Hands-on Lab

XL-320 NXC Programming – Trajectory Planning

The inverse kinematics (IK) of a 2-link planar manipulator was derived previously. In that lab the user defined a desired task space location (x, y). IK would compute the necessary joint angles (θ_1, θ_2) to move the end-effector to that location. To move the end-effector along a line, one approach is to define a set of points for that line and use IK to compute the joint angles. One will observe that there are some pros and cons to this approach.

Preliminary: Parametric Equation of Line and Waypoints

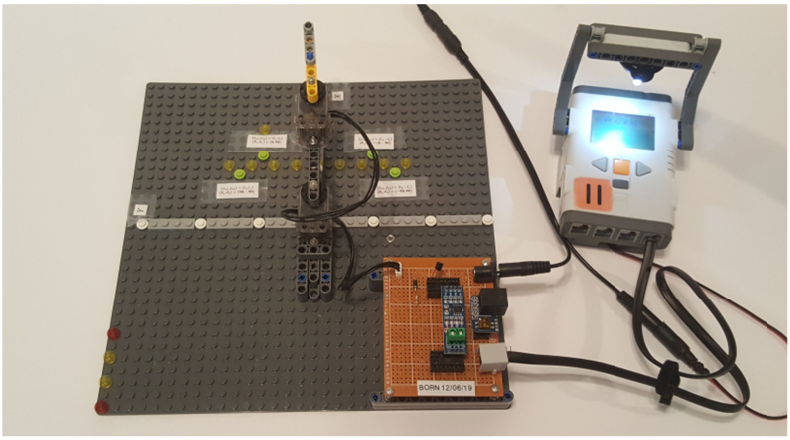


Figure A: There are 9 yellow colored 1-stud bricks on the LEGO base plate. These bricks define set of points (called waypoints) that characterize a straight line parallel to the Y-axis.

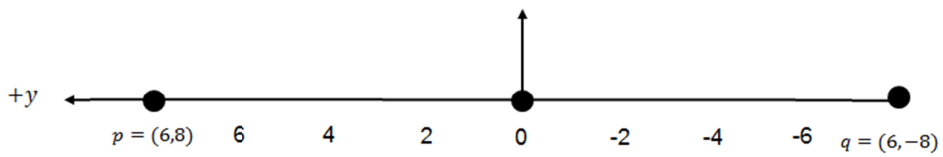
Suppose $\boldsymbol{p}$ and $\boldsymbol{q}$ are the start and end points of line. Then, the parametric equation of line $\boldsymbol{l}(t)$ has form $\boldsymbol{l}(t) = \boldsymbol{p} + t\boldsymbol{v}$. If say that $\boldsymbol{v} = \boldsymbol{q} - \boldsymbol{p}$ then one has

$$\boldsymbol{l}(t) = \boldsymbol{p} + t\boldsymbol{v} = \boldsymbol{p} + t(\boldsymbol{q} - \boldsymbol{p}) = \boldsymbol{p}(1 - t) + t\boldsymbol{q} \tag{1}$$

If one wants n equally spaced segments between $\boldsymbol{p}$ and $\boldsymbol{q}$ then define n values of $t \in [0,1]$ as

$$t = \frac{i}{n} \quad \text{where i} = 0,\ 1,\ \ldots n - 1 \tag{2}$$

Example: Suppose just want 2 equally spaced segments between $p = (6,8)$ and $q = (6,-8)$. One visually one sees that this is 1 point should be $(0,0)$:



Here, $n = 2$ equally spaced segments so $t = \frac{1}{2} = 0.5$ and $l\left(\frac{1}{2}\right) = 8 \times \left(1 - \frac{1}{2}\right) + \frac{1}{2}(-8) = 0$ for the point on the y-axis. The point on the x-axis will still be 6.

