Proceedings of IDETC/CIE 2008
ASME 2008 International Design Engineering Technical Conferences &
Computers and Information in Engineering Conference
August 4-7, 2008, New York City, USA

DETC2008-49399

AN INTEGRATED CONTROL AND SIMULATION ENVIRONMENT FOR MOBILE ROBOT SOFTWARE DEVELOPMENT

Stephen Balakirsky[*]
Frederick M. Proctor
Christopher J. Scrapper
Thomas R. Kramer
National Institute of Standards and Technology
Gaithersburg, MD USA 20899

ABSTRACT
In order to expedite the research and development of robotic systems and foster development of novel robot configuration, it is essential to develop tools and standards that allow researchers to rapidly develop, communicate, and compare experimental results. This paper describes the Mobility Open Architecture Simulation and Tools Framework (MOAST). The MOAST framework is designed to aid in the development, testing, and analysis of robotic software by providing developers with a wide range of open source robotic algorithms and interfaces. The framework provides a physics-based virtual development environment for initial testing and allows for the seamless transition of algorithms to real hardware. This paper details the design approach, software architecture and specific module-to-module interfaces.

1 Introduction and Related Work
The usefulness of simulation for developing control systems is well established. The role of simulation is to provide convincing sensor measurements in response to a controller's actuator outputs in an environment observable to developers. Ideally the simulation should be accurate enough so that performance parameters tuned in simulation work as well in the real world. In practice, attaining this level of simulation is often more costly than real-world testing, and simulators that respond plausibly if not accurately are acceptable. Plausible simulation then complements real-world testing to minimize the time and effort needed to build controllers that work well. Several such simulation systems exist; several of which are open source, including the Unified System for Automation and Robot Simulation (USARSim) [1] and the Stage and Gazebo components of Player/Stage [2].

While the typical simulation system allows one to directly connect and experiment with servo-level controllers, they in general lack any form of intelligence or ability to interpret sensor readings and issue meaningful commands. For this reason, it is necessary to connect the simulation engine to a control framework. Several such systems exist in the literature and on the web. Perhaps the most popular of which is the Player portion of Player/Stage.

Player/Stage combines a robot server interface, called Player, with a simulation system, called Stage, so that Player-enabled robots can be easily interchanged with each other and their simulated counterparts. The Player interface is installed on robotic vehicles, providing an interface to the robot's sensors and actuators over a TCP/IP network. Player was originally ported to robots in the ActivMedia Pioneer 2 family, but other robots and sensors are supported. Stage simulates a population of robotic vehicles and sensors in a 2-D environment. Gazebo is a 3-D counterpart provided for outdoor simulation. While Player started as a robot interface with drivers that directly control hardware, it has grown to include several abstract drivers since then. These abstract drivers use other drivers, instead of hardware, as

[*]Address all correspondence to this author at stephen.balakirsky@nist.gov.

the sources for data and the sinks for commands. Several well-known algorithms are now included with the system thus providing services such as way-point navigation and obstacle avoidance.

A new entry to the robot simulation/control arena is Microsoft's Robotics Studio [3]. Robotics Studio includes support for simulation and implements services to control robotic platforms. These services may be created in a variety of programming languages, or by using Microsoft's visual programming language. While Robotics Studio is not as mature as Player/Stage, it promises to build a library of services that will be available to robot developers.

1.1 Adding an Architecture: MOAST

Player and Robotics Studio both focus on the interfaces to mobile robots that allow developers to build their own controllers, with portability across robots that support Player or Robotics Studio made easier. Neither Player nor Robotics Studio defines an overall architecture or high-level interface specification that guides the development of robot controllers. We have found that an architecture is essential to the efficient development of intelligent systems. An architecture assigns roles and responsibilities among controllers and dictates what services are necessary. It defines module timing, data and control interfaces, and planning extents. An architecture also provides the framework in which the rest of the intelligent system resides and dictates the rules that the modules must follow. For these reasons, we built the Mobility Open Architecture Simulation and Tools Framework (MOAST). MOAST begins with a well defined architecture, and adds simulations, services, and controllers. The entire MOAST framework is intended to provide tools to lead a researcher through all of the phases of development and testing of an autonomous agent system.

MOAST is made up of the following components:

1. A reference model architecture that dictates how control responsibilities are divided between modules.
2. Communication interface specifications that dictate how and what modules will communicate.
3. Sample control modules for the control of a sample simulated robotic platform. These modules include sensor processing, world modeling, and behavior generation for 4 levels of the hierarchical architecture and provide a complete control system.
4. Validated sensors and robot models in the simulation.
5. Tools to aid in development and debug of the control system.

The remainder of this paper will address the components of MOAST. Section 2 describes the reference model architecture that is utilized by MOAST. Section 3 describes the various services and capabilities that are provided by the framework, Section 4 describes where in the architecture these services reside and their interfaces. Finally, Section 5 describes future work and concludes the paper.

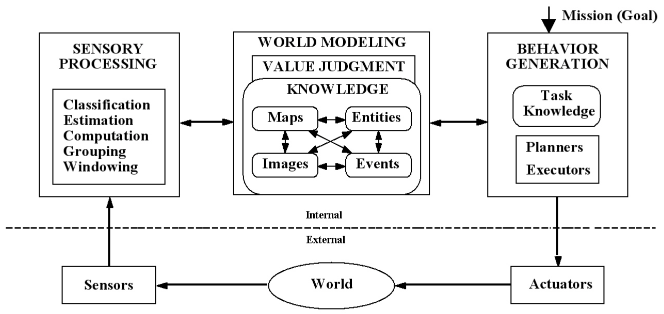


Figure 1. Generic 4D/RCS Control Node.

2 Reference Model Architecture

The capabilities of the MOAST framework are encapsulated in components that are designed based on the Four-Dimensional Real-Time Control System (4D/RCS) Reference Model Architecture [4] [5]. The 4D/RCS reference model architecture is a hierarchical, distributed, real-time control system architecture that decomposes a robotic system into manageable pieces while providing clear interfaces and roles for a variety of functional elements.

Figure 1 depicts the general structure of each echelon (level) of the 4D/RCS hierarchy. Each echelon in 4D/RCS has a regular structure comprised of control nodes that perform the same general type of functions: *sensory processing* (SP), *world modeling* (WM), *value judgment* (VJ), and *behavior generation* (BG). Sensory processing is responsible for populating the world model with relevant facts. These facts are based on both raw sensor data and the results of previous SP (in the form of partial results or predictions of future results). WM must store this information, information about the system itself, and general world knowledge and rules. Furthermore, it must provide a means of interpreting and accessing this data. BG computes possible courses of action to take based on the knowledge in the WM, the system's goals, and the results of plan simulations. VJ aids in the BG process by providing a cost/benefit ratio for possible actions and world states.

