An Evolutionary Approach to the Extraction of Object Construction Trees from 3D Point Clouds

Pierre-Alain Fayolle
The University of Aizu
fayolle@u-aizu.ac.jp

Alexander Pasko
Bournemouth University
apasko@bournemouth.ac.uk

Abstract

In order to extract a construction tree from a finite set of points sampled on the surface of an object, we present an evolutionary algorithm that evolves set-theoretic expressions made of primitives fitted to the input point-set and modeling operations. To keep relatively simple trees, we use a penalty term in the objective function optimized by the evolutionary algorithm. We show with experiments successes but also limitations of this approach.

Keywords: Shape Modeling, Solid Modeling, Genetic Programming, Segmentation, Fitting, Reverse Engineering, Construction tree, Function Representation

1 Introduction

Reverse engineering can be considered as a process of reconstruction from scanned point clouds of geometric models suitable for further re-use and modifications. These modifications can be performed on different levels. The lowest level allows for tweaking of the polygonal mesh vertices or control points of parametric surfaces. To support a higher level of interaction with reconstructed objects, obtained models have to be parameterized such that the user could modify individual parts and the entire logic of the object construction including its topology. Such a parameterized model reconstruction is required in mechanical engineering, bio-engineering, computer animation and other application areas [13].

Parametric solid models usually consist of geometric features composed using surface patches and a design history tree (or a feature tree) that represents the design intent and the sequence of operations for making a particular model. It is expected that a modification of any parameter has to be propagated through the entire model and it is automatically rebuilt accordingly. The feature recognition is a reverse engineering approach to generating feature-based parametric solids with the feature tree defining the model rebuilding procedure [57]. In practice, the rebuilding procedure using the feature tree can fail or result in an invalid object for some parameter modifications, which is a known problem of parametric solid modeling.

An alternative approach to creating parameterized models is constructive solid modeling based on the Constructive Solid Geometry (CSG) [40] or the Function Representation (FRep) [37]. A constructive model is represented by a binary (CSG) or n-ary (FRep) construction tree structure with primitive solids at the leaves and operations at the internal nodes of the tree. For any given point in space, an evaluation procedure traverses the tree and evaluates a binary point membership predicate (CSG) or a value of a real function (FRep) at this point. Such a model can be fully parameterized and any changes to the parameters are taken into account during the next evaluation inquiry without failure. Although an empty set can be generated as a result of parameter modifications, it is still a valid model.

We are interested in a reverse engineering procedure that can extract a construction FRep tree from a given input point cloud. The point-cloud is first segmented and primitives from a set of templates are fitted to each subset. A tree structure involving the fitted primitives and modeling operations is then discovered. The tree extraction is a complex optimization problem. We employ an evolutionary approach to solve it. Although evolutionary techniques have been already used for data fitting, geometric shapes generation and optimization [39], extraction of a construction tree is a new application area for genetic algorithms and genetic programming.

Contributions In this work we present an algorithm to:

- segment an input point-set in several subsets and fit primitives to each of them;
- combine these fitted primitives by modeling operations in order to obtain a constructive model for the input point-set;
- limit the size of the evolved construction tree to allow its reusability.

2 Related works

The work presented here is at the intersection of different research areas: surface reconstruction from discrete point-sets, reverse engineering of solid models and evolutionary approaches for the synthesis of shapes.

Surface reconstruction from discrete point-sets The problem of reconstructing a surface from sample points is extensively discussed in the computer graphics and geometric modeling literature. Recent popular methods for surface reconstruction often involve fitting implicit surfaces. The reconstruction method proposed by Muraki [35] consists in fitting blobby models to range data. In [25], Hoppe et al. reconstruct a surface by computing a signed distance function. Savchenko et al. [43] and Turk et al. [52] proposed to fit a linear combination of radial basis functions to the input point-set. Compactly supported radial basis functions were introduced by Morse et al. [34] to decrease the complexity in time and memory of the previous methods. In [36], Ohtake et al. glue locally fitted quadrics by applying a partition of unity. In [26, 27] the authors recover a characteristic function defining a solid by solving a Poisson equation. Different variations of the Moving Least Squares method were used in [15, 28, 48]. In a recent paper [12], the authors compute a smoothed distance function to the surface underlying the point-set by minimizing some energy function. For a deeper survey of this topic, the reader is referred to the work of Berger et al. [6]. These methods tend to produce verbose models, containing for example a large number of coefficients when splines are used in addition to the original point-set. These models lack semantic information and can not be used for inspection or reuse of the structure of the object in contrary to objects built using constructive solid modeling methods.

Reverse engineering The goal of reverse engineering is the creation of accurate and consistent geometric models from various types of input data including three-dimensional point-clouds [56]. The reconstructed model should be a valid solid model, ready to undergo further operations in some interactive modeling system. The problems to be solved include: identifying sharp edges and creases, segmentation, fitting of patches to subsets, treatment of blends and providing continuity and smoothness between the patches (see section 2 in [4]). Another important problem is the creation of geometric models respecting constraints [3] (e.g. parallelism of planes, concentricity of spheres and others). One of the first steps (similar to the approach presented here) is a segmentation of the input data. It consists in clustering the original point-set into subsets. After the segmentation, primitives are associated and fitted to each subset. The final step is the creation of the solid model. It consists in grouping the fitted primitives to generate a valid boundary representation model [3].

Parametrized models reflecting semantics and logic of their construction are most suitable for further operating on them [13]. Feature-based parametric solid models include information on semantically meaningful parts (geometric features) with their parameters and a history tree (or a feature tree) representing the sequence of operations for constructing the model. Reverse engineering history of modeling operations from a sequence of 3D objects is discussed in [16]. Given as input a series of 3D objects, each object is segmented and correspondences are estimated between consecutive objects. This information is then used to establish the operations between consecutive objects. Reverse engineering of parametric feature-based solids has been addressed for mechanical features [51] and for more general user-defined features [57]. Wang et al. present several techniques for reconstructing features such as extrusion, sweep, blend and others [58]. A known problem with parametric solid modeling is that the rebuilding process using the feature tree can fail or result in an invalid object for some combination of parameter values.

Constructive solid modeling based on CSG [40] or FRep [37] provides fully parametrized valid models. Fitting preliminary modelled parameterized FRep templates to point clouds was presented in [18] as a first step to the automation of the reconstruction of constructive models. A reverse engineering procedure that can extract a construction FRep tree from the given point cloud is an open research problem addresed in this paper.

Segmentation A necessary step in most of the reverse engineering techniques is the segmentation of the input point-set. There are various techniques available depending on the domain of application, see e.g. the section 1.1 of [17] for a more comprehensive survey of existing methods. Common techniques used in reverse engineering are described in [54, 4, 5, 55] and references therein. After the segmentation, we need to fit patches to each cluster. This fitting step can either be done separately from the segmentation step as for example in [54, 32, 4, 53] or be done jointly with the segmentation as in [44, 1, 17].

Boundary representation to CSG conversion Related to the problem that we are trying to solve is the problem of boundary representation to CSG conversion. This problem was first investigated in the two-dimensional case for linear polygons in [42, 2]. Later, Shapiro extended the algorithm to handle curved polygons [46]. Similar algorithms were adapted to three-dimensional polyhedra [61], but they do not work for some types of polyhedra. For the three-dimensional case, the problem was solved for solids bounded by second degree

surfaces in [47, 9, 10, 11]. These algorithms may require some additional half-spaces not available from the surface faces information or from the segmentation.

Evolutionary approaches for shape synthesis and optimization Evolutionary methods have already been used for geometric shape generation and optimization as in [39]. Hamza and Saitou used a tree-based genetic algorithm to optimize the shape of CSG solids satisfying some given constraints in [23]. Weiss used genetic programming for the structural optimization of CAD specification trees in [59]. Analysis of facade is carried out in [63] by finding a hierarchical decomposition that maximizes some measure of symmetry. A genetic algorithm is used as an heuristic for finding the decomposition optimizing this measure. In reverse engineering, an evolutionary search for numerical parameter values was applied in [22]. A first attempt to recover construction trees from point-cloud data was discussed in [49]. The authors used strongly typed genetic programming; leaves consist in primitives parameters and internal nodes are either algebraic operations, set-operations or primitives selected from a set of candidates. Parsimony is used to control the tree size. In spite of this, the size of the generated trees is still large, making the application limited to simple data-sets. Another problem is that it tries to solve several independent tasks at the same time by genetic programming: segmentation, fitting and construction tree extraction. This makes the approach probably unsuitable for complex objects. Given a list of fitted primitives and an input point-set, the authors of [19] used a genetic algorithm to evolve a linear tree (encoded in an array) with set-operations in the nodes and the given primitives in the leaves. While complicated objects can not be represented by a linear tree in general, it is possible to iterate the algorithm to build left heavy trees. The problem is the repetition of primitives not used in the solid description.

3 Background

Implicit surfaces, Function Representation An implicit surface is a surface representing points with a constant value: $\{(x, y, z) \in R^3 : f(x, y, z) = 0\}$ of some trivariate function f (see [8] and references therein). For example, a unit sphere can be represented by the set of points satisfying: $1.0 - \sqrt{x^2 + y^2 + z^2} = 0$. The Function Representation (FRep) (see [37]) considers not only the surface but also the bounded interior given by: $\{(x, y, z) \in R^3 : f(x, y, z) > 0\}$. Complex objects can be modeled either by numerical techniques (such as fitting polynomials or splines) or by applying modeling operations to simpler objects. The set-operations (union, intersection, subtraction, complement) can be implemented by:

- Min/max [41]:

$$intersection(S_1, S_2) := min(f_{S_1}, f_{S_2}) \tag{1}$$
$$union(S_1, S_2) := max(f_{S_1}, f_{S_2}) \tag{2}$$
$$complement(S) := -f_S \tag{3}$$
$$subtraction(S_1, S_2) := intersection(S_1, complement(S_2)) \tag{4}$$

  where S, S_1 and S_2 are solid objects and f_S, f_{S_1} and f_{S_2} their corresponding functions.

- R-functions [45, 37]:

$$intersection(S_1, S_2) := f_{S_1} + f_{S_2} - \sqrt{f_{S_1}^2 + f_{S_2}^2} \tag{5}$$
$$union(S_1, S_2) := f_{S_1} + f_{S_2} + \sqrt{f_{S_1}^2 + f_{S_2}^2} \tag{6}$$

  complement() is defined as in eq. (3). And *subtraction*() is defined as in eq. (4) in terms of *intersection*() (eq. (5)) and *complement*() (eq. (3)).

This allows for a constructive representation of objects by a set-theoretic expression or equivalently by a trivariate function. Modifications of eq. (5) and (6) allow for modeling blends (smooth transitions between two surfaces - see eq. (9) and Fig. 15). Other available operations include all rigid body transformations as well as non-linear deformations.

4 Overview

The input of the proposed algorithm is a finite point-set S given by coordinates (x, y, z) of points sampled on the object surface by some scanning device and vectors (n_x, n_y, n_z) normal to the surface at each point (x, y, z) (see Fig. 1, top row, left). If only the point coordinates are available, the normal vector field can be estimated

by the Principal Component Analysis with propagation of a selected orientation (see the algorithms in sec. 3.2 and 3.3 of [25] and in [33]).

Given the finite point-set S of points with normals, the goal is to discover an expression for $f(x, y, z)$ involving primitives (such as sphere, cylinder, torus and others) and geometric operations (such as union, intersection, subtraction and others). It is assumed that the zero level-set of the function $f(x, y, z)$ defines an approximation of the surface from which the input points in S are sampled (see for example the top row, right picture in Fig. 1). The expression f can also equivalently be seen as a tree with the corresponding geometric operations in its internal nodes and instances of primitives in its leaves (see as an example the tree in Fig. 1, bottom row).

The main steps of the approach are given in Algorithm 1. The first step is used to determine a list of fitted primitives. For this purpose, the input point-set is first segmented into subsets and for each of these subsets a primitive most likely describing this subset is determined and its parameters are fitted (see the middle picture in the top row of Fig. 1). Details for this step are given in section 5.1. At the second step, a list of additional primitives (called separating primitives) needed for deriving the final expression is compiled and added to the list of already fitted primitives (see section 5.2). At the final step, given the list of fitted primitives computed at the previous steps and the input point-set, we use genetic programming [29] to evolve a construction tree with the fitted primitives in the leaves and geometric operations in the internal nodes (see Fig. 1, bottom row). The details of this step are given in section 5.3.

