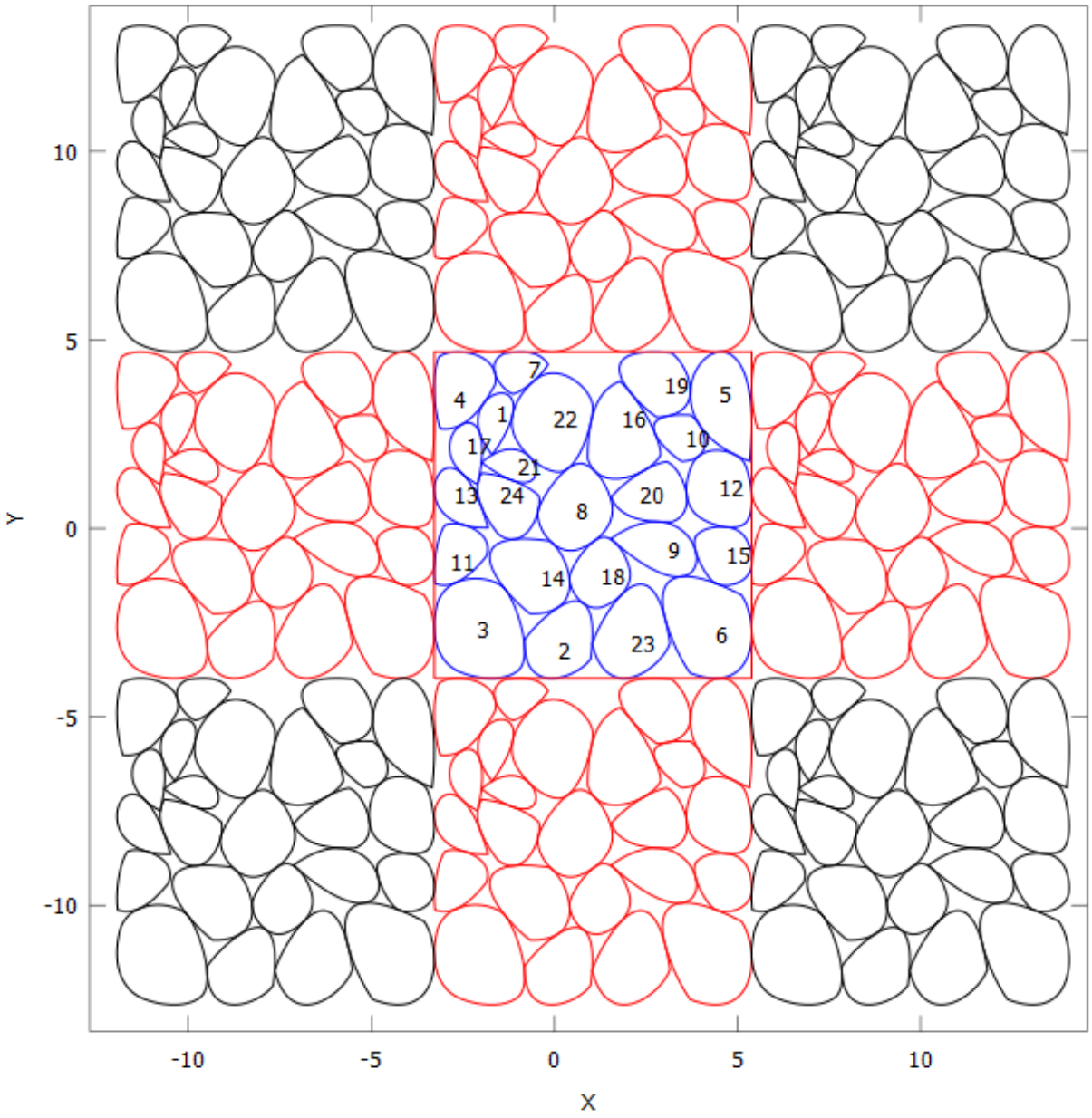

Figure 54: Stage-1 two-dimensional puzzle-piece packing of 24 poly-Bézier shapes surrounded by its eight nearest neighbors.

