Simulation vs. Real-time Control; with Applications to Robotics and Neural Networks

Joseph T. Wunderlich, Ph.D. EE
Elizabethtown College
Computer Engineering Program

Abstract

Simulations are often used to model real physical systems prior to electrical, mechanical, and computer hardware development. This allows engineers and scientists to experiment with various concepts before committing time and effort into hardware. Simulations can also be run concurrently with real-time systems to build knowledge of the environment that the real-time system is operating in, then provide feedback to the system to optimize its performance. For both of these types of simulations, the simulation must accurately model the real physical system. A comparison of simulations to real-time controlled physical systems is illustrated in this paper using several simple robotic and artificial Neural Network examples. The robotics examples show how real-time control of mobile robots and robotic-arms, and the resultant governing equations and software algorithms can provide several interesting simulation problems to overcome if the simulation is to accurately model the physical system. The Neural Network example demonstrates how computational speed and numerical precision can become an issue when comparing simulations to real-time Neural Network hardware. In general, comparison of simulations to real physical systems often enhances understanding of the underlying governing principles and equations, and results in simulations that accurately model the real world.

I. Introduction

A comparison of real-time controlled systems to computer simulations is made below by simultaneously discussing the design and development of each. Both of these engineering processes can be accomplished by performing the following steps:

1) Define problem
2) Simplify
3) Find governing equations
4) Build
5) Test (and rebuild as needed)

The "Define problem" step often involves a written problem definition accompanied by a conceptual sketch with some variables identified. For computer simulations, this can include sketching flow-charts and program-control-blocks. For real-time controlled systems, this may involve sketching mechanical free-body diagrams, and control-scheme diagrams (i.e., for open-loop or closed-loop control). It is also important to make an initial assessment of what information is available and what information needs to be found.

The "Simplify" step involves making assumptions and considering different approaches. The selection of control schemes, computer hardware, and programming languages can significantly effect the complexity, precision, and speed of both simulations and real-time controlled systems. These decisions should be made simultaneously for both environments to guarantee the most success in accurately modeling the real-time system with a computer simulation; this becomes even more significant if the real-time system is to interactively communicate with a concurrently running simulation.

The "Find governing equations" step involves identifying the fundamental principles that govern the behavior of the physical system being controlled and simulated. This may involve deriving new equations. Different approaches to solving the governing equations should also be considered. For real-time controlled systems, this may involve selecting a solution that provides the fastest real-time response. For computer simulations, the selection and implementation of a solution may be more dependent on available mathematical programming constructs and functions, and perhaps on a choice of available numerical techniques. However, care should be taken in ensuring that the selection of different solutions for these different environments does not cause discrepancies between the simulation and real-time system.

The "Build" step involves writing computer programs and assembling computers, electronics, and mechanical systems. The engineering of communication hardware and software may be required if the real-time system is to interactively communicate with a concurrently running simulation.

The "Test (and rebuild as needed)" step involves verifying the proper performance of systems by performing tests of hardware and software under various operating scenarios. This step can also involve hand-checking computations and assessing resultant data for realistic results (e.g., *order-of-magnitude* checks). It can also involve gathering empirical data from observing the real-time system performance, then modifying the simulation to create a more accurate model -- or possibly redesigning and rebuilding the real-time system. Assumptions made as part of the "Simplify" step may also need to be reconsidered.

This paper discusses three case studies to demonstrate the above methodology:

- Case study #1: Mobile robots in a constrained space
- Case study #2: Robotic-arm in an unconstrained space
- Case study #3: Neural Network

The first two case studies are from a course taught to juniors and seniors in Computer Engineering and Computer Science at Elizabethtown College. The course is titled "*Simulation & Modeling Physical Systems*". The first case study is a required semester project; the second case study is a lecture example. The third case study is also from a course taught to juniors and seniors in Computer Engineering and Computer Science at Elizabethtown College ("*Digital Design and Interfacing*"), and is taught as a lecture example with students given the opportunity to build Neural Network hardware during the laboratory part of the course.

II. Case study #1: Mobile robots in a constrained space

1) Define problem: The following problem was assigned to three groups of four students in the course: "*Simulation & Modeling Physical Systems*" at Elizabethtown College: [1]

"*Program a real-time controlled mobile robot to seek a light source in a four-foot by four-foot pen. A one-foot by one-foot square obstacle is located at the center of the pen. When the robot finds the light, it should stop and play a song. Also, write a simulation of the mobile robot. The simulated robot, light, and environment should model the real physical system as closely as possible. The real-time control program code is to control a robot constructed from a "Lego Mindstorm" programmable "RCX" block (supplied by professor). The following programming languages must be used by each group to program the real-time controlled robot:*

- *Group #1: Visual-Basic* [2]
- *Group #2: A variation of C ("Not Quite C")* [3,4]
- *Group #3: Standard RCX code* [5]

The location of the light, and the initial location and orientation of the robot must be chosen by mouse-click at the beginning of every simulation program run. The professor will select the light location and initial robot location and orientation for both your simulation and real-time robot demonstrations on the day of your presentation. Once all of the robots have been demonstrated to perform the above task, designate one of the robots to find the light, then search for the remaining two robots and notify them that the light has been found by sending an encoded message via IR communication; the remaining two robots will then acknowledge this message by playing a tune. The professor will select the light location and initial robot locations and orientations on the day of your presentation. A sketch of a robot, light-source, enclosure, and obstacle is shown in Fig. 1".

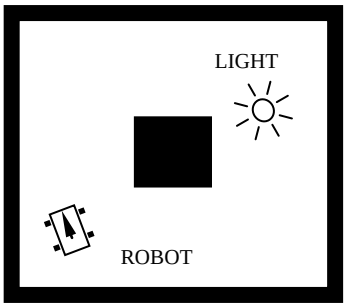


Figure 1. Sketch of a robot, light-source, enclosure, and obstacle.

