DynaMechs: An Object Oriented Software Package for Efficient Dynamic Simulation of Underwater Robotic Vehicles

McMillian, Scott

http://hdl.handle.net/10945/45121

Downloaded from NPS Archive: Calhoun
DynaMechs: An Object Oriented Software Package for Efficient Dynamic Simulation of Underwater Robotic Vehicles

Scott McMillan  David E. Orin  Robert B. McGhee

Department of Electrical Engineering  Department of Computer Science
The Ohio State University  Naval Postgraduate School
Columbus, Ohio 43210  Monterey, CA 93943

Keywords: object oriented design, dynamics, hydrodynamics, simulation, robotics, articulated mechanism, software package, UUV.

Abstract

In this chapter, a real-time, graphical simulation system, called DynaMechs, for underwater robotic vehicle (URV) systems is presented. An efficient dynamic simulation algorithm is developed based on an efficient $O(N)$ robot dynamics algorithm where $N$ is the number of links in the system. It has been extended to include the effects of various hydrodynamic forces that are exerted on these systems in underwater environments including added mass, viscous drag, fluid acceleration, and buoyancy forces. Very efficient implementation of the simulation algorithm is achieved through the use of object oriented design (OOD) techniques. This chapter describes the software package, written in C++, that results which is capable of simulating a large class of tree-structured mechanisms having star topologies which include multiple manipulator systems and multi-legged vehicles.

\footnote{Chapter 3 in Underwater Robotic Vehicles: Design and Control, (J. Yuh, ed.), TSI Press, pp. 73-98, 1995.}
1. Introduction

The importance of unmanned underwater vehicles for marine research and subsea development continues to grow because their manned counterparts are much more expensive to develop and maintain (AUVS, 1993). This increase in use has brought about a concomitant need for accurate simulations of these systems (Zyda et al., 1990). For underwater robotic vehicle (URV) systems with manipulators or legs, such simulations must become more sophisticated. As with land- and space-based robotics, accurate dynamic simulation can be a cost effective tool in the development of URV systems. Specifically, dynamic simulation can reduce the need for costly prototypes by eliminating many candidate designs early in the development process. In addition, simulators can aid in the design of control algorithms for these systems. By using the simulated URV to test such algorithms, the possibility of potentially damaging instabilities due to algorithm errors is eliminated, and risks encountered when the control system is finally implemented in hardware are reduced. This is especially important when the alternative is testing in the hazardous and unpredictable underwater environment.

With real-time simulation rates, other uses for dynamic simulators are possible. The first is hardware-in-the-loop simulation where control system hardware and software is tested by interfacing it to a computer running a real-time simulation of the URV (Brutzman et al., 1992). Human-in-the-loop applications can also be implemented when the simulation is coupled with a realistic 3-D graphical display of the system. One such application is to train URV pilots and mission specialists much like aircraft simulators are used to train aviators. Another application can be found in the teleoperation of untethered systems where human control is significantly degraded when significant acoustic communication delays occur. By providing the teleoperator with a simulated display of the system, on-line with no delay, enhanced performance of human-machine interaction can be realized (Funda and Paul, 1990).

In order to achieve real-time rates, efficient dynamic simulation algorithms must be developed for these systems. A number of efficient algorithms have been developed to simulate the dynamics of more common land-based robotic systems. The two most notable approaches to this problem are the Composite Rigid Body (CRB) method (Walker and Orin, 1982) and the Articulated Body (AB) method (Featherstone, 1983). The latter has an advantage because its computation grows linearly with the number of degrees of freedom (DOF), whereas the CRB method has cubic complexity.

The goal of our work is to develop an efficient simulation system for two specific URV systems. The first is a remotely-operated vehicle (ROV), Tiburon, with a Schilling manipulator that is currently under development at the Monterey Bay Aquarium Research Institute (MBARI) and shown in Figure 1. This vehicle is designed to be operated at depths of up to 4000 meters and will be used to deploy and retrieve scientific equipment, and acquire samples for scientific research (Newman and Robison, 1992). The second is the six legged vehicle, Aquarobot, shown in Figure 2, which is under development at the Port and Harbor Research Institute (PHRI) in Japan for surveying and inspection during construction of seawalls (Iwasaki et al., 1987).

To achieve this goal, the first part of this chapter describes the development of an efficient
AB-based algorithm for the computation of the dynamics and hydrodynamics of these systems. In the next section, dynamic equations of motion for a link in a robotic mechanism are presented and the structure of the AB dynamics algorithm is described. Particular emphasis is placed on the spatial notation that is used and the form the equations take. In Section 3, equations to compute hydrodynamic forces exerted on submerged bodies are developed using the same spatial notation. These are used to obtain the dynamic equations of motion for a submerged rigid body. With this, changes made to the AB algorithm lead to a very efficient dynamic and hydrodynamic algorithm for both URV systems.

