5 DOF Robotic Arm

ECE 478/578 Embedded Robotics

Fall 2009

Portland State University

Matt Blackmore

Jacob Furniss

Shaun Ochsner

Project Description

This project involves designing and building a mobile robot capable of human-like behaviors. The means to which these behaviors are achieved are contained in three aspects of the robot's design. The first is a mobile base that allows the robot to navigate through an indoor space. The second is a head and neck capable of producing several human-like gestures coordinated with speech. The third is an arm and hand assembly used to supplement the robot's gestures and allow it to grasp and move objects. Each of these design aspects is integrated to complete the first phase of the project and produce a platform for further research. Subsequent phases will involve using existing artificial intelligence, vision, speech, and dialog software to expand the capabilities of the robot. Also, the robot will be used to research a new type of artificial neural network at Ohio University.

This paper details the design and development of the arm and hand assembly within the first phase. The arm and hand, henceforth referred to as arm, are designed to meet the following requirements. First, it must have the ability to grasp an object and place it in a different location. Second, it must be similar in scale to that of a human arm and be able to replicate similar motions. The final design should be made with standard components, such that it could be easily reproduced and mirrored to create left and right versions. Finally, the arm should be easily mounted to the mobile base.

The scope of the first phase of the arm's development is defined by the following limitations. The arm is built with as many un-modified components as possible. In addition to reducing the amount of time necessary for fabrication, this facilitates the final design being easily reproduced. The ability of the arm to move an object is primarily symbolic. With this in consideration, an object with a negligible mass is chosen. This reduces the torque requirements for the arm's actuators and allows the overall design to be built to a larger scale. To simplify the software design, each joint on the arm will be manually controlled. Although future phases of the project will likely require more sophisticated motion control, this simplified approach allows the robots actuators and ranges of motion to be controlled and tested in a straightforward way. As a first step to higher levels of motion control, the forward kinematic equations for position are included in the first phase.

With these limitations in mind, the details of the projects are broken down into the following. The mechanical design of the arm deals with its physical construction and range of motion of each joint. System modeling relates the position of the hand to the angles of each joint in the arm (kinematics). Software design includes the programming methods for commanding actuators to move a joint to a specified position in addition to a description of the programming environment. The electrical components encompass the hardware necessary to control and power the system, actuators, and the devices to send command signal to them. The first step is to complete the mechanical design, the details of which are described below.

Mechanical Design

This project involves using an existing head and neck and modifying an existing mobile base. The arm, however, is designed and built from scratch. For this reason, the majority of work on the arm in the first phase revolves around its mechanical design and construction.

The first step in the mechanical design of the arm is to define its degrees of freedom. A degree of freedom, or DOF, is an independent displacement associated with a particular joint. Joints can be ether prismatic or revolute, or both. Prismatic joints are capable of linear motions while revolute joints are capable of rotating. In this case each of the arm's joints is revolute, and thus, each degree of freedom is a rotation. Each of these DOFs is controlled by an actuator.

The human arm is considered to have seven degrees of freedom. These consist of three rotations at the shoulder, one at the elbow, and three rotations at the wrist. The actuators that control the shoulder and, to a lesser degree, the elbow have to carry the load of the entire arm, hand, and payload. These actuators must be capable of producing substantially greater torque than actuators at other joints. To reduce the number of high-torque actuators required, the shoulder is designed with only two DOFs. Although the wrist does not have to carry a high load like the shoulder, space at this point on the arm is limited. For this reason, the wrist is given only two DOFs. This leaves a total of five degrees of freedom for the arm instead of seven. The human hand has twenty seven degrees of freedom, most of which are associated with the fingers. To grasp a simple object, the motions of the fingers are not needed. This assumption allows the hand to be designed with one degree of freedom, thus greatly simplifying the design. A simple representation of the arm is shown in the Figure 1 below. The red arrows represent the axis that each DOF can rotate about. Although the hand is shown, its DOF is not labeled.

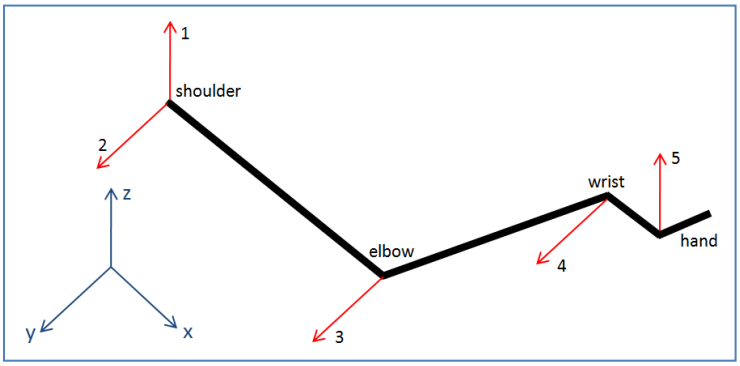


Figure 1: Robot arm's degrees of freedom.

As mentioned above, it is important that the final robot design be easy to reproduce and mirror. This is facilitated by using TETRIX components whenever possible. TETRIX is a component system originally designed for use in high school robotics competitions. The system consists of a variety of prefabricated aluminum components that are designed to be easily modified and connected to one another. Also included are high torque DC gear motors, servos, and motor drivers. These components are compatible with the LEGO Mindstorms system. The LEGO system not only includes components for building robots, but includes a series of Plug 'N Play

sensors and peripherals in addition to a controller and programming environment. Together these systems allow a designer to quickly build robot prototypes with little or no fabrication. The details of the LEGO controller, programming environment, and electronic components are described in later sections. Figure 2 shows the basic TETRIX robotics kit.

Figure 2: TETRIX robotic kit.

Although the use of TETRIX components reduces the effort and time required to design and build the system, not all of the components were initially available. Thus, the arm needed to be designed before the components were acquired. The Solid Works CAD tool was used to accomplish this. Solid Works is a modeling and simulation environment capable of representing three dimensional shapes in space in addition to material properties. An online CAD library was used to acquire models of most of the TETRIX and LEGO components. These individual models are combined in an assembly that defines the spatial and kinematic relationships between them. The resulting virtual assembly is used to evaluate the moments of inertia, mass, volume, physical dimensions, etc. at a component or system level. Also, this assembly is used to simulate motion between the components. This allows the designer to check for collision between parts, analyze the range of motion of the entire system, and visualize its performance before anything is physically made. Approximately eight design iterations were investigated with this tool before parts were ordered and very little was changed from the CAD model once it was actually built. Figure 3 shows the final model of the arm without any custom fabricated parts.