An important part of defining the problem is to make an initial assessment of what information is available and what information needs to be found (or selected):

Known information:

- Dimensions of enclosure and obstacle.
- Robotic kits come with 2 motors, 2 touch-switches, a light sensor, and an IR comm port.

Information to find (or select):

- Knowledge of environment to be obtained through robot movement.
- A path-planning search algorithm (while avoiding obstacles) to find light.
- Selection of an open-loop or closed-loop control scheme.
- Programming language(s).

2) Simplify: The easiest way to simplify the real-time robot code is to not try to learn the environment, but simply *bounce* off the walls and obstacles while looking for the light. This involves putting the bump switches and light sensor on the front of the robot, and developing a search algorithm that prevents the robot from getting stuck by endlessly repeating unsuccessful paths. This results in a simple open-loop control scheme (i.e., no feedback).

The simulation should accurately model the motion of the real-time robot, however the following assumptions can be made (at the risk of introducing errors that need to be accounted for):

- Assume the robot always rotates about its geometric center.
- Ignore lack of wheel traction.

The simulation and real-time code can be further simplified by making assumptions about the light source. The sensitivity of the robot's light sensor provides for identifying a certain intensity of light at the sensor; however, the distance at which the light can be seen (i.e., the "range") varies depending on where the light is within a "cone of vision" defined by the boundaries of the sensor's peripheral vision (see Fig. 2). [2,3] At the risk of introducing discrepancies between the simulation and real-time robot's performance, the light can be assumed to be seen within a fixed distance anywhere within the "cone of vision" defined by the peripheral vision boundaries. Another way to further simplify this problem is to give the robot tunnel vision by affixing a tube in front of the light sensor; this reduces the need to define an accurate "cone of vision".[3]

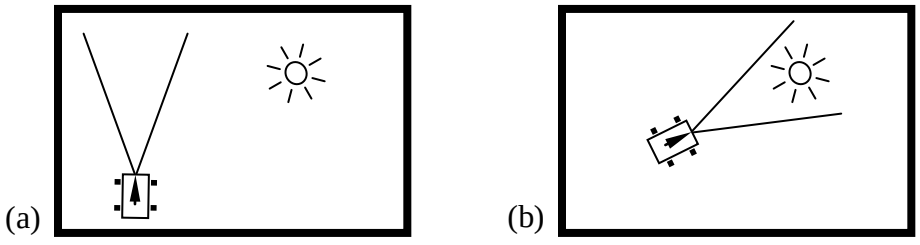


Figure 2. "Cone of vision" a) Light not within "cone" b) Light within "cone".

3) Find governing equations: A typical search strategy to find the light is:

- First rotate the robot about its geometric center to see if the light is in sight.
- Then begin moving forward until an obstacle is hit or a certain distance is traversed.
- If an obstacle is hit, backup and rotate the robot by some amount.
- If a certain distance is traversed, spin the robot again to see if light is in sight.
- Once the light is within the "cone of vision", move towards it.
- Stop and play a tune when the robot reaches a *short*, fixed distance from the light.

Since the real-time robot is assumed to <u>not</u> be gathering information about its environment, it has no means of knowing where it is in space; and the simulation should reflect this. The temptation to use knowledge of the robot and light locations prematurely in the simulation must be avoided. The simulation should only define a unit direction vector pointing at the light after it has actually found the light in the same manner that the real-time robot has. This vector is shown in Fig. 3 and is given by:

$$\vec{u} = \begin{bmatrix} \vec{u}_x \\ \vec{u}_y \end{bmatrix} = \begin{bmatrix} (x_{light} - x_{robot}) \Big/ \sqrt{(x_{light} - x_{robot})^2 + (y_{light} - y_{robot})^2} \\ (y_{light} - y_{robot}) \Big/ \sqrt{(x_{light} - x_{robot})^2 + (y_{light} - y_{robot})^2} \end{bmatrix} \qquad (1)$$

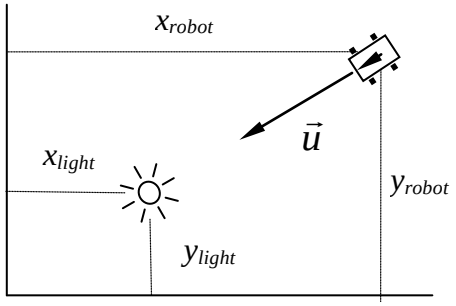


Figure 3. Unit vector defining direction from robot to light.

The governing equations for the motion of the real-time robot can be simply a function of wheel radii, drive-train gear ratios, and how much wheel rotation is attained per time that the motors are turned on; assuming *straight-forward* motion when both wheels are turned on in the same direction, and rotation of the robot about its geometric center when both motors are turned on in opposite directions.

The governing equations for the motion of the robot in the simulation are similar to that of the real-time robot with one major exception: the simulation must keep track of where the robot is and how it is oriented so that a graphical representation of the robot and its "cone of vision" can be drawn. This means that a coordinate system with respect to an origin must be defined; and the location and orientation of the robot identified with respect to it (see Fig. 4). The coordinate system is also needed to draw the light, enclosure, and obstacle.

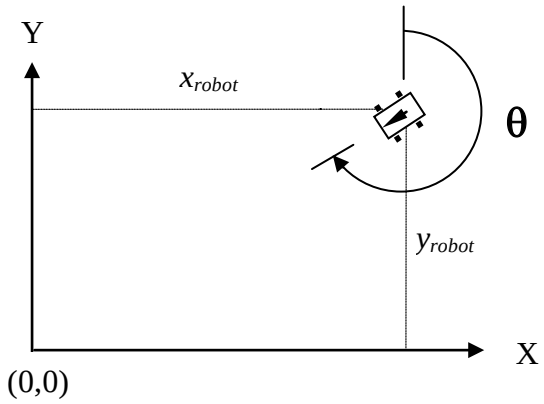


Figure 4. Some information needed for drawing robot graphics in simulation. (i.e., Coordinate system, robot position, and robot orientation).

