Approximate Congruence Detection of Model Features for Reverse Engineering

C. H. Gao F. C. Langbein A. D. Marshall R. R. Martin

Department of Computer Science, Cardiff University
PO Box 916, 5 The Parade, Cardiff, CF24 3XF, UK

Abstract

Reverse engineering allows the geometric reconstruction of simple mechanical parts. However, the resulting models suffer from inaccuracies caused by errors in measurement and reconstruction so such models do not have the exact congruences, symmetries and other regularities the original designer intended. We wish to impose such regularities in a beautification process. This paper discusses the particular problem of detecting approximate congruences between parts (e.g. a pair of handles) of a reconstructed B-rep model, so that a subsequent step can enforce them exactly. A practical detection algorithm is given for models defined using planes, spheres, cylinders, cones and tori. Analysis of the algorithm and experimental results show that expected congruences are detected reasonably quickly.

Keywords: *Approximate Congruence Detection; Beautification; Reverse Engineering; Geometric Modelling.*

1. Introduction

Reverse engineering the shape of a 3D object entails reconstructing a geometric model of the object from measured data [22, 23]. Our goal is to create a system that reconstructs B-rep models of simple engineering objects with a minimum of human interaction, usable by naive users as well as engineers. For maximum utility, the generated model should have all the intentional *geometric regularities* present in the original, ideal, design of the part.

This paper considers objects made from planar, spherical, cylindrical, conical and toroidal surfaces that either meet at edges, or are connected by fixed-radius rolling ball blends. It has been shown [14, 19] that a wide range of mechanical components can be described solely using these surfaces; algorithms are available that reliably determine such faces from point clouds [15]. We assume here that the blends are represented as edge attributes in the initial model. Thus, they are derived from primary surfaces in a final model building step, and so can be ignored in the rest of the paper.

In reverse engineering, numerical errors occur in the reconstruction algorithms, and noise is present in measured data. Additional errors may arise due to wear of the object, and the manufacturing method used to make it (e.g. added draught angles). Reverse engineering practice usually fits each face independently of the other faces in the model. However, we wish to recover a geometric model of the *ideal* object that the designer conceived. We propose to improve the reconstructed B-rep model by adjusting it in a separate *beautification* post-processing step [11, 12, 13, 18, 19].

To be able to enforce desired regularities on the model, they must first be detected. Here, we consider the problem of detecting *approximate congruences* between parts of the object ('features'). This will allow us to make features exactly congruent where before they were only approximately congruent, and also align features in special relative orientations where before they were only approximately aligned.

For simplicity, we assume that all faces are of the correct type, e.g. a large radius cylinder is not mistakenly represented as a plane. We also assume that the topology of congruent parts is the same (e.g. that one of them does not have very small faces or edges where the other does not). Allowing for such errors could easily be incorporated into the framework described here, at a relatively small extra cost. Merging adjacent points caused by small topological errors when looking for symmetries is addressed in [19]; a similar approach is applicable here.

This paper considers detection of congruences; complete object symmetries are studied elsewhere [18]. A given feature may be symmetric itself, or a set of congruent features may be arranged on a symmetric pattern. Sometimes an object symmetry may be incomplete, but we still wish to find congruences between parts. Finding incomplete symmetry is harder than complete symmetry, but finding congruences can provide some hints.

When we detect a congruence, we also want the isometry (transformation mapping) relating the congruent features, as we not only wish to make each copy of the feature identical, but we also wish to arrange them in the model in a beautified relationship.

Our overall strategy is to identify sets of congruent fea-

tures, and the isometries relating them. We also check if each feature posseses an approximate symmetry, and if the set of features is related by an approximate symmetry; this processing is not described further in this paper.

Our problem can be seen as a generalized registration problem between sets of *discrete, identified* points. Registration algorithms in computer vision [5] are related, but operate on large amounts of data. Our models have relatively few points, so such algorithms are not directly useful. (Also in computer vision, the correspondences between points to be registered are generally not known in advance.)

Our method seeks all sets of approximately congruent features of a model; a feature is defined as some set of connected faces. The number of detected congruence sets should be minimal in the sense that none of the elements of one set is approximately congruent to an element of another set. The features should be maximal in the sense that if two features are congruent, it is not possible to extend them by adding further connected faces, while retaining congruence.

For example, the model in Plate 1(a) comprises three cubical bosses located on top of a octahedron; the bosses are identical except that the left hand boss also has a cylindrical pocket in it. Our algorithm finds the following information: (i) The approximate congruences between the side faces of the octahedron—the 8 red faces in Plate 1(a). They posses 8-fold approximate symmetry. (ii) The approximate congruences between the three object parts each composed of the four side faces of one of the blocks—the 12 red faces in Plate 1(b). These parts each have a 4-fold approximate symmetry. (iii) The approximate congruence between the $five$ faces of the two right-hand bosses—the 10 red faces in Plate 1(c), also having 4-fold approximate symmetry.

2. Previous work

Exact congruence between polyhedra has been studied extensively. Alt et al [2] and Atkinson [4] present optimal $O(n \log n)$ algorithms for geometrical congruence (n is the number of vertices). Akutsu [1] presents a randomised algorithm taking $O(n^{(d-1)/2} \log n)$ time for determining the congruence of point sets in d-dimensions. The above algorithms calculate the transformation between two point sets in $O(n \log n)$ time, but can be hard to implement.