The principal difference between control nodes at the same echelon is in the set of resources managed, while the principal difference between nodes at different echelons is in the knowledge requirements and the resolution of the planning space.

This regularity in the structure enables flexibility in the system architecture that allows scaling of the system to any arbitrary size or level of complexity [6]. Each level within 4D/RCS has a characteristic range and resolution in space and time. Each level has characteristic tasks and plans, knowledge requirements, values, and rules for decision-making.

Every module in each level has a limited span of control, a limited number of tasks to perform, a limited number of resources to manage, a limited number of skills to master, a limited planning horizon, and a limited amount of detail with which to cope.

2.1 Generic Module

While 4D/RCS provides a reference model for the architecture, MOAST is an implementation of that architecture. Therefore, specific responsibilities, knowledge requirements, and interfaces have been designed for each control module. Each control module is based upon a generic core controller that is shown in Figure 2. The MOAST hierarchical decomposition of in terms of its control modules is depicted in Figure 3.

The control module core has the following flow:

1. **Initialize:** The initialization opens any communication buffers, places the system in a safe known state, and initializes any control parameters.
2. **Read Command:** Command information is received, and the system is prepared to execute the command. When a new command is received, the old command is immediately replaced by this command. If the command is not able to be interrupted, error flags are set and will be reported during the writing of status.
3. **Read Config:** Configuration information is received, and the system is prepared to change its settings. A separate configuration channel is provided to allow for control parameters to be changed without interrupting the current controller. For example, a user may want to change a system's cycle time without interrupting a complex control function. The configuration channel is used to change the cycle time parameters without aborting the control algorithm.
4. **Handle Command State Tables:** All of the modules run on a fixed cycle time. Therefore, command functions must either guarantee that they finish in under the cycle time, or provide for being reentrant. Although this text refers to the command execution as being finite state machine (FSM) based, search based planning systems have also been implemented under this framework. The searcher simply stores its current location in the search when its execution time expires and then resumes searching on the next cycle.
5. **Handle Config State Tables:** Requests to change configuration settings are carried out similarly to handling command state tables.
6. **Write Status and Settings:** The current module status and the configuration's settings are sent out over communication buffers.

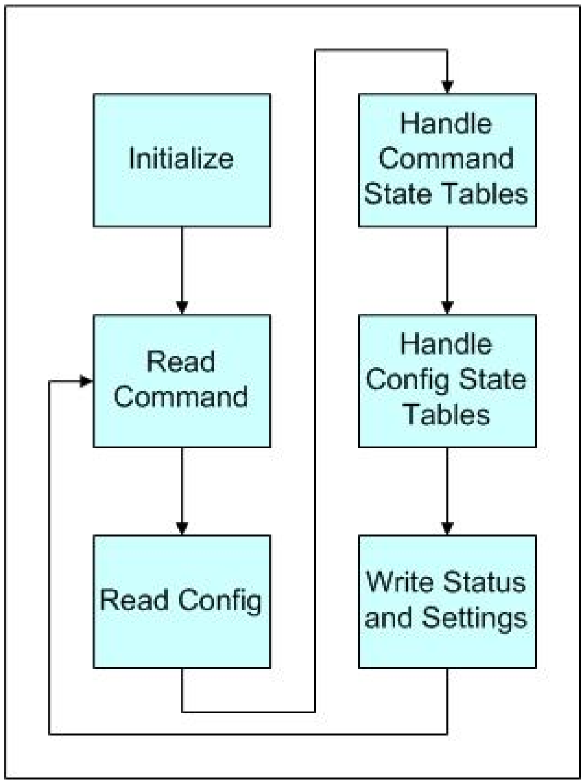
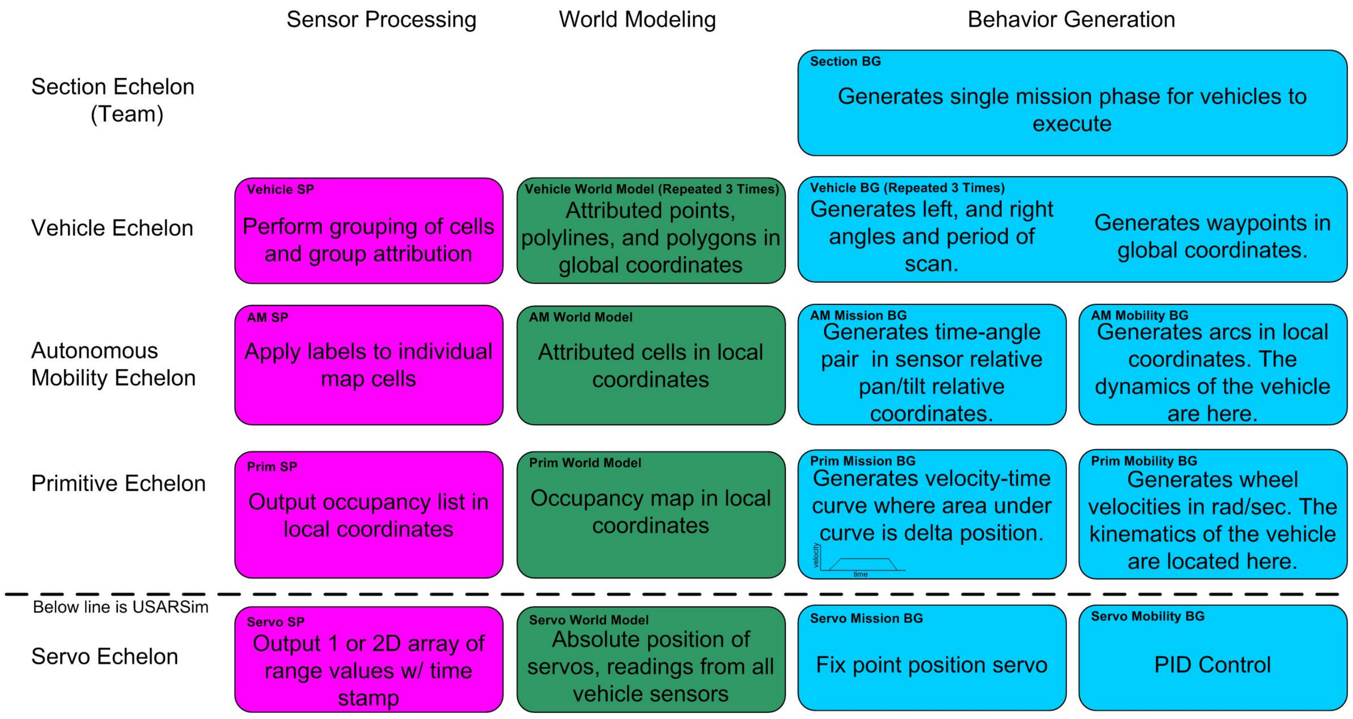

Initialize
Read Command
Read Config
Handle Command State Tables
Handle Config State Tables
Write Status and Settings

Figure 2. Generic core of control module.