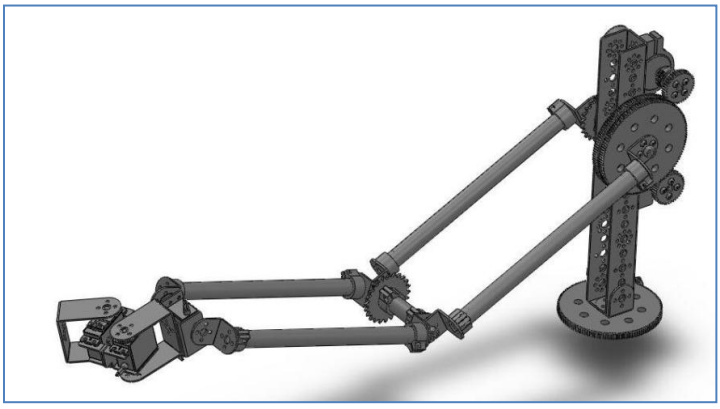


Figure 3: Solid model of robot arm and hand assembly.

This model does not include some of the hard ware necessary to complete the assembly in addition to the hand. Figure 4, shows the complete TETRIX assembly and hand.

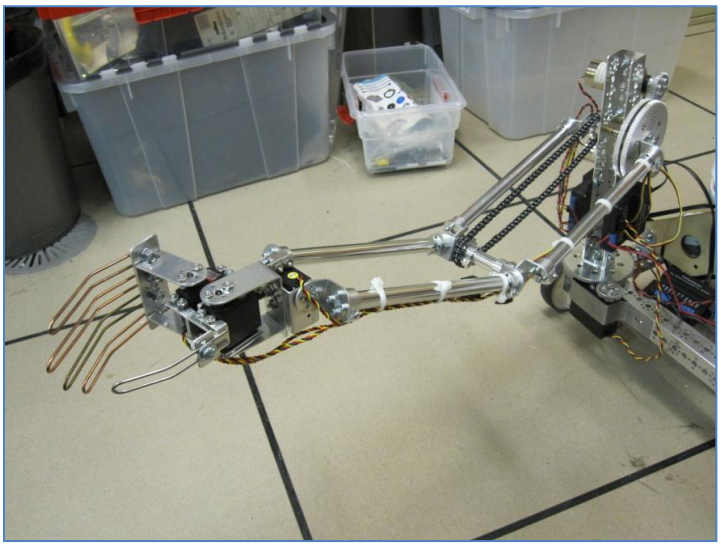


Figure 4: Final TETRIX robot arm and hand assembly.

In addition to the stock TETRIX components, welding rod was used to fashion the fingers of the hand. The plate that the fingers attach to was also fabricated. A list of the modified TETRIX components is in the appendix.

Electronic Components

The electronic components used to operate the arms consisted of two electronic motors, four electronic servo motors, one motor controller, one servo controller, and the NXT brick. The motors were used on the elbow and shoulder joints to provide more torque and stability while servo motors were used to control the hand, wrist, and arm rotation. All these components are part of the Lego Textrix Robotics division. Using the Tetrix parts along with the NXT brick allowed for less time spent integrating and developing drivers, because when programmed with RobotC, the drivers and control functions are already integrated into the system allowing for more of a plug-and-play environment. This saved time in developing code for controlling the arm.

The main control of out arm is done by the NXT brick. This control unit is run by a 32bit ARM7 microprocessor and an 8 bit AVR microcontroller. It has 4 six wire input ports, and 3 six wire output ports. It also contains a USB port for programming and debugging. It is mainly programmed using the NXT graphical interface language, LabVIEW, RobotC, or NXT++. We chose to use RobotC, which is a subset of the C programming language since that is what our group was

the most familiar with. This will be discussed further later on in the report. The RobotC interface allowed us to download and run programs on the NXT unit, and once downloaded could be run directly from the NXT without needing to be hooked to a computer. For our application we were using Tetrix products to interface with the NXT we ran all our components from Sensor Port 1 of the NXT. The NXT allows up to four controllers to be daisy chained to each sensor port. These controllers can be a combination of servo controllers and motor controllers which will be discussed later. Any sensor that will be used for additions for arm control will also be plugged in to the NXT.

The motors we used were Textrix DC motors available from Lego Robotics. The motors run at 152rpm at full power and provide 300oz-in torque and require 12V to operate. Within the software the speed can be controlled by setting the percentage of the motor speed to lower the RPM of the shaft. This gives the motors more versatility when used in projects where more torque than can be provided by a servo is needed, but the slower speed of the servo is still desired. This was useful in our application a servo motor would not have been able to hold up the weight of our robotic arm, but we still needed slower movement for a more realistic appearance and allow more control for the user. The disadvantage of using motors in this situation is they are heavy and more difficult to mount than a servo would be. We installed encoders for position control, but we did not use them for this part of the project. The operation of the encoders will be talked about later in the report.

The motors are powered and controlled using a HiTechnic DC motor controller. This motor controller interfaces the motor with the NXT brick as well as providing power to the motor itself. Each motor controller can operate two 12V Tetrix motors as well as interface with motor encoders which will be discussed later. It is this motor controller that allows the motor speed to be adjusted by changing the power level supplied to the motor by using an internal PID algorithm.

Encoders are installed on the two motors used on the robot. These encoders are made by US Digital. They are used to allow position control of the motors so they can perform similar to servos. The encoders used are optical quadrature encoders. These encoders use two output channels (A and B) to sense position. Using two code tracks with sectors positioned 90 degrees out of phase, the two output channels of the quadrature encoder indicate both position and direction of rotation. If A leads B, for example, the disk is rotating in a clockwise direction. If B leads A, then the disk is rotating in a counter-clockwise direction. The encoder also allows the system to use PID control to adjust the speed of the shaft.

The servo motors used were three HS-475HB servos and one HS-755HB all made by Hitec. Both servos are 3 pole with karbonite gears that can be run at 4.8V or 6V. The 475HB provides about 80 oz-in of torque and the 755HB provides 183 oz-in of torque. The 755HB is a larger servo than normal is used with the Tetrix system, but the servo wire is the same for both servo types, so they can both be used with servo controller. The downside of this servo type not being available for the Tetrix system is that there is not mounting hardware available so a mount had to be fabricated to attack the servo to the Tetrix stock parts. The servos have a range of 0 to 255 so they give you excellent position control. The motors inside the servo only hold position when powered so when the power is removed any weight bearing servos release. The wrist on the robot is an example of this. When the program is running the wrist servo supports the hand, but as soon as power is removed or the program is ended the hand falls to one of the servo extremes.

