1996

A Computational Framework for Simulation of Underwater Robotic Vehicle Systems

McMillian, Scott

Autonomous Robots 3, 253–268 (1996)
© 1996 Kluwer Academic Publishers. Manufactured in The Netherlands.

A Computational Framework for Simulation of Underwater Robotic Vehicle Systems

SCOTT McMILLAN
High Techsplanations, Inc., 6001 Montrose Road, Suite 902, Rockville, MD 20852-4874
scott@ht.com

DAVID E. ORIN
Department of Electrical Engineering, The Ohio State University, Columbus, OH 43210
orin@ee.eng.ohio-state.edu

ROBERT B. McGHEE
Department of Computer Science, Naval Postgraduate School, Monterey, CA 93943
mcghee@cs.nps.navy.mil

Abstract. This paper presents a computational framework for efficiently simulating the dynamics and hydrodynamics of Underwater Robotic Vehicle (URV) systems. Through the use of object-oriented mechanisms, a very general yet efficient version of the Articulated-Body (AB) algorithm has been implemented. An efficient solution to branching within chains is developed in the paper so that the algorithm can be used to compute the dynamics for the entire class of open-chain, tree-structured mechanisms. By including compliant contacts with the environment, most closed-chain systems can also be modeled. URV systems with an extended set of topologies can be simulated including proposed underwater walking machines with intra-body powered articulations. Using the encapsulation inherent in C++, the hydrodynamics code has been confined to a single class, thereby explicitly defining this framework and providing an environment for readily implementing desired hydrodynamics algorithms. Resulting simulations are very efficient and can be used in a number of applications both in the development and use of URV systems.

Keywords: underwater robotics, hydrodynamics, simulation, articulated mechanisms, tree topologies, object-oriented design

1. Introduction

Underwater Robotic Vehicle (URV) systems represent an important and growing area of advanced robotics because of their potential for applications in the hazardous and unstructured subsea environment. The range of applications is quite wide and is typified by those presented at a recent workshop in Japan on Robotic Technologies in Oceanic Engineering (Yuh, 1995b). These include installation, maintenance, and repair of subsea structures in support of the offshore-oil industry (Lane, 1995), survey of the deep sea floor to serve the needs of marine biologists and geologists who study hydrothermal vents (Yoerger et al., 1995), and inspection of underwater telecommunications cables (Kato, 1995). The current state-of-the-art and future research directions in underwater robotics are briefly summarized by Yuh (1995a), and representative research of a number of international groups may be found in papers from a recent conference (Basu et al., 1995; Sayers et al., 1995; Bono et al., 1995; Santos et al., 1995; Yuh, 1995a; Fujii and Ura, 1995; Vagany and Rigaud, 1995; Lane and Knightbridge, 1995).

Because of the relatively high cost and risk involved, simulation is an important alternative to sea trials and demonstrations especially during the early stages of the control development of a URV system. Numerical simulations with reasonable execution speeds can permit trials to be repeated at much more frequent rates. Beyond this are the computational requirements of telepresence applications in which a remote operator needs immediate graphical feedback to gain a sense of being present at the site (Boman, 1995). Real-time simulation is required to overcome the transmission delays and communications bandwidth limitations that are often associated with these systems.

As the degree of autonomy of URV systems is increased to reduce the operator workload or to extend the range of operation of the system, even greater demands on the computational rate may be placed during tele-assistance. The URV system automatically performs most of the lower-level coordination and control functions while the operator supervises the execution of the task by, for instance, teaching a series of actions or assisting with error recovery. Simulation-in-the-loop control can be effective in these modes of operation to provide a predictive capability. The effects of the present or an alternative guidance and control strategy can be simulated ahead of real time and viewed to judge its viability for successfully completing the task or mission. This may often imply computation of seconds of motion in milliseconds at super-real-time rates (Wong and Orin, 1995).

In order to meet the real-time or even super-real-time simulation requirements of some applications, this paper develops a computational framework for simulation of URV systems. This computational framework builds on the most efficient methods available for dynamic and hydrodynamic simulation. The articulated-body (AB) method for a single serial chain was first developed in Featherstone (1983). This was extended to URV systems systems (still single chain) with the incorporation of various hydrodynamic forces exerted on these systems in underwater environments in our previous work (McMillan et al., 1995b). While particular models were developed in that paper to compute these forces due to added mass, viscous drag, fluid acceleration, and buoyancy, many other models exist that make tradeoffs between computational simplicity and simulation fidelity (Yuh, 1990; Ioi and Itoh, 1990; Fossen, 1995; Goheen, 1995; Bono et al., 1995).

This work is extended in this paper with the development of a new algorithm that is capable of handling the branching of tree structures so that URV systems with general topologies may be efficiently simulated. This paper also presents the computational framework that is necessary to achieve a very efficient simulation of this general class of systems. The result is the ability to simulate, for example, dual redundant manipulators mounted on a free-flying base such as that described in Lane (1995) or Aquarobot, an underwater walking robot for surveying and construction of seawalls (Iwasaki et al., 1987). Terrain adaptive vehicles such as K^2T's eight-legged walker or the Wheeled Actively Articulated Vehicle (WAAV) (Yu and Waldron, 1991), which have intra-body powered articulations, can also be simulated.

Furthermore, the computational framework has been developed so that a variety of joint types, actuator models, and hydrodynamic models may be incorporated with ease. The degrees of freedom in each joint, the actuator forces/torques, and hydrodynamic forces, along with the link velocities, accelerations, etc., on which they are dependent, are effectively integrated into the recursions of the AB method so that the computational complexity is still $O(N)$, where N is the number of links in the system.

Finally, the computational framework that has been developed uses object-oriented design (OOD) techniques. Implementation of real-time simulation software can be a complex programming task. This paper presents the OOD techniques that are especially suited for managing and reducing this complexity such as *information hiding* or *encapsulation*, *inheritance*, and *polymorphism*. These mechanisms also support code reuse and extensibility so that productivity is increased and the simulation code can easily be adapted to model new mechanisms.

In the next section, the kinematic topologies of the URV systems considered are discussed in more detail. Considering the most general tree-structured topology, the development of the AB algorithm is presented in Section 3. Then, the computational framework to support the efficient incorporation of the hydrodynamic effects is summarized in Section 4. Section 5 presents the object-oriented implementation of the resulting algorithm which is followed by some results.

2. Topologies of URV Systems

With advances in underwater robotics, the kinematic topologies found in URV systems are becoming much more diverse. The simplest is the sub-like URV without

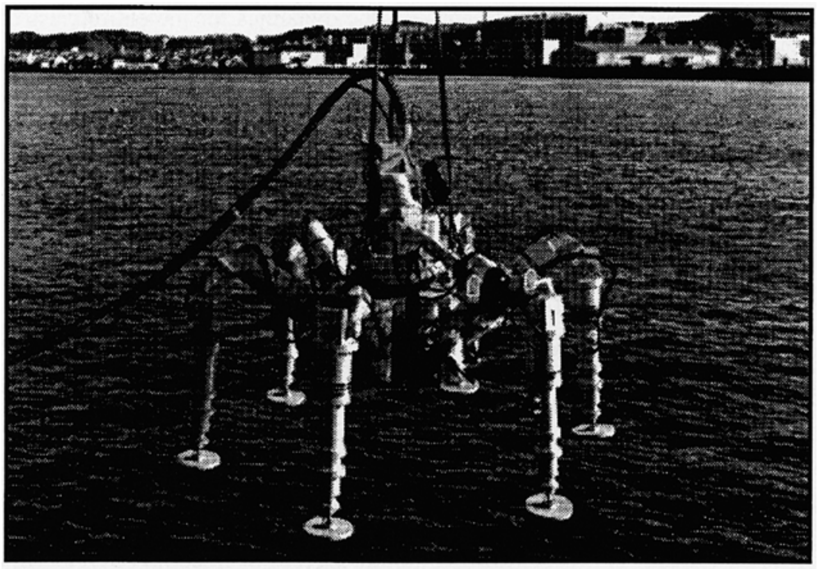


Figure 1. Aquarobot: an underwater walking robot for surveying and construction of seawalls (Iwasaki et al., 1987).

articulations. This represents the majority of URV systems and can be modeled as a single rigid body. To interact with the environment single manipulators have appeared on some systems such as MBARI's ROV, *Tiburon*, equipped with a Schilling manipulator (Newman and Robison, 1992). The corresponding topology is a simple articulated structure called a *serial chain*.