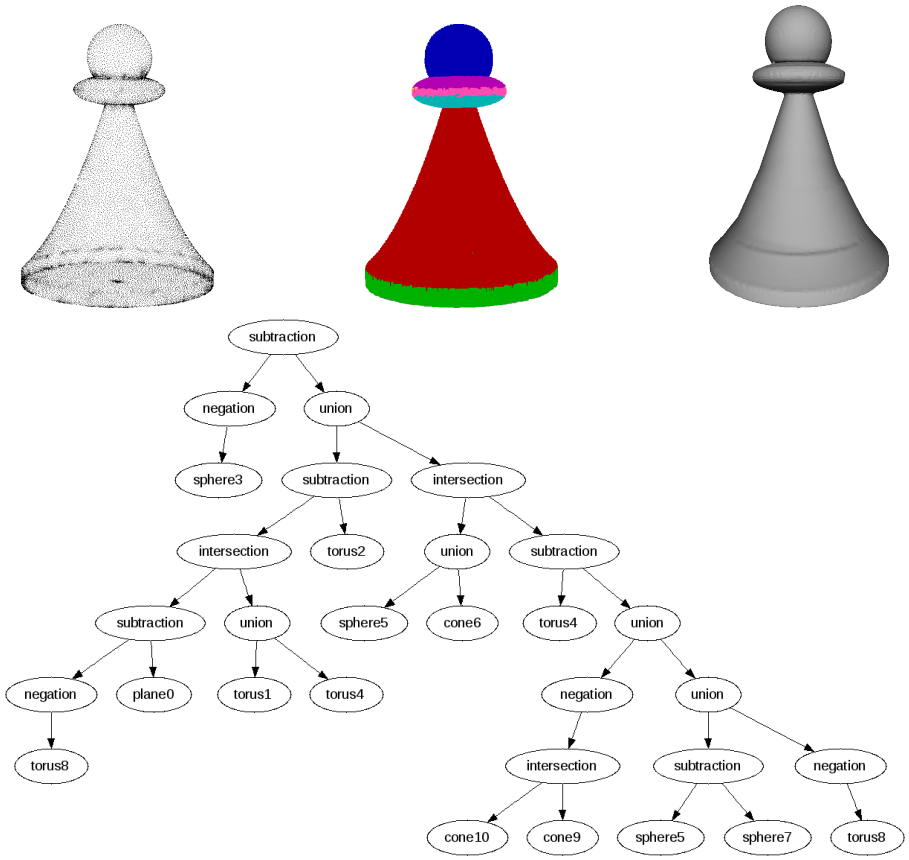

Figure 1: A graphical illustration of the tree extraction algorithm: 1) The input consists of a set of points sampled on the surface of an object (top row, left); 2) The input point-set is segmented into subsets (or clusters) and a primitive is fitted to each subset (top row, middle); 3) Visualization of the corresponding object (top row, right); 4) Genetic programming is used to evolve a construction tree with the fitted primitives as leaves and operations as internal nodes (bottom row)

Algorithm 1 Construction tree extraction from a finite point-set

Require: Finite point-set S and list of candidate primitives

1: Segmentation of S into clusters and fitting of a primitive to each cluster
2: Computation of separating primitives
3: Tree evolution by genetic programming with the fitted and separating primitives in the leaves and geometric operations in the internal nodes

For the experiments described in section 6, the set operations (union, intersection, difference and comple-

ment) are used as geometric operations. Primitives are implemented with implicit surfaces and operations can be implemented with either min/max or R-functions (min/max were used in the experiments).

5 Algorithm

5.1 Segmentation and fitting

The first step consists in the segmentation of the input point-set into clusters and fitting of primitives to each of these clusters. During the segmentation of the input point-set, we also need to identify a primitive from a set of candidates, such as plane, sphere, cone and others, and fit its parameters to the points of a given cluster. We are not directly interested in the clusters, but in the list of fitted primitives describing or approximating each of these clusters. For this purpose, we can use the approach based on RANSAC [21] described in [44] that performs the segmentation and fitting at the same time. Given a finite point-set, the best fitted plane, sphere, cylinder, cone and torus to the point-set are computed using the RANSAC approach. Then the fitted primitive that best describes the data-set is selected and the corresponding points are removed from the point-set. In order to determine which of the fitted primitives best describes the data, the authors of [44] propose to count the number of points from the input point-set S that are on or near the surface of the fitted primitive (see section 4.4 in [44]). These two steps are then repeated until the number of points left in the point-set is below some given threshold.

An alternative approach for fitting primitives chosen from a list of candidate primitives is described in [17]. For each type of primitive, its parameters maximizing an objective function (see eq. (1) in [17]) are computed. The primitive best describing the data is then selected among all fitted primitives at this step (see section 5 in [17]) and the corresponding points are removed from the point-set. These two steps are repeated until the number of points left in the point-set is below some given threshold. The latter method has the advantage of allowing for a larger and extensible set of possible primitives (such as ellipsoid or super-ellipsoid for example) but is slower.

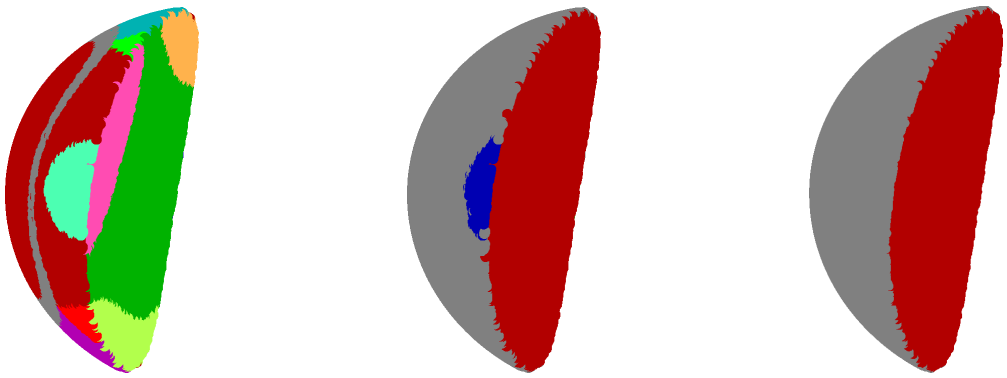

Figure 2: Segmentation/fitting obtained by the efficient RANSAC algorithm [44] (left and middle) and the segmentation algorithm of [17] (right).

Figure 2 illustrates the result of the segmentation/fitting by the efficient RANSAC algorithm [44] (left and middle pictures) and the fitting algorithm discussed in [17] (right picture) both applied to a set of points sampled on the surface of an object made of ellipsoids. The pictures on the left and middle were obtained with two different sets of parameters. The segmentation shown at the left is made of 14 subsets with corresponding primitives made of spheres and tori. The segmentation shown in the middle has 4 subsets made as well of spheres and tori. Compare these results with the segmentation shown at the right obtained with the algorithm from [17] that consists of two subsets each associated with an ellipsoid. This approximation of the surface by spheres and tori will result in some visible artifacts on the surface of the object obtained after the final step of Algorithm 1.

The output of the segmentation and fitting step is a list of fitted primitives. In the example of the synthetic cube data-set shown in Fig. 5, the result of the first step is the list of fitted primitives given in Table 1. While we did not implement it for our experiments, it is possible to improve the fitting results of the methods discussed in [44] or [17] by identifying relations between primitives (examples of such relations include parallelism of planes, coaxiality of cylinders, etc.) and re-fitting all primitives while enforcing these relations. This approach is discussed in the work [30].

5.2 Separating primitives

Primitives detected on the surface of an object are not always sufficient to describe the object with a constructive approach. It is sometimes necessary to introduce additional primitives that do not appear in the segmentation. These are called separating primitives or separators. The idea of introducing such primitives in order to describe an object by a CSG expression was introduced by Shapiro and Vossler in [47].

Let us illustrate the idea by an example. The object shown in Fig. 3 can not be described by a constructive expression using only set-theoretic operations and the primitives resulting from the segmentation step. The left-most image shows the segmented input point-set. The image in the middle shows the reconstructed object obtained by our algorithm without using separating primitives. Note that the smoothed surface joining the front and top planes is not properly retrieved even if it was correctly identified in the segmentation, where a cylinder was fitted to the corresponding point-set. The figure on the right shows the reconstructed object obtained by using two additional planar half-spaces (plane primitives). Note how the smooth transition is correctly reconstructed this time.

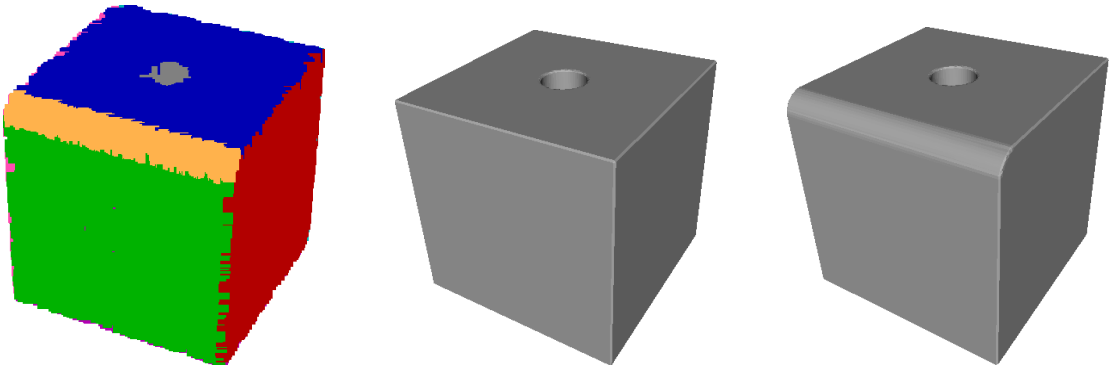

Figure 3: An example of using separating primitives. Left: segmented point-set. Middle: reconstructed object without using separating primitives. Right: reconstructed object using two additional separating planar half-spaces; the blend (smooth transition) is properly reconstructed.

We use Algorithm 2 in order to compute the list of potential separating primitives. We iterate through all the primitives identified during the segmentation step. If the primitive is not a plane, we retrieve the subset of points on or near this primitive's surface. Then we compute the oriented bounding box corresponding to this point-set and add all the planes of the bounding box to the list of primitives obtained from the segmentation. If the plane to be added is already present in the list, we discard it. While it is easier to compute the axis aligned bounding box instead of the oriented bounding box, the former will not work in some cases (for example, consider the object in Fig. 3 after some arbitrary rotations).

Algorithm 2 Compute separating primitives

Require: List of fitted primitives and segmented subsets from step 1
1: **for** each primitive P_i and its corresponding subset S_i **do**
2: **if** P_i is not a plane **then**
3: Compute the oriented bounding box b to S_i
4: **for** each face f of b **do**
5: Determine the plane supporting f
6: Append it to the list of primitives if it is not already in it
7: **end for**
8: **end if**
9: **end for**
10: Return the list of the original primitives with the additional separating primitives

The primary reason for adding these separating half-spaces is to clip some of the primitives fitted to the surface. The example in Fig. 3 illustrates their use in clipping the cylinder, the top and front planes such that the cylinder smoothly joins the two planar surfaces. However, using only separating planes seems to be limited to cases involving canal surfaces whose sphere centers lie on straight lines. In the other cases, limiting the separating primitives to planes can be insufficient. For example, consider the object obtained from two cylinders and a torus smoothly blending these two cylinders, as shown in the leftmost image in Fig. 4. In this particular case, using separating planes only with the approach described above produces the result illustrated in the middle image of Fig. 4. It is not possible to improve on this recovered object. One way to resolve the issue is to extend Algorithm 2 by appending to the list of separating primitives one cylinder for each identified

torus. The cylinder radius is given by the torus major radius. Using this extended list of separating primitives, we were able to generate with Algorithm 1 the object shown in Fig. 4, rightmost image.

