Applications of objected-oriented programming design in robot kinematics

Chien-Jung Chiu
Lehigh University

Follow this and additional works at: http://preserve.lehigh.edu/etd

Recommended Citation
AUTHOR:
Chiu, Chien-Jung

TITLE: Applications of Object-Oriented Programming Design in Robot Kinematics

DATE: January 1992
APPLICATIONS OF OBJECTED-ORIENTED PROGRAMMING
DESIGN IN ROBOT KINEMATICS

BY

CHIEN-JUNG CHIU

A THESIS
PRESENTED TO THE
INDUSTRIAL ENGINEERING DEPARTMENT
OF
LEHIGH UNIVERSITY
IN CANDIDACY FOR THE DEGREE OF
MASTER OF SCIENCE
IN
INDUSTRIAL ENGINEERING
This thesis is accepted and approved in partial fulfillment of requirements for the degree of Master of Science.

[Signature]

[Name]

[Signature]

[Name]

[Signature]

[Name]
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>SECTION</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSTRACT</td>
<td>1</td>
</tr>
<tr>
<td>1.0 INTRODUCTION</td>
<td>2</td>
</tr>
<tr>
<td>1.1 ASPECTS OF OOP</td>
<td>2</td>
</tr>
<tr>
<td>1.2 C++ AND OOP</td>
<td>4</td>
</tr>
<tr>
<td>1.3 CASE STUDY</td>
<td>4</td>
</tr>
<tr>
<td>2.0 ROBOT ARM KINEMATICS</td>
<td>5</td>
</tr>
<tr>
<td>2.1 HOMOGENEOUS MATRIX TRANSFORMATION</td>
<td>5</td>
</tr>
<tr>
<td>2.2 COMPOSITE HOMOGENEOUS TRANSFORMATION MATRIX</td>
<td>7</td>
</tr>
<tr>
<td>2.3 DENAVIT-HERTENBERG REPRESENTATION</td>
<td>8</td>
</tr>
<tr>
<td>3.0 SETTING OOP MODEL</td>
<td>10</td>
</tr>
<tr>
<td>3.1 CLASS</td>
<td>10</td>
</tr>
<tr>
<td>3.2 PUBLIC, PRIVATE, PROTECTED</td>
<td>12</td>
</tr>
<tr>
<td>3.3 CONSTRUCTORS AND DISTRACTERS</td>
<td>13</td>
</tr>
<tr>
<td>3.3.1 COPY CONSTRUCTOR</td>
<td>14</td>
</tr>
<tr>
<td>3.4 CLASS METHODS</td>
<td>16</td>
</tr>
<tr>
<td>3.4.1 OPERATOR OVERLOADING</td>
<td>16</td>
</tr>
<tr>
<td>3.5 DERIVED CLASS</td>
<td>19</td>
</tr>
<tr>
<td>3.5.1 CONSTRUCTORS OF THE DERIVED CLASS</td>
<td>20</td>
</tr>
<tr>
<td>3.5.2 METHODS OF THE DERIVED CLASS</td>
<td>21</td>
</tr>
<tr>
<td>3.6 MORE DERIVED CLASS</td>
<td>25</td>
</tr>
<tr>
<td>3.7 ROBOT CLASS</td>
<td>27</td>
</tr>
<tr>
<td>3.7.1 MEMBER CLASSES</td>
<td>27</td>
</tr>
<tr>
<td>3.7.2 MULTIPLE INHERITANCE</td>
<td>30</td>
</tr>
<tr>
<td>3.7.3 METHODS FOR ROBOT CLASS</td>
<td>32</td>
</tr>
<tr>
<td>4.0 PROGRAM IMPLEMENTATION</td>
<td>36</td>
</tr>
<tr>
<td>4.1 PROGRAM STRUCTURE</td>
<td>36</td>
</tr>
<tr>
<td>4.1.1 STEP 1:</td>
<td>37</td>
</tr>
<tr>
<td>4.1.2 STEP 2:</td>
<td>39</td>
</tr>
<tr>
<td>4.1.3 STEP 3,4</td>
<td>39</td>
</tr>
<tr>
<td>4.2 OUTPUT ANALYSIS</td>
<td>40</td>
</tr>
<tr>
<td>5.0 FURTHER DISCUSSION AND SUMMARY</td>
<td>44</td>
</tr>
<tr>
<td>5.1 ADVANTAGES OF OOP</td>
<td>44</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>5.2 DISADVANTAGES OF OOP</td>
<td>48</td>
</tr>
<tr>
<td>5.3 SUMMARY</td>
<td>55</td>
</tr>
<tr>
<td>REFERENCES</td>
<td>56</td>
</tr>
<tr>
<td>APPENDIX A: PROJECT PROGRAM</td>
<td>57</td>
</tr>
<tr>
<td>APPENDIX B: OUTPUT DATA</td>
<td>76</td>
</tr>
<tr>
<td>VITA</td>
<td>81</td>
</tr>
</tbody>
</table>
The purpose of this project is to apply the Object-Oriented Programming design methods to implement the robot arm motion.

This project is divided into five chapters, organized as follow:

**Chapter 1** Introduces some OOP aspects, such as data abstraction, inheritance. Also the features of C++ language applied to OOP design will be mentioned.

**Chapter 2** provides the basic tools for calculating the robot coordinate system, such as homogeneous matrix transformation, D-H representation. These formulas will be used for the program later.

**Chapter 3** sets up the OOP model needed in this project, including defining the base class array and the derived classes such as joint, robot. All the methods will be explained in detail in this chapter.

**Chapter 4** shows the program implementation and the output analysis. The features of the OOP will be clearly demonstrated during the execution.

**Chapter 5** summarizes the advantages and disadvantages of the OOP design, some improvements will also be suggested in this chapter.
1.0 INTRODUCTION

Object-Oriented Programming (OOP) has become the most important design concept for the software engineering. With the increasing size and complexity of the program, maintenance and redesign are always costly. The best solution is to integrate the codes with the hardware. However, it is not feasible in reality. OOP differs from the conventional programming mainly in the design construction. Instead of developing the sophisticated algorithm, OOP tends to create simple, visible independent components exactly like the hardware, and links the objects together with a much smoother mechanism. In the follow sections, we will discuss all these issues in detail.

1.1 ASPECTS OF OOP

Before we start talking about the features of OOP, let's first see what the OOP really means. For example, suppose someone wanted to paint a house but had no idea how to proceed, he might decide to check a book on painting (procedures) and read the chapter on houses (objects): this is the conventional procedural approach. Alternatively, he might check a book on houses and read the chapter on painting: this is the object-oriented approach.

OOP treats data structures rather than procedures as primary. In a conventional
program, a program is written for every action, and then the procedure decides how to handle arguments of various types. IN OOP, we define each type of data item, and its definition includes procedures for handling it.

There are four distinct features for the OOP:

* Data abstraction
* Encapsulation and data hiding
* Polymorphism
* Inheritance

Data abstraction is the crucial step of representing information in terms of its interface with the users. Since its internal structure is hidden, the same abstract data type can therefore have different implementations at different times without affecting the code that uses it. This is one of the most important design considerations - reusability.

Encapsulation and data hiding together provide a special feature of OOP other than the conventional data record. First, the data and so called member functions(methods) are built in the same slot, this is known as encapsulation. Second, the data can only be accessed by these member functions unless otherwise specified, also known as data hiding. In this way, the program is well protected from careless overriding. Error detection is much easier since the privacy belongs to its member function only.

Polymorphism allows the user to create multiple definitions for operators and functions, with the program context determining which definition is used. For example, one may try to print out a program under different situations, but just wants to use a single function name (print). Polymorphism allows this by using different arguments with the same prototype.
Inheritance is the main tool for OOP to apply the reusable codes. When one develops a new project, particularly if the project is large, it's preferable to reuse proven codes rather than to reinvent them. Employing old codes not only saves the time taken to redesign, more importantly, it helps suppress the introduction of bugs into a program. The concept is splendid, however, proper implementation in all situations requires some challenging technical adjustments.

1.2 C++ AND OOP

In the later discussion, we will use C++ to present all the OOP issues. C++ has become an increasing popular programming language. Its appeal is due not only to the popularity of its language C, but also to its data abstraction and object-oriented features. For example, AT&T is rewriting UNIX in C++ because C++ improves the reliability, maintainability and reusability of the code. Apple is also using C++ to develop system software for its Macintosh line for the same reasons and because OOP techniques are a natural match to program features such as window and dialog boxes. Generally speaking, C++ is a very good language to implement all the OOP features, which we will see later.

1.3 CASE STUDY

The most recent applications of OOP focus on the graphic system, since these objects are visible and obvious. There are already many CAD packages starting to use the OOP design methods. In our case study, we have selected the robot arm motion to show how to implement the OOP concepts. Since the robot itself is an assembly of all different parts such as base, joints, links, etc., it's also a dynamic object. Therefore, it's clear that the object and method together fulfill the entire task.
2.0 ROBOT ARM KINEMATICS

Robot arm kinematics deals with the analytical study of the geometry of motion of a robot arm with respect to a fixed reference coordinate system as a function of time. Thus, kinematics analytically describes the spatial displacement and the position and orientation of the end-effector of a robot arm. For a robot with six joints, one may wonder what the position and the orientation are for each after a series of rotations. This question is usually referred to as the direct (or forward) kinematics problem.