More recently, systems with greater complexity have been developed which require corresponding increases in the complexity of simulation and control systems. Examples include URVs with multiple manipulators such as the dual redundant manipulator system described in Lane (1995), and underwater legged systems such as the Aquarobot (Iwasaki et al., 1987) shown in Fig. 1. These systems are said to have a more general star topology which consists of a single central body, called the reference member, to which any number of serial chains is attached (one of these chains is shown in Fig. 3(a)).

However, some of the most advanced robotic systems being designed have more general topologies still. Examples of these systems include the energy-efficient "frame walkers" such as Dante II (Apostolopoulos and Bares, 1995), a rappelling vehicle for exploration of volcanoes. Its torso consists of two bodies that slide back and forth and rotate with respect to one another, with a set of four legs attached to each that can be lowered and raised for the support and transfer phases, respectively. Another is the WAAV (Wheeled Actively Articulated Vehicle), a terrain-adaptive vehicle for planetary exploration, containing three modules with two independently powered wheels each, and interconnected by powered ball joints (Yu and Waldron, 1991).

These design approaches are now being applied to URV systems. As shown in the concept drawing in Fig. 2, the topology for K^2T's autonomous robot for shallow water and surf-zone mine clearance operations is more complex than a simple star topology because the body of the vehicle contains a joint that allows the top half to rotate about a vertical axis with respect

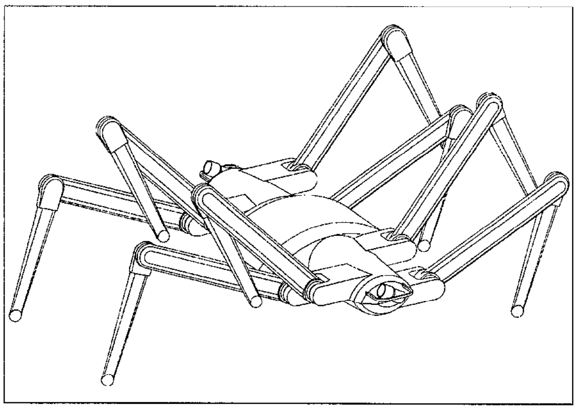


Figure 2. Concept drawing of an autonomous mobile robot under development for shallow water and surf-zone mine clearance applications (Courtesy of K^2T, Inc.).

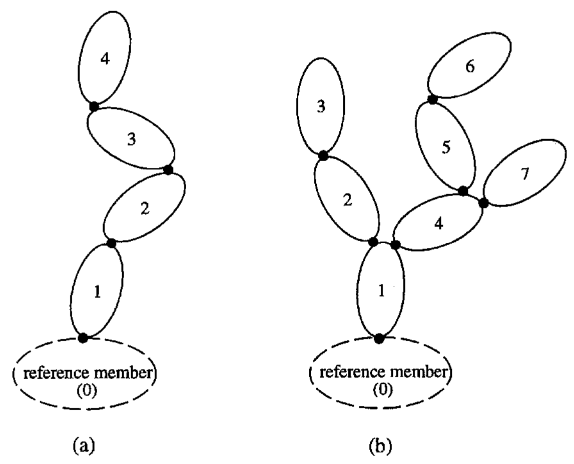


Figure 3. Open-chain articulations connected to a reference member: (a) a serial chain, and (b) a (general) tree structure.

to the bottom half. Similar to Dante II, four legs are attached to each half which alternate in the support and transfer phases.

All of these systems are characterized by intra-body, powered articulations within the torso that cannot be modeled as simple star topologies. Instead, the most general open-chain topology, the tree, must be employed and is illustrated in Fig. 3(b). The intra-body articulations can be modeled by allowing for branching in the chains that are attached to the reference member. This same branching capability allows for modeling of open-chain systems with arbitrary branching elsewhere in the system as well. To this point, our **DynaMechs** software package was capable of simulating only systems with star topologies (McMillan et al., 1995a). This paper presents the extensions that allow the entire class of open-chain articulated mechanisms with tree structures to be modeled. Also, by modeling the compliance at the contacts where these mechanisms interact with the terrain/environment, the closed loops that occur within time-varying topological structures can be handled as well. The required simulation algorithm for tree-structured topologies with contacts at the "leaf" nodes is presented in the next section.

3. The Articulated-Body Algorithm for Tree Structures

In this section, a very general yet efficient version of the Articulated-Body (AB) algorithm is presented. An efficient solution to branching within chains is developed so that the algorithm can be used to compute the dynamics for the entire class of open-chain, tree-structured mechanisms. It builds upon the structure of the AB algorithm (Featherstone, 1983) for computing robot dynamics, that is presented in the appendix, to provide a basis for the computational framework to simulate URV systems. This is a very efficient method for computing the dynamics of multibody systems (Roberson and Schwertassek, 1988).

In this discussion, a general tree structure is assumed as shown in Fig. 3(b) which has a reference member and N links that are numbered from 1, attached to the reference member through joint 1, to N, one of the "leaf" links (end-effectors). With this algorithm, the reference member can be either mobile or fixed, for simulation of AUV/ROV systems as well as manipulation systems mounted on stationary platforms, and the joints between the links can have an arbitrary number of degrees of freedom (DOFs). In addition, soft constraints are used to model the contact of chains (specifically, the end-effectors) with the terrain/environment (Freeman and Orin, 1991). This means that the closed loops that are thus formed through the time-varying contacts can be modeled as well. In fact, soft constraints are appropriate in most cases since they model the compliance at the contact.

The method for modeling the branches in general tree structures is presented in the following discussion and is based on the work of Khalil and Kleinfinger (1986). For serial chains with single DOF joints, modified Denavit-Hartenberg (MDH) parameters specify the kinematic transformations between successive links most efficiently (Craig 1986). However, they are inadequate in tree-topology systems where branching occurs. In this case, a pair of constant parameters, in addition to the MDH parameters, are needed to specify the transformation between the links as illustrated in Fig. 4.

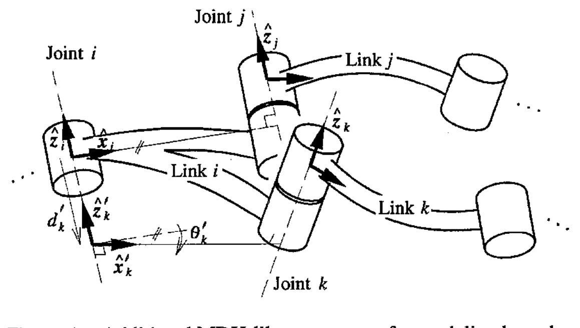


Figure 4. Additional MDH-like parameters for modeling branches in tree structures with 1 DOF joints.

This figure shows a branch that occurs outboard of link i to links j and k. Again, assuming single DOF joints, link i's coordinate system (indicated by $\hat{z}_i$ and $\hat{x}_i$) can be defined so that $\hat{x}_i$ lies on the common normal between joint i's axis and *one* of the outboard joint axes (joint j in this case); therefore, a set of MDH parameters can specify this transformation. For each additional branch, an intermediate coordinate system (indicated by $\hat{z}'_k$ and $\hat{x}'_k$ in the figure) attached to link i is specified from which MDH parameters can then be used. This intermediate coordinate system also lies on joint i's axis and is specified with an axial screw transformation with constant parameters, d'_k and θ'_k.