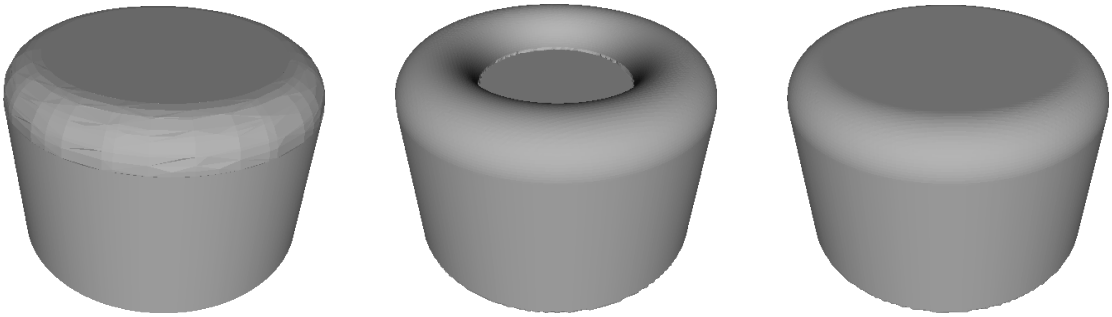

Figure 4: Left: a test object made of cylinders and a torus. Middle: the best recovered constructive model using separating planes only does not properly describe the original object. Right: the best recovered constructive model when separating planes and cylinders are used.

Finally, note that all the additional separating primitives do not need to appear in the final expression evolved by the genetic programming step. Only a subset (potentially empty) of them will be used in the final expression.

5.3 Construction tree extraction with genetic programming

Given the input point-set S and a list of fitted primitives obtained from the previous steps (see Algorithm 1), we want to discover an expression with the fitted primitives as terminal symbols and the geometric operations as function symbols, such that the surface of the geometric object defined by the zero-level set of the evolved expression interpolates or approximates the points from S. Primitives are implemented as implicit surfaces. We use the signed distance to the primitive's boundary if easily computable otherwise we compute an approximation as proposed by Taubin: a first order approximation of the distance to the zero level of f is given by $f/||\nabla f||$ [50]. Geometric operations can be implemented in terms of min/max or R-functions. The expression can be equivalently interpreted as a tree with the fitted primitives in the leaves and the geometric operations in the internal nodes.

The expression is evolved by genetic programming [29]. Specifically, we implemented Algorithm 3. A creature in the population corresponds to an expression with fitted primitives as terminal symbols and geometric operations as function symbols. The set of terminals (leaves in the tree corresponding to the expression) consists of all the fitted primitives obtained from the previous steps of Algorithm 1 (see also details in sub-sections 5.1 and 5.2). The set of functions (or internal nodes in tree representation) consists of geometric operations applied to the leaves and sub-trees. This set should at least contain the set-operations: union, intersection, subtraction and complement. In order to work with solids bounded by implicit surfaces, these set-operations can be implemented with R-functions or min/max. Other modeling operations are available such as, for example: blending [38], tapering, bending and others. However, they require additional arguments that would need to be computed by genetic programming, which would make the procedure and the search more complicated. We have not worked on this issue and it will be a topic of further investigations.

In Algorithm 3 the population is initialized with s random creatures. The generation of a random creature uses two parameters: the probability to create a subtree at the given node (set to 0.7 in the experiments) and the maximum depth for the tree (set to 10 in the experiments). A random tree is created by drawing a sample from a unit uniform distribution and comparing it with the probability to create a subtree. If the sample value is lower and the maximum depth has not been reached for the tree, an operation is selected at random (each operation has the same probability to be selected), and the procedure is recursively called to generate subtrees. Otherwise, a primitive is selected at random (each primitive has the same probability to be selected) and the procedure terminates.

In all the experiments described below, we use as a stopping criterion a maximum number of iterations of the outer loop. It is possible to use different criteria that uses, for example, the mean score of the population and the score of the best creature. The section within lines $8 - 15$ can easily be run in parallel by splitting the loop among several threads since each computation is independent.

Fitness function From one iteration to the next, we always keep the best n creatures (we used $n = 2$ in all our experiments). Ranking the creatures of the current population is done as detailed in Algorithm 4. Given a

Algorithm 3 Tree extraction

Require: Finite point-set S and list of fitted primitives
Let s be the population size
Let μ be the mutation rate
Let χ be the crossover rate
$p = InitializeRandomPopulation(s)$ {Create a random initial population}
while Stopping criterion is not satisfied **do**
  $p = Rank(p, S)$ {Evaluate and sort the curent population}
  $p' = [p(1), ..., p(n)]$ {Save the n best creatures in the next population}
  while $Length(p') <= s$ **do**
    Select two creatures c_1 and c_2 from p by tournament
    $c_1', c_2' = Crossover(\chi, c_1, c_2)$
    $c_1' = Mutate(\mu, c_1')$
    $c_2' = Mutate(\mu, c_2')$
    $p' = Append(p', c_1')$ {Append c_1' and c_2' to the new population}
    $p' = Append(p', c_2')$
  end while
  $p = p'$
end while

point-set S, we first compute a raw score for each creature c using the following objective function:

$$E(c, S) := \sum_{i=1}^{N} (exp(-d_i(c)^2) + exp(-\theta_i(c)^2)) - \lambda \; size(c) \tag{7}$$

where:

- $\mathbf{x}_i$ are the points from the input point-set S,
- N is the number of points in the point-set S,
- $d_i(c) = \frac{f(\mathbf{x}_i)}{\epsilon_d}$, with $f()$ is the expression corresponding to the creature c and ϵ_d a user defined parameter,
- $\theta_i(c) = \frac{ArcCos(-\frac{\nabla f(\mathbf{x}_i)}{||\nabla f(\mathbf{x}_i)||} \cdot \vec{n_i})}{\alpha}$, with ∇f the gradient of the expression corresponding to the creature c, $\vec{n_i}$ the normal vector to the surface at the point $\mathbf{x}_i$, and α a user defined parameter,
- $size(c)$ the function that counts the number of nodes (internal and leaves) in the tree corresponding to the creature c,
- λ a user defined parameter

The goal is to maximize E. This objective function smoothly penalizes models that disagree with the point-cloud data (in terms of being on or near the points and having a normal agreeing with the points' normal). With this objective function, for a given creature, a point at a distance greater than $\epsilon_d\sqrt{ln(4)}$ and at which the angle between the gradient and the point normal is greater than $\alpha\sqrt{ln(4)}$ would contribute less than $1/2$ to the objective value. The second term in (7) is also required to avoid trivial solutions (for example an expression evaluating to zero everywhere). An alternative approach is to voxelize the input data and count how many voxels are properly classified with the given expression, however it will also increase the computational time. This approach is used, for example, in [62]. We are using equal weights for the first two terms under summation in (7) as we did not see any particular reason to favor one term over the other. We performed some experiments with different weights but did not see any significant difference.

Penalization of large trees There are essentially two reasons for trying to maintain a limited size for the trees during the algorithm execution. The first one is related to the speed of execution: as trees become bigger, it takes longer to evaluate the corresponding functions, making the whole algorithm slower. The second reason, already mentioned above, is practical: for the trees to be easy to use in further applications (analysis, modeling, rendering, etc.), we need to limit the repetition of sub-trees and primitives that do not contribute additional information.

In order to prevent unnecessarily large trees, we add a penalty term proportional to the size of the tree ($\lambda \; size(c)$ in eq. (7)). For the coefficient λ, we used in our experiments $log(N)$, where N is the number of points in the input point-set. This term is important to obtain trees of limited size that can be further processed.

As an illustration, we have applied the algorithm with and without this term to the data-set illustrated in Fig. 5, which corresponds to points sampled on the surface of a cube. The optimal expression obtained when the term $\lambda\ size(c)$ is used contains only 5 internal nodes (corresponding to operations); the corresponding tree can be seen in Fig. 6. For comparison, the optimal tree obtained when running the algorithm without the term $\lambda\ size(c)$ contains 170 internal nodes, which is a lot considering that a cube can be defined using 6 planar half-spaces and 5 set-theoretic intersection operations.

Of course this penalty term does not guarantee an optimal size for the tree when Algorithm 3 is run a fixed number of iterations. But in practice, we found that it is effective (see the experimental results described in section 6). If the operations used in the tree are limited to set-theoretic operations, then it is possible to use standard algorithms for simplifying Boolean expressions as a final step of Algorithm 3. For example, it is possible to use the function $FullSimplify$ of Mathematica [60] to perform such a simplification. This approach was not used in the results presented in section 6.

For a creature that provides a very good approximation (or even an exact interpolation) of the input point-set, the sum in eq. (7) will be in the order of N. It is important to keep λ low relative to this number, otherwise simple creatures will be favored at the expense of their accuracy in describing the input data-set.

Algorithm 4 Rank

Require: Population p, point-set S
1: $sum = 0$
2: **for** each creature c in population p **do**
3: $score[c] = E(c, S)$ {Evaluate c with the fitness function}
4: $normalized[c] = 1/(1 + score[c])$
5: $sum = sum + normalized[c]$
6: **end for**
7: **for** each creature c in population p **do**
8: $score[c] = normalized[c]/sum$
9: **end for**
10: Sort the population p with respect to $score$
11: Return p

Rescaling and ranking The final fitness of a creature is then obtained by normalizing its raw score: $\frac{1/(1+E(c,s))}{\sum_{c\in p} 1/(1+E(c,S))}$. Maximizing E is equivalent to minimizing this scaled fitness function. Computing the fitness of each creature in the population is described in Algorithm 4. The rest of Algorithm 3 is standard and involves functions to mutate a creature and to perform a crossover between two creatures. These functions are detailed in Algorithms 6 and 5 respectively.

Crossover operation The two creatures passed to the function $Crossover()$ (and then to function $Mutate()$) are selected by tournament. The crossover between two creatures consists in randomly selecting cut points in each creature and exchanging the corresponding sub-trees. In order to prevent the size of the creatures to grow uncontrolled, the depth of both offsprings obtained after crossover is compared against a user-defined maximum depth and the crossover is cancelled if the depth of any offspring is larger than the maximum allowed depth.

Algorithm 5 Crossover

Require: Crossover rate χ, two creatures c_1 and c_2
1: $r = RandomNumberInUnitInterval()$
2: **if** $r < \chi$ **then**
3: Select randomly cut points in c_1 and c_2
4: Exchange the sub-trees at the two cut points above
5: Return the modified creatures $(\widetilde{c_1}, \widetilde{c_2})$
6: **else**
7: Return (c_1, c_2)
8: **end if**

Mutation Each creature is then passed to the function $Mutate()$ where it has a probability μ to be altered (see Algorithm 6).

Altering a creature consists either in creating a new random tree (with a probability μ_0) or selecting at random an internal node and replacing the sub-tree with a newly created random sub-tree (see Algorithm 7).

Algorithm 6 Mutate

Require: Mutation rate μ and a creature c

1: $r = RandomNumberInUnitInterval()$
2: **if** $r < \mu$ **then**
3: Return $MutateCreature(c)$
4: **else**
5: Return c
6: **end if**

Algorithm 7 MutateCreature

Require: Creature c

1: $r = RandomNumberInUnitInterval()$
2: **if** $r < \mu_0$ **then**
3: Return a new random tree
4: **else**
5: Select randomly a mutation point in c
6: Generate a random subtree and attach it at the mutation point to c
7: Return c
8: **end if**

In order to prevent uncontrolled growth of creatures after mutation, we limit the depth of the newly generated random subtree such that the depth of the new creature does not exceed the user-defined maximal depth.

Visualization The surface approximating the input point-set is given by the zero level-set of the best evolved creature $f(x, y, z)$. For visualization purposes, the zero level-set of f can be rendered by techniques such as ray-tracing [24] or approximated by meshing algorithms (e.g. Marching Cubes based algorithms [31]). It is possible that the zero level-set of the best creature contains extra surfaces not present in the original object. In general, these unnecessary parts are naturally clipped by the other primitives during the tree evolution (in particular by the separating primitives). In addition, we also compute an object oriented bounding box for the input point-set and take its intersection with the evolved solid. This prevents unwanted components away from the input point-set.

6 Experiments

6.1 Artificial data

6.1.1 Samples on a cube

First we test the algorithm against synthetic data. We start with a simple example consisting of 65000 points sampled on the surface of a cube. The input data-set is shown in Fig. 5. Segmentation and primitives fitting are first applied to the input data. Six subsets are found during the segmentation (see Fig. 5, middle). Each subset corresponds exactly to one of the cube faces.