4) Build: Building the real-time robot requires both programming and mechanical construction. No construction of computer hardware is required since the robot kits comes with an embedded microcontroller that is easily programmed via code developed on, then downloaded from, a PC. The kit's I/O ports also allow easy peripheral control. The mechanical design and construction requires design of:

- Gear-ratio's for the gears between the motors and wheels.
- The number and location of wheels for good robot maneuverability.
- The number and location of bump-switches for good robot maneuverability.
- The location, orientation, and modification of the light-sensor for good target-acquisition.

The complexity of the real-time robot code depends on the programming language chosen. The simple RCX programming environment supplied with the kits is entirely visual with *lego-block-like* program-code modules that are assembled like blocks; an extremely easy language to use, but very much lacking in programming power. Visual Basic and "Not-Quite-C" [4] languages allow much more direct control over the microcontroller within the robot, and provide the desirable programming constructs of powerful high-level languages.

The complexity of the simulation code also depends on the programming language used; Matlab (version 5) provides many powerful built-in functions for graphics and matrix manipulation. It is also a portable, compact way of getting good results quickly. [6] However, it lacks some of the powerful constructs of a high-level language (e.g., subroutines using global variables), and can run somewhat slow when executing complex code (e.g., iterative graphics, or numerical routines seeking convergence). For the simulated robot to appear to move on the monitor at the same velocity as the real-time robot, the velocity of the real-time robot motion should be either observed or calculated, then accurately simulated by putting appropriate delays in the simulation loop-graphics routines. The best way to this is by using timers in the code.

5) Test (and rebuild as needed): This step involves verifying the proper performance of the real-time robot and simulation by performing tests of hardware and software under various operating scenarios (i.e., by picking different light locations and different initial robot locations and orientations). It should also involve gathering empirical data from observing real-time robot performance, then modifying the simulation to more accurately model the real-time robot. This may involve reconsidering some of the assumptions made as part of the "Simplify" step, and can result in rebuilding the real-time robot after learning from many simulation runs (e.g., changing the design of wheels and gearing to provide more maneuverability, changing the light-sensor location and orientation for better target acquisition, or changing the real-time search algorithm to find the light more quickly).

The above problem was successfully completed by three groups of students [2,3,5] in the Fall, 2000 "*Simulation & Modeling Physical Systems*" course[1] at Elizabethtown College. On the day of demonstrations, the professor defined the light location and initial robot location(s) and orientation(s) as shown in Fig. 5.

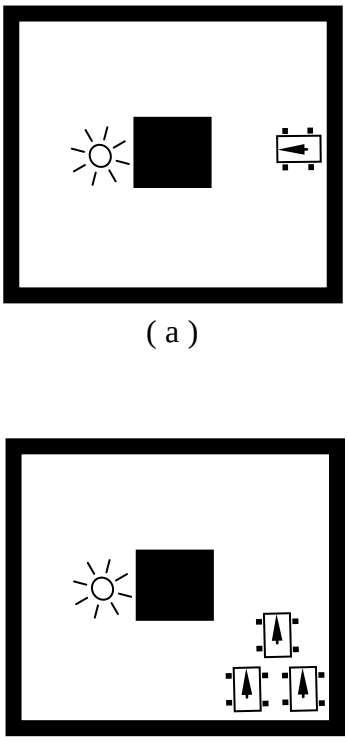


Figure 5. Light location and initial robot location(s) and orientation(s) defined by professor on day of demonstration. a) Single-robot task. b) Multi-robot task.

III. Case study #2: Robotic-arm in an unconstrained space

1) Define problem: The following problem is taught as a lecture example in the course: "*Simulation & Modeling Physical Systems*" at Elizabethtown College: [1]

"Design a robotic-arm to pick up a one gram metal object located 100 centimeters from it's base and lift it strait up at a rate of one centimeter per second to a height of 20 centimeters. The robot has a magnetic end-effector to pick up the object, and the object and robot are located in an obstacle-free environment. A sketch of the robotic-arm and object is shown in Fig. 6."

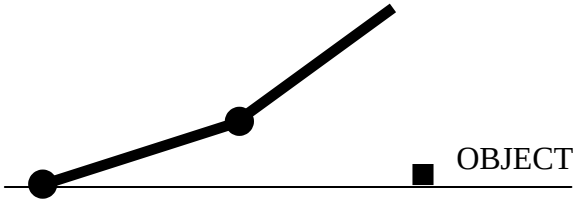


Figure 6. Sketch of robotic-arm and object.

Known information:

- Distance from the base to the object is 100cm.
- The object is to be moved at a velocity of 1 cm/sec.
- Gravity is 9.8 m/sec.

Information to find (or select):

- DOF ("Degrees of Freedom" -- i.e., number of links in robotic-arm).
- Length of each link.
- Joint angle displacements, velocities, or accelerations to command robot motion.
- Mass of each link.
- Knowledge of environment to be obtained through movement of robot.
- A robotic-arm motion algorithm.
- Selection of an open-loop or closed-loop motor control scheme.
- Programming language(s).