Since transformation of kinematic and dynamic quantities between coordinate systems accounts for the majority of the computation in the simulation of articulated-body mechanisms, efficient implementation of these functions is paramount to achieving efficient simulation algorithms. Our previous work showed how the various dynamic quantities are most efficiently transformed using MDH parameters (McMillan and Orin, 1995). By realizing that these operations consist of two successive axial (x and z) screws and that the additional parameters for branching define an additional z-axial screw, the result in that paper can be applied to the additional parameters to achieve the most efficient computation for these general tree structures.

The previous discussion assumed revolute and prismatic joints only. Other joint types can be modeled using a number of single DOF joints. However, it has been shown that more efficient methods are available for specifying the transformation across multiple DOF joints when parameters suitable for the joint are used. One example from our work is the ball joint which contains three revolute DOFs. With or without branching, a constant position vector and three variable Euler angles are sufficient and no intermediate coordinate system would need to be specified.

Finally, note that in tree-structured systems each link has a single parent link, also called its inboard neighbor. Therefore, it is notationally easier to present unambiguous kinematics and dynamics algorithms that refer only to the current link and its inboard neighbor. To accomplish this, the predecessor function, $p(\cdot)$, is used extensively in the rest of this paper to refer to a link's predecessor. As an example, $p(j) = p(k) = i$ for the links in Fig. 4.

Previous work (McMillan et al., 1995a, b) presented the kinematics and dynamics equations required to simulate a serial chain system. The links were numbered, in order, from 1 (attached to the reference member) to the last link, N, as shown in Fig. 3(a). The implementation of the AB algorithm's recursions along the serial chains was implemented in a straightforward fashion with `for` loops, using the loop variable to index the necessary functions for the links in the correct order. An increasing counter is used for the forward recursions and a decreasing one for the backward recursion.

Using this predecessor function, the new kinematic and dynamic equations used to derive the AB algorithm for tree-structured topologies has been included in the appendix. For the kinematic equations, it is a straightforward conversion from serial chain code to replace all references to link $i - 1$ (the inboard neighbor to link i in a serial chain) with $p(i)$, because all quantities computed in the kinematics step refer only to the inboard neighbor. On the other hand, the force balance (dynamics) equation for a link requires the sum of forces exerted on its outboard neighbors which requires a new approach. Since the outboard links equations would be computed first, an *accumulation* operation must be implemented in the new computational framework which adds to the forces exerted on a link $p(i)$ and the code for each link i is computed in the backward recursion. The equation from the appendix is repeated here for emphasis:

$$\mathbf{f}^{+}_{p(i)} := \mathbf{f}^{+}_{p(i)} + {}^{i}\mathbf{X}^{T}_{p(i)}\, \mathbf{f}_i \quad \forall i \text{ given } p(i), \tag{1}$$

where $\mathbf{f}^{+}_{p(i)}$ is the sum of the forces exerted by link $p(i)$ onto all of its outboard (successor) links and $\mathbf{f}_i$ is the part of that force exerted onto link i.

With the proper computational framework presented below, the serial chain method can be easily modified to support the simulation of tree-structured systems by employing a specific link numbering scheme suggested by Brandl et al. (1986). That is, the links are still numbered from 1 to N with the requirement that a link's number always be greater than its predecessor. Either depth-first or breadth-first numbering of the links will satisfy this requirement, and the former is used in Fig. 3(b). This ensures that `for` loops can still be used and that the computations proceed in the proper order with an inboard link's computation occurring before/after an outboard link's computation during a forward/backward recursion. The only major differences between general tree structures and serial chains are that more end-effectors are possible, additional transformation parameters may be needed for branching in the tree, and the accumulation of forces and AB inertias at branch points with equations similar to 1 must be performed.

4. Hydrodynamic Simulation Framework

With an efficient AB algorithm for land/space-based robotic systems well understood, this section presents the computational framework for developing an efficient hydrodynamic 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 identifiable 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 due to added mass, viscous drag, buoyancy, and fluid acceleration which are presented in the following discussion.

Inclusion of the hydrodynamic forces in the simulation is accomplished at the rigid body (link or mobile reference member) level where the force balance equation for each was defined in Eq. (A6) in the appendix. Including the hydrodynamic effects, this equation is replaced with the following:

$$\mathbf{f}_i = \mathbf{I}_i^H \mathbf{a}_i - \beta_i^H + \mathbf{f}_i^+, \qquad (2)$$

where

$$\mathbf{I}_i^H = \mathbf{I}_i + \mathbf{I}_i^A, \qquad (3)$$

$$\beta_i^H = \beta_i + \beta_i^A + \mathbf{f}_i^D + \mathbf{f}_i^B + \mathbf{f}_i^F, \qquad (4)$$

and which are called the *hydrodynamic inertia* and *hydrodynamic bias force*, respectively. These are analogous to the spatial inertia of the link, $\mathbf{I}_i$, and bias force, β_i, that are used in land- and space-based simulation.

The new (hydrodynamic) inertia is needed when a rigid body accelerates through a fluid, because some of the surrounding fluid also accelerates with the body. For water, this fluid has significant inertia properties that can be specified with a 6×6 positive definite added mass matrix, $\mathbf{I}_i^A$. A detailed derivation of the spatial (6×1) reaction force, $\mathbf{f}_i^A$, due to this effect was carried out in McMillan et al. (1995). The resulting form has an acceleration-dependent term which leads to Eq. (3) above, and a bias force that is a function of the following terms:

$$\beta_i^A = f\left(\omega_i, v_i^r, {}^i\boldsymbol{a}_f\right), \qquad (5)$$

where v_i^r is the linear velocity of the body *relative* to the fluid and ${}^i\boldsymbol{a}_f$ is the linear acceleration of the fluid. In order to compute $\mathbf{f}_i^A$ in the local coordinates of the link for inclusion in the AB algorithm, the components of both, v_i^r and ${}^i\boldsymbol{a}_f$ must be determined relative to the body's local coordinate system. Also note that the fluid rotation is assumed to be negligible in this framework, in which case only the link's angular velocity is included.

When an object translates through a viscous fluid, lift and drag forces are also exerted on it and are included in the hydrodynamic bias force. For a general body, surface integrals over the entire body are required to compute the resultant force and moment, $\mathbf{f}_i^D$, which is a function of the body's velocity relative to the fluid:

$$\mathbf{f}_i^D = f\left(\omega_i, v_i^r\right). \qquad (6)$$

While lift and related forces due to vortex shedding are usually much smaller for the systems and applications under consideration here, large viscous drag forces can be exerted on URV systems even for reasonably slow motions because water density is significant. Drag can be further 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 in McMillan et al. (1995b) was on modeling pressure drag. It arises from non-zero normal components of relative velocity between the body's surface and the fluid. As was shown in McMillan et al. (1995b), this computation can be significantly simplified for regular body shapes such as cylinders.

Finally, both buoyancy and fluid acceleration forces are translational forces exerted through the center of buoyancy of the body and proportional to the mass of the fluid that is displaced by the body. The buoyant force is also proportional to gravitational acceleration, ${}^i\boldsymbol{a}_g$, and opposite in direction:

$$\mathbf{f}_i^B = f\left({}^i\boldsymbol{a}_g\right), \qquad (7)$$

while the fluid acceleration force is proportional to and exerted in the same direction as the acceleration of the surrounding fluid, ${}^i\boldsymbol{a}_f$:

$$\mathbf{f}_i^F = f\left({}^i\boldsymbol{a}_f\right). \qquad (8)$$

Note that these require the gravitational and fluid acceleration terms to be expressed in the body's local coordinates as well, hence, the leading superscript.