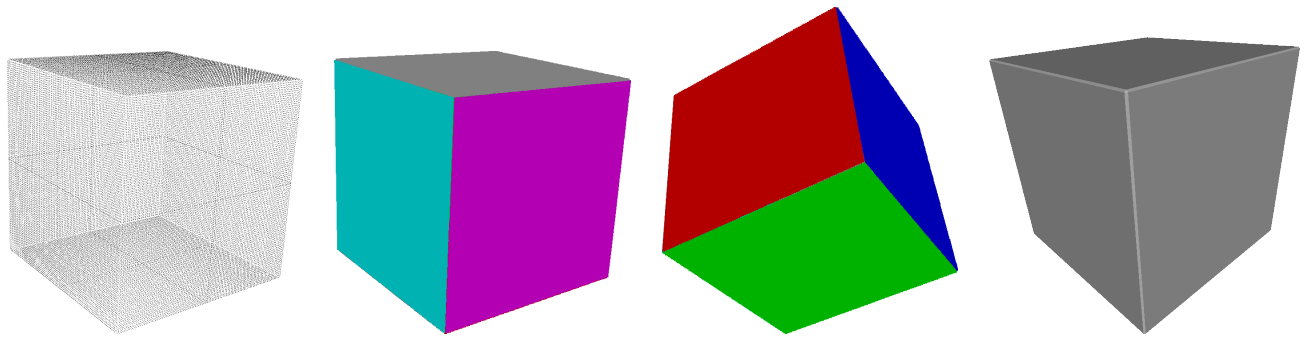

Figure 5: CSG reconstruction of points sampled on the surface of a cube. Left: Samples on the surface. Middle: Result of the segmentation and primitive fitting; the input data is clustered in six subsets; each subset is assigned a different color. Right: The surface obtained from the zero level-set of the evolved function, here approximated by a meshing algorithm.

Segmentation, identification of the most probable primitives and fitting of their parameters are done jointly. For this particular example, each cluster is identified as a planar primitive with parameters fitted during the segmentation step. The parameter values obtained during the fitting are given in table 1 below.

name	n_x	n_y	n_z	d
plane0	0	-1	0	-0.5
plane1	0	1	0	-0.5
plane2	0	0	-1	0.5
plane3	1	0	0	0.5
plane4	0	0	1	0.5
plane5	-1	0	0	0.5

Table 1: Fitted parameter values for the six detected planes.

Note that in this example, no separating primitives are needed as all segmented subsets correspond to planes. Given the input point-set and the list of fitted primitives, genetic programming is then used to evolve an expression that combines the fitted primitives and geometric operations. The final expression is used to describe the geometry corresponding to the point-set. The corresponding surface is obtained as the zero level-set of this expression. For this first example, the following expression was found by genetic programming:

$$\begin{aligned} intersection[plane2, intersection[intersection[subtraction[plane3, \\ union[plane0, plane1]], plane5], plane4]] \end{aligned} \tag{8}$$

Each primitive is defined by a function of the point coordinates (x, y, z), but the coordinates are removed here for readability. An equivalent CSG tree for this expression is shown in Fig. 6. The zero level-set of of the function given by the expression (8) is illustrated in Fig. 5, right. It was produced by sampling the function values on a three-dimensional regular grid and approximating the zero level-set with a meshing algorithm.

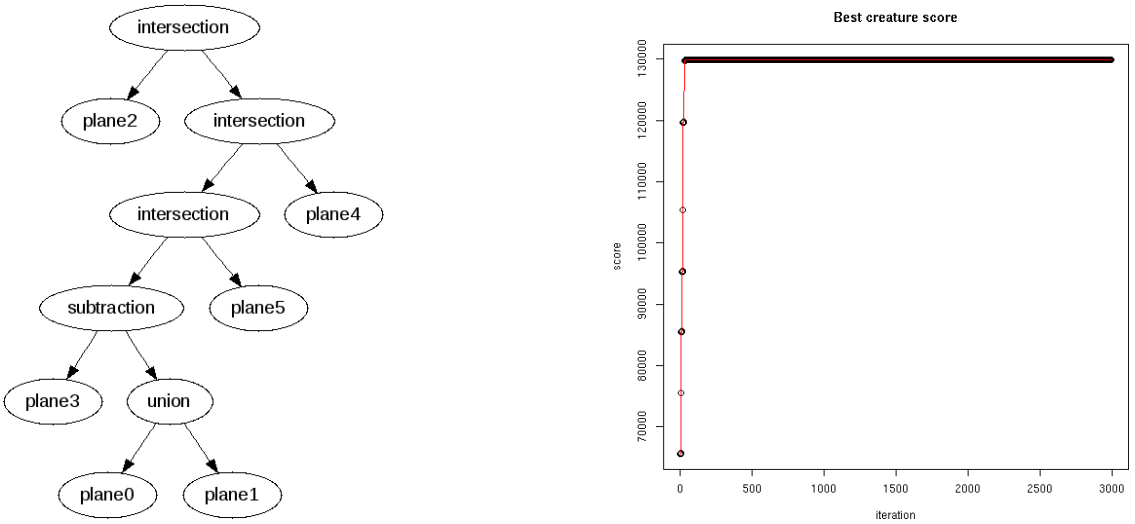

Figure 6: Left: The equivalent tree-based representation of the final expression evolved for the cube. Right: Evolution of the raw fitness function against the number of iterations of the genetic programming.

It is interesting to notice that the expression 8 is minimal. Given six planes, a cube can be defined with five intersections. Another remark is that this evolved expression contains some *subtraction*() and *union*() instead of *intersection*() operations only. The reason is that not all the planes were fitted with the same orientation; plane0 and plane1 have, for example, a different orientation than the other planes. The orientation of these planes needs to be flipped during the evolution of the expression corresponding to the whole object.

The following values were used for the parameters appearing in the fitness function (7): $\epsilon_d = 0.01$ (this value is then scaled by the length of the diagonal of the point-set bounding box) and $\alpha = 10°$. We used the same values for most of the other experiments. For noisy point-sets *alpha* was increased to 35°. The other parameters of the algorithm are the genetic programming parameters. The following values were used: a mutation rate of 0.3, a crossover rate of 0.3, a population of 150 creatures. The initial population of 150 creatures is randomly initialized. A large value for the mutation rate is used to prevent premature convergence to a uniform population.

We let Algorithm 3 run for 3000 iterations. The evolution of the raw fitness function (the value obtained before population scaling) for the best creature of the population against the number of iterations is given in

Fig. 6. This data-set is simple enough such that that after a small number of iterations, the best creature provides already a correct model for the data; its zero level-set describes the surface of the cube. This can be seen in the plot in Fig. 6 with the the raw fitness function value for the best creature reaching large values early on in the process. The subsequent iterations are spent in trying to find simpler expressions and only contribute a small improvement to the raw score.

6.1.2 Double-torus

The second example is a bit more complex, and involves a larger number of primitives. For this example, 17408 points (with associated normals) are sampled on the surface of a double torus object made of planar parts. In total, eighteen clusters are identified and all are recognized correctly as planar surfaces.

One of the expressions discovered by the algorithm results in the tree shown in Fig. 7. This tree (and the corresponding expression) is naturally more complicated than the previous example. This result was obtained with the same parameter values as for the precedent example.

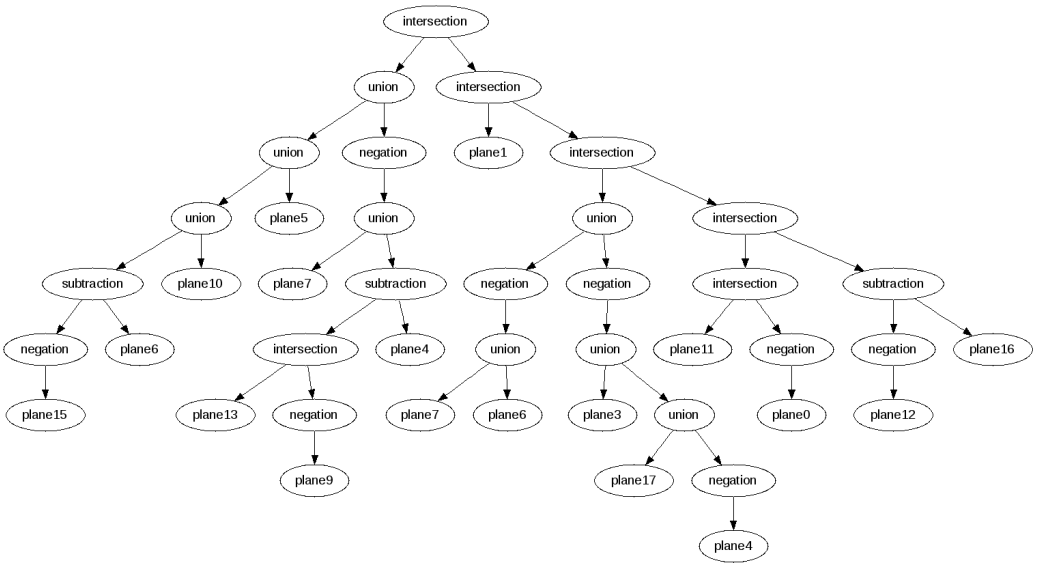

Figure 7: A tree corresponding to one of the expressions describing the double-torus data-set and extracted by our algorithm.

Different runs of the algorithm can produce different expressions, but their zero level-set should identically describe the input data-set. However, the expressions can be different and they may also have different contour levels (but identical zero level-set) as illustrated by the contour plots in Fig. 8 of two different expressions obtained by two runs of the algorithm for 1000 iterations on a given cross-section (here by the plane $y = 0$). Here, the contour plots in the left images are similar to the contour plots of the distance to the boundary of the input object. On the other hand, the contour plots in the right image show that the function value stays almost constant in the upper right corner. It is not particularly surprising that expressions with different level-sets can be obtained, since the constraints that we are trying to enforce control that the input points are on or near the zero level-set of the evolved expression and that the normals at each points are colinear with the gradient of the expression. In this example, both expressions satisfy these criteria. In general, it is preferable to work with distance-like fields, such as in the left image of Fig. 8, as the distance to the boundary can be used in further modeling operations [7, 20]. One possibility to enforce this would be through additional constraints such as function or gradient values inside or outside the object. This would be at the expense of the runtime speed and it deserves further investigation. For now, we can notice that the constraints on the gradient of the expression will guarantee that the function behaves similarly to the distance function at least close to the boundary.

6.2 Scanned and CAD data

6.2.1 Fandisk data

Figure 9 illustrates the results obtained on a more complicated object. In contrary to the previous examples, this object does not involve only planar surfaces. The input point-set and its segmentation are shown in the top row of Fig. 9. The best CSG tree at the end of 3000 iterations is shown in Fig. 10. The zero level-set of the corresponding expression is approximated with a meshing algorithm and illustrated in the bottom row of Fig. 9. The recovered shape is visually a good approximation of the input point-set (see also Fig. 19 for log-plots of the error at each point).