2.2 RCS Library

Support for developing software conforming to the 4D/RCS methodology is provided by the RCS Library [7]. The RCS library includes portable utilities for creating and synchronizing real-time tasks following the 4D/RCS architecture. Code generation and diagnostics tools simplify much of the application setup and debugging. Communication between RCS control modules is provided by the Neutral Messaging Language (NML), a software library for communication ported to a variety of platforms including Linux, Solaris, VxWorks, LynxOS, QNX, Windows and MacOS. Applications using NML define a message vocabulary as C++ classes and call C++ methods to open buffers, read and write messages. Java bindings are also available. NML applications running on one platform can communicate with ones running on any other platform. It is based on message buffers of a fixed maximum size that can contain messages of variable length. NML supports blocking- and non-blocking reads, queued and non-queued writes, polled or publish-subscribe communication. A uniform application programming interface enables the same user source code regardless of computing platform; targeting new platforms requires recompiling, not recoding. Support for different communication protocols is provided via configuration files, such as operating system shared memory, backplane



Sensor Processing
World Modeling
Behavior Generation
Section Echelon (Team)
Section BG
Generates single mission phase for vehicles to execute
Vehicle Echelon
Vehicle SP
Perform grouping of cells and group attribution
Vehicle World Model (Repeated 3 Times)
Attributed points, polylines, and polygons in global coordinates
Vehicle BG (Repeated 3 Times)
Generates left, and right angles and period of scan.
Generates waypoints in global coordinates.
Autonomous Mobility Echelon
AM SP
Apply labels to individual map cells
AM World Model
Attributed cells in local coordinates
AM Mission BG
Generates time-angle pair in sensor relative pan/tilt relative coordinates.
AM Mobility BG
Generates arcs in local coordinates. The dynamics of the vehicle are here.
Primitive Echelon
Prim SP
Output occupancy list in local coordinates
Prim World Model
Occupancy map in local coordinates
Prim Mission BG
Generates velocity-time curve where area under curve is delta position.
velocity
time
Prim Mobility BG
Generates wheel velocities in rad/sec. The kinematics of the vehicle are located here.
Below line is USARSim
Servo Echelon
Servo SP
Output 1 or 2D array of range values w/ time stamp
Servo World Model
Absolute position of servos, readings from all vehicle sensors
Servo Mission BG
Fix point position servo
Servo Mobility BG
PID Control

Figure 3. Modular decomposition of MOAST framework that provides modularity in broad task scope and time.

global memory, TCP/IP sockets and UDP datagrams. New protocols can be added without affecting application code. NML converts between processor data formats (big and little endian), handles compiler structure padding and mutual exclusion. The location of buffers and processes that connect to them is selected at run time, and a running application can be extended to communicate with new processes dynamically. NML source code is freely available at [8] and documented in [7].

3 MOAST Provided Functions

The development and maintenance of an advanced mobile robot requires expertise ranging from sensor processing, path planning, and communications protocols, to basic auto repair. While many of the algorithms for accomplishing these functions are well known, freely available code that implements these functions tends to be incompatible with other code or robotic platforms. This necessitates interface and functional tweaks before the code modules become useful.

Part of the original design philosophy of MOAST was to provide "out of the box" functionality that would reduce the breadth of expertise required to conduct research with mobile robots. The developers of MOAST have taken many well known algorithms and implemented them within the 4D/RCS framework. The result is a fully functional framework that allows researchers and students to immediately begin to experiment with functional robots in simulated environments. Researchers are then free to examine the code modules that address functions in their areas of expertise. The hope is that as improvements are made, the researchers will contribute the improved modules back to the community. The basic functionality of the mobile robot may be broken down in the the areas of sensor processing and mobility.

Table 1. Sensor processing requirements and responsibilities.

Data Out	Description
Primitive Echelon laser scan data	Beam start and hit point
Autonomous Mobility Echelon height map	Cellular map of 2.5D elevations
Autonomous Mobility Echelon obstacle probability	Cellular map of obstacle probabilities

3.1 Sensor Processing

The majority of the sensor processing work performed in MOAST is in the detection of obstacles. The decomposition

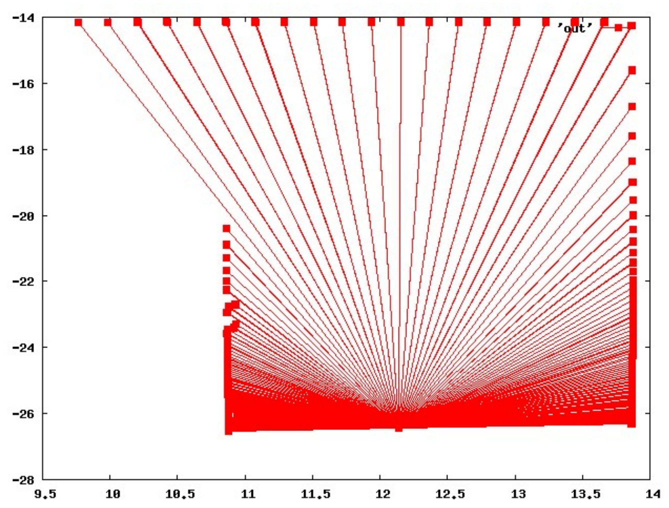


Figure 4. Laser range data from Prim includes the start and end of each beam.

of this responsibility by echelon in the MOAST framework is shown in Table 1. For the laser scanner, the Primitive (Prim) Echelon provides a series of data tuples as shown in Figure 4. The data is available over the communications interface and includes the location of the device when the beam was fired and the beam hit point in vehicle relative coordinates. Under the current system, the laser is constrained to be fixed mounted and facing straight ahead of the vehicle in a level orientation. While this presents an instantaneous snapshot of the environment, the data tends to be noisy, and encompass a very small region. This data is further processed to produce a cellular height map of the environment as shown in Figure 5. Due to the mounting constraint on the laser, whenever the vehicle is driving on a flat level surface the height of every cell that has been observed is either the height of the floor or the height of the laser above the floor.