Checking exact congruence is much easier than checking approximate congruence, because in the exact case we only need check if two things are exactly the same. In the approximate case, two things may match within tolerance, but this matching is no longer unambiguous: matching cannot be done sequentially. For example, given lengths 1, 2, 3, 4, if we try to match them with 2, 3, 4, 1 in turn, with tolerance 1.5, then we notice only in the last step that 4 does not match 1. We must try other matchings to find the solution.

For *approximate* congruence detection between two sets of n points, Alt et al. [2] present an $O(n^8)$ algorithm. Schirra's [20] approximate congruence algorithm takes $O(n^4)$ time to test ε-congruence in the plane, while Heffernan [9] presents an $O((\varepsilon/\delta)^6 n^3)$ approximate congruence algorithm for point matching, where ε is a tolerance, δ is a positive real number smaller than ε. Ambuhl et al [3] describe an $O(n^{8.5})$ algorithm for computing approximate congruence of largest point sets. All these algorithms are based on distance checking using a given tolerance ε.

Our approximate congruence algorithm has $O(n^{4.5})$ time complexity, given certain assumptions concerning the nature of the object being analysed. It finds congruent parts of a single object, and is capable of handling a wider range of models than point sets and polyhedral objects. As well as finding congruences, it also reports the isometric mapping(s) relating congruent features. Our algorithm does not need a predetermined tolerance, but automatically chooses a tolerance level. The way in which the tolerance is chosen allows a greedy approach to matching which is the basic reason for the good performance.

3. Congruence of faces from a set of points

One main step in our algorithm determines if two faces are congruent. This can be done by ensuring that the faces are portions of surfaces with the same defining parameters (e.g. radii) and patch boundaries (edge loops). The boundaries can only match if they are made up of corresponding congruent curves. The curves are congruent if and only if they represent the same portion of the same space curve with the same parameters.

If the underlying faces are *compatible* (see Sections 5.3), then we choose a set of characteristic points according to the type of face. These are sufficient to guarantee that if the characteristic points have a given congruence, so do the underlying faces; we must do the same for the curves forming the face boundaries as part of this process.

In certain special cases, e.g. two complete circles of the same radius, no points at all are needed to check congruence. However, when we later perform tests on *groups* of faces, we still need a representative point for e.g. a circular face with such a boundary, as described later.

We now consider congruence of edges and faces, explaining which characteristic points are used, and briefly justifying that these are sufficient. We first consider edges, as congruent faces must have congruent edges.

3.1. Congruent edges

For straight line segments, we take both end points as the characteristic points. For circular arcs of equal radii, we take end points and the mid-point of the arc. Any 3 points define a unique circle; the mid-point allows us to choose

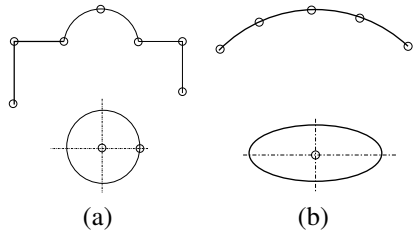


Figure 1. Characteristic points of straight lines, circular arcs and elliptical arcs

between the longer and shorter arc with given end points. Complete circles are congruent if they have the same radii; we use their centres as characteristic points when comparing groups of faces. See Figure 1(a).

For elliptical arcs of the same major and minor radii, we take 5 points which are the end points and points $\frac{1}{4}$, $\frac{1}{2}$, $\frac{3}{4}$ of the way around the arc—see Figure 1(b). Five points define an ellipse, and can disambiguate between the larger and shorter arc. Complete ellipses of the same radii are congruent; we use the centre and the 4 points where the principal axes meet the ellipse as the characteristic points when comparing groups of faces.

We assume that freeform edges are represented as NURBS curves. Control points give suitable characteristic points for matching using algorithms like those in [7, 10, 16]. As the curves may come from different sources, such as intersection routines or fitting procedures, degree elevation and / or reparametrisation may be needed before they are comparable. To match free-form curves with lines, circles, or ellipses, they must be converted to NURBS form first.

3.2. Congruent faces

For a plane, at least 3 points are needed to uniquely define it; 4, 5, 6, and 7 points are needed for a sphere, cylinder, cone, and torus respectively [6].

We initially determine possible congruence of any two faces having matching face types by checking if their boundary edge loops are congruent. We then compare sufficient identified points to determine the underlying surface uniquely. These points come from the face boundary, and in the case of *closed* underlying surfaces, we also include a point inside each face: a boundary splits a closed surface into two finite pieces, and we need to select the correct one. The inner point is chosen as the centre of the face.

In detail, for planar faces, at least 3 points are needed; we normally take the end points of each edge in the boundary of the face. However, if only 1 closed edge bounds the face, we take points at the start, $\frac{1}{3}$, and $\frac{2}{3}$ of the way around the edge; if 2 edges form the edge loop, we take the end and mid-points for each edge. See Figure 2.

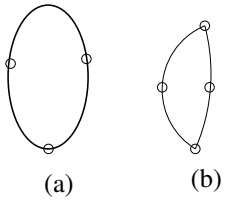


Figure 2. Characteristic points of planar faces

For spherical faces, at least 4 points are needed including the centre of the face. The other 3 are normally the ends of the edges bounding the patch. If there are less than 3 edges, we proceed as in the planar case to obtain sufficient points. For cylindrical, conical and toroidal faces, we use the same general principles, noting whether they are open or closed underlying surface types.