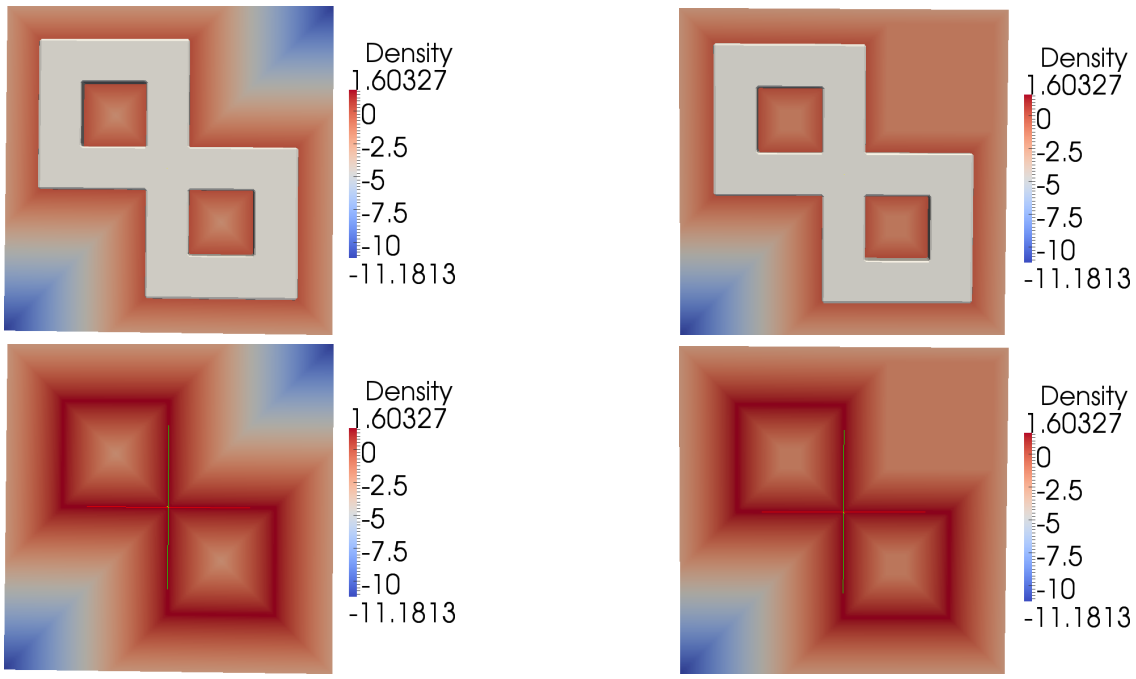

Figure 8: Contour plots on a given cross-section (by the plane $y = 0$) of two expressions obtained by two runs of the algorithm. Top row: visualization of the contour levels with the surface (corresponding to the zero level-set of the expression) superimposed. Bottom row: visualization of the contour levels on a given cross-section ($y = 0$) of the evolved expressions.

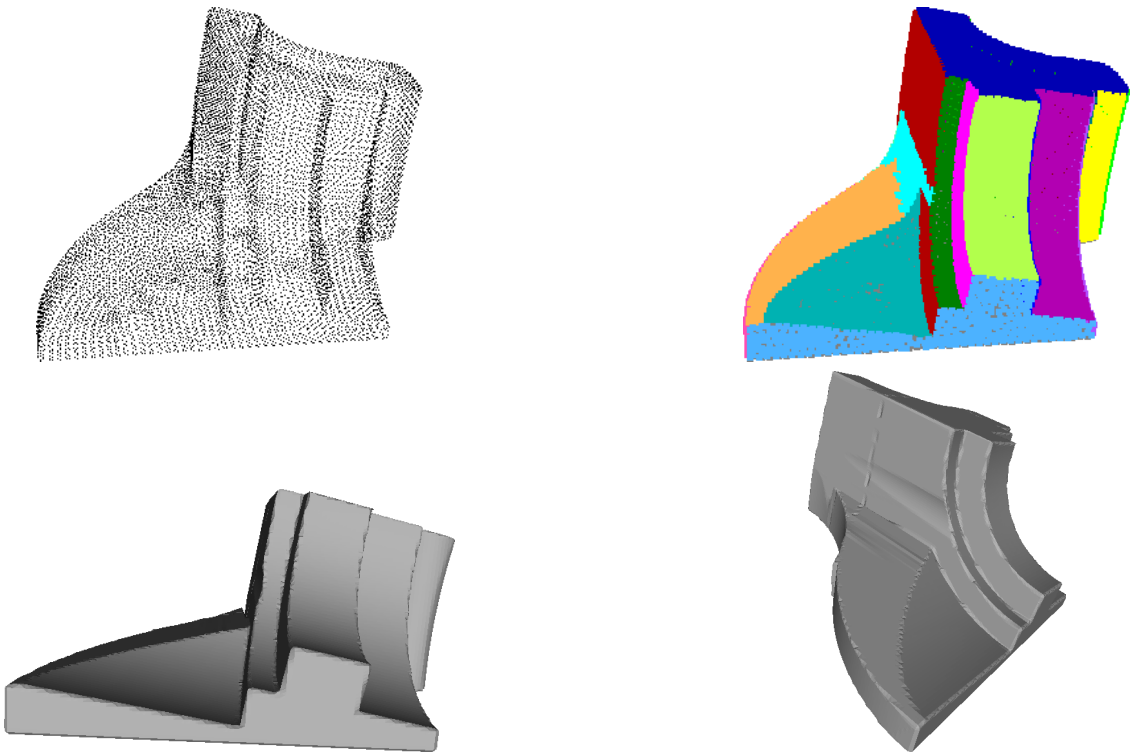

Figure 9: Top row: Left: Samples on the surface of a fandisk. Right: The segmented point-set with randomly colored subsets. Bottom row: The zero level-set of one of the evolved expressions approximated by a meshing algorithm.

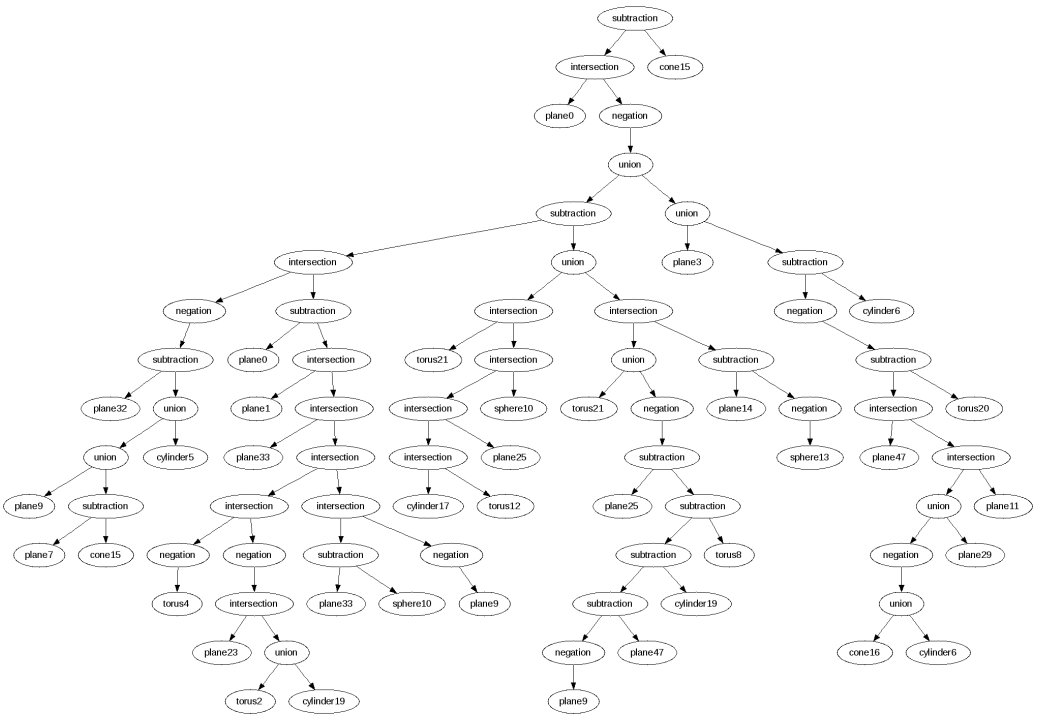

Figure 10: The tree corresponding to one of the solutions discovered by our algorithm.

Evolution of the raw fitness function value of the best creature at each iteration of the algorithm is shown in Fig. 11. The early jumps in the fitness value are large and contribute mostly to the final shape of the object. The later jumps are smaller in amplitude and correspond to the improvement of smaller details or simplification of the expression while preserving the shape satisfying the constraints at the input point-set. As shapes become more complicated with smaller features and details, it takes a larger number of iterations for the algorithm to converge to an expression corresponding to a good approximation of the shape. Significant improvements continue to occur in the raw fitness score even after a large number of iterations.

6.2.2 Results for additional data-sets

Results of the proposed approach applied to additional input point-sets are shown in Fig. 12 and 13. The point-sets shown in Fig. 12 are obtained by sampling from triangle meshes. All expressions were obtained by running the algorithm for 5000 iterations with the same parameters as in the precedent examples. The initial population contains 150 random creatures. The size of the population is kept unchanged during the iterations. Both objects illustrate that reasonably complex shapes can be processed by our algorithm. The recovered object in the bottom row of Fig. 12 also illustrates the effect of the separating primitives with the recovered smooth transition surfaces between the planar surfaces (side) and subtracted cylinders. The behavior of the algorithm on scanned data-sets is also illustrated in Fig. 13. Scanned data can contain various defects such as noise, flipped normals and others. The algorithm seems to be robust to such defects.

6.3 Applications

The recovered expression or the construction tree can be inspected and edited in order to perform further modeling operations on the recovered object. Figure 14 illustrates the contour plots obtained by replacing min/max operations by R-functions [45, 37] in the cube expression (8). In general, R-functions or alternative implementation of the set-operations [20] are smoother than min/max, which is a property required for some applications or modeling operations such as material distribution modeling in heterogeneous objects [7], blending or metamorphosis operations [38].

To smooth out the creases (sharp features) resulting from the surface-surface intersections, it is possible to replace the set operations by their blending counter-parts [38]. This is illustrated in Fig. 15 for the cube object. The expression was modified by replacing some of the intersection operations by the blending intersections.

Specifically, all set operations in (8) were implemented in terms of R-functions (5) and (6) and the intersection operation (5), was then replaced by the blending intersection defined by:

$$blendIntersection(S_1, S_2, a_0, a_1, a_2) := intersection(S_1, S_2) + \frac{a_0}{1 + \frac{f_1}{a_1}^2 + \frac{f_2}{a_2}^2} \tag{9}$$

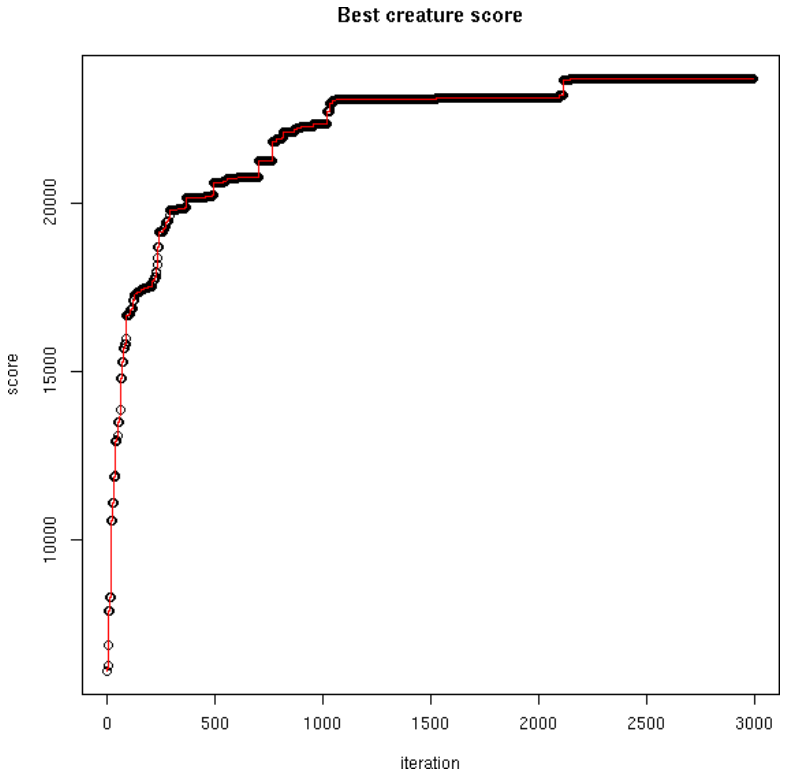

Figure 11: Plot of the raw fitness function value against the number of iterations for the best creature in the population.