The implementation of the algorithm and development of the desired real-time graphical simulation system, called DynaMechs*, is a complex programming task that is described in Section 4. Object oriented design (OOD) techniques, such as information hiding or encapsulation, are presented that are especially suited for overcoming complexity management.

---

*Pronounced “dynamics,” and stands for Dynamics of Mechanisms.
problems. Code reuse mechanisms, such as inheritance and polymorphism, which increase productivity are also described. Finally, examples of the use of this package on a Silicon Graphics workstation are given.

2. Articulated Body Dynamics: Background and Notation

The topology of both of the URV systems under consideration are included in a class of tree-structured mechanisms, having star topologies. These have a single body, called the reference member, with a number of serial chains attached. The central body acts as a mobile base for each chain, and its joints are numbered from link 1, which is attached to the body through joint 1, through link N, which is the tip of the chain. The AB dynamics for a single serial chain was first developed in (Featherstone, 1983). This has since been extended to include star topologies in (Freeman and Orin, 1991).

The derivation of these algorithms begins with the set of dynamics equations for the force balance on each rigid body in the system. For a link $i$ in a serial chain shown in Figure 3, the force balance equation is given as follows:

$$f_i - i+1X_i^tf_{i+1} + iF_g = Ia_i - \beta_i,$$

where $f_i$ is the spatial force exerted onto link $i$ by its inboard neighbor and contains the effect of joint $i$'s generalized joint force, $\tau_i$, $f_{i+1}$ is the spatial force exerted by link $i$ onto the next link outboard and contains the effect of $\tau_{i+1}$, and $F_g$ is the gravitational force. These are six-element spatial force vectors combining the three-dimensional moment, $n$, and translational
force, $\mathbf{f}$, vectors such that $\mathbf{f} = [\mathbf{n}^T \allowbreak \mathbf{f}^T]^T$. Note that italic bold variables, such as $\mathbf{f}$, refer to a Cartesian (three-dimensional) vector representing either a rotational or translational quantity, and block bold variables, such as $\mathbf{f}$, refer to a spatial (six-dimensional) vector.

The $6 \times 6$ spatial transformation matrix, $i+1 \mathbf{X}_i$, is used to transform spatial vectors between coordinate systems $i$ and $i + 1$. For single DOF joints this can be specified with four modified Denavit-Hartenberg (MDH) parameters as described in (Craig, 1986). Link $i$’s spatial inertia, $\mathbf{I}_i$, relates the spatial acceleration of the link to the resultant spatial force. This $6 \times 6$ inertia matrix combines the link’s mass and inertia quantities (first and second mass moments). The vector, $\mathbf{a}_i$, is the spatial acceleration of link $i$ which contains both angular, $\dot{\mathbf{ω}}_i$, and translational acceleration, $\mathbf{a}_i$, such that $\mathbf{a}_i = [\dot{\mathbf{ω}}_i^T \allowbreak \mathbf{a}_i^T]^T$. The last term, $\mathbf{β}_i$, is the vector of velocity-dependent Coriolis and centripetal forces.

In the efficient AB algorithm for ground, air, or space vehicles, the gravitational effects can be combined with the spatial acceleration to reduce the amount of computation. This is accomplished with the following force balance equation:

$$\mathbf{f}_i - i+1 \mathbf{X}_i^T \mathbf{f}_{i+1} = \mathbf{I}_i \mathbf{a}'_i - \mathbf{β}_i,$$  \hspace{1cm} (2)

where $\mathbf{a}'_i$ is the acceleration of the body biased by gravitational acceleration, $\mathbf{a}_i = [\mathbf{0} \allowbreak \mathbf{a}_i^T]^T$. This equation forms the basis for the AB algorithm, and with Featherstone’s approach, a new equation is derived which relates $\mathbf{f}_i$ to the dynamic properties of links $i$ through $N$. As illustrated in Figure 4, this relationship is given as follows:

$$\mathbf{f}_i = \mathbf{I}_i^* \mathbf{a}'_i - \mathbf{β}_i^*.$$  \hspace{1cm} (3)

The matrix, $\mathbf{I}_i^*$, is the articulated body inertia of links $i$ through $N$ which is the inertia that is “felt” at the coordinate system attached to link $i$ when the actuator joint torques or forces ($\tau$) from $i + 1$ to $N$ are set to zero. Likewise, the vector, $\mathbf{β}_i^*$, is the bias force exerted on the $i$th link due to the joint and external forces that are exerted on the links of the articulated body. Details of the derivation of the equations needed to efficiently compute $\mathbf{I}_i^*$ and $\mathbf{β}_i^*$ can be found in (McMillan, 1994).
Eq. (3), along with kinematics equations for each link of the chain, can be used to compute the accelerations in the system and results in an algorithm containing three $O(N)$ recursions (Featherstone, 1983). The first is a Forward Kinematics recursion in which velocity and velocity-dependent terms for each body in the system are computed. It is a forward recursion that begins with the reference member and propagates outward along each chain to the tips. In the second step, the articulated-body inertias, $I_i$, and bias forces, $\beta_i$, are computed in a Backward Dynamics recursion from the tips of each chain to the reference member. The final step begins by computing the reference member’s acceleration, $a_0$, and in the Forward Acceleration recursions along each chain, the joint accelerations are computed from joint 1 to joint $N$.