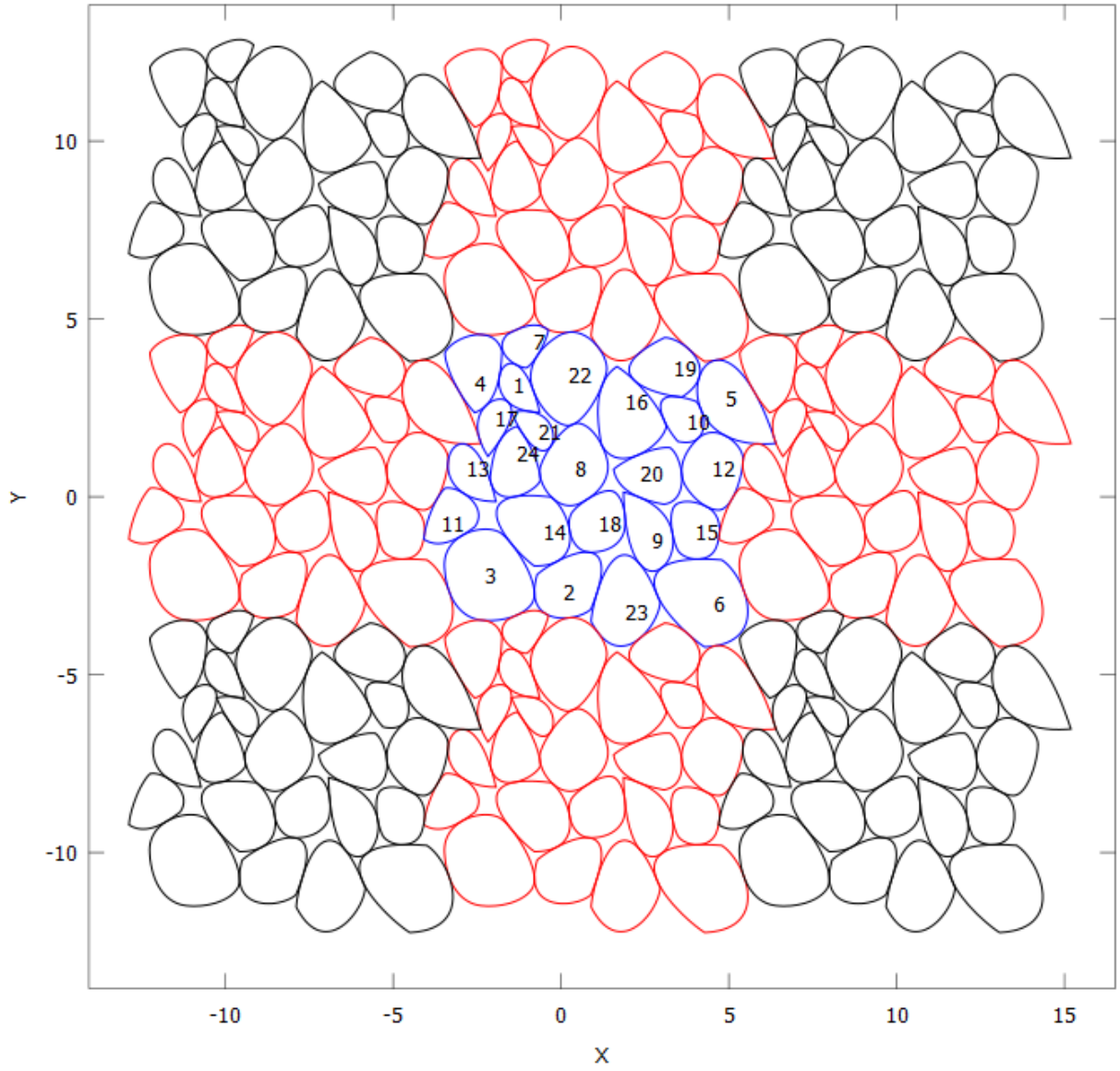

Figure 55: Stage-2 two-dimensional puzzle-piece packing of 24 poly-Bézier shapes surrounded by its eight nearest neighbors.

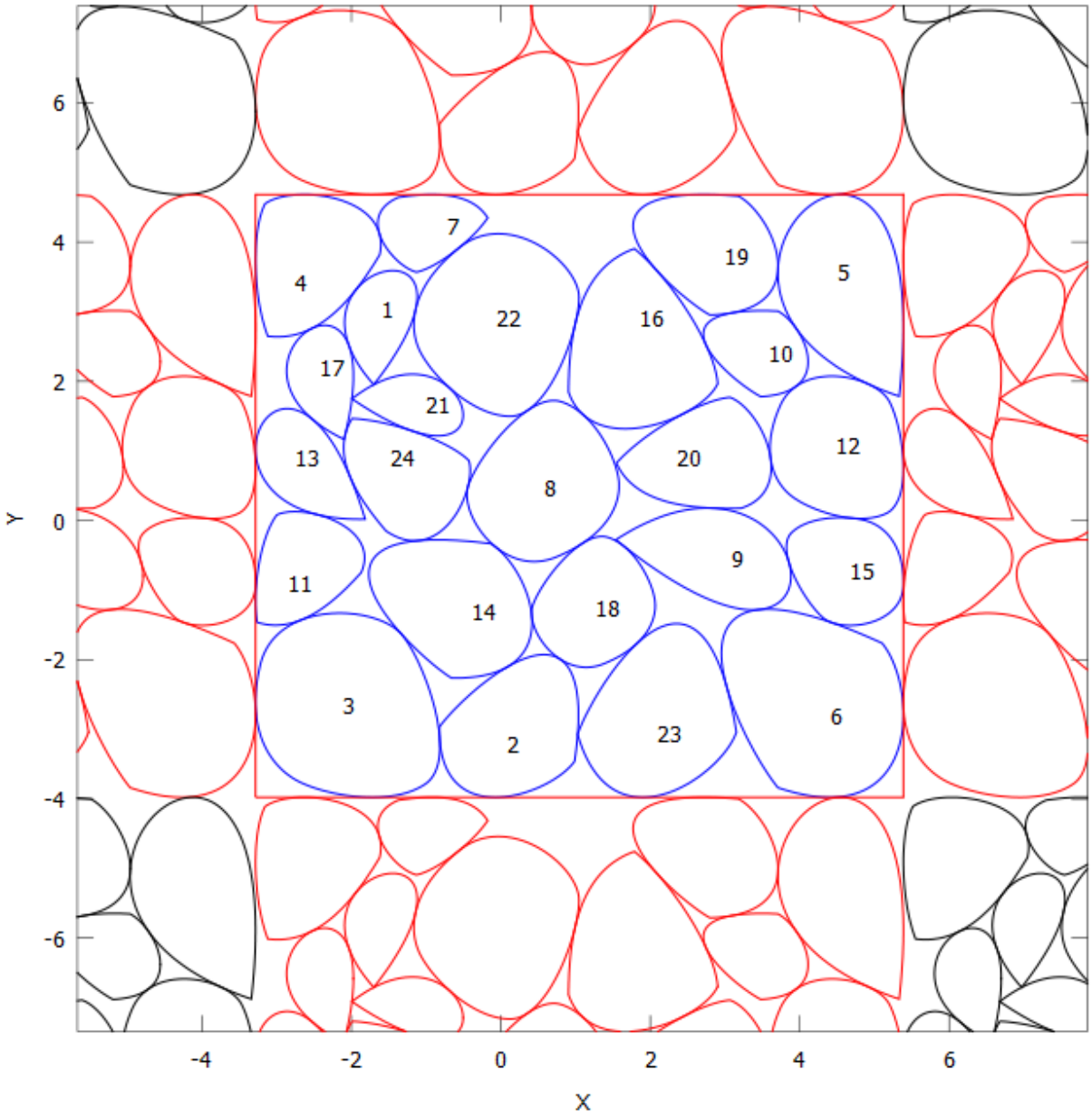

Figure 56: Enlarged view of the Stage-1 packing of 24 poly-Bézier shapes shown in *Figure 54*.