Many special cases may arise, and must be carefully handled. A few examples follow. A further planar case is a face bounded by two loops, an inner and outer loop, each having only one edge. Not all 5-tuples of points serve to define a cylinder, so we must be careful to choose a generic set of 5 points. Given a circular edge with no vertices on it, we must be careful to choose three representative points which are in the same relative positions to the surrounding faces—we must consider more than just the edge itself.

4. Algorithm outline and example

Our algorithm aims to find approximate congruences between parts of an input 3D B-rep model; the output is a list of approximate congruences, together with the isometries which relate the congruent features. A single pair of features may be related by more than one isometry if the features themselves have an approximate symmetry [2]. E.g. the two right-hand bosses in Plate 1 may be aligned in more than one way because of their symmetry.

We look for seed congruences within the model, then use a region growing approach to grow congruences from the seeds. The top level of the algorithm checks for approximate congruence between every compatible face pair in the model using the `congruence` method. The `compatible` method quickly eliminates pairs which can not be congruent. For each congruent face pair found and not already used, the main algorithm calls `expand` in an attempt to extend this congruence to include adjacent faces, then their neighbours, and so on. Again `compatible` and `congruence` are used.

In `expand`, for each initial pair of congruent faces, two lists are set up, each containing one of the faces. Further pairs of faces are then considered, each being a neighbour

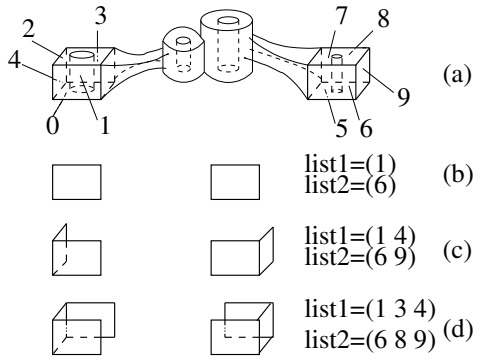


Figure 3. Illustration of finding approximate congruent faces

of some face in each list respectively. If the two new faces are congruent, then we try adding them to the lists which they neighbour, and see if a congruence still exists between the extended lists. If so, the new lists are retained, otherwise the new pair of faces is discarded. This process is repeated until there are no pairs of faces neighbouring the lists left to consider, resulting in a maximal congruence of two sets of adjacent faces.

As each such congruence is found, it is entered into a result list, together with the related mapping(s). After all congruences have been found, this list is rationalised to merge common congruences.

A simple example is used to demonstrate the algorithm. Suppose there are two cubical features at either end of an object (see Figure 3(a)). Suppose the front faces (1, 6) of the two cube features (see Figure 3(b)) are considered as a pair. Firstly, we call `compatible` to check whether congruence of the two faces is possible. They are both planar faces with one external edge loop comprising four straight edges of unit length, so `compatible` returns true for this pair of faces. This causes a call to `congruence` using the coordinates of the characteristic points of each face. These faces are congruent, so the algorithm calls `expand` to extend the congruence to neighbouring congruent face pairs.

In `expand`, from the initial face pair, we put face (1) in `list1` and face (6) in `list2`. Suppose we next find that bottom face (0) is adjacent to `list1`, and bottom face (5) is adjacent to `list2`. We call `compatible` to check possible congruence of the faces (0 5). They can not be congruent because the circle radii of their internal edge loops are not equal. These two faces are ignored. We might then find face (2) to be adjacent to `list1` and face (7) to be adjacent to `list2`. They are also not compatible and are ignored.

Continuing, we might find that face (4) is adjacent to `list1` and face (9) is adjacent to `list2`. The `compatible` and `congruence` methods show that faces

```
00: find_congruences(body)
01: INPUT: body
02: OUTPUT: list of congruence lists,
03:         list of mapping lists
04: integer n = number_of_faces(body)
05: global array of faces f[n]
06: f[1..n] = get_faces(body)
07: array of integers checked[n, n]
08: checked[1..n, 1..n] = 0
09:        //-1 means not congruent
10:        // 0 means not checked
11:        // 1 means congruent
12:        // 2 means already added
13: list of congruence_lists = {},
14: list of mapping_lists = {},
15: FOR i = 1 to n
16:  FOR j = i + 1 to n
17:   IF checked[i, j] != 2 THEN
18:    IF checked[i, j] == 0 THEN
19:     checked[i, j] = compatible(f[i], f[j]) &&
20:                     congruence(f[i], f[j])
21:     IF checked[i, j] == 1 THEN
22:      result = expand(i, j)
23:      congruence_lists = append(result.face_lists)
24:      mapping_lists = append(result.map_list)
25:  END FOR
26: END FOR
27: rationalise(congruence_lists, mapping_lists)
28: RETURN congruence_lists, mapping_lists.
```

Figure 4. The main algorithm

(4 9) are congruent. We add face (4) to `list1` and face (9) to `list2`. We then call `congruence` on the two lists, `list1` = (1 4) and `list2` = (6 9). They are congruent (see Figure 3(c)), so faces 4 and 9 are kept in these lists. In a similar way, faces (3) and (8) are added to `list1` and `list2` (see Figure 3(d)).

No further adjacent congruent face pairs can be found for those faces in the face lists. Thus, the faces (1 3 4) in `list1` form a feature which is congruent to the feature comprising faces (6 8 9) in `list2`. This fact is then stored in the result list as ((1 3 4) (6 8 9)); the corresponding isometry is also stored.

The algorithm continues processing all further face pairs. Any other congruent face groups are also found.

5. Algorithm details

5.1. The main algorithm

The main algorithm is shown in Figure 4. The top level

of the algorithm finds congruent face pairs as seeds, and then tries to expand them to congruent adjacent face groups.