2.1 HOMOGENEOUS MATRIX TRANSFORMATION

Figure 2.1 Coordinate transformation

Figure 2.1 shows the coordinate transformation. A base coordinate system \((U,V,W)\) with origin \(O\) moves to the new origin \(O'\) with a series of rotations forming a new coordinate system \((U',V',W')\). Then the relation between the two can be expressed as:

\[
[U,V,W] = T [U'V'W'],
\]

\(T\) is called a transformation matrix.
If T is set as a 4 by 4 matrix, then T is a homogeneous transformation matrix. Now the relation between the new and old coordinate system must be found in order to obtain the T matrix. There are four possible conditions for the coordinate movement

1. Translation only (x,y,z directions): $T_{\text{tran}}$

2. Rotation about the X axis (with angle $\alpha$): $T_{x,\alpha}$

3. Rotation about the Y axis (with angle $\phi$): $T_{y,\phi}$

4. Rotation about the Z axis (with angle $\theta$): $T_{z,\theta}$

Then we can get:

$$T_{\text{tran}}[1, 1] = 1, \ T_{\text{tran}}[1, 2] = 0, \ T_{\text{tran}}[1, 3] = 0, \ T_{\text{tran}}[1, 4] = dx,$$

$$T_{\text{tran}}[2, 1] = 0, \ T_{\text{tran}}[2, 2] = 1, \ T_{\text{tran}}[2, 3] = 0, \ T_{\text{tran}}[2, 4] = dy, \quad \ldots \ldots \ldots \ (2.1)$$

$$T_{\text{tran}}[3, 1] = 0, \ T_{\text{tran}}[3, 2] = 0, \ T_{\text{tran}}[3, 3] = 1, \ T_{\text{tran}}[3, 4] = dz,$$

$$T_{\text{tran}}[4, 1] = 0, \ T_{\text{tran}}[4, 2] = 0., \ T_{\text{tran}}[4, 3] = 0, \ T_{\text{tran}}[4, 4] = 1,$$

$$T_{x,\alpha}[1, 1] = 1, \ T_{x,\alpha}[1, 2] = 0, \ T_{x,\alpha}[1, 3] = 0, \ T_{x,\alpha}[1, 4] = 0,$$

$$T_{x,\alpha}[2, 1] = 0, \ T_{x,\alpha}[2, 2] = \cos \alpha, \ T_{x,\alpha}[2, 3] = -\sin \alpha, \ T_{x,\alpha}[2, 4] = 0, \quad \ldots \ldots \ldots \ (2.2)$$

$$T_{x,\alpha}[3, 1] = 0, \ T_{x,\alpha}[3, 2] = \sin \alpha, \ T_{x,\alpha}[3, 3] = \cos \alpha, \ T_{x,\alpha}[3, 4] = 0,$$

$$T_{x,\alpha}[4, 1] = 0, \ T_{x,\alpha}[4, 2] = 0, \ T_{x,\alpha}[4, 3] = 0, \ T_{x,\alpha}[4, 4] = 1,$$

$$T_{y,\phi}[1, 1] = \cos \phi, \ T_{y,\phi}[1, 2] = 0, \ T_{y,\phi}[1, 3] = \sin \phi, \ T_{y,\phi}[1, 4] = 0,$$

$$T_{y,\phi}[2, 1] = 0, \ T_{y,\phi}[2, 2] = 1, \ T_{y,\phi}[2, 3] = 0, \ T_{y,\phi}[2, 4] = 0, \quad \ldots \ldots \ldots \ (2.3)$$

$$T_{y,\phi}[3, 1] = -\sin \phi, \ T_{y,\phi}[3, 2] = 0, \ T_{y,\phi}[3, 3] = \cos \phi, \ T_{y,\phi}[3, 4] = 0,$$

$$T_{y,\phi}[4, 1] = 0, \ T_{y,\phi}[4, 2] = 0, \ T_{y,\phi}[4, 3] = 0, \ T_{y,\phi}[4, 4] = 1,$$
2.2 COMPOSITE HOMOGENEOUS TRANSFORMATION MATRIX

A mechanical manipulator consists of a sequence revolute or prismatic joints. Each link-joint pair constitutes one degree of freedom. Hence, for a joint link pairs, it begins with link 0 attached to a supporting base, where an initial coordinate frame is usually established for this dynamic system. The joints and links are numbered outwardly from the base; thus joint 1 is the point of connector link 1 and the supporting base.

To establish the relationship of the joints, we first set up the coordinate transformation between two conjunctive joints. For example, it always refers the current joint to its previous joint. Suppose $^0A_1$ represents the joint 1 relative to the joint 0, $^1A_2$ represents the joint 2 relative to joint 1, etc... Then

$^0T_i = ^0A_1^1A_2^2A_3.......^i-1A_i$ ........................................(2.5)

where $^0T_i$: joint i referred to the base coordinate
2.4 DENAVIT-HERTENBERG REPRESENTATION

The remaining question becomes how to decide $i^{-1}A_i$, i.e., what is the general formula to describe the joint position referred to as the previous one?

Figure 2.2 shows the joint and link. From joint $i$ to joint $i+1$, there are four successive transformations:

1. Rotate about the $Z_{i-1}$ axis of $\theta_i$ align the $X_{i-1}$ axis with the $X_i$ axis.

2. Translate along the $Z_{i-1}$ axis a distance of $d_i$ to bring the $X_{i-1}$ axis and $Z_{i-1}$ axis into coincidence.

3. Translate along the $X_i$ axis a distance of $a_i$ to bring the two origins as well as the $X$ axis into coincidence.

4. Rotate about the $X_i$ axis of $\alpha_i$ to bring the two coordinate systems into coincidence.

Figure 2.2 Link coordinate system and its parameters
i^{-1}A_i = T_{x,d} \ T_{z,\theta} \ T_{x,a} \ T_{x,\alpha} \ \ \ \ \ (2.6)

i^{-1}A_i[1,1] = \cos \theta_i; \quad i^{-1}A_i[1,2] = -\cos \alpha_i \sin \theta_i; \quad i^{-1}A_i[1,3] = \sin \alpha_i \sin \theta_i;

i^{-1}A_i[1,4] = a_i \cos \theta_i; \quad i^{-1}A_i[2,1] = \sin \theta_i; \quad i^{-1}A_i[2,2] = \cos \alpha_i \cos \theta_i;

i^{-1}A_i[2,3] = -\sin \alpha_i \cos \theta_i; \quad i^{-1}A_i[2,4] = a_i \sin \theta_i; \quad i^{-1}A_i[3,1] = 0;

i^{-1}A_i[3,2] = \sin \alpha_i; \quad i^{-1}A_i[3,3] = \cos \alpha_i; \quad i^{-1}A_i[3,4] = d_i; \quad i^{-1}A_i[4,1] = 0;

i^{-1}A_i[4,2] = 0; \quad i^{-1}A_i[4,3] = 0; \quad i^{-1}A_i[4,4] = 1;

Figure 2.3 shows the coordinate system of PUMA robot. We will use this for our OOP program test.

Figure 2.3 Establishing link coordinate systems for a PUMA robot

<table>
<thead>
<tr>
<th>Joint</th>
<th>( \theta_1 )</th>
<th>( \alpha_1 )</th>
<th>( a_1 )</th>
<th>( d_1 )</th>
<th>Joint range</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>90</td>
<td>-90</td>
<td>0</td>
<td>0</td>
<td>-160 to +160</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>431.8 mm</td>
<td>149.09 mm</td>
<td>-225 to 45</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>90</td>
<td>-20.32 mm</td>
<td>0</td>
<td>-45 to 225</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>-90</td>
<td>433.07 mm</td>
<td>0</td>
<td>-110 to 170</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-100 to 100</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>56.25 mm</td>
<td>-266 to 266</td>
</tr>
</tbody>
</table>
3.0 SETTING OOP MODEL

From the previous discussions, the joint can be represented by a homogeneous matrix. A robot consists of several joints linked together. In the OOP's point of view, robot and joint are the real objects. However, there may be some object which is not visible. For example, an array which represents the general property may be treated as an object also. Definition of the class pertaining to requirements for the object will be addressed first.

3.1 CLASS

In the following discussion, C++ will be used to explain all the features of OOP. The class in C++ is a vehicle for translating a data abstraction to a user-defined type. Class consists of both data representation and methods, which manipulate the data internally. Generally, a class specification has two parts:

* A class definition, which describes the component member (both data member and function member) of the class

* The class method definitions, which describe how certain class member functions are implemented

Figure 3.1 lists all the classes we use in this project, which include the previously mentioned parts. However, they are different from the record or structure.
Figure 3.1: class prototype of the robot system
3.2 PUBLIC, PRIVATE, PROTECTED

As we can see from the classes, data and member functions begin with a key word: public, private or protected. The selection of the type is important. Since OOP has the data hiding, we would like to see the data declared as private, and the member functions are treated publicly. But what about those functions which are protected? Protected functions are related to the class inheritance, which will be discussed later. However, the concept applied to the protected functions will be considered at this time.

* **Private** data is the hidden information of a class and can be accessed directly by the object itself. The program may access the private data through the member functions only.

* **Public** functions are open to the main program to access the private data.

In the array class, the array element and size are defined as the **protected** type. The difference between the private and protected type is that the protected data can be inherited by its derived class. For example, the joint class can inherit the protected data without creating a new one. Fig 3.2 shows the inheritance relation between the array and joint class. Since joint is a 4 by 4 matrix, it can be treated as a one dimension array with 16 elements. In this way, definition of a two dimension array is unnecessary, one can just use all the features from the array class. The reusability of the code is one of the most important goals of OOP. Public, private, and protected together control the data hiding and the proper way to gain data accessibility.
3.3 CONSTRUCTORS AND DESTRUCTORS

Constructors and destructors are the two special ways for user to evoke and release the class. In order to make it easy to use, C++ uses the same name as the class, but a scope to separate the class name from the function. For example, in the array class, we use:
array :: array(unsigned int n, double val)
{
    arr = new double[n];
    ..........
}

When we use a constructor to create an object, the program undertakes the responsibility of tracking that object until it expires. At that time, the program automatically calls a special function bearing the formidable title of destructor. The destructor should clean up any debris, so it actually serves a constructive purpose. The destructor has the similar name to the class but adds ~ ahead of the function. For example, the destructor of the array class is:

    array :: ~array()

Constructors and destructors are important if the class has many derived classes. For a single class, the way to evoke or delete a class is simple. However, failure to use the right parameters to evoke the constructors will result in a multitude problems. These cases will arise later. Now another special constructor- copy constructor will be discussed.

3.3.1 COPY CONSTRUCTOR

In the array class, we define a copy constructor like:

    array(array & a);
array:: array(array & a)
{
    size=a.size;
    arr = new double[size];
    for (int i=0; i<size; i++)
        arr[i]=a.arr[i];
}

From the program, we must first pay attention to the dynamic allocation of the keyword `new`. If the class includes the pointer to find the address of the object, then the initialization must be checked very carefully. C++ provides a mechanism for showing how the initialization works by defining a copy constructor. The copy constructor is used in three situations:

1. When initializing one class object to another, the constructor sets values for the initialized objects.

2. When passing a class by value as an argument, the constructor initializes the corresponding formal argument in the called function.

3. When having a function return an object, the constructor initializes a temporary object used to convey the object's values to the calling program.
3.4 CLASS METHODS

Beside the constructors, there are other functions called methods in the array class. One type of the methods is exactly like the function in C language, except it is now used as a member function, which means it provides the method to access the private data. The other type of function is a little different and represents the special feature of the OOP- polymorphism. This function is sometimes known as the operator overloading.