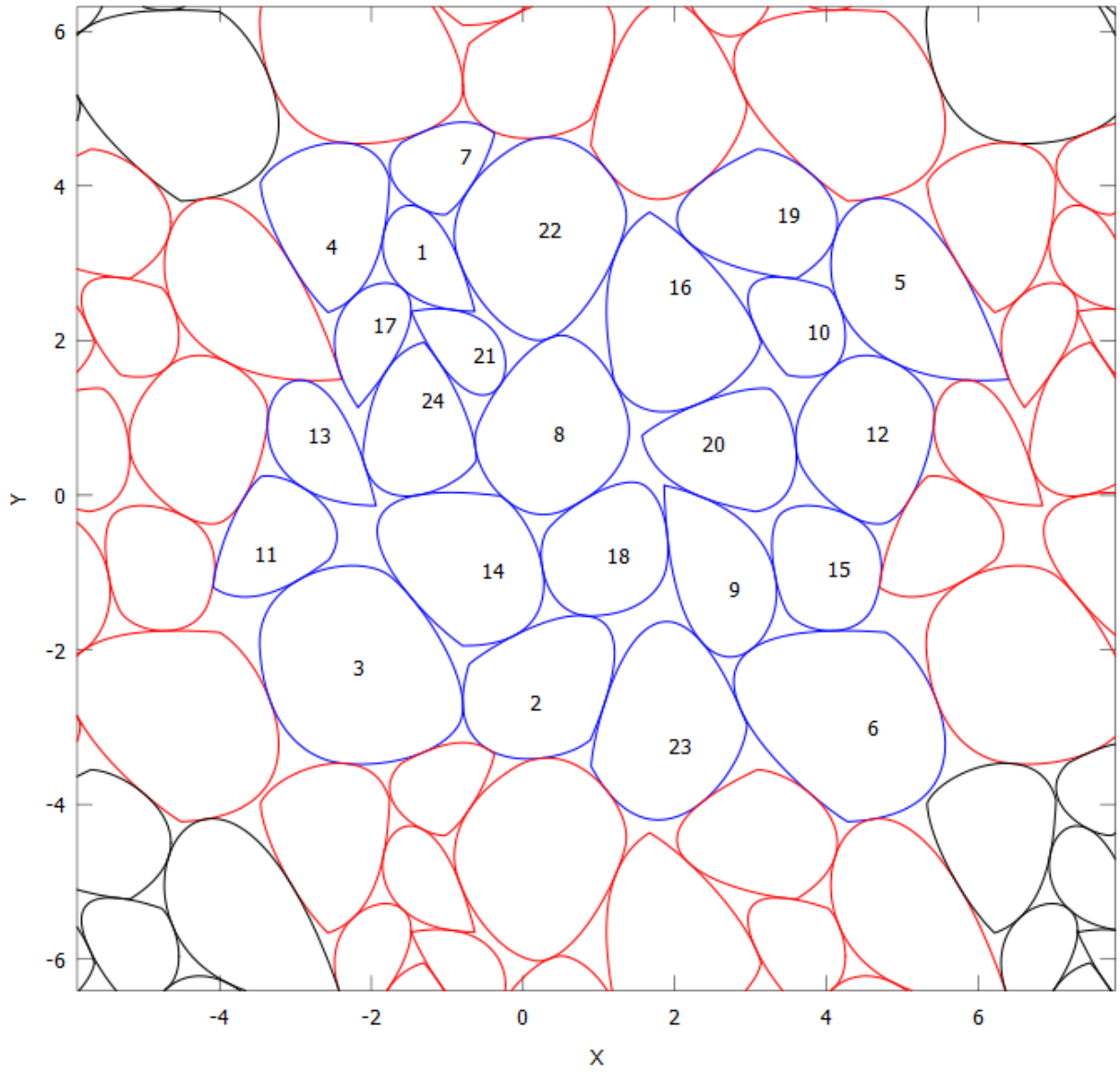

Figure 57: Enlarged view of the Stage-2 packing of 24 poly-Bézier shapes shown in *Figure 55*.

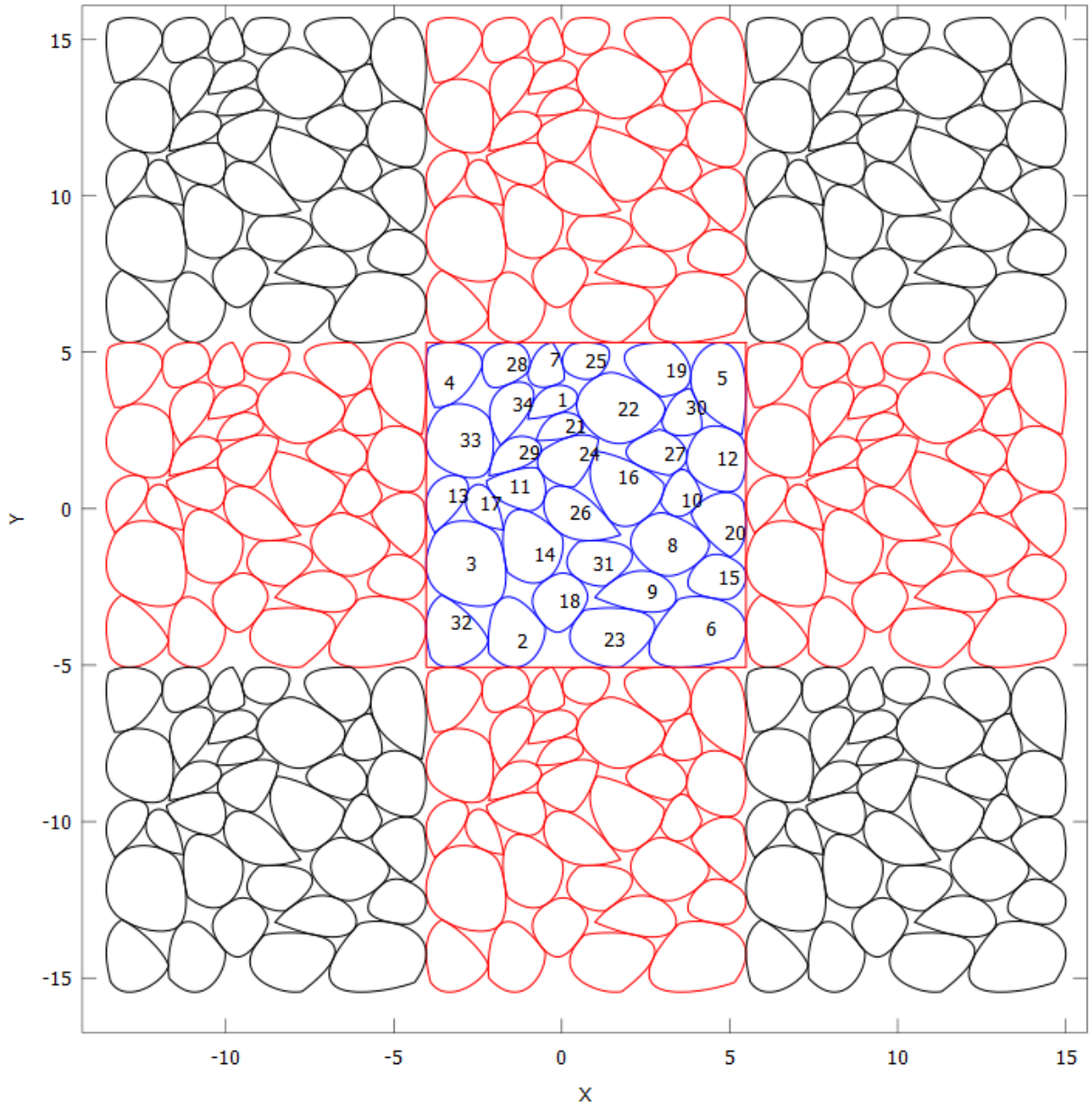

Figure 58: Stage-1 two-dimensional puzzle-piece packing of 34 poly-Bézier shapes surrounded by its eight nearest neighbors.