First, the array `f` collects all faces of the B-rep model (line 6). A global integer array, `checked` (line 8), is set up to describe the congruence state of each possible pair of faces (see later). Then (lines 15–26), in two `FOR` loops, the algorithm checks all face pairs. Any pair which has already been output as part of a congruent group of faces (`checked` = 2) is skipped.

If congruence has not yet been decided for the pair (`checked` = 0), we check if the pair is potentially congruent using the `compatible` method. If so, we call `congruence` to decide if they are related by an approximate congruence. If this face pair is congruent and has not yet been used (line 21), we call `expand` to expand this pair to include congruent adjacent face pairs, which gives two lists of congruent adjacent sets of faces which are as large as possible (line 22). Two face lists (`face_lists`) representing the two congruent features, and a list giving their isometric mapping(s) (`map_list`), are returned by `expand`. These are added to the list of congruences found, and the list of mappings (lines 23–24).

Congruences may be related. For example, suppose that face set A is congruent to face set B, and furthermore that face set B is congruent to face set C. We should merge the congruence pairs (A, B) and (B, C) to give the overall congruence list (A, B, C), and the mapping list must also be kept in step. Line 27 carries out this merging. Finally, line 28 returns the list of congruences and associated mappings.

In the main algorithm, and in `expand`, checking congruence of two faces is done frequently, and the result for any face pair may be needed repeatedly. To avoid repeating the computation, the results are cached. A global array `checked` is used to mark the status of each face pair; it is used by both the main algorithm and `expand`. A value of -1 means the two faces are not congruent. 0, the initial value for all face pairs (line 8), means the face pair has not yet been checked. 1 means the two faces have been checked and are congruent. 2 means the two faces are congruent and they have already been used as part of some face group in a congruence. Only if a face pair has not already been considered (line 18), is it checked: if the `checked` value is still 0, the congruence of the face pair is determined and the result is written to the `checked` array (line 19). If the `checked` value of the face pair is 1, i.e. the face pair is congruent but not yet used, we call `expand` to expand this congruent face pair to adjacent congruent face pairs. If the `checked` value of the face pair is -1, this pair is ignored and is not processed by `expand`.

The `expand` method is shown in Figure 5. The input is a congruent face pair and the output comprises two approximately congruent features, as face lists, and a list of isometric mappings relating them. The initial face pair is expanded

```
00: expand(i, j)
01: INPUT: face pair indices i, j,
02: OUTPUT:face_lists and mapping_list
03: list of faces list1 = list(f[i])
04: list of faces list2 = list(f[j])
05: list of faces list3 = f-f[i]-f[j]
06: WHILE an unused neighbour f[m] of list1 remains
07:  list3 = list3 - f[m]
08:  WHILE an unused neighbour f[n] of list2 remains
09:   IF checked[m, n] == 0 THEN
10:    checked[m, n] = compatible(f[i], f[j]) &&
11:                    congruence(f[i], f[j])
12:    IF checked[m, n] == 1 THEN
13:     list of faces lista = list1 + f[m]
14:     list of faces listb = list2 + f[n]
15:     IF congruence(lista, listb) THEN
16:      checked[m, n]=2
17:      list1 = list1 + f[m]
18:      list2 = list2 + f[n]
19       list3 = list3 - f[n]
20:      BREAK
21:  END WHILE
22: END WHILE
23: result.face_lists = list(list1, list2)
24: result.map_list = map_list
25: RETURN result
```

Figure 5. The `expand` method

to include all their neighbours which preserve congruence, then the neighbours' neighbours, and so on.

5.2. The `expand` method

The two initial faces are put into two separate lists (lines 3–4), and these lists are expanded. Faces are added to each of the two lists such that (i) each list is a connected group of faces, (ii) each list is related by a congruence to the other, and (iii) each list is as large as possible.

The two `WHILE` loops process all congruent face pairs (lines 9–11) neighbouring each list, as each list grows. If face `f[m]` is adjacent to `list1`, `f[n]` is adjacent to `list2`, and `f[m]` and `f[n]` are related by an unused approximate congruence (line 12), we create two temporary lists: `lista = f[m] + list1` and `listb = f[n] + list2` (lines 13–14). We then call `congruence` to determine if `lista` and `listb` are congruent (line 15). If so, we permanently add `f[m]` and `f[n]` to `list1` and `list2` respectively (lines 17–19). We repeatedly add further such pairs to `list1` and `list2` until no more adjacent congruent face pairs exist in the face list (lines 6–22). The `checked` array is used for efficiency as explained earlier. As face pairs are added to `list1` and `list2`, their

`checked` values are set to 2 to avoid being considered further (line 16). `list1` and `list2` are returned in a result list (line 23). The list of isometries is also returned; it comes from a global variable set by the `congruence` method (line 24).

5.3. The `compatible` method

Both the main algorithm and `expand` call `congruence` to find an approximate congruence between two sets of points if any exists. The `congruence` method is expensive, especially for complex faces. Thus, we use the `compatible` method to decide quickly if a pair of faces can potentially be congruent before we run `congruence`. For example, if one face is planar and the other face is cylindrical, the two faces cannot be congruent. The principles used by the `compatible` method are that faces can only be congruent if they satisfy the following requirements (within tolerances where appropriate):

Faces must have the same face type (planar, cylindrical, etc.). Cylinders, cones, spheres and tori must have the same convexity or concavity. Radii of cylinders, cones, spheres and tori must agree. Semi-angles of cones must agree. Faces must have the same number of edge loops. Corresponding loops must agree in type (loops may be external, internal or end loops). Corresponding loops must have the same number of edges. Corresponding edges must agree in length. Corresponding vertices must have the same number of edges around them. Face types around corresponding vertices must be consistent.