Like the motors, in order to interact with the NXT device the servos must attach to a HiTechnic servo motor controller. The servo controller requires a 12V supply and it divides this down to 6V to operate the individual servos. The servo controller can hold up to six servos together, and like the motor controllers the can be chained together to allow the use of more servos than on controller could handle.

Software

The programming of the arm movement was done using RobotC language. RobotC was developed by Carnegie Mellon University and is a subversion of the C programming language. It uses the same syntax as C but a does not have access to the same libraries, so the command availability is somewhat limited. There are specific libraries for some aftermarket parts, and libraries can be made to incorporate new parts for use with the NXT. The PSP-Nx-lib.c library was used in order to use as PS2 controller to operate the arm.

The software to control the hand can be broken up into three sections: controlling the DC motors, controlling the servos, and integrating the PS2 controller. The code for each was tested prior to being compiled into the final program. We will start with describing how the DC motors are programmed, followed by the servos, the controller integration, and finally how the finished control program works.

The software allows the DC motors to be turned on, turned off, set the power level as well as allowing encoders to be used. In order to use the motors the configuration coding should be entered at the top of the top of the program. The line “#pragma config(Hubs, S1, HTMotor, HTMotor, HTServo, none)” sets sensor port 1 (S1), and configures it to have two motor controllers and one servo motor controller chained together. After that each hub must be set. To

configure a motor you use the line “#pragma config(Motor, mtr_S1_C1_1,   motorD, tmotorNormal, openLoop, reversed, encoder)” this line sets the first controller (C1) on sensor port 1 (S1) as a DC motor plugged into the motor 1 slot. The command motorD, sets the name of the motor to be used in the program (motorA, motorB, and motorC are designated for NXT motors) and tmotorNormal sets the motor in normal mode. The motor can be set in openLoop or PID to use the internal PID controller. The PID mode can only be used if an encoder is attached to the motor and activated. The motors can also be switched between forward and reversed modes in this line. Once these lines at entered it allows you to use motor control commands. The following code is a sample motor program:

```
#pragma config(Hubs,  S1, HTMotor,  HTServo,  none,     none)
#pragma config(Motor,  mtr_S1_C1_1,     motorD,
tmotorNormal, openLoop)

task main()
{
      motor[motorD] = 75;    // Motor D is run at a 75 power
level.
      wait1Msec(4000);       // The program waits 4000
milliseconds

      motor[motorD] = 75;    // Motor D is run at a 75 power
level.
      wait1Msec(750);        // The program waits 750
milliseconds
}
```

The code runs the motor forward for 40 seconds and backwards for 7.5 seconds.

Servos are programmed in a similar way. The hub must be configured for a servo controller in one of the spots. The line “#pragma config(Servo, srvo_S1_C3_1,   , tServoNormal)” sets the third controller (C3) on sensor port 1 (S1) as a servo plugged into the servo1 slot. Unlike motors the tServoNormal command is the only command that needs to be entered, but an empty placeholder spot still have to be left. The following code is a sample servo program.

```
#pragma config(Hubs,  S1, HTServo,  HTServo,  none,     none)
#pragma config(Servo,  srvo_S1_C1_1,    ,
tServoNormal)

task main()
{

  while(true)
  {
     if(ServoValue[servo1] < 128)          // If servo1 is closer
to 0 (than 255):
    {
      while(ServoValue[servo1] < 255)  // While the ServoValue of
servo1 is less than 255:
      {
        servo[servo1] = 255;                    // Move servo1
to position to 255.
      }
    }
```

```
    wait1Msec(1000);                                  // Wait 1
second.
    if(ServoValue[servo1] >= 128)         // If servo1 is closer
to 255 (than 0):
    {
      while(ServoValue[servo1] > 0)      // While the ServoValue
of servo1 is greater than 0:
      {
        servo[servo1] = 0;                             // Move servo1
to position to 0.
      }
    }
    wait1Msec(1000);                                 // Wait 1 second.
  }
}
```

This program reads the servo value and moves it to the closest end stop.

The controller we used required the add on library "PSP-Nx-lib.c" to make the buttons resond properly. A wireless PSP controller was used to control the robot using one button to control each degree of freedom. The layout of the controller buttons is as follows and their names:

```
      L1                R1
      L2                R2
      d                 triange
   a     c          square     circle
      b                  cross
     l_j_b             r_j_b
     l_j_x             r_j_x
     l_j_y             r_j_y
```

The line “PSP_ReadButtonState(SensorPort, Addr, currState)” checks to see if any of the buttons have been pressed using a Boolean state, 0 for pressed, 1 for not pressed. The joysticks return a 0 at center and has a range from -100 to 100.

Combining the above knowledge we were able to create a program to run all the above components of the arm. Motor 1 controls the shoulder, motor 2 controls the elbow, servo 1 controls the wrist up and down, servo 2 controls the wrist left and right, servo 3 open and closes the hand, and servo 4 moves the entire arm left and right. The pseudo code for the control program is as follows:

```
If triangle is pressed move shoulder up
If square pressed move shoulder down
If circle pressed move elbow up
If x pressed move elbow down
If joystick2 pushed up move wrist up
If joystick2 pushed down move wrist down
If joystick2 pushed left move wrist left
If joystick2 pushed right move wrist right
If R1 pushed close hand
If L1 pushed open hand
If R2 pushed move arm right
If L2 pushed move arm left
```

System Modeling

As explained previously, this phase of the project is limited to manually controlling each degree of freedom. The operator moves each joint to a new angle and this places the arm in a new configuration. For each configuration the hand is moved to a specific location and orientation. The equations that relate the arm's configuration to the hand's location and orientation are called the forward kinematic equations for position. What is more useful however, is the ability to determine the arm configuration that will achieve a desired hand location and orientation. In other words, the position and orientation of the hand must be defined in terms of the joint angles. This is called inverse kinematics. The forward kinematic equations for the arm are developed below followed by some possible solution techniques for the inverse kinematic problem. Developing these equations is the first step to implementing a more sophisticated method of motion control. Although this development is not an exhaustive description of the mathematics involves, it highlights the basic concepts. References are given in the appendix.

Before developing the forward kinematic equations it is necessary to describe how a frame in space can be represented by a matrix. Also, it is necessary to understand how a transformation matrix can map a frame with particular position and orientation to another. The following 4x4 matrix represents a frame in Cartesian space.

$$T = \begin{pmatrix} n_x & o_x & a_x & P_x \\ n_y & o_y & a_y & P_y \\ n_z & o_z & a_z & P_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Here, the P elements represent components of a position vector that defines the location of the frame relative to a fixed frame. The n, o, and a elements are components of unit vectors that define the x, y, and z axis of the frame respectively. These vectors determine the frame's orientation relative to the fixed frame. The bottom row is necessary to keep the matrix square.