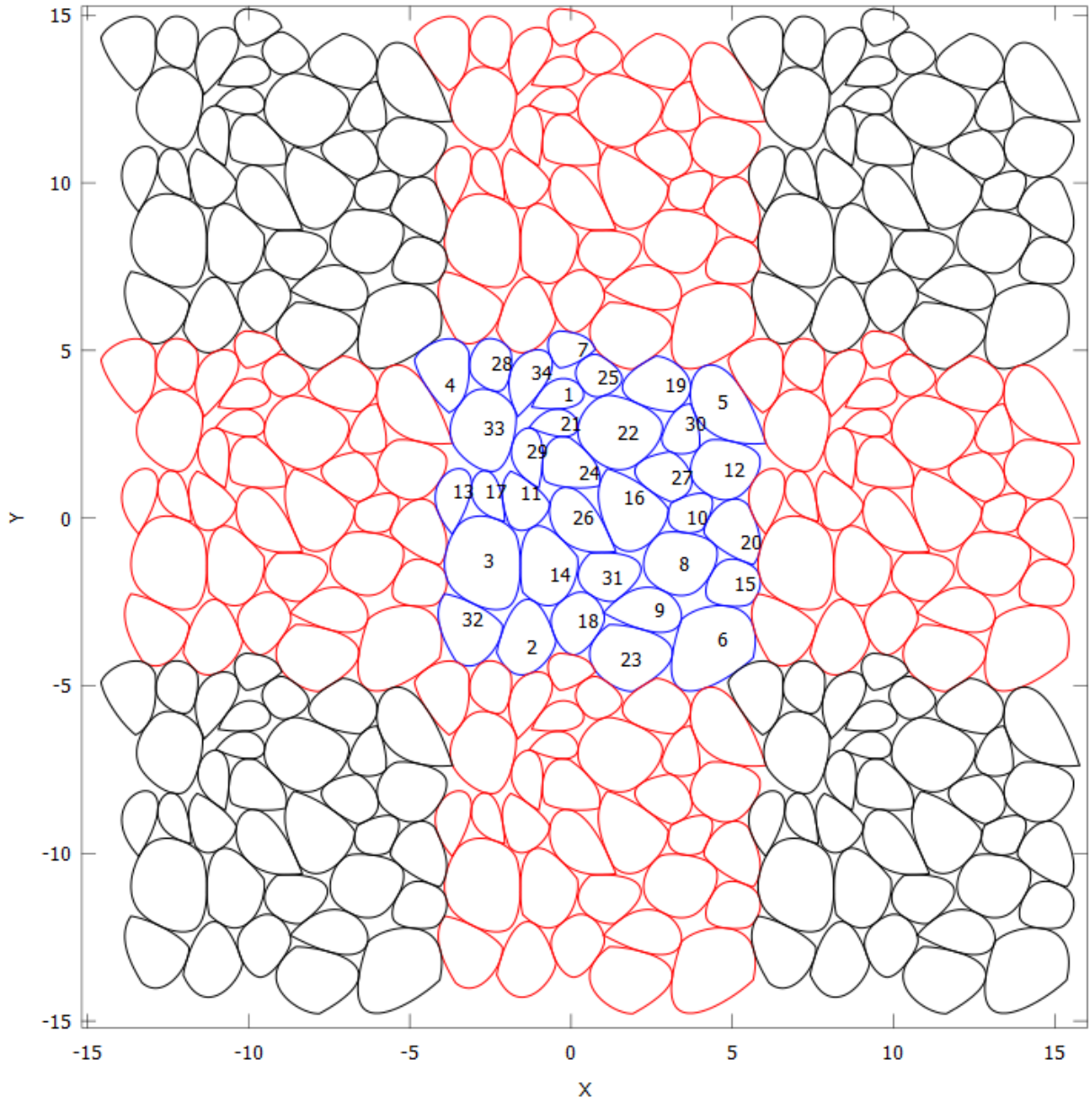

Figure 59: Stage-2 two-dimensional puzzle-piece packing of 34 poly-Bézier shapes surrounded by its eight nearest neighbors.