While the external representation is transmitted as a cellular height map, internally, the cell's height, range, hits history, and obstacle probability are stored. The model for the terrain being observed is like a 3D bar chart, where solid blocks of various heights extend through cells in the XY plane. The height of each cell records the estimated distance its block extends above the local XY plane. The height is negative if the top of the block is below the XY plane. The range of a cell records the largest distance from which a cell has been seen to contain an obstacle. Some obstacles are seen only when they are close to the sensor. It is desired to avoid having the system decide that an obstacle no longer exists because it is not seen when the system is farther away than its range. It is expected that if a cell containing an obstacle is viewed from within the cell's range, the obstacle will be seen again, but if the cell is viewed from beyond its range, the obstacle might not be seen. The range is used in setting the hits history, as described below.

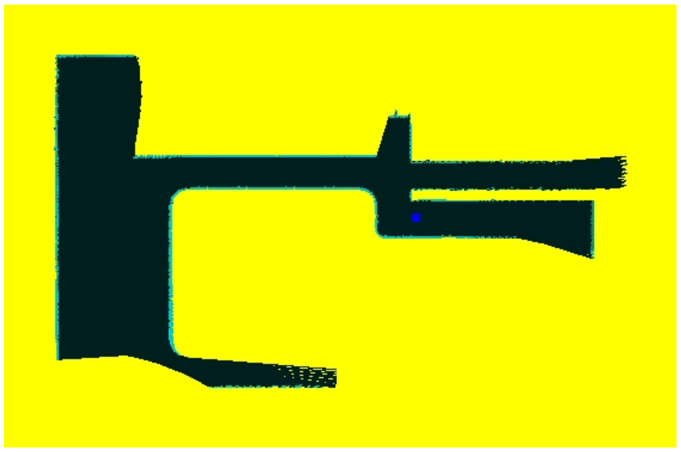


Figure 5. Cellular height map generated from laser data. Yellow represents cells that have never been seen, and cells that are observed are shown in shades of aqua based on their height. Due to the mounting configuration of the laser, only heights of "ground," shown as very dark aqua (i.e. black), or heights of above the laser, shown as bright aqua, are displayed (hard copies of this paper should be printed in color).

Table 2. Mobility planning requirements and responsibilities.

Plan Out	Command In	Knowledge In
Prim actuator commands	Constant curvature arcs	Kinematics
AM constant curvature arcs	Way-points	Dynamics
Vehicle way-points	Named location	*a priori* map
Section vehicle actions	Behaviors	Vehicle Capabilities

The hits for a cell encodes the seven most recent viewings of the cell. A cell is regarded has having been viewed whenever a ladar ray passes through it (the cell is not seen) or bounces off an obstacle in it (the cell is seen).

Obstacle probability is a real number from 0.0 to 1.0. It represents the system's best estimate of the chances that the cell is occupied by an obstacle. A separate map of obstacle probability is exported over a communication channel for use by other modules.

3.2 Mobility

The mobility functions consist of a family of planning algorithms that are able to compute obstacle free paths for Ackerman, skid-steered, and omni-drive ground robot platforms as well as helicopter-like air platforms and sub-like underwater vehicles. When examining the planning systems, it is useful to note the knowledge required by each module as well as the module's output format, i.e., the form the plan takes. This information is represented in Table 2.

At the lowest echelon, the output of the planning system consists of actuator and motor commands that are sent through MOAST's generic interface known as SIMware [9]. These commands are platform steering type dependent and consist of such things as left and right wheel velocities for a skid-steered vehicle or steering curvature and velocity for an Ackerman steered vehicle. This module requires a series of obstacle free constant curvature arcs as input. In addition to the command input, the module requires knowledge of the specific robot kinematics. Such items as wheelbase, tire diameter, and minimum turning radius must be provided.

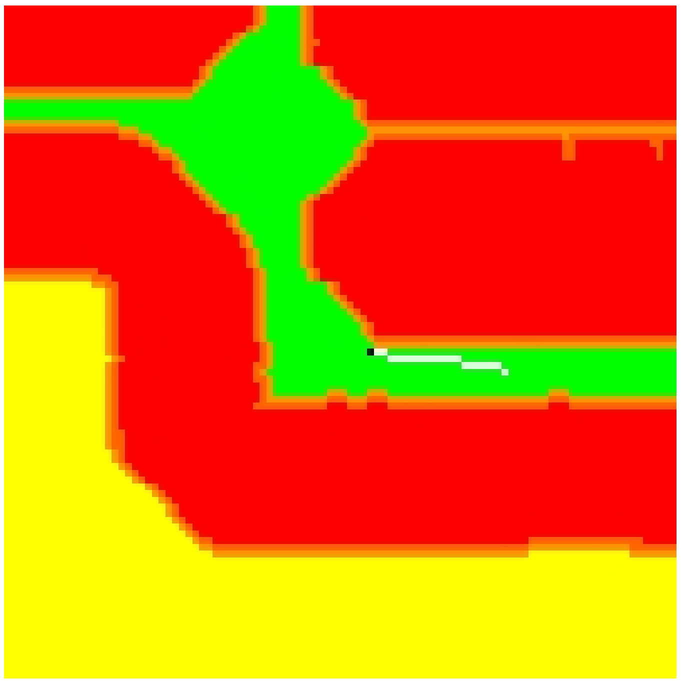


Figure 6. Cellular obstacle map generated from obstacle probability data. Yellow represents cells that are unknown. Green represents free-space, orange represents the edge of obstacles, and red represents obstacles. The obstacles are grown by half the vehicle width to allow for the planner to plan on a point-sized robot. The white path represents the planned path for the platform and the platform's current location is shown as the black dot (hard copies of this paper should be printed in color).

An additional way-point interface exists into the planning hierarchy. This interface accepts a series of way-points as its commands and computes a series of obstacle-free constant curvature arcs as output. This module reads in the obstacle probability map from the sensor processing chain and also has knowledge of the vehicle dynamics. A graphical example of this module's output is shown in Figure 6.