These two accelerations and the linear velocity relative to the fluid are additional terms that must be provided in the computational framework to support hydrodynamic simulation. The relative linear velocity is most efficiently incorporated by replacing the computation of the link's angular velocity in Eq. (A1) in the appendix with the following for the spatial velocity of the link relative to the fluid:

$$v_i^r = \begin{bmatrix} \omega_i \\ v_i^r \end{bmatrix} = {}^i\mathbf{X}_{p(i)}\, v_{p(i)}^r + \phi_i\, \dot{q}_i. \tag{9}$$

Note that the top half of this spatial vector is the angular velocity from before because we assume irrotational fluid flow, and the last term has been generalized for arbitrary joint types whose motion space is defined by the $6 \times n$ vector, ϕ_i (Lilly and Orin, 1991). Since the gravitational and fluid acceleration are free vectors, they can be expressed with respect to each coordinate system with the appropriate rotation operation:

$${}^i\boldsymbol{a}_{f,g} = {}^i\mathbf{R}_{p(i)}\, {}^{p(i)}\boldsymbol{a}_{f,g}. \tag{10}$$

These equations are incorporated into the Forward Kinematics recursion of the AB algorithm along with the subsequent computation of β_i^H for each rigid body in the system.

Although detailed derivations of equations to compute the additional hydrodynamic forces were presented in McMillan et al. (1995b), the specifics have been purposefully omitted from this discussion. This is to emphasize the fact that, with the framework provided here, different hydrodynamic algorithms (Yuh, 1990; Ioi and Itoh, 1990; Fossen, 1995; Goheen, 1995; Bono et al., 1995) with varying degrees of accuracy or modeling additional effects can be readily implemented.

5. An Object-Oriented Robot Simulation System

The resulting AB algorithm has been implemented in C++ using object-oriented design (OOD) techniques as part of a more extensive goal to develop a real-time graphical simulation system for underwater, land, and space-based robotic systems. Among the advantages of OOD are improved complexity management, code reuse and increased productivity, and easier maintenance and expandability[1]. This section of the paper presents an overview of the design of this simulation system, called **DynaMechs** (McMillan et al., 1995a), with emphasis on the use of the OOD mechanisms of

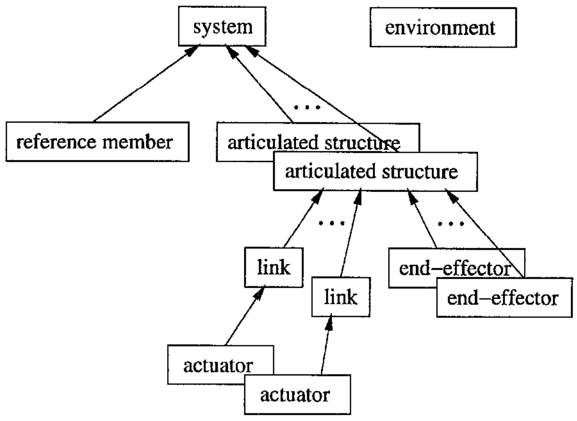


Figure 5. Object hierarchy for **DynaMechs**.

encapsulation, *inheritance*, and *polymorphism* to accomplish this goal.

5.1. *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 Fig. 5. 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 work is on the system object. The system is decomposed into the reference member and a number of general articulated structures that are attached to the reference member. The articulated structure is further decomposed into a number of links and a set of end-effectors that are associated with the "leaf" links of the articulated structure and allow for interaction with the environment. Finally, actuator models have also been added to this version of the software and are considered a component of each link.

5.2. *Class Hierarchy*

The second step in OOD concentrates on the development of the classes to be used. Classes essentially define the "types" of the elements that appear at the nodes of the object hierarchy. This is accomplished with a *kind-of* classification of these objects defined by inheritance and polymorphism techniques which leads to the class hierarchy as shown in Fig. 6.

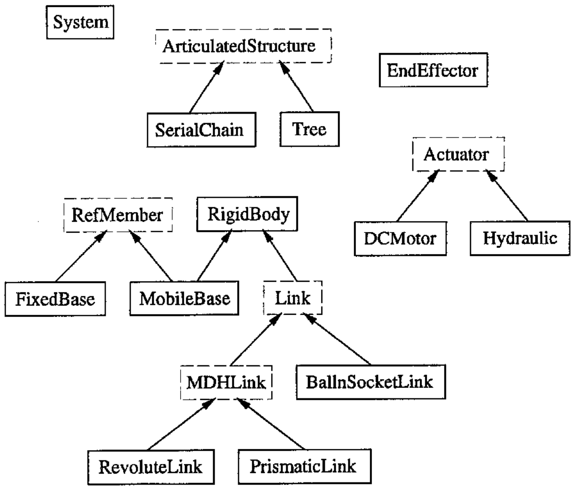


Figure 6. Class hierarchies for **DynaMechs** (the dashed boxes indicate abstract classes).

As implied by the object hierarchy, the `System`'s member variables include an instance of a `RefMember` object and an array of `ArticulatedStructure` objects that are dynamically allocated when the `System` object is created. This allows for simulation of an arbitrary number of articulated structures attached to the reference member. Polymorphism (see below) in the `RefMember` class, as indicated by the dashed box in the figure, provides `System` objects with a uniform interface for functions whose implementation depends on whether the reference member is a `FixedBase` or `MobileBase` object which enables simulation of systems with stationary platforms as well as vehicles. Likewise, polymorphism in the `ArticulatedStructure` class allows for simulation of general `Tree` structures as well as `SerialChain` objects. While serial chains can be represented as trees, code specialized to serial chains is more efficient than the general tree code and has been retained from the previous version of this software.

The `ArticulatedStructure` class member variables include dynamically allocated arrays of `EndEffector` and `Link` objects which allows for an arbitrary number of links and end-effectors to be simulated for each articulated structure. Polymorphism in the `Link` and `MDHLink` classes provides `ArticulatedStructure` objects with a uniform interface for functions whose implementation depends on the type of joint associated with the link: revolute (`RevoluteLink`), prismatic (`PrismaticLink`), or ball (`BallnSocketLink`) joints. Also note that inheritance (see below) is used so that both the `MobileBase` and `Link` classes can access the member data and functions associated with the general `RigidBody` class. The discussion that follows presents the pertinent OOD mechanisms that are used including encapsulation, inheritance, and polymorphism in the design of this hierarchy.

Encapsulation. A class has two distinct features: *member variables* that store data or state, and *member functions* (or *methods*) which correspond to a set of behaviors that operate on the member variables. Figure 7 shows excerpts from C++ classes that are used in **DynaMechs**. For example, the member variables of the `RigidBody` class (at the top of the figure) include all of its mass and inertia properties, and hydrodynamic parameters such as center of buoyancy, drag coefficients, and the added mass matrix. Note that user-defined Cartesian and spatial vector and matrix types have been used to clarify the declarations of these variables. In this example, member functions include the constructor (given the same name as the class) for creating instances of the class and code to compute the hydrodynamic bias force.

The user usually only has access to a set of member functions in the section of the definition indicated by the C++ keyword, `public`. Encapsulation is enforced in two ways. Not only is the implementation of these functions hidden from the user, but the variables and functions in the `private` or `protected` (a feature to support inheritance mechanisms) sections of the class are also inaccessible to the user. Normally, all member variables and some functions are encapsulated, so that limited and controlled access by the user can only be gained through calls to the `public` functions.