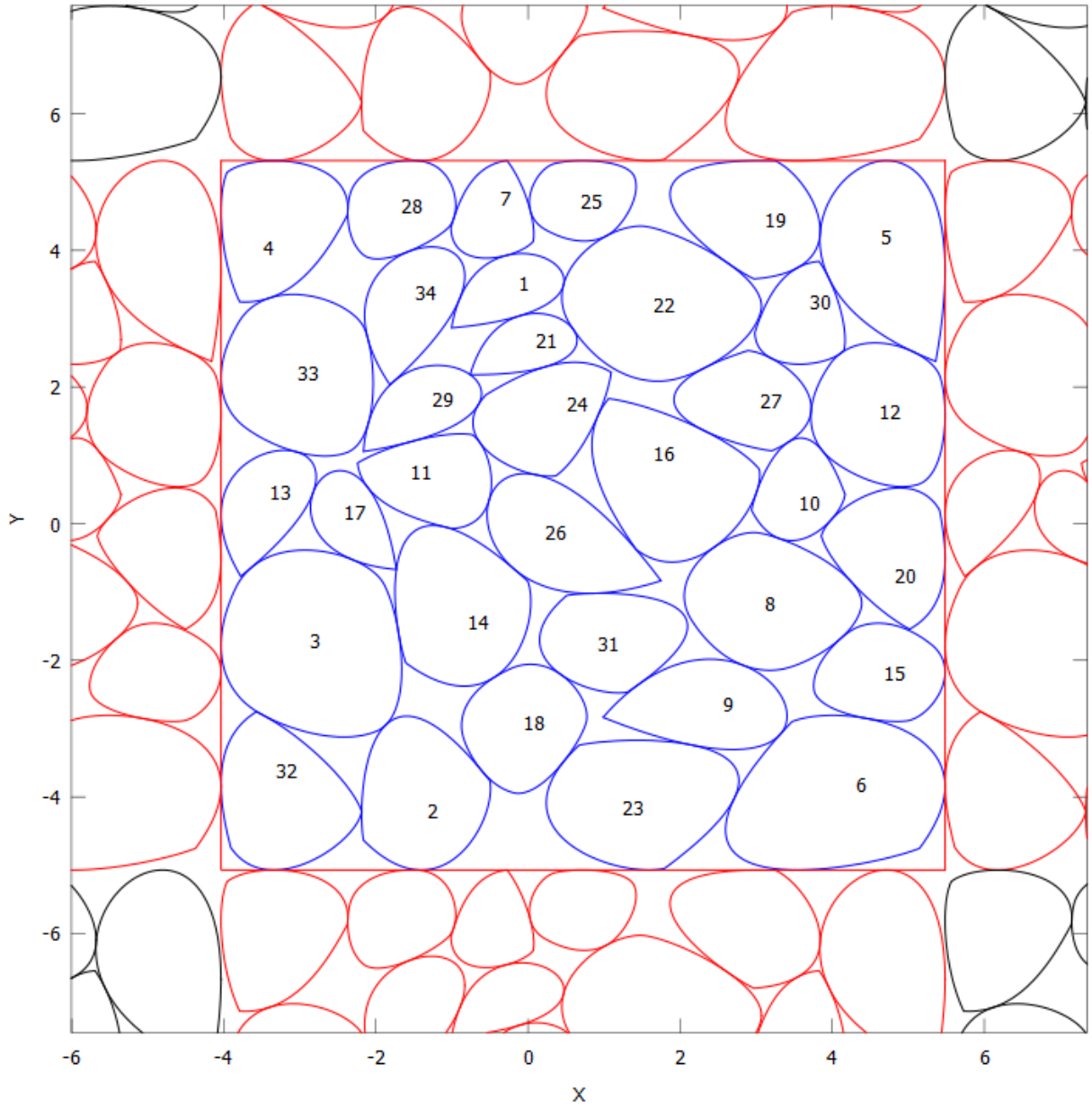

Figure 60: Enlarged view of the Stage-1 packing of 34 poly-Bézier shapes shown in *Figure 58*.

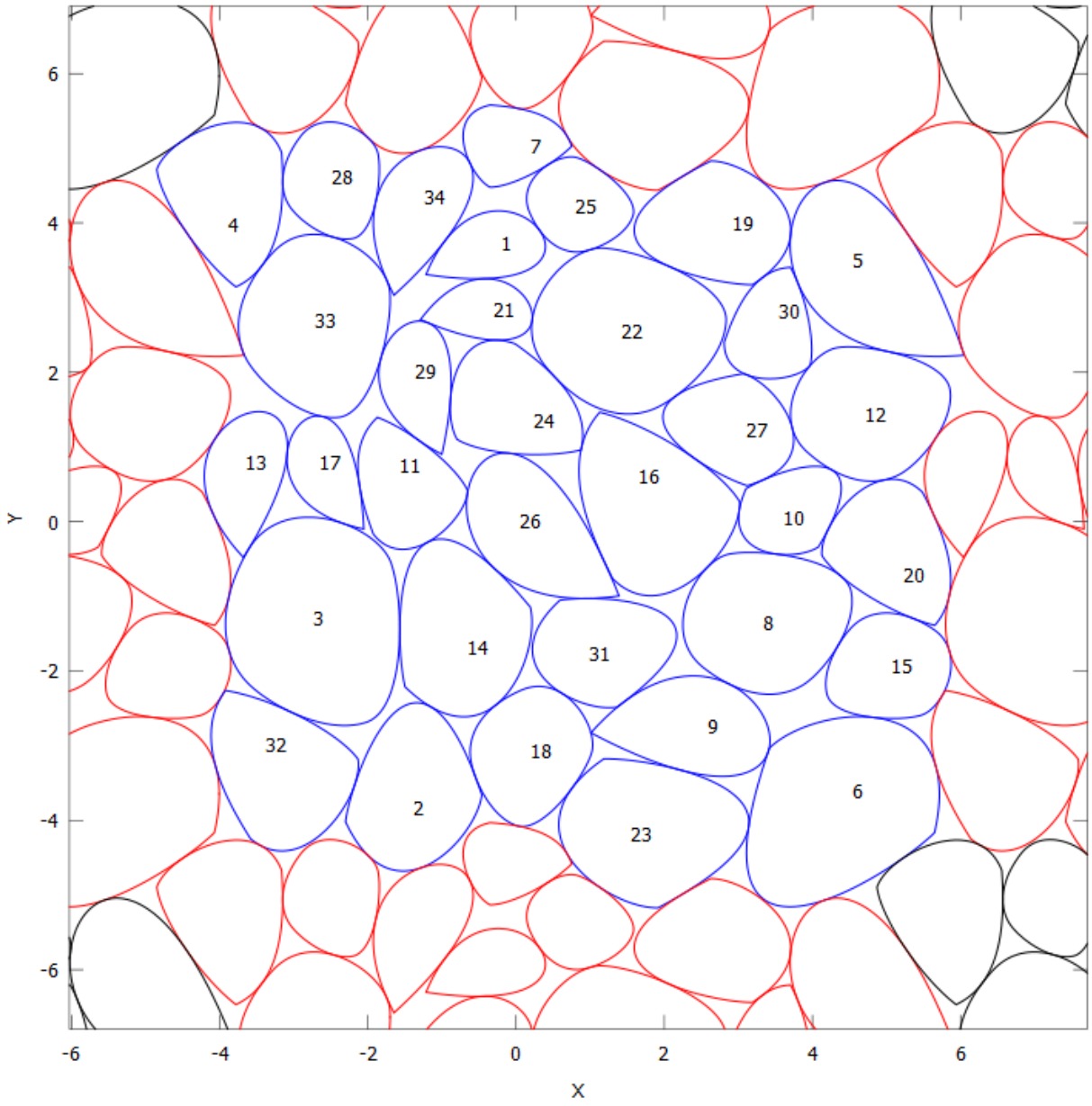

Figure 61: Enlarged view of the Stage-2 packing of 34 poly-Bézier shapes shown in *Figure 59*.