This planning module has two main strengths. In well-behaved simulated environments, it quickly plans realistic smooth paths with appropriate speeds and curve radii while keeping within the allowed deviation and avoiding obstacles. Second, it plans paths dynamically in environments with moving obstacles (such as other vehicles). Weaknesses of this planner stem primarily from not getting enough sensory information and not attempting to use all the information available. Knowing the slope of the surface on which the vehicle is running is critical for setting speeds and radii of turns that can be achieved without having the vehicle slide or overturn. Height data is collected by sensor processing, but is not currently sent to the planner. The planner could process height data to create slope data, but there are currently no functions for generating or using slope data. As noted below, however, the current method of collecting height data is not very good. No information on surface properties (such as coefficient of friction or hardness) is available from sensor processing, and the planner has no functions for using such information.

The obstacle information that is available to the planner is very limited because of the way in which laser range (ladar) data is being collected. The ladar looks straight ahead of the vehicle at the height at which the ladar is mounted on the vehicle. As a result, sensor processing cannot see holes of any sort and cannot see obstacles lower than the height of the ladar. Thus, the range of environments in which the plans are realistic is very small, namely flat surfaces with no holes and on which every obstacle is tall. Better obstacle information would be used effectively with no change to the planner. Putting the ladar on a pan/tilt platform would allow good height data to be collected and is being implemented. A different sort of weakness is the inability of the searcher to include the cost of traversal time while choosing a path. Time is a critical element of cost. Distance and traversal difficulty are known before a path is chosen, so they can be used in finding an optimal cost path. The amount of time taken, however, cannot be chosen beforehand, since the speed depends on how sharp the corners are, and the corners are not rounded until an approximate path has been chosen.

If *a priori* data is available, then a planning module exists to take advantage of this data. This module ingests *a priori* vector data and computes a visibility graph based plan that starts at the way-point planner's planning horizon and terminates at a named point (for example an address). This system currently reads .mif formatted vector data. An example of the plan output is shown in Figure 7. The system accepts a named point as its input and

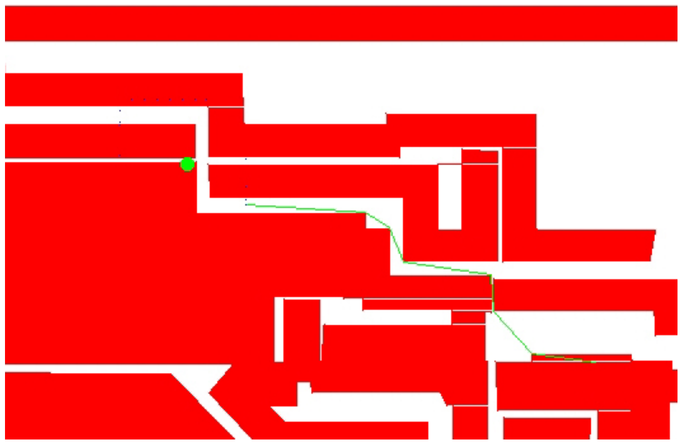


Figure 7. Vector-based *a priori* map used by the *a priori* planner. The red areas are obstacle polygons and the white is free space. The computed path is shown in green, and the robot location is shown with a large green circle. The small green dots represent the planning horizon of the way-point planning system.

outputs a list of way-points for the platform to follow.

Finally, a planning system capable of coordinating groups of vehicles exists. This planner accepts behavior based commands (i.e. explore, or deliver packages) and coordinates the actions of several platforms to accomplish the tasks. The system accepts the behavior command as its input and outputs named points and tasks for the platforms to accomplish. The system must have knowledge of individual platform capabilities.

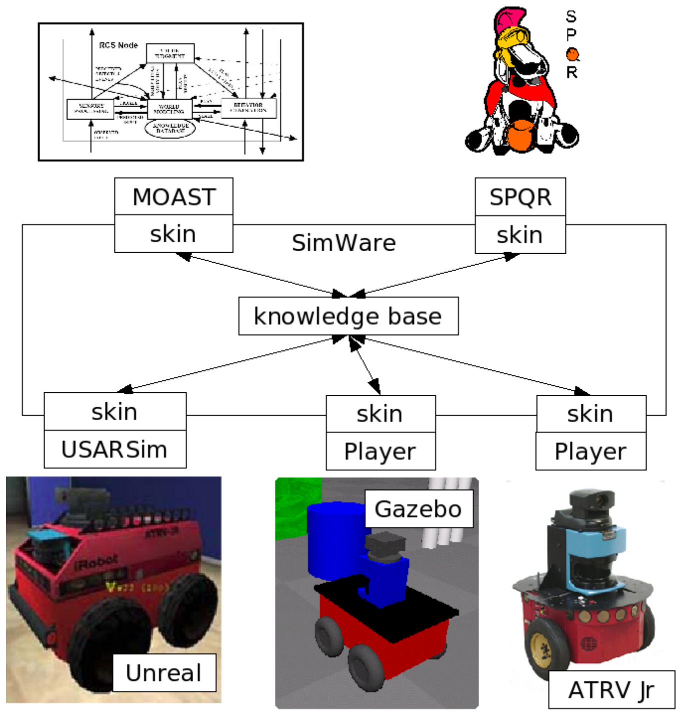


Figure 8. The SIMware Integration Architecture. SIMware bridges particular implementations of the Servo Echelon, such as Unreal or Gazebo vehicle simulations or real vehicles, with controllers such as MOAST or SPQR from the University of Rome.

4 Hierarchical Decomposition

This section describes each of the MOAST echelons in detail, beginning at the bottom with the interface to the simulation system or real robots, and continuing up through increasing levels of abstraction to the Section Echelon, which coordinates multiple vehicles.

4.1 SIMware

The Servo Echelon in Figure 3 is implemented outside of MOAST by real or simulated vehicles. To limit the spread of vehicle-specific source code into MOAST, an external middleware layer was built that bridges different controllers to different vehicles or vehicle simulations. This Simulation Interface Middleware, called “SIMware,” defines a software application programming interface (API) based on the notion of “skins” that customize an environment to particular controllers and simulations or real vehicles [9]. Skins are divided into superior skins that interface SIMware to vehicle controllers, and inferior skins that interface SIMware to simulations. Programmers build a SIMware middleware layer by instantiating a particular superior skin that interfaces to a controller, instantiating a particular inferior skin that interfaces to a robot and sensor environment, and defining configuration information specific to each skin.

The SIMware middleware layer is an executing program, typically with several concurrent threads, that provides a knowledge base that is updated and read via the skin interface. SIMware itself does not contain any threads of execution; rather, the skins may spawn threads if needed to help in the conversions. Typically, a thread in the superior skin will read command messages, and a thread in the inferior skin will read sensor reports, and each will query and update the knowledge base.