2) Simplify: Since the object and robot are located in an obstacle-free environment, we can simplify the real-time robot code by not designing the robot to learn its environment or to even keep track of the position of its joint(s). This results in a simple open-loop control scheme for the motors (i.e., no feedback). For robotic-arm control, we also need to select between three different types of *motion* control schemes:

1. Position-control (command joint-angle displacements)
2. Velocity-control (command joint-angle velocities)
3. Acceleration-control (command joint-angle accelerations)

Let's choose a velocity-control scheme since the problem statement asks for velocities. This will also result in a kinematic model instead of the dynamic model needed for acceleration-control. This simplifies the problem in that link-masses, gravity, momentum, and inertial effects can be ignored. For this problem, this is a valid choice since the robot is moving very slowly, and is lifting a very small mass. However, we therefore must also assume that the robot *never* moves at a velocity much greater than when it is moving the object. Let's also assume two degrees-of-freedom (2-DOF) is sufficient for performing the desired task. This will simplify the mathematics needed to model the system. If we also assume the robotic-arm will never be fully extended or contracted, we will eliminate the need to control mathematical "singularities" that occur at these undesirable robotic-arm configurations.

3) Find governing equations: The first step in deriving the governing equations is to draw a detailed diagram of the system as shown in Fig. 7.

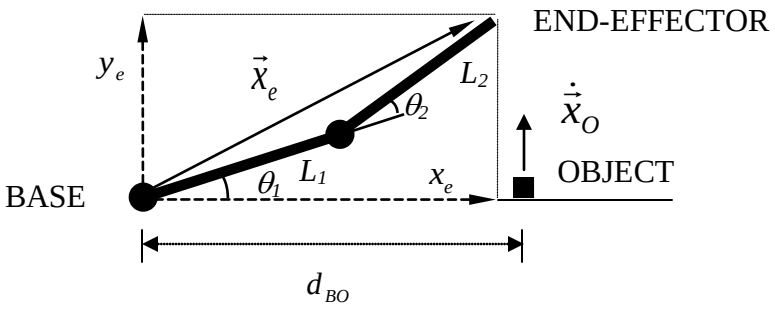


Figure 7. A two degree-of-freedom robotic-arm.

where $\vec{x}_e$ is the vector that locates the Cartesian position of the robotic-arm's end-effector with respect to the base. The end-effector position $\vec{x}_e$ is related to the joint angles by

$$\vec{x}_e = \begin{bmatrix} x_e \\ y_e \end{bmatrix} = \begin{bmatrix} L_1 \cos(\theta_1) + L_2 \cos(\theta_1 + \theta_2) \\ L_1 \sin(\theta_1) + L_2 \sin(\theta_1 + \theta_2) \end{bmatrix} \qquad (2)$$

and the end-effector velocity (i.e., the derivative of $\vec{x}_e$ with respect to time) is

$$\dot{\vec{x}}_e = d\vec{x}_e / dt = \begin{bmatrix} \dot{x}_e \\ \dot{y}_e \end{bmatrix} = \begin{bmatrix} dx_e / dt \\ dy_e / dt \end{bmatrix} \qquad (3)$$

where, using the "chain-rule" of differentiation for multi-variable functions,

$$\frac{dx_e}{dt} = \frac{\partial x_e}{\partial \theta_1}\frac{d\theta_1}{dt} + \frac{\partial x_e}{\partial \theta_2}\frac{d\theta_2}{dt} \quad (4)$$

$$\frac{dy_e}{dt} = \frac{\partial y_e}{\partial \theta_1}\frac{d\theta_1}{dt} + \frac{\partial y_e}{\partial \theta_2}\frac{d\theta_2}{dt} \quad (5)$$

we obtain:

$$\begin{bmatrix} dx_e/dt \\ dy_e/dt \end{bmatrix} = \begin{bmatrix} \partial x_e/\partial \theta_1 & \partial x_e/\partial \theta_2 \\ \partial y_e/\partial \theta_1 & \partial y_e/\partial \theta_2 \end{bmatrix} \begin{bmatrix} d\theta_1/dt \\ d\theta_2/dt \end{bmatrix} \quad (6)$$

or simply:

$$\dot{\vec{x}}_e = J_e \dot{\vec{\theta}} \quad (7)$$

where J_e is the "Jacobian" matrix:

$$\mathbf{J}_e = \begin{bmatrix} -L_1 \sin(\theta_1) - L_2 \sin(\theta_1 + \theta_2) & -L_2 \sin(\theta_1 + \theta_2) \\ L_1 \cos(\theta_1) + L_2 \cos(\theta_1 + \theta_2) & L_2 \cos(\theta_1 + \theta_2) \end{bmatrix} \quad (8)$$

which gives us our linear transformation between Cartesian end-effector velocities and robotic-arm joint-angle velocities. For simulating this system, we also need the Cartesian position of the robotic-arm's elbow with respect to it's base:

$$\vec{x}_{elbow} = \begin{bmatrix} x_{elbow} \\ y_{elbow} \end{bmatrix} = \begin{bmatrix} L_1 \cos(\theta_1) \\ L_1 \sin(\theta_1) \end{bmatrix} \quad (9)$$

To command the end-effector to perform a task in Cartesian space, we need to command joint-angle velocities. This is accomplished by manipulating equation (7) to form:

$$\dot{\vec{\theta}} = J_e^{-1} \dot{\vec{x}}_e \quad (10)$$

where J_e^{-1} is the inverse of J_e.

4) Build: Building the real-time robot requires both programming and electromechanical construction including construction of computer hardware for motor-control. The complexity of the real-time robot code depends on the programming language chosen; however a high-level language such as C or Fortran combined with the appropriate math library routines for matrix computation would be the best choice.