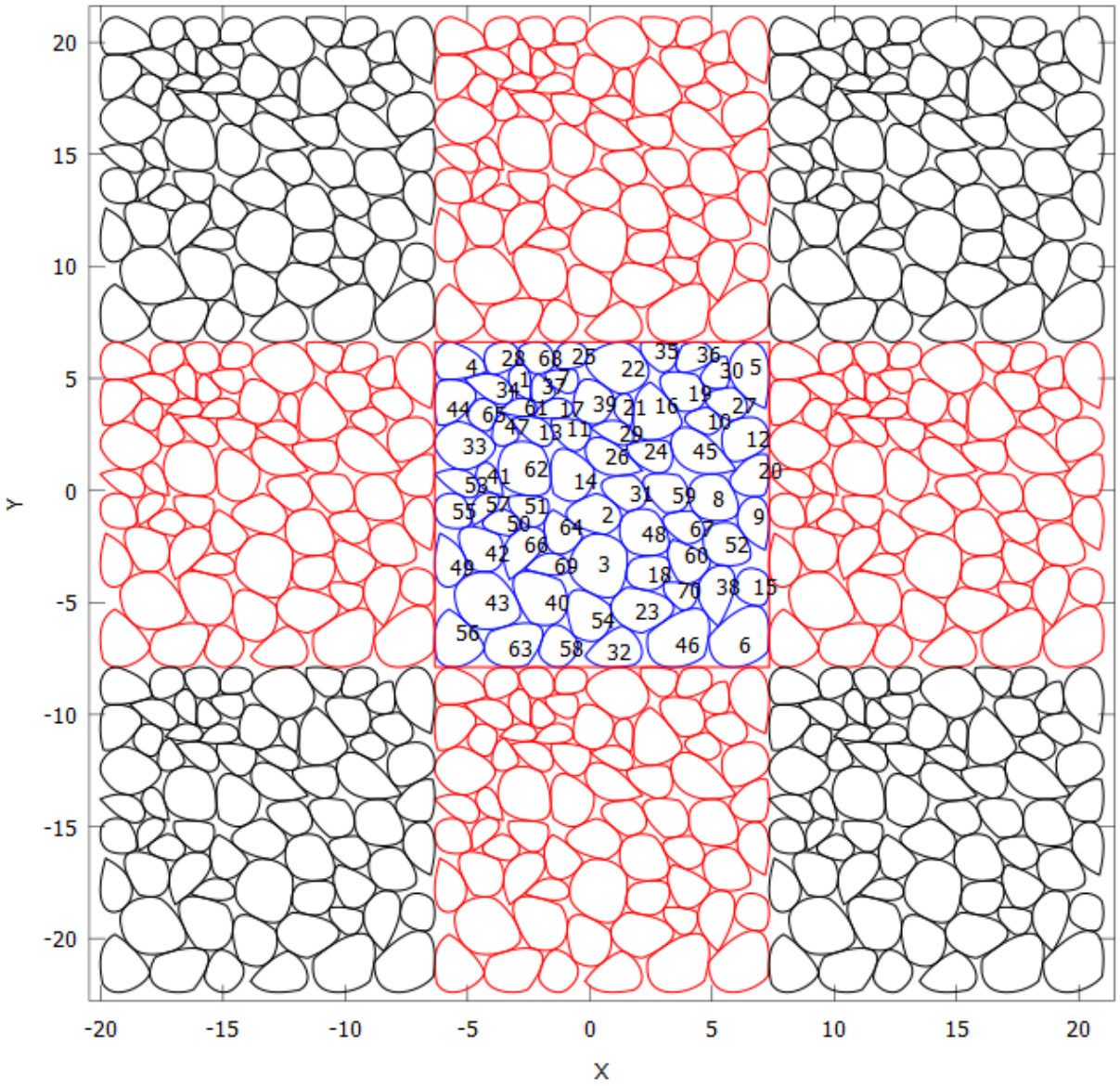

Figure 62: Stage-1 two-dimensional puzzle-piece packing of 70 poly-Bézier shapes surrounded by its eight nearest neighbors.

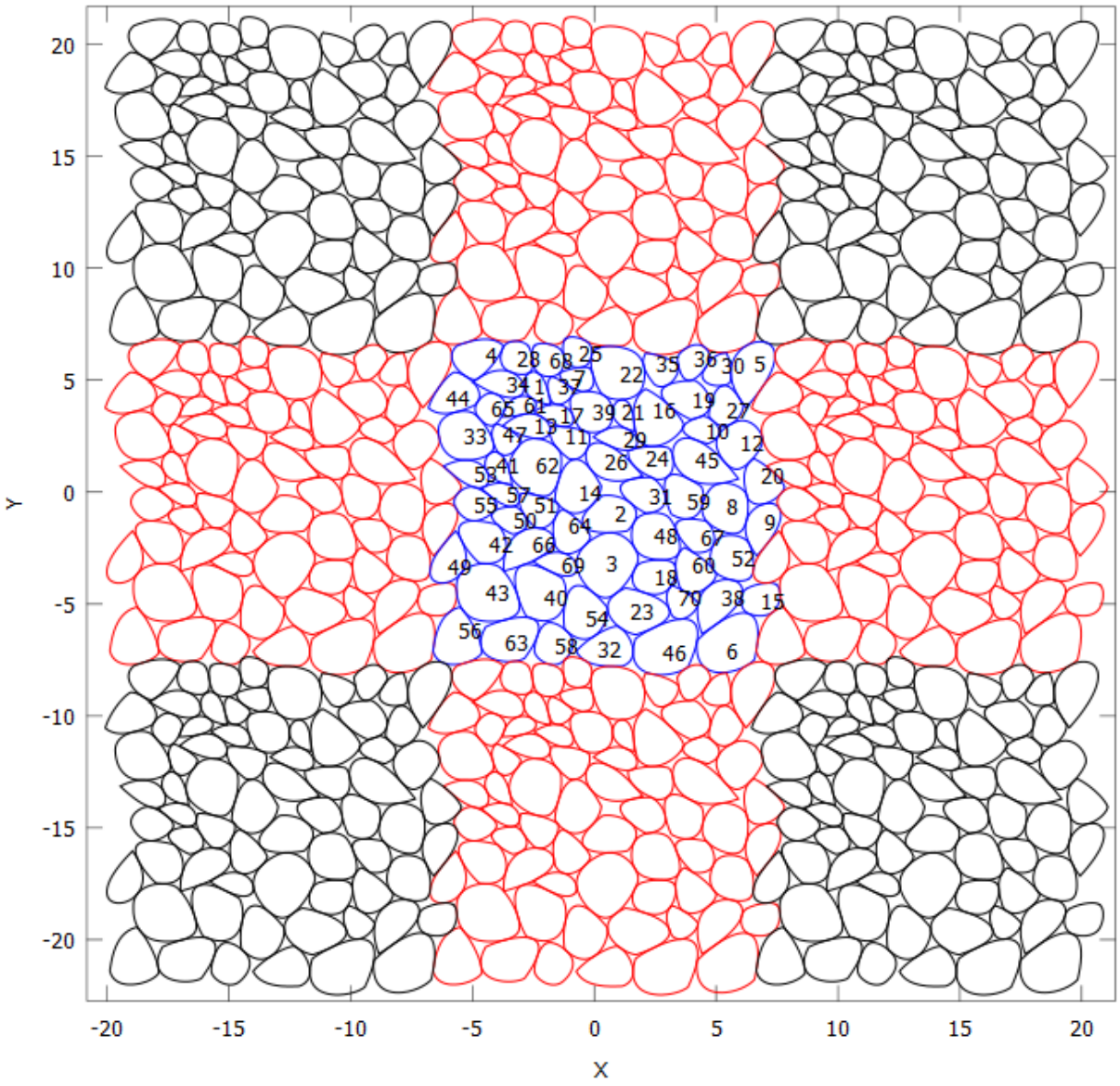

Figure 63: Stage-2 two-dimensional puzzle-piece packing of 70 poly-Bézier shapes surrounded by its eight nearest neighbors.