Typically, a SIMware-based application links in skins for specific controllers and simulations. It is possible to build skins that can dynamically switch between different controllers and simulations, or synchronize parts of a simulation with the real world. For example, a SIMware user may decide to integrate all of their simulation skins into a single combined skin, enabling run-time switching between various simulations and the real world. Figure 8 shows the SIMware concept.

4.2 Prim-Servo

The Prim Echelon interfaces via SIMware's MOAST superior skin to the Servo Echelon. Messages between Prim and Servo are divided between mobility, mission packages and sensor processing.

Prim mobility is responsible for planning a series of velocity states given a list of arcs with tolerances and desired tangential speeds. The velocity state is a screw of linear and angular velocity that is converted into vehicle actuator setpoints using the inverse Jacobian function associated with each vehicle type.

Planning is complicated by the unpredictable performance of the vehicle. For example, the vehicle may have slipped off the nominal circular arc path due to wet conditions. Prim continually monitors the navigation state from sensors such as a Global Position System (GPS) receiver or an Inertial Navigation System (INS) to determine if the vehicle is within the allowable neighborhood. If not, Prim plans a move directly toward the path in order to prevent collisions with obstacles presumed to lie outside.

Within Prim, arcs are converted to a series of closely-spaced way-points based on the tolerance. The tolerance sets a neighborhood around each point within which the vehicle is free to move toward the next way-point. For the next target way-point, Prim computes the deviation in heading, and sets the vehicle velocity and angular velocity according to a cutoff angle θ_c. Translational speed is reduced linearly from its desired tangential speed v_{max} at zero heading deviation to some minimum speed v_{min} at the cutoff angle, and clamped to the minimum beyond that. Angular speed is set to zero when there is no heading deviation, and increases linearly to its maximum at and beyond the cutoff angle. In particular we have

$$\begin{aligned}\omega &= \max(-\omega_{max}\theta/\theta_c,\ -\omega_{max}) \quad && \theta \geq 0 \\ \omega &= \min(-\omega_{max}\theta/\theta_c,\ \omega_{max}) \quad && \theta < 0 \\ v &= \max(v_{max}(1-|\theta|/\theta_c),\ v_{min}) &&\end{aligned}$$

where θ is the angular difference, θ_c is the cutoff angle, v_{max} is the desired tangential maximum speed, and v_{min} is the minimum achievable speed from the constraint $v_{min} = \omega r_{min}$ for vehicles with a minimum turning radius r_{min}. The relationships between the speed, angular speed and heading deviation are shown in Figure 9. For skid-steer vehicles that have a zero minimum turning radius, the minimum translational speed can be zero. Figure 10 shows the behavior of such a vehicle for various values of the cutoff angle. For heading deviations outside the cutoff angle, the translational speed is zero and the vehicle rotates until it is pointed toward the goal, at which point the translational speed picks up and moves the vehicle. For larger cutoff angles, the vehicle is free to move when it is pointed far from the goal, and wide deviations are seen. As the cutoff angle is made smaller, the

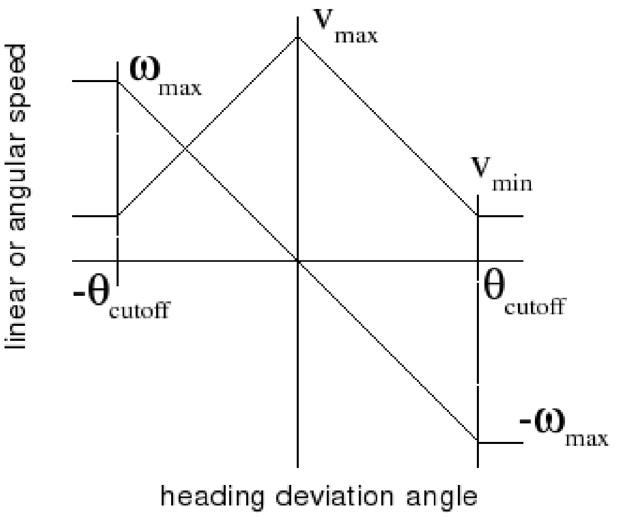


Figure 9. Vehicle speed and angular speed relative to heading deviation. As the deviation between the commanded heading and actual heading increases, the commanded vehicle speed decreases and its angular speed increases. The increased angular speed is a response that points the vehicle in the proper direction.

vehicle never deviates far from a direct line to the goal, although the vehicle speeds are smaller.

Each Prim planning cycle's v and ω are run through the vehicle's inverse Jacobian function to give actuator setpoints. SIMware provides the vehicle steering type (car-like or Ackerman steering, tank-like or skid steering) and associated parameters such as wheel radius, wheelbase or skid separation. For car-like steering, the speed v and angular speed ω are transformed into a steering angle γ as

$$\gamma = \tan^{-1}(\omega L/v)$$

where L is the wheelbase length. For skid steering, the speeds v and ω are transformed into individual left and right wheel speeds ω_l and ω_r respectively as

$$\begin{aligned}\omega_l &= \tfrac{1}{R}(v - \omega L/2) \\ \omega_r &= \tfrac{1}{R}(v + \omega L/2)\end{aligned}$$

Due to contraints on a vehicle's minimum turning radius, J^{-1} may compute infeasible steering angles. For example, a mobility command with zero translational velocity and some non-zero angular velocity can only be executed by a vehicle with a zero

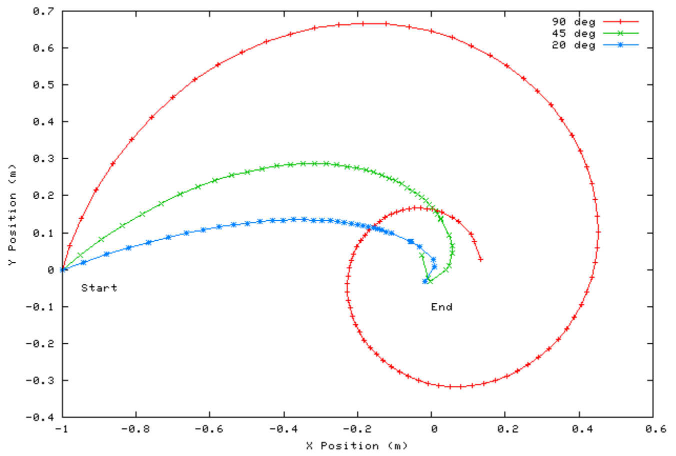


Figure 10. Trajectories for various values of the cutoff angle. Smaller cutoff angles inhibit translational motion, allowing rotation to point the vehicle more closely toward the goal before translation takes place. As the cutoff angle increases, the vehicle speed is greater, allowing for faster motion toward the goal but with broader deviation. If the cutoff angle is too great, the vehicle may endlessly "orbit" the goal, as seen in the 90-degree case.

turning radius, i.e., one that can spin in place. Depending upon system design, the Servo Level may choose to execute the tightest turn possible, or stop with a failure status. In either case the supervisory Prim Level must be prepared to handle deviations between its requested velocity states and those achieved by the Servo Level.