3. Hydrodynamics

For the URV systems, hydrodynamic effects must be added to obtain an efficient hydrodynamic and dynamic simulation algorithm. While the net hydrodynamic force results from incompressible fluid flow determined by the Navier-Stokes (distributed fluid-flow) equations, it is assumed here that it can be represented as a sum of separately identified components for which “lumped” approximations have been used. Using these assumptions, (Yuh, 1990) and (Ioi and Itoh, 1990) have identified the most significant hydrodynamic forces of added mass, viscous drag, buoyancy, and fluid acceleration which are presented in the following discussion. Particular emphasis is placed on developing equations that are consistent with the notation of the previous section so that finally they may be included in Eq. (2) in order to rederive the AB algorithm.

A. Added Mass

To those acquainted with the dynamics of manipulators in space or air, probably the most surprising hydrodynamic effect is the added mass force. When a body is accelerated
through a fluid, some of the surrounding fluid is also accelerated with the body. In an underwater environment, this fluid has significant inertia properties that can be specified with a $6 \times 6$ added mass matrix, $I_b^A$, and a reaction force, $f_b^A$, is exerted onto the body due to this effect. Since the added mass is a function of the body’s surface geometry, there is no concept of principal axes as in rigid body analysis, along which torque and angular momentum are collinear. In fact, with added mass, unlike the rigid body’s mass, an applied translational force can result in a non-collinear acceleration of the center of gravity as well. Consequently, the added mass matrix does not have the same structure as the spatial inertia of a link. For a general body shape, the matrix will be full which leads to notably different dynamic behavior as compared to the rigid body counterpart.

The derivation of this force was performed in (Newman, 1977) for a rigid body accelerating through an unbounded, inviscid fluid undergoing steady, irrotational flow. This was found by taking the derivative of the total momentum of the fluid. As derived in (McMillan, 1994), the equation is written in spatial notation as follows:

$$f_b^A = -I_b^A \begin{bmatrix} \dot{\omega}_b \\ \dot{v}_b \end{bmatrix} - \begin{bmatrix} \omega_b & \dot{v}_b \\ 0 & \omega_b \end{bmatrix} I_b^A \begin{bmatrix} \omega_b \\ v_b \end{bmatrix},$$

(4)

where $\omega_b$ and $v_b$ are the angular and translational velocities of the body, respectively, and the tilde is the cross product operator. The term, $\dot{v}_b$, is the time derivative of the components of $v_b$ with respect to the body’s rotating reference frame.

Since efficient robotics algorithms use the true acceleration of the body, $a_b$, the following relationship is substituted into Eq. (4):

$$a_b = \dot{v}_b + \omega_b \times v_b,$$

(5)

where $\dot{v}_b$ is frequently referred to as the rate of growth of the velocity vector, and $\omega_b \times v_b$ its rate of transport. Then, the effects of fluid translational acceleration are incorporated by replacing $a_b$ with the body’s translational acceleration relative to the surrounding fluid, $a_b^r$ ([Newman, 1977], p. 150] which is given by:

$$a_b^r = a_b - b a_f,$$

(6)

where $b a_f$ is the translational acceleration of the fluid expressed in the body-fixed coordinate system. Likewise, the translational velocity term is replaced with the body’s relative translational velocity, $v_b^r$. Finally, it is necessary to write the equation using the biased acceleration, $a_b^A$, which yields:

$$f_b^A = -I_b^A a_b^A + \beta_b^A,$$

(7)

where

$$\beta_b^A = I_b^A \begin{bmatrix} (b a_f - b a_g) + \omega_b \times v_b^r \\ 0 \end{bmatrix} - \begin{bmatrix} \omega_b & \dot{v}_b^r \\ 0 & \omega_b \end{bmatrix} I_b^A \begin{bmatrix} \omega_b \\ v_b^r \end{bmatrix},$$

(8)

which is the called the added mass bias force, and is a function of the known state, fluid velocity and acceleration, and gravity.
B. Drag and Lift

When an object moves through a viscous fluid, drag and lift forces are exerted on it. Since water density is significant, large viscous forces can be exerted on URV systems even for reasonably slow motions. Lift and the related forces due to vortex shedding are believed to be small for the applications at hand and are ignored. Drag can be decomposed into pressure drag which is normal to the surface of the body and shear drag which is tangential. For underwater manipulation, the shear drag will also typically be small, so that the emphasis here is on the modeling of the pressure drag.

