Design For Assembly:
A Computational Approach to Construct Interlocking Wooden Frames

By

Alan Song-Ching Tai

Bachelor of Science in Electrical Engineering
National Taiwan University, 2003
Master of Architecture
University of Pennsylvania 2009

Submitted to The Department of Architecture in Partial Fulfillment of The Requirements for The Degree of

MASTER OF SCIENCE IN ARCHITECTURE STUDIES

AT THE
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

JUNE 2012

Signature of Author: _____
Department of Architecture
May 24, 2012

Certified by: _____
Takehiko Nagakura
Associate Professor of Architecture

Accepted by: _____
Takehiko Nagakura
Associate Professor of Architecture
Chairman, Committee for Graduate Students

Design For Assembly:
A Computational Approach to Construct Interlocking Wooden Frames

By

Alan Song-Ching Tai

Thesis Committee

Takehiko Nagakura
Associate Professor of Architecture
Thesis Advisor

Dennis Shelden
Associate Professor of Practice
Thesis Reader

Design for Assembly: A Computational Approach to Construct Interlocking Wooden Frames

by
Alan Song-Ching Tai

Submitted to the Department of Architecture
on May 24, 2012 in Partial Fulfillment of the Requirements for the Degree of Master of Science in Architecture Studies

ABSTRACT

This thesis explores the computational process of generating and constructing interlocking frames. Its outcome delivers a sophisticated software tool that creates a three dimensional interlocking pattern, analyzes the intersecting conditions between members, and immediately provides instruction of its assembly sequence in animated visualization.

An interlocking frame is a system that consists of short members spanning on a large surface where members lock each other at their mid-spans by simple notches. Such a system should be designed with consideration of its assembly sequence, as a static interlocking form may be described but impossible to assemble in any sequence. Given a three dimensional digital model of an interlocking frame, the feasibility of the disassembly sequence can be assessed by analyzing the geometric contact constraints between each member. The assembly sequence can then be obtained by reversing the disassembly sequence, and helps a designer to evaluate different options in the earlier stage of design.

The proposed tool uses the genetic algorithm and graph searching algorithms to find optimized notching configurations that guarantee an assembly sequence. It can analyze various types of assemblies defined by planar surface contact constraints, and has a potential for further development into a versatile, automated 4D simulation tool.

Thesis Supervisor: Takehiko Nagakura
Title: Associate Professor of Architecture

Acknowledgement

First of all, I would like to thank my thesis advisor Professor Takehiko Nagakura for his support and guidance in the thesis.

I would like to thank Professor Dennis Shelden for joining my committee as a reader during his year on leave.

I would like to thank Professor Patrick Winston for his valuable input on problem solving techniques. I would also like to thank Professor Erik Demaine and Marty Demaine for introducing me the world of wood puzzles.

I want to say thanks to Duks, Rizal, Skylar, Yanni, Dimitris, Masoud for their discussions that help to shape this thesis.

I would like to thank all my colleagues in the SMarchS program, Carolina, Carl, Will, Moritz, Sarah, Josh, Theodora, Katia, and Masoud. You guys are a fantastic group to work with!

Last, I would like to thank Bruce, Iouyu and, most importantly, my parents for their greatest support. Without your support, it would be not be possible for me to continue my studies in architecture.

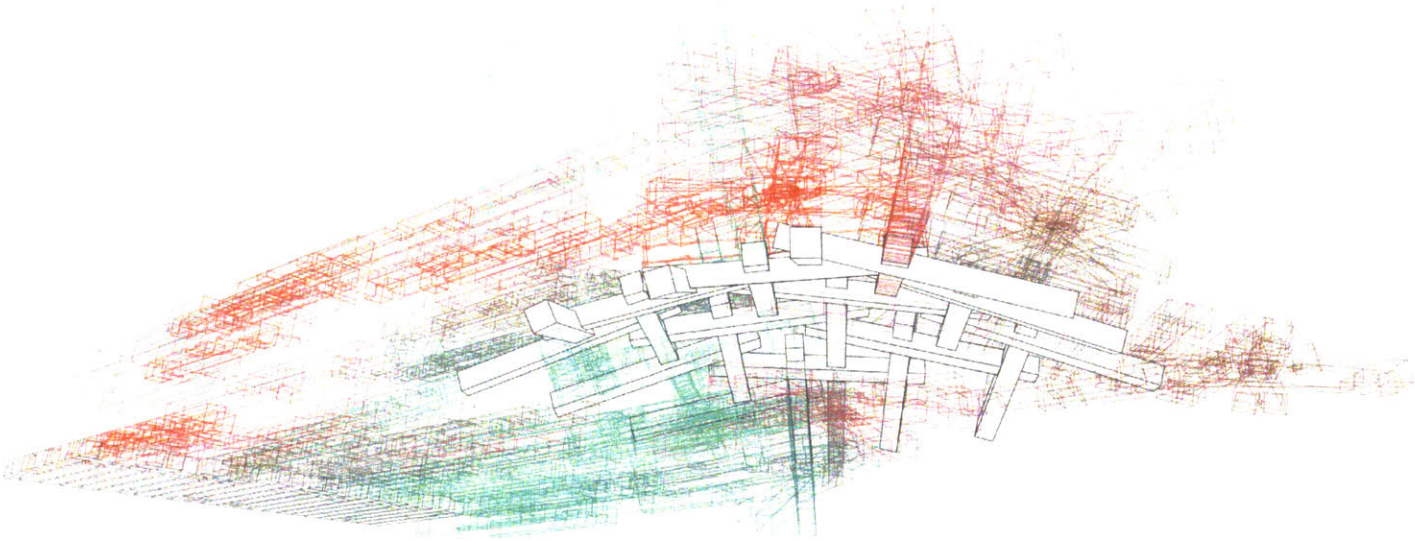
Table of Content

1. Introduction

An interlocking frame is a system that consists of short members spanning across a large surface. In this system, all the members are locked by the adjacent members, resulting in an assembly that is geometrically stable. In the analysis of geometric stability, the material of each member is totally rigid and deformation is not allowed. An interlocking frame should always be designed with the consideration of its assembly sequence. The 3D CAD (Computer-Aided Design) model usually represents the finished state of assembly and does not guarantee a valid assembly sequence. By examining the geometric constraints between members, the free members can be identified in the assembly. The disassembly sequence can be solved by recursively examining and removing the free members. Once the successful disassembly sequence is discovered, the assembly sequence can then be obtained by reversing the disassembly sequence. The algorithm for determining the assembly sequence will be visited in the Geometric Assembly Planning chapter. One of the important goals of this thesis is to introduce the knowledge of geometric assembly planning to the field of architecture, encouraging architects to integrate this knowledge into the design process.

The process of designing interlocking frames involves in transforming a two dimensional grid pattern into a three dimensional interlocking pattern on a free-form surface. Members with the desired dimension will be placed on the three dimensional pattern with respect to the surface orientation. These members will intersect in the three dimensional space and each of the intersection result in a joint between members. These joints require proper notching to ensure the interlocking behavior between members. Because of the angle of intersection is different at every joint, every notching geometry will need to be uniquely defined. The final product is assumed to be digitally fabricated using a five-axis CNC (computer numerical control) machine with a flat end milling bit. This assumption imposes several constraints on the geometry of notching, and limits the number of possible joint configuration. However, even under these fabrication constraints, it is still impossible to examine all configurations due to their large quantity. Therefore, the genetic algorithm and various tree searching algorithms are deployed to search for the optimized joint configuration that allow a valid assembly sequence. The entire process of designing an interlocking frame will be documented in the Methodology chapter.

A plugin program is implemented in the Rhinoceros 3D software to perform the required tasks to design an interlocking frame. This program creates a three dimensional interlocking pattern, analyzes the intersecting conditions between members, and immediately provides instruction of its assembly sequence in an animated visualization. Besides interlocking frames, it can also analyze generic assemblies defined by planar surface contact constraints. By adding real world construction parameters, this tool has the potential for further development into a versatile, automated 4D simulation tool. The detail of this software program will be documented in the Software Implementation chapter.



2. Background

An interlocking frame is a system that consists of short members spanning across a large surface. In this system, all the members are locked by the adjacent members, resulting in an assembly that is geometrically stable.

The motivation of studying interlocking frames originates from my interest in "nail-less constructions," which are construction methods that do not use any glue or fasteners. I am always intrigued by the homogeneity of the material and the intelligence built into geometry in such a construction method. Because interlocking frames does not rely on the friction fit of its material, it allows more tolerance in fabrication. It is also easier to assemble, disassemble and reassemble.

Because of its interlocking behavior, an interlocking frame should always be designed with the consideration of its assembly sequence. This particular constraint leads to the re-evaluation of the digital design and fabrication process, where the assembly process is often overlooked by the designer. This thesis hopes to bring the notion of assemblability to the attention of architects.

2.1 Interlocking Frames

An interlocking frame is a system that consists of short members spanning across a large surface, where members locks each other by simple notches. All of the members in this system do not have any degrees of freedom because of the interlocking geometric constraints. The more rigorous definition of locking and constrains will be introduced in chapter three.

Fig.2.1 shows examples of interlocking frames. The image on the left is a wooden puzzle, and the one on the right is an experimental structure built in the campus of ETH Zürich. The term, interlocking frame, is invented in this paper, and does not exist in any reference. It originates from the name of both families of objects shown in the examples, which one is an "interlocking" puzzle and the other is a reciprocal "frame" structure.

Fig. 2.1
Icosahedron Frequency by Philippe Dubois (Left) Pergola Science City, Pergola Science City by Udo Thönnissen(Right)

http://www.johnrausch.com/puzzleworld/puz/img/lg/icosahedron_frequency_2_1.jpg
http://eat-a-bug.blogspot.com/2011/08/experimental-wood-structures-at-eth.html

2.2 Nail-less Construction: Construction without Mechanical Fastener

The motivation of this thesis emerges from my interest in nail-less constructions, which are constructions process that does not use glue or mechanical fasteners. Fig.2.2 compares the complexity of fabrication and the complexity of assembly for different types of nail-less constructions.

This type of constructions has a long tradition in the oriental countries such as China and Japan. The background research starts by studying various forms of Japanese Wood Joinery. Japanese carpenters have developed a sophisticated collection of joinery techniques, where the geometry of joints varies depending on its function and location

in the building. [1] Sometimes the nail-less construction method has a much deeper meaning than its geometry and appearance. For example, the nail-less construction technique that allows the rebuilding process of Ise Jingu Shrine is the manifestation of death and renewal, which are the central belief of Shintoism. [2] The traditional Japanese joinery techniques require significantly high amount of fabrication complexity, and it is also extremely difficult to apply them in systems that members do not meet orthogonally.

The "yourHouse" project is part of the "Home Delivery: Fabricating the Modern Dwelling" exhibition in the Museum of Modern Art in New York. This project is built by the Digital Design Fabrication Group, led by Professor Larry Sass, at the Massachusetts Institute of Technology. This project relies mostly on the friction joints between planar members to lock the members in place. This approach is documented in detail in the thesis by Daniel Cardoso.[3]

Comparing with other types of nail-less constructions, interlocking frames has a low fabrication complexity but an extremely high assembly complexity. This is the reason why this thesis intends to find the design solution of interlocking frames using computational methods. This approach of using computer programs to describe the complex design processes through trial and simulation is depicted in the book, *The Sciences of the Artificial,* by Herbert Simon. [4]

1 Sato and Nakahara, The Complete Japanese Joinery.
2 Henrichsen, Japan - Culture of Wood, 50.
3 Cardoso, "A Generative Grammar for 2D Manufacturing of 3D Objects."
4 Simon, The Sciences of the Artificial, 135.

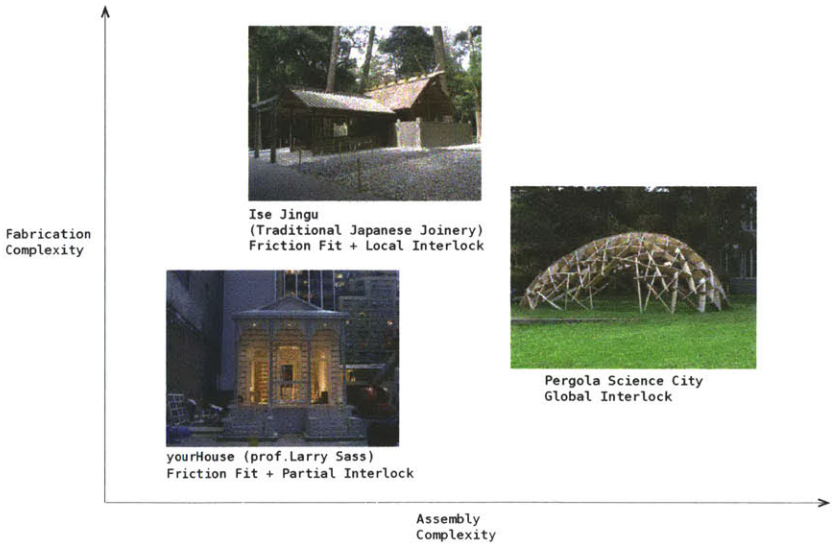


Fig. 2.2
Assembly and Fabrication Complexity in Different Types of Nail-less Constructions

http://inhabitat.com/prefab-friday-prefab-takes-up-residence-at-the-moma/
http://www.gardenvisit.com/garden/ise_jingu_shrine

2.3 Interlocking Frames and Reciprocal Frames

The two examples shown in section 2.1 also belong to the family of reciprocal frames. The difference between interlocking frames and reciprocal frames is -- interlocking frames are defined geometrically; reciprocal frames are defined structurally. Interlocking frames are systems that every member is interlocked by geometric constraints, while reciprocal frames are systems that every member is stabilized by the equilibrium of interacting forces. [5](Fig.2.4) Although interlocking frames and reciprocal frames might have a similar form and composition, they are fundamentally different in their definition. Most of the reciprocal frames are not interlocking frames because their joint conditions usually do not satisfy the requirement of interlocking frames. Also, many of the interlocking frames do not follow the principle of reciprocity in reciprocal frames. However, in this thesis, all of the design examples shown in chapter four and five are both interlocking frames and reciprocal frames. The principle of reciprocity is used as a engineering rule of thumb to construct structurally sound interlocking frames.

Evidence has shown that the earliest reciprocal frames existed in the late twelfth century in Japan. [6] In western, the pattern of a reciprocal frame could be found in the drawings of Leonardo Da Vinci in his famous collection, the Codex Atlanticus.[7] Many contemporary examples of reciprocal frames are found in books and research papers. Some of the papers focus on the structural analysis of a reciprocal frame system, while others emphasize the method of generating its configuration. For example, the paper by Kohlhammer and Kotnik focuses on the methods of structural analysis in a reciprocal system using an diffused iteration process.[8] The paper by Douthe and Barvel designs the reciprocal frame system using a dynamic relaxation method.[9] These references provide valuable insight for constructing interlocking frames.

Fig. 2.3
Bamboo Roof by Shigeru Ban and Cecil Balmond

http://www.saatchi-gallery.co.uk/museums/museum-profile/Rice+University+Art+Gallery++/153.html

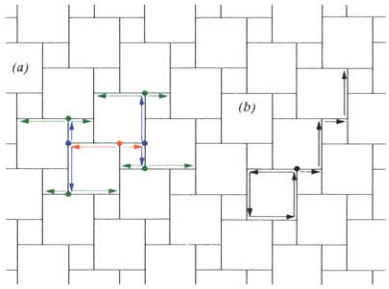


Fig. 2.4
Force Propagation of Reciprocal Frames

Kohlhammer and Kotnik,"Systemic Behaviour of Plane Reciprocal Frame Structures."

5 Kohlhammer , "Discrete Analysis–A Method to Determine the Internal Forces of Lattices."

6 Larsen, *Reciprocal Frame Architecture*, 7.

7 Roelofs, "Two-and Three-Dimensional Constructions Based on Leonardo Grids."

8 Kohlhammer and Kotnik, "Systemic Behaviour of Plane Reciprocal Frame Structures."

9 Douthe and Baverel, "Design of Nexorades or Reciprocal Frame Systems with the Dynamic Relaxation Method."

2.4 Design For Assembly

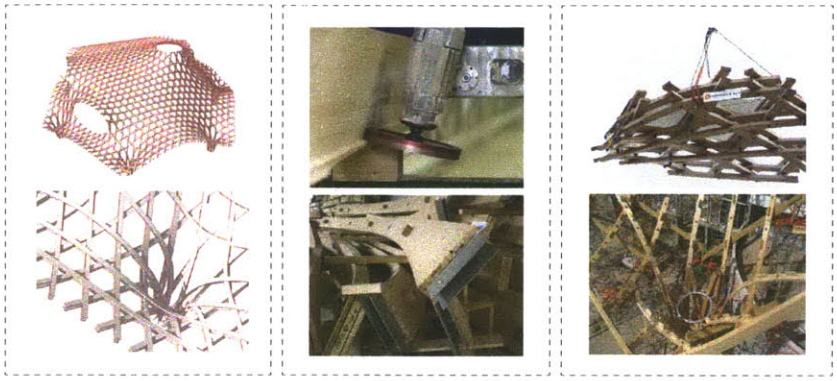


Fig. 2.5
Three Stages of Digital Design and Digital Fabrication Process: Centre Pompidou Metz as Example

http://www.designtoproduction.ch/content/view/75/54/

In the article, Information Master Builder, Branko Kolarevic states that thorough the development of digital processes of production, architects are gaining more control and responsibility in the entire building process. "The new relationships between the design and the built work place more control, and, therefore more responsibility and more power in the hands of architects".[1] Architects are able to move beyond the role of a drafter and participate in the entire production process.

In the digital design and fabrication process, three different stages can be identified -- pre-fabrication, fabrication and post-fabrication. (Fig. 2.5) These three stages should form a loop that information at later stages can be feedback into the previous stages. Through the recent advancement in digital fabrication, the designers have gain more awareness of the constraints of fabrication process; however, the constraints of the post-fabrication process are still often overlooked. In the thesis by Dimitris Papanikolaou, he observes a similar paradox in most of the digital design and fabrication process -- designs are generated with high-end computational methods, but the assemblability of designs are assessed by a manual trial and error process.[2]

---

1 Kolarevic, Architecture in the Digital Age, 57.

2 Papanikolaou, "Attribute Process Methodology: Feasibility Assessment of Digital Fabrication Production Systems for Planar Part Assemblies Using Network Analysis and System Dynamics."

The success of a digital design and fabrication process relies on two major criteria: fabricability and assemblability. If either one of these criteria fails, the entire process fails. This thesis focuses on the post-fabrication process, especially on the evaluation of assemblability. It hopes to bring the post-fabrication assessment to the attention of architects, completing the information loop in the digital design and fabrication process.

3. Geometric Assembly Planning

An interlocking frame is an assembly that each member is supported by each other through global interlocking behavior. Every single member in the assembly has no freedom to move except a few "key" pieces, which are meant to be assembled last and stabilized with possible external connections. The term "global interlocking" means that every piece relies on more than one joint to lock itself in place. The locking or blocking condition between members can be determined by evaluating the geometric contact constraints.

Researchers in robotics invented the non-directional blocking graph (NDBG) and directional blocking graph (DBG) to explain the blocking conditions. [1] These graphs can be easily stored and analyzed using computational methods. However, in the common architectural assemblies, most of the parts have little to no freedom. Instead of using the blocking graph to store blocking directions, I use the free graph to store free directions. This representation allows fewer amounts of data to be stored in the program and faster computation of possible free movements.

In certain assemblies, although none of the individual member can be separated from the assembly, a group of several members may be separated together. These groups can be found by analyzing the strongly connected components (SCC) in the directional block graph. The analysis for free strongly connected components requires far more computation than the analysis of a single free member. It should always be performed after the single member separability test fails,

In this paper, the approach to generate assembly sequence follows the principle of "assembly-by-disassembly."[2]The disassembly algorithm can be decided by recursively applying separability tests of single member and strongly connected component. The whole assembly sequence consists of many steps, where each step removes one member of one strongly connected component. There is often more than one members available to be removed.

---

1 Wilson, "On Geometric Assembly Planning."

2 Le, Cortés, and Siméon, "A Path Planning Approach to (dis) Assembly Sequencing."

3.1 Non-Directional Blocking Graph (NDBG) and Directional Blocking Graph (DBG)

In the PHD thesis written by Randall Wilson, he invented the Non-Directional Blocking Graph (NDBG) and Directional Blocking Graph (DBG) that describe the directions blocked by other objects in contact.[3] These graphs can be directly applied to analyze the interlocking condition of members in an interlocking frame assembly.

The translational movement of an object in two dimension can be represented by a two dimensional vector pointing from the center of a unit circle to a point on the circle. (Fig.3.1) If an object is free to move in all directions without constraints, its NDBG will be empty. On the other hand, if an object is locked in position, its NDBG will be the entire circle.

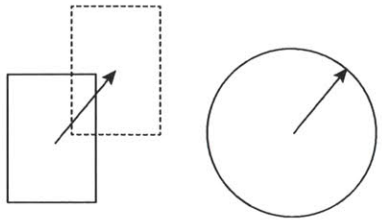


Fig. 3.1
Directional of Movement in two dimension

When two objects are placed side by side with an edge in contact, the NDBG of both objects will become half circles on the opposite side. For example, in Fig.3.2, object A is placed on the left side of object B. The NDBG of object A becomes the right half circle. In all the directions within this half circle, the blocking relationship between A and B is described by the Directional Blocking Graph that has an edge pointing from vertex A to vertex B. This graph read as A is blocked by B. Each of the segment or point in an NDBG has an associated DBG which describes the blocking relationship between objects.

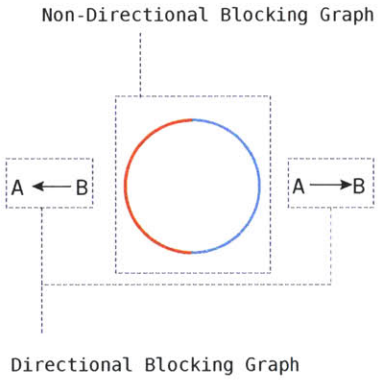


Fig. 3.2
Non-Directional Blocking Graph and Directional Blocking Graph

If more than one edge contact exists between two objects, the NDBG of this two-object assembly can be synthesized from NDBG of each edge contact. If a vertex in DBG does not have any incoming edges, it is free in the directions associated with the DBG.

3 Wilson, "On Geometric Assembly Planning."

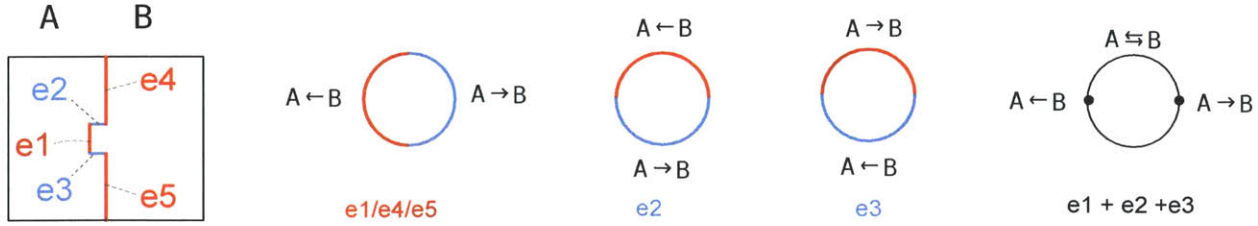


Fig. 3.3
NDBG and DBG in Multiple Edge Contact Condition

3.2 Non-Directional Free Graph (NDFG)

NDBG and DBG can also be applied to determine locking conditions in three dimension. The translational movement of an object in three dimension can be represented by a three dimensional vector pointing from the center of a unit sphere to a point on the sphere. A surface contact constraint in three dimension will be described by the half sphere in NDBG.

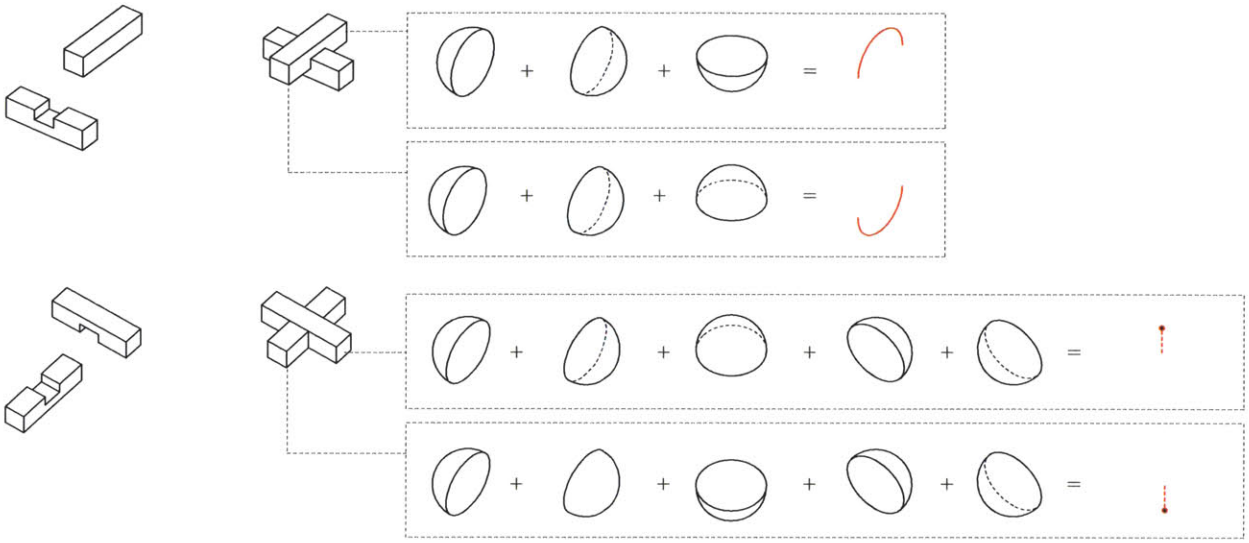
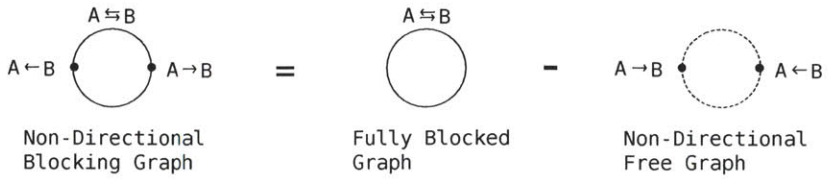


Fig. 3.4
Non-Directional Free Graph

In Fig.3.4, NDBGs of two joint conditions are synthesized from NDBGs defined by their surface contacts. The joint in the top of the diagram results in all the directions being blocked except a half great circle, while joint in the bottom results in all the directions being blocked except a point. Most of the joint conditions in an interlocking frame leave little freedom to the member. I propose that rather than using the blocking direction to describe contact conditions, it is more convenient to use the "free direction" to describe the contact conditions. The free direction denotes the direction of movement that frees the member from its contact constraint. The resulting graph is called Non-directional Free Graph (NDFG).