A transformation matrix, in this context, defines the necessary translations and rotations to move from one such reference frame to another. These transformations can be combined for a series of reference frames such that the resulting relationship defines the last frame relative to the first. In the case of the robot arm, the first frame is the fixed origin and the last is the hand. This is done by simply post-multiplying each transformation matrix with the next. For example, if T_{12} represents the transformation between frames 1 and 2 and T_{23} represents the transformation between frames 2 and 3, the total transformation between 1 and 3 can be calculated as follows.

$$T_{.13} = T_{.12} \cdot T_{.23}$$

Using this methodology, a reference frame can be assigned to each joint on the robot arm. Through successive transformations between each frame, the total transformation can be

determined starting at the fixed base of the arm and ending at the hand. This will define the absolute position and orientation of the hand and be the basis for the forward kinematic equations.

The Denavit-Hartenberg representation specifies a systematic method for assigning these reference frames such that the form of the transformation matrix between successive frames is the same. The details of this method are not described here, but the assignments of each frame according to this conversion are shown in Figure 5. It is important to note that, although this robot has only revolute joints, the Denavit-Hartenberg method works for prismatic joints or a combination of the two. It will not however, model robots with motions in the Y-direction.

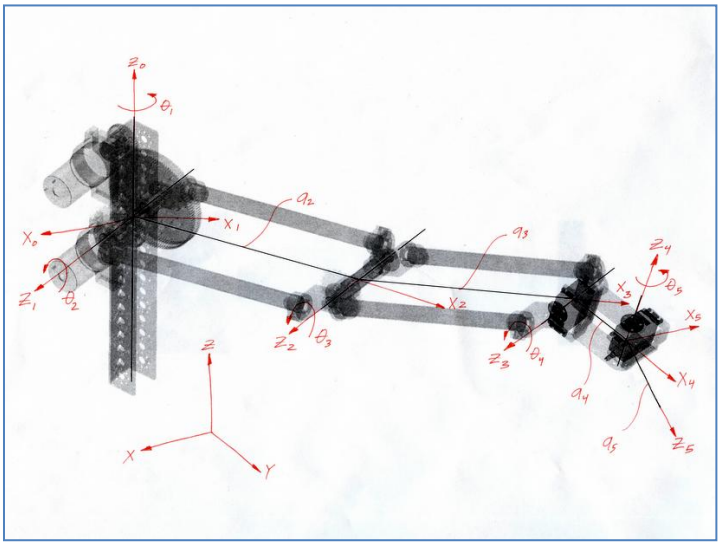


Figure 5: Reference frames bases on Denavit-Hartenberg representation.

Using the schematic above, the so called DH parameters are determined. These are shown in Table 1 below.

#	ϑ	d	a	α
1	θ1	0	0	90
2	θ2	0	a2	0
3	θ3	0	a3	0
4	θ4	0	a4	-90
5	θ5	0	0	90

Table 1: DH parameters for robot arm.

Indices for each degree of freedom are listed on the left. The values of each DOF are represented by the θ values which are unknown. The 'joint offset' is represented by d. This is zero in all cases for this robot because each joint is in the same plane. The lengths of each link, in meters, are listed in the column labeled a. The last column lists the angles between the x-axis of successive frames.

These parameters are used to define the transformation matrix between frames. This general form of this matrix is shown below.

$$A_i = \begin{pmatrix} \cos(\theta_i) & -\sin(\theta_i)\cdot\cos(\alpha_i) & \sin(\theta_i)\cdot\sin(\alpha_i) & a_i\cdot\cos(x) \\ \sin(\theta_i) & \cos(\theta_i)\cdot\cos(\alpha_i) & -\cos(\theta_i)\cdot\sin(\alpha_i) & a_1\cdot\sin(\theta_i) \\ 0 & \sin(\alpha_i) & \cos(\alpha_i) & d_i \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Using this matrix, the following relationship defines the forward kinematic equation where each A matrix is written in terms of the corresponding parameters from the table above.

$$\begin{pmatrix} n_x & o_x & a_x & P_x \\ n_y & o_y & a_y & P_y \\ n_z & o_z & a_z & P_z \\ 0 & 0 & 0 & 1 \end{pmatrix} = A_1\cdot A_2\cdot A_3\cdot A_4\cdot A_5$$

The individual equations for each element in terms of the joint angles are given in the appendix in addition to MATLAB code that can be used to compute the result for a given set of angles.

As can be seen by the resulting equations in the appendix, the inverse kinematic solution will be difficult to achieve. Each equation involves multiple coupled angles which make the problem difficult to solve analytically. A closed form solution for a simple five DOF robot such as this does exist, but in general the solution must be achieved numerically.

An attempt was made to use an artificial neural network to map the desired location and orientation to the corresponding joint angles. This was implemented using MATLAB's Neural Network Toolbox. A two layer, feed-forward network, with 20 neurons was trained using the Levenberg-Marquardt method. This was done with a built-in GUI tool. The results of this experiment were not accurate. Without understanding neural network theory better, these results can't be further interpreted. A link to the MATLAB code is listed in the appendix.

Performance

Structural

The mechanical arm is built from almost entirely pre-fabricated Tetrix aluminum components, two DC motors with gears, several small-scale servos, and is built to full-scale human arm size. Due to this, it takes a minimal amount of torque to cause vibration in or possibly warp the base components. This means that the mechanical arm cannot carry large amounts of weight. It is estimated that it can pick up slightly less than three pounds at full extension. However, the design is robust and allows large range of movement without detrimental effects on the structure, thus providing the possibility for a very human-like interaction with this arm.

Position Control

Currently there are encoders attached to the two DC motors which control the ‘shoulder’ and ‘elbow’ vertical movements however they are not used. The encoders cause difficulty with the motors because the motors resist instantaneous position correction and they lock-up. Currently all position control is manual and user-operated through a RF wireless joystick controller.

Object Grasping

The hand attached to the mechanical arm is designed to mirror a human hand. Currently it only has one DOF, its ability to open and close by moving the thumb. This however is sufficient for grasping and picking up objects. The movements are relatively slow so that they are somewhat more realistic. Additionally, if the servos speed is increased accuracy is lost.

Future Work

Sensors

Future work that can be performed on this mechanical arm includes adding sensors. These could include sonar range finder, stereo (two) camera vision, or even an experimental smell detector. This would allow automation of the entire robot

Specify End-effecter Position

Additionally, and more specifically for the mechanical arm, future work could involve solving the inverse kinematic equations for all degrees of freedom in the arm. This would allow the user or an automated intelligent program utilizing sensors to specify a position and orientation that the hand should be in. All the angles of rotation for each motor and servo would be automatically calculated and moved to that position.