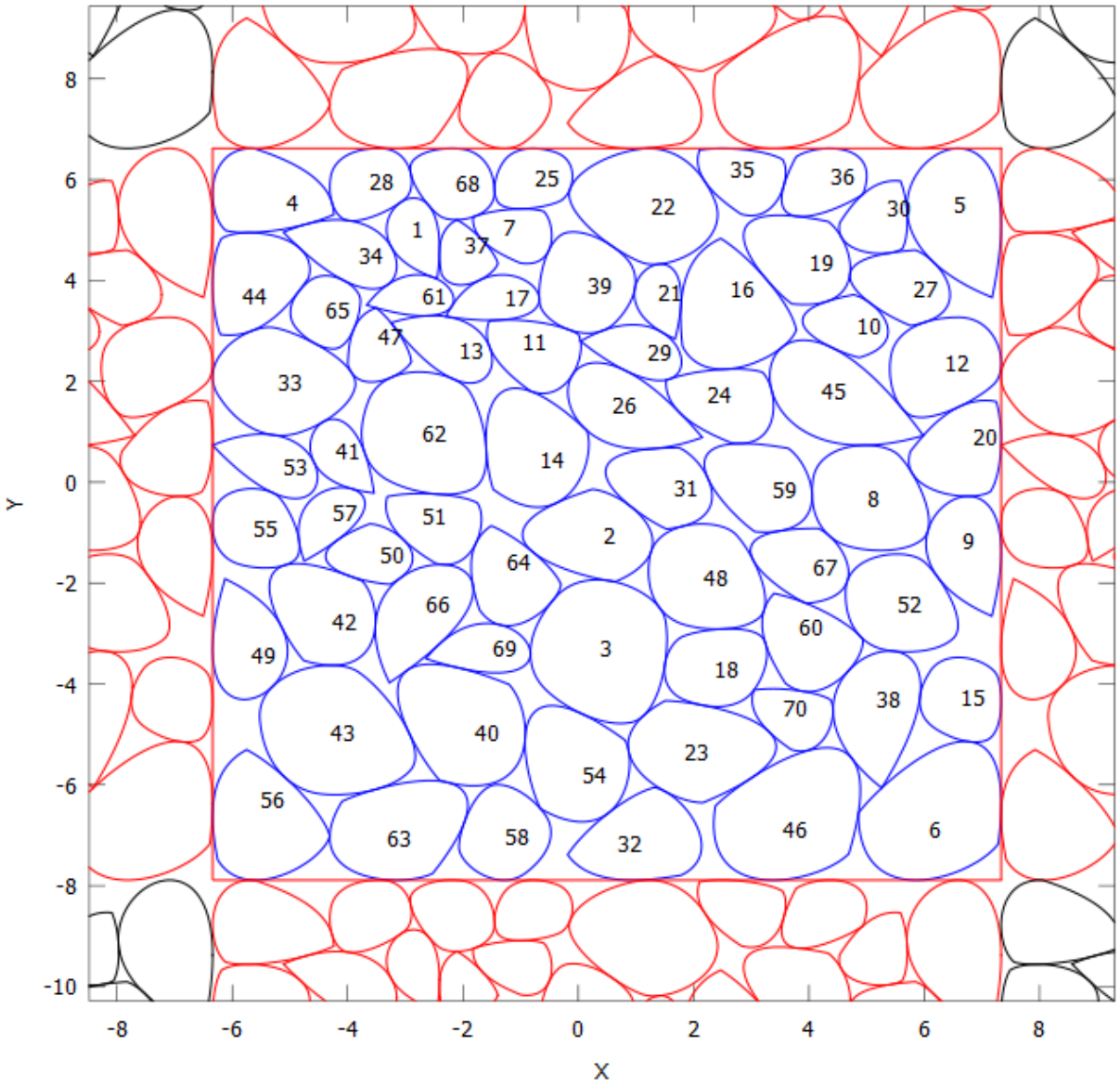

Figure 64: Enlarged view of the Stage-1 packing of 70 poly-Bézier shapes shown in *Figure 62*.

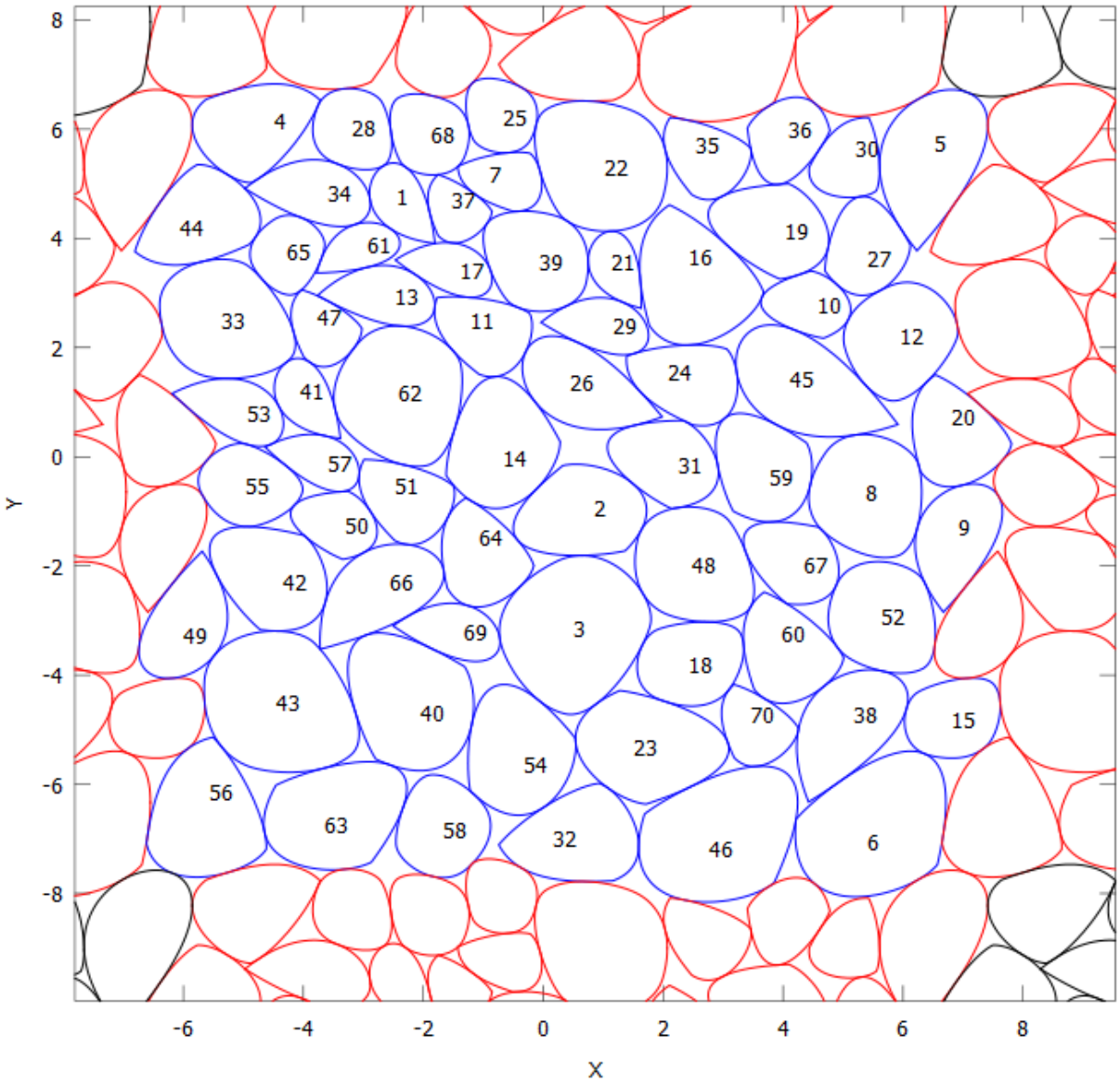

Figure 65: Enlarged view of the Stage-2 packing of 70 poly-Bézier shapes shown in *Figure 63*.