Fig. 3.5
Non-Directional Free Graph and Directional Free Graph



Using the free graphs not only saves the amount of information stored in the program, but also simplifies the calculation of overall free direction of a single member. The overall free direction of a member with multiple joints can be calculated by boolean intersection operation of NDFGs from all joints. The free graphs can also be conveniently converted back to blocking graphs, (Fig.3.5) which is required for identifying strongly connected components. (section 3.4)

3.3 Six Degrees of Freedom (DOF)

The previous analysis takes into account the translational movement of objects in two dimension and three dimension. However, an object can also have rotational movement which is not described in the analysis. In three dimension, a rigid object will have a total of six degrees of freedom (DOF), where three are translational and three are rotational.

In Fig.3.6, a rigid body R has a point in contact with the plane Pl at point Pt. The normal vector of Pl is $N=(x_n,y_n,z_n)$. The rigid body movement can be described as 6D vector:

$$X=(x,y,z,a,b,c)$$

x,y,z is the translational movement, a,b,c is the Euler angle of rotational movement. The movement of Pt caused by this rigid body motion can be described by a 3D vector

$$d= J * X,$$

J is the Jacobian 3x6 matrix that relates the differential motion of B to the motion of Pt. If $N^T * J * X < 0$ then the motion X violate the contact constraint at Pt. [4]

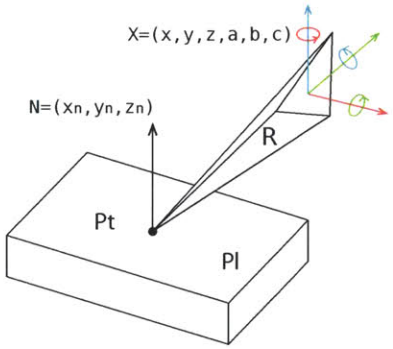


Fig. 3.6
Point Contact Constraint in three dimension

$$\begin{bmatrix} x_n, y_n, z_n \end{bmatrix} \begin{bmatrix} \frac{\partial x_n}{\partial x} & \frac{\partial x_n}{\partial y} & \frac{\partial x_n}{\partial z} & \frac{\partial x_n}{\partial a} & \frac{\partial x_n}{\partial b} & \frac{\partial x_n}{\partial c} \\ \frac{\partial y_n}{\partial x} & \frac{\partial y_n}{\partial y} & \frac{\partial y_n}{\partial z} & \frac{\partial y_n}{\partial a} & \frac{\partial y_n}{\partial b} & \frac{\partial y_n}{\partial c} \\ \frac{\partial z_n}{\partial x} & \frac{\partial z_n}{\partial y} & \frac{\partial z_n}{\partial z} & \frac{\partial z_n}{\partial a} & \frac{\partial z_n}{\partial b} & \frac{\partial z_n}{\partial c} \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ a \\ b \\ c \end{bmatrix} < 0$$

4 Wilson, "On Geometric Assembly Planning."

The calculation of NDBG with six DOF requires much more computation comparing to NDBG with three DOF. Any movement with six DOF can be described as a 6D vector pointing from the center of a hyper sphere to the point on the hyper sphere. Calculating NDBG with 6 DOF results in boolean operations on the hyper sphere in six dimension. Moreover, in previous section, a surface contact constraint directly results in a blocking direction of half sphere in NDBG. If rotational movement is taken into consideration, a surface contact need to be represented by multiple point contacts. Therefore, NDBG of a surface contact needs to be synthesized from NDBGs of multiple point contacts. In an interlocking frame, rotational movements are rarely involved in the assembly. In order to reduce the calculation time, only 3-DOF NDBG is implemented in the software.

3.4 Strongly Connected Component (SCC)

In certain assembly, although none of the member can be separated individually from other members, some members as a group can be separated together. This assembly operation of moving a group of members in the same direction is classified as a non-linear assembly operation. [5] In order to find the non-linear operation in an assembly, the strongly connected components must be identified in Directional Blocking Graph.[6] In a directed graph, a strongly connected component is defined as a group of vertices that every vertex in the group has path to any other vertices. A SCC in a DBG means that these members are locked as a group and need to move together as a whole in the directions associated with DBG. If the SCC does not have any incoming edges, it can move freely in the directions associated with its DBG.

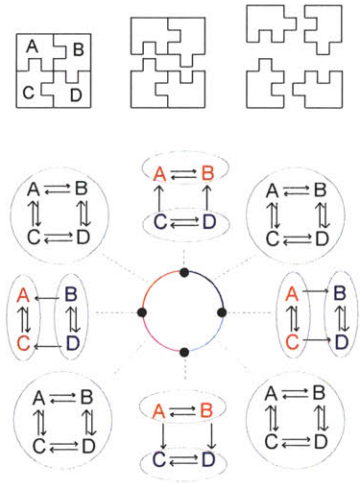


Fig. 3.7
An Example of Strongly Connected Component

For example, in the assembly shown in Fig.3.7, no member can be separated individually. However, in up, down, left and right direction, its DBG can be divided into two SCC. In each of these DBGs, one of the SSC, which is marked in red, does not have any incoming edge. Therefore, these SCCs can be separated from the assembly in the direction associated with their DBG. The Kosaraju's theorem can be used to find SCCs in a directed graph. [7] This algorithm consists of two depth first search, which one is performed on the original graph and the other is performed on the transposed graph.

5 Jiménez, "Survey on Assembly Sequencing."

6 Wilson and Latombe, "Geometric Reasoning About Mechanical Assembly."

7 Cormen et al., Introduction to Algorithms, 615.

3.5 Disassembling the Diagonal Burr Puzzle

The Diagonal Burr Puzzle (Fig.3.8) is another example of assembly that can only be disassembled by strongly connected component analysis. The first step of disassembling the puzzle is to determine the contact condition between members. These pairwise contact conditions can be stored in a matrix. (Fig.3.9) In this particular puzzle, every piece in the puzzle is in contact with one another. The matrix is filled with True value, indicating that contacts exist between every pair of members. For every pairwise contact, the associated NDFG is calculated. (Fig.3.9) After combing all thirty NDFGs into one NDFG, all the intersections are extracted as directions of interest. These directions are the ones that allow most freedom between members and are potentially the separation directions. For every direction of interest, a DBG needs to be built.

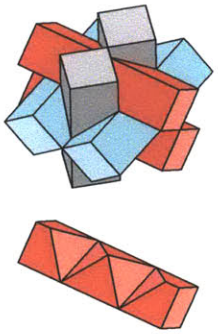


Fig. 3.8
The Diagonal Burr Puzzle Consists of Six Identical Members

	1	2	3	4	5	6
1		T	T	T	T	T
2	T		T	T	T	T
3	T	T		T	T	T
4	T	T	T		T	T
5	T	T	T	T		T
6	T	T	T	T	T	

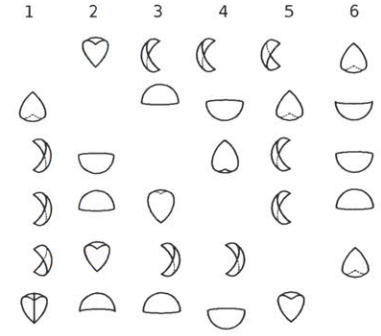


Fig. 3.9
Contact Matrix of the Diagonal Burr Puzzle (Left) NDFG for Each Contact

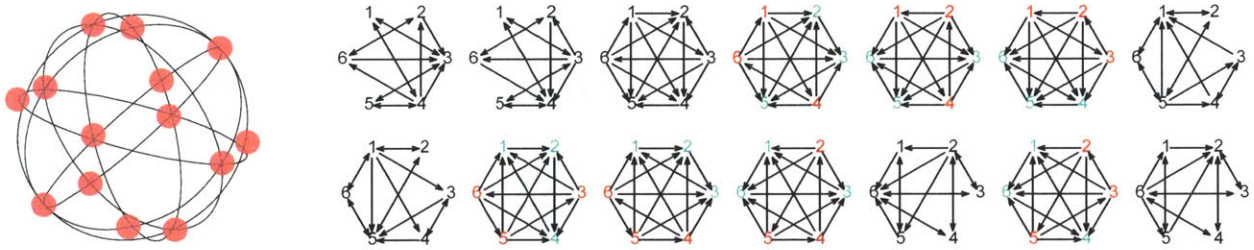
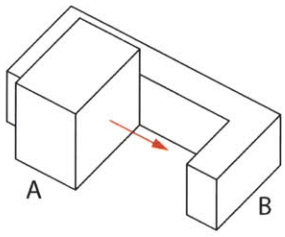


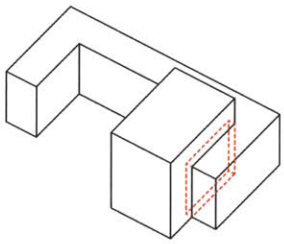
Fig. 3.10
Strongly Connected Component Analysis for 14 Directions of Interest

In order to obtain the DBG of a single direction, that direction will be checked against all thirty NDFG. If the direction is within the NDFG of a constraint, that constraint will be removed from the fully connected graph. Fig.3.10 shows the DBG of all fourteen directions of interest, where eight of them have strongly connected components in the graph. Therefore, eight options are available for the first step to disassemble a Diagonal Burr Puzzle.[8] After the first step, only single member separability test is needed to disassemble the puzzle. (Fig.5.7)

8 Coffin, Geometric Puzzle Design, 81.



local infinitesimal movement



collision in extended movement

Fig. 3.11
Infinitesimal Movement and Extended Movement

3.6 Extended Motion

NDBG and DBG are designed to analyze the local infinitesimal movement of members in an assembly. It does not take into account the extended movement of parts after it is freed from the constraint. In Fig. 3.11 the member B is an U shaped element. Although the NDBG provides the direction to separate A form B in the top state, it does not guarantee that A will not collide into B after it moves for a certain distance as shown in the bottom state. In the paper, *On Geometric Assembly Planning*, the author tries to solve the problem with an Extended Blocking Graph; however, this method does not provide a clear instruction of how an obstruction-free path can be generated. [9] Recently research papers have adopted the path planning approach to disassembly automation.[10] These studies are usually based on the Rapid-exploring Random Tree (RRT) algorithm to search for possible disassembly path. RRT is a stochastic process of generating movement with collision detection.

In an interlocking frame disassembly process, the local infinitesimal movement can usually be extended without collision. Nevertheless, a collision detection mechanism can be embedded into the process. The collision detection between geometry is usually a costly operation in computation. A fast collision detection algorithm is required to speed up this process. Fortunately, in an interlocking frame system, each member is derived from an rectangular box geometry, which cores ponds to the oriented bounding box (OBB) used in computer graphics. Therefore, the Separating Axis Theorem[11] for oriented bounding box can be directly applied to the collision detection of members in an interlocking frame. Moreover, in large assemblies, various spatial division algorithms, such as octree or k-d tree, will reduce the number of intersection queries significantly, resulting in a much faster collision detection process.

3.7 Assembly Sequencing

In this thesis, the assembly sequence is obtained by reversing a successful assembly sequence. This approach is referred to as assembly-by-disassembly. In order to generate an assembly sequence, the final state of the assembly needs to be given. This final state of assembly is analyzed for possible disassembly process. If a successful

9 Wilson, "On Geometric Assembly Planning."

10 Le, Cortés, and Siméon, "A Path Planning Approach to (dis) Assembly Sequencing."

11 Huynh, "Separating Axis Theorem for Oriented Bounding Boxes."

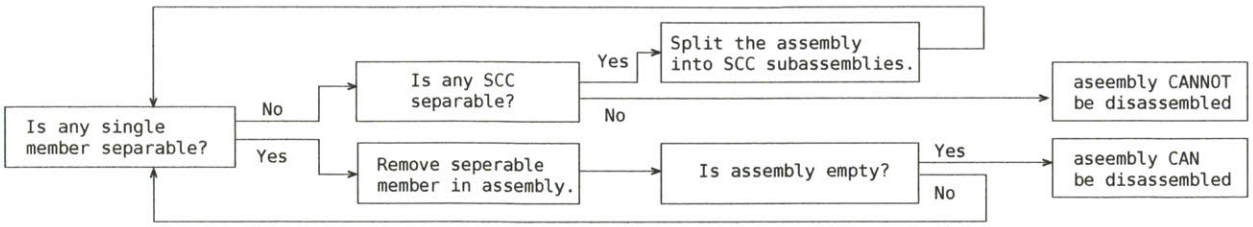


Fig. 3.12
The Disassembly Procedure

disassembly sequence is found, it is reversed to obtain a assembly sequence. The disassembly procedure can be described by Fig. 3.12. This procedure recursively analyze and remove parts that can be separated from the assembly. In the disassembly process, there is usually more than one part that can be removed at a single step, and one of these parts is chosen to be removed first. These options can be represented as an AND/OR tree structure.[12] If the assembly process does not take into account the removal of strongly connected components, the tree can be simplified to an OR tree, which is the most generic tree data structure. (Fig.3.13)

12 Jiménez, "Survey on Assembly Sequencing."

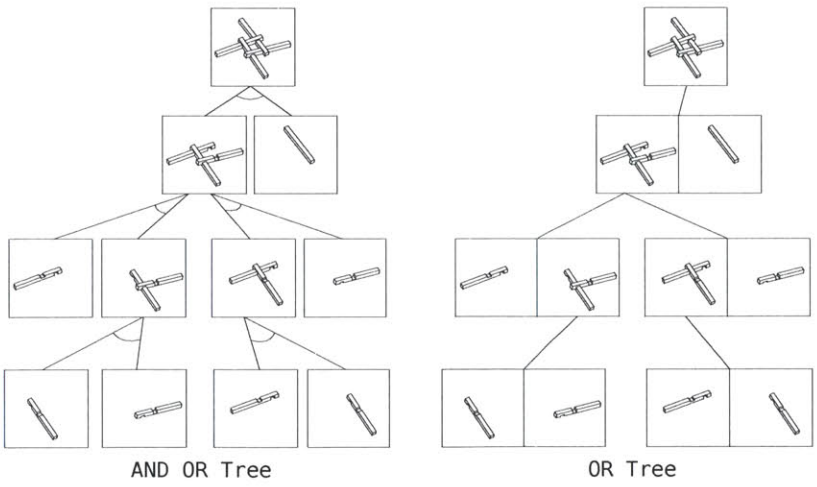


Fig. 3.13
Assembly Sequence as a Tree Structure

4. Designing Interlocking Frames

The methodology to design an interlocking frame can be characterized with five different stages: original grid, 2D interlocking pattern, 3D interlocking pattern, 3D member configurations, and assembly sequence. The step between each stage will be introduced sequentially in this chapter. In this thesis, a new process to generate interlocking pattern is invented. Given any 2D grid pattern, this process will be able to transform the 2D grid pattern into a 3D interlocking member configuration.

In this five-stage process, the most important step is to create the appropriate joint configuration from the 3D member configuration and generate assembly sequence. Because not every joint configuration would guarantee an assembly sequence, the genetic algorithm is used to optimize the joint configuration that result in a successful assembly sequence. This procedure of finding design solutions follows the structure of the generate-and-test process.[1]

In large assemblies, the Genetic Algorithm becomes ineffective and performs similar to a random searching process. I propose a recursive genetic algorithm process that is specifically designed to find the appropriate joint configuration for larger assemblies.

1 Mitchell, "The Logic of Architecture," 180

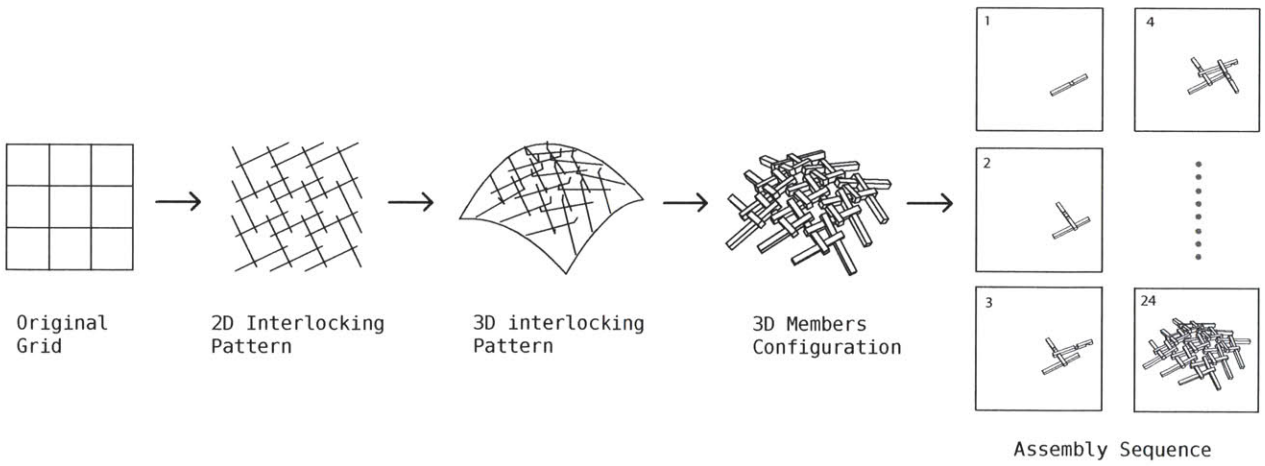


Fig. 4.1
Steps to Construct Interlocking Frames

4.1 From 2D grid to 3D interlocking Pattern

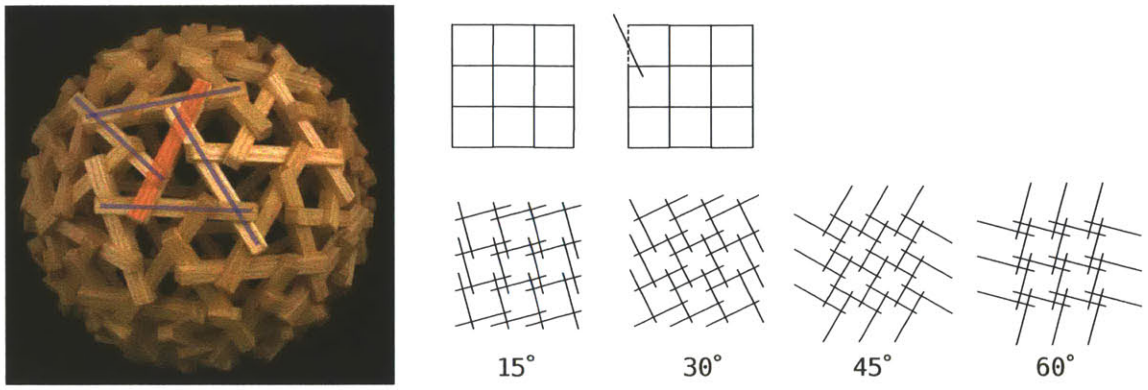


Fig. 4.2
Pattern Study of Existing Interlocking Frames

Architects uses grids to produce systematic organization of space. The methods of generating different grids are well documented in the thesis by Ari Kardasis.[2] In graph theory, these grid can be categorized as linear straight graphs, where all line segments are break at the intersections.

By observing the patterns in existing interlocking frames, we can discover that the number of edges joining at the same vertex is limited to two. (Fig.4.2) In a general grid pattern, the number of edges joining at a vertex can be arbitrary. If the appropriate rotation and extension is applied to every line segment in the graph, the resulting graph becomes a 2D interlocking pattern. (Fig.4.2) The rotation of each member "opens up" the intersection at the vertex, and transform a vertex with N connections into a polygon with N edges. All the intersections will be consists of two edges meeting at a single vertex. (Fig.4.4)

2 Kardasis, "The Soft Grid."

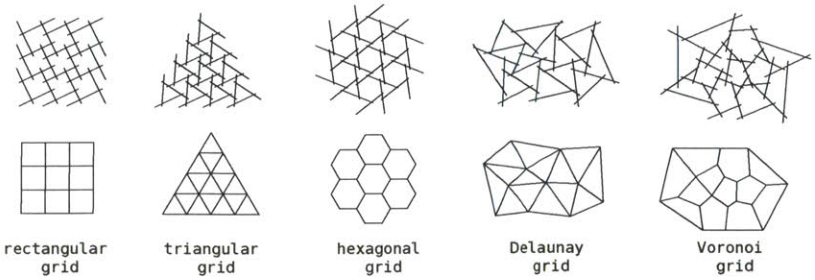


Fig. 4.3
Interlocking Pattern From Regular and Irregular 2D Grids

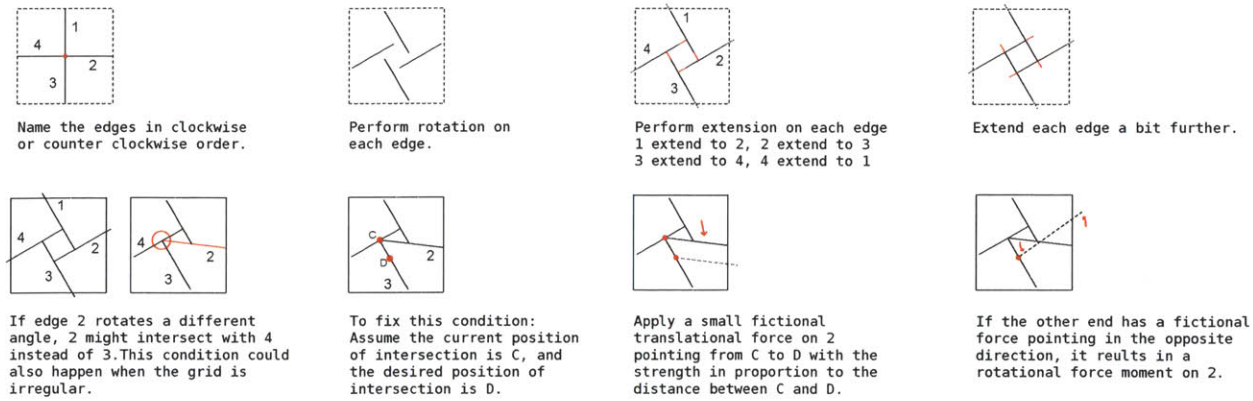


Fig. 4.4
Algorithm to Construct Interlocking Pattern from 2D Grid

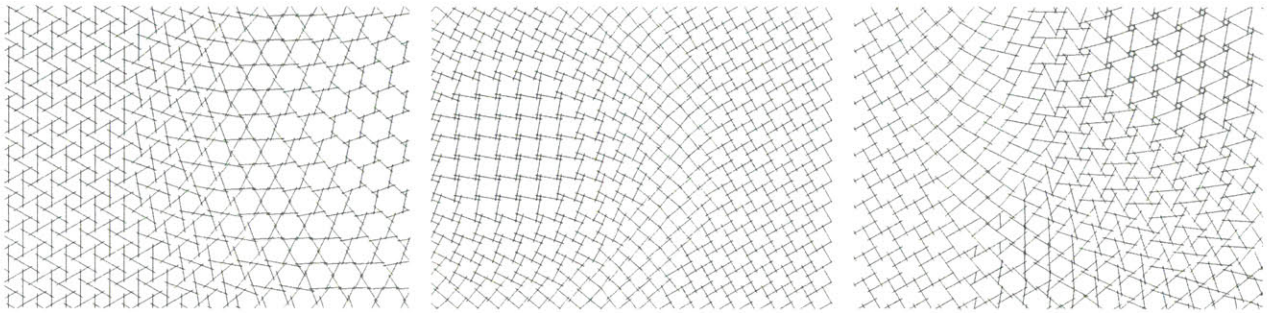


Fig. 4.5
Three Examples of Transformational Interlocking Grid

This process can be applied to arbitrary grid pattern, which can be regular and irregular. (Fig.4.3) The rotation and center of rotation can also be parameterized to produce an transformational pattern that could respond to different designer intention or performance criteria. (4.5) The way of generating grids is itself beyond the scope of this paper, but more information on generating grids can be found in the book,*Grid Index*[3].

Sometimes the pattern will produce undesired intersection condition near the vertices. A iterative method that introduce a fictional force on members are proposed to fix the problem. Other possible fixes can be found in the paper by Goto et al.[4]

3 Nicolai, Grid Index.
4 Goto, Ryota, and Takeshi, "Rokko Mountain Observatory."

4.2 2D Interlocking Pattern to 3D Interlocking Pattern

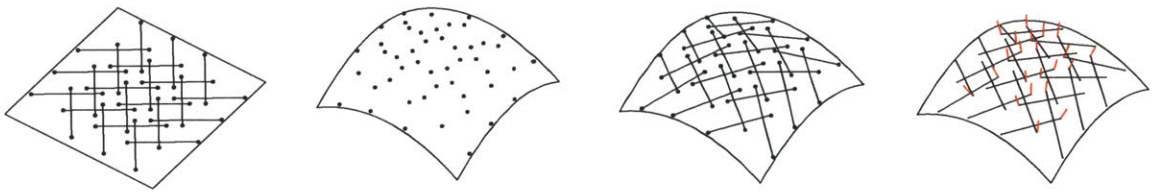


Fig. 4.6
From 2D Interlocking Pattern to 3D Interlocking Pattern

This step transfers a two dimensional flat interlocking pattern into a three dimensional interlocking pattern on a free-form surface. (Fig.4.6) The two dimensional pattern is first placed within a flat surface, where each end point of line segment is evaluated for its UV coordinates. These points are then mapped onto the desired free-form surface according to their UV coordinates. Once the points are mapped, each pair of end points is connected with a straight line. The point on the surface that is closest to the midpoint of the straight line is evaluated for its surface normal. This normal becomes the guide to orient members during the next step.