Trajectory planning

With the addition of sensors, the arm and hand could potentially utilize trajectory planning. This would entail sensing an object coming toward the robot, calculating its speed and trajectory, and moving the arm and hand to a position along that trajectory to potentially deflect or catch the incoming object. The movement of the arm and hand would have to be sped up and position control accuracy would have to be increased for this to be possible.

Genetic Search Algorithm

As long as memory in the NXT brick allows, genetic algorithms could be implemented to allow for room mapping and searching. This would allow the robot, and more specifically the arm, to know the position of objects in the room and potentially interact with them.

Image processing

Image processing would be an essential upgrade with vision sensors so that the incoming data could be interpreted properly. Intelligent processing would allow more accurate readings and would provide optimized responses.

Conclusion

Overall, the goals of the first phase of the robot arm have been meet. The final assembly is made almost entirely of stock TETRIX components. The parts that were modified were done in such a way would be easy to reproduce. Although the method of motion control is limited in the current state, it serves as a strong foundation on which to test the performance and interface of the electronic components. The forward kinematic equations have been developed and the process has been well documented for future research with this robot.

Appendix

References:

Niku, A.B. (2001). *Introduction to Robotics: Analysis, Systems, Applications.*
Upper Saddle River, New Jersey: Prentice Hall.

Braunl, T. (2008). *Embedded Robotics: Mobile Robot Design and Application with Embedded Systems.*
Berlin Heidelberg: Springer-Verlag.

Hartenberg, Richard and Jacques Danavit.(1964). *Kinematic Synthesis of Linkages*
New York: McGraw-Hill

Useful Links and Downloads:

TETRIX solid model library (SolidWorks 2009):

http://web.cecs.pdx.edu/~furnissj/tetrix_models.zip

Source for TETRIX solid models:

http://www.3dcontentcentral.com/Search.aspx?arg=tetrix

Solid model of robot (SolidWorks 2009):

http://web.cecs.pdx.edu/~furnissj/final_assembly.zip

Optical encoder manufacture's link:

http://usdigital.com/products/encoders/incremental/rotary/kit/e4p/

Optical encoder data sheet:

http://usdigital.com/products/encoders/incremental/rotary/kit/e4p/

TETRIX gear motor data sheet:

http://web.cecs.pdx.edu/~furnissj/Tetrix%20DC%20drive%20motor.pdf

C code:

http://web.cecs.pdx.edu/~furnissj/armjoystick.c

Photo gallery:

http://www.flickr.com/photos/furnissj/sets/72157622850565385/

Video of final demo:

http://web.cecs.pdx.edu/~furnissj/robot_arm.wmv

MATLAB code for neural network inverse kinematic solution:

http://web.cecs.pdx.edu/~furnissj/neural_network_MATLAB.zip

RobotC code:

```
#pragma config(Hubs,  S1, HTMotor,  HTServo,  none,     none)
#pragma config(Motor,  mtr_S1_C1_1,     motorD,        tmotorNormal, openLoop)
#pragma config(Motor,  mtr_S1_C1_2,     motorE,        tmotorNormal, openLoop)
#pragma config(Servo,  srvo_S1_C1_1,    ,                    tServoNormal)
//*!!Code automatically generated by 'ROBOTC' configuration wizard               !!*//

/*-----------------------------------------------------------------------*/
/*                                                                       */
/* Program Name: PSP-Nx-motor-control.c                                  */
/* ============================                                          */
/*                                                                       */
/* Copyright (c) 2008 by mindsensors.com                                 */
/* Email: info (<at>) mindsensors (<dot>) com                            */
/*                                                                       */
/* This program is free software. You can redistribute it and/or modify */
/* it under the terms of the GNU General Public License as published by */
/* the Free Software Foundation; version 3 of the License.               */
/* Read the license at: http://www.gnu.org/licenses/gpl.txt              */
/*                                                                       */
/*-----------------------------------------------------------------------*/

const ubyte Addr = 0x02;
const tSensors SensorPort = S2;        // Connect PSPNX sensor to this port!!
#include "PSP-Nx-lib.c"

int nLeftButton = 0;
int nRightButton = 0;
int nEnterButton = 0;
int nExitButton = 0;
int tri, squ, cir, cro, a, b, c;

////////////////////////////////////////////////////////////////////////////
//
//     Gaskin     11/1/09   Modified to work wth ECE578 program
//
////////////////////////////////////////////////////////////////////////////

task
main ()
{
  int powerD = 0;
  int powerE = 0;

       psp currState;

  //
  // Note: program cannot be terminated if we hijack the 'exit' button. So there has
to be an escape sequence
  //       that will return buttons to system control! We'll use a triple click
  //
  nNxtExitClicks = 3;                // Triple clicking EXIT button will terminate
program
  //
  nI2CBytesReady[SensorPort] = 0;

  //SensorType[SensorPort] = sensorI2CCustom9V;        // or perhaps use
'Lowspeed_9V_Custom0'??
  SensorType[SensorPort] = sensorI2CMuxController;
  wait10Msec (100);


  while ( true )
    {
      wait1Msec (5);

//Move shoulder up
      if ((int)currState.triang==0)
      {
```

```
        powerD = -50;
        tri=(int)currState.triang;
      }
  //Move shoulder down
      if ((int)currState.square==0)
      {
        powerD = 10;
        squ=(int)currState.square;
      }
  //Move elbow up
      if ((int)currState.circle==0)
      {
        powerE = -50;
        cir=(int)currState.circle;
      }
  //Move elbow down
      if ((int)currState.cross==0)
      {
        powerE = 5;
        cro=(int)currState.cross;
      }
  //Turn off motors
      if ((int)currState.cross==1 && (int)currState.circle==1 &&
(int)currState.square==1 && (int)currState.triang==1)
      {
        powerD = 0;
        powerE = 0;
      }
  //Move wrist L/R
      if ((int)currState.r_j_y<-50)
      {
        servo[servo1]=ServoValue[servo1]+2;
        while(ServoValue[servo1] != servo[servo1])
        {
          a=ServoValue[servo1];

        }
      }

      if ((int)currState.r_j_y>50)
      {
        servo[servo1]=ServoValue[servo1]-2;
        while(ServoValue[servo1] != servo[servo1])
        {
          a=ServoValue[servo1];

        }
      }
  //Move wrist U/D
        if ((int)currState.r_j_x<-50)
      {
        servo[servo2]=ServoValue[servo2]-2;
        while(ServoValue[servo2] != servo[servo2])
        {
          b=ServoValue[servo2];

        }
      }

      if ((int)currState.r_j_x>50)
      {
        servo[servo2]=ServoValue[servo2]+2;
        while(ServoValue[servo2] != servo[servo2])
        {
          b=ServoValue[servo2];

        }
      }
  //Close hand
      if ((int)currState.r1==0)
      {
```

```
        servo[servo3]=ServoValue[servo3]+2;
        while(ServoValue[servo3] != servo[servo3])
        {
          c=ServoValue[servo3];

        }
      }
  //Open hand
      if ((int)currState.l1==0)
      {
        servo[servo3]=ServoValue[servo3]-2;
        while(ServoValue[servo3] != servo[servo3])
        {
          c=ServoValue[servo3];

        }
      }
  //Move arm right
      if ((int)currState.r2==0)
      {
        servo[servo4]=ServoValue[servo4]+2;
        while(ServoValue[servo4] != servo[servo4])
        {
          d=ServoValue[servo4];

        }
      }
  //Move arm left
      if ((int)currState.l2==0)
      {
        servo[servo4]=ServoValue[servo4]-2;
        while(ServoValue[servo4] != servo[servo4])
        {
          d=ServoValue[servo4];

        }
      }
    }

  wait10Msec (100);

  StopAllTasks ();
}
```