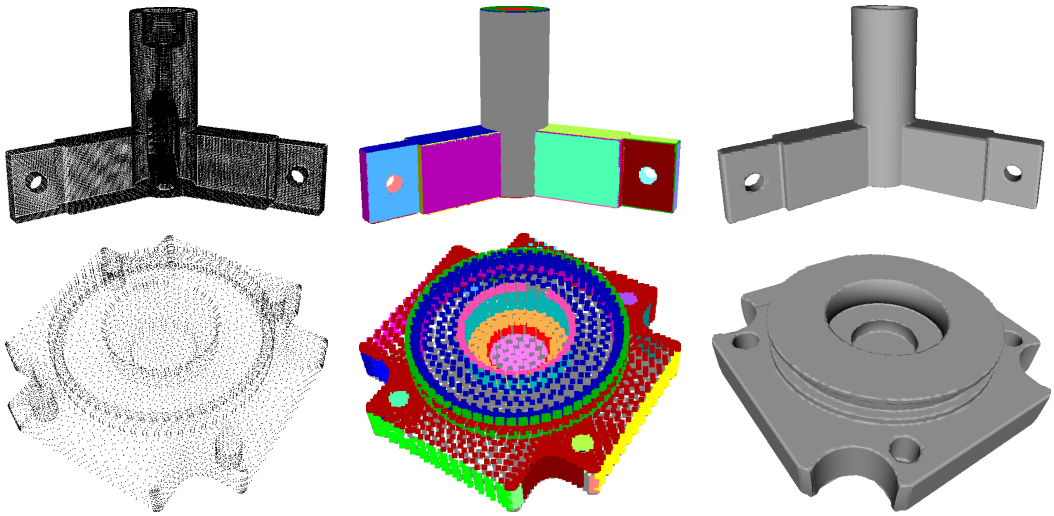

Figure 12: The results of the proposed approach on some CAD parts. For each row, the input, the segmented point-set and the zero level-set of the best expression are shown.

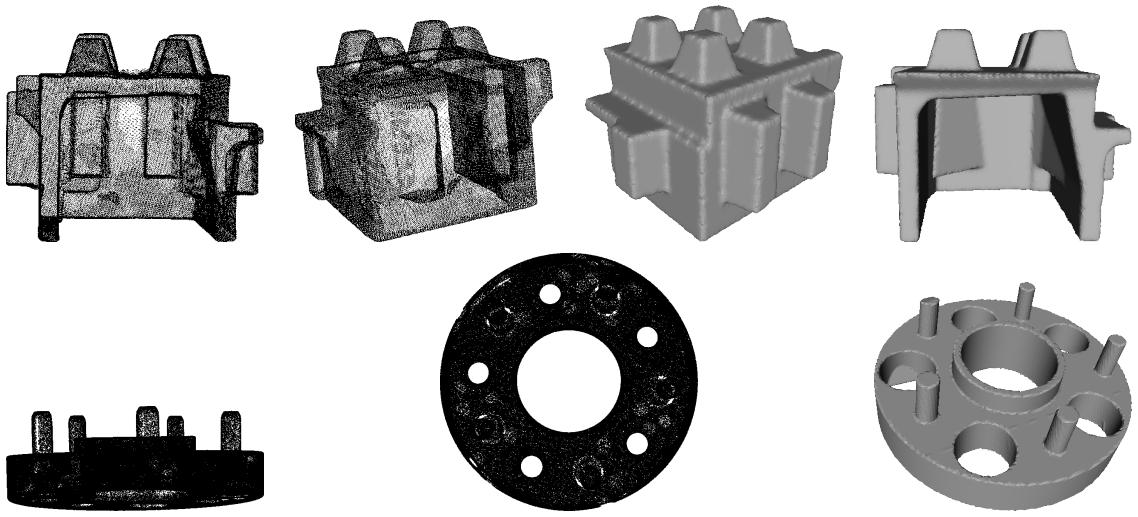

Figure 13: Results obtained on scanned data with the initial point-sets and the surfaces extracted from the recovered expressions.

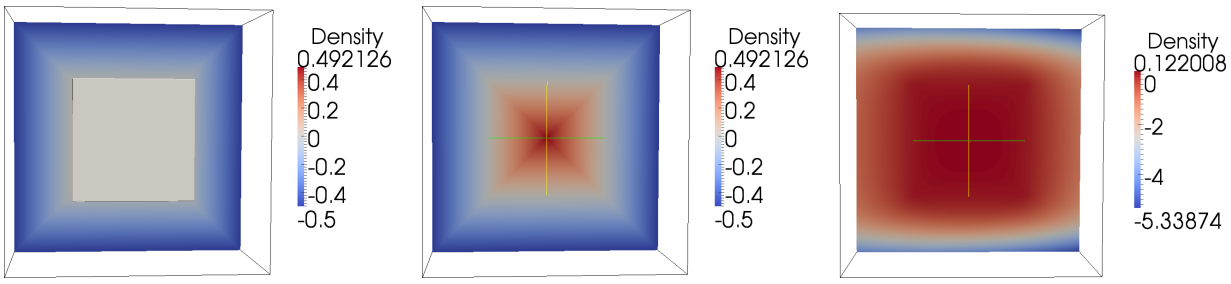

Figure 14: Contour plots of the recovered cube expression when using min/max (left and middle) or R-functions (right) to implement the set-operations.

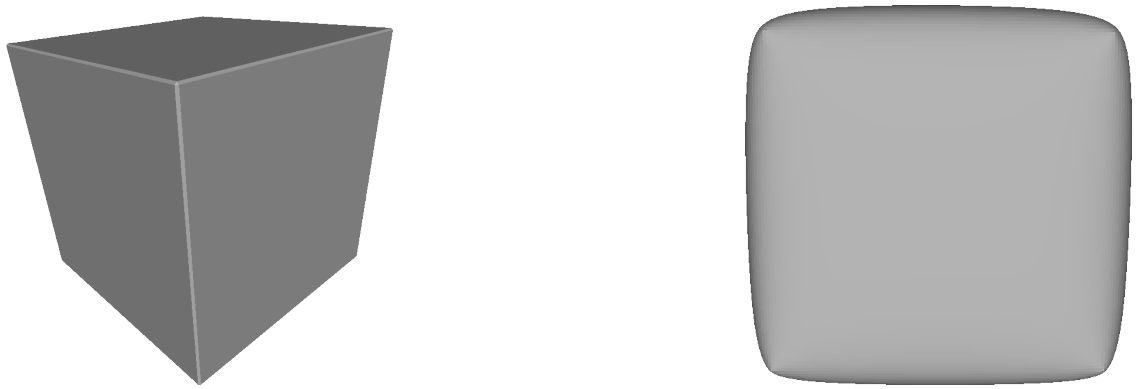

Figure 15: Left: original cube object recovered by our approach. Right: the edited cube with some of the set-theoretic operations replaced by their blended counter-parts.

It is also easy to edit the extracted model by changing some of the primitives or the values of some parameters of these primitives. In the example shown in Fig. 16, the expression for the recovered data-set was modified by doubling the parameter values of the holes (middle image) or the main cylinder (rightmost image). Parameters of extracted expressions can be modified manually by a designer or automatically by some algorithms in order to satisfy some modeling criteria [18, 14].

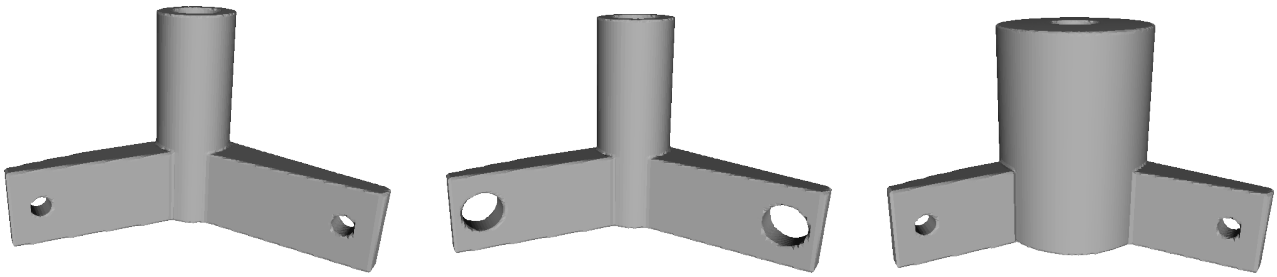

Figure 16: The zero level-set of a recovered object (left). Edited versions where parameters of some of the primitives were changed: diameter of the holes in the middle image and diameter of the main cylinder in the right image.

6.4 Computation time

We made a prototype implementation of the algorithms described in this paper in C++. This implementation is not particularly optimized for speed. The experiments were run on a regular desktop computer with an Intel Core i3 at 3.30 GHz and 4 GB of RAM. The running times for the examples presented in this paper as well as some additional experimental datasets range from dozens of minutes for the simpler shapes to several hours for the more complex shapes. Only one thread was used for the computation.

The computation time depends essentially on the following factors: the complexity of the model, the number of creatures in a population and the number of iterations. The population size needs to be large enough to allow for diversity in the population. In all our experiments, we used 150 creatures per population. For simple shapes such as the cube or the double-torus, convergence is reached quickly as shown by the plots in Fig. 6. Furthermore, since the objects are relatively simple and involve few primitives, the size of the corresponding trees is small and therefore the evaluation is relatively fast. For complex shapes such as the CAD parts or the scanned data, a larger number of iterations is required. The size of the corresponding trees is also much larger. Both factors contribute to a significantly larger running time for these objects.

There are several techniques that could be used for decreasing the overall running time. First, it is possible to use multiple threads, where each thread is responsible for evaluating the objective function (7). Another optimization technique consists in caching some of the results. Each primitive's value and gradient at each input point can be precomputed at the beginning and cached. The value of the objective function (7) for creatures that are not altered by mutation or crossover and selected to the next population can be cached instead of being recomputed.

6.5 Limitations

The limitations of the proposed approach come essentially from two sources: the segmentation and fitting step and the tree extraction step.

Parameters One limitation of the approach is that it depends on various parameters. The fitness function (7) optimized by genetic programming contains three parameters: ϵ_d, α and λ. Additionally there are the usual parameters of genetic programming: the mutation rate, the crossover rate and the size of the population. And finally, there are the parameters related to the segmentation and fitting step (section 5.1). The parameters of this first step usually play a similar role to the parameters ϵ_d and α of the fitness function and are selected based on the quality of the input point-set (presence of noise). Finding appropriate values for the parameter λ required experiments. We found that setting it to $log(N)$ (N is the number of input points) produced good results in all our tests. The genetic programming parameters certainly have influence on the convergence of the algorithm. We used the parameter values given in the previous section for all experiments, which gave acceptable results.

Fitting The tree extraction step tries to find a good expression given the list of fitted primitives computed in the precedent steps. Fitting is usually done within a threshold controlling the accuracy of fit. This threshold needs to account for noise in the input data and can not be too tight. This can result in imprecision in fitting

and under-segmentation (when fitting is used in the segmentation step). Figure 17 illustrates the final objects obtained from two different results from the segmentation and fitting step. For example, in the segmented point-set shown in Fig. 17 - left, separate planar surfaces (e.g., the ones in light orange) are identified as a single cluster and a single primitive is fitted to the corresponding points instead of two planes (compare the second and fourth images in Fig. 17). Sometimes parts of objects are approximated by primitives that do not exactly correspond to the data being described. It results in approximation. See for example the artificial object in Fig. 2 made of two ellipsoids but approximated by tori and spheres in the left and middle rows.

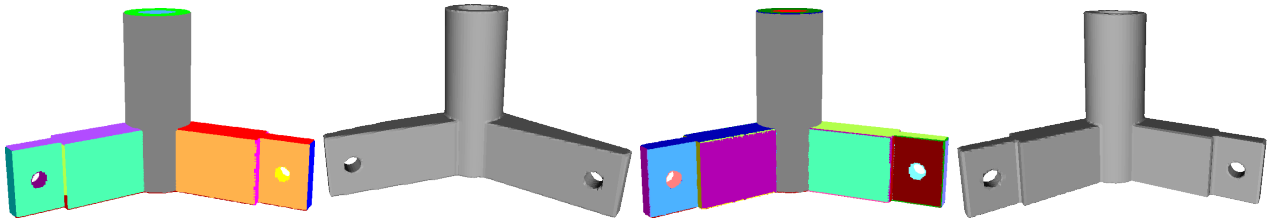

Figure 17: Influence of the results of the segmentation/fitting step on the final object.

The opposite problem occurs when the input data is over-segmented. In this case, the initial data is clustered in lots of subsets of small size. A large number of primitives is used to match these subsets.