4.3 Member Placement in 3D

The members used for interlocking frames are assumed to have a rectangular section. In order to place these members in three dimension, the three coordinates axes need to be identified. The Frenet-Serret Frame in differential geometry can be applied to the member placement: the tangent equals the vector point from one end to the other; the binormal equals to the tangent cross surface normal; the actual normal equals the binormal cross tangent. The tangent, normal and binormal vectors becomes the x,y, and z axis vector for member placement. (Fig.4.7)

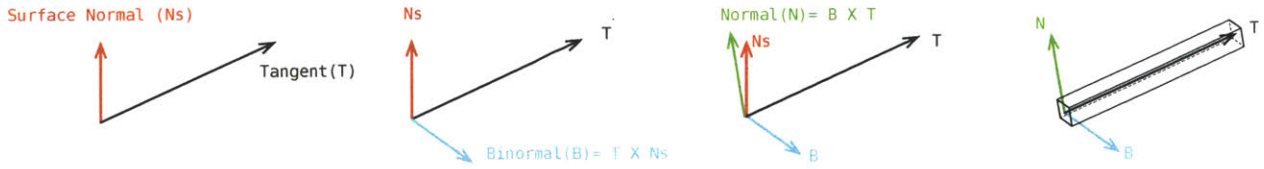
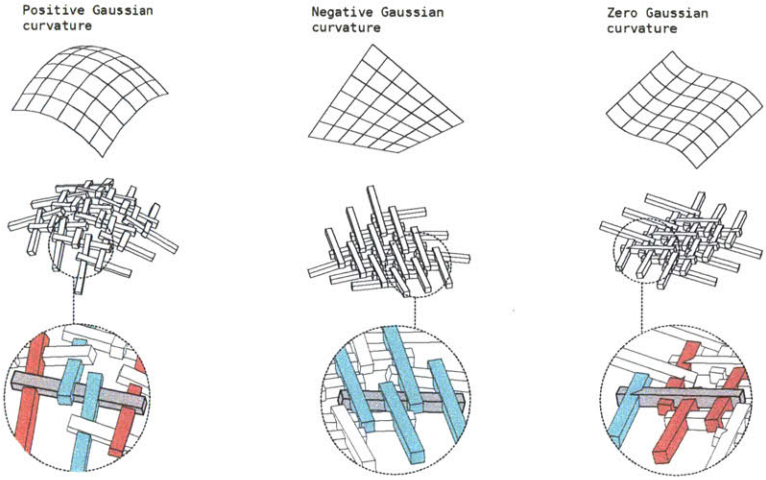


Fig. 4.7
Placing of Members on the 3D pattern

Fig. 4.8
Intersection Conditions Between Members



The surface to be projected on can have different Gaussian Curvature. (Fig.4.8) When the Gaussian Curvature is positive, the intersection condition usually guarantees reciprocity between members. When the Gaussian Curvature is Negative or zero, the intersection condition is fairly inconsistent in terms of preserving reciprocity, and will also result in difficulties for fabrication.

In order to identify these difficulties in fabrication, the intersection conditions need to be analyzed. This can be done by projecting the profile of one member to the coordinates frame of the second member.(Fig.4.9) Depending on the number of vertices of the second member contained in the profile of the first member, six different conditions is established. Only two of these conditions fulfill the fabrication constraints, which will be introduced in the next section.

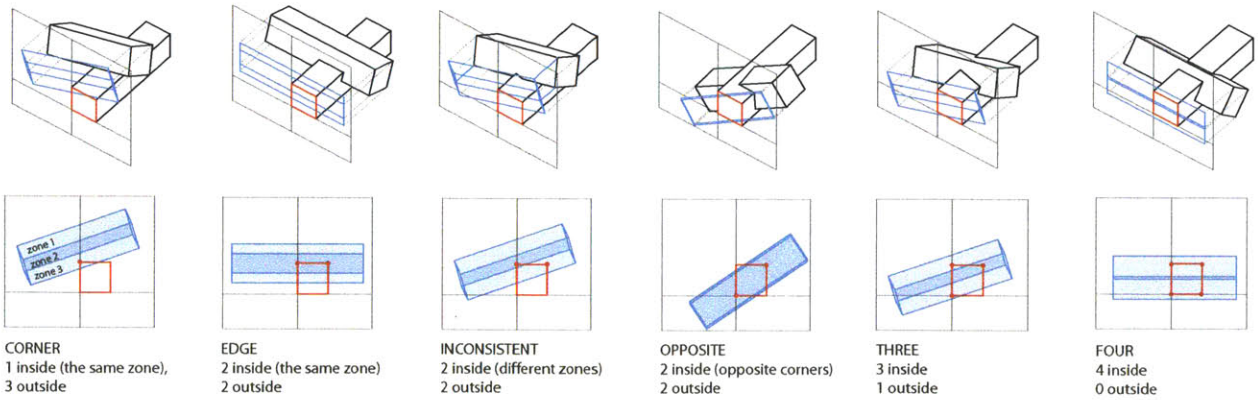


Fig. 4.9
Intersection Conditions Between Members

4.4 Fabrication Constraints

The tool for fabrication is set to a five axis milling machine with flat end mill due to the increasing popularity of multi-axis milling tool in architectural applications. The appropriate milling methods is end milling and flank milling, where the bit orientation is either perpendicular or parallel to the intended milling surface. The trajectory of the bits will create a clearance zone, which should not intersect with the geometry to be milled. (Fig.4.10)

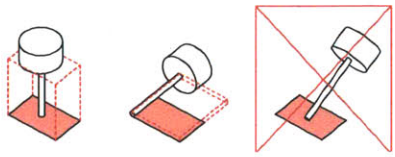


Fig. 4.10
End Milling and Flank Milling

The geometry to be milled can be simply categorized to two conditions: inner cuts and through cuts.(Fig 4.11) While inner cuts produce edges and vertices that does not extend to the boundary surface, through cuts produce edges that always extend to boundary surface. The inner vertices cannot be milled accurately because the end bits cannot accurately reach these spots without removing excessive material around them. If the dihedral angle between the bordering surface at the inner edge is less than 90 degrees, this inner edge will also become unreachable for end bits. Therefore, through cuts should always be enforced in the assembly.

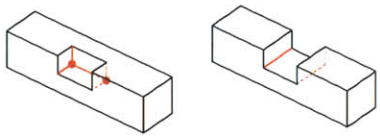
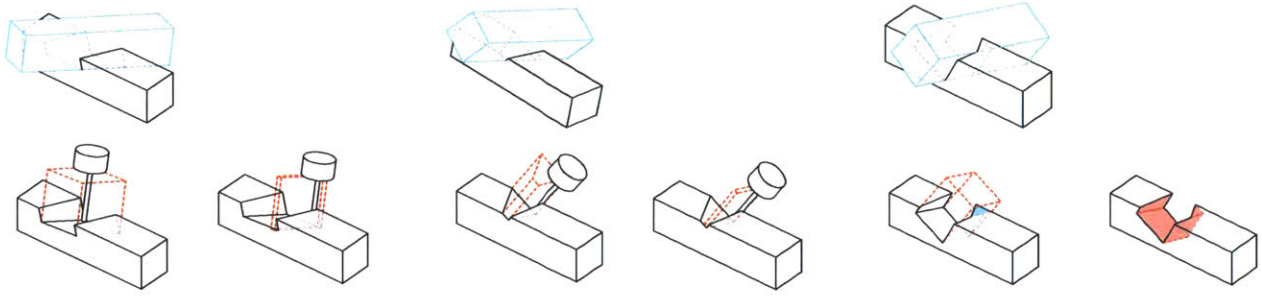


Fig. 4.11
Inner Cut and Through Cut



EDGE intersect condition.
The geometry can be milled accurately.

CORNER intersect condition.
The geometry can be milled accurately.

All other intersect conditions
The geometry CANNOT be milled accurately.

Fig. 4.12
Fabricability of Different Intersection Conditions

After the through cut condition is enforced, the intersections mentioned in the previous section can be analyzed using the clearance zone created by the end bits. The EDGE and CORNER intersection conditions will always result in geometry which will not clash with the clearance zone, while all the other four intersection conditions will. (Fig.4.12)Thus, if any of these four intersections happen in the configuration, the members which are involved in the intersection should be moved to the proper location that will result in EDGE and CORNER intersection condition.

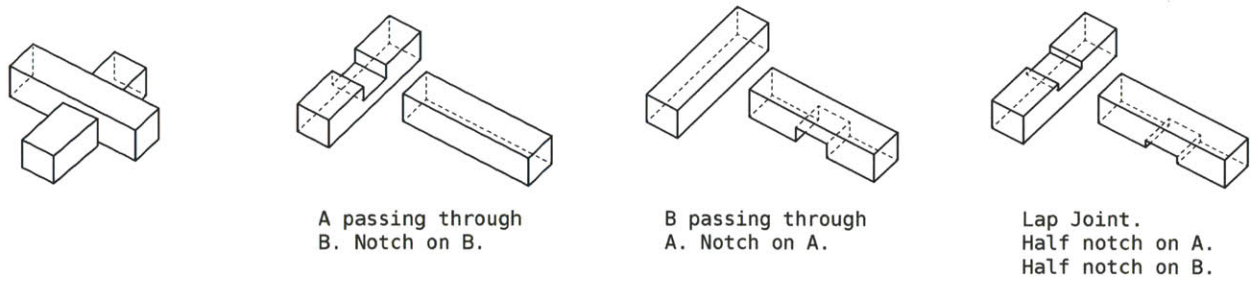
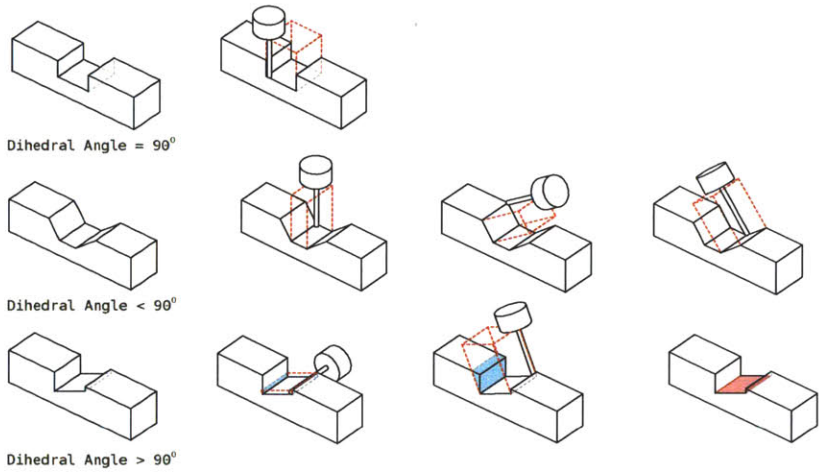


Fig. 4.13
Three Joint Configurations

In the process of the genetic algorithm in section 4.5, three possible joint configurations are defined at each joint. (Fig.4.13) These three joint configurations are chosen because they satisfy the fabrication constraints and produce distinctively different NDBGs. The lap joint condition, which consists of notches on both members, requires further analysis to fulfill the fabrication constraints.

Fig. 4.14
Dihedral Angle and Fabricability



In through cut conditions, the fabricability of the geometry is still limited by the dihedral angle between surfaces. When the dihedral angle between two surfaces are less than 90 degrees, it will produce edges that cannot be reached by the end mill. It is necessary to keep all the dihedral angles in the geometry equal or greater than ninety degrees. In the lap joint configuration, the boolean intersection volume between two members need to divided into two half by a dividing surface. These two halves will become the volume removed by the end bits in the fabrication process. The dihedral angle between the dividing surface and boundary surfaces need to be exactly ninety degrees but not greater. (Fig.4.15) If it is greater than ninety degrees at one edge, the angle at the opposite edge will be less than ninety degrees, which violates the constraint.

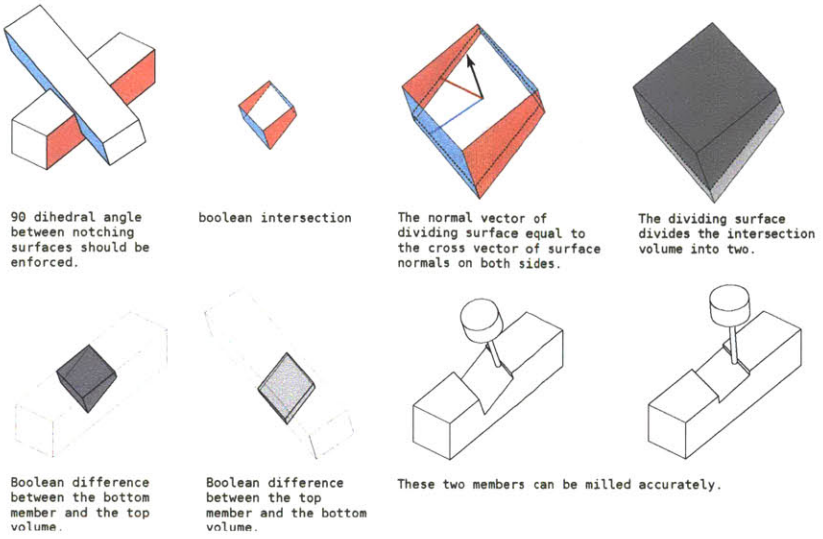


Fig. 4.15
Creation of Lap Joint

4.5 Searching Joint Condition using the Genetic Algorithm (GA)

The 3D member configuration is a collection of members located in the three dimensional space, where every intersection results in a joint condition. There are three possible joint conditions defined at each joint as described in section 4.4. In this particular example, the assembly is a collection of 24 members and 36 joints, producing 3^{36} possible variations of joint configuration. Of course, most of these configurations will not ensure a successful assembly sequence.

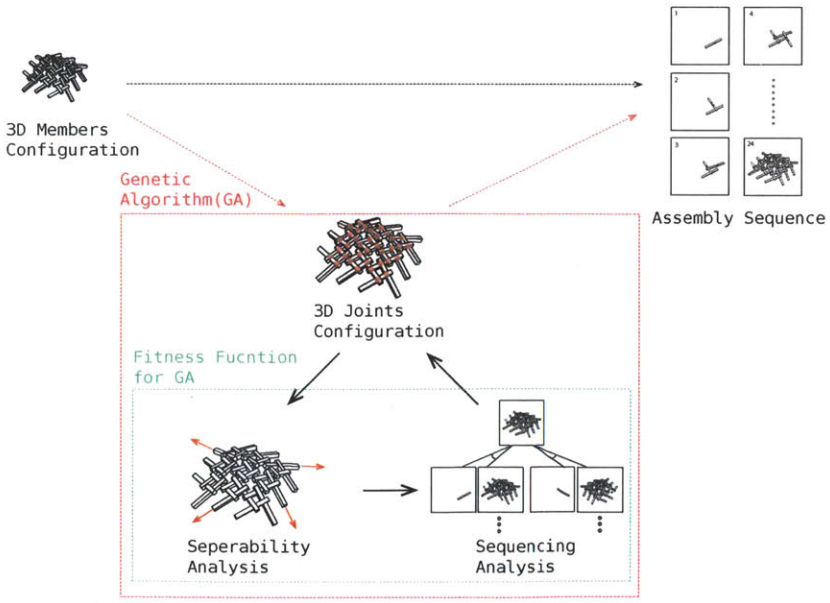


Fig. 4.16
From 3D Member Configuration to Assembly Sequence

The genetic algorithm(GA) is used to search for the joint configurations that will produce a successful assembly sequence. GA belongs to the class of optimization algorithms which search the multi-dimensional solution space for an optimized goal. Other algorithms such as Simulated Annealing, Particle Swarm Optimization, and Ant Colony Optimization can be also applied to this problem.

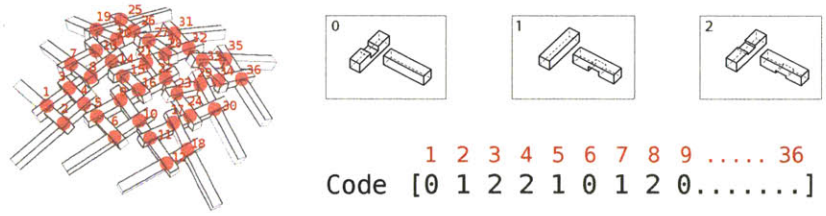


Fig. 4.17
Coding the Joint Configuration for Genetic Algorithm

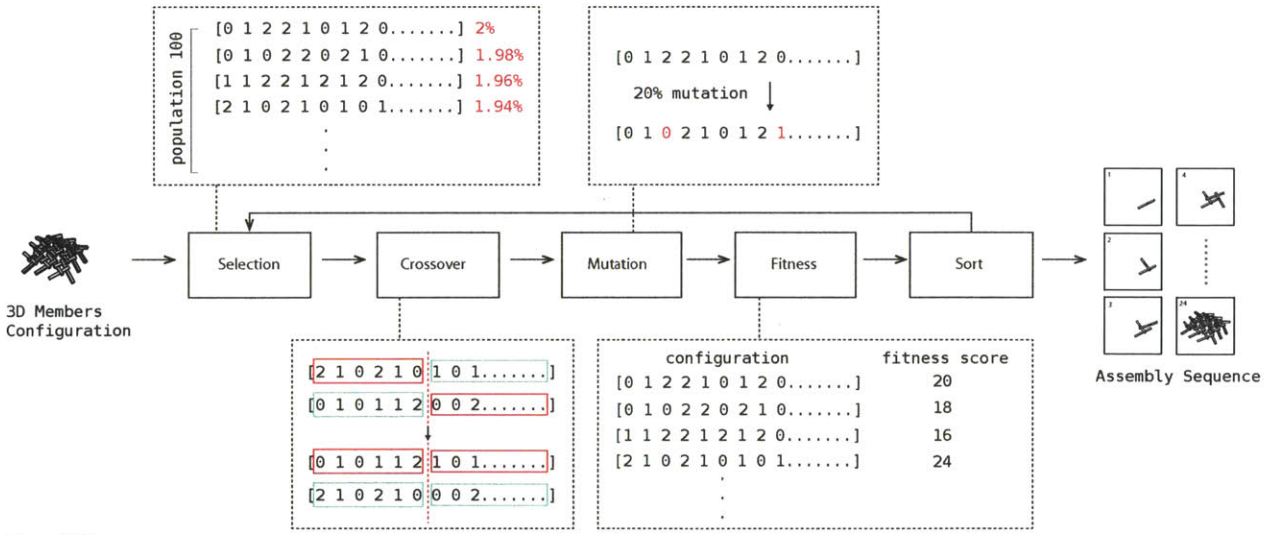


Fig. 4.18
The Genetic Algorithm Procedure

Each joint configuration is coded as an array with 36 integers, which every integer represents one of the possible three joint conditions. The generated codes are processed with the standard genetic algorithm procedure, which includes selection, crossover, mutation, fitness, and sort. (Fig.4.18)The fitness function is the function that describes the goal for optimization. This function is customized based on the intention of the designer. In the interlocking frame problem, these intentions may be: maximizing the number of members that can be disassembled, minimizing the members that needs to be supported during the assembly process, or minimizing the "key" pieces, members that are not locked, in the assembly.

4.6 Defining the Fitness Function for GA

The fitness function for assembly analysis consists of three parts: separability analysis, sequencing analysis and storing cumulative fitness score. An assembly is first analyzed for possible separable options using Non-directional Free Graph, and these options are expanded as a sequence tree. Based on the goal of optimization, different algorithms, such as greedy, best first search, A* can be used to search the tree. The fitness score of the function is cumulatively updated through the searching process. If our goal is maximizing the number of members that can be disassembled, the options can be selected arbitrarily, since all the path will lead to same number of members that can be disassembled. On the other hand, if the goal is minimizing the members that need to be supported during the assembly process, which is the same goal as minimizing the number possible free options at each level, best first search algorithm should be utilized to find the disassembly sequence and the fitness score. (Fig.4.19)

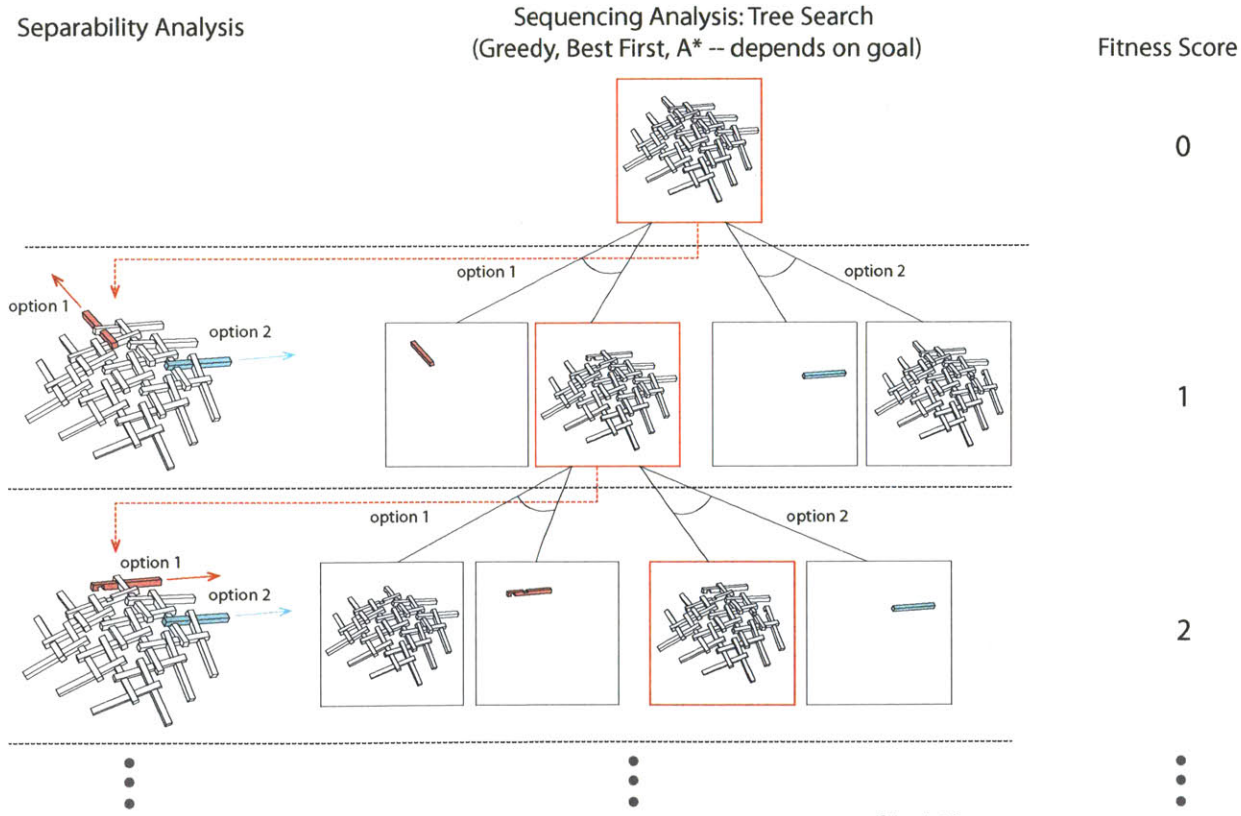


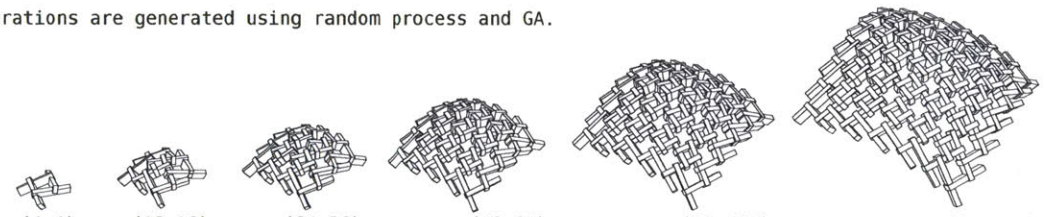
Fig. 4.19
Fitness Function for GA

4.7 Recursive Genetic Algorithm for Large Assembly

The effectiveness of genetic algorithm (GA) is compared with the random searching process for different size of assemblies based on rectangular grids. The chart compares if the methods can successfully find a joint configuration that allows for assembly sequence, the average fitness score and the best fitness score. The fitness score is defined as the number of members that can be separated from the assembly.

Each of the method generates ten thousands new joint configurations for these different size of assemblies. The random process fails to find any successful joint configuration in the 24-member assembly, while the GA fails to find a successful joint configuration in the 48-member assembly. GA always results in a better average fitness; however, in the larger assembly the percentage of members being disassembled drops significantly. (Fig.4.20)

10000 joint configurations are generated using random process and GA.