Pressure drag arises from non-zero normal components of relative velocity between the body's surface and the fluid. For a general body, a surface integral over the entire body is required to compute the resultant force and moment, \( \mathbf{f}^D_{b} \). To avoid this integration, links are approximated by cylinders as shown in Figure 5 and the resulting procedure to compute \( \mathbf{f}^D_{b} \) is based on one in (Sarpkaya and Isaacson, 1981) which has been extended in our work to include the effects of arbitrary angular and translational velocity of the cylinder as well.

Strip theory is used to replace the surface integral with a line integral along the length of the cylinder. Therefore, the cylinder is partitioned into circular disk elements with width \( dx \), and the translational velocity relative to the fluid and normal to the edge of each disk, \( \mathbf{v}^n \), must be determined. The translational velocity of a disk relative to the fluid at a distance \( d \) along the cylinder’s axis (the \( x \)-axis in this example) is approximated, assuming its radius is small compared to the length, as follows:

\[
\mathbf{v}^r(d) = \mathbf{v}^{\prime}_b + \omega_b \times \begin{bmatrix} d & 0 & 0 \end{bmatrix}^T,
\]

where \( \omega_b \) is the angular velocity of the cylinder (irrotational fluid assumed) and \( \mathbf{v}^{\prime}_b \) is the translational velocity of the cylinder relative to the fluid at the body-fixed coordinate.
system. The partial force and moment is a function of the component of $\mathbf{v}(d)$ normal to the surface, $\mathbf{v}^n(d)$ (McMillan, 1994), as follows:

$$
\begin{align*}
\delta f^D_d &= -0.5 \rho C_D |\mathbf{v}^n(d)| \mathbf{v}^n(d) (2r \, dx), \\
\delta n^D_d &= -0.5 \rho C_D |\mathbf{v}^n(d)| \left( [d \ 0 \ 0]^T \times \mathbf{v}^n(d) \right) 2r \, dx. 
\end{align*}
$$

These are integrated along the length of the cylinder to obtain $\mathbf{f}^D$. A numerical solution to these integrals has been implemented using Gauss-quadrature. To complete the procedure, drag forces on the flat ends of cylinder can be separately computed. The assumption that this component of the drag force can be computed in this manner from normal components of relative velocity for these surfaces while ignoring components computed for the other surfaces is consistent with the independence principle (Sarpkaya and Isaacson, 1981) and the work of (Chakrabarti et al., 1975).

Another assumption that has been made is that the added mass matrix and drag coefficients are known and constant. In actuality, these quantities are coupled and extremely difficult to compute with a high degree of accuracy, and vary nonlinearly with respect to velocity and other parameters (Sarpkaya and Isaacson, 1981). However, we believe that over the range of operating conditions typically encountered by a URV, a constant coefficient assumption is the only reasonable approach and is adequate for the purposes of the desired applications.

C. Total Buoyancy: Buoyancy and Fluid Acceleration

Because of the similarity between buoyancy and fluid acceleration forces, they are presented together in this section. Both are translational forces, as illustrated in Figure 6, that are exerted through the center of buoyancy, $\mathbf{b}$, of the body and proportional to the mass of the fluid that is displaced by the body, $m_f$. The buoyant force, $f^B_b$, is exerted on the body in the direction opposite of gravity. This force results from Archimedes principle which states that a body immersed in a fluid is buoyed up with a force equal to the weight of the fluid displaced by the body. Likewise, a force resulting from acceleration of the surrounding fluid, $f^F_b$, is exerted in the direction of the fluid acceleration. This force is often referred to...
as the horizontal buoyancy force. Both forces are combined as follows:

\[
f_b^{TB} = m_f (b_a_f - b_a_g),
\]

(12)

where \(f_b^{TB}\) is named the total buoyancy force. The corresponding moment of the spatial force, \(n_b^{TB}\), is computed as follows.

\[
n_b^{TB} = b \times f_b^{TB}.
\]

(13)

Combining these two forces in this fashion results in less computation. The quantity, \(b_a_f - b_a_g\), is also required in the computation of the added mass bias force, \(\beta_i^H\), and because the fluid or gravitational acceleration terms are not needed individually, the subtraction is performed once with respect to the inertial coordinate system and only one vector needs to be transformed to the body’s coordinate system.