This encapsulation is key to achieving the benefits of complexity management with OOD. From the user's standpoint, the objects of each class have a certain functionality provided by the `public` functions, and the emphasis in the top-down design phase focuses on the interaction of objects using only these functions. At the same time, the implementation of the classes' internal data and methods (a bottom-up process) is decoupled from the top-down design decisions. Should the implementation of the functions and variables change at a later date, no other code would be affected 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,

```
class RigidBody
{
protected:
    float           mass;
    CartesianVector cg_pos;  // pos of c.g.
    CartesianVector cb_pos;  // pos of c.b.
    CartesianTensor I_bar;   // 3x3 inertia
    SpatialTensor Spatial_Inertia, Added_Mass;
    float C_d, length, radius; // drag params.
    float disp_fluid_vol;

public:
    RigidBody(FILE *cfg_ptr);  // Constructor
    void computeHydroBias(CartesianVector a_g,
                          CartesianVector a_f,
                          SpatialVector v_r,
                          SpatialVector betaH);
};
```

```
class Link: public RigidBody  // inherit R.B.
{
public:
    Link(FILE *cfg_ptr);
    virtual
      void txToInboard(SpatialVector f,
                       SpatialVector f_ib)=0;
};
```

```
class MDHLink: public Link    // inherit Link
{
protected:
    float d_branch, theta_branch;
    float a_MDH, alpha_MDH, d_MDH, theta_MDH;

public:
    MDHLink(FILE *cfg_ptr);
    void txToInboard(SpatialVector f,
                     SpatialVector f_ib);
};
```

```
class BallnSocketLink: public Link
{
private:
    EulerAngles q;      // orient. and pos.
    CartesianVector p;  // wrt inboard link.

public:
    BallnSocketLink(FILE *cfg_ptr);
    void txToInboard(SpatialVector f,
                     SpatialVector f_ib);
};
```

Figure 7. Excerpts from some **DynaMechs** classes.

and will significantly reduce the complexity of the programming task and make code maintenance and modifications much easier.

This point is extremely important in the context of our computational framework for supporting hydrodynamic simulation because nearly all of the code specific to the hydrodynamic effects have been encapsulated into the `RigidBody` class. The hydrodynamic inertia is computed in the constructor during initialization and assigned to the `Spatial_Inertia` variable which is accessed by the remaining code during the Backward Dynamics recursion, and the hydrodynamic bias force is computed by calling the `computeHydroBias` member function during the Forward Kinematics recursion. The remainder of the software provides the framework for simulating general, open-chain, tree-structured systems. Provided the arguments passed to the `computeHydroBias` member function are sufficient, the user can arbitrarily change its implementation to compute any desired hydrodynamic algorithm (even computing time-varying added mass matrices) without needing to modify any other classes.

Inheritance. A second goal in the design of the classes is to increase code reuse with a mechanism called *inheritance*. This is accomplished by moving variables and functions that are present in more than one class into a single, more general class, called a *superclass*. Its variables and functions are then inherited by *subclasses* that need to use them. For example, the `RigidBody` class is the superclass for the `Link` (shown in Fig. 7) and `MobileBase` (not shown) classes that are both examples of rigid bodies. The `public RigidBody` declaration in the first line of the `Link` class definition will give `Link` objects access to the `RigidBody`'s member variables and functions as well as the variables and functions specific to links such as the function, `txToInboard`, to transform spatial vectors to the coordinate system of the link's inboard neighbor.

Other examples of inheritance in this design can be found in the reference member classes (Fig. 6). `FixedBase` and `MobileBase` objects share common attributes that are defined in and inherited from the `RefMember` superclass. The `MobileBase` class also inherits the `RigidBody` class which results in *multiple inheritance*. Another form of multiple inheritance can also be achieved by inheriting classes that are themselves subclasses, such as the `MDHLink` and `BallnSocketLink` classes which contain attributes of the `Link` and `RigidBody` classes.

The subclass/superclass terminology implies the hierarchical relationship between classes and is one motivation behind the development of the class hierarchy. The key to successful class 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 reduces the amount of coding and testing required.

Polymorphism. *Polymorphism* is the other important consideration in the development of the class hierarchy, which is defined in Booch (1986) as the ability to refer to methods of different classes by a common function name, and through this name have them "respond to a common set of operations in different ways." To illustrate this point, the `txToInboard` is used as an example (Fig. 7). First, the class hierarchy related to the `Link` class is expanded to include the `MDHLink` and `BallnSocketLink` subclasses that contain code *specialized* to different types of joints.

For efficient transformations, the four modified Denavit-Hartenberg (MDH) parameters and two branch parameters (if applicable) are needed for single DOF (revolute and prismatic) joints as defined in the `MDHLink` class, and three Euler angles and a Cartesian position vector are used for the three DOF ball joints as shown in the `BallnSocketLink` class. As a result, each requires a different version of `txToInboard`. To hide these details from the user, a uniform function interface to both, using C++'s `virtual` keyword, is provided in the `Link` superclass. The "= 0" indicates that this is a *pure virtual function* which *must* be defined by its subclasses. Because it has undefined methods, the `Link` class is called an *abstract* class as indicated by the dashed box in Fig. 6.

With this framework, the following code shows how polymorphism is used in C++:

```
1 Link *link[2];
2 link[0] = new MDHLink(cfgptr);
3 link[1] = new
            BallnSocketLink (cfgptr);
                :
4 for (i=2; i>0; i--)
5   link[i-1]->txToInboard(f[i],
                           f[i-1]);
```

In this example, an array of two `Link` objects (line 1) is allocated. The first is dynamically allocated as an `MDHLink` and the second a `BallnSocketLink` using C++'s `new` command (lines 2–3). With the common name (`Link`) and the uniform function interface (`txToInboard`) polymorphism can now be used. This is illustrated with the spatial transformation of a spatial vector `f[2]`, expressed in `link[1]`'s coordinate system, to the coordinate system of the links' inboard neighbors in a backward recursion (lines 4–5). Note that this example assumes a serial chain for simplicity, in which case the predecessor function has simplified to $p(i) = i - 1$.

Regardless of the actual type of joint, the user's function call is the same. Internally, however, `BallnSocketLink`'s function was executed for `link[1]` and `MDHLink`'s function for `link[0]`. This mechanism is also called *dynamic binding* where the particular function to be executed is determined at runtime (when lines 2 and 3 are executed). With polymorphism no extra variables indicating which type of joint, case statements or conditionals are needed which keeps the code cleaner and more maintainable. Other than the instantiation of the link objects themselves, this code supports encapsulation because it does not depend on the various types of links that may be defined, and it also does not need to be modified when the system is extended to include new link subclasses.

Polymorphism and inheritance, together, in the `Link` class, illustrate the ease with which the software has been extended to include the new ball joint types. Figure 6 also indicates that polymorphism was extended to the `MDHLink` class, and inheritance from that class was used to create the `RevoluteLink` and `PrismaticLink` classes. Inheritance (code reuse of the `MDHLink` class) reduces the amount of new code that must be written while polymorphism through both the `Link` and `MDHLink` classes eliminates the need to modify code that calls methods of `Link` objects. This extensibility is a key requirement of this simulation package since it is targeted to experimental vehicle systems where a vast range of configurations may be proposed.

6. Hydrodynamic AB Algorithm Implementation

Using the design presented in the previous section, the hydrodynamic AB algorithm was implemented efficiently. The distribution of the tasks in the algorithm across the various classes is illustrated in Fig. 8. In this figure, each box corresponds to one of the objects in the system and contains their class names (more than one where polymorphism is illustrated). The arrows indicate the order in which the objects execute code to complete the three steps 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).

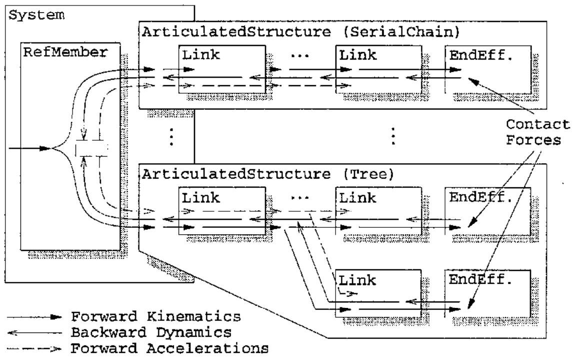


Figure 8. Hydrodynamic AB algorithm distribution among the **DynaMechs** classes.