(members,joints)	(4,4)	(12,16)	(24,36)	(40,64)	(60,100)	(84,144)
Random						
success rate	57.08%	3.09%	0%	0%	0%	0%
average fitness	2.28	2.97	3.61	4.25	5.14	6.17
best fitness	4	12	20	27	24	24
GA						
sequence found?	Yes	Yes	Yes	No	No	No
average fitness	2.90	5.02	5.60	6.29	7.25	8.25
best fitness	4	12	24	32	34	31

Fig. 4.20
Comparing GA and Random Search

In the 84-member assembly, the best solution disassembles only 31 members. As the number of members increase in the assembly, the successful solution in the solution space becomes sparse. The crossover and mutation in GA becomes less effective and performs similar to a random searching process. Even if the number of generations in GA increases, the average fitness and best fitness do not increase significantly. Therefore, I propose the method of using GA recursively to find at least one successful solution. The method starts the same way as the usual GA, but after the average fitness and best fitness does not show improvement in twenty generations, the GA is halted. The disassembly sequence of the best solution

is operated on the assembly, separating members until the whole assembly is locked. Then a new GA process is initiated on this reduced-size subassembly. This process repeats between applying GA to find the best partial solution, disassembling the assembly into a subassembly, and applying GA to the subassembly. This recursive process stops when the whole assembly is disassembled. (Fig.4.21)

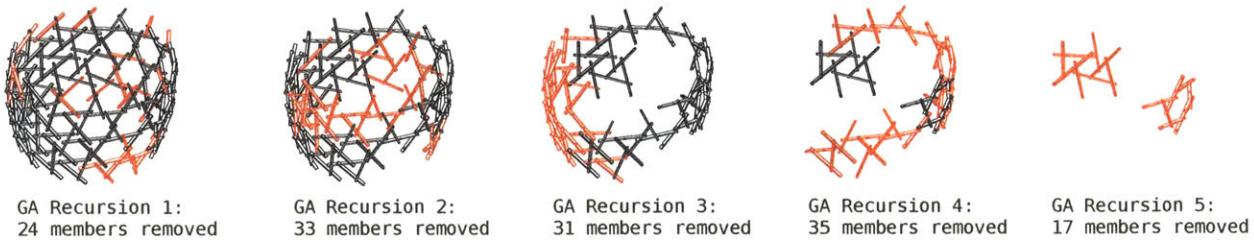


Fig. 4.21
Recursive GA Process

Investigating the methods to improve the effectiveness of GA is one the possible future research directions. More details on how to improve the generic GA process can be found in different books and papers. However, I would like to point out a few possibilities in the context of the interlocking frame assembly problem. The first possible improvement is sorting the joints according to connectivity before coding in the GA process. In my current program, the joints are sorted according the z value of Cartesian coordinates, which does not reflect any relationship about the joints. Instead of being sorted by the Cartesian coordinates, the joints are sorted by a breadth first search or a depth first search to provide some logic in structuring the code of GA. The second possible improvement is altering the crossover function. The current crossover function selects a random crossover point in the code, and selects the items in the code array that are before the crossover point for switching. These items represent joints in the assembly. Instead of being selected by the sequential order in code, the joints could be selected by the proximity of connectedness. In this new procedure, a random joint is selected and the a random number of neighboring joints are grouped for crossover. These two methods may help to improve the effectiveness of GA in optimizing the joint configuration.

5. Software Implementation

The deliverable for this thesis is a software program that helps designer to design interlocking frames. This software includes a pattern generator, a joint generator, a sequence player and a disassembler. It is implemented as a plugin program in the Rhinoceros 3D[1] modelling environment, with the exception of the pattern generator, which is a definition file in Grasshopper [2]parametric modeler. All of the relevant source code will be included in the appendix.

The pattern generator corresponds to the process from 2D grid to 3D interlocking pattern in the methodology chapter. Because this process is an iterative design process, it is more appropriate to be implemented in the parametric modeling environment.

The joint generator uses the genetic algorithm to generate the joint configuration and assembly sequence from a 3D member configuration. It also records the assembly sequence in a text file format that is similar to the G-Code used for computer numerical control system.

The sequence player can read and play the recorded assembly sequence generated by the joint generator. It is a useful tool that provides 4D assembly instructions for assembly or construction.

The disassembler is tool that generates disassembly sequence for models that defined by planar surface contact constraints. This tool can be used to evaluate the assembly sequence of complex geometric assemblies other than interlocking frames.

---

1 Rhinoceros NURBS modeling for Windows, http://www.rhino3d.com/

2 Grasshopper- Generative Modeling for Rhino, http://www.grasshopper3d.com/

5.1 Pattern Generator

The pattern generator uses a 2D pattern and a surface as input, and produces a 3D interlocking pattern as output. Two different versions of pattern generator are implemented: one applies a constant rotation on every member; the other applies different degrees of rotation on members based on their proximity to reference points or curves. Both versions are implemented in the Grasshopper environment using a customized python scripting component. The software also provides a "intersection checker", which allows the user to input the dimension for the members and identifies the problematic intersection conditions in the assembly. The user is expected to fix these conditions manually before proceeding to the Joint generator.

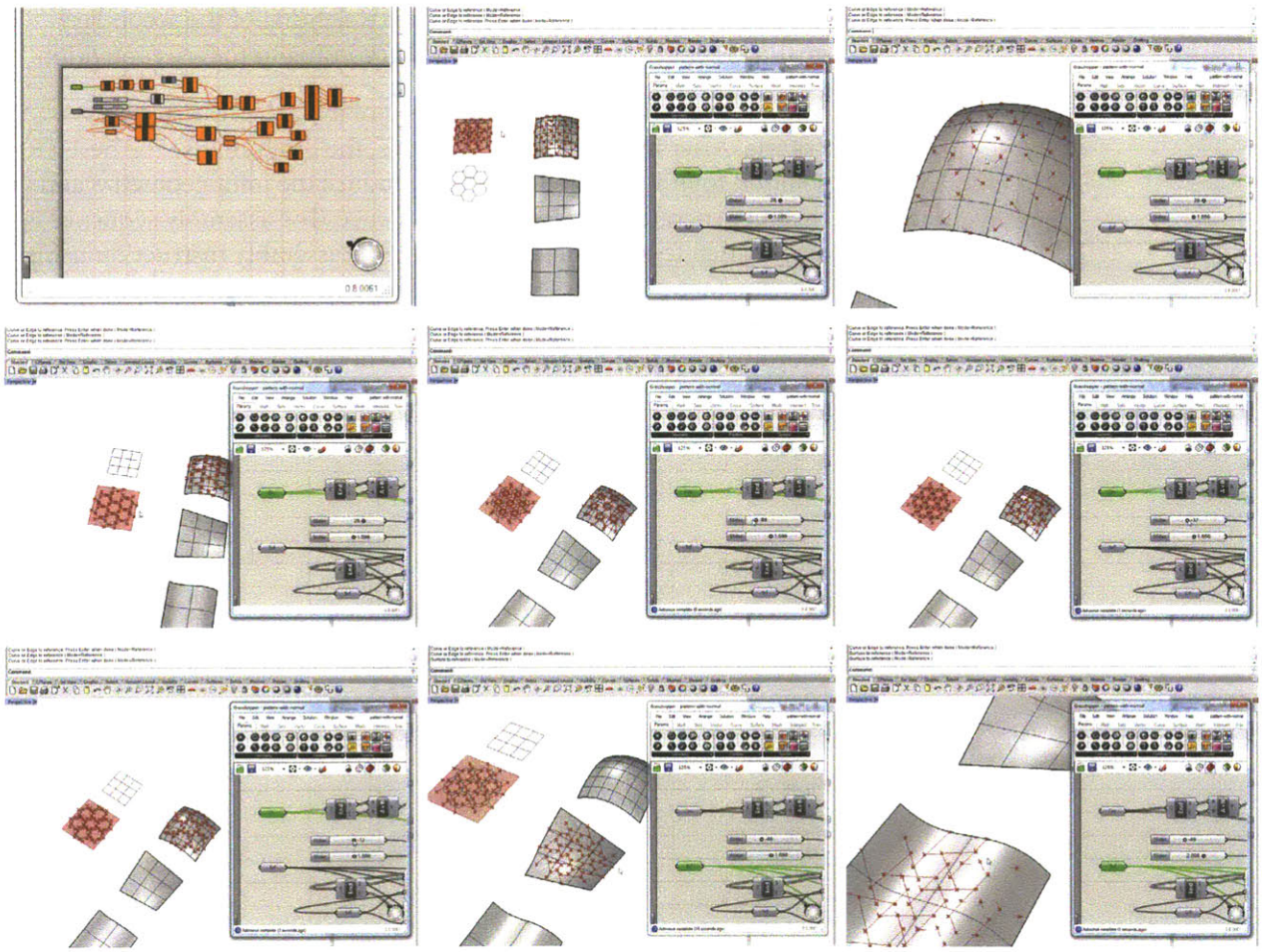


Fig. 5.1
Screenshots of Pattern Generator at Work

5.2 Joint Generator

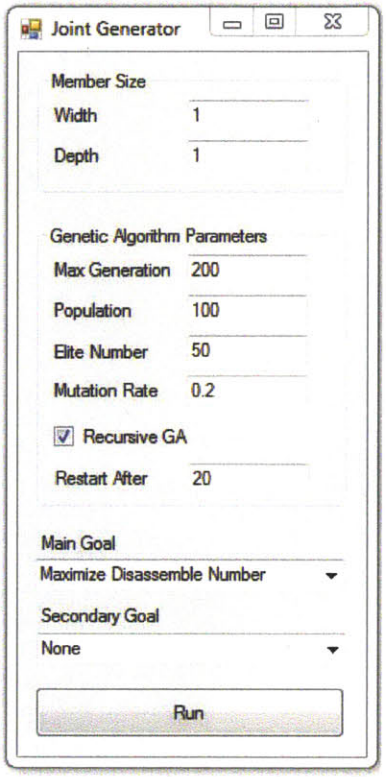


Fig. 5.2
Joint Generator Interface

In the joint generator, the input is a 3D interlocking pattern produced by the pattern generator. The user is able to set the dimension of members, parameters of genetic algorithm, and goals for optimization. The joint generator is also capable of solving large assembly using the recursive GA algorithm in section 4.7. The goal or fitness score for GA can be chosen between three different options: maximizing the number of members that can be disassembled, minimizing the members that needs to be supported during the assembly process, or minimizing the "key" pieces in the assembly. Both of the minimizing goals are implemented as a penalty, a negative value, in the fitness function. The main goal and secondary goal represents the weighting of linear combination between these goals, where the main goal is three times the weighting of the secondary goal. This weighting can be changed according to the user intention.

The joint generator will run the GA until it reaches the stopping criteria. After finishing the GA process, the program use the best solution found through GA to first produce the joint geometry, and then to generate the disassembly sequence. The assembly sequence is also written to a text file that provides 4D assembly instruction using the sequence player in section 5.5. Each member will be numbered according to the order of assembly and oriented to the world coordinate system for the convenience of fabrication.

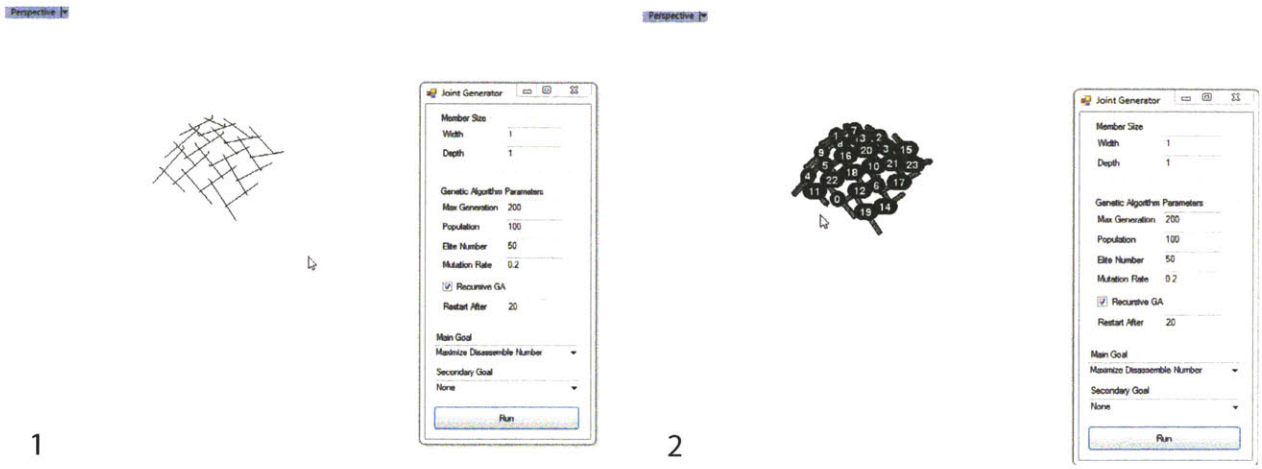


Fig. 5.3
Screenshots of Joint Generator at work

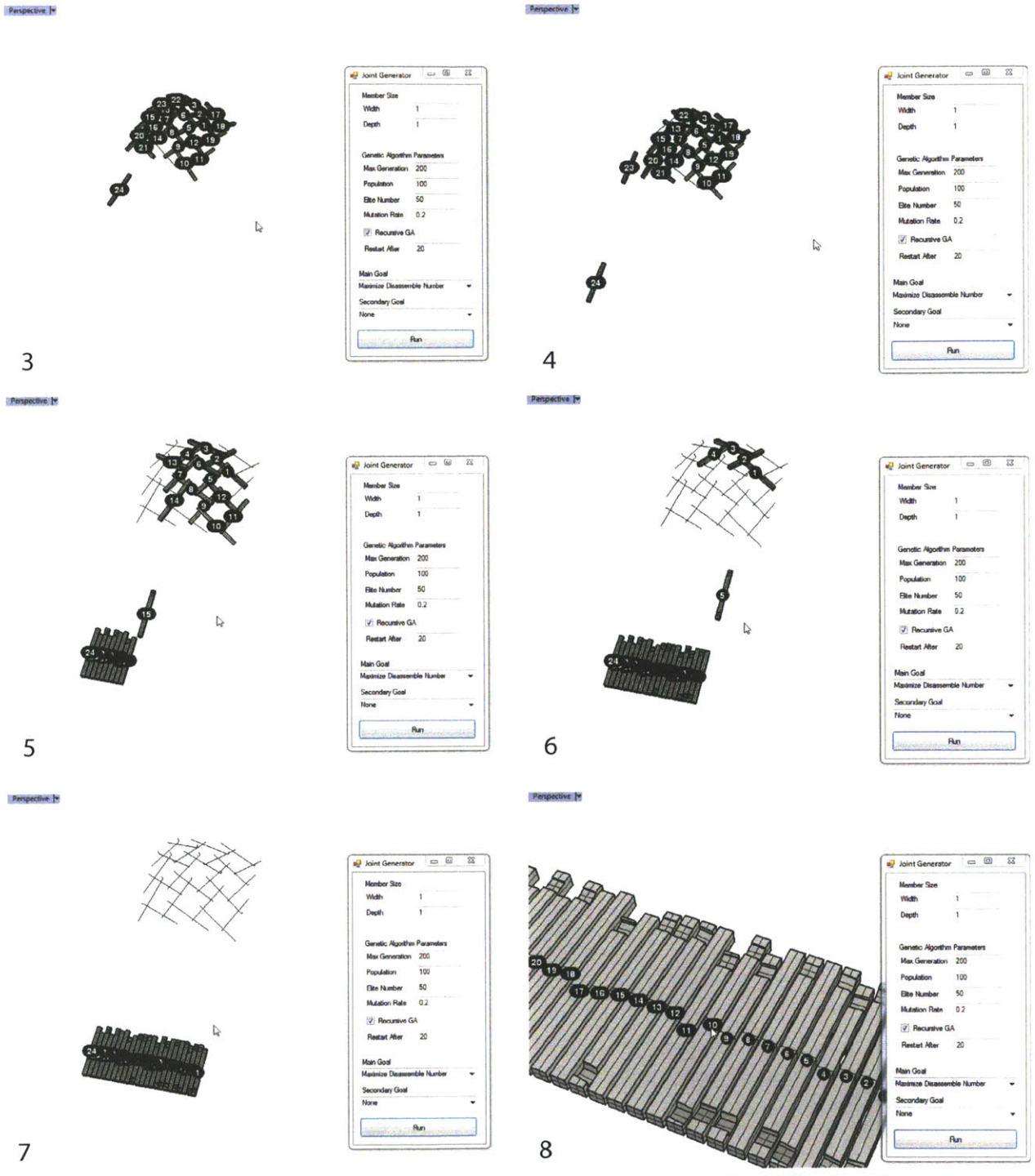


Fig. 5.3 (continued)
Screen Shots of Joint Generator at work

5.3 Sequence Player

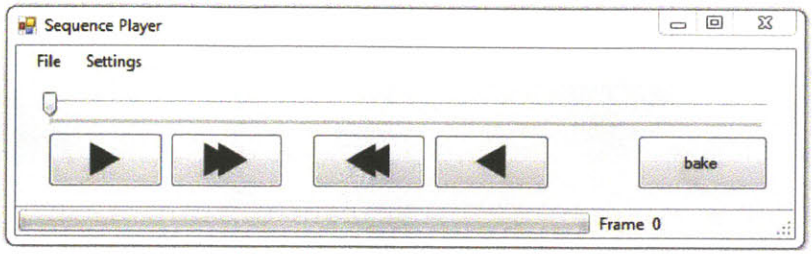


Fig. 5.4
Sequence Player Interface

The sequence player reads the file that is recorded from the joint generator. This file is structured in a way similar to G-Code used by computer numeric control system. For example, if one line reads "m 10 20 10", it is interpreted by the software as "move 10, 20,10 units x,y, z directions respectively". The software interpolates this movement into certain number of frames depending on the speed settings. It has the flexibility to generate additional frames for the more complex procedure in the sequence. The smooth interpolation of rotation can be achieved by the Quaternion SLERP function. As opposed to storing every frame in the file, the program only stores a small number of key moves. This approach greatly reduces the size of the sequence file, increasing its portability.

The user can control the frames in the sequence using the slider or the buttons below the slider. The left most button advance one frame per, while the second to left button advances five frames per click. (Fig.5.6) These buttons are useful for zooming in a particular time of interest, especially when significant amount of frames are loaded into the programs. (Fig.5.7)

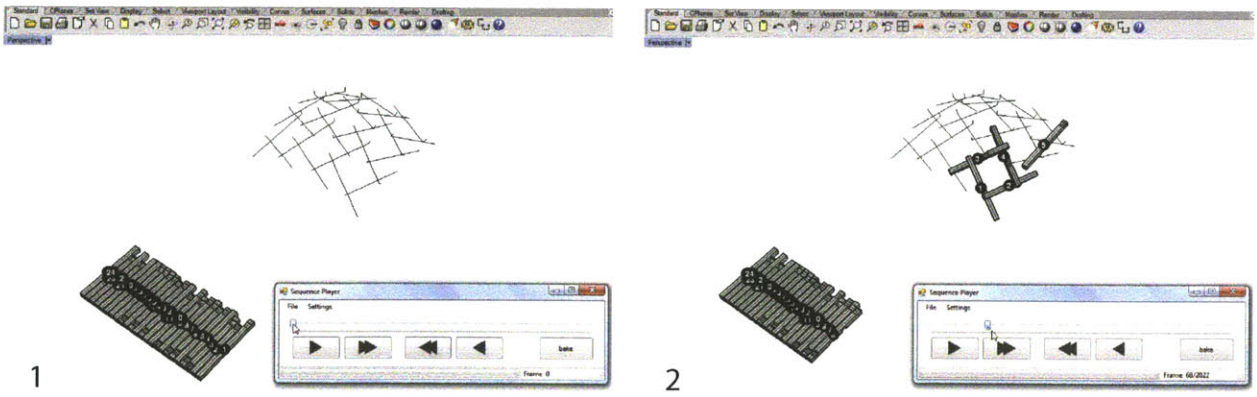


Fig. 5.5
Screen Shots of Sequence Player in Work

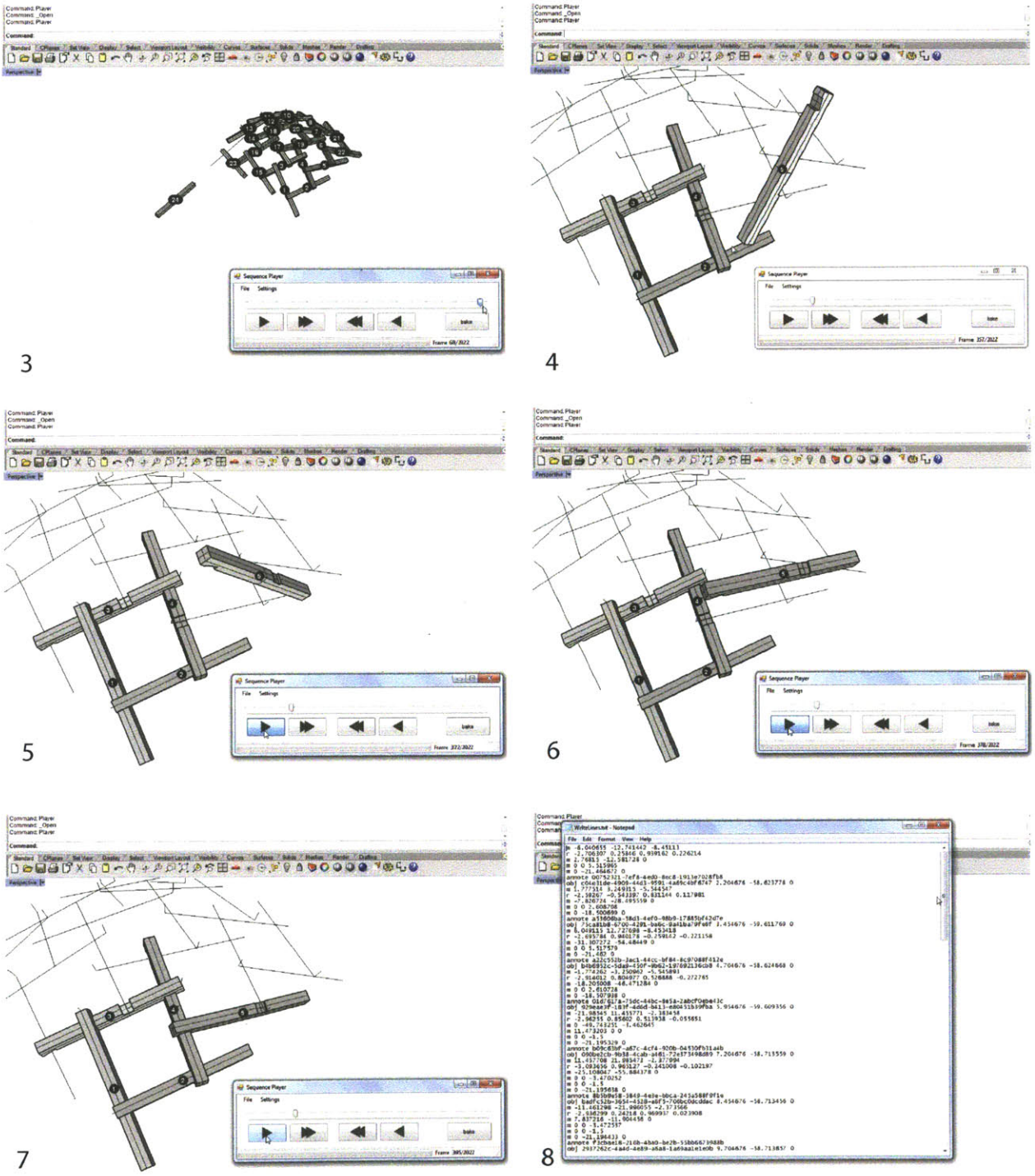


Fig. 5.5 (continued)
Screen Shots of Sequence Player at Work

5.4 Disassembler

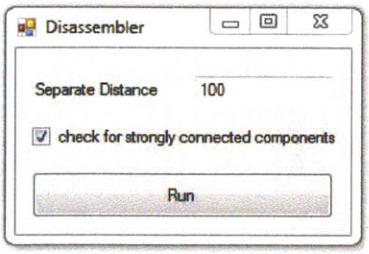


Fig. 5.6
Disassembler Interface

The disassembler not only generates disassemble sequence for interlocking frames, but also for generic assemblies defined by planar surface contact constraints. It is an extremely useful tool to examine the assembly sequence of systems with high geometric complexity, which is often found in the digital fabrication design process. This tool can be distributed as a standalone plugin for designers to evaluate assemblability of their designs at an early stage.