The complexity of the simulation code also depends on the programming environment chosen. Matlab is a good choice for the simulation since it provides many powerful built-in functions for matrix manipulation. [6]

To simulate this system, we first rewrite equation (10):

$$\begin{bmatrix} d\theta_1/dt \\ d\theta_2/dt \end{bmatrix} = \begin{bmatrix} \partial x_e/\partial\theta_1 & \partial x_e/\partial\theta_2 \\ \partial y_e/\partial\theta_1 & \partial y_e/\partial\theta_2 \end{bmatrix}^{-1} \begin{bmatrix} dx_e/dt \\ dy_e/dt \end{bmatrix} \quad (11)$$

then change our notation:

$$\begin{bmatrix} \Delta\theta_1/\Delta t \\ \Delta\theta_2/\Delta t \end{bmatrix} = \begin{bmatrix} \partial x_e/\partial\theta_1 & \partial x_e/\partial\theta_2 \\ \partial y_e/\partial\theta_1 & \partial y_e/\partial\theta_2 \end{bmatrix}^{-1} \begin{bmatrix} \Delta x_e/\Delta t \\ \Delta y_e/\Delta t \end{bmatrix} \quad (12)$$

where Δt is our simulation time step (i.e., the amount of computer computation time between successive evaluations of equation (12)). Δx_e and Δy_e are the incremental desired changes in the end-effector to achieve our desired task, and the $\Delta\theta$'s are the incremental changes in the robotic-arm's joint angles to achieve the desired end-effector movements. If Δt is controlled in simulation code by manipulating the time lapsed between successive plots of robot graphics, it can be removed from equation (12) to result in the following simulation-code algorithm:

$$\begin{bmatrix} \Delta\theta_1 \\ \Delta\theta_2 \end{bmatrix} = \begin{bmatrix} \partial x_e/\partial\theta_1 & \partial x_e/\partial\theta_2 \\ \partial y_e/\partial\theta_1 & \partial y_e/\partial\theta_2 \end{bmatrix}^{-1} \begin{bmatrix} \Delta x_e \\ \Delta y_e \end{bmatrix} \quad (13)$$

The simulation must be initialized by specifying link-lengths and initial joint-angles (or by specifying link-lengths and an initial end-effector position, then solving for the θ's in equation (2)). The algorithm for calculating each new joint angle position is

$$\begin{bmatrix} \theta_1 \\ \theta_2 \end{bmatrix}_{NEW} = \begin{bmatrix} \theta_1 \\ \theta_2 \end{bmatrix}_{OLD} + \begin{bmatrix} \Delta\theta_1 \\ \Delta\theta_2 \end{bmatrix} \quad (14)$$

To visualize the entire robot, we also need to track the position of the robot's elbow (using equation (9)) so that we can draw each of the robot's links; the first link is visualized by drawing a line from the robot's base to it's elbow -- the second link by drawing a line from the elbow to the end-effector. The simplest way to do this (for this simple 2-DOF robot) is to substitute each θ_1 found using equation (14) into equation (9). However, it should be noted that for robots with higher degrees of freedom, the elbows' Cartesian coordinates are tracked by computing separate linear transformations for each elbow joint. [7]

Once we have initialized the system, we simply command Δx and Δy in equation (13) to move to the end-effector to the object, then move straight up; where the simulated velocity is controlled through real-time clock timers in simulation graphics plotting routines, or specified time delays in the main loop of the simulation code (i.e., by controlling Δt in (12)). The robot can be repeatedly drawn by plotting lines from the base to the elbow, and from the elbow to the end-effector. The

velocity of the end-effector is controlled by manipulating a desired velocity and desired Cartesian step-size input by the user such that the timing of successive plots of the robot results in a visualization on the computer screen that moves at the desired velocity. This is accomplished by changing the Δx and Δy in equation (13) such that the resultant geometric length is equal to the desired step size; and the step is always taken along a unit vector from the present end-effector location to the target-location (i.e., the object or other specified goal). This unit vector is given by:

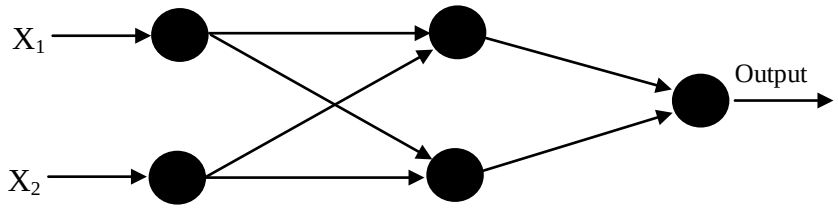
$$\vec{u} = \begin{bmatrix} \vec{u}_x \\ \vec{u}_y \end{bmatrix} = \begin{bmatrix} (x_{goal} - x_e) \Big/ \sqrt{(x_{goal} - x_e)^2 + (y_{goal} - y_e)^2} \\ (y_{goal} - y_e) \Big/ \sqrt{(x_{goal} - x_e)^2 + (y_{goal} - y_e)^2} \end{bmatrix} \quad (15)$$

5) Test: For this problem, we only need to verify that the real-time robot and simulated robot perform the same simple task equally well (i.e., go directly to the object and pick it straight up at the same specified velocity).

IV. Case study #3: Neural Network

1) Define problem: Computational speed and numerical precision can become an issue when comparing Neural Network simulations to real-time Neural Network hardware. The following problem is taught in the course "*Digital Design and Interfacing*" at Elizabethtown College; and is taught as a lecture example with students given the opportunity to build Neural Network hardware during the laboratory part of the course: [8]

"Design, simulate, and build a high-speed embedded back-propagation Neural Network [9,10,11] with two inputs, one hidden layer (with two neurons), and one output neuron (i.e., a "2-2-1" network) that performs the exclusive-or (XOR) of two binary inputs. A sketch of the Neural Network is shown in Fig. 8."



X_1	X_2	Output
0	0	**0**
0	1	**1**
1	0	**1**
1	1	**0**

Figure 8. Sketch of Neural Network to perform XOR.

Known information:

- Desired input/output pairs.
- Neural Network architecture.

Information to find (or select):

- Magnitude of interconnections weights.
- Programming languages.
- Type of embedded real-time hardware to implement:
    - Microcontroller.[12,13]
    - Digital integrated circuit.
    - Analog circuit.
    - Hybrid digital/analog circuit.
- Precision needed for computations.
- Where to do Neural Network "learning" (i.e., in simulation or in real-time hardware).