MATLAB code:

```
function position_orientation = forward_kinematics(joint_angles)

% ========================================================================
% forward_kinematics calculates the position and orientation of the end-effector
% for a 5 DOF open kinematic chain.
%
% Input:  joint_angles = [theta1 theta2 theta3 theta5 theta5]
%         1x5 vector of joint angles measured in degrees
%
% Output: position_orientation = [nx ny nz ox oy oz ax ay az Px Py Pz]
%         1x12 vector containing components of unit vectors, (a,n,o)
%         that define the x,y, and z axies of the end-effector (orientation)
%         and componented of a vector P that defines the position
% ========================================================================

joint_angles = (pi/180).*joint_angles;  % convert to radians
theta1 = joint_angles(1);
theta2 = joint_angles(2);
theta3 = joint_angles(3);
theta4 = joint_angles(4);
theta5 = joint_angles(5);

% link lengths [meters]:
a1 = 0; a2 = 0.269; a3 = 0.269; a4 = 0.063; a5 = 0;
% joint offsets [meters]:
d1 = 0; d2 = 0; d3 = 0; d4 = 0; d5 = 0;
% angles between sucessive joints [radians]:
alpha1 = pi/2; alpha2 = 0; alpha3 = 0; alpha4 = -pi/2; alpha5 = pi/2;
% transformation matricies between sucessive frames:
A1 = [cos(theta1) -sin(theta1)*cos(alpha1)  sin(theta1)*sin(alpha1) a1*cos(theta1);
      sin(theta1)  cos(theta1)*cos(alpha1) -cos(theta1)*sin(alpha1) a1*sin(theta1);
      0            sin(alpha1)              cos(alpha1)             d1            ;
      0            0                        0                       1]            ;

A2 = [cos(theta2) -sin(theta2)*cos(alpha2)  sin(theta2)*sin(alpha2) a2*cos(theta2);
      sin(theta2)  cos(theta2)*cos(alpha2) -cos(theta2)*sin(alpha2) a2*sin(theta2);
      0            sin(alpha2)              cos(alpha2)             d2            ;
      0            0                        0                       1]            ;

A3 = [cos(theta3) -sin(theta3)*cos(alpha3)  sin(theta3)*sin(alpha3) a3*cos(theta3);
      sin(theta3)  cos(theta3)*cos(alpha3) -cos(theta3)*sin(alpha3) a3*sin(theta3);
      0            sin(alpha3)              cos(alpha3)             d3            ;
      0            0                        0                       1]            ;

A4 = [cos(theta4) -sin(theta4)*cos(alpha4)  sin(theta4)*sin(alpha4) a4*cos(theta4);
      sin(theta4)  cos(theta4)*cos(alpha4) -cos(theta4)*sin(alpha4) a4*sin(theta4);
      0            sin(alpha4)              cos(alpha4)             d4            ;
      0            0                        0                       1]            ;

A5 = [cos(theta5) -sin(theta5)*cos(alpha5)  sin(theta5)*sin(alpha5) a5*cos(theta5);
      sin(theta5)  cos(theta5)*cos(alpha5) -cos(theta5)*sin(alpha5) a5*sin(theta5);
      0            sin(alpha5)              cos(alpha5)             d5            ;
      0            0                        0                       1]            ;
% total transformation matrix:
A = A1*A2*A3*A4*A5;
A = A(1:3,:); %eliminate bottom row

position_orientation = reshape(A,1,[]); % [nx ny nz ox oy oz ax ay az Px Py Pz]]

% round down small numbers
tolerance = 0.0001;
for i = 1:length(position_orientation)
    if abs(position_orientation(i)) < tolerance
        position_orientation(i) = 0;
    end
end
```

Forward Kinematic Equations:

$$> \quad T(1,1) = A(1,1);$$

$$n_x = \left(\left(\cos(\theta_1)\cos(\theta_2)\cos(\theta_3) - \cos(\theta_1)\sin(\theta_2)\sin(\theta_3)\right)\cos(\theta_4) + \left(-\cos(\theta_1)\cos(\theta_2)\sin(\theta_3) - \cos(\theta_1)\sin(\theta_2)\cos(\theta_3)\right)\sin(\theta_4)\right)\cos(\theta_5) - \sin(\theta_1)\sin(\theta_5) \tag{1}$$

$$> \quad T(2,1) = A(2,1);$$

$$n_y = \left(\left(\sin(\theta_1)\cos(\theta_2)\cos(\theta_3) - \sin(\theta_1)\sin(\theta_2)\sin(\theta_3)\right)\cos(\theta_4) + \left(-\sin(\theta_1)\cos(\theta_2)\sin(\theta_3) - \sin(\theta_1)\sin(\theta_2)\cos(\theta_3)\right)\sin(\theta_4)\right)\cos(\theta_5) + \cos(\theta_1)\sin(\theta_5) \tag{2}$$

$$> \quad T(3,1) = A(3,1);$$

$$n_z = \left(\left(\sin(\theta_2)\cos(\theta_3) + \cos(\theta_2)\sin(\theta_3)\right)\cos(\theta_4) + \left(-\sin(\theta_2)\sin(\theta_3) + \cos(\theta_2)\cos(\theta_3)\right)\sin(\theta_4)\right)\cos(\theta_5) \tag{3}$$