D. Hydrodynamic AB Algorithm

The derivation of the AB algorithm for URV systems begins with the force balance equation on link \(i\) including the hydrodynamic effects just presented. Rearranging the terms to resemble the form of Eq. (2) leads to the desired equation:

\[
f_i - i+1X_i^T f_{i+1} = I_i^H a_i' - \beta_i^H,
\]

(14)

where

\[
I_i^H = I_i + I_i^A, \quad \text{and}
\]

\[
\beta_i^H = \beta_i + \beta_i^A + f_i^D + f_i^{TB},
\]

(15)

(16)

which are referred to here as the hydrodynamic inertia and hydrodynamic bias force, respectively. The hydrodynamic AB algorithm can be derived as before by using Eq. (14) in place of Eq. (2). The differences between these equations provide the information necessary to modify the AB algorithm to simulate URV systems. First, all occurrences of \(I_i\) and \(\beta_i\) are replaced with \(I_i^H\) and \(\beta_i^H\) which changes the equations for \(I_i\) and \(\beta_i\) in the Backward Dynamics recursions, but has a negligible effect on the amount of computation.

It is the computation of \(\beta_i^H\), itself, that accounts for nearly all additional computation. This also includes the computation of the velocity of the link relative to the fluid, and fluid and gravitational acceleration with respect to each link’s coordinate system. These terms are most efficiently computed in the Forward Kinematics recursions. From a detailed analysis of the resulting algorithm in (McMillan, 1994), Table 1 lists the computational requirements of the general AB dynamics algorithm for systems containing \(m\) serial chains with \(N\) revolute joints each on land and under water. The second row \((m,N = 1,6)\) corresponds to the computational requirements for the MBARI ROV with the Schilling manipulator, and the last row \((m,N = 6,3)\) corresponds to the Aquarobot (without foot pads). Hydrodynamic simulation increases the amount of computation by approximately 65%. 

Table 1: Computational requirements of the general AB simulation algorithm in terms of the number of multiplications and divisions ($\times$) and additions and subtractions ($+$) for a system with $m$ serial chains with $N$ revolute joints each.

<table>
<thead>
<tr>
<th></th>
<th>Land Based</th>
<th>Underwater</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\times$</td>
<td>$+$</td>
</tr>
<tr>
<td>Total</td>
<td>$224mN+$</td>
<td>$205mN+$</td>
</tr>
<tr>
<td></td>
<td>$59m + 99$</td>
<td>$63m + 68$</td>
</tr>
<tr>
<td>$m, N = 1, 6$</td>
<td>1502</td>
<td>1361</td>
</tr>
<tr>
<td>$m, N = 6, 3$</td>
<td>4485</td>
<td>4136</td>
</tr>
</tbody>
</table>

4. DynaMechs: An Object Oriented Robot Simulation Package

The resulting AB algorithm has been implemented using object oriented design (OOD) techniques in C++ as part of a larger project to develop a real-time graphical simulation system for URVs. Among the advantages of OOD are improved complexity management through the use of information hiding or encapsulation, code reuse and increased productivity through mechanisms like inheritance and polymorphism, and easier maintenance and expandability. This section presents an overview of the design and use of this software package, called DynaMechs.$^\dagger$

A. Object Hierarchy

The first step in OOD is to decompose the task, in a top-down fashion, into a part-of hierarchy of component objects as shown in Figure 7. First the domain is divided into two parts, the robotic system and the surrounding environment, as indicated by the two top-level objects. While the latter contains the attributes for the environment with which the system interacts such as gravity, fluid characteristics, and terrain models, the emphasis in this chapter is on the system object. It must be capable of representing robots with star topologies like the URV systems, and it is also useful to include fixed-base manipulators. Therefore, the system is further decomposed into the reference member (a fixed or mobile base) and a number of serial chains. The serial chains are decomposed into a number of links and an end-effector.

B. Class Hierarchy

With a proper object decomposition, the second step in OOD concentrates on the development of the classes to be used. Booch states that “a class is a set of objects that share a common structure and a common behavior” (Booch, 1986). In other words, classes define the “types” of the objects that appear at the nodes of the object hierarchy. This is accomplished with a kind-of classification of these objects which leads naturally to a second

$^\dagger$This package is available via anonymous FTP from taurus.cs.nps.navy.mil in the /pub/dynmechs/ directory.
hierarchy called the class hierarchy. In the following discussion, the class hierarchy developed for DynaMechs is presented along with the pertinent OOD mechanisms that were used including encapsulation, inheritance, and polymorphism.