2) Simplify: The real-time hardware can be simplified by designing the Neural Network to have the minimum required computational precision needed to implement the XOR function. To accomplish this, it is important to recognize where the relatively *greater* precision is needed:

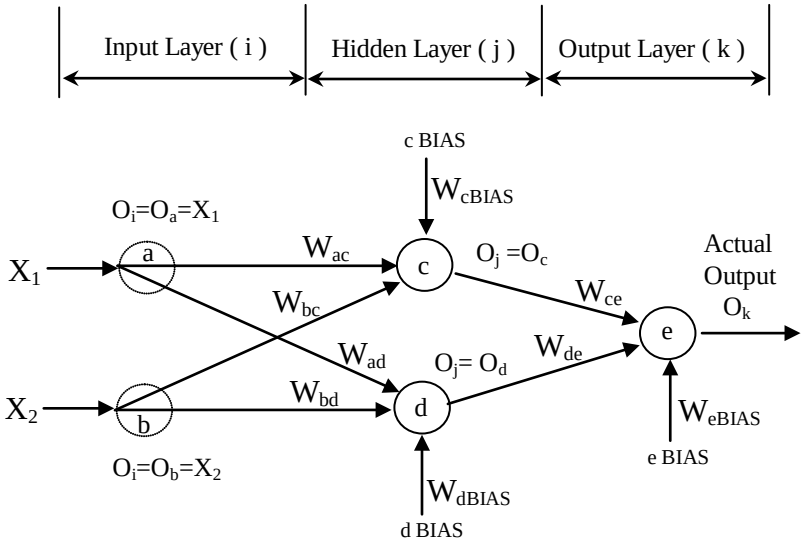
- For evaluation of each Neuron "transfer-function" (defined below).
- During the "learning phase" (defined below).

In general, a minimum of 16-bits of precision is typically used for these computations whereas as little as 8-bits can sometimes suffice for all other computations. [11,12] This observation will also help determine what role the simulation will have in the implementation (i.e., should Neural Network "learning" be done in real-time hardware or in the simulation).

We can simplify the governing equations below by assuming that although the real-time hardware must perform the XOR function as fast as possible, the time and number of back-propagation learning iterations to "teach" the network to perform the XOR does not need to be minimized.

This problem has been specified with the minimum Neural Network architecture (i.e., 2-2-1) to solve this problem (i.e., a non-linearly-separable two-input binary problem); therefore simplification of the architecture (i.e., by removing neurons) is not warranted. [9,10, 11]

3) Find governing equations: The first step in finding the governing equations for "backpropagation" Neural Network learning (also referred to as the "generalized delta rule"), is to draw a detailed diagram of the Neural Network as shown in Fig. 9.



Input / Output "**Exemplars**"

X_1	X_2	Desired Output (d_k)
0	0	**0**
0	1	**1**
1	0	**1**
1	1	**0**

Figure 9. Detailed diagram of Neural Network.

The desired outputs paired with their corresponding inputs are referred to as "exemplars". The process of Neural Network "learning" involves repeatedly feeding the network these exemplars; each time changing interconnection weights as a function of a "back-propagated" error between the desired output and the actual output, until the desired outputs are observed. This process is detailed as follows: [9,10,11,12]

1) Choose small random initial values for all interconnection weights (W's), and choose BIAS values (typically set to 1).
2) Feed the input layer an input vector (X_1, X_2) from the first exemplar.
3) Propagate the signals forward via the non-linear neuron transfer functions:

$$O_j = \frac{1}{1+e^{-(jBIAS*W_{jBIAS})+\sum_i(-O_i*W_{ij})}} \qquad (16)$$

$$O_k = \frac{1}{1+e^{-(kBIAS*W_{kBIAS})+\sum_j(-O_j*W_{jk})}} \qquad (17)$$

4) Create an error signal from the difference between the actual and desired output signal, and use it to change the connection weights between the output layer (k) and the hidden layer (j), and also between the output neuron and bias:

$$\Delta W_{jk} = \eta * \left[(d_k - O_k) * O_k * (1 - O_k) \right] * O_j \qquad (18)$$

$$\Delta W_{kBIAS} = \eta * \left[(d_k - O_k) * O_k * (1 - O_k) \right] * kBIAS \qquad (19)$$

where η is the learning rate (typically set to between 0.01 and 1).

5) *Back-propagate* a weighted error signal from the hidden layer (j) to the input layer (i) and use it to change the connection weights between the hidden layer (j) and the input layer (i), and also between the hidden layer neurons and bias':

$$\Delta W_{ij} = \eta * \left(O_j * (1 - O_j) \right) * \sum_k \left[(d_k - O_k) * O_k * (1 - O_k) * W_{jk} \right] * O_i \qquad (20)$$

$$\Delta W_{jBIAS} = \eta * \left(O_j * (1 - O_j) \right) * \sum_k \left[(d_k - O_k) * O_k * (1 - O_k) * W_{jk} \right] * jBIAS \qquad (21)$$

6) Repeat steps 1 to 5 for each exemplar.
7) Repeat steps 1 to 6 until the desired outputs have been *approximately* obtained.

4) Build: For our problem, the main function of the simulation is to find (i.e., "learn") the Neural Network interconnection weights to be implemented in the real-time Neural Network hardware; this is because of our assumption (i.e., design goal) to simplify the real-time hardware. The simulation is well suited for "learning" since the microprocessor in the computer running the simulation has the required computational precision to allow Neural Network back-propagation learning to work. After learning the Neural Network interconnection weight values, the precision required for the Neural Network to function is not as stringent. [10,11,12] Therefore the interconnection weight values can be rounded as shown in Table 1. We have also found that our circuit (after learning completed) will still properly function when all computations of equations (16) and (17) are truncated to one decimal place. These *reduced-precision* numbers will result in a simpler hardware implementation.