$$> \quad T(1,2) = A(1,2);$$

$$o_x = -\left(\cos(\theta_1)\cos(\theta_2)\cos(\theta_3) - \cos(\theta_1)\sin(\theta_2)\sin(\theta_3)\right)\sin(\theta_4) + \left(-\cos(\theta_1)\cos(\theta_2)\sin(\theta_3) - \cos(\theta_1)\sin(\theta_2)\cos(\theta_3)\right)\cos(\theta_4) \tag{4}$$

$$> \quad T(2,2) = A(2,2);$$

$$o_y = -\left(\sin(\theta_1)\cos(\theta_2)\cos(\theta_3) - \sin(\theta_1)\sin(\theta_2)\sin(\theta_3)\right)\sin(\theta_4) + \left(-\sin(\theta_1)\cos(\theta_2)\sin(\theta_3) - \sin(\theta_1)\sin(\theta_2)\cos(\theta_3)\right)\cos(\theta_4) \tag{5}$$

$$> \quad T(3,2) = A(3,2);$$

$$o_z = -\left(\sin(\theta_2)\cos(\theta_3) + \cos(\theta_2)\sin(\theta_3)\right)\sin(\theta_4) + \left(-\sin(\theta_2)\sin(\theta_3) + \cos(\theta_2)\cos(\theta_3)\right)\cos(\theta_4) \tag{6}$$

$$> \quad T(1,3) = A(1,3);$$

$$a_x = \left(\left(\cos(\theta_1)\cos(\theta_2)\cos(\theta_3) - \cos(\theta_1)\sin(\theta_2)\sin(\theta_3)\right)\cos(\theta_4) + \left(-\cos(\theta_1)\cos(\theta_2)\sin(\theta_3) - \cos(\theta_1)\sin(\theta_2)\cos(\theta_3)\right)\sin(\theta_4)\right)\sin(\theta_5) + \sin(\theta_1)\cos(\theta_5) \tag{7}$$

$$> \quad T(2,3) = A(2,3);$$

$$a_y = \left(\left(\sin(\theta_1)\cos(\theta_2)\cos(\theta_3) - \sin(\theta_1)\sin(\theta_2)\sin(\theta_3)\right)\cos(\theta_4) + \left(-\sin(\theta_1)\cos(\theta_2)\sin(\theta_3) - \sin(\theta_1)\sin(\theta_2)\cos(\theta_3)\right)\sin(\theta_4)\right)\sin(\theta_5) - \cos(\theta_1)\cos(\theta_5) \tag{8}$$

$$> \quad T(3,3) = A(3,3);$$

$$a_z = \left(\left(\sin(\theta_2)\cos(\theta_3) + \cos(\theta_2)\sin(\theta_3)\right)\cos(\theta_4) + \left(-\sin(\theta_2)\sin(\theta_3) + \cos(\theta_2)\cos(\theta_3)\right)\sin(\theta_4)\right)\sin(\theta_5) \tag{9}$$

$$> \quad T(1,4) = A(1,4);$$

$$P_x = 0.063\left(\cos(\theta_1)\cos(\theta_2)\cos(\theta_3) - \cos(\theta_1)\sin(\theta_2)\sin(\theta_3)\right)\cos(\theta_4) + 0.063\left(-\cos(\theta_1)\cos(\theta_2)\sin(\theta_3) - \cos(\theta_1)\sin(\theta_2)\cos(\theta_3)\right)\sin(\theta_4) + 0.269\cos(\theta_1)\cos(\theta_2)\cos(\theta_3) - 0.269\cos(\theta_1)\sin(\theta_2)\sin(\theta_3) + 0.269\cos(\theta_1)\cos(\theta_2) \tag{10}$$

$$> \quad T(2,4) = A(2,4);$$

$$P_y = 0.063\left(\sin(\theta_1)\cos(\theta_2)\cos(\theta_3) - \sin(\theta_1)\sin(\theta_2)\sin(\theta_3)\right)\cos(\theta_4) + 0.063\left(-\sin(\theta_1)\cos(\theta_2)\sin(\theta_3) - \sin(\theta_1)\sin(\theta_2)\cos(\theta_3)\right)\sin(\theta_4) + 0.269\sin(\theta_1)\cos(\theta_2)\cos(\theta_3) - 0.269\sin(\theta_1)\sin(\theta_2)\sin(\theta_3) + 0.269\sin(\theta_1)\cos(\theta_2) \tag{11}$$

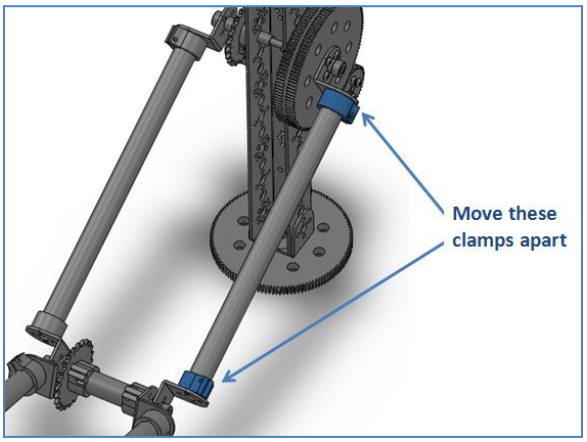
$$> \quad T(3,4) = A(3,4);$$

$$P_z = 0.063\left(\sin(\theta_2)\cos(\theta_3) + \cos(\theta_2)\sin(\theta_3)\right)\cos(\theta_4) + 0.063\left(-\sin(\theta_2)\sin(\theta_3) + \cos(\theta_2)\cos(\theta_3)\right)\sin(\theta_4) + 0.269\sin(\theta_2)\cos(\theta_3) + 0.269\cos(\theta_2)\sin(\theta_3) + 0.269\sin(\theta_2) \tag{12}$$

Trouble Shooting Guide

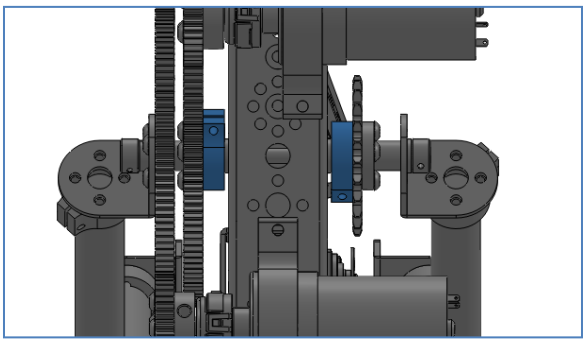
Although we didn't experience any major technical difficulties after the arm was built and tested, there are a few aspects of the assembly that may need attention after more use. The overall structure of the arm is pushing the bounds of the TETRIX components. To keep the arm in good working condition the following should be considered.