Note that polymorphism in the `Articulated-Structure` class provides a uniform interface so the same algorithm can simulate both general tree topologies and the streamlined serial chain code, in the same way the `RefMember` class enables the software to easily simulate fixed and mobile base systems and the `Link` class to easily compute the dynamics for numerous types of joints. In addition, dynamic array allocation allows for simulation of an arbitrary number of articulated structures each with an arbitrary number of links.

Figure 9 shows the *Aquarobot* simulation running on a Silicon Graphics (SGI) Indigo^2 workstation with Extreme graphics using SGI's Performer software. The runtime performance of the resulting simulation algorithm for the Aquarobot has also been measured on this workstation containing a 150 MHz MIPS R4400 processor and the IRIX 5.2 operating system. Including foot pad and camera boom dynamics and the hydrodynamic algorithm described in McMillan et al. (1995b), one iteration of the Aquarobot dynamics computation (28 rigid bodies/45 DOFs) requires about 2.6 ms. With reasonably stiff contacts with the environment, this appears adequate for real-time performance, and is undergoing further study. In addition, much work is yet to be done to compare the simulation results with experimental results for Aquarobot. First, the derivation of the hydrodynamic parameters required by this simulator must be accomplished which is a difficult task and must be the subject of further study.

7. Conclusions

A computational framework for simulating the dynamics and hydrodynamics of URV systems has been developed in this paper. Systems with general tree-structured topologies may be simulated within this framework, and these include URVs equipped with multiple manipulators and underwater walking machines. The closed loops that occur when the arms or legs interact with the terrain/environment are effectively managed by using soft constraints which model the compliance at the contacts (Freeman and Orin, 1991). The dynamics algorithm used within the

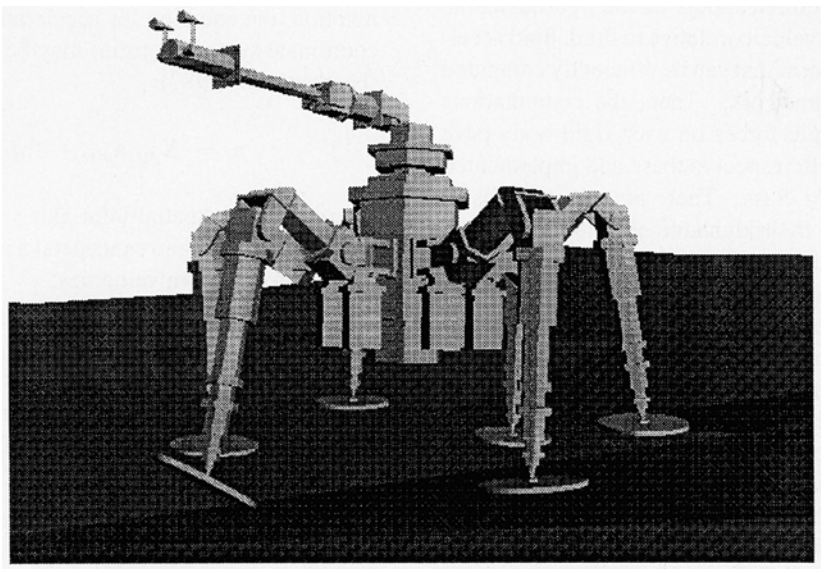


Figure 9. Scene from the *Aquarobot* simulation.

framework is based on the Articulated-Body (AB) method (Featherstone, 1983) which was extended to support the topologies of tree-structured systems Brandl et al. (1986) by providing an efficient solution to branching with chains. The hydrodynamic effects of added mass, drag, buoyancy, and fluid acceleration forces McMillan et al. (1995b) may be readily computed within the framework since the velocities, accelerations, and other quantities on which they are dependent, are provided. The resulting simulation algorithm has $O(N)$ complexity where N is the number of links in the system.

Object-oriented design techniques were used to support an efficient, yet general, implementation of the algorithm in C++. The capability exists to simulate systems with multiple articulated structures (with efficient code in place for the prevalent serial chain structure, as well) connected to a reference member that can be either fixed or mobile. The joints that connect the rigid bodies together can be revolute, prismatic, or ball joints, and the capability exists to easily extend the system to include other types.

One of the important results of the object-oriented design is the implementation of the hydrodynamic computation. While the rest of the system provides the framework for computing the dynamics of these general tree-structured mechanisms, the computation of the hydrodynamic forces on each rigid body in the system becomes encapsulated in a single (`RigidBody`) class. In fact, it can be implemented in a single member function of the class. This has been accomplished by defining the required inputs of the hydrodynamic computation (e.g., velocity relative to fluid, fluid acceleration, etc.) in a form that can be efficiently computed within the AB framework. Then, the computations of the hydrodynamic forces on each rigid body have been developed with respect to these and implemented in the `RigidBody` class. There are two benefits to this approach: (a) hydrodynamic algorithms of varying complexity and accuracy, that have been developed by others for submerged rigid bodies (there are many), can be readily implemented because the implementation itself is hidden from the other parts of the software system, and (b) the resulting software will yet be very efficient for complex structures.

In addition to the aspects of the computational framework that have been described in the paper, other work has proceeded to provide a graphical capability with the basic **DynaMechs** simulation system. This is the subject of continuing work that will hopefully advance the employment of simulation in the development and use of URV systems. Released versions of the code can be obtained via anonymous FTP from `ftp://cs.nps.navy.mil/pub/dynamechs` or visiting the **DynaMechs** Home Page on the WWW at `http://cs.nps.navy.mil/research/eod/dynamechs.html`.

Appendix: Articulated-Body Algorithm

In this appendix, the equations required to develop the AB algorithm for tree-structured topologies is discussed. In particular, the kinematic and dynamic equations that result from the use of the predecessor function that was defined in Section 3 are given.

A.1. Kinematics

Derivation of the AB algorithm begins with the set of kinematic equations that compute the velocities and accelerations for each link. For land- and space-based algorithms, only the angular velocity is needed which depends on the angular velocity of the inboard link, $\omega_{p(i)}$ and the velocity of the intervening joint (assuming a revolute joint), q_i, as follows:

$$\omega_i = {}^i\boldsymbol{R}_{p(i)}\omega_{p(i)} + \dot{q}_i\hat{z}_i, \qquad \text{(A1)}$$

where ${}^i\boldsymbol{R}_{p(i)}$ is the 3×3 rotation matrix between link i's coordinate system and its predecessor's. Using spatial notation, the equation for acceleration of link i (at its coordinate system's origin) may be written as follows (Featherstone, 1987):

$$\mathbf{a}_i = {}^i\mathbf{X}_{p(i)}\mathbf{a}_{p(i)} + \phi_i\ddot{q}_i + \zeta_i, \qquad \text{(A2)}$$

where ϕ_i is the spatial joint axis vector, and ζ_i is the vector of Coriolis and centripetal accelerations that are a function of known velocities:

$$\zeta_i = f(\omega_{p(i)}, \omega_i, \dot{q}_i). \qquad \text{(A3)}$$

The spatial acceleration is a six-element vector combining the three-dimensional angular, $\dot{\omega}_i$, and linear acceleration, $\boldsymbol{a}_i$, vectors as follows:

$$\mathbf{a}_i = \begin{bmatrix} \dot{\omega}_i \\ \boldsymbol{a}_i \end{bmatrix}. \qquad \text{(A4)}$$

Note that the convention in this paper is to use an italic bold variable, such as $\boldsymbol{a}$, to refer to a Cartesian (three-dimensional) vector representing either a rotational or translational quantity, and a block bold variable, such as $\mathbf{a}$, to refer to a spatial (six-dimensional) vector.