If two faces meet all the above requirements, they *may* be a congruent face pair. If any one of the above tests fails, we do not continue checking the others, and mark the face pair as non-congruent.

5.4. The `congruence` method

The `congruence` method decides if two sets of faces are congruent. Its input comprises two face lists. Its output is a boolean variable indicating whether the two face sets are congruent or not. If they are, the global variable `map_list` is set to the list of isometries relating the two face sets.

First, the characteristic points are collected from each face. If there is at least one transformation which maps the corresponding characteristic points in the two face sets, the two face sets are congruent. A pair of corresponding tetrahedra are sufficient to determine an isometry. Thus, to try to find the transformation, we compute a special tetrahedron from each point set, and find the mapping relating the tetrahedra, if it exists—if not the sets are not congruent. We then simply have to test if this mapping correctly maps the rest of the points in the two sets. This is much quicker than using all points to find the isometry; a similar method

```
00: congruence(ls1, ls2)
01: INPUT: two face lists
02: OUTPUT: boolean congruent and mapping(s)
03: list p = get_characteristic_points(ls1)
04: list q = get_characteristic_points(ls2)
05: integer n = length(p)
06: bool congruent = false; global list map_list = {}
07: point c_p = centroid(p); point c_q = centroid(q)
08: real tol = min(min_{i≠j}(d(p_i, p_j)), min_{i≠j} d(q_i, q_j))/2
09: real max_length = 0; point t_1
10: FOR i = 1 to n
11:  IF max_length < d(c_p, p_i) THEN
12:   max_length = d(c_p, p_i); t_1 = p_i
13: END FOR
14: real max_area = 0; point t_2
15: FOR i = 1 to n
16:  IF max_area < area(c_p, t_1, p_i) THEN
17:   max_area = area(c_p, t_1, p_i); t_2 = p_i
18: END FOR
19: real max_volume = 0; point t_3
20: FOR i = 1 to n
21:  IF max_volume < volume(c_p, t_1, t_2, p_i) THEN
22:   max_volume = volume(c_p, t_1, t_2, p_i); t_3 = p_i
23: END FOR
24: IF max_volume ≈ 0 THEN return plane_cong(ls1,ls2);
25: map_list = {}
26: FOR i =1 to n
27: IF |d(c_p t_1) − d(c_q q_i)| < tol THEN
28:  FOR j = 1 to n, j != i
29:   IF |d(c_p t_2) − d(c_q q_j)| < tol &&
30:      |d(t_1 t_2) − d(q_i q_j)| < tol THEN
31:    FOR k = 1 to n, k != i, k != j
32:     IF |d(c_p t_3) − d(c_q q_k)| < tol &&
33:        |d(t_1 t_3) − d(q_i q_k)| <tol &&
34:        |d(t_2 t_3) − d(q_j q_k)| < tol THEN
35:      transf = calculate_matrix(c_p, t_1, t_2, t_3, c_q, q_i, q_j, q_k)
36:      IF pointset_isometric(transf) &&
37:         face_mapping(transf) THEN
38:       append(map_list, transf)
39:       congruent = true;
40:    END FOR
41:  END FOR
42: END FOR
43: RETURN congruent
```

Figure 6. The `congruence` method

is used in Mills' symmetry algorithm [17]. (The tetrahedra may not be unique, leading to multiple mappings, as noted earlier.) See Figure 6.

Lines 3–4 get the characteristic points for the faces in each face set. We then compute the special tetrahedron for one point set (lines 7–23). Ideally we would like to use the

largest non-degenerate tetrahedron formed by the point set. This is expensive to compute, so instead we find the tetrahedron with initial vertex the centroid of the point set (line 7), together with those points in the set which maximise the length (lines 9–13), area (lines 14–18) and volume (lines 19–23) of the simplices formed as the second, third and fourth points are added. Line 8 sets the tolerance for matching points after transformation to half the smallest distance between any pair of points in either set; this ensures that the wrong points are not matched.

The whole point set is two dimensional if the computed tetrahedron volume (line 24) is approximately zero, in which case we call a similar `plane_cong` method (omitted here) to return the congruence of the two planar faces.

In lines 25–42, we try to map the tetrahedron of the first point set to all suitable tetrahedra from the second point set (there may be more than one), and then check if the distances between all corresponding points between the two point sets are preserved. If they are, we also check that the transformation has mapped the correct face types into each other to guarantee this is a correct congruence.

Lines 26–34 identify points from the second set which form a tetrahedron matching the tetrahedron from the first set. Line 35 determines the transformation by solving a linear system relating the mapped points of the tetrahedra. Lines 36–37 check that the mapping preserves distances from centroids for corresponding points in the two sets, and preserves face types. If so, lines 38–39 add the transformation matrix for the congruence to the list of mappings, and set the result to true.

6. Algorithm analysis

We now consider the running time of the algorithm. For simplicity, we use n to interchangeably denote the number of faces, edges, and vertices; doing so is justified by Euler's formula which is a linear relation between these quantities. Furthermore, while it is possible to construct objects where a few faces have many vertices, and most faces have just a small number, such objects are uncommon in engineering practice, so we will also assume that there is approximately a limited number (i.e. bounded by a constant) of vertices and edges per face. This assumption means that n is proportional to the number of characteristic points for the object. It also means that each face has a bounded number of neighbours. Thus, each face has at most m characteristic points, and each face has at most j neighbours where m and j are small constants. While it is possible to construct objects which *do not* meet these assumptions, many objects which are being reverse engineered will meet them, at least approximately. Thus, what follows is not an analysis of the worst case performance of our algorithm, but a discussion of how well it might perform in practice.