1) At full extension, the motors have to resist a torque that is very near their limit. For this reason, motions that are programmed into the arm must be very slow. Quick movements (accelerations) increase the risk of stripping, bending, or loosening components. If you want the arm to move faster or carry a greater load, a first step would be to make it smaller by shortening the tubes.
2) The chain is tensioned by positioning the tube clamps at the ends of the tubes on the upper arm. When there is load on the arm, the chain will become loose due to flex in the system. This can't be avoided. If the chain becomes excessively loose, follow these steps to adjust it.
   - Loosen the set screw that binds the 40T gear to the DC motor shaft that controls the elbow joint. This will make the elbow joint move freely.
   - Loosen the 4 tubing clamps on the upper arm. Make sure it doesn't fall apart.
   - On the non-chain side, slide the clamps away from the center of the tube a couple of millimeters and re-tighten the clamps. See below. The tubing plugs may slide away from the ends of the tube when you loosen the clamps. Make sure you slide them back before you retighten the clamps.

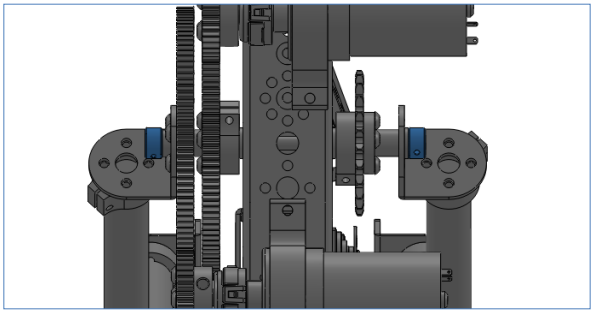


   - Do the same on the other side. As you move the clamps away, you will feel the chain getting tight. Make the chain as tight as you can without sliding the clamp off the end of the tube. This process it a bit awkward and may take several attempts.
   - Make sure you don't rotate the clamps relative to one another. This will put a twist in the arm.
   - Re-tighten the set screw on the 40T gear. Make sure the screw is against the flat on the motor shaft.

3) The main shoulder pivot may become loose. Follow these steps to correct the problem.
   - Loosen the set screws that bind the inside split clamps to the main axle. These are highlighted in the figure below. On the real robot, these pieces have been modifies and don't look exactly the same as in the figure.



   - Sandwich these components together and simultaneously tighten the set screws. This will take two people. Try to get at least one setscrew on each clamp to tighten against the flat on the axle.
   - Now, do the same thing with the axle set collars. See figure below.



My final word of advice is in regard to the supposed position control of the DC motors with RobotC. There is no PID algorithm for position control, only for speed. After reading everything I could find about this on RobotC forums and in the documentation, there does not seem to be a reliable way to command the motors to a position. The main approach that I found was to drive the motor at a constant speed (with the PID speed control) in a while loop until some encoder value was reached. If you want to implement some sophisticated method of motion control with the robot, I don't think this will cut it. If you read about other's frustrations with this on the forums, you will see that it hasn't cut it for anyone else ether. My advice would be to design a controller on your own unless you find successful solution on the internet.

Bill of Materials

Part Name	*Part Number*	*Vendor*	*Quantity*	*Cost*	*Notes*
TETRIX Single-Servo Motro Bracket	W739060	Lego Education	3	10.95/2	
TETRIX Single-Servo Motro Bracket*	W739060	Lego Education	1	10.95/2	*modified for thumb
HiTech HS-475 HB Servo	W739080	Lego Education	3	24.95/1	
Large Servo*	NA	Robotics Lab	1	NA	*mountin tabs cut off
TETRIX Tube (220mm)	W739076	Lego Education	2	9.95/2	
TETRIX Tube (220mm)*	W739076	Lego Education	2	9.95/2	*cut to make chain tight
TETRIX Tube (220mm)*	W739076	Lego Education	1	9.95/2	*cut to for elbow joint
TETRIX Split Clamp	W739078	Lego Education	1	7.95/2	
TETRIX Split Clamp*	W739078	Lego Education	3	7.95/2	*clamp cut for clearance, set screws added
TETRIX Split Clamp*	W739078	Lego Education	1	7.95/2	*set screw added
TETRIX Tubing Plug Pack*	W739193	Lego Education	8	2.95/2	*8mm holes drilled for servo wire routing
TETRIX L Bracket	W739062	Lego Education	8	5.95/2	
TETRIX L Bracket*	W739062	Lego Education	2	5.95/2	*cut to mount base to 120T gear
TETRIX Axle Set Collar	W739092	Lego Education	3	3.95/6	
TETRIX Bronze Bushing	W739091	Lego Education	5	15.95/12	
TETRIX Bronze Bushing*	W739091	Lego Education	2	15.95/12	*cut to length for shoulder joint
TETRIX Axle Spacer (1/8")	W739100	Lego Education	8	1.95/12	
TETRIX Gear (40T)	W739028	Lego Education	3	24.95/2	
TETRIX Gear (120T)	W739085	Lego Education	2	29.95/1	
TETRIX Gear (120T)*	W739085	Lego Education	1	29.95/1	*tapped to mount L bracket at shoulder
TETRIX DC Drive Motor	W739083	Lego Education	2	29.95/1	
TETRIX Motor Encoder Pack	W739140	Lego Education	2	79.95/1	same as US Digital E4P-360-236-D-H-T-3
TETRIX 16T Sprocket	W739165	Lego Education	2	18.95/2	
TETRIX Chain	W739173	Lego Education	1	14.95/1	
TETRIX Motor Shaft Hubs	W739079	Lego Education	2	7.95/2	
TETRIX 100mm Axle	W739088	Lego Education	3	17.95/6	
TETRIX Motor Mount	W739089	Lego Education	2	19.95/1	

TETRIX Channel (416mm)	W739069	Lego Education	1	19.95/1	
HiTechnic DC Motor Controller	W991444	Lego Education	1	79.95/1	
HiTechnic Servo Controller	W991445	Lego Education	1	79.95/1	
HiTech Servo Horn	W739020	Lego Education	3	5.95/1	
TETRIX Tube Clamp	W739077	Lego Education	10	7.95/2	
Servo Extension	W739081	Lego Education	3	1.95/1	
TETRIX Servo Joint Pivot Bracket	W739063	Lego Education	1	11.95/1	
TETRIX Servo Joint Pivot Bracket*	W739063	Lego Education	1	11.95/1	*cut for nut clearance on hand
Coat hanger*	NA	NA	5	NA	*cut and bent for fingers
Custom sheet metal servo bracket	NA	machine shop	1	NA	mounts large servo to base