1. The C++ Class

As Booch’s definition suggests, a class has two distinct features: member variables that store data or state, and member functions which correspond to a set of behaviors that operate on the member variables. To enforce encapsulation, the user usually has access only to the functions, whose implementation along with the member data is hidden. Figure 8 shows the definition of a C++ class, RigidBody, that is used in DynaMechs. Its member variables include all of its mass and inertia properties, and hydrodynamic coefficients like the drag coefficients and the added mass matrix. Note that Cartesian and spatial vector and matrix types have been defined (using typedef) to clarify the declarations of these variables. Member functions include the constructor (given the same name as the class) and an implementation of the drag force computation.

The C++ keyword, protected, indicates the section containing the variables and functions that cannot be accessed by the user. The public section contains the user accessible variables and functions. Normally, all variables and some functions are encapsulated, and it is through calls to the public functions that the variables are manipulated. This mechanism reinforces the concept of encapsulation which is the key to achieving the benefits of complexity management with OOD. Given the public interface, each object has a certain functionality, and the emphasis in the top-down design phase focuses on the interaction of objects using only these functions. At the same time, it decouples the implementation of the classes’ internal data and methods (a bottom-up process) from the top-down design decisions. In addition, should the implementation of the functions and variables change at a later date, users would not be required to change their code provided the interface (function names and their argument lists) does not change. This emphasizes the need for early and complete specification of the classes’ public interfaces during the design process, and will
class RigidBody
{
protected:
    float mass;        // position of center of gravity
    CartesianVector cgPos;   // 3x3 inertia tensor about body axes
    CartesianTensor Ibar;   // position of center of buoyancy
    SpatialTensor SpatialInertia;
    float Cd, length, radius;   // drag parameters.
    float dispFluidVol;        // volume of fluid displaced by body
    CartesianVector cbPos;      // position of center of buoyancy
    SpatialTensor AddedMass;   // 6x6 added mass matrix

public:
    RigidBody(FILE *cfg_ptr);   // Constructor
    void Compute_Drag(SpatialVector vrel, SpatialVector fd);
};

Figure 8: RigidBody class.

class Link: public RigidBody
{
protected:
    SpatialTransform X;        // Spatial tx matrix from inboard link

public:
    Link(FILE *cfg_ptr);       // Link-to-link spatial transforms
    void tx_To_Inboard(SpatialVector f, SpatialVector fib);
    void tx_From_Inboard(SpatialVector fib, SpatialVector f);
};

Figure 9: Link class.

significantly reduce the complexity of the programming task.

2. Inheritance

    Another goal in the design of the classes is to increase code reuse through a mechanism
called inheritance. This is accomplished by moving variables and methods that are needed
by more than one class into a single, more general class. The new class is called a superclass
and its member functions and variables are inherited by its subclasses. Since links are one
example of rigid bodies in robotic systems, the RigidBody class is an example of a superclass
that must be inherited by the Link class as shown in the first line of Figure 9. With the
public keyword, a Link object will then provide access to the the drag force computation
for the link as well as the transformation of spatial vectors to and from the coordinate
system of the link's inboard neighbor.

    Other examples of inheritance in this design can be found in the reference member
classes. Fixed and mobile base objects share common attributes that are defined in the `RefMember` superclass, and both `FixedBase` and `MobileBase` classes will inherit this class.

The `MobileBase` class must also inherit the `RigidBody` class which results in multiple inheritance and is implemented by adding more than one class name in the first line of the class definition. Another form of multiple inheritance can also be achieved by inheriting classes that are themselves subclasses. Examples of this are shown in Figure 10 with the further refinement of the `Link` class below.

The subclass/superclass terminology implies a hierarchical relationship between classes and is one motivation behind the development of the class hierarchy. The key to successful decomposition is to move attributes to the most general level possible. As a result, the amount of code reuse is maximized because the inherited variables and functions are only implemented once. Productivity increases because this also reduces the amount of coding and testing that is required.

3. Virtual Functions

`Polymorphism` is another important consideration in the development of the class hierarchy, which is defined as the ability to refer to objects of different classes by a common function name, and through this name have them “respond to a common set of operations in different ways” (Booch, 1986). To illustrate this, the spatial transformation across links paired with different types of joints is used. The first step is to expand the class hierarchy related to the `Link` class to include subclasses that are specialized for different joint types. The resulting classes are illustrated in the top of Figure 10. Note that single DOF (revolute and prismatic) joints use MDH parameters as defined in the `MDHLink` class, and ball-in-socket joints use Euler angles and a position vector as shown in the `BallnSocketLink` class. Both require different functions to achieve an efficient implementation of the spatial transformation function, `tx_to_inboard`. To hide these details, a uniform function interface is provided by the `Link` superclass. In C++, this is accomplished with the `virtual` keyword. The `= 0` indicates that this is a `pure virtual function` which is to be defined by its subclasses. Because it is undefined, the `Link` class is called an `abstract` class.