3.4.1 OPERATOR OVERLOADING

Suppose creation of two objects a1,a2, and both are array class:

```c
main ()
{
    array a1,a2;
}
```

How is the operation between the two objects, e.g., a1*a2, to be done? Since array is no longer a simple data type like an integer, any action must be defined in the member function of the class in order to access the data. We may define a function like as:

```c
void array multiple:(.....)
{
    .......
}
```
So if we try to do the multiplication, we can use the member function as:

\[ a1.multiple(a2); \]

But how about using \( a1 \times a2 \)? Once again, C++ provides a standard function called the overloading function to allow the operator (such as *,=,<<) to be redefined by the users for different purposes. For example, we define the \([ ]\) as:

```cpp
double & array::operator[](int i)
{
    return arr[i];
}
```

If this operator is not defined, but access to the array element is desired, one must write:

\[ a1.arr[i]; \]

However, one now simply writes:

\[ a1[i]; \]

If one wants to multiple two array \( a1,a2 \), it is necessary to write all the procedures if the \( \times \) operator is not defined. If one wants to assign an object to the other object (like \( a1=a2 \)) without knowledge of the array size, it's even impossible to do in the main program.

In the array class we define the "=" operator as:

```cpp
array & array::operator=(array & a)
{
    if(this == &a)
        return *this;
    return *this;
}
```
delete arr;
size=a.size;
arr=new double[size];
for(int i=0; i<size; i++)
arr[i]=a.arr[i];
return *this;
}

This function does four things which are not likely to be done in the main program:

1. This function uses the object argument passed by reference (array &)
2. The function returns a reference to a new object, which is a copy of the one passed as an argument.
3. The function uses delete to free memory to which arr points. This clears the old data, preparing arr to point to new data.
4. The function allocates space for the data being copied, assigns the address to arr, then copies the data from the old array to the new.

With the "=" operator overloading, then assigning one object to the other object becomes very simple, just by writing a1=a2. All the derived classes of the array class will use this operator to assign the object.
3.5 DERIVED CLASS

The base class array has been defined in the previous sections. Let's now start to define a derived class from the array class.

A joint class is the derived class from array and used to represent a 4 by 4 homogeneous matrix. Since redefining a two dimension array is difficult, instead we treat the 4 by 4 matrix as a one dimension array with 16 elements. In this case, we can inherit all the array features. The way to define a derived class is shown as:

```cpp
class joint: public array
{
    public:
        joint();

        ............ inheritance declaration

    joint();

    ............

};
```

Here public means the derived class inherits the protected and public parts from the array class, but not the private. That is, a derived class can not access the private data from its ancestor unless through public functions.

In C++, constructors and destructors cannot be inherited. So we need to define them for a new derived class.
3.5.1 CONSTRUCTORS OF THE DERIVED CLASS

Because the same name is used for the constructor and destructor as the class name, this rule must be kept consistently. However, the joint class initializes the value by using the array constructor. In the prototype, constructor is defined as:

```
joint(unsigned int n, double val); similar to the array class
```

In the function definition, a special mechanism is used for the argument passing so that the derived class can actually evoke the value from the ancestor. It is shown in the Fig. 3.3.

```
joint:: joint(unsigned int n,double val) : array(n,val)
{
    .......
}
```

Fig. 3.3 Passing arguments through to a base class constructor

Once a derived class is created, the program first calls the constructor for the base class, then the constructor of the derived class. When the object of a derived class expires, the program first calls the derived class destructor, if any, then the base class destructor.
3.5.2 METHODS OF THE DERIVED CLASS

The derived class inherits all the methods from the base class, even the operators. However, the derived class usually has additional methods of its own. Although the joint class can initialize the values by passing the arguments to the array constructor, as seen above, the services may be insufficient for the new need. For example, using constructor

```
joint:: joint(unsigned int n, double val): array(n, val)
```

We can create an object like:

```
joint j1(16,5);
```

What this does is create an array with 16 elements where each value equals 5. However, recall the equation (2.6), where

\[
i^{-1}A_i = T_{z,d} T_{z,\theta} T_{x,a} T_{x,\alpha}
\]

\[
i^{-1}A_i[1,1] = \cos \theta_i;
\]

\[
i^{-1}A_i[1,2] = -\cos \alpha_i \sin \theta_i;
\]

\[
i^{-1}A_i[1,3] = \sin \alpha_i \sin \theta_i;
\]


Obviously, the initial values of the array class may not afford the values we want. Is it necessary to create another constructor? The answer is no. In C++, the derived constructor has the same arguments as the base class in order to evoke its ancestor. Creation of a constructor with a different signature from the base class will cause some unexpected results. The alternative way is to create a member function to meet the requirements.
We define a member function to initialize the joint elements as:

```cpp
void joint::start(double t, double s, double a, double d)
{
    double rad = 2 * 3.14159 / 360;
    arr[0] = cos(t * rad);
    arr[1] = sin(t * rad);
}
```

In the main program, we can initialize a joint as:

```cpp
main()
{
    joint j1(16, 0);  // initialize an array with size 16 and all values 0
    j1.start(90, -90, 0, 0);  // start the homogeneous matrix
}
```

Next, we have to define another method for the joint operator. Recall equation (2.5):

\[ 0T_i = 0A_1^1A_2^2A_3^3\ldots i^{-1}A_i \] ..................................................\( (2.5) \)

The `start` function just defines the values for the matrix \( i^{-1}A_i \). Knowledge of the joint position relative to the base, that is finding the \( 0T_i \), requires doing the multiplication. For example,
\[ ^0T_1 = ^0A_1 \]
\[ ^0T_2 = ^0A_1 ^1A_2 \]
\[ ^0T_3 = ^0A_1 ^1A_2 ^2A_3 \]

Although the * operator has been defined in the array class and has been inherited by the joint class, the function is not adequate. The array defines the * operator based on the one dimension array, however, the multiplication here is two 4 by 4 matrices. In order to fulfill this requirement, the operator * must be redefined. In the joint class, definition of the 4 by 4 matrix multiplication is accomplished by using one dimension such as:

```cpp
joint joint :: operator*(joint &a)
{
    joint pr(size,0);
    pr[0]=arr[0]*a[0]+arr[4]*a[1]+arr[8]*a[2]+arr[12]*a[3];
    ..............................................................
    return pr;
}
```

The program returns the joint pr to the main program. The copy constructor will be evoked to set a temporary joint variable before the joint is assigned.
In the main program, once we create two joint objects, one can simply write the multiplication as:

```c
int main()
{
    joint j1(16,2);
    joint j2(16,3);
    joint j3 = j1 * j2;
}
```

The other method in the joint class is to show the elements of the array. This is defined as:

```c
void joint :: show(int n);
{
    .............
}
```

If one wants to see the result of $j3 = j1 \times j2$, he can use this function:

```c
j3.show(3);
```

The remaining method in the joint class is the same.
3.6 MORE DERIVED CLASS

The second derived class we define in the program is \texttt{joint1,joint2,...,joint6}. These classes are based on the joint class, except with one more function-rotate. The joint class simply initiates the homogeneous matrix, however, in the robotic system, joint has some degrees of freedom to manipulate the task. In this program, these joints are for the PUMA robot, which has 6 degrees of freedom (6 joints), each being a rotation joint. The reason this function wasn't defined in the joint class is because each joint has a different response when receiving a \texttt{rotate} command. In reality, the values of the homogeneous matrix change differently for each joint. Thus we define the \texttt{joint1,joint2,...,joint6} as:

\begin{verbatim}
class joint1:public joint

class joint2:public joint

............................................

And the \texttt{rotate} function looks like:

\begin{verbatim}
void joint1:: rotate(double t)
{

double rad=2*3.14159/360;
arr[0]=cos(t*rad);
arr[1]=sin(t*rad);
arr[8]=-sin(t*rad);
arr[9]=cos(t*rad);
}
\end{verbatim}
void joint2 ::rotate(double t) 
{
    double rad=2*3.14159/360;
    arr[0]=cos(t*rad);
    arr[1]=sin(t*rad);
    arr[4]=-sin(t*rad);
    arr[5]=cos(t*rad);
    arr[12]=431.8*cos(t*rad);
    arr[13]=431.8*sin(t*rad);
}

void joint3 :public joint
{
    ....................

The constructor inherits from the joint class and is defined as:

    joint1 :: joint1(double *pn, unsigned int n)
    :
    joint(*pn,n)
3.7 ROBOT CLASS

So far 3 class types: array, joint, joint1 (joint2, ...) have been explained. Since the main goal is to create a robot which can perform the rotation motion, the next step is to define a new class robot to incorporate the 6 joints.

There are two ways to define a class to access multiple classes:

1. Member classes

2. Multiple inheritance

Which one is better? It depends on the circumstances. In this program, multiple inheritance is used. Next, the member classes will be discussed and compared to the multiple inheritance.

3.7.1 MEMBER CLASSES

The type of a variable in a class can be the other class type. For example, we define the class joint1, joint2, ......., in the robot class as:

Class robot
{
    Private:
        joint1 j1;
        joint2 j2;
        joint3 j3;
        ............
}

In this case, we need to initialize j1, j2, j3, ... j6 while the construction of the robot
class is evoked. The way to define the constructor of the robot is similar to the previous one as:

```cpp
robot :: robot(unsigned int n, double val) :
    j1(n, val), j2(n, val), j3(n, val), j4(n, val), j5(n, val), j6(n, val)
```

And to start the initial robot position, we define the function as:

```cpp
void robot :: start(double t1, double s1, double a1, double d1,
    double t2, double s2, double a2, double d2,
    double t3, double s3, double a3, double d3,
    double t4, double s4, double a4, double d4,
    double t5, double s5, double a5, double d5,
    double t6, double s6, double a6, double d6)
{
    j1.start(t1, s1, a1, d1);
    j2.start(t2, s2, a2, d2);
    j3.start(t3, s3, a3, d3);
    j4.start(t4, s4, a4, d4);
    j5.start(t5, s5, a5, d5);
    j6.start(t6, s6, a6, d6);
}
```

Although the member classes satisfy the initializing condition, however, there are two disadvantages in using the member classes:
1. One can't use the methods of the member class directly. Suppose one creates a robot puma, and wants to access the rotate function of the joint1 j1 like:

\[
\text{cout}\ll puma.j1.rotate(25);
\]

Unfortunately, this is not permissible. The reason is that \text{puma.j1} is the \text{protected} member of the robot class, hence one can't use it publicly. That's what the OOP restricts the private or protected data to be accessed. Instead, one must define a member function of robot to execute the rotation of joint1 like:

\[
\text{void robot :: rotate1(double t)}
\]

\[
\{ \text{return j1.rotate(t);} \}
\]

\[
\text{main()}
\]

\[
\{ \text{puma.rotate1(25);} // now it is acceptable} \}
\]

2. Using the member classes will lose the inheritance features. For example, one can't write a program like:

\[
\text{robot puma(16,5);} \quad // \text{create a robot}
\]

\[
\text{cout}\ll puma[1]; \quad // \text{show the first element}
\]

The operator [ ] is defined in the array class, which is the ancestor of j1, j2, j3, ..., j6, but not accessible to the robot class. If one wants to inherit some functions from the ancestor of the member classes, one should use multiple inheritance.
3.7.2 MULTIPLE INHERITANCE

Multiple inheritance means deriving a class from more than one base class. With multiple inheritance, the new class inherits all the data members and all the class methods of all the base classes. Therefore, an object of the new class can use the base class methods directly, unlike the case of a class with object members.