The only parameter input for the disassembler is the separation distance, which determines how far the disassembled member will travel before it stops. Fig.5.10 shows the process of disassembling a Diagonal Burr Puzzle. (Section 3.5

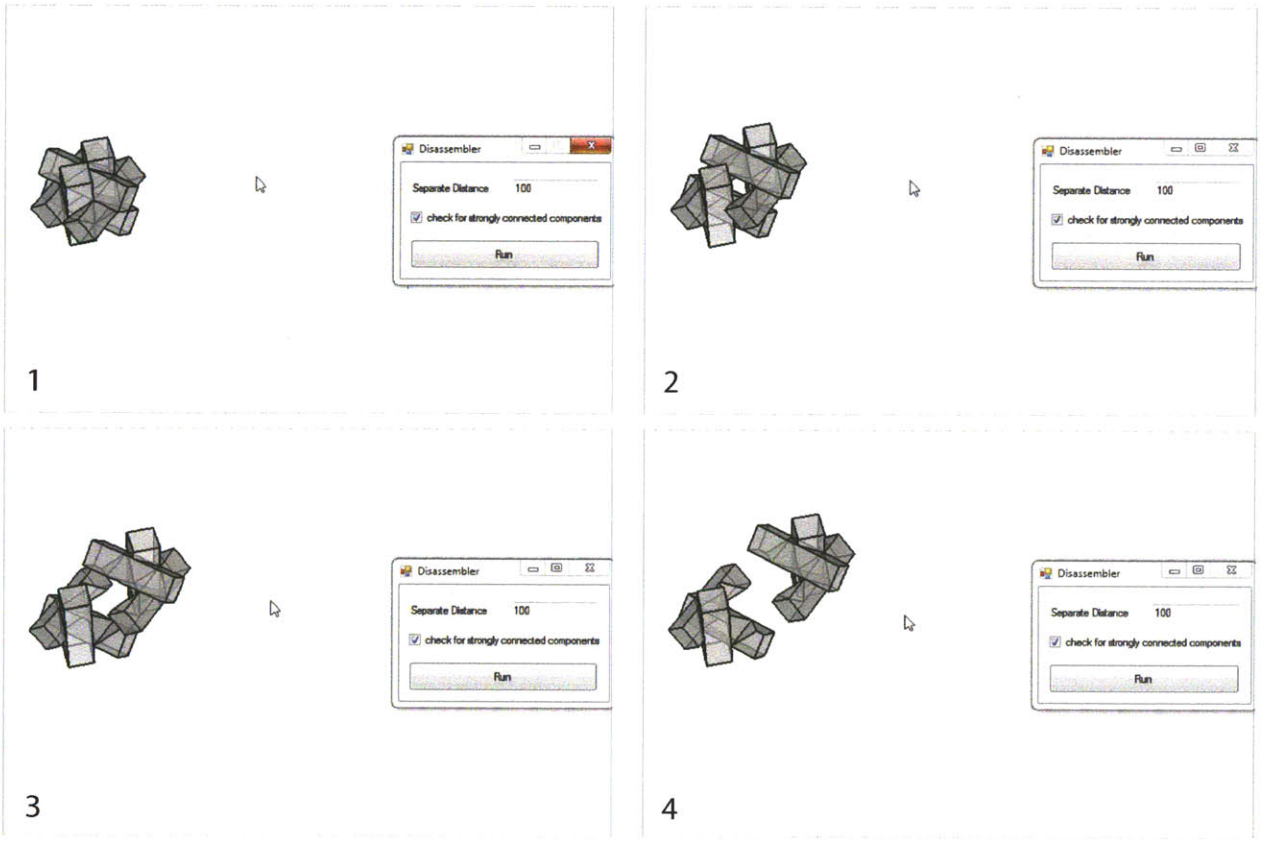


Fig. 5.7
Screen Shots of Disassembler at Work

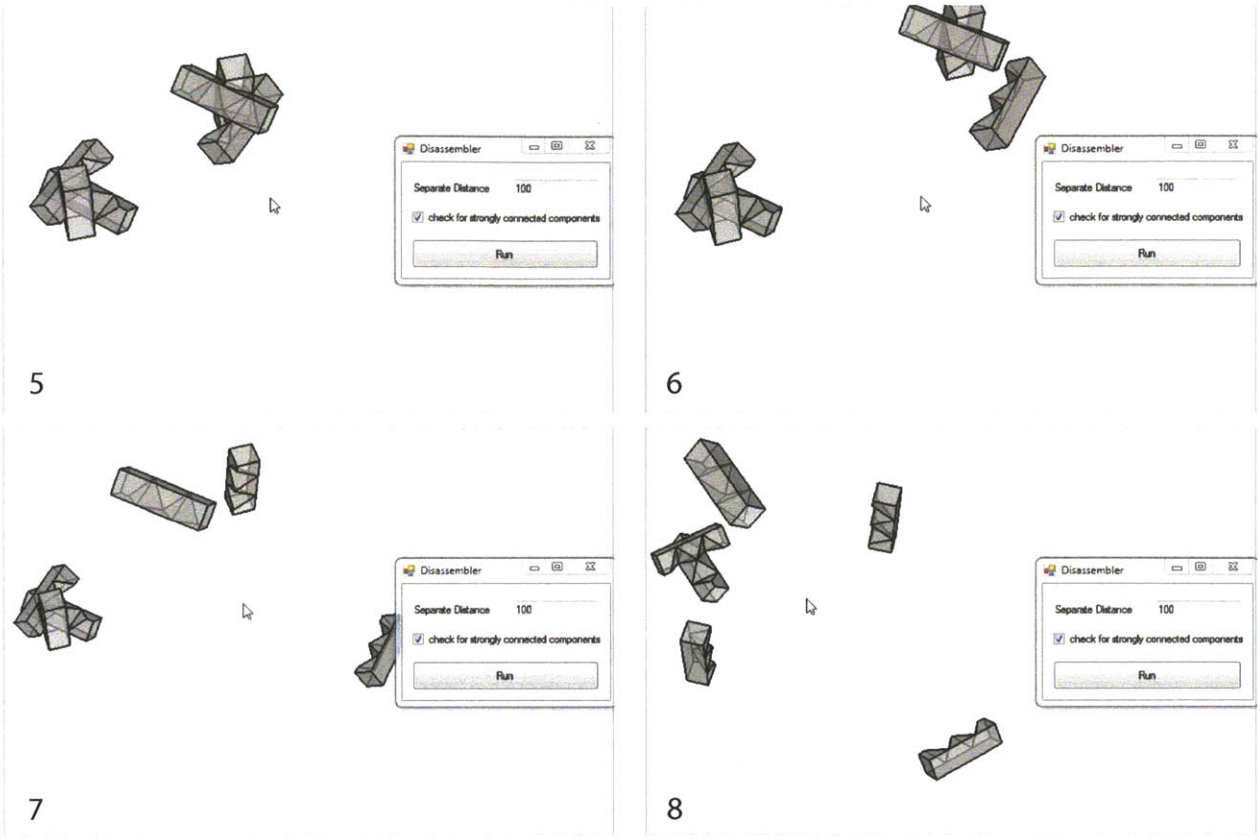


Fig. 5.7 (Continued)
Screen Shots of Disassembler at Work

6. Conclusion and Contribution

The contributions of this thesis are summarized as follow:

• Invents a new process to construct wooden frames without mechanical fasteners using global interlocking mechanism.

This process includes transforming an arbitrary two dimension grid into a three dimensional interlocking pattern, creating the joint details in members, and generating the assembly sequence. Because the interlocking wooden frame relies on geometric constraints rather than friction to interlock the members, it can be easily assembled and disassembled. All the members are reusable and possibly reconfigurable.

• Develops a tool that generates architectural details based on the assembly knowledge.

This thesis explores the question of how the assembly knowledge can help architects in the design process. The example of interlocking frames shows that the assembly knowledge can be used to inform the creation of joint details in an architectural assembly. Other possibilities are still waiting to be explored.

• Develops a tool that is integrated into the CAD software to provide 4D assembly instructions.

Instructions for assembly usually have two types of representation: illustration and animation. Neither of these types is sufficient to describe the complex space-time relationship of an assembly process. The sequence player offers users the flexibility to "zoom in" at a particular location or moment in both space and time, providing a clear instruction for assembly.

• Introduces the fundamental knowledge of geometric assembly planning to the field of architecture.

In the context of digital design and fabrication process, the designers often overlook the importance of post-fabrication process. This thesis focus on the generation of assembly sequence using the knowledge from the field of geometric assembly planning. This knowledge is fundamental and essential to the inquiry of automated assembly process in architecture, which includes robotic assembly and possibly self-assembly.

• Provides the basis for development of an intelligent and versatile 4D CAD simulation tool.

A quick survey of the current tools for 4D construction simulations, such as Navisworks[1] and Envoia[2], indicates that setting up 4D simulations in CAD environment is almost a manual process. These software programs provide the ability to check for collision detection on the manually scheduled path; however, it does not automatically generate path or provide possible options for path to the user. The deficiency of this manual approach toward construction simulation is pointed out by Tulkeand and Hanff.[3] It is possible to improve the efficiency of 4D simulation processes by integrating the geometric analysis methods presented in this thesis. The software developed in this thesis has the potential for development into an intelligent and versatile tool for 4D construction simulation.

---

1 Naviswroks - Project Review Software http://usa.autodesk.com/navisworks/

2 Enovia - Collaborative Innovation www.3ds.com/products/enovia

3 Tulke and Hanff, "4D Construction Sequence Planning – New Process And Data Model."

7. Bibliography

Coffin, Stewart. Geometric Puzzle Design. 2nd ed. AK Peters, 2007.

Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to Algorithms. third ed. The MIT Press, 2009.

Douthe, C., and O. Baverel. "Design of Nexorades or Reciprocal Frame Systems with the Dynamic Relaxation Method." Computers & Structures 87, no. 21 (2009): 1296–1307.

Goto, Kazuma, Kidokoro Ryota, and Mastuo Takeshi. "Rokko Mountain Observatory." The Arup Journal (February 2011).

Halperin, Dan, Jean-Claude Latombe, and Randall Wilson. "A Genral Framwork for Assembly Planning: The Motion Space Approach." 19–28, n.d.

Henrichsen, Christoph. Japan - Culture of Wood: Buildings, Objects, Techniques. 1st ed. Birkhäuser Basel, 2004.

Huynh, Johnny. "Separating Axis Theorem for Oriented Bounding Boxes" (2009). http://www.jkh.me/files/tutorials/Separating Axis Theorem for Oriented Bounding Boxes.pdf.

Jiménez, P. "Survey on Assembly Sequencing: a Combinatorial and Geometrical Perspective." Journal of Intelligent Manufacturing (August 11, 2011). http://www.springerlink.com/index/10.1007/s10845-011-0578-5.

Kardasis, Ari. "The Soft Grid". Massachusetts Institute of Technology, 2011.

Cardoso Llach, and others. "A Generative Grammar for 2D Manufacturing of 3D Objects". Massachusetts Institute of Technology, 2007.

Kohlhammer, T. "Discrete Analysis–A Method to Determine the Internal Forces of Lattices" 2011.

Kohlhammer, Thomas, and Toni Kotnik. "Systemic Behaviour of Plane Reciprocal Frame Structures." Structural Engineering International 21, no. 1 (February 1, 2011): 80–86.

Kolarevic, Branko, ed. Architecture in the Digital Age: Design and Manufacturing. New Ed. Taylor & Francis, 2005.

Larsen, Olga Popovic. Reciprocal Frame Architecture. 1st ed. Architectural Press, 2008.

Le, D. T, J. Cortés, and T. Siméon. "A Path Planning Approach to (dis) Assembly Sequencing." In Automation Science and Engineering, 2009. CASE 2009. IEEE International Conference On, 286–291, 2009.

Mitchell, William J. "The Logic of Architecture: Design, Computation, and Cognition" (April 26, 1990).

Nicolai, Carsten. Grid Index. Die Gestalten Verlag, 2009.

Papanikolaou, Dimitrios. "Attribute Process Methodology: Feasibility Assessment of Digital Fabrication Production Systems for Planar Part Assemblies Using Network Analysis and System Dynamics". Massachusetts Institute of Technology, 2008.

Roelofs, R. "Two-and Three-Dimensional Constructions Based on Leonardo Grids." Nexus Network Journal 10, no. 1 (2008): 17–26.

Sato, Hideo, and Yasua Nakahara. The Complete Japanese Joinery. Translated by Koichi Paul Nii. Hartley and Marks Publishers, 1995.

Simon, Herbert A. The Sciences of the Artificial - 3rd Edition. third ed. The MIT Press, 1996.

Tulke, Jan, and Jochen Hanff. "4D Construction Sequence Planning – New Process And Data Model", 2007.

Wilson, R. H, and J. C Latombe. "Geometric Reasoning About Mechanical Assembly." Artificial Intelligence 71, no. 2 (1994): 371–396.

Wilson, Randall. "On Geometric Assembly Planning". Stanford University, 1992.

8. Appendix

The source code developed in this thesis is attached in the appendix. The language for development is Microsoft C#. The plugin program runs in Rhinoceros 3D version 5, referencing the Rhino Common library.

The Assembly, Joint, Member section defines the geometry of members and joints. The assembly is a collection of both joints and members. It includes code that defines the intersection conditions, joint creation, quick intersection checking using Separating Axis Theorem and various utility methods.

The Graph for DBG section defines the graph in a Directional Blocking Graph. It contains depth first search and Kasaraju's Theorem for finding strongly connected components.

The Spherical Geometry for NDBG section is defines the spherical geometry operations in non-directional blocking graph. It includes the definition of great circles, segments in a great circle, regions on sphere, and Boolean operations between regions. This is the core library that determines the free movement in an interlocking condition.

The last section is the sequence player. It includes methods that parse the sequence file, interpolate movement, and display the sequence in the Rhinoceros display viewport.