Tree extraction A problem of the presented approach is that some of the fitted primitives corresponding to surface patches may be skipped in the final representation, especially when they correspond to small-size features. For example, in the segmented data-set shown in Fig. 18 (left), cylinders with small radii are recognized and fitted to the smooth edges of the object (in yellow and light blue colors), but they are not all used in the recovered object in Fig. 18 (right). As we let Algorithm 3 run for only a finite number of time-steps, it is likely that it will finish in a configuration that is not a global optimum. As a consequence, some of the features may not be reconstructed correctly. One possible way to solve this problem is to identify in the input data the points corresponding to large values of the recovered expression; these points are then extracted from the input data and Algorithm 1 is run again on this new point-set. This obtained expression can be combined with the previous expression to improve the result.

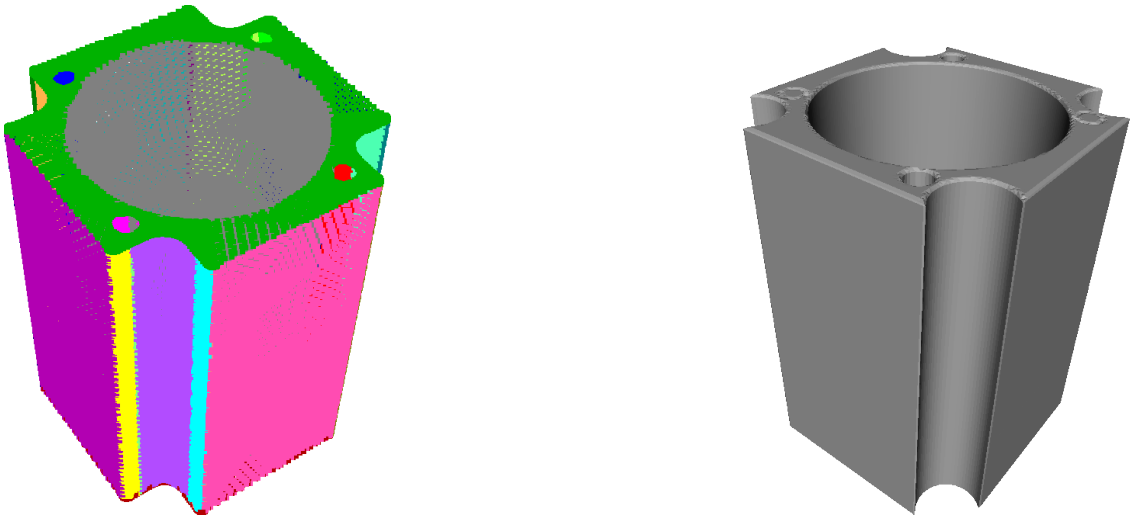

Figure 18: A CAD object with smooth edges (blend). The blend are properly identified during the segmentation and cylinders are fitted to them (left). The recovered object does not correctly reconstruct all these features (right).

Approximation For simple shapes such as the cube shown in Fig. 5 or the double-torus, the segmentation and fitting step produce accurate results and the tree extraction step generates minimal accurate expressions. For more complicated shapes, the tree extraction is most likely not going to yield an exact result but rather some approximation. The approximation can be caused by imprecision in fitting (see the discussion above) or it can be caused by extracted expressions that simplify the shape as in Fig. 18. Figure 19 illustrates the error (log of the error and distribution of the error) between the extracted expression and the original data-set at each input point. The computed error is an approximation of the distance between each point and the object

surface. It is obtained by evaluating the evolved expression at each given point. In order to make this error independent of the object size, we have divided each value by the length of the input point-set bounding box. The histograms in the right column illustrate the distribution of the error.

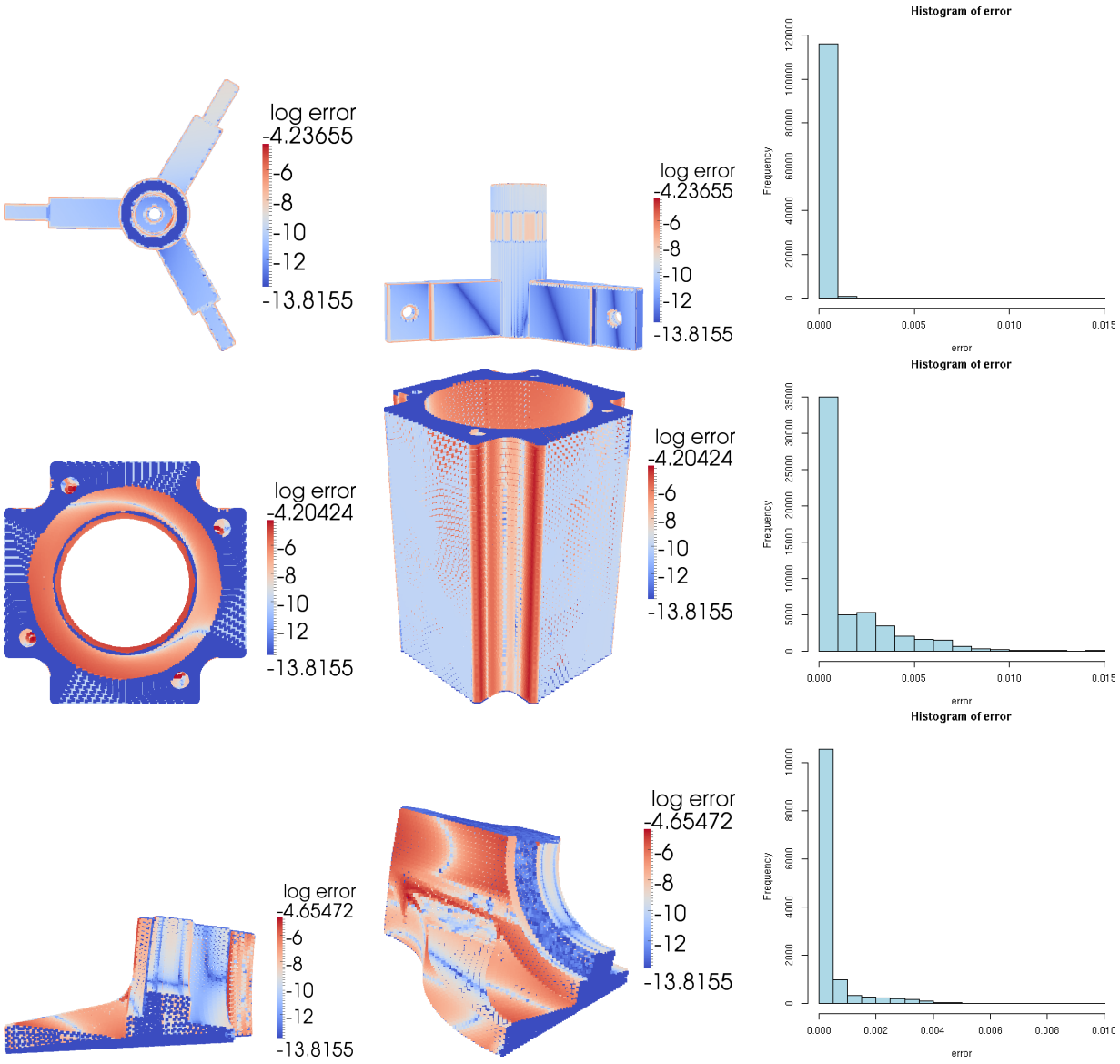

Figure 19: Plots of the log of the error and distribution of the error at each input point as an approximation of the distance between each point and the surface represented by the expression.

Tree bloat A common problem found in approaches using genetic programming is the creation of bloated trees (large trees with redundant information). In this work, we try to limit the tree size by penalizing large trees in the fitness function. This solution seems to be working for our particular case as can be observed from the various trees shown in Fig. 6, 7 or 10 for example. The expressions generated for more complex models seem also to be optimized with respect to the tree size, but it is more difficult to verify. However, there is no guarantee that the trees generated by this method are optimal and do not contain any redundant information at all.

Additional components As discussed in section 5.3, extra surface patches can be produced by this approach. However, such parts are generally clipped by the other primitives during the tree evolution and we also clip the final object by the bounding box of the input point-set to remove unwanted components far away. In practice, we did not experience problems with extra components during our experiments. An approach for handling this problem would consist in penalizing objects with unwanted components during the tree generation step. It requires an extension of the fitness function (7) with a volumetric term that would penalize objects with unwanted components away from the surface. One possible approach is to create a regular grid for the bounding

box containing the point-cloud, and classify each cell in this grid with respect to the surface underlying the point-cloud. The classification of each cell can then be compared to the object generated by genetic programming by looking at the sign of its corresponding function at each cell center. The disadvantage of this approach is that it increases the computational time. It also adds new parameters corresponding to the resolution of the grid and its corners. An alternative semi-automatic solution is to incorporate user-interaction in the process (see also the related discussion in section 6.6 below).

6.6 User-assisted tree extraction

In practice, the loop in Algorithm 3 is executed a finite number of iterations. Consequently, the best creature obtained at the end may be incomplete, have some defects or only approximate the input shape as noted previously. As an example, the pot shown in Fig. 20 was incompletely recovered after 3000 iterations (see the left image). Given the recovered model and the input point-cloud, the set of points corresponding to the handle was automatically extracted. Algorithm 1 was then iteratively applied on this residual point-cloud (the point-cloud corresponding to the handle). The pot body (Fig. 20, left) and the recovered handle were then attached together with the union operation. The final result is shown in the middle image, Fig. 20.

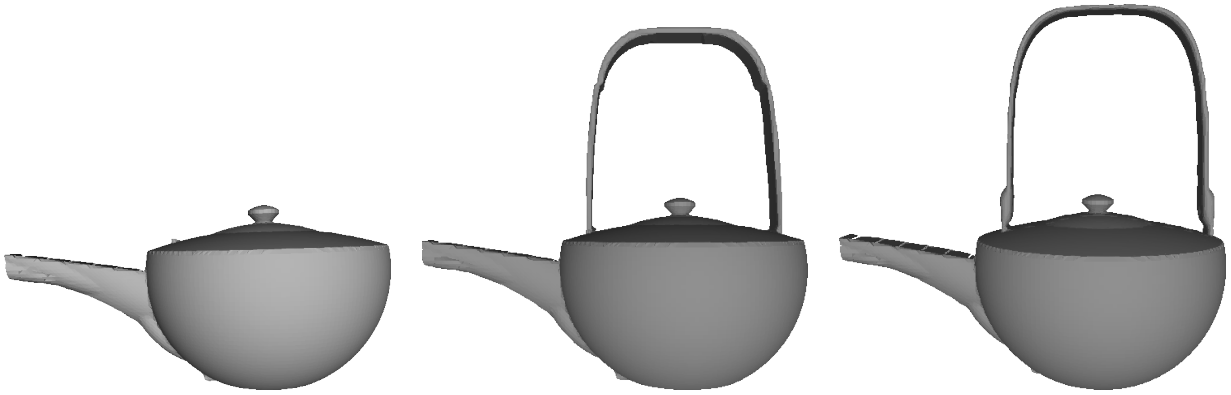

Figure 20: A recovered pot. Left: An incomplete pot recovered after a first iteration of Algorithm 1. Middle: The pot obtained by attaching a handle to the body. The handle is obtained by Algorithm 1. Right: The pot obtained by attaching a handle computed by MPU [36] to the body.

Some of the smallest components near the bottom of the handle are not completely recovered. In order to obtain the model shown in the right image, Fig. 20, the handle was first reconstructed with splines (we used MPU [36]) and the handle with its obtained function was then attached to the pot body with the union operation.

7 Conclusion

We presented a method for extracting a construction tree model from an input discrete point-set. First, the point-set is segmented into several subsets and the best describing primitives are fitted to each of them. In the second step, separating primitives are identified and added to the list of fitted primitives. Finally, genetic programming is used to evolve a constructive model with the fitted primitives in the leaves and geometric operations in the nodes such that the model fits the input point-set. In order to be able to manipulate the extracted tree in further modeling operations, we limit the tree size by using a penalty term in the objective function optimized by genetic programming. Simple applications illustrate manipulation of the extracted construction tree.

Future works The examples used to illustrate our algorithm are mostly man-made parts. One direction of future work is to apply the algorithm to data-sets corresponding to freeform shapes such as, for example, scanned sculptures. The segmentation process may be more difficult with more complex primitives to be fitted.

As our algorithm can produce models that do not completely recover the surface of the given input point-set (features or details missing, for example), it seems important to combine the presented approach with interactive tools to help the user in the reverse engineering process. A preliminary approach is described in section 6.6. However, combining this algorithm with interactive tools deserves further investigation.