In our example, let's define the robot class with multiple inheritance:

```cpp
Class robot: public joint1, public joint2, public joint3,
           public joint4, public joint5, public joint6
{
       .......... 
}
```

The definition with the keyword `public` enables the robot class to inherit all the data and functions from the `joint1, joint2, joint3, .... joint6`.

To initialize a robot class, it's similar to the process for member classes, except there is no member object like `j1, j2..... j6`. One writes it as:

```cpp
robot:: robot(unsigned int n, double val) :
       joint1(n, val), joint2(n, val), joint3(n, val),
       joint4(n, val), joint5(n, val), joint6(n, val)
```

Once the robot is created, all the ancestor classes are evoked at the same time.
To start a robot, there is a slight difference. The scope :: is used to indicate the function one wants to access from the multiple inheritance classes.

```c
void robot :: start(double t1, double s1, double a1, double d1,
                     double t2, double s2, double a2, double d2,
                     double t3, double s3, double a3, double d3,
                     double t4, double s4, double a4, double d4,
                     double t5, double s5, double a5, double d5,
                     double t6, double s6, double a6, double d6)
{
    joint1 :: start(t1,s1,a1,d1);
    joint2 :: start(t2,s2,a2,d2);
    joint3 :: start(t3,s3,a3,d3);
    joint4 :: start(t4,s4,a4,d4);
    joint5 :: start(t5,s5,a5,d5);
    joint6 :: start(t6,s6,a6,d6);
}
```

The difference between the multiple inheritance and the member classes is that one can access the function directly from the classes inherited. For example, if one wants to rotate joint1 an angle for the robot puma, he can use:

```c
puma.joint1 :: rotate(45); // use scope to specify the function
// inherited
```
Unlike the member classes, it is not required to define a member function to execute the rotate command. Since the member functions from the joint class are needed, we select the multiple inheritance to define the robot class in this project.

### 3.7.3 METHODS FOR THE ROBOT CLASS

Beside the start function and the functions inherited, three other functions are defined as:

```c
void rotate(int,double);
void show(int n);
void showall( );
```

Although these three methods are new member functions, actually they apply the previous methods defined in the ancestor classes. For example, the rotate function:

```c
void robot :: rotate(int n,double t)
{
    switch(n)
        case 1 : joint1 :: rotate(t);
        break;
        case 2 : joint2 :: rotate(t);
        break;
        case : joint3 :: rotate(t);
```

..............................

```c
}
For the method `show`, it shows the joint positions relative to the base. Recall equation (2.5), seeking the value of

\[ ^0T_i = ^0A_1 A_2^2 A_3 \ldots \ldots i^{th} A_i \quad i=1,2,3,\ldots,6 \]

Thus,

\[ ^0T_1 = ^0A_1 \]
\[ ^0T_2 = ^0A_1 A_2 \]
\[ ^0T_3 = ^0A_1 A_2^2 A_3 \]

Here is how one defines the function `show`:

```cpp
void robot :: show(int n)
{
    switch(n)
    {
    case 1 :
        joint a1(16,0);
        for(int i=0; i<16; i++)
            a1[i]=joint1 :: value(i);
        a1.show(1);
        break;
    case 2 :
        joint b1(16,0);
        joint b2(16,0)
        for(int j=0; j<16; j++)
```
{ 
    b1[j]=joint1 :: value(i);
    b2[j]=joint2 :: value(i);
}

joint b=b1*b2;

b.show(2);
break;

...........................................

} 

In case 1, we want to show $\mathbf{T}_1$, which equals $\mathbf{A}_1$, it simply shows the joint1 values. Since joint1 is not the member object, a joint a1 is created and assigned the values of joint1 and then is shown as:

    a1[i]= joint :: value(i);
    a1.show(1);

In case 2, it's necessary to find the $\mathbf{T}_2$, which equals $\mathbf{A}_1 \cdot \mathbf{A}_2$, so two joints b1,b2 are created; and the values of joint1,joint2 are assigned to b1,b2, another joint b is created, then calculate:

    joint b=b1*b2;

We have defined the operator * in the joint class, now one can see how easily the joint multiplication is handled. Similarly, case 3, case4,........ are using the operator * to do more multiplications.
The `showall` function shows all the 6 joints, it's very easily to be defined as:

```cpp
 void robot :: showall( )
{
    show(1);
    show(2);
    show(3);
    show(4);
    show(5);
    show(6);
}
```

In C++, the member function can access the other member functions in the same class, just as shown above.

This completes the definition of all the class data and member functions for the project. In the next sections, their executions will be shown.
4.0 PROGRAM IMPLEMENTATION

The purpose of OOP is to build both data and methods in the class, and all the executions are completed by sending the message to the object. Theoretically, one doesn’t want to see the complicated algorithms or data structures in the main program, just the object and message being sent. In the conventional program, in which complex logic is developed, reading the program is difficult. However, OOP design is more concise, readable and easier to debug. Looking the final stage to see the main program for the robot arm motion will make this clear.

4.1 PROGRAM STRUCTURE

In the main program, the goal is to send the rotate message to the robot and discover the new positions after the command has been executed. There are four steps for the test:

1. Create a robot object puma and initialize the starting position
2. Rotate joint 5, joint 6, and observe the new positions
3. Rotate joint 1 only, and observe the result
4. Rotate all the joints and show the new positions

Since all the classes needed in the robot arm motions have been developed, it can be easily performed without having the program skills involved.
4.1.1 STEP 1:

The first step in the program, a robot $i$ created by the constructor, assigning the starting positions by the method start, and showing all the positions by the function showall. In order to understand the entire process, the procedures can be traced back to clearly see what classes are really involved.

```plaintext
robot puma(16,0);
puma.start(90,-90,0,0,
0,0431.8,149.09,
90,90,-20.32,0,
0,-90,0,433.07,
0,90,0,0,
0,0,0,56.25);
puma.showall( );
```

Trace each command as:

```
puma(16,0) => joint1(16,0), joint2(16,0), ................... joint6(16,0)

joint(16,0), joint(16,0), ................... joint(16,0)

array(16,0), array(16,0), ................... array(16,0)
```

Plainly, creating a robot object will evoke all the ancestor constructors, then the values will be sent back from the top parent class to the current class.
puma.start(90,-90,0,0,
0,0,431.8,149.09,
90,90,-20.32,0,
0,-90,0,433.07,
0,90,0,0,
0,0,0,56.25);

=> joint1.start(90,-90,0,0), joint2.start(0,0,431.8, 149.09),............joint6.start(0,0,0,56.25)

joint.start(90,-90,0,0), joint.start(0,0,431.8,149.09), .......... joint.start(0,0,0,56.25)

The puma start calls the joint1, joint2,...joint6 first. Since the start function is defined in the joint class, the message will be sent to the joint class. All these abilities are due to the inheritance feature.

puma.showall( ) => puma.show(1), puma.show(2),......... puma.show(6)

joint1 :: value(i), joint2 :: value(j),............ joint6 :: value(n)

joint.show(1), joint.show(2),............ joint.show(6)

The showall evokes all the classes needed to display the joint positions, including the robot, joint1, joint2,....joint6, and the joint class. The show function is defined in different classes(both robot and joint). This will be discussed in the next section.
4.1.2 STEP 2:

In the second step, two rotations are made for the joint 5 and joint 6. Two rotate functions are used and the showall function displays all the joint positions.

```cpp
puma.rotate(6,25.5);
puma.rotate(5,34.3);
puma.showall();
```

Here showall will remain the same, the rotate command will use the joint5, joint6 functions as:

```plaintext
puma.rotate(6,25.5) => joint5 :: rotate(25.5)
puma.rotate(5,34.3) => joint6 :: rotate(34.3)
```

Since joint5, joint6 are not the members of the robot class, we use :: scope to identify that the function is accessed through the multiple inheritance.

4.1.3 STEP 3,4

Step 3 and step 4 repeat the same procedures for the rotate function of different joints and display the positions by using showall. The output and analysis of the result will be discussed in the next section.
4.2 OUTPUT ANALYSIS

The output of the program shows all the joint positions relative to the base coordinate (x, y, z). There are four variables to describe the positions for each joint:

- $X_i$: new x axis relative to the base (x, y, z) $i=1,2,...,6$
- $Y_i$: new y axis relative to the base (x, y, z) $i=1,2,...,6$
- $Z_i$: new z axis relative to the base (x, y, z) $i=1,2,...,6$
- $P_i$: new origin relative to the base (x, y, z) $i=1,2,...,6$

Fig. 4.1 shows the representations of variables above.

Fig 4.1 Joint variables relative to the base coordinate system
From step 1, we get the output for the initial positions of the robot puma:

**COORDINATE OF JOINT : 1**

<table>
<thead>
<tr>
<th></th>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>X1</td>
<td>0.0000</td>
<td>1.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>Y1</td>
<td>0.0000</td>
<td>0.0000</td>
<td>-1.0000</td>
</tr>
<tr>
<td>Z1</td>
<td>-1.0000</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>P1</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
</tbody>
</table>

**COORDINATE OF JOINT : 2**

<table>
<thead>
<tr>
<th></th>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>X2</td>
<td>0.0000</td>
<td>1.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>Y2</td>
<td>0.0000</td>
<td>0.0000</td>
<td>-1.0000</td>
</tr>
<tr>
<td>Z2</td>
<td>-1.0000</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>P2</td>
<td>-149.0894</td>
<td>431.8002</td>
<td>0.0002</td>
</tr>
</tbody>
</table>

**COORDINATE OF JOINT : 3**

In step 2, we rotate joint 6 with the angle of 25.5, joint 5 with angle 34.3. Since the positions relative to the base coordinate are calculated by:

\[ ^0T_i = ^0A_1^1A_2^2A_3^{i-1}A_i \quad i=1,2,3,\ldots,6 \]

Thus,

\[ ^0T_1 = ^0A_1 \]

\[ ^0T_2 = ^0A_1^1A_2 \]

\[ ^0T_3 = ^0A_1^1A_2^2A_3 \]
\[ 0T_4 = 0A_1^1A_2^2A_3^3A_4 \]
\[ 0T_5 = 0A_1^1A_2^2A_3^3A_4^4A_5 \]
\[ 0T_6 = 0A_1^1A_2^2A_3^3A_4^4A_5^5A_6 \]

Rotating joint 5, joint 6 means changing the value of \( ^4A_5 \) and \( ^5A_6 \), so only \( ^0A_5 \) and \( ^0A_6 \) will be changed. That is, joint 1 to joint 4 should remain in the same positions. The output shows that joint 1 to joint 4 remain the same, joint 5 and joint 6 move to the new positions as:

**COORDINATE OF JOINT : 5**

<table>
<thead>
<tr>
<th></th>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>X5</td>
<td>0.0000</td>
<td>-0.5635</td>
<td>-0.8261</td>
</tr>
<tr>
<td>Y5</td>
<td>-1.0000</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>Z5</td>
<td>0.0000</td>
<td>0.8261</td>
<td>-0.5635</td>
</tr>
<tr>
<td>P5</td>
<td>-149.0894</td>
<td>864.8702</td>
<td>20.3208</td>
</tr>
</tbody>
</table>

**COORDINATE OF JOINT : 6**

<table>
<thead>
<tr>
<th></th>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>X6</td>
<td>-0.4305</td>
<td>-0.5086</td>
<td>-0.7456</td>
</tr>
<tr>
<td>Y6</td>
<td>-0.9026</td>
<td>0.2426</td>
<td>0.3556</td>
</tr>
<tr>
<td>Z6</td>
<td>0.0000</td>
<td>0.8261</td>
<td>-0.5635</td>
</tr>
<tr>
<td>P6</td>
<td>-149.0895</td>
<td>911.3383</td>
<td>-11.3775</td>
</tr>
</tbody>
</table>
In step 3, although only joint 1 is rotated, however, the values of $^0A_1$ will be changed. Once $^0A_1$ is changed, it will affect the $^0T_1, ^0T_2, \ldots, ^0T_6$, so all the joints will move to the new positions.

The output shows all the values are changed. Refer to Appendix B to see the data.

In step 4, all the joints are rotated. Of course, all the joint positions will be changed. All these data are shown in Appendix B.
5.0 FURTHER DISCUSSION AND SUMMARY

So far an OOP model has been developed to implement the robot arm motions. During the whole process, some advantages are found by using OOP design concept. However, as many other new products, OOP does evidence some problems and needs to be improved. Some important issues will be addressed in the following discussion.

5.1 ADVANTAGES OF OOP

The advantages of using OOP has been mentioned in the beginning. Through the entire programming procedures, the advantages concerning the OOP design will be summarized as:

1. REUSABILITY:

   In this program, four main classes: array, joint, joint1(joint2,...) robot are created. These classes are stored in different files and compiled separately. Once the main program is run, these classes are linked together. In many cases, some errors were found in one of the files, so corrections and recompilation were made. Since these classes are independent, every time the main program is executed, one can always access the files which are already compiled. That is, the error of one class will not affect the other classes, so the old codes can be reused quickly. In the conventional program, one sometimes wastes too much time in the compilation due to the lack of modularity. For example, the array class is never recompiled no matter what happens in other classes. In the C++ environment, this advantage is extremely distinct.
2. POLYMORPHISM

Polymorphism means the user can create multiple definitions for operators and functions. In this project, any function can be used variously as:

- `double & operator[](int i);`  // define for array class
- `array operator*(array & a);`
- `joint operator*(joint & a);`
- `void show();`  // defined for array class
- `void show(int i);`  // defined for joint class
- `void show(int n);`  // defined for robot class
- `void start(double t, double s, double a, double d);`  // defined for joint class
- `void start(double t, double s, double a, double d);`  // defined for robot class

Clearly, the operator [], *, function show, start are used more than one time in different situations. Use of different names for identification would be very inconvenient. Most importantly, all the ancestors will need to be checked to verify that there is no duplication. However, with the polymorphism feature, one can define the function in a very simple way. The operator overloading like [], *, really makes the program much easier to write. For example,

- `joint j1(16,5);`
- `cout<< j1[5];`  // instead of defining a function like j1.get(5)
- `joint j2=j1*j1;`  // instead of defining a function like j1.mult(j1)

That’s one of the reason one can access the program very efficiently and correctly and even more easily to extend new functions.
3 INHERITANCE

Inheritance is probably the most important feature for the OOP environment, especially the multiple inheritance. During the program execution, the robot class uses many functions inherited from the parent classes, including:

    // from array class:
    array(unsigned int, double val);
    double & operator[](int i);
    array & operator=(array & a);

    // from joint class
    joint operator*(joint & a);
    void start(double t,double s,double a,double d)
    void show(int n);
    double value(int i);

    // from joint1 to joint6
    void rotate(double t);

Using the existing classes to define a new class not only saves time but also avoids errors. In this example, once all the array, joint, joint1, joint2, ..., joint6 classes are verified as correct, then one can build the robot class very easily.
4. extensibility

Suppose one wants to build another robot class, such as Stanford robot, he simply has to change the joint condition. Puma is a robot with 6 rotation joints, while the Stanford robot has two translation joints. The array and joint classes remain the same, the only changes are to the joint1, joint2, ..., joint6 class, and modification of the robot class. For example, if joint 2 and 3 are the translation joints, one can replace the rotate command to translate as:

```c
joint2 : public joint
{
    joint2( );
    ..........
    trans(double I);
}
```

The same thing works for the joint3 class. In the robot class, adjustments can be made as:

```c
Class robot : public joint1, public joint2, ..........
{
    .................
    trans( int n, double I); // add one more command
}
```

All else remain the same. One can see how easily extension from one class to the other by using the OOP design method is accomplished, since these are real visible objects.
5.2 DISADVANTAGES OF OOP

Although there are many good features of the OOP design, however, some problems arose during the process. Just because the goal of OOP is to avoid using independent function in the main program, the user has to try everything possible to encapsulate the functions in the class; that is, make every function as a member of some class. The way to combine the functions is to create the class by inheriting some other classes. For example, one creates the robot class by multiple inheritance to relate the joints together. Too many inheritances can result in several disadvantages:

1. The inherited functions are too many and difficult to memorize:

Recall the robot class, the total functions (except the constructors) inherited are:

```cpp
// from array class:

double & operator[ ](int i);

array operator*(array & a);

array operator^*(array & a);

array & operator=(array & a);

void show();

// from joint class

joint operator*(joint & a);

void start(double t, double s, double a, double d)

void show(int n);

double value(int i);

// from joint1 to joint6

void rotate(double t);
```
Since these functions come from different levels, unless the user makes a list, confusion can result. Although the robot class inherits 15 functions, it may use only part of them. Inheriting too many useless functions does not help the user. How to make proper inheritance and avoid the depth? The suggestion is to combine the member classes and inheritance. For example, one can create the robot class like:

```cpp
class robot {
    protected:
        joint1 j1;
        joint2 j2;
        joint6 j6;
    public:
        ...........
}
```

As mentioned before, using the member classes will not render possible access to the functions of the member classes directly in the main program. However, if only one function is needed from the member classes, it can be defined in the robot class. For example:

```cpp
puma.j1.rotate(25);  // not allowed in the main program
```

Then, we can define a function in the robot program as:

```cpp
void robot :: rotate1(double t)
```
{  
  j1.rotate(t);  
}

In this case, we can use this in the main program:

puma.rotate1(25);  // acceptable since rotete1 is the member function of robot class

Even though the functions are not inherited from the joint1, joint2, ..., joint6, they can still be accessed inside the member functions.

2. Functions with the same names need adjustments:

As visible on page 48, the bold size functions mean they are defined more than one time. This is acceptable. But if the object is created by using the dynamic allocation, that is, using the pointer, then it will cause problems. For example, in the main program:

main()  
{
  robot puma(16,0);  
  robot *pu = &puma;  // create a pointer pu  
  pu->show(1);  // display the first joint
}

Although show is a function of the robot class, however, using the pointer will cause the program to trace to the base class and use the base function. That is, it will evoke either the show in the array class or the show in the joint class. In C++, in order to
solve the problem, one must define the base class function as virtual. For example, declare the virtual in the array class as:

```cpp
class array
{
    protected:
        double *arr;
        int size;
    public:
        virtual void show(); // declare a virtual function
};
```

Once the function is declared as virtual, all of its derived class will have the same function as the virtual feature. The program then understands to find the current object rather than using the base class method. Anyway, one must know in advance in order to create the same methods in the derived classes.

### 3. COMMON ANCESTOR FOR MULTIPLE INHERITANCE:

The inheritance will cause another problem when derived classes have a common ancestor. For example, the robot class inherits the protected data from joint1, joint2, joint3, joint4, joint5, and joint6, but joint1, joint2, joint3, joint4, joint5, and joint6 have a common ancestor joint. In some cases, it will duplicate the inherited data. The whole flow is show in fig. 5.1
Fig 5.1 common ancestor for the joint1, joint2, ..., joint6 in multiple inheritance
The protected data has been inherited to the robot class through joint1, joint2,...joint6. Actually, only one inheritance is necessary. This problem depends on the compiler, which may or may not accept inheritance from common ancestors. In C++, it is suggested to declare a virtual class as:

```cpp
class joint1 : virtual public joint
class joint2 : virtual public joint

........................................
```

As long as one declares a virtual base, the robot class will skip the joint1, joint2,...joint6 class and inherits the data directly from the joint class, so one uses only one path. However, if the inheritance is too deep, sometimes it's difficult to discover the problem. Once again, making a good arrangement for inheritance is very important.

**4. COMMUNICATIONS AMONG UNRELATED OBJECTS:**

Here the so-called unrelated objects refers to lack of inheritance relation. In C++, one can declare a `friend` function to access two objects. For example, one can declare a `friend` function in the joint1 and joint2 class like:

```cpp
class jopint1 : public joint
{
  public :
    ........
    friend double sum(joint1 & a1,joint2 & a2)
}
```
class jopint2 : public joint
{
    public :

    ..........

friend double sum(joint1 & a1, joint2 & a2)
}

void sum(joint1 & a1, joint2 & a2)
{
    return a1+a2;
}

Although friend function makes it possible to access two objects, its use is not encouraged. The future OOP developments will focus more on object communications as the network model.
5.3 SUMMARY

Object-Oriented Programming really shows great advantages for future software design. In the integrated manufacturing environment, OOP will play an important role for the CIM areas. This project is just the beginning to show the potential adventure. We do believe this challenge will continue motivating the engineer to an new era in the computer technology world.
REFERENCES:


Ben Ezzell, "Turbo C++ Programming", Addison-Wesley, 1990

Lee Atkinson and Mark Atkinson, "Using Borland C++", QUE Corp. 1991

C.J.Date, "Database Systems", Addison-Wesley, 1990, Chapter 5

Jeffrey D. Ullman, "Database And Knowledge-Base Systems", Computer Science Press, 1988, Chapter 5

M.A. Couington, Donald Nute and Andre Vellino, "Prolog Programming in Depth", Scott Foresman and Company, P. 206-211


Dennis Mcleop, "Research Foundations in Object-Oriented and Semantic Database Systems", Prentice-Hall, 1990


Setrag Khoshafian and Razmik Abnons, "Object Orientation Concepts, Languages, Database, User Interfaces", John Wiley & Sons, 1990

APPENDIX A: PROJECT PROGRAM

1. array.h
2. array.cpp
3. joint.h
4. joint.cpp
5. joints.h
6. joints.cpp
7. robot.h
8. robot.cpp
9. chiu.cpp
// array.h -- define array class //

class array
{
    protected:
    double* arr;
    int size;

    public:
    // constructors //
    array();                      // default constructor //
    array(unsigned int n,double val);
    array(double *pn,unsigned int n);
    array(array & a);
    // destructor //
    array();
    // overloaded operator //
    double & operator[](int i);
    array operator*(array & a);
    array operator^(array & a);
    array & operator=(array & a);
    // method the shoe the elements //
    void show();
};
// array.cpp -- define array methods //

#include <iostream.h>
#include <stdlib.h>
#include <iomanip.h>
#include "a:\array.h"

// default constructor //
array :: array()  
{  
    arr = NULL;  
    size = 0;  
}

// initialize the array with the same value //
array :: array(unsigned int n,double val)  
{  
    arr = new double[n];  
    size = n;  
    for (int i=0; i<size; i++)  
        arr[i] = val;  
}

// assign a double array to the array class //
array · :: array (double *pn, unsigned int n)  
{  
    arr = new double[n];  
    size = n;  
    for (int i=0; i<size; i++)  
        arr[i] = pn[i];  
}

// copy constructor //
array :: array(array & a)  
{  
    size=a.size;  
    arr = new double[size];  
    for (int i=0; i<size; i++)  
        arr[i] = a.arr[i];  
}

// destructor //
array :: ~array()  
{  
    delete arr;  
}

// overloaded orerator return the nth element //
double & array :: operator[](int i)  
{  
    return arr[i];  
}

// overloaded operator to multiplycate two arrays //
array array :: operator*(array & a)  
{  
    array product(size,0);  
    for (int i=0; i<size; i++)  
        product[i] = arr[i] * a.arr[i];  
    return product;  
}

// overloaded operator used as the dot function to return a value //
double e;
for (int i=0; i<size; i++)
e += arr[i] * a.arr[i];
array elem(size, e);
return elem;
}

// overloaded operator to assign the array object from one to the other
array & array::operator=(array & a)
{
    if (this == &a)
        return *this;
    delete arr;
    size = a.size;
    arr = new double[size];
    for (int i=0; i<size; i++)
        arr[i] = a.arr[i];
    return *this;
}

// method to show the all elements of the array
void array::show()
{
    for (int i=0; i<size; i++)
        cout << setw(15) << setprecision(4) << arr[i] << "\n";
}
// joint.h -- define joint class //

#include "a:\array.h"

class joint : public array
{

public:

    // Constructors and destructor inherited from the array class //
    joint();
    joint(unsigned int n, double val);
    joint(double *pn, unsigned int n);
    joint(joint & a);
    ~joint();

    // new methods for the joint class //
    joint operator*( joint & a);
    void start(double t, double s, double a, double d);
    void show(int n);
    double value(int i);

};
// joint.cpp -- joint class methods //

#include <iostream.h>
#include <stdlib.h>
#include <iomanip.h>
#include <math.h>
#include "a:\joint.h"

// default constructor -- no arguments //
joint ::= joint()
{
}

// initial constructor inherited from array class //
joint ::= joint(unsigned int n,double val) : array(n,val)
{
}

// initial constructor inherited from array class //
joint ::= joint(double *pn,unsigned int n) : array(*pn,n)
{
}

// copy constructor inherited from array class //
joint ::= joint(joint & a) : array(a)
{
}

// destructor inherited from array class //
joint ::= ~joint()
{
}

// method to start the joint coordinates //
// known as the Denavit-Hartenberg transformation matrix //
void joint ::= start(double t,double s,double a,double d)
{
    double rad = 2*3.14159/360;
    arr[0]=cos(t*rad);
    arr[1]=sin(t*rad);
    arr[2]=0;
    arr[3]=0;
    arr[4]=-cos(s*rad)*sin(t*rad);
    arr[5]=cos(s*rad)*cos(t*rad);
    arr[6]=sin(s*rad);
    arr[7]=0;
    arr[8]=sin(s*rad)*sin(t*rad);
    arr[9]=-sin(s*rad)*cos(t*rad);
    arr[10]=cos(s*rad);
    arr[11]=0;
    arr[12]=a*cos(t*rad);
    arr[13]=a*sin(t*rad);
    arr[14]=d;
    arr[15]=1;
}

// New operator * defined for joint multiplication //
joint joint ::= operator*(joint & a)
{
    joint pr(size,0);
pr[0]=arr[0]*a[0]+arr[4]*a[1]+arr[8]*a[2]+arr[12]*a[3];
pr[3]=0;
pr[7]=0;
pr[11]=0;
pr[15]=1;

return pr;

void joint :: show(int n)
{
    cout.setf(ios::showpoint);
    cout<<"\n\nCOORDINATES OF JOINT : "<<n<<"\n";
    cout<<" x y z";
    for (int i=0;i<3;i++)
    {
        if(-0.00001<arr[i] && arr[i] <0.00001)
            arr[i]=0;
        cout<< setw(15)<<(setprecision(4)<<arr[i];
    }
    cout<<" X"<< n <<" :";
    for (int j=4;j<7;j++)
    {
        if(-0.00001<arr[j] && arr[j]<0.00001)
            arr[j]=0;
        cout<< setw(15)<<(setprecision(4)<<arr[j];
    }
    cout<<" Y"<< n <<" :";
    for (int k=8;k<11;k++)
    {
        if(-0.00001<arr[k] && arr[k]<0.00001)
            arr[k]=0;
        cout<< setw(15)<<(setprecision(4)<<arr[k];
    }
    cout<<" Z"<< n <<" :";
    for (int l=12;l<15;l++)
    {
        if(-0.00001<arr[l] && arr[l]<0.00001)
            arr[l]=0;
        cout<< setw(15)<<(setprecision(4)<<arr[l];
    }
    cout<<" P"<< n <<" :";
    char ch;
    ch=cin.get();
}

double joint :: value(int n)
{ // Method to get the value of the array explicitly
    // New method to show the joint coordinates referred to the base
}
return arr[n];
//joints.h : define joints class //

#include "a:\joint.h"

// All constructors and destructors are derived from the joint class //
// New method rotate for each joint //

class joint1 : public joint
{
    public:
        joint1();
        joint1(unsigned int n,double val);
        joint1(double *pn,unsigned int n);
        joint1(joint1 & a);
        joint1();
        void rotate(double t);
};

class joint2 : public joint
{
    public:
        joint2();
        joint2(unsigned int n,double val);
        joint2(double *pn,unsigned int n);
        joint2(joint2 & a);
        joint2();
        void rotate(double t);
};

class joint3 : public joint
{
    public:
        joint3();
        joint3(unsigned int n,double val);
        joint3(double *pn,unsigned int n);
        joint3(joint3 & a);
        joint3();
        void rotate(double t);
};

class joint4 : public joint
{
    public:
        joint4();
        joint4(unsigned int n,double val);
        joint4(double *pn,unsigned int n);
        joint4(joint4 & a);
        joint4();
        void rotate(double t);
};

class joint5 : public joint
{
    public:
        joint5();
        joint5(unsigned int n,double val);
        joint5(double *pn,unsigned int n);
        joint5(joint5 & a);
        joint5();
        void rotate(double t);
};
class joint6 : public joint
{
    public :
        joint6();
        joint6(unsigned int n, double val);
        joint6(double *pn, unsigned int n);
        joint6(joint6 & a);
        joint6();
        void rotate(double t);
};
// joints.cpp -- joint1 to joint6 class method //

#include <iostream.h>
#include <stdlib.h>
#include <math.h>
#include "a:\joints.h"

// default constructor //
joint1 :: joint1()
{
}

// initial constructor inherited from joint //
joint1 :: joint1(double *pn,unsigned int n) : joint(*pn,n)
{
}

// initial constructor inherited from joint //
joint1 :: joint1(unsigned int n,double val) : joint(n,val)
{
}

// copy constructor inherited from joint class //
joint1 :: joint1(joint1 & a) : joint(a)
{
}

// method for the joint rotation //
void joint1 :: rotate(double t)
{
    double rad = 2*3.14159/360;
    arr[0]=cos(t*rad);
    arr[1]=sin(t*rad);
    arr[8]=-sin(t*rad);
    arr[9]=cos(t*rad);
}

// destructor inherited from joint class //
joint1 :: ~joint1()