8. Appendix

Assembly, Member, and Joint

```
using System;
using System.Collections.Generic;
using System.Text;
using Rhino.Geometry;
using Rhino;

namespace Disassembler
{
  public enum intersect_type { Corner_TL = 0, Corner_TR, Corner_BR, Corner_BL, Edge_L, Edge_T, Edge_R, Edge_B,
Opposite, Three, None, Inconsistent };
  public enum joint_type { None=0,Full,Half};

  class Assembly
  {
    private List<Member> initial_member;
    private List<Joint> initial_joints;
    public List<Member> members { get; set; }
    public List<Joint> joints { get; set; }
    private BoundingBox bounding_box;

    public Assembly(List<Member> members, List<Joint> joints)
    {
      initial_member = members;
      initial_joints = joints;
      this.members = new List < Member > (members);
      this.joints = new List<Joint> (joints);
      relink_member_from_joints();
      bounding_box = calculate_bounding_box(); //anchor for first
    }

    public Assembly(List<Member> members, List<Joint> joints, bool proxy) //creates a light weight assembly
instance that only includes initial members and joints
    {
      initial_member = members;
      initial_joints = joints;
    }

    public int[] modify_code(int[] final_code,int[] code)
    {
      for (int i = 0; i < initial_joints.Count; i++)
        final_code[initial_joints[i].index]=code[i];
      return final_code;
    }

    public BoundingBox calculate_bounding_box()
    {
      BoundingBox box = new BoundingBox();
      if (members.Count > 0) box = members[0].brep.GetBoundingBox(true);
      if (members.Count>1)
      {
```

```
    for (int i = 1; i < members.Count;i++)
      box.Union(members[i].brep.GetBoundingBox(true));
    }
    return box;
}

public void reset(int[] joint_types)
{
    this.members = new List<Member>(initial_member);
    this.joints = new List<Joint>(initial_joints);
    for (int i = 0; i < joint_types.Length; i++)
      joints[i].assign_type((joint_type)joint_types[i]);
    relink_member_from_joints();
}

public int movable_parts(int[] joint_types) //this is fitness function for GA
{
    reset(joint_types);
    if (joint_types.Length != joints.Count) throw new Exception("code and joint count not matching");
    List<int> score;
    disassemble_single(false,out score); //(loose,up)
    //if (score[2] == initial_member.Count)
    //    return score[2] + score[0] + score[1] - score[3] / 3;
    //else
    return score[2];// + score[0];
}

public Assembly create_assembly_from_lock_state(int[] joint_types,RhinoDoc doc)
{
    reset(joint_types);
    List<int> score;
    List<int> sequence=disassemble_single(false, out score);
    sequence.Sort();
    int count=0;
    for (int i = 0; i < initial_member.Count && count < sequence.Count; i++)
    {
      if (initial_member[i].index == sequence[count])
      {
        Util.object_color(initial_member[i].guid, System.Drawing.Color.Magenta, doc);
        count++;
      }
    }
    doc.Views.Redraw();
    return new Assembly(members, joints);
}


public void restore_and_redraw(int[] joint_types,RhinoDoc doc)
{
    reset(joint_types);
    rearrange_member_by_order(joint_types);
    annotate(doc);
    draw_volume(doc);
    List<int> score;
    disassemble_single(true,out score);
```

```
    //List<int> sequence = disassemble_single(true);
    //return sequence;
}

public void rearrange_member_by_order(int[] joint_types)
{
    List<int> score;
    List<int> seq = disassemble_single(false, out score); //(loose,up)
    reset(joint_types);
    for (int i = 0; i < seq.Count; i++)
        members[seq[i]].index = seq.Count - i;
}

public Vector3d[] extract_directions_from_joints()
{
    List<Vector3d> directions=new List<Vector3d>();
    List<Edge> edges=new List<Edge>();
    foreach(Joint j in joints)
    {
        if (j.NDBG_m1.is_point())
        {
            foreach(Vector3d pt in j.NDBG_m1.get_points())
                if (!VecUtil.contain(directions,pt)) directions.Add(pt);
        }
        else
        {
            foreach(Edge e1 in j.NDBG_m1.get_edges())
            {
                bool contain=false;
                if (!e1.geodesic.is_greatcircle())
                {
                    if (!VecUtil.contain(directions,e1.geodesic.start_pt()))
                        directions.Add(e1.geodesic.start_pt());
                    if (!VecUtil.contain(directions,e1.geodesic.end_pt()))
                        directions.Add(e1.geodesic.end_pt());
                }
                foreach(Edge e2 in edges)
                {
                    Vector3d[] pt=Edge.intersect(e1,e2);
                    foreach(Vector3d p in pt)
                        if (!VecUtil.contain(directions,p)) directions.Add(p);
                    if (e1 == e2) contain = true;
                }
                if (!contain) edges.Add(e1);
            }
        }
    }
    int count = directions.Count;
    for (int i = 0; i < count; i++)
    {
        if (!VecUtil.contain(directions, -directions[i])) directions.Add(-directions[i]);
    }
    foreach (Edge e in edges) e.draw(RhinoDoc.ActiveDoc);
    foreach (Vector3d p in directions) RhinoDoc.ActiveDoc.Objects.AddPoint(new Point3d(p));
    return directions.ToArray();
```

```
}

public Graph<Member> consturct_DBG(Vector3d dir)
{
  //member.index must equal its index in members
  Graph<Member> graph=new Graph<Member>();
  for (int i = 0; i < members.Count; i++)
    members[i].index = i;
  for (int i = 0; i < members.Count; i++)
    graph.nodes.Add(new Node<Member>(members[i],i));

  for(int i=0;i<joints.Count;i++)
  {
    int n1 = joints[i].member1.index;
    int n2 = joints[i].member2.index;
    if (!joints[i].NDBG_m1.within(dir)) graph.nodes[n1].links.Add(graph.nodes[n2]);
    if (!joints[i].NDBG_m2.within(dir)) graph.nodes[n2].links.Add(graph.nodes[n1]);
  }
  return graph;
}

void relink_member_from_joints()
{
  for (int i = 0; i < members.Count; i++)
    members[i].joints = new List<Joint>();
  for (int i = 0; i < joints.Count; i++)
  {
    joints[i].member1.joints.Add(joints[i]);
    joints[i].member2.joints.Add(joints[i]);
  }
  foreach (Member m in members) m.NDBG_from_joint();
}

public void annotate(RhinoDoc doc)
{
  for (int i = 0; i < members.Count; i++)
    members[i].draw_annotation(doc);
}

public void draw_intact_volume(RhinoDoc doc) //delete current, set notches, draw
{
  for (int i = 0; i < members.Count; i++)
  {
    members[i].delete_volume(doc);
    members[i].initialize_brep();
  }
  foreach (Member m in members)
    m.draw_volume(doc);
  RhinoDoc.ActiveDoc.Views.Redraw();
}

public void draw_volume(RhinoDoc doc,int[] types) //delete current, set notches, draw
{
  reset(types);
```

```
        draw_volume(doc);
    }

    public void draw_volume(RhinoDoc doc) //delete current, set notches, draw
    {
        for (int i = 0; i < members.Count; i++)
        {
            members[i].delete_volume(doc);
            members[i].initialize_brep();
        }
        for (int i = 0; i < joints.Count; i++)
            joints[i].set_notch_brep();
        foreach (Member m in members)
            m.draw_volume(doc);
        RhinoDoc.ActiveDoc.Views.Redraw();
    }

    public List<int> disassemble_single(bool draw,out List<int> score) //predefined anchor point
    {
        Vector3d anchor = new Vector3d(bounding_box.Min) - new Vector3d(0, (members[0].depth * 3), 0);
        anchor.Z = 0;
        double offset = members[0].width * 1.25;
        return disassemble_single(draw, anchor, offset,out score);
    }

    public List<int> disassemble_single(bool draw, Vector3d anchor, double offset,out List<int> score) //custom
anchor point
    {
        List<int> sequence = new List<int>();
        List<string> record = new List<string>();
        int good, up, count, loose;
        loose = good = up = count = 0;
        while (count < initial_member.Count)
        {
            if (members.Count == 0) break;
            List<int> remove = new List<int>();
            for (int i = 0; i < members.Count; i++)
            {
                if (members[i].NDBG != null)
                    remove.Add(i);
            }
            if (remove.Count == 0) break;
            int min = int.MaxValue;
            int min_index = 0;

            //select the one with least connections, selection of branch is happening here
            for (int i = 0; i < remove.Count; i++)
            {
                if (members[remove[i]].joints.Count < min)
                {
                    min = members[remove[i]].joints.Count;
                    min_index = remove[i];
                }
            }
```

```
            if (count == 0) good -= 2 * remove.Count;
            loose += remove.Count;
            //if (members[min_index].NDBG != null && members[min_index].NDBG.within(new Vector3d(0, 0, 1))) up
+= 1;
            //if (members[min_index].joints.Count == 1) good+=1;
            //else if (members[min_index].joints.Count == 2) good++;
            //else if (members[min_index].joints.Count == 4) good -= 8;
            //else if (members[min_index].joints.Count == 3) good -= 4;
            if (draw) //record movement
               record_movement(members[min_index], record, anchor + new Vector3d(count * offset, -5, 0), 5);
            sequence.Add(members[min_index].index);
            remove_member(members, min_index);
            count ++;
         }
         if (draw)
            System.IO.File.WriteAllLines(@"C:\Users\sctai\Desktop\WriteLines.txt", record);

         score = new List<int>(new int[] { good, up, count,loose});
         return sequence; //for reordering the members
      }

      public void record_movement(Member m,List<string> record,Vector3d position,int speed)
      {
         Vector3d dir = new Vector3d();
         if (m.NDBG.is_ball())
            dir = new Vector3d(0, 0, 1);
         else if (m.NDBG.is_point())
            dir = m.NDBG.get_points()[0];
         else
            dir = m.NDBG.get_edges()[0].geodesic.mid_pt();
         record.AddRange(m.move_avoid_bounding_box(dir, position, bounding_box,speed));
         record.Add("annote " + m.guid_annote.ToString());
         record.Add("obj " + m.guid.ToString() + string.Format(" {0:0.######} {1:0.######} {2:0.######}",
m.center.X, m.center.Y, m.center.Z));
      }

      public void remove_member(List<Member> members,int index)
      {
         Member m=members[index];
         for (int i = 0; i < m.joints.Count; i++)
         {
            //delete connected member joint storage ,delete all joints //modify NDBG
            Member[] links=m.links();//members linking to
            for(int j=0;j<links.Length;j++)
            {
               links[j].joints.Remove(m.joints[i]);
               joints.Remove(m.joints[i]);
               links[j].NDBG_from_joint();
            }
         }
         members.Remove(m);
      }

      public int[] replace_half_joint(int[] joint_types)
      {
```

```
        List<int> joints=new List<int>();
        int temp;
        int score=movable_parts(joint_types);
        for (int i = 0; i < joint_types.Length; i++)
        {
            if (joint_types[i] != (int)joint_type.Half)
            {
                temp = joint_types[i];
                joint_types[i] = (int)joint_type.Half;
                int new_score = movable_parts(joint_types);
                if (new_score == score) joints.Add(i);
                else joint_types[i] = temp;
            }
        }
        if (movable_parts(joint_types) == score)
            return joint_types;
        else
            throw new Exception("switch to half joints failed");
    }
}



class Joint
{
    public Member member1 { get; set; }
    public Member member2 { get; set; }
    public Region NDBG_m1 { get; set; }
    public Region NDBG_m2 { get { return NDBG_m1.reverse(); } }
    //private joint_type m_type;
    public joint_type type { get; set; }
    public int index { get; set; }
    private intersect_type[] intersect_condition;

    public Joint(Member m1,Member m2)
    {
        this.member1 = m1;
        this.member2 = m2;
        m1.joints.Add(this);
        m2.joints.Add(this);
        intersect_condition = Member.intersect(m1, m2);
    }
    public Joint(Member m1, Member m2, joint_type joint)
        : this(m1, m2)
    {
        assign_type(joint);
    }

    public Joint(Member m1, Member m2, joint_type joint, int index)
        : this(m1, m2, joint)
    {
        this.index = index;
    }

    public void assign_type(joint_type type)
```

```
    {
        this.type = type;
        NDBG_m1 = NDBG_from_intersection(type, member1, member2);
        if (NDBG_m1 == null) throw new Exception("null NDBG in joint");
    }

    public void set_notch_brep()
    {
        if (type == joint_type.Full)
            member1.create_full_notch( member2);
        else if (type == joint_type.None)
            member2.create_full_notch(member1);
        else if (type == joint_type.Half)
        {
            Brep[] breps=Member.create_intersection_half_volume(member1,member2);
            member1.create_full_notch(new Member(breps[0]));
            member2.create_full_notch(new Member(breps[1]));
        }
        else
            throw new Exception("joint type error");
    }

    public Region NDBG_from_intersection(joint_type joint,Member m1, Member m2) //passing in m1,m2 is
necessary for code reuse
    {
        intersect_type[] inter=Member.intersect(m1, m2);
        if (joint == joint_type.None) //depend on self
            return free_direction(inter[0], m1);
        else if (joint == joint_type.Full) //depend on other
            return NDBG_from_intersection(joint_type.None, m2, m1).reverse();
        else //Half*************************
        {
            List<Vector3d> block_all = new List<Vector3d>();
            //block.AddRange(block_direction(inter[0], m1));
            List<Vector3d> block = new List<Vector3d>(block_direction(inter[0], m1));
            //int i=(block.Count==3)? 1:0;
            int i=0; //need to modify this too*********************************************if its half joint in
corner edge intersection
            for (int j=i; j < block.Count; j++)
                block_all.Add(block[j]);
            block = new List<Vector3d>(block_direction(inter[1], m2));
            i=(block.Count==3)? 1:0;
            for (int j = i; j < block.Count; j++)
                block_all.Add(block[j]);
            Region free = Region.get_free_direction(block_all.ToArray());
            //if (free.is_point() && free.get_points().Count==2)
            //else
            //   throw new Exception("cannot find two free directions");
            int test = 0;
            if (Region.get_free_direction(block_all.ToArray()) == null)
                test = 1;
            return Region.get_free_direction(block_all.ToArray());
        }
    }
```

```
    public Region free_direction(intersect_type inter, Member m1) //can be moved to member
    {
      if ((int)inter >= 0 && (int)inter < 4)
      {
        if (inter == intersect_type.Corner_TL) return Region.get_free_direction(new Vector3d[] { -m1.xDir,
m1.yDir });
        else if (inter == intersect_type.Corner_TR) return Region.get_free_direction(new Vector3d[] { m1.xDir,
m1.yDir });
        else if (inter == intersect_type.Corner_BR) return Region.get_free_direction(new Vector3d[] { m1.xDir, -
m1.yDir });
        else if (inter == intersect_type.Corner_BL) return Region.get_free_direction(new Vector3d[] { -m1.xDir, -
m1.yDir });
      }
      if ((int)inter >= 4 && (int)inter < 8)
      {
        Segment s=new Segment(new Vector3d(),new Vector3d());
        if (inter == intersect_type.Edge_L) s = new Segment(-m1.zDir, m1.zDir, m1.xDir);
        else if (inter == intersect_type.Edge_T) s = new Segment(-m1.zDir, m1.zDir, -m1.yDir);
        else if (inter == intersect_type.Edge_R) s = new Segment(-m1.zDir, m1.zDir, -m1.xDir);
        else if (inter == intersect_type.Edge_B) s = new Segment(-m1.zDir, m1.zDir, m1.yDir);
        return new Region(new Edge[] { new Edge(s, false, new Vector3d()) }, true);
      }
      else throw new Exception("intersection condition not defined");
    }

    public Vector3d[] block_direction(intersect_type inter,Member m1) //*************************the sign
might be wrong?
    {
      if (inter == intersect_type.Corner_TL) return new Vector3d[] { -m1.xDir, m1.yDir };
      else if (inter == intersect_type.Corner_TR) return new Vector3d[] { m1.xDir, m1.yDir };
      else if (inter == intersect_type.Corner_BR) return new Vector3d[] { m1.xDir, -m1.yDir };
      else if (inter == intersect_type.Corner_BL) return new Vector3d[] { -m1.xDir, -m1.yDir };
      else if (inter == intersect_type.Edge_L) return new Vector3d[] { -m1.xDir, m1.yDir, -m1.yDir };
      else if (inter == intersect_type.Edge_T) return new Vector3d[] { m1.yDir, m1.xDir, -m1.xDir };
      else if (inter == intersect_type.Edge_R) return new Vector3d[] { m1.xDir, m1.yDir, -m1.yDir };
      else if (inter == intersect_type.Edge_B) return new Vector3d[] { -m1.yDir, m1.xDir, -m1.xDir };
      else throw new Exception("intersection condition not defined");
    }

  }

  class Member
  {
    public Vector3d start;
    private Vector3d end;
    private double height;
    public double width { get; set; }
    public Vector3d zDir { get; set; }
    public Vector3d xDir{ get; set; }
    public Vector3d yDir{ get; set; }
    public List<Joint> joints { get; set; }
    public Region NDBG { get; set; }
    public Vector3d center { get; set; } //for player ************************

    public Vector3d mid { get { return (start + end) / 2; } }
```

```
public double depth { get { return (start - end).Length; } }
public int index { get; set; }
public Brep brep { get; set; }
public Guid guid { get; set; }
public Guid guid_annote { get; set; }
//private double rotation = 0;

public Member(Vector3d start,Vector3d end)
{
  initialize(start, end);
  set_frame();
  initialize_brep();
  center = mid;
}

public Member(Vector3d start, Vector3d end,Vector3d up)
{
  initialize(start, end);
  set_frame(up);
  initialize_brep();
  center = mid;
}

public void initialize(Vector3d start, Vector3d end)
{
  this.start = start;
  this.end = end;
  this.height = 1;
  this.width = 1;
  zDir = VecUtil.unitize(start - end);
  joints = new List<Joint>();

}

public void set_frame(Vector3d up)
{
  xDir = VecUtil.unitize(Vector3d.CrossProduct(zDir, up));
  yDir = VecUtil.unitize(Vector3d.CrossProduct(xDir, zDir));
}

public void set_frame()
{
  if (!(zDir.X == 0 && zDir.Y == 0))
    xDir = VecUtil.unitize(Vector3d.CrossProduct(zDir, new Vector3d(0, 0, 1)));
  else
    xDir = new Vector3d(1, 0, 0);//-1,0,0?
  yDir = VecUtil.unitize(Vector3d.CrossProduct(xDir, zDir));
}

public Member(Vector3d start, Vector3d end, int index, Vector3d up)
  : this(start, end, up)
{
  this.index = index;
}
```

```
    public Member(Vector3d start, Vector3d end, int index)
      : this(start, end)
    {
      this.index = index;
    }

    public Member(Vector3d start, Vector3d end, int index, Vector3d up, double height, double width)
      : this(start, end,index,up)
    {
      this.height = height;
      this.width = width;
    }
    public Member(Brep brep) { this.brep = brep; } //only for boolean operation
    public Member(Guid guid, Vector3d center, int index)
    {
      this.guid = guid;
      this.center = center;
      this.index = index;
    } //only for animation player

    public Member[] links()
    {
      Member[] links=new Member[joints.Count];
      for (int i = 0; i < links.Length;i++)
        links[i] = (this == joints[i].member1) ? joints[i].member2 : joints[i].member1;
      return links;
    }

    public bool is_unnotched() { return (brep.Faces.Count == 6) ? true : false; }


    public void create_full_notch(Member member)
    {
      double tolerance = RhinoDoc.ActiveDoc.ModelAbsoluteTolerance;
      Brep[] breps = Brep.CreateBooleanDifference(this.brep, member.brep, tolerance);
      if (breps==null || breps.Length > 1) throw new Exception("boolean failure");
      else
      {
        brep = breps[0];
        if (brep.SolidOrientation == BrepSolidOrientation.Inward) brep.Flip();
      }
    }

    public List<string> move_avoid_bounding_box(Vector3d initial_direction, Vector3d final_position, BoundingBox
bb_all,int speed)
    {
      List<string> record = new List<string>();
      final_position.Y = final_position.Y + (depth / 2);
      Vector3d dir = initial_direction;
      if (dir.IsZero) throw new Exception("zero vector error");
      BoundingBox bb_self = brep.GetBoundingBox(true);
      List<double> distance = new List<double>();
      double x = bb_self.Max.X - bb_self.Min.X;
      double y = bb_self.Max.Y - bb_self.Min.Y;
      double z = bb_self.Max.Z - bb_self.Min.Z;
```

```
        if (dir.X > 0) distance.Add((bb_all.Max.X - bb_self.Max.X + x) / dir.X);
        else if (dir.X < 0) distance.Add((bb_all.Min.X - bb_self.Min.X - x) / dir.X);
        else distance.Add(10000);

        if (dir.Y > 0) distance.Add((bb_all.Max.Y - bb_self.Max.Y + y) / dir.Y);
        else if (dir.Y < 0) distance.Add((bb_all.Min.Y - bb_self.Min.Y - y) / dir.Y);
        else distance.Add(10000);

        if (dir.Z > 0) distance.Add((bb_all.Max.Z - bb_self.Max.Z + z) / dir.Z);
        else if (dir.Z < 0) distance.Add((bb_all.Min.Z - bb_self.Min.Z - z) / dir.Z);
        else distance.Add(10000);

        double dis=0;
        int axis=0;
        double move_speed = 2 * speed;
        double rotation_speed = 0.2 * speed;

        for (int i = 0; i < 3; i++) //distance.count can be 1 or 2 or 3
        {
          if (distance[i] <= distance[(i + 1) % 3] && distance[i] <= distance[(i + 2) % 3])
          {
            dis = distance[i];
            axis = i;
          }
        }

        Vector3d move = dir * dis;
        record.Add(animate_move(move, move_speed));
        record.Add(animate_rotate_to_WCS(rotation_speed));

        Vector3d inter_pos = final_position + new Vector3d(0, depth * 1.5, height * 1.5);
        if (axis == 0)
        {
          record.Add(animate_move(new Vector3d(center.X, inter_pos.Y, inter_pos.Z) - center, move_speed));
          record.Add(animate_move(new Vector3d(inter_pos.X - center.X, 0, 0), move_speed));
          record.Add(animate_move(new Vector3d(final_position.X - center.X, 0, final_position.Z - center.Z),
move_speed));
        }
        else if (axis == 1)
        {
          if (mid.Y > bb_all.Max.Y)
          {
            double clear_height = bb_all.Max.Z + z;
            record.Add(animate_move(new Vector3d(0, 0, clear_height - center.Z), move_speed));
            record.Add(animate_move(new Vector3d(inter_pos.X, inter_pos.Y, center.Z) - center, move_speed));
            record.Add(animate_move(new Vector3d(0, 0, inter_pos.Z - center.Z), move_speed));
            record.Add(animate_move(new Vector3d(final_position.X - center.X, 0, final_position.Z - center.Z),
move_speed));
          }
          else
          {
            record.Add(animate_move(new Vector3d(inter_pos.X, inter_pos.Y, center.Z) - center, move_speed));
            record.Add(animate_move(new Vector3d(0, 0, inter_pos.Z - center.Z), move_speed));
            record.Add(animate_move(new Vector3d(final_position.X - center.X, 0, final_position.Z - center.Z),
move_speed));
```

```
            }
        }
        else //axis==2
        {
            record.Add(animate_move(new Vector3d(inter_pos.X, inter_pos.Y, center.Z) - center, move_speed));
            record.Add(animate_move(new Vector3d(0, 0, final_position.Z - center.Z), move_speed));
        }
        //record.Add(animate_rotate_to_WCS(rotation_speed));
        record.Add(animate_move(new Vector3d(0, final_position.Y - center.Y, 0), move_speed));
        return record;
    }

    public string animate_rotate_to_WCS(double speed)
    {
        Plane p1 = new Plane(new Point3d(center), xDir, yDir);
        Plane p2 = new Plane(new Point3d(0, 0, 0), new Vector3d(1, 0, 0), new Vector3d(0, 0, 1));
        Quaternion q = Quaternion.Rotation(p1, p2);
        double angle;
        Vector3d axis;
        q.GetRotation(out angle, out axis);
        if (angle > Math.PI) angle -= 2 * Math.PI;
        animate_rotate(angle, axis, speed);
        return string.Format("r {0:0.######} {1:0.######} {2:0.######} {3:0.######}", angle, axis.X, axis.Y, axis.Z);
//set precision for file
    }

    public void animate_rotate(double angle, Vector3d axis, double speed)
    {
        int frame = (int)Math.Abs(angle / speed);
        for (int i = 0; i <= frame; i++)
        {
            RhinoApp.Wait();
            rotate(angle / (frame + 1), axis);
        }
    }

    public void rotate(double angle, Vector3d axis)
    {
        Transform t = Transform.Rotation(angle, axis, new Point3d(center));
        if (guid != null) RhinoDoc.ActiveDoc.Objects.Transform(guid, t, true);
        if (brep != null) brep.Transform(t);
        draw_annotation(RhinoDoc.ActiveDoc);

        //start and end rotation will cause error...in order to keep mid at the same point, don't transform

        Rhino.RhinoDoc.ActiveDoc.Views.Redraw();
    }

    public void move(Vector3d vec)
    {
        Transform t = Transform.Translation(vec);
        if (guid != null) RhinoDoc.ActiveDoc.Objects.Transform(guid, t, true);
        if (brep != null) brep.Translate(vec);
        center += vec;
        draw_annotation(RhinoDoc.ActiveDoc);
```

```
        //start += vec;
        //end += vec;
        Rhino.RhinoDoc.ActiveDoc.Views.Redraw();
    }

    public string animate_move(Vector3d vec, double speed)
    {
        int frame = (int)(vec.Length / speed);
        for (int i = 0; i <= frame; i++)
        {
            move(vec / (frame + 1));
            RhinoApp.Wait();
        }
        return string.Format("m {0:0.######} {1:0.######} {2:0.######}", vec.X, vec.Y, vec.Z); //set precision for file
    }

    public void initialize_brep()
    {
        Point3d p1 = new Point3d(start - (width / 2) * xDir - (height / 2) * yDir);
        Point3d p2 = new Point3d(start - (width / 2) * xDir + (height / 2) * yDir);
        Point3d p3 = new Point3d(start + (width / 2) * xDir + (height / 2) * yDir);
        Point3d p4 = new Point3d(start + (width / 2) * xDir - (height / 2) * yDir);
        Point3d p5 = new Point3d(end - (width / 2) * xDir - (height / 2) * yDir);
        Point3d p6 = new Point3d(end - (width / 2) * xDir + (height / 2) * yDir);
        Point3d p7 = new Point3d(end + (width / 2) * xDir + (height / 2) * yDir);
        Point3d p8 = new Point3d(end + (width / 2) * xDir - (height / 2) * yDir);
        brep = Brep.CreateFromBox(new Point3d[] { p1, p2, p3, p4, p5, p6, p7, p8 });
        if (brep.SolidOrientation == BrepSolidOrientation.Inward) brep.Flip();  //oreintation flipped will cause boolean
error
    }

    public void draw_annotation(RhinoDoc doc)
    {
        if (guid_annote != new Guid()) doc.Objects.Delete(guid_annote, true);
        TextDot dot = new TextDot(index.ToString(),new Point3d(center));
        guid_annote = doc.Objects.AddTextDot(dot);
    }

    public void draw_volume(RhinoDoc doc)
    {
        guid = doc.Objects.AddBrep(brep);
    }

    public void delete_volume(RhinoDoc doc)
    {
        if (guid!=new Guid()) doc.Objects.Delete(guid, true);
        guid = new Guid();
    }

    public Vector3d side_normal_direction(intersect_type inter) //can be moved to member
    {
        //if (inter == intersect_type.Corner_TL) return new Vector3d[] { -m1.xDir, m1.yDir };
        //else if (inter == intersect_type.Corner_TR) return new Vector3d[] { m1.xDir, m1.yDir };
        //else if (inter == intersect_type.Corner_BR) return new Vector3d[] { m1.xDir, -m1.yDir };
        //else if (inter == intersect_type.Corner_BL) return new Vector3d[] { -m1.xDir, -m1.yDir };
```

```
        if (inter == intersect_type.Edge_L) return yDir;
        else if (inter == intersect_type.Edge_T) return xDir;
        else if (inter == intersect_type.Edge_R) return yDir;
        else if (inter == intersect_type.Edge_B) return xDir;
        else throw new Exception("side_normal_direction not defined");
    }

    public static Brep create_surface(Vector3d normal, Vector3d centroid,double length)
    {
        Vector3d vec = new Vector3d(0, 0, 1);
        if (VecUtil.vector_equal(normal, new Vector3d(0, 0, 1)) || VecUtil.vector_equal(normal, new Vector3d(0, 0, -
1)))
            vec = new Vector3d(1, 0, 0);
        Vector3d vecX = VecUtil.unitize(Vector3d.CrossProduct(vec, normal)) * length;
        Vector3d vecY = VecUtil.unitize(Vector3d.CrossProduct(normal, vecX)) * length;
        Point3d p1 = new Point3d(centroid + vecX / 2 + vecY / 2);
        Point3d p2 = new Point3d(centroid + vecX / 2 - vecY / 2);
        Point3d p3 = new Point3d(centroid - vecX / 2 - vecY / 2);
        Point3d p4 = new Point3d(centroid - vecX / 2 + vecY / 2);
        return Brep.CreateFromCornerPoints(p1, p2, p3, p4, 0.01);
    }

    public static Vector3d brep_centroid(Brep brep)
    {
        Point3d[] pt = brep.DuplicateVertices();
        Point3d sum = new Point3d();
        foreach (Point3d p in pt) sum += p;
        Vector3d centroid = new Vector3d(sum / pt.Length);
        return centroid;
    }

    public static Brep[] create_intersection_half_volume(Member member1,Member member2) //only work for
edge edge intersection, corner needs implementataion
    {
        double tolerance = RhinoDoc.ActiveDoc.ModelAbsoluteTolerance;
        Brep[] breps = Brep.CreateBooleanIntersection(member1.brep, member2.brep, tolerance);
        if (breps.Length > 1) throw new Exception("boolean intersection to multiple parts");
        else
        {
            Brep brep = breps[0];
            Vector3d centroid = brep_centroid(brep);
            intersect_type[] inter = Member.intersect(member1, member2);
            Vector3d n1 = member1.side_normal_direction(inter[0]);
            Vector3d n2 = member2.side_normal_direction(inter[1]);
            Vector3d normal=VecUtil.unitize(Vector3d.CrossProduct(n1, n2));
            Brep cut = create_surface(normal, centroid, member1.height * 5);
            breps = brep.Split(cut, tolerance);
            //foreach (Brep b in breps) RhinoDoc.ActiveDoc.Objects.AddBrep(b);

            for (int i = 0; i < breps.Length; i++)
            {
                breps[i]=breps[i].CapPlanarHoles(tolerance);
                if (breps[i].SolidOrientation == BrepSolidOrientation.Inward) breps[i].Flip();
            }
            if (breps.Length == 2 && breps[0].IsSolid && breps[1].IsSolid)
```

```
      {
        Vector3d vec = brep_centroid(breps[1])-brep_centroid(breps[0]);
        double dotX=vec* member1.xDir;
        double dotY=vec* member1.yDir;
        if (inter[0] == intersect_type.Edge_L)
          return (dotX > 0) ? breps : new Brep[] { breps[1], breps[0] };
        else if (inter[0] == intersect_type.Edge_T)
          return (dotY > 0) ? new Brep[] { breps[1], breps[0] } : breps;
        else if (inter[0] == intersect_type.Edge_R)
          return (dotX > 0) ? new Brep[] { breps[1], breps[0] } : breps;
        else if (inter[0] == intersect_type.Edge_B)
          return (dotY > 0) ? breps : new Brep[] { breps[1], breps[0] };
      }
      else
        throw new Exception("intersection splitting error");
    }
    return new Brep[] { };
  }

  public void NDBG_from_joint()
  {
    if (joints.Count > 0)
    {
      if (this == joints[0].member1) NDBG = joints[0].NDBG_m1;
      else NDBG = joints[0].NDBG_m2;
    }
    else
      NDBG = new Region();
      //throw new Exception("NDBG from empty joints");
    if (joints.Count > 1)
    {
      for (int i = 1; i < joints.Count; i++)
      {
        if (this == joints[i].member1) NDBG=NDBG.boolean_intersect(joints[i].NDBG_m1);
        else NDBG=NDBG.boolean_intersect(joints[i].NDBG_m2);
        if (NDBG==null) return;
      }
    }
  }


  public static intersect_type[] intersect_check(Member m1, Member m2)//,RhinoDoc doc)
  {
    intersect_type[] inter = new intersect_type[2];
    inter[0] = m1.intersect(m2);
    inter[1] = m2.intersect(m1);
    return inter;
  }

  public static intersect_type[] intersect(Member m1, Member m2)//,RhinoDoc doc)
  {
    intersect_type[] inter = new intersect_type[2];
    inter[0] = m1.intersect(m2);
    inter[1] = m2.intersect(m1);
    if ((int)inter[0] >= 8 || (int)inter[1] >= 8)
```

```
            throw new Exception("fabrication constraints violation");
        return inter;
    }

    public int point_within_zone(double p, List<double> zone)
    {
        if (p <= zone[0]) return 0;
        for (int i=0; i < zone.Count-1; i++)
        {
            if (p > zone[i] && p <= zone[i + 1])
                return i + 1;
        }
        return zone.Count;
    }

    //public intersect_type intersect(Member m/*,RhinoDoc doc*/)
    public intersect_type intersect(Member m)
    {
        Vector3d[] corner = m.corner_points();
        for (int i = 0; i < 8; i++) corner[i] = change_basis(corner[i]);
        //for (int i = 0; i < 8; i++) doc.Objects.AddPoint(corner[i].X,corner[i].Y,0);
        List<double> y1 = new List<double>();
        List<double> y2 = new List<double>();
        int within = 0;
        if (VecUtil.vector_perpendicular((corner[4] - corner[0]), xDir))
        {
            List<double> x = new List<double>();
            for (int i = 0; i < 4; i++) x.Add(corner[i].X);
            if (x[0] < 0 && x[3] > 0) within += 2;
            if (x[0] < width && x[3] > width) within += 2;
        }
        else
        {
            for (int i = 0; i < 4; i++)
            {
                double a=(corner[i + 4].Y - corner[i].Y) / (corner[i + 4].X - corner[i].X);
                double b=(corner[i + 4].X * corner[i].Y - corner[i].X * corner[i + 4].Y) / (corner[i + 4].X - corner[i].X);
                y1.Add(b);
                y2.Add(a*width+b);
                //doc.Objects.AddPoint(0, b, 0);
                //doc.Objects.AddPoint(width, a * width + b, 0);
            }
            y1.Sort();
            y2.Sort();

            int[] corner_zone = new int[4];
            corner_zone[0] = point_within_zone(0, y1);
            corner_zone[1] = point_within_zone(height, y1);
            corner_zone[2] = point_within_zone(height, y2);
            corner_zone[3] = point_within_zone(0, y2);
            bool[] corner_within = new bool[4];
            for (int i = 0; i < corner_zone.Length; i++)
            {
                if (corner_zone[i] != 0 && corner_zone[i] != 4)
                {
```

```
                corner_within[i] = true;
                within++;
            }
        }

        if (within == 0) return intersect_type.None;
        else if (within == 3) return intersect_type.Three;
        else if (within == 2)
        {
            for (int i = 0; i < corner_within.Length; i++)
            {
                if (corner_within[i] && corner_within[(i + 1) % 4])
                {
                    if (corner_zone[i] != corner_zone[(i + 1) % 4]) return intersect_type.Inconsistent;
                    return (intersect_type)(i + 4);
                }
            }
            return intersect_type.Opposite;

        }
        else if (within == 1)
        {
            if (corner_within[0]) return intersect_type.Corner_BL;
            else if (corner_within[1]) return intersect_type.Corner_TL;
            else if (corner_within[2]) return intersect_type.Corner_TR;
            else if (corner_within[3]) return intersect_type.Corner_BR;
        }
    }
    return intersect_type.None;
}


public static bool is_intersect(Member A,Member B)
{
    Vector3d T = B.mid - A.mid;
    if (Math.Abs(2 * T * A.xDir) > A.width + Math.Abs(B.width * A.xDir * B.xDir) + Math.Abs(B.height * A.xDir *
B.yDir) + Math.Abs(B.depth * A.xDir * B.zDir)) return false;
    if (Math.Abs(2 * T * A.yDir) > A.height + Math.Abs(B.width * A.yDir * B.xDir) + Math.Abs(B.height * A.yDir *
B.yDir) + Math.Abs(B.depth * A.yDir * B.zDir)) return false;
    if (Math.Abs(2 * T * A.zDir) > A.depth + Math.Abs(B.width * A.zDir * B.xDir) + Math.Abs(B.height * A.zDir *
B.yDir) + Math.Abs(B.depth * A.zDir * B.zDir)) return false;
    if (Math.Abs(2 * T * B.xDir) > B.width + Math.Abs(A.width * A.xDir * B.xDir) + Math.Abs(A.height * A.yDir *
B.xDir) + Math.Abs(A.depth * A.zDir * B.xDir)) return false;
    if (Math.Abs(2 * T * B.yDir) > B.height + Math.Abs(A.width * A.xDir * B.yDir) + Math.Abs(A.height * A.yDir *
B.yDir) + Math.Abs(A.depth * A.zDir * B.yDir)) return false;
    if (Math.Abs(2 * T * B.zDir) > B.depth + Math.Abs(A.width * A.xDir * B.zDir) + Math.Abs(A.height * A.yDir *
B.zDir) + Math.Abs(A.depth * A.zDir * B.zDir)) return false;
    if (Math.Abs(2 * T * Vector3d.CrossProduct(A.xDir, B.xDir)) > Math.Abs(A.height * A.zDir * B.xDir) +
Math.Abs(A.depth * A.yDir * B.xDir) + Math.Abs(B.height * A.xDir * B.zDir) + Math.Abs(B.depth * A.xDir * B.yDir))
return false;
    if (Math.Abs(2 * T * Vector3d.CrossProduct(A.xDir, B.yDir)) > Math.Abs(A.height * A.zDir * B.yDir) +
Math.Abs(A.depth * A.yDir * B.yDir) + Math.Abs(B.width * A.xDir * B.zDir) + Math.Abs(B.depth * A.xDir * B.xDir))
return false;
```

```
        if (Math.Abs(2 * T * Vector3d.CrossProduct(A.xDir, B.zDir)) > Math.Abs(A.height * A.zDir * B.zDir) +
Math.Abs(A.depth * A.yDir * B.zDir) + Math.Abs(B.width * A.xDir * B.yDir) + Math.Abs(B.height * A.xDir * B.xDir))
return false;
        if (Math.Abs(2 * T * Vector3d.CrossProduct(A.yDir, B.xDir)) > Math.Abs(A.width * A.zDir * B.xDir) +
Math.Abs(A.depth * A.xDir * B.xDir) + Math.Abs(B.height * A.yDir * B.zDir) + Math.Abs(B.depth * A.yDir * B.yDir))
return false;
        if (Math.Abs(2 * T * Vector3d.CrossProduct(A.yDir, B.yDir)) > Math.Abs(A.width * A.zDir * B.yDir) +
Math.Abs(A.depth * A.xDir * B.yDir) + Math.Abs(B.width * A.yDir * B.zDir) + Math.Abs(B.depth * A.yDir * B.xDir))
return false;
        if (Math.Abs(2 * T * Vector3d.CrossProduct(A.yDir, B.zDir)) > Math.Abs(A.width * A.zDir * B.zDir) +
Math.Abs(A.depth * A.xDir * B.zDir) + Math.Abs(B.width * A.yDir * B.yDir) + Math.Abs(B.height * A.yDir * B.xDir))
return false;
        if (Math.Abs(2 * T * Vector3d.CrossProduct(A.zDir, B.xDir)) > Math.Abs(A.width * A.yDir * B.xDir) +
Math.Abs(A.height * A.xDir * B.xDir) + Math.Abs(B.height * A.zDir * B.zDir) + Math.Abs(B.depth * A.zDir * B.yDir))
return false;
        if (Math.Abs(2 * T * Vector3d.CrossProduct(A.zDir, B.yDir)) > Math.Abs(A.width * A.yDir * B.yDir) +
Math.Abs(A.height * A.xDir * B.yDir) + Math.Abs(B.width * A.zDir * B.zDir) + Math.Abs(B.depth * A.zDir * B.xDir))
return false;
        if (Math.Abs(2 * T * Vector3d.CrossProduct(A.zDir, B.zDir)) > Math.Abs(A.width * A.yDir * B.zDir) +
Math.Abs(A.height * A.xDir * B.zDir) + Math.Abs(B.width * A.zDir * B.yDir) + Math.Abs(B.height * A.zDir * B.xDir))
return false;
        return true;
    }

    public Vector3d[] corner_points()
    {
        Vector3d[] p = new Vector3d[8];
        p[0] = start - (width / 2) * xDir - (height / 2) * yDir;
        p[1] = start - (width / 2) * xDir + (height / 2) * yDir;
        p[2] = start + (width / 2) * xDir + (height / 2) * yDir;
        p[3] = start + (width / 2) * xDir - (height / 2) * yDir;
        p[4] = end - (width / 2) * xDir - (height / 2) * yDir;
        p[5] = end - (width / 2) * xDir + (height / 2) * yDir;
        p[6] = end + (width / 2) * xDir + (height / 2) * yDir;
        p[7] = end + (width / 2) * xDir - (height / 2) * yDir;
        return p;
    }

    public Vector3d change_basis(Vector3d vec)
    {
        Vector3d origin = start - (width / 2) * xDir - (height / 2) * yDir;
        vec -= origin;
        Transform t= Transform.ChangeBasis(new Vector3d(1, 0, 0), new Vector3d(0, 1, 0), new Vector3d(0, 0, 1),
xDir, yDir, zDir);
        vec.Transform(t);
        return vec;
    }


  }
}
```