The spatial transformation matrix, ${}^{i}\mathbf{X}_{p(i)}$, is used to transform spatial vectors between coordinate system i and its predecessor and is defined as follows (Lilly and Orin, 1991):

$$ {}^{i}\mathbf{X}_{p(i)} = \begin{bmatrix} {}^{i}\boldsymbol{R}_{p(i)} & 0 \\ {}^{i}\boldsymbol{R}_{p(i)}{}^{p(i)}\tilde{\boldsymbol{r}}_i^T & {}^{i}\boldsymbol{R}_{p(i)} \end{bmatrix}, \tag{A5} $$

where ${}^{p(i)}\boldsymbol{r}_i$ is the Cartesian vector specifying the position of the origin of link i's coordinate system with respect to link $p(i)$'s, and the tilde above the vector signifies that its components should be combined in a skew symmetric matrix such that $\tilde{\boldsymbol{b}}\boldsymbol{c} = \boldsymbol{b} \times \boldsymbol{c}$.

A.2. Dynamics

To complete the derivation of the AB algorithm, a set of dynamics equations for the force balance on each link is needed. For link i, this is given as follows:

$$ \mathbf{f}_i = \mathbf{I}_i \mathbf{a}_i - \beta_i + \mathbf{f}_i^+, \tag{A6} $$

where the vector, β_i, is a vector of bias forces that are a function of the link's angular velocity as follows:

$$ \beta_i = f(\omega_i). \tag{A7} $$

The vector $\mathbf{f}_i$ is the spatial force exerted onto link i by its inboard (predecessor) link, and $\mathbf{f}_i^+$ is the sum of the forces exerted by link i onto all of its outboard (successor) links. This quantity is *accumulated* using the predecessor function as follows:

$$ \mathbf{f}_{p(i)}^+ := \mathbf{f}_{p(i)}^+ + {}^{i}\mathbf{X}_{p(i)}^T\,\mathbf{f}_i \quad \forall i \text{ given } p(i). \tag{A8} $$

Like the spatial acceleration, spatial forces exerted on/by the link are also six-element vectors combining the moment, $\boldsymbol{n}$, and translational force, $\boldsymbol{f}$, vectors.

In the first term on the right side of Eq. (A6), link i's spatial inertia, $\mathbf{I}_i$, relates the spatial acceleration of the link to the resultant spatial force. This 6×6 matrix combines the link's mass and inertia quantities (first and second mass moments) as follows:

$$ \mathbf{I}_i = \begin{bmatrix} \bar{\boldsymbol{I}}_i & \tilde{\boldsymbol{h}}_i \\ \tilde{\boldsymbol{h}}_i^T & m_i \mathbf{1}_3 \end{bmatrix}, \tag{A9} $$

where $\bar{\boldsymbol{I}}_i$ is the 3×3 moment of inertia tensor for the link with respect to its own coordinate system, and $\mathbf{1}_3$ is the 3×3 identity matrix. The 3×1 vector, $\boldsymbol{h}_i$, is its first mass moment which is equal to $(m_i\, \boldsymbol{s}_i)$, where m_i is its mass and $\boldsymbol{s}_i$ is the vector from the link's coordinate system origin to its center of mass.

From Eq. (A2), the acceleration of link i depends on the acceleration of the inboard link. This implies that a recursion from the base at the root of the tree to the end-effectors may be used to compute the link accelerations when the joint accelerations, $\ddot{q}_i$, are known. Eq. (A6) implies a recursion from the end-effectors to the base to determine all of the link forces once the link accelerations have been determined. Together these two sets of equations define the outward and inward recursions associated with the *inverse* dynamics problem which computes the joint torques for a specified motion. For the case of forward dynamics computations, however, the joint accelerations are unknown so that the link accelerations, and hence the link forces cannot be determined in a direct manner from these equations. Consequently, a different approach is required to solve the dynamic simulation problem. In the AB algorithm, this involves the computation of AB inertias.

A.3. Articulated Bodies

Instead of using the force balance equation for a single link, an expression relating $\mathbf{f}_i$ to the dynamic properties of all the links outboard of it (i.e., the subtree of links whose root is link i) is used. This relationship is given as follows:

$$ \mathbf{f}_i = \mathbf{I}_i^* \mathbf{a}_i - \beta_i^*. \tag{A10} $$

The matrix, $\mathbf{I}_i^*$, is the 6×6 AB inertia of the outboard links which is the inertia that is "felt" at the ith coordinate system when the outboard joints are free to move. Likewise, the vector, β_i^*, is the bias force exerted on the ith link due to the resultant bias forces within the articulated body including the effects of all outboard joint torques. Details of the derivation of the equations needed to efficiently compute $\mathbf{I}_i^*$ and β_i^* can be found in McMillan (1994).

The resulting AB algorithm for forward dynamics contains three $O(N)$ recursions. The first is a Forward Kinematics recursion which computes the velocities and velocity-dependent terms, β_i and ζ_i, of each link from the base to the end-effectors. In the second step, the AB inertias, $\mathbf{I}_i^*$, and bias forces, β_i^*, are computed in a Backward Dynamics recursion from the end-effectors to the base. The final step begins with the computation of the base acceleration. In the case of a fixed base, this quantity is known, $\mathbf{a}_0 = \mathbf{0}$, and for mobile bases it can be computed from the AB inertia of the entire system and the resultant force on the reference member (the results of the second recursion). This result enables the computation of the first joint's acceleration, $\ddot{q}_1$, with an equation derived from Eqs. (A2) and (A10). This enables the computation of link 1's acceleration from Eq. (A2). These results are used to compute the joint and link accelerations (in that order) for the successor links in the tree. This procedure defines the final Forward Accelerations recursion from the base to the end-effectors.

Acknowledgments

The authors wish to recognize the financial support of the Office of Naval Research and the Naval Explosive Ordnance Disposal Technical Center, Indian Head, MD as managed by Mr. Chris O'Donnel under contract N0464A95WR01915. This work was also supported in part by a DuPont Fellowship and an AT&T Ph.D. Scholarship both at The Ohio State University, by Grant No. BCS-9306252 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.

Note

1. Of course, C++ is not the only language possessing object oriented features. However, it is the defacto standard for real-time graphical simulation and was therefore chosen for this research.

References

Apostolopoulos, D. and Bares, J. 1995. Configuration of a robust rappelling robot. In *IEEE/RSJ Int. Conf. on Intelligent Robotics and Systems*, Pittsburgh, PA.

Basu, A., Elnagar, A., and Fiala, M. 1995. Surface integration for inspection tasks. In *IEEE Int. Conf. on Robotics and Automation*, Nagoya, Japan, pp. 1560–1566.

Boman, D.K. 1995. International survey: Virtual-environment research. *Computer*, 28(6):57–65.

Bono, R., Caccia, M., and Veruggio, G. 1995. Simulation and control of an unmanned underwater vehicle. In *IEEE Int. Conf. on Robotics and Automation*, Nagoya, Japan, pp. 1573–1578.

Booch, G. 1986. *Object Oriented Design with Applications*, Benjamin/Cummings Publishing Co. Inc.: Redwood City, CA.

Brandl, H., Johanni, R., and Otter, M. 1986. A very efficient algorithm for the simulation of robots and similar multibody systems without inversion of the mass matrix. In *Proc. of IFAC/IFIP/IMACS Int. Symp. on Theory of Robots*, Vienna, Austria.

Craig, J.J. 1986. *Introduction to Robotics: Mechanics and Control*, Addison-Wesley: Reading, MA.

Featherstone, R. 1983. The calculation of robot dynamics using articulated-body Inertias. *The Int. Journal of Robotics Research*, The MIT Press, 2:13–30.

Featherstone, R. 1987. *Robot Dynamics Algorithms*, Kluwer Academic Publishers: Boston.

Fossen, T.I. 1995. Underwater vehicle dynamics. In *Underwater Robotic Vehicles: Design and Control*, J. Yuh (Ed.) TSI Press: Albuquerque, NM, pp. 15–40.

Freeman, P.S. and Orin, D.E. 1991. Efficient dynamic simulation of a quadruped using a decoupled tree-structured approach. *Int. Journal of Robotics Research*, The MIT Press, 10(6):619–627.