{
}

// default constructor //
joint2 :: joint2()
{
}

// initial constructor inherited from joint //
joint2 :: joint2(double *pn,unsigned int n) : joint(*pn,n)
{
}

// initial constructor inherited from joint //
joint2 :: joint2(unsigned int n,double val) : joint(n,val)
{
}

// copy constructor inherited from joint class //
joint2 :: joint2(joint2 & a) : joint(a)
{
}
// method for the joint rotation //
void joint2 ::rotate(double t)
{
    double rad = 2*3.14159/360;
    arr[0]=cos(t*rad);
    arr[1]=sin(t*rad);
    arr[4]=-sin(t*rad);
    arr[5]=cos(t*rad);
    arr[12]=431.8*cos(t*rad);
    arr[13]=431.8*sin(t*rad);
}

// destructor inherited from joint class //
joint2 :: ~joint2()
{
}

// default constructor //
joint3 :: joint3()
{
}

// initial constructor inherited from joint //
joint3 :: joint3(double *pn,unsigned int n) : joint(*pn,n)
{
}

// initial constructor inherited from joint //
joint3 :: joint3(unsigned int n,double val) : joint(n,val)
{
}

// copy constructor inherited from joint class //
joint3 :: joint3(joint3 & a) : joint(a)
{
}

// method for the joint rotation //
void joint3 ::rotate(double t)
{
    double rad = 2*3.14159/360;
    arr[0]=cos(t*rad);
    arr[1]=sin(t*rad);
    arr[8]=sin(t*rad);
    arr[9]=-cos(t*rad);
    arr[12]=-20.32*cos(t*rad);
    arr[13]=-20.32*sin(t*rad);
}

// destructor inherited from joint class //
joint3 :: ~joint3()
{
}

// default constructor //
joint4 :: joint4()
{
}
// initial constructor inherited from joint
joint4 :: joint4(double *pn,unsigned int n) : joint(*pn,n) {
}

// initial constructor inherited from joint
joint4 :: joint4(unsigned int n,double val) : joint(n,val) {
}

// copy constructor inherited from joint class
joint4 :: joint4(joint4 & a) : joint(a) {
}

// method for the joint rotation
void joint4 :: rotate(double t) {
   double rad = 2*3.14159/360;
   arr[0]=cos(t*rad);
   arr[1]=sin(t*rad);
   arr[8]=-sin(t*rad);
   arr[9]=cos(t*rad);
}

// destructor inherited from joint class
joint4 :: ~joint4() {
}

// default constructor
joint5 :: joint5() {
}

// initial constructor inherited from joint
joint5 :: joint5(double *pn,unsigned int n) : joint(*pn,n) {
}

// initial constructor inherited from joint
joint5 :: joint5(unsigned int n,double val) : joint(n,val) {
}

// copy constructor inherited from joint class
joint5 :: joint5(joint5 & a) : joint(a) {
}

// method for the joint rotation
void joint5 :: rotate(double t) {
   double rad = 2*3.14159/360;
   arr[0]=cos(t*rad);
   arr[1]=sin(t*rad);
   arr[8]=sin(t*rad);
   arr[9]=-cos(t*rad);
}
// destructor inherited from joint class //
joint5 :: joint5()
{
}

// default constructor //
joint6 :: joint6()
{
}

// initial constructor inherited from joint //
joint6 :: joint6(double *pn,unsigned int n) : joint(*pn,n)
{
}

// initial constructor inherited from joint //
joint6 :: joint6(unsigned int n,double val) : joint(n,val)
{
}

// copy constructor inherited from joint class //
joint6 :: joint6(joint6 & a) : joint(a)
{
}

// method for the joint rotation //
void joint6 :: rotate(double t)
{
    double rad = 2*3.14159/360;
    arr[0]=cos(t*rad);
    arr[1]=sin(t*rad);
    arr[4]=-sin(t*rad);
    arr[5]=cos(t*rad);
}

// destructor inherited from joint class //
joint6 :: joint6()
{
}
// robot.h -- define robot class //

#include "a:\joints.h"

// multiple inheritance from joint1, joint2, joint3, joint4, joint5, joint6 //
// for the robot class //
class robot : public joint1, public joint2, public joint3, public joint4,
                   public joint5, public joint6
{
public:

    // constructors and destructor derived from the joints class //
    robot();
    robot(unsigned int n, double val);
    robot();

    // 4 new methods for the robot class //
    void start(double t1, double s1, double a1, double d1,
               double t2, double s2, double a2, double d2,
               double t3, double s3, double a3, double d3,
               double t4, double s4, double a4, double d4,
               double t5, double s5, double a5, double d5,
               double t6, double s6, double a6, double d6);
    void rotate(int n, double t);
    void show(int n);
    void showall();
};
// robot.cpp -- define the robot methods

#include <iostream.h>
#include <stdlib.h>
#include "a:\robot.h"

// default constructor //
robot :: robot()
{
}

// initialize the robot with the ancestors //
robot :: robot(unsigned int n,double val) :
    joint1(n,val),joint2(n,val),
    joint3(n,val),joint4(n,val),
    joint5(n,val),joint6(n,val)
{
}

// destructor //
robot :: ~robot()
{
}

// method to start the robot by invoking the ancestor joints //
void robot :: start(double t1,double s1,double a1,double d1,
    double t2,double s2,double a2,double d2,
    double t3,double s3,double a3,double d3,
    double t4,double s4,double a4,double d4,
    double t5,double s5,double a5,double d5,
    double t6,double s6,double a6,double d6)
{
    joint1::start(t1,s1,a1,d1);
    joint2::start(t2,s2,a2,d2);
    joint3::start(t3,s3,a3,d3);
    joint4::start(t4,s4,a4,d4);
    joint5::start(t5,s5,a5,d5);
    joint6::start(t6,s6,a6,d6);
}

// rotate method for robot with 6 joints //
void robot :: rotate(int n,double t)
{
    switch(n)
    {
        case 1: joint1 :: rotate(t);
            break;

        case 2: joint2 :: rotate(t);
            break;

        case 3: joint3 :: rotate(t);
            break;

        case 4: joint4 :: rotate(t);
            break;

        case 5: joint5 :: rotate(t);
            break;

        case 6: joint6 :: rotate(t);
            break;
    }
}
// method to show the coordinates of the robot joint
void robot :: show(int n)
{
    switch(n)
    {
        case 1 :
        {
            joint a1(16,0);
            for(int i=0;i<16;i++)
                a1[i]=joint1 :: value(i);
            a1.show(1);
            break;
        }
        case 2 :
        {
            joint b1(16,0);
            joint b2(16,0);
            for(int j=0;j<16;j++)
            {
                b1[j]=joint1 :: value(j);
                b2[j]=joint2 :: value(j);
            }
            joint b=b1*b2;
            b.show(2);
            break;
        }
        case 3 :
        {
            joint c1(16,0);
            joint c2(16,0);
            joint c3(16,0);
            for(int k=0;k<16;k++)
            {
                c1[k]=joint1 :: value(k);
                c2[k]=joint2 :: value(k);
                c3[k]=joint3 :: value(k);
            }
            joint c=c1*c2*c3;
            c.show(3);
            break;
        }
        case 4 :
        {
            joint d1(16,0);
            joint d2(16,0);
            joint d3(16,0);
            joint d4(16,0);
            for(int l=0;l<16;l++)
            {
                d1[l]=joint1 :: value(l);
                d2[l]=joint2 :: value(l);
                d3[l]=joint3 :: value(l);
                d4[l]=joint4 :: value(l);
            }
            joint d=d1*d2*d3*d4;
            d.show(4);
            break;
        }
        case 5 :
        {
            joint e1(16,0);
            joint e2(16,0);
            joint e3(16,0);
            joint e4(16,0);
            joint e5(16,0);
            for(int m=0;m<16;m++)
            {
                e1[m]=joint1 :: value(m);
                e2[m]=joint2 :: value(m);
                e3[m]=joint3 :: value(m);
                e4[m]=joint4 :: value(m);
                e5[m]=joint5 :: value(m);
            }
        }
    }
}


} joint e=e1*e2*e3*e4*e5;
e.show(5);
break;

case 6 :
 joint f1(16,0);
 joint f2(16,0);
 joint f3(16,0);
 joint f4(16,0);
 joint f5(16,0);
 joint f6(16,0);
 for(int n=0;n<16;n++)
{
    f1[n]=joint1 : value(n);
    f2[n]=joint2 : value(n);
    f3[n]=joint3 : value(n);
    f4[n]=joint4 : value(n);
    f5[n]=joint5 : value(n);
    f6[n]=joint6 : value(n);
}
 joint f=f1*f2*f3*f4*f5*f6;
 f.show(6);
 break;

} // method to show all the joints position //
void robot ::showall()
{
    show(1);
    show(2);
    show(3);
    show(4);
    show(5);
    show(6);
}
// chiu.cpp -- main program to show the robot kinematics //

#include <iostream.h>
#include <stdlib.h>
#include "a:\robot.h"

int main(void)
{
    // initialize the robot PUMA and show its starting positions for each joint

    robot puma(16,0);
    puma.start(90,-90,0,0,
              0,0,431.8,149.09,
              90,90,-20.32,0,
              0,-90,0,433.07,
              0,90,0,0,
              0,0,0,56.25);

    cout<<"The initial 6 joint positions referred to the base(x,y,z) for the robot PUMA :");
    puma.showall();

    // rotate only joint 5,6 to see the position change/

    puma.rotate(6,25.5);
    puma.rotate(5,34.3);
    cout<<"After two rotations for joint 5,6, the positions ");
    cout<<"referred to the base(x,y,z)";
    puma.showall();

    // rotate only joint 1, to see the position change/

    puma.rotate(1,22.2);
    cout<<"After rotations for joint 1, the positions ");
    cout<<"referred to the base(x,y,z)";
    puma.showall();

    // rotate each joint with different degrees and show the positions after rotation /

    puma.rotate(1,30);
    puma.rotate(2,45);
    puma.rotate(3,125);
    puma.rotate(4,-23.1);
    puma.rotate(5,46.7);
    puma.rotate(6,14.5);