Graph For DBG

```
using System;
using System.Collections.Generic;
using System.Text;
namespace Disassembler
{
  public class Graph<T>
  {
    public List<Node<T>> nodes { get; set; }

    public Graph() { nodes = new List<Node<T>>(); }

    public Graph<T> reverse()
    {
      Graph<T> graph=new Graph<T>();
      for (int i = 0; i < this.nodes.Count; i++)
        graph.nodes.Add(new Node<T>(this.nodes[i].data, i));
      for (int i = 0; i < this.nodes.Count;i++)
      {
        int to = nodes[i].index;
        for (int j = 0; j < this.nodes[i].links.Count; j++)
        {
          int from = this.nodes[i].links[j].index;
          graph.nodes[from].links.Add(graph.nodes[to]);
        }
      }
      return graph;
    }
    public List<int> DFS_index(Node<T> start)
    {
      start.visited=true;
      List<int> list=new List<int>();
      foreach (Node<T> l in start.links)
        if (!l.visited) list.AddRange(DFS_index(l));
      list.Add(start.index);
      return list;
    }
    public List<int> DFS_index(Node<T> start,List<int> group)
    {
      start.visited=true;
      List<int> list=new List<int>();
      foreach (Node<T> l in start.links)
        if (!l.visited && group.Contains(l.index)) list.AddRange(DFS_index(l,group));
      list.Add(start.index);
      return list;
    }
    public List<Node<T>> DFS(Node<T> start)
    {
      start.visited=true;
      List<Node<T>> list=new List<Node<T>>();
      foreach (Node<T> l in start.links)
        if (!l.visited) list.AddRange(DFS(l));
      list.Add(start);
      return list;
    }
```

```
  public List<List<int>> SCC() //Kosaraju's theorom
  {
    List<List<int>> scc=new List<List<int>>();
    List<List<int>> group=new List<List<int>>();
    for(int i=0;i<nodes.Count;i++)
    {
      if(!nodes[i].visited)
        group.Add(DFS_index(nodes[i]));
    }
    Graph<T> reverse_graph=this.reverse();
    for(int i=0;i<group.Count;i++)
    {
      for (int j = group[i].Count - 1; j >= 0; j--)
      {
        if (!reverse_graph.nodes[group[i][j]].visited)
          scc.Add(reverse_graph.DFS_index(reverse_graph.nodes[group[i][j]], group[i]));
      }
    }
    foreach (Node<T> n in nodes) n.visited = false;
    return scc;
  }
  public bool is_SCC_outward(List<int> scc)
  {
    foreach(int index in scc)
    {
      foreach(Node<T> l in nodes[index].links)
        if (!scc.Contains(l.index)) return true;
    }
    return false;
  }
  public void add_link(int i, int j) { nodes[i].links.Add(nodes[j]); }

  public void reset_links()
  {
    foreach (Node<T> n in nodes)
      n.links = new List<Node<T>>();
  }

}
public class Node<T>
{
  public T data { get; set; }
  public List<Node<T>> links { get; set; }
  public int index { get; set; }
  public bool visited { get; set; }

  public Node(T data,int index)
  {
    this.data = data;
    this.index = index;
    links = new List<Node<T>>();
    visited = false;
  }
}
```

Spherical Geometry For NDBG

```
using System;
using System.Collections.Generic;
using System.Text;
using Rhino.Geometry;
using Rhino;
namespace Disassembler
{
  interface IGeodesic
  {
    void draw(RhinoDoc doc);
    GreatCircle to_greatcircle();
    bool is_endpoint(Vector3d pt);
    Vector3d[] intersect(GreatCircle gc);
    Vector3d[] intersect(Segment sg);
    bool is_greatcircle();
    Vector3d start_pt();
    Vector3d end_pt();
    Vector3d mid_pt();
    bool within(Vector3d vec);
    IGeodesic reverse();
  }
  class GreatCircle : IGeodesic
  {
    public Vector3d dome_normal{get; set;}

    public GreatCircle(Vector3d dome_normal) { this.dome_normal = dome_normal; }

    public GreatCircle to_greatcircle() { return this; }
    public bool within(Vector3d vec) { return VecUtil.vector_perpendicular(vec, dome_normal) ? true : false; }
    public bool is_endpoint(Vector3d pt) { return false; }
    public static bool overlap(GreatCircle g1, GreatCircle g2) { return (GreatCircle.equal(g1, g2) ||
GreatCircle.reverse(g1, g2)) ? true : false; }
    public static bool equal(GreatCircle g1, GreatCircle g2) { return (VecUtil.vector_equal(g1.dome_normal,
g2.dome_normal)) ? true : false; }
    public static bool reverse(GreatCircle g1, GreatCircle g2) { return (VecUtil.vector_equal(g1.dome_normal, -
g2.dome_normal)) ? true : false; }
    public bool is_greatcircle() { return true; }
    public Vector3d start_pt() { throw new Exception("start point not defined "); }
    public Vector3d end_pt() { throw new Exception("end point not defined "); }
    public Vector3d mid_pt() { throw new Exception("mid point not defined "); }

    public void draw(RhinoDoc doc)
    {
      Plane plane=new Plane(new Point3d(0,0,0),dome_normal);
      Circle circle = new Circle(plane, 1);
      doc.Objects.AddCircle(circle);
    }
    public IGeodesic reverse() { return new GreatCircle(-dome_normal);
    public Vector3d[] intersect(GreatCircle gc)
    {
      Vector3d cross=new Vector3d();
      if (GreatCircle.overlap(this, gc))
      {
        return new Vector3d[]{};
```

```
        }
        else
        {
            cross = Vector3d.CrossProduct(this.dome_normal, gc.dome_normal);
            cross.Unitize();
        }
        return new Vector3d[] { cross, -cross };
    }
    public Vector3d[] intersect(Segment sg)
    {
        return sg.intersect(this);
    }
}
class Segment : IGeodesic
{
    private Vector3d start;
    private Vector3d end;
    private Vector3d mid;

    public Segment(Vector3d start, Vector3d end) //no sign is implemented
    {
        this.start = start;
        this.end = end;
        this.mid = VecUtil.unitize((start + end) / 2);
    }

    public Segment(Vector3d start, Vector3d end, Vector3d mid)
        : this(start, end)
    {
        this.mid = VecUtil.unitize(mid);
        if (this.mid.Length == 0) throw new Exception("ambiguous geodesic creation: half circle");
    }
    public bool is_greatcircle() { return false; }
    public Vector3d start_pt() { return start;}
    public Vector3d end_pt() { return end;}
    public Vector3d mid_pt() { return mid;}
    public Vector3d segment_normal
    {
        get
        {
            Vector3d normal = Vector3d.CrossProduct(start, mid);
            normal.Unitize();
            return normal;
        }
    }
    public IGeodesic reverse() { return new Segment(-start, -end, -mid); }
//**********************************need test

    public bool within(Vector3d vec,double tol)
    {
        Vector3d[] cross = new Vector3d[] { Vector3d.CrossProduct(start, vec), Vector3d.CrossProduct(vec, mid),
Vector3d.CrossProduct(mid, vec), Vector3d.CrossProduct(vec, end) };
        for (int i = 0; i < cross.Length; i++) cross[i].Unitize();
        if (VecUtil.vector_equal(vec, mid) || VecUtil.vector_equal(vec, start) || VecUtil.vector_equal(vec, end))
return true;
```

```
        else if (VecUtil.vector_equal(cross[0], segment_normal,tol) && VecUtil.vector_equal(cross[1],
segment_normal,tol)) return true;
        else if (VecUtil.vector_equal(cross[2], segment_normal,tol) && VecUtil.vector_equal(cross[3],
segment_normal,tol)) return true;
        else return false;
    }
    public bool within(Vector3d vec)
    {
        Vector3d[] cross = new Vector3d[] { Vector3d.CrossProduct(start, vec), Vector3d.CrossProduct(vec, mid),
Vector3d.CrossProduct(mid, vec), Vector3d.CrossProduct(vec, end) };
        for (int i = 0; i < cross.Length; i++) cross[i].Unitize();
        if (VecUtil.vector_equal(vec, mid) || VecUtil.vector_equal(vec, start) || VecUtil.vector_equal(vec, end))
return true;
        else if (VecUtil.vector_equal(cross[0], segment_normal) && VecUtil.vector_equal(cross[1], segment_normal))
return true;
        else if (VecUtil.vector_equal(cross[2], segment_normal) && VecUtil.vector_equal(cross[3], segment_normal))
return true;
        else return false;
    }
    public void draw(RhinoDoc doc)
    {
        Arc arc = new Arc(new Point3d(start), new Point3d(mid), new Point3d(end));
        doc.Objects.AddArc(arc);
        //doc.Views.Redraw();
    }
    public void draw()
    {
        draw(RhinoDoc.ActiveDoc);
    }
    public Vector3d[] intersect(GreatCircle gc)
    {
        GreatCircle gc_self = new GreatCircle(this.segment_normal);
        Vector3d[] inter = gc.intersect(gc_self);
        List<Vector3d> pt = new List<Vector3d>();
        foreach (Vector3d p in inter)
            if (within(p)) pt.Add(p);
        return pt.ToArray();
    }
    public Segment intersect_overlap(Segment sg)
    {
        double tol = 0.001;//ovelap tolerance
**********************************************************
        GreatCircle g1 = to_greatcircle();
        GreatCircle g2 = sg.to_greatcircle();
        if (GreatCircle.overlap(g1, g2))
        {
            List<Vector3d> within1 = new List<Vector3d>();
            List<Vector3d> within2 = new List<Vector3d>();
            if (sg.within(this.start,tol)) within1.Add(this.start);
            if (sg.within(this.end, tol)) within1.Add(this.end);
            if (this.within(sg.start, tol)) within2.Add(sg.start);
            if (this.within(sg.end, tol)) within2.Add(sg.end);
            if (within1.Count == 1 && within2.Count == 1) return new Segment(within1[0], within2[0]);
            else if (within1.Count == 2 && within2.Count == 0) return this;
            else if (within1.Count == 2 && within2.Count == 0) return sg;
```

```
        }
        return null;
      }
      public Vector3d[] intersect(Segment sg)
      {
        GreatCircle g1 = to_greatcircle();
        GreatCircle g2 = sg.to_greatcircle();
        if (!GreatCircle.overlap(g1, g2))
        {
          Vector3d[] inter = g1.intersect(g2);
          List<Vector3d> pt = new List<Vector3d>();
          foreach (Vector3d p in inter)
            if (within(p) && sg.within(p)) pt.Add(p);
          return pt.ToArray();
        }
        else //overlap
        {
          List<Vector3d> within1 = new List<Vector3d>();
          List<Vector3d> within2 = new List<Vector3d>();
          if (sg.within(this.start)) within1.Add(this.start);
          if (sg.within(this.end)) within1.Add(this.end);
          if (this.within(sg.start)) within2.Add(sg.start);
          if (within1.Count == 2 && within2.Count == 2) return new Vector3d[] { this.start, this.end };
          return new Vector3d[] { };          }
      }
      public GreatCircle to_greatcircle() { return new GreatCircle(segment_normal); }

      public Vector3d dome_normal_from_pt(Vector3d pt)
      {
        Vector3d vec=segment_normal;
        if (Vector3d.Multiply(pt,vec)>0) return vec;
        else if (Vector3d.Multiply(pt,vec)<0) return -vec;
        else throw new Exception("ambiguous dome normal");
      }
      public bool is_endpoint(Vector3d pt) { return (VecUtil.vector_equal(pt, start) || VecUtil.vector_equal(pt, end)) ?
true : false; }
    }
    class Edge :IEquatable<Edge>
    {
      public IGeodesic geodesic{get; set;}
      public Nullable<bool> open {get; set;}
      public Vector3d dome_normal{get; set;}
      private bool start_open=false;
      private bool end_open=false;

      public Edge(IGeodesic geodesic, Nullable<bool> open, Vector3d dome_normal)
      {
        this.geodesic = geodesic;
        this.open = open;
        this.dome_normal = dome_normal;
      }

      public Edge(IGeodesic geodesic, Nullable<bool> open, Vector3d dome_normal, bool start_open, bool end_open)
        : this(geodesic, open, dome_normal)
      {
```

```
        this.start_open = start_open;
        this.end_open = end_open;
    }

    public Edge reverse() //**********************************need test
    {
        return new Edge(geodesic.reverse(), open, -dome_normal, start_open, end_open);
    }

    public static Segment intersect_overlap(Edge e1, Edge e2)
    {
        if (!e1.geodesic.is_greatcircle() && !e2.geodesic.is_greatcircle())
        {
            Segment s=(Segment)e1.geodesic;
            Segment overlap=s.intersect_overlap((Segment)e2.geodesic);
            return overlap;
        }
        return null;
    }
    public static Vector3d[] intersect(Edge e1,Edge e2)
    {
        if (e2.geodesic.is_greatcircle())
            return e1.geodesic.intersect((GreatCircle)e2.geodesic);
        else
            return e1.geodesic.intersect((Segment)e2.geodesic);
    }
    public void draw(RhinoDoc doc) { geodesic.draw(doc); }

    public bool Equals(Edge e)
    {
        if (geodesic.is_greatcircle() && e.geodesic.is_greatcircle())
            return (VecUtil.vector_equal(((GreatCircle)geodesic).dome_normal,
((GreatCircle)geodesic).dome_normal)) ? true : false;
        else if (!geodesic.is_greatcircle() && !e.geodesic.is_greatcircle())
        {
            if (VecUtil.vector_equal(geodesic.mid_pt(), e.geodesic.mid_pt()))
            {
                if (VecUtil.vector_equal(geodesic.start_pt(), e.geodesic.start_pt()) ||
VecUtil.vector_equal(geodesic.end_pt(), e.geodesic.end_pt()))
                    return true;
                else if (VecUtil.vector_equal(geodesic.start_pt(), e.geodesic.end_pt()) ||
VecUtil.vector_equal(geodesic.end_pt(), e.geodesic.start_pt()))
                    return true;
            }
        }
        return false;
    }
}
struct IntersectEvent
{
    private Vector3d m_pt;
    private Edge m_edge1;
    private Edge m_edge2;
    public Vector3d pt { get { return m_pt; } }
    public Edge edge1 { get { return m_edge1; } }
```

```
    public Edge edge2 { get { return m_edge2; } }
    public IntersectEvent(Vector3d p, Edge e1, Edge e2)
    {
      m_pt = p;
      m_edge1=e1;
      m_edge2=e2;
    }
  }
  class Region
  {
    private List<Edge> edges=new List<Edge>();
    private List<Vector3d> points=new List<Vector3d>();
    private bool degenerate_edge=false;
    private bool ball=false;

    public Region() { ball = true; }
    public Region(List<Edge> edges) { this.edges = edges; }
    public Region(Edge[] edges) { this.edges = new List<Edge>(edges); }
    public Region(List<Edge> edges, bool degenerate_edge) : this(edges) { this.degenerate_edge =
degenerate_edge; }
    public Region(Edge[] edges, bool degenerate_edge) : this(edges) { this.degenerate_edge = degenerate_edge; }
    public Region(List<Vector3d> points) { this.points = points; }
    public Region(Vector3d[] points) { this.points = new List<Vector3d>(points); }
    public void add_edge(Edge edge) { edges.Add(edge); }
    public bool is_degenerate_edge() { return degenerate_edge; }
    public bool is_dome() { return (edges.Count == 1) ? true : false; }
    public bool is_point() { return (points.Count > 0) ? true : false; }
    public bool is_ball() { return ball; }
    public List<Vector3d> get_points() { return points; }
    public List<Edge> get_edges() { return edges; }
    public void draw(RhinoDoc doc)
    {
      if (edges.Count > 0)
        foreach (Edge e in edges) e.draw(doc);
      if (points.Count > 0)
        foreach (Vector3d p in points) doc.Objects.AddPoint(new Point3d(p));
      doc.Views.Redraw();
    }
    public Region reverse()
    {
      if (is_point())
      {
        List<Vector3d> pts = new List<Vector3d>();
        for (int i = 0; i < points.Count; i++)
          pts.Add(-points[i]);
        return new Region(pts);
      }
      else
      {
        List<Edge> e=new List<Edge>();
        for (int i = 0; i < edges.Count; i++)
          e.Add(edges[i].reverse());
        return new Region(e,degenerate_edge);
      }
    }
```

```
    public bool within(Vector3d pt)
    {
      bool flag = true;

      if (is_point())
      {
        flag = false;
        foreach(Vector3d p in points)
          if (VecUtil.vector_equal(p,pt)) return true;
      }
      else if (is_degenerate_edge())
      {
        flag = false;
        foreach(Edge e in edges)
          if (e.geodesic.within(pt)) return true;
      }
      else
      {
        foreach (Edge e in edges)
        {
          if (e.open.HasValue)
          {
            if ((bool)e.open)
              flag = flag & Vector3d.Multiply(pt, e.dome_normal) > 0.001;
            else
              flag = flag & Vector3d.Multiply(pt, e.dome_normal) >= -0.001;
          }
          else
            throw new Exception("edge.open is null");
        }
      }
      return flag;
    }
    public Vector3d centroid
    {
      get
      {
        Vector3d sum=new Vector3d();
        foreach (Edge e in edges)
          sum += (e.geodesic.start_pt() + e.geodesic.end_pt()) / 2;
        sum /= edges.Count;
        return sum;
      }
    }
    public Edge[] overlap(Region region)
    {
      foreach (Edge e1 in edges)
      {
        foreach (Edge e2 in region.edges)
          if (GreatCircle.overlap(e1.geodesic.to_greatcircle(), e2.geodesic.to_greatcircle())) return new Edge[] { e1,
e2 };
      }
      return new Edge[] { };
    }
```

```
    public Region[] split_by_dome(Region region)
    {
      IntersectEvent[] intersect = Region.intersect(this, region);
      if (!region.is_dome()) throw new Exception("splitter is not dome");
      if (this.degenerate_edge & intersect.Length > 0)
      {
        if (intersect.Length == 2) //great circle intersect by dome
        {
          Vector3d mid = VecUtil.unitize(Vector3d.CrossProduct(intersect[0].pt, edges[0].dome_normal));
          Edge e1 = new Edge(new Segment(intersect[0].pt, intersect[1].pt, mid), false, new Vector3d());
          Edge e2 = new Edge(new Segment(intersect[0].pt, intersect[1].pt, -mid), false, new Vector3d());
          Region r1 = new Region(new Edge[] { e1 }, true);
          Region r2 = new Region(new Edge[] { e2 }, true);
          return new Region[] { r1, r2 };
        }
        else if (intersect.Length == 1)
        {
          if (VecUtil.vector_equal(intersect[0].pt, edges[0].geodesic.start_pt()) ||
VecUtil.vector_equal(intersect[0].pt, edges[0].geodesic.end_pt()))
            return new Region[] { this };
          else
          {
            Edge e1 = new Edge(new Segment(edges[0].geodesic.start_pt(), intersect[0].pt), false, new Vector3d());
            Edge e2 = new Edge(new Segment(intersect[0].pt, edges[0].geodesic.end_pt()), false, new Vector3d());
            Region r1 = new Region(new Edge[] { e1 }, true);
            Region r2 = new Region(new Edge[] { e2 }, true);
            return new Region[] { r1, r2 };
          }
        }
        else
          return new Region[] { };
      }
      else if (intersect.Length > 0)
      {
        if (intersect.Length == 2 && !is_dome()) //region dome
        {
          int[] index = new int[2];
          for (int i = 0; i < intersect.Length; i++)
            index[i] = edges.IndexOf(intersect[i].edge1);
          Vector3d normal = new Segment(intersect[0].pt, intersect[1].pt).segment_normal;
          normal = (Vector3d.Multiply(edges[0].geodesic.start_pt(), normal) > 0) ? normal : -normal;
          Edge edge_split1 = new Edge(new Segment(intersect[0].edge1.geodesic.start_pt(), intersect[0].pt),
intersect[0].edge1.open, intersect[0].edge1.dome_normal);
          Edge edge_split2 = new Edge(new Segment(intersect[0].pt, intersect[1].pt), null, normal); //undecided
open....
          Edge edge_split3 = new Edge(new Segment(intersect[1].pt, intersect[1].edge1.geodesic.end_pt()),
intersect[1].edge1.open, intersect[1].edge1.dome_normal);
          Edge edge_split4 = new Edge(new Segment(intersect[0].pt, intersect[0].edge1.geodesic.end_pt()),
intersect[0].edge1.open, intersect[0].edge1.dome_normal);
          Edge edge_split5 = new Edge(new Segment(intersect[1].edge1.geodesic.start_pt(), intersect[1].pt),
intersect[1].edge1.open, intersect[1].edge1.dome_normal);
          Edge edge_split6 = new Edge(new Segment(intersect[1].pt, intersect[0].pt), null, -normal);
          List<Edge> edge1 = new List<Edge>();
          edge1.AddRange(edges.GetRange(0, index[0]));
          edge1.AddRange(new Edge[] { edge_split1, edge_split2, edge_split3 });
```

```
            edge1.AddRange(edges.GetRange(index[1] + 1, edges.Count - 1 - index[1]));
            List<Edge> edge2 = new List<Edge>();
            edge2.Add(edge_split4);
            edge2.AddRange(edges.GetRange(index[0] + 1, index[1] - index[0] - 1));
            edge2.AddRange(new Edge[] { edge_split5, edge_split6 });
            return new Region[] { new Region(edge1), new Region(edge2) };
          }
          else if (intersect.Length == 2 && is_dome()) //dome/dome intersect need to add dome vector
          {
            Vector3d midSelf1 = Vector3d.CrossProduct(edges[0].dome_normal, intersect[0].pt);
            Vector3d midSelf2 = Vector3d.CrossProduct(edges[0].dome_normal, intersect[1].pt);
            Vector3d midSplit1 = Vector3d.CrossProduct(region.edges[0].dome_normal, intersect[0].pt);
            Vector3d midSplit2 = Vector3d.CrossProduct(region.edges[0].dome_normal, intersect[1].pt);
            Vector3d midSplit = (Vector3d.Multiply(edges[0].dome_normal, midSplit1) > 0) ? midSplit1 : midSplit2;
            Vector3d normal = region.edges[0].dome_normal;
            normal = (Vector3d.Multiply(midSelf1, normal) > 0) ? normal : -normal;
            Edge edge_split1 = new Edge(new Segment(intersect[0].pt, intersect[1].pt, midSelf1), edges[0].open,
edges[0].dome_normal);
            Edge edge_split2 = new Edge(new Segment(intersect[1].pt, intersect[0].pt, midSplit), null, normal);
            Edge edge_split3 = new Edge(new Segment(intersect[1].pt, intersect[0].pt, midSelf2), edges[0].open,
edges[0].dome_normal);
            Edge edge_split4 = new Edge(new Segment(intersect[0].pt, intersect[1].pt, midSplit), null, -normal);
            Edge[] edge1 = new Edge[] { edge_split1, edge_split2 };
            Edge[] edge2 = new Edge[] { edge_split3, edge_split4 };
            return new Region[] { new Region(edge1), new Region(edge2) };
          }
          else
          {
            return new Region[] { };
          }
        }
        else
          return new Region[] { };
      }
      public static IntersectEvent[] intersect(Region r1, Region r2)
      {
        List<IntersectEvent> events = new List<IntersectEvent>();
        foreach (Edge e1 in r1.edges)
        {
          foreach (Edge e2 in r2.edges)
          {
            Vector3d[] inter = Edge.intersect(e1, e2);
            if (inter.Length > 0)
            {
              foreach (Vector3d p in inter)
                events.Add(new IntersectEvent(p, e1, e2));
            }
          }
        }
        return events.ToArray();
      }
      public Region boolean_intersect(Region region)      {
        if (region.is_point())
        {
          List<Vector3d> list = new List<Vector3d>();
```

```
    foreach (Vector3d p in region.points)
        if (this.within(p)) list.Add(p);
    if (list.Count > 0) return new Region(list);
  }
  else if (this.is_point())
  {
    return region.boolean_intersect(this);
  }
  else if (this.is_degenerate_edge())
  {
    if (region.is_degenerate_edge())
    {
      Edge[] overlap = this.overlap(region);
      if (overlap.Length > 0)
      {
        if (this.edges[0] == region.edges[0]) return this;
        else
        {
          Segment s = Edge.intersect_overlap(this.edges[0], region.edges[0]);
          if (s != null) return new Region(new Edge[] { new Edge(s, false, new Vector3d()) }, true);

          Vector3d[] inter=Edge.intersect(this.edges[0],region.edges[0]);
          if (inter.Length > 0)
            return new Region(inter);
          else return null;
        }

      }
      else
      {
        Vector3d[] points;
        if (this.edges[0].geodesic.is_greatcircle())
          points = region.edges[0].geodesic.intersect((GreatCircle)this.edges[0].geodesic);
        else
          points = region.edges[0].geodesic.intersect((Segment)this.edges[0].geodesic);
        if (points.Length > 0)
          return new Region(points);
      }
    }
    else //edge to region case use edge and boolean_diff_dome?
    {
      Region r = Region.create_dome(-region.edges[0].dome_normal, true); ;
      if (region.edges.Count > 0)
      {
        for (int i = 1; i < region.edges.Count; i++)
        {
          Region diff = Region.create_dome(-region.edges[i].dome_normal, true);
          r = r.boolean_diff_dome(diff);
        }
      }
    }
  }
  else
  {
    if (region.is_degenerate_edge())
```

```
        return region.boolean_intersect(this);
      else
        throw new Exception("intersection overlap not implemneted");
    }
    return null;
}
public Region boolean_diff_dome(Region dome)
{

    if (is_point())
    {
      List<Vector3d>  list=new List<Vector3d>();
      foreach(Vector3d p in points)
        if (!dome.within(p)) list.Add(p);
      if (list.Count==0)
        return null;
      else
        return new Region(list);
    }
    else if (is_degenerate_edge())
    {
      IntersectEvent[] intersect=Region.intersect(this,dome);
      if (intersect.Length==2)
      {
        if (edges[0].geodesic.is_endpoint(intersect[0].pt) && edges[0].geodesic.is_endpoint(intersect[1].pt))
        {
          if (dome.within(edges[0].geodesic.mid_pt()))
            return new Region(new Vector3d[]{intersect[0].pt,intersect[1].pt});
          else
            return this;
        }
        else//dome dome
        {
          Vector3d mid=VecUtil.unitize(Vector3d.CrossProduct(intersect[0].pt,edges[0].dome_normal));
          if (!dome.within(mid))
          {
            Edge e1=new Edge(new Segment(intersect[0].pt,intersect[1].pt,mid),false,new Vector3d());
            return new Region(new Edge[]{e1},true);
          }
          else
          {
            Edge e2=new Edge(new Segment(intersect[0].pt,intersect[1].pt,-mid),false,new Vector3d());
            return new Region(new Edge[]{e2},true);
          }
        }
      }
      else if (intersect.Length==1)
      {
        if (edges[0].geodesic.is_endpoint(intersect[0].pt))
        {
          if (dome.within(edges[0].geodesic.mid_pt()))
            return new Region(new Vector3d[]{intersect[0].pt});
          else
            return this;
        }
```

```
        else
        {
          Edge e1=new Edge(new Segment(edges[0].geodesic.start_pt(),intersect[0].pt),false,new Vector3d());
          Edge e2=new Edge(new Segment(intersect[0].pt,edges[0].geodesic.end_pt()),false,new Vector3d());
          if (!dome.within(e1.geodesic.mid_pt()))
            return new Region(new Edge[]{e1},true);
          else
            return new Region(new Edge[]{e2},true);
        }
      }
      else// no intersection what about overlap?
      {
        if (!dome.within(centroid))
          return this;
        else
          return null;
      }
    }

    else //if not degenerate edge or point
    {
      Edge[] e=overlap(dome);
      if (e.Length>0)
      {
        if (VecUtil.vector_equal(e[0].dome_normal,e[1].dome_normal))
          if ((!(bool)e[0].open) && (bool)e[1].open)
            return new Region(new Edge[]{e[0]},true);
          else
            return null;
        else //normal reverse, do nothing, need to consider edge open condition
          return this;
      }
      else
      {
        Region[] split=split_by_dome(dome);
        if (split.Length>0)
        {
          Region r1=split[0];
          Region r2=split[1];
          Region r= (!dome.within(r1.centroid))? r1: r2;
          for(int i=0;i<r.edges.Count;i++)
            if (!r.edges[i].open.HasValue) r.edges[i].open=!dome.edges[0].open;
          return r;
        }
        else
        {
          if (!dome.within(centroid))
            return this;
          else
            return null;
        }
      }
    }
  } //public Region[]boolean_diff_dome(Region dome)
```

```
    public static Region create_dome(Vector3d normal,bool open){return new Region(new Edge[]{new Edge(new
GreatCircle(normal),open,normal)});}
    public static Region create_degenerate_edge(IGeodesic edge){return new Region(new Edge[]{new
Edge(edge,false,new Vector3d())});}
    public static Region create_point(Vector3d point){return new Region(new Vector3d[]{VecUtil.unitize(point)});}

    public static Region create_polygon(Segment[] segments,bool open)
    {
      List<Edge> boundaries=new List<Edge>();
      foreach(Segment s in segments)
        boundaries.Add(new Edge(s,open,new Vector3d()));
      Vector3d centroid=new Region(boundaries).centroid;
      boundaries.Clear();
      foreach(Segment s in segments)
        boundaries.Add(new Edge(s,open,s.dome_normal_from_pt(centroid)));
      return new Region(boundaries);
    }//dome normal not assigned

    public static Region get_free_direction(Vector3d[] normals) //input blocking normals
    {
      Region r=Region.create_dome(-normals[0],false);
      if (normals.Length > 1)
      {
        for (int i = 1; i < normals.Length; i++)
        {
          if (r == null)
            return r;
          else
            r = r.boolean_diff_dome(Region.create_dome(normals[i], true));
        }
      }
      return r;
    }
  }//class Region
}//namespace
```