Table 1. Neural Network interconnection weights for XOR simulation and real-time hardware.

Neural Network Interconnection Weight	Initial Specified-Value (for Simulation)	Learned-Value (from Simulation)	Rounded *Learned-Value* to use for Real-Time Hardware
W_{ac}	0.50000000000000	-6.23906875623139	-6.2
W_{bc}	0.70000000000000	-6.24027721135562	-6.2
W_{cBIAS}	1.00000000000000	8.72789554926565	8.7
W_{ad}	0.60000000000000	5.49914908963778	5.5
W_{bd}	0.80000000000000	5.56853185491428	5.6
W_{dBIAS}	1.00000000000000	0.901408728088887	0.9
W_{ce}	0.90000000000000	9.55840499686730	9.6
W_{de}	1.00000000000000	41.7820655518547	41.8
W_{eBIAS}	1.00000000000000	-45.8904383750475	-45.9

The simulation that produced the data in Table 1 implemented the Neural Network of Fig. 9 with (cBIAS = dBIAS = eBIAS = 1.0), a relatively large learning rate ($\eta = 5.0$), and stopping-criteria of (0.990 ~ 1.0) and (0.020 ~ 0.0). For this simple problem, Neural Network learning time was not a significant issue; the network learned with only 2829 cycles of the four exemplars through the learning process (i.e., learning "epochs") as shown in Fig. 10 (with every 100th epoch plotted). It is important to note that the learning time for a Neural Network can become a very significant issue for more complex problems -- and that designing Neural Network hardware with relatively high-precision learning performed in real-time hardware can become a significant undertaking.[12]

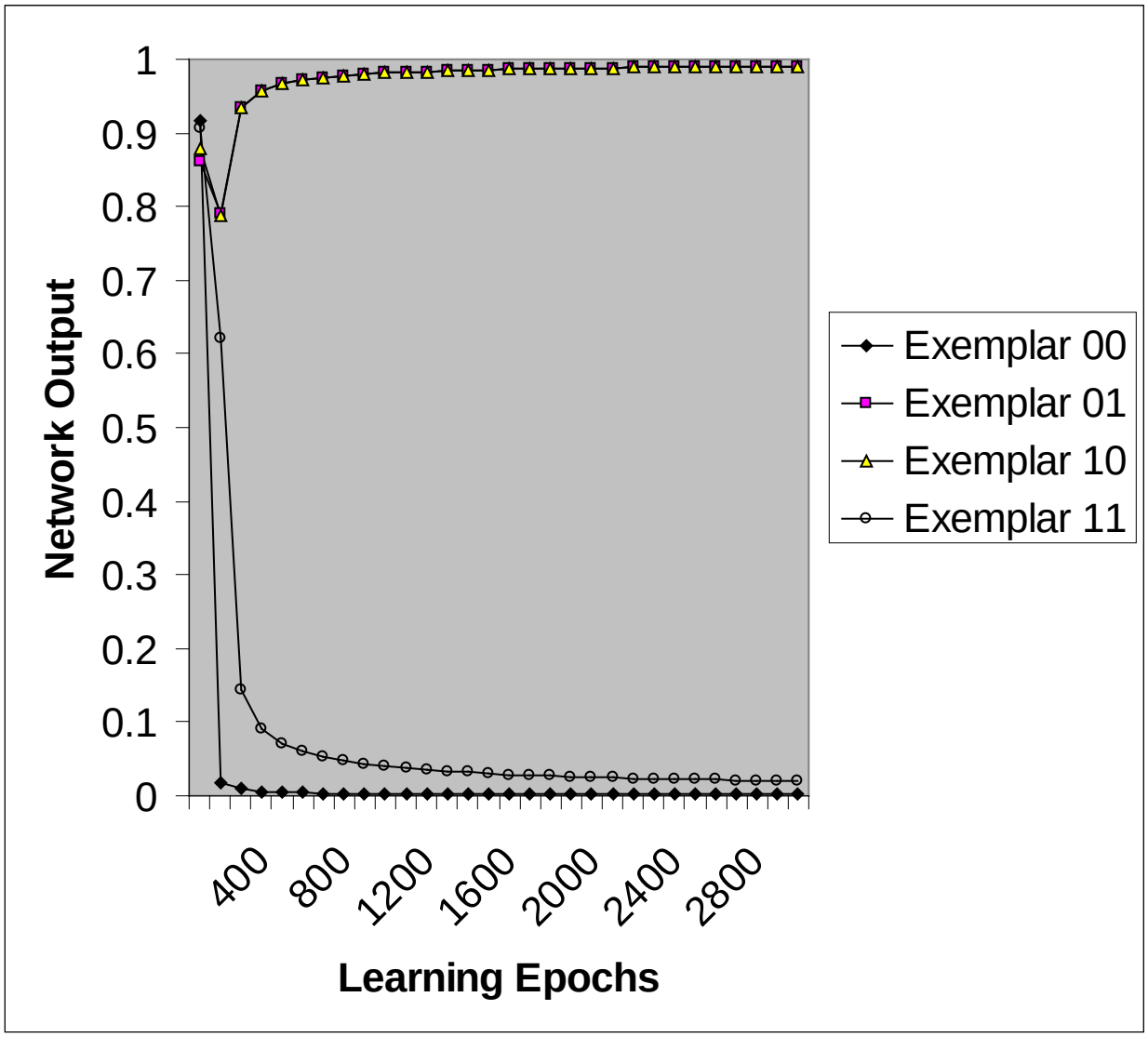


Figure 10. 2-2-1 Neural Network learning to perform XOR function (every 100th epoch plotted).

A high-speed real-time hardware implementation of this simple Neural Network can be created in one of several ways:

- An embedded microcontroller for all computations, with the transfer function implemented as either an analog circuit or a look-up table.
- Medium-Scale-Integration digital circuits (FPGA or discrete IC chips) for all computations, with the transfer function implemented as either an analog circuit or a look-up table.
- An embedded microcontroller for all computations including a polynomial approximation of the transfer function.[12]
- Using all discrete analog components.