    cout<<"After 6 rotations, the joints positions referred to the";
    cout<<"base(x,y,z) : ");
    puma.showall();

    return 0;
}
The initial 6 joint positions referred to the base(x,y,z) for the robot PUMA:

COORDINATES OF JOINT : 1
\begin{align*}
X_1 & : 0.0000 & Y_1 & : 1.0000 & Z_1 & : 0.0000 \\
Y_1 & : 0.0000 & Y_1 & : 0.0000 & Z_1 & : -1.0000 \\
Z_1 & : -1.0000 & Y_1 & : 0.0000 & Z_1 & : 0.0000 \\
P_1 & : 0.0000 & Y_1 & : 0.0000 & Z_1 & : 0.0000
\end{align*}

COORDINATES OF JOINT : 2
\begin{align*}
X_2 & : 0.0000 & Y_2 & : 1.0000 & Z_2 & : 0.0000 \\
P_2 & : -149.0894 & Y_2 & : 0.0000 & Z_2 & : 0.0000 \\
Y_2 & : 0.0000 & Y_2 & : 0.0000 & Z_2 & : -1.0000 \\
Z_2 & : -1.0000 & Y_2 & : 0.0000 & Z_2 & : 0.0000
\end{align*}

COORDINATES OF JOINT : 3
\begin{align*}
X_3 & : 0.0000 & Y_3 & : 0.0000 & Z_3 & : -1.0000 \\
P_3 & : -149.0894 & Y_3 & : 0.0000 & Z_3 & : 0.0000 \\
Y_3 & : -1.0000 & Y_3 & : 0.0000 & Z_3 & : 0.0000 \\
Z_3 & : 0.0000 & Y_3 & : 1.0000 & Z_3 & : 0.0000
\end{align*}

COORDINATES OF JOINT : 4
\begin{align*}
X_4 & : 0.0000 & Y_4 & : 0.0000 & Z_4 & : -1.0000 \\
P_4 & : -149.0894 & Y_4 & : 0.0000 & Z_4 & : 0.0000 \\
Y_4 & : 0.0000 & Y_4 & : -1.0000 & Z_4 & : 0.0000 \\
Z_4 & : -1.0000 & Y_4 & : 0.0000 & Z_4 & : 0.0000
\end{align*}

COORDINATES OF JOINT : 5
\begin{align*}
X_5 & : 0.0000 & Y_5 & : 0.0000 & Z_5 & : -1.0000 \\
P_5 & : -149.0894 & Y_5 & : 0.0000 & Z_5 & : 0.0000 \\
Y_5 & : -1.0000 & Y_5 & : 0.0000 & Z_5 & : 0.0000 \\
Z_5 & : 0.0000 & Y_5 & : 1.0000 & Z_5 & : 0.0000
\end{align*}

COORDINATES OF JOINT : 6
\begin{align*}
X_6 & : 0.0000 & Y_6 & : 0.0000 & Z_6 & : -1.0000 \\
P_6 & : -149.0894 & Y_6 & : 0.0000 & Z_6 & : 0.0000 \\
Y_6 & : -1.0000 & Y_6 & : 0.0000 & Z_6 & : 0.0000 \\
Z_6 & : 0.0000 & Y_6 & : 1.0000 & Z_6 & : 0.0000
\end{align*}
After two rotations for joint 5,6, the positions referred to the base(x,y,z)

<table>
<thead>
<tr>
<th>COORDINATES OF JOINT : 1</th>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>X1 :</td>
<td>0.0000</td>
<td>1.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>Y1 :</td>
<td>0.0000</td>
<td>0.0000</td>
<td>-1.0000</td>
</tr>
<tr>
<td>Z1 :</td>
<td>-1.0000</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>P1 :</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>COORDINATES OF JOINT : 2</th>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>X2 :</td>
<td>0.0000</td>
<td>1.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>Y2 :</td>
<td>0.0000</td>
<td>0.0000</td>
<td>-1.0000</td>
</tr>
<tr>
<td>Z2 :</td>
<td>-1.0000</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>P2 :</td>
<td>-149.0894</td>
<td>431.8002</td>
<td>0.0000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>COORDINATES OF JOINT : 3</th>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>X3 :</td>
<td>0.0000</td>
<td>0.0000</td>
<td>-1.0000</td>
</tr>
<tr>
<td>Y3 :</td>
<td>-1.0000</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>Z3 :</td>
<td>0.0000</td>
<td>1.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>P3 :</td>
<td>-149.0894</td>
<td>431.8002</td>
<td>20.3202</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>COORDINATES OF JOINT : 4</th>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>X4 :</td>
<td>0.0000</td>
<td>0.0000</td>
<td>-1.0000</td>
</tr>
<tr>
<td>Y4 :</td>
<td>0.0000</td>
<td>-1.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>Z4 :</td>
<td>-1.0000</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>P4 :</td>
<td>-149.0894</td>
<td>864.8702</td>
<td>20.3208</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>COORDINATES OF JOINT : 5</th>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>X5 :</td>
<td>0.0000</td>
<td>-0.5635</td>
<td>-0.8261</td>
</tr>
<tr>
<td>Y5 :</td>
<td>-1.0000</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>Z5 :</td>
<td>0.0000</td>
<td>0.8261</td>
<td>-0.5635</td>
</tr>
<tr>
<td>P5 :</td>
<td>-149.0894</td>
<td>864.8702</td>
<td>20.3208</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>COORDINATES OF JOINT : 6</th>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>X6 :</td>
<td>-0.4305</td>
<td>-0.5086</td>
<td>-0.7456</td>
</tr>
<tr>
<td>Y6 :</td>
<td>-0.9026</td>
<td>0.2426</td>
<td>0.3556</td>
</tr>
<tr>
<td>Z6 :</td>
<td>0.0000</td>
<td>0.8261</td>
<td>-0.5635</td>
</tr>
<tr>
<td>P6 :</td>
<td>-149.0895</td>
<td>911.3383</td>
<td>-11.3775</td>
</tr>
</tbody>
</table>
After rotations for joint 1, the positions referred to the base \((x,y,z)\)

**COORDINATES OF JOINT : 1**

<table>
<thead>
<tr>
<th></th>
<th>(x)</th>
<th>(y)</th>
<th>(z)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(X_1) :</td>
<td>0.9259</td>
<td>0.3778</td>
<td>0.0000</td>
</tr>
<tr>
<td>(Y_1) :</td>
<td>0.0000</td>
<td>0.0000</td>
<td>-1.0000</td>
</tr>
<tr>
<td>(Z_1) :</td>
<td>-0.3778</td>
<td>0.9259</td>
<td>0.0000</td>
</tr>
<tr>
<td>(P_1) :</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
</tbody>
</table>

**COORDINATES OF JOINT : 2**

<table>
<thead>
<tr>
<th></th>
<th>(x)</th>
<th>(y)</th>
<th>(z)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(X_2) :</td>
<td>0.9259</td>
<td>0.3778</td>
<td>0.0000</td>
</tr>
<tr>
<td>(Y_2) :</td>
<td>0.0000</td>
<td>0.0000</td>
<td>-1.0000</td>
</tr>
<tr>
<td>(Z_2) :</td>
<td>-0.3778</td>
<td>0.9259</td>
<td>0.0000</td>
</tr>
<tr>
<td>(P_2) :</td>
<td>343.4587</td>
<td>301.1896</td>
<td>0.0002</td>
</tr>
</tbody>
</table>

**COORDINATES OF JOINT : 3**

<table>
<thead>
<tr>
<th></th>
<th>(x)</th>
<th>(y)</th>
<th>(z)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(X_3) :</td>
<td>0.0000</td>
<td>0.0000</td>
<td>-1.0000</td>
</tr>
<tr>
<td>(Y_3) :</td>
<td>-0.3778</td>
<td>0.9259</td>
<td>0.0000</td>
</tr>
<tr>
<td>(Z_3) :</td>
<td>0.9259</td>
<td>0.3778</td>
<td>0.0000</td>
</tr>
<tr>
<td>(P_3) :</td>
<td>343.4587</td>
<td>301.1896</td>
<td>20.3202</td>
</tr>
</tbody>
</table>

**COORDINATES OF JOINT : 4**

<table>
<thead>
<tr>
<th></th>
<th>(x)</th>
<th>(y)</th>
<th>(z)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(X_4) :</td>
<td>0.0000</td>
<td>0.0000</td>
<td>-1.0000</td>
</tr>
<tr>
<td>(Y_4) :</td>
<td>-0.9259</td>
<td>-0.3778</td>
<td>0.0000</td>
</tr>
<tr>
<td>(Z_4) :</td>
<td>-0.3778</td>
<td>0.9259</td>
<td>0.0000</td>
</tr>
<tr>
<td>(P_4) :</td>
<td>744.4253</td>
<td>464.8215</td>
<td>20.3208</td>
</tr>
</tbody>
</table>

**COORDINATES OF JOINT : 5**

<table>
<thead>
<tr>
<th></th>
<th>(x)</th>
<th>(y)</th>
<th>(z)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(X_5) :</td>
<td>-0.5218</td>
<td>-0.2129</td>
<td>-0.8261</td>
</tr>
<tr>
<td>(Y_5) :</td>
<td>-0.3778</td>
<td>0.9259</td>
<td>0.0000</td>
</tr>
<tr>
<td>(Z_5) :</td>
<td>0.7649</td>
<td>0.3121</td>
<td>-0.5635</td>
</tr>
<tr>
<td>(P_5) :</td>
<td>744.4253</td>
<td>464.8215</td>
<td>20.3208</td>
</tr>
</tbody>
</table>

**COORDINATES OF JOINT : 6**

<table>
<thead>
<tr>
<th></th>
<th>(x)</th>
<th>(y)</th>
<th>(z)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(X_6) :</td>
<td>-0.6336</td>
<td>0.2064</td>
<td>-0.7456</td>
</tr>
<tr>
<td>(Y_6) :</td>
<td>-0.1164</td>
<td>0.9273</td>
<td>0.3556</td>
</tr>
<tr>
<td>(Z_6) :</td>
<td>0.7649</td>
<td>0.3121</td>
<td>-0.5635</td>
</tr>
<tr>
<td>(P_6) :</td>
<td>787.4487</td>
<td>482.3791</td>
<td>-11.3775</td>
</tr>
</tbody>
</table>
After 6 rotations, the joints positions referred to the base \((x,y,z)\):

**COORDINATES OF JOINT : 1**

\[
\begin{array}{ccc}
X & Y & Z \\
X_1 & 0.8660 & 0.5000 & 0.0000 \\
Y_1 & 0.0000 & 0.0000 & -1.0000 \\
Z_1 & -0.5000 & 0.8660 & 0.0000 \\
P_1 & 0.0000 & 0.0000 & 0.0000 \\
\end{array}
\]

**COORDINATES OF JOINT : 2**

\[
\begin{array}{ccc}
X & Y & Z \\
X_2 & 0.6124 & 0.3536 & -0.7071 \\
Y_2 & -0.6124 & -0.3536 & -0.7071 \\
Z_2 & -0.5000 & 0.8660 & 0.0000 \\
P_2 & 189.8773 & 281.7801 & -305.3283 \\
\end{array}
\]

**COORDINATES OF JOINT : 3**

\[
\begin{array}{ccc}
X & Y & Z \\
X_3 & -0.8529 & -0.4924 & -0.1737 \\
Y_3 & -0.5000 & 0.8660 & 0.0000 \\
Z_3 & 0.1504 & 0.0868 & -0.9848 \\
P_3 & 207.2076 & 291.7857 & -301.7997 \\
\end{array}
\]

**COORDINATES OF JOINT : 4**

\[
\begin{array}{ccc}
X & Y & Z \\
X_4 & -0.5883 & -0.7927 & -0.1597 \\
Y_4 & -0.1504 & -0.8683 & 0.9848 \\
Z_4 & -0.7945 & 0.6034 & -0.0681 \\
P_4 & 272.3344 & 329.3876 & -728.2902 \\
\end{array}
\]

**COORDINATES OF JOINT : 5**

\[
\begin{array}{ccc}
X & Y & Z \\
X_5 & -0.5129 & -0.6068 & 0.6072 \\
Y_5 & -0.7945 & 0.6034 & -0.0681 \\
Z_5 & -0.3250 & -0.5174 & -0.7916 \\
P_5 & 272.3344 & 329.3876 & -728.2902 \\
\end{array}
\]

**COORDINATES OF JOINT : 6**

\[
\begin{array}{ccc}
X & Y & Z \\
X_6 & -0.6955 & -0.4364 & 0.5708 \\
Y_6 & -0.6408 & 0.7361 & -0.2180 \\
Z_6 & -0.3250 & -0.5174 & -0.7916 \\
P_6 & 254.0516 & 300.2864 & -772.8203 \\
\end{array}
\]
**VITA:**

Mr. Chien-Jung Chiu was born in July 17, 1963 in Taiwan. He received his B.S. degree from National Chiao Tung University (Hsinchu, Taiwan) in 1985, majoring in Civil Engineering. After two year military service, he worked in an engineering design company for one year in Taipei. In 1988, he came to Lehigh University for the graduate study in Civil Engineering. Next semester, he switched to the Industrial Engineering department to continue his master program. This is the thesis he presented for the master candidacy.
END
OF
TITLE