The main algorithm runs over all face pairs, and tests if the pair is congruent using the `congruence` method. For each congruent face pair, it calls `expand` to expand the pair to a list of adjacent congruent faces.

Suppose the `congruence` checker is called on two sets of p points. Lines 3–23 take time $O(p^2)$ because of line 8. Lines 26–35 find mapped tetrahedra, taking $O(p^3)$ time. However, there are no more than $O(p^{1.5})$ matches possible [8]. Lines 36–37 call `pointset_isometric` and `compatible`; each takes time $O(p^2)$. Thus overall, the `congruence` method takes time $O(p^{3.5})$.

Let us now examine the overall algorithm. We will do so by considering three cases, the third of which is worst.

Case one: the object has no congruent features. Only $O(n^2)$ pairs of faces with m points each are checked for congruence; no expanding of lists occurs. Each call to the congruence checker takes time $O(m^{3.5}) = O(1)$. Thus the overall algorithm is $O(n^2)$.

Case two: there are $n/2$ separate congruences, and each congruent face pair has no neighbouring congruent faces. Each call to the congruence checker takes time $O(m^{3.5})$, as no face pairs are expanded; there are $n/(2m)$ such calls. Again, in this case, the algorithm takes time $O(n^2)$.

Case three: there is a single congruence between the two halves of the object. Starting with a candidate face pair, `expand` is only called once. There are many repeated calls to `congruence` as the face lists are expanded. We make j calls at each stage, on sets of $m, 2m, \ldots, n/2$ points. This takes time $O(n^2) + j\sum_{i=1}^{n/2-1} O((mi)^{3.5}) = O(n^{4.5})$

Case three is the worst case under the assumptions we have made, which may not hold in practice—for example, if the object parts have a lot of symmetries, then the `congruence` checker may be called more often. However, such are not normally likely to arise. Thus, overall, we expect the running time to be somewhere between $O(n^2)$ and $O(n^{4.5})$, assuming that each face has a bounded number of neighbours.

7. Experimental results

7.1. Test objects

Various objects were used to validate the congruence detection algorithm. Objects 1 to 4 are simple models with a variety of features. Although Tate's algorithm was designed to detect partial symmetry rather than feature congruence, we also compare some of our results with hers. Objects 5 and 6 are two models from her thesis [21].

Object 1 is a hexagonal block with a cubic pocket (Plate 2(a)). Object 2 is a hexagonal block with two congruent toroidal "handles", two congruent cylindrical faces, two congruent hexagonal bosses, a cubical boss, two congruent spherical bosses and a spherical pocket, as shown in

(Plate 2(b)). Object 3 is an octahedral block with three cubical bosses, one of which has a cylindrical hole (Plate 1). Object 4 is an octahedral block with two congruent cylindrical faces, two congruent hexahedral bosses, one cubical boss, two congruent spherical bosses, and a spherical pocket (Plate 3(a)). Object 5 is a cylinder with a hole (Plate 4); it is symmetric. Object 6 is a piston head (Plate 3(b)).

7.2. Congruences detected

Table 1 shows the algorithmic results. Column 3 is the number of approximate congruences detected between parts of the object. Column 4 is the time taken by our algorithm in seconds. Column 5 lists the number of partial symmetries detected by Tate's algorithm. Column 6 is the time taken by Tate's algorithm in seconds.

In tests 1 to 5 our algorithm found all congruences which we expected from a manual analysis. For object 1, two congruent sets of side faces were found. For object 2, the congruences found were: the two hexagonal bosses, two toroidal surfaces, two cylindrical surfaces, two spherical bosses, and the sides of the cubical boss. For model 3, the congruences found are shown in Plate 1. For model 4, some of the congruences found are shown in Plate 3(a). Altogether they were: two hexagonal bosses, two spherical pockets, the side faces of the cubical boss, the side faces of the main octahedron, and the faces of the two cylindrical bosses. For test object 5, three congruences were found, relating planes 1 and 3, planes 6 and 7, and cylindrical faces 2 and 4 (Plate 4). For object 6, twenty approximate congruences were found. (Some congruent features containing freeform surfaces, were not found, as these were outside the scope of our algorithm.)

7.3. Running time

We performed further tests on the time taken by the algorithm on a 450MHz Pentium III GNU/Linux machine with 256Mb of RAM, using ACIS as the modeller, to analyse a variety of objects, including Objects 1–6 and others; some are shown in Plate 5.

The averaged timings for several runs in each test case, and the number of approximate congruences found, are given in Table 2. A nonlinear analysis was then performed to find the best fit to the timing data of the form $t = k \times n^x$, where t was the time taken and n was the number of faces. This gave an empirical performance for the algorithm of time $O(n^{3.24})$, see Figure 7.