Test: Testing of our simulation involves taking the Neural Network interconnection weights obtained from the last exemplar in the "training-set" during the last training epoch, and feeding it forward through the Neural Network for each of the exemplars to verify desired output is maintained. This is necessary since every pass of every exemplar during training changes the interconnection weights.

Testing of our real-time Neural Network hardware depends on the type of implementation:

- Testing an embedded microcontroller implementation involves testing the code with an assembly-code simulator before embedding it in the microcontroller; Once the code is embedded, the network is tested by feeding the microcontroller each of the exemplars (via toggle switches connected to one of the microcontroller's I/O ports) to verify that desired outputs are obtained (via LED's connected to one of the microcontroller's I/O ports).
- Testing a Medium-Scale-Integration digital circuit implementation (FPGA or discrete IC chips) or an analog circuit implementation involves feeding the network each of the exemplars to verify desired outputs are obtained. A digital or analog circuit simulation may also be created and tested prior to building the hardware.

It is important to note that testing more complex Neural Network hardware implementations (especially those with complex instruction-sets) can involve all of the steps involved in testing new computer hardware: [14,15]

1. Architectural verification programs run in a simulated prototype-machine environment.
2. Digital and analog VLSI circuit simulation testing.
3. Machine architectural verification programs run on top of a VLSI circuit simulation.
4. Machine architectural verification programs run on prototype hardware.
5. Various instruction-mix and performance benchmark testing.

V. Conclusions

A comparison of simulations to real-time controlled physical systems using simple robotic and artificial Neural Network examples enhances the understanding of how to engineer both well-designed real-time hardware aided by simulation, and carefully programmed computer simulations that accurately model real physical systems. The robotics examples show how modeling real-time robot movement can provide interesting simulation problems to solve; and the Neural Network example demonstrates how computational speed and numerical precision can become an issue when comparing simulations to real-time Neural Network hardware. The above comparison of real-time systems to computer simulations has been made by simultaneously discussing the design and development of each -- with both following five simple design steps: "Define Problem", "Simplify", "Find Governing Equations", "Build", and "Test". In general, comparison of simulations to real physical systems often enhances understanding of the underlying governing principles and equations, and results in simulations that accurately model the real world.

Bibliography

1. URL: http://users.etown.edu/w/wunderjt; CS/ENGR 344: Simulation/Modeling Physical System course syllabus, Elizabethtown College, Fall 2000.
2. Lister, M., Simione, D., Bopp, M., and Crawford, T., Simulation and real-time control of HewEbot: an Elizabethtown College mobile robot, CS/ENGR 344 semester project report, Elizabethtown College, Fall 2000.
3. Allen, W., Nikles, D., Kaplan, E., and Winters, J., Simulation and real-time control of LewEbot: an Elizabethtown College mobile robot, CS/ENGR 344 semester project report, Elizabethtown College, Fall 2000.
4. Baum, D., *Definitive guide to Lego Mindstorms (and Not-Quite-C)*. Emeryville, CA: Apress, 2000.
5. Pittinger, B., Drill, T., Glasby, W., and Shank, K., Simulation and real-time control of DewEbot: an Elizabethtown College mobile robot, CS/ENGR 344 semester project report, Elizabethtown College, Fall 2000.
6. Palm, W., *Matlab for Engineering Applications*. Boston, MA: McGraw-Hill, 1999.
7. Wunderlich, J. T., and Boncelet, C. G., Local optimization of redundant manipulator kinematics within constrained workspaces, in *Proc. of IEEE Int'l Conf. on Robotics and Automation*, 1996, Minneapolis, MN.
8. URL: http://users.etown.edu/w/wunderjt; CS/ENGR 333: Digital Design and Interfacing course syllabus, Elizabethtown College, Fall 2000.
9. Widrow, B., 30 years of adaptive neural networks: perceptron, madaline, and backpropagation, *Proc. 2nd IEEE Intl. Conf. on Neural Networks*, 1990.
10. Rumelhart, D.E., and McClelland, J. L., *Parallel Distributed Processing*. Cambridge, MA: M.I.T. Press, 1986.
11. Soucek, B., *Neural and Concurrent Real-Time Systems, The Sixth Generation*. New York: John Wiley & Sons, 1989.
12. Wunderlich, J. T., *Design of a Neurocomputer Vector Microprocessor (with on-chip learning),* Masters thesis, Pennsylvania State University, Jan. 1992.
13. Wunderlich, J. T., Focusing on the blurry distinction between microprocessors and microcontrollers, *in Proc. of ASEE Nat'l Conf.*, 1999, Charlotte, NC.
14. Wunderlich, J. T., Random number generator macros for the system assurance kernel product assurance macro interface , *Systems Programmers User Manual for IBM S/390 Systems Architecture Verification*, IBM S/390 Hardware Development Lab, Poughkeepsie, NY, 1997.
15. Wunderlich, J. T., Branch-prediction verification of S/390 processors, IBM S/390 systems architecture verification department report, IBM S/390 Hardware Development Lab, Poughkeepsie, NY, 1996.

Dr. JOSEPH T. WUNDERLICH
Dr. Wunderlich is an Assistant Professor of Computer Science and Computer Engineering at Elizabethtown College. He came to Elizabethtown from Purdue University where he was an Assistant Professor of Electrical and Computer Engineering Technology. Prior to that he worked as a researcher and hardware development engineer for IBM on large-scale multiprocessor computer systems. Dr. Wunderlich received his Ph.D. in Electrical and Computer Engineering from the University of Delaware, his Masters in Engineering Science/Computer Design from The Pennsylvania State University, and his BS in Engineering from the University of Texas at Austin.