Concept 1: Implement Parametric Equation of a Line

```
// FILE: xl320-twoLinkFunctions1_0a.h
// DATE: 01/21/20 12:09
// AUTH: P.Oh
// DESC: XL-320 based 2-DOF planar manipulator related functions
// VERS: 1.0a: rotateMotorAbsolutely(); goHome(); twoLinkInverseKinematics()
// REFS: H-files xl320-defines1_0a.h and xl320-functions1_0d.h
// NOTE: This example uses an XL-320 configured with ID# 3 and ID# 7

#define ID_ALL_MOTORS 0XFE // 0XFE commands all XL-320 motors
#define ID_MOTOR01    0X03 // Assumes Motor 1 configured with ID = 3
#define ID_MOTOR02    0X07 // Assumes Motor 2 configured with ID = 7
#define mmPerStud     8    // 8 millimeters per LEGO stud

struct theta {                      // angles in [rad]
float theta1, theta2;
};
struct thetaInDegrees {             // angles in [deg]
  float theta1InDegrees, theta2InDegrees;
};

typedef theta THETA;
typedef thetaInDegrees THETA_IN_DEGREES;

void rotateMotorAbsolutely(float angle01, float angle02) { //------------------
 // Rotates the two Dynamixel XL-320 motors to their desired angles
 // Assumes motor count of 512 denotes 0 degrees and right-hand rule for
 // rotational direction

  float desiredAngle01InDegrees;       // Angle Motor 1 to move to [deg]
  float desiredAngle02InDegrees;       // Angle Motor 2 to move to [deg]
  float degreesPerCount;               // Conversion 0.29 [degrees/count]
  float calculatedCount;               // Count equivalent of desired angle [count]
  int   motor01Offset;                 // Motor 1's offset [count]
  float theta01InDegrees;              // Motor 1 angle [counts]
  int   theta01InCounts;               // Motor 1 angle [deg]
  int   motor02Offset;                 // Motor 2's offset [count]
  float theta02InDegrees;              // Motor 2 angle [counts]
  int   theta02InCounts;               // Motor 2 angle [deg]
  string msg01, msg02;                 // dummy strings to print values to screen

  motor01Offset = 512; // Dynamixel Wizard fixes Link 1 at zero deg (i.e. 512 counts)
  motor02Offset = 512; // Dynamixel Wizard fixes Link 2 at zero deg (i.e. 512 counts)

  // Note 1: Looking into horn from Top, count > 512 is CCW (i.e. +Z axis)
  // and count < 512 is CW (i.e. -Z axis)
  degreesPerCount = 0.29; // [deg/count] found from XL-320 data sheet

  // ClearScreen();
  desiredAngle01InDegrees = angle01;
  theta01InCounts = motor01Offset + desiredAngle01InDegrees/degreesPerCount;
  desiredAngle02InDegrees = angle02;
  theta02InCounts = motor02Offset + desiredAngle02InDegrees/degreesPerCount;

  TextOut(0, LCD_LINE4, "Going to" );
  sprintf(msg01, "(%2.1f," ,desiredAngle01InDegrees);
  sprintf(msg02, "%2.1f) deg" , desiredAngle02InDegrees);
  TextOut(0, LCD_LINE5, strcat(msg01, msg02));

  XL320_servo(ID_MOTOR01, theta01InCounts, 200); // motor position at speed 200
  // Wait(1500); // Uncomment and change value e.g. 1500 ms if troubleshooting
  XL320_servo(ID_MOTOR02, theta02InCounts, 200); // motor position at speed 200
  Wait(1000); // Uncomment to see impact on communications
  PlayTone(TONE_B3,50);

}; // end rotateMotorAbsolutely function ---------------------------------
```