The code at the bottom of Figure 10 shows how polymorphism is used in C++. In this example, an array of two pointers to `Link` objects (line 1) will provide the common name through which the spatial transformation function is called. Then, `MDHLink` and `BallnSocketLink` objects are dynamically allocated with C++’s `new` command (Lines 4 and 5). With the common name (`link`) and the uniform function interface (`tx_to_inboard`) polymorphism can now be used. This is illustrated with the spatial transformation of `f[2]` across both links in a backward recursion (Line 8). Regardless of the actual link type, the function call is the same. Internally, however, `BallnSocket`’s function was executed for `link[1]` and `MDHLink`’s function for `link[0]`. This is also called `dynamic binding` where the particular function to be executed is determined at runtime (lines 4 and 5). With polymorphism no case statements or conditionals are needed. Other than the instantiation of the link objects themselves, this portion of the code does not require knowledge about the various types of links that may be defined, and it also does not need to be modified when new link subclasses are added to the system. In this respect, polymorphism also supports
class Link: public RigidBody
{
public:
    Link(FILE *cfg_ptr);
    virtual void tx_to_Inboard(SpatialVector f, SpatialVector fib) = 0;
};

class MDHLink: public Link
{
protected:
    float aMDH, alphaMDH, dMDH, thetaMDH; // Modified DH parameters
public:
    MDHLink(FILE *cfg_ptr);
    void tx_to_Inboard(SpatialVector f, SpatialVector fib);
};

class BallnSocketLink: public Link
{
protected:
    EulerAngless q; // orientation wrt inboard link
    CartesianVector p; // position wrt inboard link
public:
    BallnSocketLink(FILE *cfg_ptr);
    void tx_to_Inboard(SpatialVector f, SpatialVector fib);
};

... 
1    Link *link[2];
2    SpatialVector f[3];
3    link[0] = new MDHLink(cfg_ptr);
4    link[1] = new BallnSocketLink(cfg_ptr);
5    ...
6    for (i=2; i>0; i--)
7    {
8        link[i-1]->tx_to_Inboard(f[i], f[i-1]);
9    }

Figure 10: Polymorphism in the link classes.

encapsulation.

4. The Hierarchy

The mechanisms described above lead to the class hierarchy for DynaMechs shown in Figure 11. The abstract classes are indicated by the dashed boxes. As indicated by the object hierarchy (Fig. 7), the System class’s member variables include a pointer to a
RefMember object and an array of pointers to SerialChain objects. The array of pointers to the SerialChain objects is dynamically allocated when a System object is instantiated. This allows for any number of serial chains attached to the reference member to be simulated. The RefMember class contains pure virtual functions to provide System objects with a uniform interface for functions whose implementation depends on whether the reference member is a FixedBase or MobileBase object.

The SerialChain class's member variables include an array of pointers to Link objects and a pointer to an EndEffector object. The array is also dynamically allocated when a SerialChain object is instantiated, so that an arbitrary number of links can be simulated by this software. RevoluteLink and PrismaticLink classes have also been added which contain functions specific to each single DOF joint. As a result the MDHLink class will contain virtual functions to provide the uniform interfaces to these two subclasses. Note that virtual Link functions can also be virtual functions in the MDHLink class that are finally defined in the RevoluteLink and PrismaticLink classes. In this way, the uniform interface can still be maintained at the Link class level.

C. Using DynaMechs

Figure 12 lists code that can be used with DynaMechs to perform the dynamic simulation of a URV system. The list of variables required by the algorithm (lines 1-12) include the integration stepsize, reference member state (position and velocity), acceleration, and a spatial force vector which would correspond to resultant forces exerted by thrusters. Dynamically allocated arrays of variables are also needed for the joint state, accelerations, and generalized input forces. With these defined, the System object (urv) is instantiated (line 15). The arguments include a pointer to the file containing all of the necessary kinematic, dynamic and hydrodynamic parameters for the entire URV system, and the state, acceleration, and input force variables.
float idt; // integration stepsize
EulerAngles refPose; // state variables
CartesianVector refPos;
SpatialVector refVel;
float ***jointPos, ***jointVel;
SpatialVector refAcc; // accelerations
float ***jointAcc;
SpatialVector refForce; // input forces
float ***jointForce;

// a urv system.
System urv(cfg_ptr, refPose, refPos, refVel,
    &jointPos, &jointVel, &jointAcc, &jointForce);

while (TRUE)
{
    some_control_algorithm(..., refForce, jointForce);
    urv.AB_Dynamics(refForce, jointForce);
    get_stepsize(&idt);
    urv.Update_State(idt, refPose, refPos, refVel, refAcc,
        jointPos, jointVel, jointAcc);
    urv.draw();
}

Figure 12: Using DynaMechs.