This experimental performance lies between the suggested limits of $O(n^2)$ needed to consider all pairs of faces, and the limiting case suggested of $O(n^{4.5})$. More data is needed before final conclusions can be drawn about the time needed in 'typical' cases. However, we observe from our testing that: (i) For objects with similar numbers of faces, objects with a large number of small congruences are usually analysed more quickly than those with a small number of large congruences. This is because of the repeated calls to `compatible` and `congruence` in `expand` in the latter cases. (ii) The time taken increases as the objects get larger, but not as badly as our limiting case analysis would suggest. Although it is hard to claim that our small test set are representative of 'typical' engineering objects, they do illustrate some congruences which might be found in practice. The times taken for these test objects are low, leading us to conclude that the algorithm can analyse moderately complex objects (of say 200 faces, such as might be encountered in practical reverse engineering) within a few minutes.

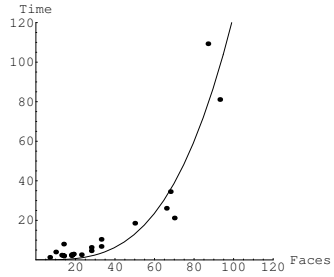


Figure 7. Algorithm performance

Number of faces	Number of congruence groups	Time taken (seconds)
7	3	1.30
10	3	4.00
13	2	2.40
14	5	8.00
14	3	2.09
14	3	1.98
18	3	2.13
18	3	2.63
19	23	2.95
23	3	2.61
28	5	4.62
28	4	6.32
33	7	6.81
33	10	10.40
50	8	18.55
66	3	26.11
68	10	34.52
70	20	21.19
87	10	109.31
93	8	81.11

Table 2. Test results

Test object	Number of faces	Our algorithm		Tate's algorithm	
		Detected congruences	Time (seconds)	Detected symmetries	Time (seconds)
1	13	2	2.20		
2	38	5	9.88		
3	33	5	10.40		
4	27	3	5.45		
5	7	3	1.43	4	3
6	70	20	21.19	14	25

Table 1. Algorithm results on test objects

8. Acknowledgements

The authors wish to thank Tamás Várady of the Hungarian Academy of Sciences and CADMUS Ltd. for providing reverse engineering software and for many helpful discussions. We also thank Susan Tate for providing her models. This research was funded by EPSRC grant GR/M78267.

References

[1] T. Akutsu, On determining the congruence of point sets in d dimensions, *Computational Geometry*, 9:247–256, 1998.

[2] H. Alt, K. Mehlhorn, H. Wagener, E. Welzl, Congruence, similarity and symmetries of geometric objects. *Discrete & Computational Geometry*, 3:237–256, 1988.

[3] C. Ambuhl, S. Chakraborty, B. Gartner, Computing largest common point sets under approximate congruence. `http://citeseer.nj.nec.com/275061.html`.

[4] M. D. Atkinson, An optimal algorithm for geometrical congruence. *Journal of Algorithms*, 8:159–172, 1987.

[5] P. Besl, N. McKay, A method for registration of 3-D shapes. *IEEE Trans. PAMI*, 14(2):239–256, 1992.

[6] BS 7172, Guide to assessment of position, size and departure from nominal form of geometric features, 1989

[7] S. Cohen, G. Elber, R. Bar-yehuda, Matching of freeform curves. *Computer-Aided Design*, 29 (5):369–378, 1997.

[8] J. E. Goodman, J. O'Rourke (eds), *Handbook of Discrete and Computational Geometry*, CRC Press, 1997.

[9] P. J. Heffernan, The translation square map and approximate congruence. *Information Processing Letters*, 39:153-159, 1991.

[10] S. Hu, X. Huang, F, Liu, J. Sun, Content-based indexing of graphics and images via feature classification and curve matching. In: *Proc. 6th Int. Cong. Computer-Aided Design and Computer Graphics*, 827-831, Shanghai, China, 2–6 December 1999.

[11] F. C. Langbein, B. I. Mills, A. D. Marshall, R. R. Martin, Approximate geometric regularities. *Int. J. Shape Modeling*, 7(2):129–162, 2001.

[12] F. C. Langbein, B. I. Mills, A. D. Marshall, R. R. Martin, Finding approximate shape regularities in reverse engineered solid models bounded by simple surfaces. In: D. C. Anderson, K. Lee (eds.), *Proc. 6th ACM Symposium on Solid Modelling and Applications*, 206–215. ACM Press, New York, 2001.

[13] F. C. Langbein, B. I. Mills, A. D. Marshall, R. R. Martin, Recognizing geometric patterns for beautification of reconstructed solid models. In: *Proc. Int. Conf. Shape Modelling and Applications*, 10–19, IEEE Computer Society Press, Los Alamitos, CA, 2001.

[14] E. H. Lockwood, R. H. Macmillan, Geometric Symmetry. *Mathematical Intelligence*, 6 (3):63–67, 1984.

[15] G. Lukács, R. R. Martin, A. D. Marshall, Faithful least-squares fitting of spheres, cylinders, cones and tori for reliable segmentation. In: H. Budkhadj, B. Neumann (eds.), *Proc. 5th European Conf. Computer Vision*, 1, 671–686, Springer, 1998.

[16] Y. Ma, Y. Lee, Detection of loops and singularities of surface intersections. *Computer-Aided Design*, 30 (14):1059–1067, 1998.

[17] B. I. Mills, F. C. Langbein, Determination of Approximate symmetry in geometric models—an exact approach, *submitted to Computational Geometry and Applications*

[18] B. I. Mills, F. C. Langbein, A. D. Marshall, R. R. Martin, Approximate symmetry detection for reverse engineering. In: D. C. Anderson, K. Lee (eds), *Proc. 6th ACM Symposium on Solid Modelling and Applications*, 241–248. ACM Press, New York, 2001.