Genetic Algorithm

```
public delegate int FitnessFunction(int[] code);

  public class GA
  {
    private double m_Goal = 100; //set the goal to stop GA
    private int m_MaxLoop = 150;

    private int m_FitSize = 20;
    private int m_AllSize = 100;

    private double m_MutationRate = 0.2;
    private static int m_GeneLength;

    private double m_TotalFitness;  //used for roulette selection
    private Random Rnd = new Random();

    private Gene[] m_Population;
    private FitnessFunction m_FitFunction;

    public GA(int geneLength, FitnessFunction fit, double goal)
    {
      m_Goal = goal;
      m_FitFunction = fit;
      m_GeneLength = geneLength;
      m_Population = new Gene[m_AllSize];
      for (int i = 0; i < m_AllSize; i++)
      {
        m_Population[i] = new Gene(m_GeneLength);
      }
    }

    public GA(int geneLength, FitnessFunction fit, double goal, int[] code)
    {
      m_Goal = goal;
      m_FitFunction = fit;
      m_GeneLength = geneLength;
      m_Population = new Gene[m_AllSize];
      m_Population[0] = new Gene(code);
      for (int i = 1; i < m_AllSize; i++)
      {
        m_Population[i] = new Gene(m_GeneLength);
      }
    }

    public GA(int geneLength,FitnessFunction fit, double goal,int fitSize, int allSize, double mutationRate, int
maxLoop)
    {
      m_Goal = goal;
      m_FitFunction = fit;
      m_GeneLength = geneLength;
      m_FitSize = fitSize;
      m_AllSize = allSize;
      m_MutationRate = mutationRate;
      m_MaxLoop = maxLoop;
```

```
        m_Population = new Gene[m_AllSize];
        for (int i = 0; i < m_AllSize; i++)
        {
          m_Population[i] = new Gene(m_GeneLength);
        }
      }

      public double Run(out int[] code)  //return fitness and
      {
        double bestFit = 0;

        for (int i = 0; i < m_MaxLoop; i++)
        {
          double averageFitness = m_TotalFitness / m_AllSize;
          SortPopulationByFitness();
          bestFit = m_TotalFitness - m_Population[0].Fitness;
          Rhino.RhinoApp.WriteLine(i.ToString() + " generation: best score " + bestFit.ToString() + "  average score " +
averageFitness.ToString());
          if (bestFit >= m_Goal)
            break;
          else
            CreateNewGeneration();
        }
        code = m_Population[0].GeneCode;
        return bestFit;
      }
      public double Run(out int[] code,out double average,out int first)  //return fitness and
      {
        double bestFit = 0;
        double sum=0;
        int found=-1;

        for (int i = 0; i < m_MaxLoop; i++)
        {
          CalculateFitness();
          for (int j = m_Population.Length / 2; j < m_Population.Length; j++)
          {
            sum += m_Population[j].Fitness;
            if (found == -1 && m_Population[j].Fitness == m_Goal)
            {
              if (i == 0) found = j + 1 - m_Population.Length / 2;
              else found = i * (m_Population.Length / 2) + j;
            }
          }
          SortPopulationByFitness1();
          bestFit = m_TotalFitness - m_Population[0].Fitness;
          CreateNewGeneration();
        }
        first = found;
        code = m_Population[0].GeneCode;
        average = sum / (m_Population.Length *m_MaxLoop/ 2);
        return bestFit;
      }
```

```
    private void CalculateFitness()
    {
      for (int i = 0; i < m_Population.Length; i++)
      {
        Rhino.RhinoApp.Wait();
        m_Population[i].Fitness = m_FitFunction(m_Population[i].GeneCode);
      }
    }
    private void SortPopulationByFitness1()
    {
      m_TotalFitness = 0;
      for (int i = 0; i < m_Population.Length; i++)
        m_TotalFitness += m_Population[i].Fitness;

      Array.Sort(m_Population);

      double culmFitness = m_TotalFitness;
      for (int i = 0; i < m_Population.Length; i++)  // for roulette selection, fitness is now culmulative fitness
      {
        culmFitness -= m_Population[i].Fitness;
        m_Population[i].Fitness = culmFitness;
      }
    }
    public double Run(out int[] code, int static_count)  //return fitness and
    {
      double bestFit = 0;
      double prev_bestFit=0;
      double count = 0;

      for (int i = 0; i < m_MaxLoop; i++)
      {
        SortPopulationByFitness();
        bestFit = m_TotalFitness - m_Population[0].Fitness;
        double averageFitness = m_TotalFitness / m_AllSize;
        if (bestFit == prev_bestFit) count += 1;
        else count = 0;
        prev_bestFit = bestFit;
        if (count > static_count) break;
        Rhino.RhinoApp.WriteLine(i.ToString() + " generation: best score " + bestFit.ToString() + "  average score " +
averageFitness.ToString());
        if (bestFit >= m_Goal)
          break;
        else
          CreateNewGeneration();
      }
      code = m_Population[0].GeneCode;
      return bestFit;
    }
    public void CreateNewGeneration()
    {
      Gene[] newPopulation = new Gene[m_AllSize];

      for (int i = 0; i < m_AllSize; i += 2)  //create new generation
      {
        if (i < m_FitSize)
```

```
            {
                newPopulation[i] = m_Population[i];
                newPopulation[i + 1] = m_Population[i + 1];
                if (i > 0)                                          {
                    newPopulation[i].Mutate(m_MutationRate);
                    newPopulation[i + 1].Mutate(m_MutationRate);
                }
                else
                    newPopulation[i + 1].Mutate(m_MutationRate);
            }
            else
            {
                //Gene[] pair = Gene.Crossover(RouletteSelection(), RouletteSelection());
                Gene[] pair = Gene.Crossover(RankSelection(), RankSelection());
                pair[0].Mutate(m_MutationRate);
                pair[1].Mutate(m_MutationRate);
                newPopulation[i] = pair[0];
                newPopulation[i + 1] = pair[1];
            }

        }
        m_Population = newPopulation;
    }
    private Gene RouletteSelection() //works on array that is sorted and fitness is culmulative fitness
    {
        Gene g = new Gene();
        g.Fitness = Rnd.NextDouble() * m_TotalFitness; //creating a dummy gene that holds the value for binary
search
        int selection = Array.BinarySearch<Gene>(m_Population, g);
        selection = (selection < 0) ? ~selection : selection;
        return m_Population[selection];
    }

    private Gene RankSelection()
    {
        int selection = (int)(Rnd.NextDouble() * m_AllSize * m_AllSize);
        selection = ((int)Math.Sqrt(selection)) + 1;
        return m_Population[m_AllSize - selection];
    }
    private void SortPopulationByFitness()
    {
        m_TotalFitness = 0;
        for (int i = 0; i < m_Population.Length; i++)
        {
            Rhino.RhinoApp.Wait();
            m_Population[i].Fitness = m_FitFunction(m_Population[i].GeneCode);
            m_TotalFitness += m_Population[i].Fitness;
        }

        Array.Sort(m_Population);

        double culmFitness = m_TotalFitness;
        for (int i = 0; i < m_Population.Length; i++)  // for roulette selection, fitness is now culmulative fitness
        {
            culmFitness -= m_Population[i].Fitness;
```

```
            m_Population[i].Fitness = culmFitness;
        }
    }
    public void SetFitnessFunction(FitnessFunction f)
    {
        m_FitFunction = f;
    }

    public double MutationRate
    {
        set { m_MutationRate = value; }
    }
}
public class Gene : IComparable<Gene>
{
    private double m_Fitness;
    private int[] m_GeneCode;
    private static Random Rnd = new Random();
    private static int m_Length;
    private static int m_upper;
    public Gene() //set the static length before using this constructor
    {
        m_GeneCode = new int[m_Length];
    }

    public Gene(int length)
    {
        m_Length = length;
        m_GeneCode = new int[length];

        for (int i = 0; i < length; i++)
        {
            m_GeneCode[i] = (int)(Rnd.NextDouble() * 3);
        }
    }
    public Gene(int[] code)
    {
        m_Length = code.Length;
        m_GeneCode = code;
    }
    public Gene(int length,int upper)
    {
        m_Length = length;
        m_GeneCode = new int[length];
        m_upper = upper;
        for (int i = 0; i < length; i++)
        {
            m_GeneCode[i] = (int)(Rnd.NextDouble() * upper);
        }
    }
    public int[] GeneCode
    {
        get { return m_GeneCode; }
    }
    public static Gene[] Crossover(Gene g1, Gene g2)  //forcing crossover
```

```
{
   Gene[] child = new Gene[2];
   child[0] = new Gene();
   child[1] = new Gene();


   int pos = (int)(Rnd.NextDouble() * (m_Length - 1));

   for (int i = 0; i < m_Length; i++)
   {
      if (i < pos + 1)
      {
         child[0].m_GeneCode[i] = g1.m_GeneCode[i];
         child[1].m_GeneCode[i] = g2.m_GeneCode[i];
      }
      else
      {
         child[1].m_GeneCode[i] = g1.m_GeneCode[i];
         child[0].m_GeneCode[i] = g2.m_GeneCode[i];
      }
   }
   return child;
}

public void Mutate(double rate)
{
   for (int i = 0; i < m_Length; i++)
   {
      if (Rnd.NextDouble() < rate)
         m_GeneCode[i] = (int)(Rnd.NextDouble() * 3);
   }
}

public override string ToString()
{
   string result = "";
   for (int i = 0; i < m_Length; i++)
      result += m_GeneCode[i].ToString() + " ";
   return result;
}

public int CompareTo(Gene g
{
   if (this.m_Fitness < g.m_Fitness)
      return 1;
   else if (this.m_Fitness == g.m_Fitness)
      return 0;
   else
      return -1;
}

public double Fitness
{
   get { return m_Fitness; }
   set { m_Fitness = value; }
```

```
    }

    public static bool operator ==(Gene g1, Gene g2)
    {
      for (int i = 0; i < m_Length; i++)
      {
        if (g1.m_GeneCode[i] != g2.m_GeneCode[i])
          return false;
      }
      return true;
    }

    public static bool operator !=(Gene g1, Gene g2)
    {
      return !(g1 == g2);
    }

    public override int GetHashCode()
    {
      double hash = 0;
      for (int i = 0; i < m_Length; i++)
      {
        hash += m_GeneCode[i];
      }
      return (int)hash;
    }

    public override bool Equals(object o)
    {
      if (!(o is Gene))
      {
        return false;
      }
      return (this == (Gene)o) ? true : false;
    }

  }
}
```

Sequence Player

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Rhino;
using Rhino.Geometry;

namespace Disassembler
{
  class PlayerConduit : Rhino.Display.DisplayConduit
  {
    //private List<Brep> breps;
    private Transform[] transforms;
    private static List<Rhino.DocObjects.RhinoObject> objs;
    private static List<Rhino.DocObjects.RhinoObject> annotes;
    //private static List<int> moving_object;

    public PlayerConduit(){ }

    public PlayerConduit(Transform[] transforms) { this.transforms = transforms; }

    public static void set_objs(List<Rhino.DocObjects.RhinoObject> obj_list, List<Rhino.DocObjects.RhinoObject>
annote_list) //don't use constructor coz conduit is disposed from time to time
    {
      objs = obj_list;
      annotes = annote_list;
    }

    protected override void PreDrawObjects(Rhino.Display.DrawEventArgs e)
    {
      for (int i = 0; i < objs.Count; i++)
      {
        if (objs[i]!=null)
          e.Display.DrawObject(objs[i], transforms[i]);
      }
    }
    protected override void DrawOverlay(Rhino.Display.DrawEventArgs e)
    {
      for (int i = 0; i < objs.Count; i++)
      {
        if (annotes[i] != null)
          e.Display.DrawObject(annotes[i], transforms[i]);
      }
    }
  }

  class Player
  {
    public double rotation_speed { get; set; }
    public double move_speed { get; set; }
    Member member;
    private List<Transform[]> frame=new List<Transform[]>();
    private List<string> guid_str = new List<string>();
```

```
private List<Rhino.DocObjects.RhinoObject> objs = new List<Rhino.DocObjects.RhinoObject>();
private List<Rhino.DocObjects.RhinoObject> annotes = new List<Rhino.DocObjects.RhinoObject>();
private List<int> moving = new List<int>();
PlayerConduit current_conduit;
PlayerConduit previous_conduit;
private bool hide_initial;

public Player(double rotation_speed,double move_speed)
{
  this.rotation_speed = rotation_speed;
  this.move_speed = move_speed;
}

public int get_frame_count()
{
  return frame.Count;
}

public void bake(int frame_count)
{
  for (int i = 0; i < objs.Count; i++)
  {
    Rhino.DocObjects.BrepObject obj = (Rhino.DocObjects.BrepObject)objs[i];
    Brep b=obj.DuplicateBrepGeometry();
    b.Transform(frame[frame_count][i]);
    RhinoDoc.ActiveDoc.Objects.AddBrep(b);
  }
  for (int i = 0; i < annotes.Count; i++)
  {
    Rhino.DocObjects.ClippingPlaneObject obj = (Rhino.DocObjects.ClippingPlaneObject)annotes[i];
    GeometryBase g = obj.DuplicateGeometry();
    g.Transform(frame[frame_count][i]);
    RhinoDoc.ActiveDoc.Objects.AddTextDot((TextDot)g);
  }
}

public void draw_outline(int frame_count, System.Drawing.Color color)
{
  int index = (int)Math.Round(29.0 * (moving.Count - frame_count) / moving.Count);

  int i=moving[frame_count];
  Rhino.DocObjects.BrepObject obj = (Rhino.DocObjects.BrepObject)objs[i];
  Brep b = obj.DuplicateBrepGeometry();
  b.Transform(frame[frame_count][i]);

  foreach (BrepEdge e in b.Edges)
  {
    Guid id = RhinoDoc.ActiveDoc.Objects.AddLine(e.PointAtStart, e.PointAtEnd);
    Rhino.DocObjects.RhinoObject temp = RhinoDoc.ActiveDoc.Objects.Find(id);
    temp.Attributes.LayerIndex = index;
    temp.CommitChanges();
  }
}

public void read_file(string file,RhinoDoc doc) //for display
```

```
{
  guid_str = new List<string>();
  string[] lines = System.IO.File.ReadAllLines(file);
  int m_index = 0;

  for (int i = lines.Length - 1; i >= 0; i--)
  {
    string[] split = lines[i].Split();
    if (split[0] == "obj") //***need to add obj and annote in file **************************
    {
      objs.Add(doc.Objects.Find(new Guid(split[1])));
      guid_str.Add(split[1]);
    }
    else if (split[0] == "annote")
      annotes.Add(doc.Objects.Find(new Guid(split[1])));
  }
  move_speed += (objs.Count / 50.0);
  Transform[] t_arr = new Transform[objs.Count];
  for(int i=0;i<t_arr.Length;i++)
    t_arr[i]=Transform.Identity;
  frame.Add(t_arr);
  advance_frame();

  moving.Add(0);
  Transform t=Transform.Identity;
  bool slow=true;//determine which segment of move
  for (int i = lines.Length - 1; i >= 0; i--)
  {
    string[] split = lines[i].Split();
    if (split[0] == "r")
    {
      double angle = double.Parse(split[1]);
      Vector3d axis = new Vector3d(double.Parse(split[2]), double.Parse(split[3]), double.Parse(split[4]));
      animate_rotate(member, -angle, axis, rotation_speed, t);
      t = Transform.Rotation(-angle, axis, new Point3d(member.center)) * t;
      slow = true;
    }
    else if (split[0] == "m")
    {
      Vector3d v = new Vector3d(double.Parse(split[1]), double.Parse(split[2]), double.Parse(split[3]));
      if (!slow)
        animate_move(member, -v, move_speed, t);
      else
        animate_move(member, -v, move_speed/2, t);
      t=Transform.Translation(-v) * t;
      slow = false;
    }
    else if (split[0] == "obj")
    {
      Guid new_id = new Guid();
      try { new_id = new Guid(split[1]); }
      catch { throw new Exception("guid not exists in file"); }
      Vector3d v = new Vector3d(double.Parse(split[2]), double.Parse(split[3]), double.Parse(split[4]));
      m_index = guid_str.IndexOf(split[1]);
      member = new Member(new_id, v, guid_str.IndexOf(split[1]));
```

```
            t = Transform.Identity;
            slow = false;
        }
    }
    frame.RemoveAt(frame.Count-1); //remove the last empty frame
    PlayerConduit.set_objs(objs, annotes);
}

public void reset_display()
{
    if (previous_conduit!=null) previous_conduit.Enabled = false;
    previous_conduit=null;
    current_conduit=null;
    if (hide_initial)
    {
        hide_initial = false;
        foreach (Rhino.DocObjects.RhinoObject r in objs)
            RhinoDoc.ActiveDoc.Objects.Show(r, true);
        foreach (Rhino.DocObjects.RhinoObject r in annotes)
            RhinoDoc.ActiveDoc.Objects.Show(r, true);
    }
    RhinoDoc.ActiveDoc.Views.Redraw();
}
public int play_frame(int index)
{
    index = (index > frame.Count - 1) ? frame.Count - 1 : index;
    if (previous_conduit!=null) previous_conduit.Enabled = false;
    current_conduit = new PlayerConduit(frame[index]);
    current_conduit.Enabled = true;
    previous_conduit = current_conduit;
    if (index != 0 && !hide_initial)
    {
        hide_initial = true;
        foreach (Rhino.DocObjects.RhinoObject r in objs)
            RhinoDoc.ActiveDoc.Objects.Hide(r, true);
        foreach (Rhino.DocObjects.RhinoObject r in annotes)
            RhinoDoc.ActiveDoc.Objects.Hide(r, true);
    }
    RhinoDoc.ActiveDoc.Views.Redraw();
    return index;
}
public void advance_frame() //create new blank movement for new frame
{
    if (frame.Count > 1)
    {
        for (int i = 0; i < frame[0].Length; i++)
        {
            if (frame[frame.Count - 1][i] == Transform.Unset)
                frame[frame.Count - 1][i] = frame[frame.Count - 2][i];
        }
    }
    Transform[] t_arr = new Transform[frame[0].Length];
    for(int i=0;i<t_arr.Length;i++)
        t_arr[i]=Transform.Unset;
    frame.Add(t_arr);
```

```
    }
    public void animate_move(Member m, Vector3d vec, double speed, Transform t)
    {
      int count = (int)(vec.Length / speed);
      for (int i = 0; i <= count; i++)
      {
        t = Transform.Translation(vec / (count + 1)) * t;
        m.center += vec / (count + 1);
        frame[frame.Count - 1][m.index] = Transform.Identity * t;  //current frame item transform
        moving.Add(m.index);
        advance_frame();
      }
    }
    public void animate_rotate(Member m, double angle, Vector3d axis, double speed,Transform t)
    {
      int count = (int)Math.Abs(angle / speed);
      for (int i = 0; i <= count; i++)
      {
        t = Transform.Rotation(angle / (count + 1), axis, new Point3d(m.center)) * t;
        frame[frame.Count - 1][m.index] = t;
        moving.Add(m.index);
        advance_frame();
      }
    }
    public void play_file(string file)
    {
      string[] lines = System.IO.File.ReadAllLines(file);

      for (int i = lines.Length - 1; i >= 0; i--)
      {
        string[] split = lines[i].Split();
        if (split[0] == "r")
        {
          double angle = double.Parse(split[1]);
          Vector3d v = new Vector3d(double.Parse(split[2]), double.Parse(split[3]), double.Parse(split[4]));
          member.animate_rotate(-angle, v, rotation_speed);
        }
        else if (split[0] == "m")
        {
          Vector3d v = new Vector3d(double.Parse(split[1]), double.Parse(split[2]), double.Parse(split[3]));
          member.animate_move( -v, move_speed);
        }
        else
        {
          Guid new_id = new Guid();
          try { new_id = new Guid(split[0]); }
          catch { throw new Exception("guid not exists in file"); }
          Vector3d v = new Vector3d(double.Parse(split[1]), double.Parse(split[2]), double.Parse(split[3]));
          member = new Member(new_id, v, 0);
        }
      }
    }
  }
}
```