8 Conclusion

In this paper we showed how to model nonoverlap as well as uniform and nonuniform boundary-distance constraints between poly-Bézier shapes using an analytical computational-geometry library. Specifically, we presented a simple and powerful framework for modeling and implementing both nonoverlap and boundary-distance constraints. Among other things, we used this framework, to develop and implement the first know solution for packing problems involving nonuniform boundary-distance constraints.

Using these capabilities, in combination with off-the-shelf open-source nonlinear-optimization software, we developed and implemented efficient and reliable analytical-optimization solutions to the minimum-area rectangular-boundary packing-problem as well as minimum-area one- and two-dimensional puzzle-piece packing-problems. Moreover, the development and implementation of all of these solutions required only a modest amount of time and effort and required no significant expertise in nonlinear optimization, mathematical modeling, algorithm development, or software engineering.

In addition, to the best of our knowledge, these are also the first analytical-optimization solutions for packing problems involving poly-Bézier shapes. Since Bézier curves are a widely used and well understood means for efficiently capturing the geometry of a wide range of curves, poly-Bézier shapes provide a compact and easy to use representation of exact, or close approximations, for virtually any convex shape. As a consequence, the ability to pack poly-Bézier shapes represents a dramatic improvement over the current state of the art which is limited to solving packing problems on a small subset of analytic shapes by a small group of domain experts.

References

[1] J. A. Bennell and J. F. Oliveira, "The geometry of nesting problems: A tutorial," *European Journal of Operationnal Research,* vol. 184(2), pp. 397-415, 2008.

[2] COIN-OR, "IPOPT," [Online]. Available: https://github.com/coin-or/Ipopt.

[3] Artelys, "Knitro," [Online]. Available: https://www.artelys.com/solvers/knitro/.

[4] nag, "Mathematical Optimization Software," [Online]. Available: https://www.nag.com/content/mathematical-optimization-software.

[5] PERFORCE, "IMSL Numerical Libraries," [Online]. Available: https://www.imsl.com/.

[6] Y. Stoyan, J. Terno, G. Scheithauer, N. Gil and T. Romanova, "Φ-functions for primary 2D-objects," *Studia Informatica Universalis,* vol. 2, no. 1, pp. 1-32, 2001.

[7] Y. Stoyan, G. Scheithauer, N. Gil and T. Romanova, "Φ-functions for complex 2D-objects," *4OR: Quarterly Journal of the Belgian, French and Italian Operations Research Societies,* vol. 2, pp. 69-84, 2004.

[8] J. Bennell, G. Scheithauer, Y. Stoyan and T. Romanova, "Tools of mathematical modeling of arbitrary object packing problems," *Annals of Operations Research,* vol. 179, pp. 343-368, 2010.

[9] N. Chernov, Y. Stoyan and T. Romanova, "Mathematical model and efficient algorithms for object packing problem," *Computational Geometry,* vol. 43, no. 5, pp. 535-553, 2010.

[10] N. Chernov, Y. Stoyan, T. Romanova and A. Pankratov, "Phi-Functions for 2D Objects Formed by Line Segments and Circular Arcs," *Advances in Operations Research,* 2012.

[11] Y. Stoyan, A. Pankratov and T. romanova, "Quasi phi-Functions and Optimal Packing of Ellipses.," *Journal of Global Optimization,* vol. 65, no. 2, pp. 283-307, 2016.

[12] F. Kampas, I. Castillo and J. Pinter, "Optimized ellipse packings in regular polygons," *Optimization Letters,* vol. 13, pp. 1583-1613, 2019.

[13] A. Pankratov, T. Romanova and I. Litvinchev, "Packing ellipses in an optimized rectangular container," *Wireless Networks,* vol. 26, pp. 4869-4879, 2020.

[14] T. Romanova, Y. Stoyan, A. Pankratov, I. Litvinchev, K. Avramov, M. Chernobryvko, I. Yanchevskyi, I. Mozgava and J. Bennell, "Optimal layout of ellipses and its application for additive manufacturing," *International Journal of Production Research,* vol. 59, no. 2, pp. 560-575, 2021.

[15] F. J. Kampas, J. D. Pinter and I. Castillo, "Packing Ovals In Optimized Regular Polygons," *Journal of gloabl optimization,* vol. 77, pp. 175-196, 2020.

[16] MonarchIP, "ACGL," [Online]. Available: https://www.monarchip.com/.

[17] P. B. Morton, The Analytical Computational Geometry Library: A Computation Geometry Library for Scientists and Engineers, Create Space, 2018.

[18] M. de Berg, O. Cheong, M. van Kreveld and M. Overmars, Computational Geometry: Algorithms and Applications, Springer-Verlag, 2008.

[19] M. E. Mortenson, Geometric Modeling, Industrial Press, 2006.

[20] J. D. Foley, S. K. Feiner, J. F. Hughes and A. van Dam, Computer Graphics: Principles and Practice, Addison-Wesley, 1996.

[21] P. E. Bézier, "Example of an existing system in the motor industry: the Unisurf system," *Proceedings of the Royal Society of London,* vol. 321, no. 1545, pp. 207-218, 1971.

[22] M. Kamermans, "A Primer on Bézier Curves," [Online]. Available: https://pomax.github.io/bezierinfo/.

[23] A. Wächter and L. T. Biegler, "On the implementation of a primal-dual interior point filter line search alrogirthm for large-scale nonlinear programming," *Mathematical Programming,* vol. 106(1), pp. 25-57, 2006.

[24] INTERNATIONAL CUTTING DIE, "Understanding Rotary Die Cutting," [Online]. Available: https://www.icd-inc.com/2019/05/09/understanding-rotary-die-cutting/.

[25] Thomas, "Understanding Rotary Die Cutting," [Online]. Available: https://www.thomasnet.com/articles/custom-manufacturing-fabricating/rotary-die-cutting/.