Fujii, T. and Ura, T. 1995. Autonomous underwater robots with distributed behavior control architecture. In *IEEE Int. Conf. on Robotics and Automation*, Nagoya, Japan, pp. 1868–1873.

Goheen, K.R. 1995. Techniques for URV modeling. In *Underwater Robotic Vehicles: Design and Control*, J. Yuh (Ed.) TSI Press: Albuquerque, NM, pp. 99–126.

Ioi, K. and Itoh, K. 1990. Modelling and simulation of an underwater manipulator. *Advanced Robotics*, 4(4):303–317.

Iwasaki, M., Akizono, J., Takahashi, H., Umetani, T., Nemoto, T., Asakura, O., and Asayama, K. 1987. Development on aquatic walking robot for underwater inspection. *Report of the Port and Harbour Research Institute*, 26(5):393–422.

Kato, N. 1995. Application of fuzzy algorithms to guidance and control of AUV. In *Workshop on Robotic Technologies in Oceanic Engineering*, Nagoya, Japan, pp. 38–46.

Khalil, W. and Kleinfinger, J.F. 1986. A new geometric notation for open and closed-loop robots. In *IEEE Int. Conf. on Robotics and Automation*, San Francisco, CA, pp. 1174–1179.

Lane, D.M. 1995. Subsea robotics for the offshore industry. In *Workshop on Robotic Technologies in Oceanic Engineering*, Nagoya, Japan, pp. 8–16.

Lane, D.M. and Knightbridge, P.J. 1995. Task planning and world modelling for supervisory control of robots in unstructured environments. In *IEEE Int. Conf. on Robotics and Automation*, Nagoya, Japan, pp. 1880–1885.

Lilly, K.W. and Orin, D.E. 1991. Alternate formulations for the manipulator inertia matrix. *International Journal of Robotics Research*, The MIT Press, 10(1):64–74.

McMillan, S. 1994. Computational dynamics for robotic systems on land and under water. Ph.D. thesis, The Ohio State University: Columbus, Ohio.

McMillan, S. and Orin, D.E. 1995. Efficient computation of articulated-body inertias using successive axial screws. *IEEE Trans. on Robotics and Automation*, 11(4):606–611.

McMillan, S., Orin, D.E., and McGhee, R.B. 1995a. DynaMechs: An object oriented software package for efficient dynamic simulation of underwater robotic vehicles. In *Underwater Robotic Vehicles: Design and Control*, J. Yuh (Ed.) TSI Press: Albuquerque, NM: pp. 73–98.

McMillan, S., Orin, D.E., and McGhee, R.B. 1995b. Efficient dynamic simulation of an underwater vehicle with a robotic manipulator. *IEEE Trans. on Systems, Man, and Cybernetics*, 25(8):1194–1206.

Newman, J.B. and Robison, B.H. 1992. Development of a dedicated ROV for ocean science. *Marine Technology Society Journal*, 26(4):46–53.

Roberson, R.E. and Schwertassek, R. 1988. *Dynamics of Multibody Systems*, Springer-Verlag: Berlin.

Santos, A., Rives, P., Espiau, B., and Simon, D. 1995. Dealing in real time with a priori unknown environment on autonomous underwater vehicles (AUVs). In *IEEE Int. Conf. on Robotics and Automation*, Nagoya, Japan, pp. 1579–1584.

Sayers, C.P., Lai, A., and Paul, R.P. 1995. Visual imagery for subsea teleprogramming. In *IEEE Int. Conf. on Robotics and Automation*, Nagoya, Japan, pp. 1567–1572.

Vagany, J. and Rigaud, V. 1995. Supervised navigation: Optimal algorithm example and needs for autonomous supervision. In *IEEE Int. Conf. on Robotics and Automation*, Nagoya, Japan, pp. 1874–1879.

Wong, H.C. and Orin, D.E. 1995. Control of a quadruped standing jump over irregular terrain obstacles. *Journal of Autonomous Robots*, 1:111–129.

Yoerger, D.R., Bradley, A.M., and Walden, B. 1995. Automatic docking system for the autonomous benthic explorer. In *Workshop on Robotic Technologies in Oceanic Engineering*, Nagoya, Japan, pp. 27–37.

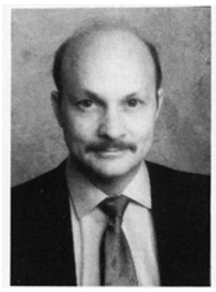
Yuh, J. 1990. Modeling and control of underwater robotic vehicles. *IEEE Transactions on Systems, Man, and Cybernetics*, 20(6):1475–1483.

Yuh, J. 1995a. Development in underwater robotics. In *IEEE Int. Conf. on Robotics and Automation*, Nagoya, Japan, pp. 1862–1867.

Yuh, J. (Ed.) 1995b. *Workshop on Robotic Technologies in Oceanic Engineering*, Nagoya, Japan, May.

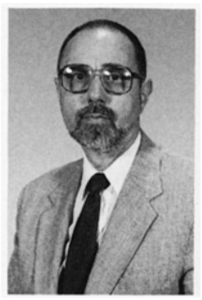
Yu, J. and Waldron, K.J. 1991. Design of wheeled actively articulated vehicle. In *Proceedings of the Second National Applied Mechanisms and Robotics Conference*, Cincinnati, OH.

Scott McMillan received the B.S. degree in Computer Engineering from Clemson University, Clemson, SC in 1988, and the M.S. and Ph.D. degrees in Electrical Engineering from The Ohio State University in 1990 and 1994, respectively. From 1994 to 1996, he was a member of the research faculty at the Naval Postgraduate School in Monterey, CA. He is currently a member of the Core Technology Group at High Techsplanations, Inc. in Rockville, MD, performing research in and development of surgical simulation systems. His research interests include legged locomotion, control systems for autonomous underwater vehicles, physically-based modeling for virtual reality simulations, parallel processing, and object-oriented design. He is a member of IEEE and Tau Beta Pi.



David E. Orin received his Ph.D. degree in Electrical Engineering from The Ohio State University, Columbus, in 1976. From 1976 to 1980, he taught at Case Western Reserve University, Cleveland, OH, in the Department of Electrical Engineering and Applied Physics. Since 1981, he has been at The Ohio State University, where he is currently a Professor of Electrical Engineering. His current research interests include underwater robotic systems, computational robot dynamics, control of enveloping grasping systems, parallel computation, and control of legged machines.

Dr. Orin is the Vice President for Finance of the IEEE Robotics and Automation Society. He is a Fellow of the IEEE and a member of Sigma Xi, Tau Beta Pi, and Eta Kappa Nu.



Robert B. McGhee was born in Detroit, Michigan in 1929. He received the B.S. degree in Engineering Physics from the University of Michigan in 1952, and the M.S. and Ph.D. degrees in Electrical Engineering from the University of Southern California in 1957 and 1963 respectively.

From 1952 until 1955, he served on active duty as a guided missile maintenance officer with the U.S. Army Ordnance Corps. From 1955 until 1963, he was a member of the technical staff with Hughes Aircraft Company, Culver City, CA, where he worked on guided missile simulation and control problems. In 1963, he joined the Electrical Engineering Department at the University of Southern California as an Assistant Professor. He was promoted to Associate Professor in 1967. In 1968, he was appointed Professor of Electrical Engineering and Director of the Digital Systems Laboratory at Ohio State University, in Columbus, OH. In 1986, he joined the Computer Science Department at the Naval Postgraduate School in Monterey, CA, where he served as Chairman from 1988 until 1992. Since 1992, he has held a joint appointment as Professor in the Computer Science Department and in the Electrical and Computer Engineering Department at the Naval Postgraduate School.

Dr. McGhee is a Fellow of the Institute of Electrical and Electronic Engineers. He currently teaches in the areas of artificial intelligence, robotics, computer languages, and feedback control theory. His research interests are centered around computer simulation and control of unmanned vehicles, especially for subsea applications.