Even if the Prim Level assures that the Servo Level can always compute feasible actuator states, the vehicle may drift from its nominal expected path due to slipping. Because the Prim Level is commanding velocities in order to achieve a position goal, these drifts can accumulate and take the vehicle far from the Prim Level's target goals. For this reason, the Prim Level must close its own control loop. This is unlike trajectory planning for robot manipulators, where open-loop paths can be executed with confidence that the joint servos can make up for any errors. Prim is also responsible for planning motions of any mission packages mounted on the vehicle, such as pan-tilt wrists or robot manipulator arms. SIMware provides Prim with the kinematics parameters for the mission packages on board the vehicle, such as the link type (revolute or prismatic), link parent, link-to-link transforms and allowable travel limits [10].

Speed or position control are possible in either joint or Cartesian reference frames. Speed control typically arises from teleoperated motion using a joystick. Joint speed control is achieved simply by sending joint speeds to the Servo Echelon. Cartesian speed control is achieved by transforming the desired

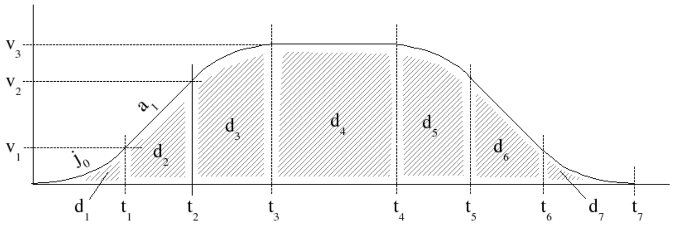


Figure 11. S-curve velocity profiling for position control. This graph shows speed versus time and the seven phases of motion. The first, third, fifth and seventh phases are constant jerk. The second and sixth phases are constant acceleration (or deceleration). The fourth phase is constant speed. Some phases may be of zero duration. S-curve motion planning is used for both translational and rotational motion.

Cartesian speed screw through the inverse Jacobian matrix,

$$\left[\dot{\theta}\right] = J^{-1}\begin{bmatrix} v \\ \omega \end{bmatrix}$$

Joint position control is achieved by computing positions for each joint according to an S-curve velocity-acceleration-jerk profile as shown in Figure 11, and scaling the joint speeds so that all joints arrive at their final position at the same time. At each time step, incremental joint positions are computed and sent to the Servo Echelon for tracking. Cartesian position control is done using S-curve profiling for both the translation and rotation portions of the move, where the rotation is considered as an angular displacement about a rotation vector. Either the translation or rotation portion is scaled so that both components arrive at their final target at the same time. At each time step, the intermediate pose is run through the inverse kinematics to get joint positions for tracking by Servo. The inverse kinematics are computed using an iterative Newton-Raphson procedure [11, 12]. An initial estimate of the joints is run through the forward kinematics to get an estimate of the Cartesian position. The error transform between the estimated and desired Cartesian positions is run through the inverse Jacobian to get correcting joint differentials, which are then subtracted from the joint estimates to move them toward the solution. The *R* and *P* rotations and translations for each link are obtained from the manipulator settings maintained by SIMware. Given these, we can incrementally build the forward Jacobian matrix as

$$J^{v}_{i+1} = \left[\begin{array}{c|c} {}^{i+1}_{i}R\left(J^{v}_{i} + J^{\omega}_{i} \otimes {}^{i}_{i+1}P\right) & \begin{matrix} 0 \\ 0 \\ 1 \end{matrix} \end{array}\right] \quad (1)$$

$$J^{\omega}_{i+1} = \left[\begin{array}{c|c} {}^{i+1}_{i}R\ J^{\omega}_{i} & \begin{matrix} 0 \\ 0 \\ 1 \end{matrix} \end{array}\right] \quad (2)$$

$$J_{i+1} = \begin{bmatrix} J^{v}_{i+1} \\ J^{\omega}_{i+1} \end{bmatrix} \tag{3}$$

where $\otimes$ in Equation (1) is the column-by-column cross product of the matrix J^{ω}_{i} with the position vector P^{i}_{i+1}.

Each control cycle the instantaneous Jacobian matrix J is computed out joint-by-joint using Equations (1-3) and inverted to get J^{-1}. If J is not square and hence not invertible, its pseudoinverse is computed. In the overdetermined case where there are fewer than six joints, the pseudoinverse that minimizes the least squares error between the desired and resulting Cartesian motion is used [13],

$$J^{I} = \left(J^{T}J\right)^{-1}J^{T} \tag{4}$$

In the underdetermined case where there are more than six joints, the pseudoinverse that minimizes the sum of the squares of the joint speeds is used,

$$J^{I} = J^{T}\left(JJ^{T}\right)^{-1} \tag{5}$$

The Prim sensor processing responsibility is to transform readings from vehicle-mounted sensors like range scanners from the local vehicle-centered coordinate system to the global world coordinate system. This is done by referencing the continual estimate of the vehicle's pose with respect to the world coordinate system as measured by sensors such as GPS, INS or wheel odometry and reported by the Servo Echelon through SIMware's MOAST superior skin.

4.3 AM-Prim

The Autonomous Mobility (AM) Echelon interfaces to the Prim Echelon using NML. Messages between AM and Prim are divided between mobility, mission package and sensor processing.

AM's mobility responsibility is to generate short-range obstacle-free paths for tracking by the Prim Echelon. Paths are comprised of lists of constant-curvature arcs, in which the end of one arc is tangent to the beginning of the next. This list is continually replanned at cycle times on the order of half a second. Strictly periodic replanning is not necessary, since Prim will follow the complete list of arc segments and stop at the end of the final arc. Replanning allows AM to compensate for Prim's drift off of the planned path and ensure that future paths remain free of obstacles. AM replanning also takes into account moving or newly-detected obstacles.