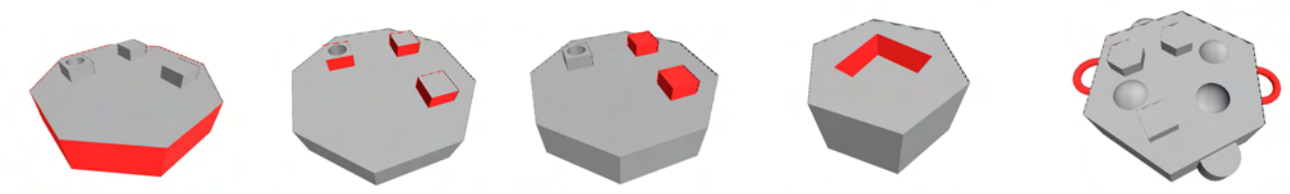
[19] B. I. Mills, F. C. Langbein, A. D. Marshall, R. R. Martin, Estimate of frequencies of geometric regularities for use in reverse engineering of simple mechanical components. Technical Report GVG 2001–1, Geometry and Vision Group, Dept. of Computer Science, Cardiff University, 2001. `http://ralph.cs.cf.ac.uk/papers/Geometry/survey.pdf`.

[20] S. Schirra, Approximate decision algorithms for approximate congruence. *Information Processing Letters*, 43:29–34, 1992.

[21] S. J. Tate, *Symmetry and shape analysis for assembly-oriented CAD*, PhD Thesis, Cranfield University, 2000.

[22] T. Várady, R. R. Martin, *Reverse Engineering*, In: G. Farin, J. Hoschek, M. S. Kim (eds), Handbook of Computer Aided Geometric Design, Ch. 26. Elsevier Science, 2002.

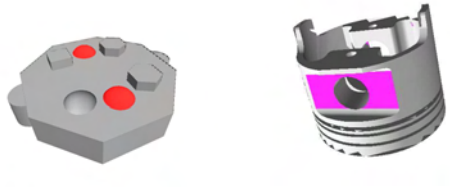
[23] T. Várady, R. R. Martin, J. Cox, Reverse engineering of geometric models—an introduction. *Computer-Aided Design*, 29 (4):255–268, 1997.



(a) (b) (c) (a) (b)

Plate 1. Test object 3 and its approximate congruences

Plate 2. Test objects 1 and 2 and some of their congruences



(a) (b)

Plate 3. Test objects 4 and 6 and some of their congruences

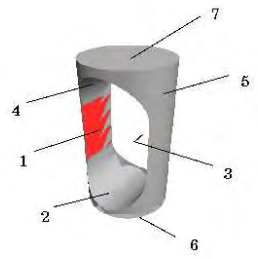


Plate 4. Test object 5 and some of its congruences

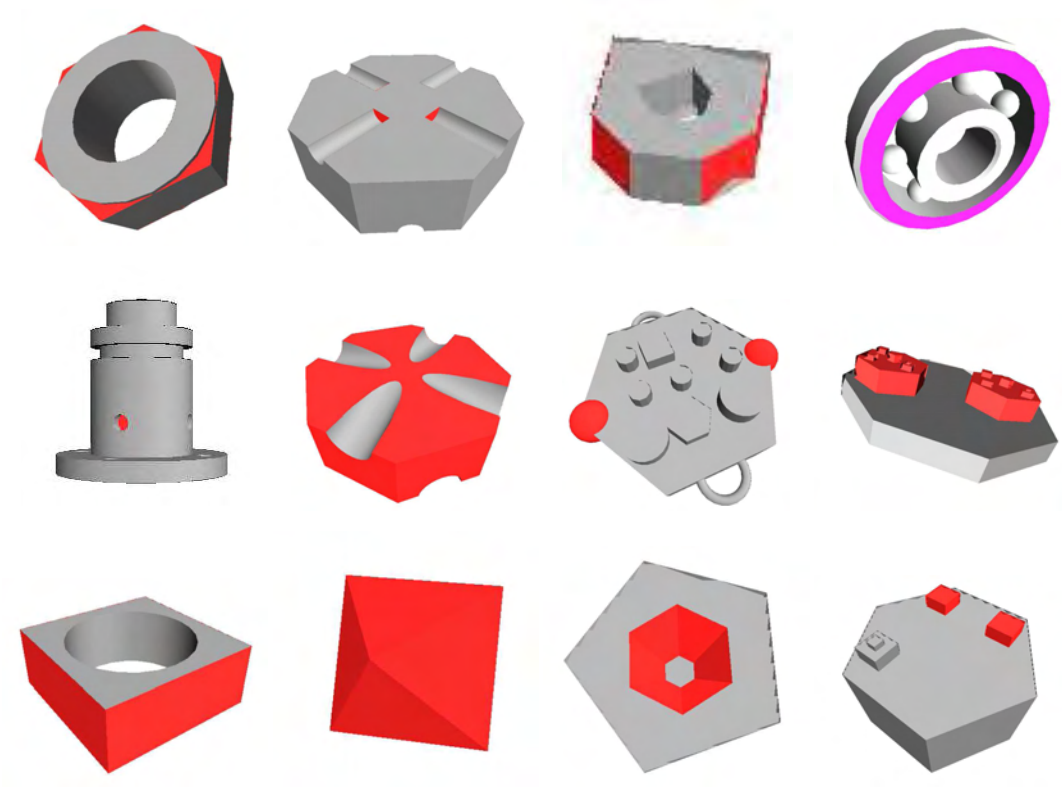


Plate 5. Sample test models used in the experiments