The rest of the initialization process is hidden in which the System object initializes its component objects, an instance of the RefMember class and an array of SerialChain instances. This is accomplished by passing the file pointer when the constructors for these objects are called. In turn, SerialChain objects initialize their Link and EndEffector objects. In the same step, the state variables that are passed to the System constructor are then passed along to the appropriate constructors of component objects which will set these to the initial conditions specified in the input file. Since the number of serial chains and the number of links in each chain are dynamically allocated at runtime, jointPos, jointVel, jointAcc, jointForce are also dynamically allocated during this initialization. The resulting variables behave as three dimensional arrays whose indices are specified by the chain number, the link number within the chain, and the particular DOF of the joint. Once allocated, the System object receives input and sends output through these variables.

The code for one iteration of the simulation algorithm is listed in lines 20–26. A control algorithm must be provided (Line 20) which sets the input forces to the desired values. Then, these are passed to the AB_Dynamics member function of the System class (Line 21) to compute the dynamics of the system. The implementation of the algorithm is also hidden from the user, but is distributed in a manner similar to the initialization of the system and illustrated in Figure 13. In this figure, each box corresponds to one of the objects in the system and contains their class names. The arrows indicate the order in which the objects
are called in order to complete the three recursions of the AB algorithm. In addition, the `EndEffector` objects contain code for modeling compliant contacts and computing external tip forces as described in (Freeman and Orin, 1991).

The accelerations that are computed remain stored in member variables of the respective objects, and a numerical integration routine is required to update the state of the system. This example supports real-time simulation where the system clock could be used to determine \( \text{idt} \) (line 23). Then, Euler integration is used by calling the `System` class’s `Update_State` function (Line 24) which returns the updated state.

The runtime performance of the simulation algorithm (excluding rendering) has been measured on a Silicon Graphics (SGI) workstation with a 150MHz MIPS R4400 processor. The ROV system requires about 0.7 ms on the average to perform one iteration of the simulation which should be adequate for real-time performance during normal operation of the ROV without contact with the environment. One iteration of the *Aquarobot* simulation requires about 2.6 ms. In either case and depending on the stiffness of the compliant contacts with the environment, this may or may not be adequate for real-time performance, and this requires further study.

A third member function of the `System` class graphically renders the resulting system (Line 26). To date SGI’s Inventor and Performer packages, and NPS’s Graphics Description Language (GDL2) (Wilson *et al.*, 1992) have been implemented for this purpose. Figure 14 shows the *Aquarobot* which was rendered on an Indigo\(^2\) workstation with Extreme graphics using GDL2. Figure 15 shows *Tiburon* which was rendered using Inventor.
Figure 14: Scene from the Aquarobot simulation.

Figure 15: Scene from the Tiburon simulation.
5. Summary and Conclusions

In this chapter, the development of a dynamic and hydrodynamic simulation algorithm for URV systems has been reviewed. The algorithm is based on the Articulated Body (AB) method (Featherstone, 1983), which is a very efficient method for simulating land-based systems. To extend this to URV systems, incorporation of a number of hydrodynamic forces has been accomplished. Effects due to added mass, drag, buoyancy, and fluid acceleration have been examined. The resulting algorithm has $O(mN)$ complexity where $m$ is the number of serial chains in the system and $N$ is the number of links in each chain, and the addition of hydrodynamic effects nearly doubles the computational requirements over simulations of equivalent land-based systems.

Then, the development of an object oriented software package, called DynaMechs, for real-time graphical simulation has been described. A primary goal of this endeavor was to investigate OOD techniques to implement the algorithm in C++ for a large class of tree-structured mechanisms (having star topologies) ranging from single chain systems with fixed bases to multiple chain systems with mobile bases, containing revolute, prismatic and ball-in-socket joints. This programming problem was decomposed into smaller, decoupled tasks with the development of object and class hierarchies. Through techniques such as inheritance, encapsulation, and polymorphism, the overall complexity of the problem is significantly reduced because interactions between code segments are limited and tightly controlled. This also significantly reduces the amount of time taken to maintain and modify such code. The implementation of the algorithm is very efficient, and runtime performance is very close to or exceeds real-time rates. Work is ongoing at the Naval Postgraduate School to achieve improved performance with efficient graphical interfaces and through parallelization of the dynamics computation as in (McMillan et al., 1994).

Acknowledgements

The authors would like to thank Dr. Anthony Healey of the Naval Postgraduate School for informative discussions on hydrodynamics, and James B. Newman of MBARI for access to and involvement in the Tiburon project. This work was supported in part by a DuPont Fellowship and an AT&T Ph.D. Scholarship both at The Ohio State University, by Grant No. BCS-9109989 from the National Science Foundation to the Naval Postgraduate School, and by Grant No. BCS-9311269 from the National Science Foundation to The Ohio State University.

References