Acknowledgements

The authors would like to thank the reviewers for their valuable comments and suggestions. In particular, the example in Fig. 4 and the corresponding discussion were suggested by one of the reviewers.

The authors thank the authors of [30] for providing the data used in their work. This data was used for Fig. 13. The authors also thank AIM@Shape for providing the data used in Fig. 9, 12 (bottom row) and 18. Finally, the authors thank E. Kartasheva for providing the data used for Fig. 12 (top row).

References

[1] M. Attene, B. Falcidieno, and M. Spagnuolo. Hierarchical mesh segmentation based on fitting primitives. *The Visual Computer*, 22(3):181–193, 2006.

[2] B. G. Batchelor. Hierarchical shape description based upon convex hulls of concavities. *Journal of Cybernetics*, 10:205–210, 1980.

[3] P. Benkó, G. Kós, T. Várady, L. Andor, and R. Martin. Constrained fitting in reverse engineering. *Computer Aided Geometric Design*, 19(3):173–205, 2002.

[4] P. Benkó, R.R. Martin, and T. Várady. Algorithms for reverse engineering boundary representation models. *Computer-Aided Design*, 33(11):839–851, 2001.

[5] P. Benkó and T. Várady. Segmentation methods for smooth point regions of conventional engineering objects. *Computer-Aided Design*, 36(6):511–523, 2004.

[6] M. Berger, J. A. Levine, L. G. Nonato, G. Taubin, and C. T. Silva. A benchmark for surface reconstruction. *ACM Trans. Graph.*, 32(2):20:1–20:17, apr 2013.

[7] A. Biswas, V. Shapiro, and I. Tsukanov. Heterogeneous material modeling with distance fields. *Comput. Aided Geom. Des.*, 21(3):215–242, 2004.

[8] J. Bloomenthal, C. Bajaj, J. Blinn, M.-P. Cani-Gascuel, A. Rockwood, B. Wyvill, and G. Wyvill. *Introduction to implicit surfaces*. Morgan-Kaufmann, 1997.

[9] S. Buchele. *Three-Dimensional Binary Space Partitioning Tree and Constructive Solid Geometry Tree Construction from Algebraic Boundary Representations*. PhD thesis, The University of Texas at Austin, 1999.

[10] S. Buchele and A. Roles. Binary space partition tree and constructive solid geometry tree representation for objects bounded by curved surfaces. In *Proceedings of the thirteenth Canadian conference on Computational Geometry*, pages 49 – 52, 2001.

[11] S.F. Buchele and R.H. Crawford. Three-dimensional halfspace constructive solid geometry tree construction from implicit boundary representations. *Computer-Aided Design*, 36(11):1063–1073, 2004.

[12] F. Calakli and G. Taubin. Ssd: Smoothed signed distance surface reconstruction. *Computer Graphics Forum*, 30(7):1993–2002, 2011.

[13] K.-H. Chang and C. Chen. 3d shape engineering and design parameterization. *Computer-Aided Design and Applications*, 8(5):681–692, 2011.

[14] J. Chen, V. Shapiro, K. Suresh, and I. Tsukanov. Shape optimization with topological changes and parametric control. *International Journal of Numerical Methods in Engineering*, 71(3):313–346, 2007.

[15] T. K. Dey and J. Sun. An adaptive mls surface for reconstruction with guarantees. In *Proceedings of the third Eurographics symposium on Geometry processing*, SGP '05, pages 43 – 52, 2005.

[16] J. Doboš, N. Mitra, and A. Steed. 3d timeline: Reverse engineering of a part-based provenance from consecutive 3d models. *Computer Graphics Forum*, 33(2):135–144, 2014.

[17] P.-A. Fayolle and A. Pasko. Segmentation of discrete point clouds using an extensible set of templates. *The Visual Computer*, 29(5):449–465, 2013.

[18] P.-A. Fayolle, A. Pasko, E. Kartasheva, and N. Mirenkov. Shape recovery using functionally represented constructive models. In *Proceedings of International Conference on Shape Modeling and Applications 2004 (SMI'04)*, pages 375–378, 2004.

[19] P.-A. Fayolle, A. Pasko, E. Kartasheva, C. Rosenberger, and C. Toinard. Automation of the volumetric models construction. In *Heterogeneous objects modelling and applications*, pages 214–238. Springer Berlin Heidelberg, 2008.

[20] P.-A. Fayolle, A. Pasko, B. Schmitt, and N. Mirenkov. Constructive heterogeneous object modeling using signed approximate real distance functions. *Journal of Computing and Information Science in Engineering, Transactions of the ASME*, 6(3):221 – 229, September 2006.

[21] M. A. Fischler and R. C. Bolles. Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. *Commun. ACM*, 24(6):381–395, 1981.

[22] R. B. Fisher. Applying knowledge to reverse engineering problems. *Computer-Aided Design*, 36(6):501–510, 2004.

[23] K. Hamza and K. Saitou. Optimization of constructive solid geometry via a tree-based multi-objective genetic algorithm. In *Proceedings of GECCO*, pages 981 – 992, 2004.

[24] J. Hart. Sphere tracing: A geometric method for the antialiased ray tracing of implicit surfaces. *The Visual Computer*, 12(10):527–545, 1996.

[25] H. Hoppe, T. DeRose, T. Duchamp, J. McDonald, and W. Stuetzle. Surface reconstruction from unorganized points. In *SIGGRAPH '92: Proceedings of the 19th annual conference on Computer graphics and interactive techniques*, pages 71–78, New York, NY, USA, 1992. ACM.

[26] M. Kazhdan, M. Bolitho, and H. Hoppe. Poisson surface reconstruction. In *Symposium on Geometry Processing*, pages 61–70, 2006.

[27] M. Kazhdan and H. Hoppe. Screened poisson surface reconstruction. *ACM Trans. Graph.*, 32(3):29:1–29:13, July 2013.

[28] R. Kolluri. *From Range Images to 3D Models*. PhD thesis, The University of California at Berkeley, 2005.

[29] J. Koza. *Genetic Programming*. MIT Press, 1992.

[30] Y. Li, X. Wu, Y. Chrysathou, A. Sharf, D. Cohen-Or, and N. Mitra. Globfit: Consistently fitting primitives by discovering global relations. *ACM Trans. Graph.*, 30(4):52:1–52:12, July 2011.

[31] W.E. Lorensen and H.E. Cline. Marching cubes: a high presolution 3d surface construction algorithm. *Computer Graphics*, 21(4):163–169, 1987. Proceedings of SIGGRAPH 87.

[32] D. Marshall, G. Lukacs, and R. Martin. Robust segmentation of primitives from range data in the presence of geometric degeneracy. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 23(3):304–314, 2001.

[33] N. J. Mitra, A. Nguyen, and L. Guibas. Estimating surface normals in noisy point cloud data. *International Journal of Computational Geometry and Applications*, 14(4-5):261–276, 2004.

[34] B. Morse, T. Yoo, D. Chen, P. Rheingans, and K. Subramanian. Interpolating implicit surfaces from scattered surface data using compactly supported radial basis functions. In *Proceedings of Shape modeling international*, pages 89–98, 2001.

[35] S. Muraki. Volumetric shape description of range data using "blobby model". In *Proceedings of SIGGRAPH*, pages 227–235, 1991.

[36] Y. Ohtake, A. Belyaev, M. Alexa, G. Turk, and H.-P. Seidel. Multi-level partition of unity implicits. *ACM Trans. Graph.*, 22(3):463–470, 2003.

[37] A. Pasko, V. Adzhiev, A. Sourin, and V. Savchenko. Function representation in geometric modeling: concept, implementation and applications. *The Visual Computer*, 11(8):429–446, 1995.

[38] A. Pasko and V. Savchenko. Blending operations for the functionally based constructive geometry. In *set-theoretic Solid Modeling: Techniques and Applications, CSG 94 Conference Proceedings*, pages 151–161. Information Geometers, 1994.

[39] G Renner, editor. *Computer-Aided Design*, volume 35(8). Elsevier, 2003. Special issue on Genetic Algorithms.

[40] A. Requicha. Representations for rigid solids: theory, methods, and systems. *ACM Computing Surveys*, 12(4):437–464, 1980.

[41] A. Ricci. A constructive geometry for computer graphics. *The Computer Journal*, 16(2):157 – 160, 1973.

[42] V. L. Rvachev, L. V. Kurpa, N. G. Sklepus, and L. A. Uchishvili. *Method of R-functions in problems on Bending and Vibrations of Plates of Complex Shape.* Naukova Dumka, 1973. 121 p. (in Russian).

[43] V. Savchenko, A. Pasko, O. Okunev, and T. Kunii. Function representation of solids reconstructed from scattered surface points and contours. *Comput. Graph. Forum*, 14(4):181–188, 1995.

[44] R. Schnabel, R. Wahl, and R. Klein. Efficient ransac for point-cloud shape detection. *Computer Graphics Forum*, 26(2):214–226, 2007.

[45] V. Shapiro. Theory of r-functions and applications: A primer. Technical report, Cornell University, November 1988.

[46] V. Shapiro. A convex deficiency tree algorithm for curved polygons. *International Journal of Computational Geometry and Applications*, 11(2):215–238, 2001.

[47] V. Shapiro and D. L. Vossler. Separation for boundary to csg conversion. *ACM Trans. Graph.*, 12(1):35–55, 1993.

[48] C. Shen. *Building Interpolating and Approximating Implicit Surfaces using Moving Least Squares.* PhD thesis, The University of California at Berkeley, 2006.

[49] S. Silva, P.-A. Fayolle, J. Vincent, G. Pauron, C. Rosenberger, and C. Toinard. Evolutionary computation approaches for shape modelling and fitting. In *Progress in Artificial Intelligence*, pages 144–155. Springer Berlin Heidelberg, 2005.

[50] G Taubin. Distance approximation for rasterizing implicit curves. *ACM Transactions on Graphics*, 13(1):3–42, 1994.

[51] W. B. Thompson, J. C. Owen, H. J. de St. Germain, Jr. Stark, S. R., and T. C. Henderson. Feature-based reverse engineering of mechanical parts. *IEEE Transactions on Robotics and Automation*, 15(1):57–66, 1999.

[52] G. Turk and J. OBrien. Shape transformation using variational implicit functions. In *Proceedings of SIGGRAPH*, pages 335–342, 1999.

[53] M. Vanco and G. Brunnett. Direct segmentation of algebraic models for reverse engineering. *Computing*, 72(1):207–220, 2004.

[54] T. Várady, P. Benkó, and G. Kos. Reverse engineering regular objects: simple segmentation and surface fitting procedures. *Int. J. of Shape Modeling*, 3(4):127–141, 1998.

[55] T. Várady, M.A. Facello, and Z. Terék. Automatic extraction of surface structures in digital shape reconstruction. *Computer-Aided Design*, 39(5):379–388, 2007.

[56] T. Varady, R.R. Martin, and J. Cox. Reverse engineering of geometric models-an introduction. *Computer-Aided Design*, 29(4):255–268, 1997.

[57] S. Venkataraman, M. Sohoni, and V. Kulkarni. A graph-based framework for feature recognition. In *Sixth ACM Symposium on Solid Modeling and Applications*, pages 194–205. ACM, 2001.

[58] J. Wang, D. Gu, Z. Gao, Z. Yu, C. Tan, and L. Zhou. Feature-based solid model reconstruction. *Journal of Computing and Information Science in Engineering*, 13(1):011004–1 – 011004–13, 2013.

[59] D. Weiss. *Geometry-based structural optimization on CAD specification trees.* PhD thesis, ETH Zurich, 2009.

[60] Wolfram Research. Mathematica 7.0, 2008.

[61] T. C. Woo. Feature extraction by volume decomposition. In *Proc. Conference on CAD/CAM Technology in Mechanical Engineering*, pages 39 – 45, Cambridge,MA, 1982.

[62] J. Xiao and Y. Furukawa. Reconstructing the world museums. *International Journal of Computer Vision*, pages 1–16, 2014.

[63] H. Zhang, K. Xu, W. Jiang, J. Lin, D. Cohen-Or, and B. Chen. Layered analysis of irregular facades via symmetry maximization. *ACM Transactions on Graphics*, 32(4):121, 2013.