Figure 6 shows one such constant-curvature arc path generated by AM. The path is initially tangent to the vehicle's heading.

Mission package control by AM allows the pointing of sensors, for example pan/tilt mounts for cameras or ladars. Sensor pointing can be done via goal pan/tilt angles, for which paths are smoothly planned by Prim, or via scans, in which a region of interest is defined by AM. In scanning, Prim plans the raster paths or other suitable path types that provide sensor coverage over the region. Prim continually maintains the timestamped position of the sensor pointing angles, so that sensor outputs can be registered with AM's world coordinate system.

4.4 Vehicle-AM

The Vehicle (Veh) Echelon interfaces to the AM Echelon using NML. There is once again a breakdown of messages between primarily mobility, mission package, and sensor processing.

Veh's mobility responsibility is to create plans that accomplish a given activity. For example, drive to a given address.The Veh Echelon has the knowledge required to plan a path from the vehicle's current location to the given destination. However, since the Veh Echelon's world model is of lower resolution than AM's, it is felt that the AM Echelon mobility planner will create better plans for regions close to the vehicle. Therefore, the flow of the Veh-AM interaction is more complex than simply a superior ordering a subordinate to take an action that is later refined by the subordinate. In this interface, the following interaction occurs:

1. The Veh Echelon requests that AM prepare to move. This causes AM to compute a plan (and a cost for the plan) from the current vehicle location to each of its peripheral nodes. Where peripheral nodes are defined as regions located at AM's planning horizon and form an encompassing rectangle as shown by the dots surrounding the vehicle in Figure 7.
2. The Veh Echelon then uses these costs to "seed" its planning graph. In a traditional graph-based planning approach, either the goal location or the vehicle location is taken as the first open node or root for the search. This node is then the plan origin and will always be part of the plan. In this modified search strategy, our planning graph has many roots. Only one of these roots will be included in any given plan.
3. The successful root (the one contained in the lowest cost plan) is then sent back to AM as a commanded destination.
4. As AM executes this command, it re-plans the cost of reaching each of its new, shifted peripheral regions. The cycle then continues at Step 2.

Sensor processing at the Veh Echelon is required to take cellular map data as input and generate a vector representation that will be stored in the Veh WM and used by the planning system. This area is under active development, but the basic mechanism is to perform grouping on the AM obstacle regions and then reduce the groups to polygons. This is accomplished by forming the concave hull around each group. The newly formed polygons are then merged with any existing polygons with which they intersect.

A very simple high-level mission package interface exists for the Veh Echelon. Basically, the Veh wants a task to be performed, but does not specify how that task is to be accomplished. For example, a region may be specified for surveillance. It is up to the AM echelon to compute how best to perform the actual surveillance.

4.5 Section-Vehicle

The Section (Sect) Echelon is the highest echelon of MOAST that has been implemented to date. Sect operates as a pure finite state machine, and decomposes its high-level behaviors into a series of commands that are executed by it Veh echelon subordinate. It monitors vehicle status and command execution to know how to proceed through its state machine. Behaviors that have been implemented include multi-vehicle exploration, and multi-vehicle package delivery.

5 Future Work and Conclusions

The MOAST framework has been used to control virtual robots in both urban search and rescue environments and manufacturing settings, and a physical robot (automated guided vehicle) on a real shop floor. By utilizing the Player inferior skin of SimWare, identical algorithms that have been tuned in simulation are being experimented with on real hardware in identical environments. The idea is to validate performance in both the real and virtual worlds in order to verify simulated models and control system utility.

In addition, new algorithms are constantly being added to the framework. Work is progressing on Simultaneous Localization and Mapping (SLAM) as well as the inclusion of a true 3D world model. The MOAST website (www.sourceforge.net/projects/moast) highlights the latest improvements.

Disclaimer: No approval or endorsement of any commercial product by the National Institute of Standards and Technology is intended or implied. Certain commercial equipment, instruments, or materials are identified in this report in order to facilitate understanding. Such identification does not imply recommendation or endorsement by the National Institute of Standards and Technology, nor does it imply that the materials or equipment identified are necessarily the best available for the purpose.

REFERENCES

[1] USARSim. `www.sourceforge.net/projects/usarsim`.

[2] The Player Project. `playerstage.sourceforge.net`.

[3] Microsoft Robotics Studio. `msdn.microsoft.com/robotics`.

[4] Albus, J., 1991. "Outline for a theory of intelligence". *IEEE Transactions on Systems Man and Cybernetics,* **21**, pp. 473–509.

[5] Albus, J., 2000. "4D/RCS reference model architecture for unmanned ground vehicles". In Proceedings IEEE International Conference on Robotics and Automation, pp. 3260–3265.

[6] Balakirsky, S., and Scrapper, C., 2004. "Planning for on-road driving through incrementally created graphs". In Proceedings of the 7^{th} IEEE Conference on Intelligent Transportation Systems, pp. 183–188.

[7] Gazi, V., Moore, M. L., Passino, K. M., Shackleford, W. P., Proctor, F. M., and Albus, J. S., 2001. *"The RCS Handbook: Tools for Real-Time Control Systems Software Development"*. John Wiley and Sons.

[8] The Real-Time Control Systems Library. `www.isd.mel.nist.gov/projects/rcslib`.

[9] Scrapper, C. J., Proctor, F. M., and Balakirsky, S., 2007. "A simulation interface for integrating real-time vehicle control with game engines". In Proceedings of the ASME Computers in Engineering Conference, September 3-7 2007, Las Vegas, Nevada USA, pp. DETC2007–34495.

[10] Proctor, F. M., Scrapper, C. J., and Balaguer, B., 2007. "Run-time integration of robotic manipulators and their controllers". In Proceedings of the ASME International Mechanical Engineering Congress and Exposition, November 11-15 2007, Seattle, Washington USA, pp. IMECE2007–43362.

[11] Press, W. H., Teukolsky, S. A., Vetterling, W. T., and Flannery, B. P., 1992. *"Numerical Recipes in C: The Art of Scientific Computing, Second Edition"*. Cambridge University Press, ch. 9, pp. 362–368.

[12] Yang, Cao, Chung, and Morris, 2005. *"Applied Numerical Methods Using MATLAB"*. John Wiley and Sons, Inc., ch. 4, pp. 186–189.

[13] Ho, C. Y., and Sriwattanathamma, J., 1990. *"Robot Kinematics: Symbolic Automation and Numerical Synthesis"*, 1st ed. Ablex Pub.