Figure 1A: Listing of **xl320-twoLinkFunctions1_0a.h**

```
void goHome() { //---------------------------------------------------------
 // Assumes motor count of 512 denotes 0 degrees.
 // Assumes Home position has Joints 1 and 2 at 0 degrees

 rotateMotorAbsolutely(0.0, 0.0);

}; // end goHome function ------------------------------------------------

THETA_IN_DEGREES twoLinkInverseKinematics(float s1, float s2, float u, float v) { // ---

  // Returns struct that contains the joint angles in [deg]
  // based on inputs: link lengths (s1, s2) and task space position (u, v)
  //

  float C, k1, k2, num, den;       // IK related variables for calculations
  THETA_IN_DEGREES phiInDegrees;   // structure defined at start of this program
  THETA phi;                       // structure defined at start of this program

  C = ( pow(u,2)+pow(v,2) - pow(s1,2)-pow(s2,2) ) / (2*s1*s2);
  // Choose +'ve root or comment out
  num = sqrt(1-pow(C,2));
  // num = -sqrt(1-pow(C,2));
  phi.theta2 = atan2(num, C); // [rad]
  phiInDegrees.theta2InDegrees = phi.theta2 * 180/PI; // [deg]
  k1 = s1 + s2*cos(phi.theta2);
  k2 = s2*sin(phi.theta2);
  phi.theta1 = atan2(v, u) - atan2(k2, k1); // [rad]
  phiInDegrees.theta1InDegrees = (phi.theta1 * 180/PI); // [deg]

  return phiInDegrees;

}; // end twoLinkInverseKinematics function
```

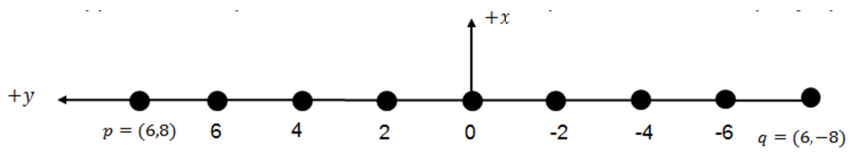
Figure 1A continued: Listing for **xl320-twoLinkFunctions1_0a.h**

To make code more readable, `xl320-ik-1_0.nxc` can be stripped out to create functions. Cut and-paste code to form the file `xl320-twoLinkFunctions1_0a.h` as shown in **Figure 1A**.

The H-file begins defining structures (yellow-highlight) `theta` and `thetaInDegrees` to contain the joint angles. Commanding the XL-320 to absolute angles (based on the 512 offset value) is often needed. Hence the function `rotateMotorAbsolutely` was moved to this H-file. Additionally, moving the home position (where joints angles are both at zero degrees) the function `goHome` was added.

Lastly, the inverse kinematics equations were cut-and-pasted into this H-file. These equations calculate the joint angles. In the C programming language, functions can only return one variable. One approach is to use a structure variable. Thus the function `twoLinkInverseKinematics` was created.

The program `xl320-line1_0.nxc` shown in **Figure 1B** contains `main` and calls these functions. To demonstrate this program, 7 way points will be used as shown in **Figure 1C** (left). (1) and (2) should calculate the locations of these way points as given in **Figure 1C** (right)



i	t	l
0	0	8
1	0.125	6
2	0.25	4
3	0.375	2
4	0.5	0
5	0.625	-2
6	0.75	-4
7	0.875	-6
8	1	-8

Figure 1C: 7 way points to characterize the line between start (6,8) studs and (6,-8) studs

```
// FILE: xl320-line1_0.nxc
// DATE: 01/22/20 17:12
// AUTH: P.Oh
// DESC: XL-320 based 2-DOF planar manipulator.  Trajectory planned based on
//       a parametric line.  User prescribes the desired number of equally
//       spaced points given start and end points of that line
// VERS: 1.0a: xl320-twoLinkFunctions1_0a.h contains functions
//       rotateMotorAbsolutely(); goHome(); twoLinkInverseKinematics()
// REFS: H-files xl320-defines1_0a.h and xl320-functions1_0d.h
// NOTE: If factory default XL-320 used, then ID is 0x01
//       ID of 0xFE commands any and all XL-320 motors
//       This example uses an XL-320 configured with ID# 3

#include "xl320-defines1_0a.h"
#include "xl320-functions1_0d.h"
#include "xl320-twoLinkFunctions1_0a.h"

// Global variables
bool  orangeButtonPushed;       // Detect Brick Center button state
bool  rightArrowButtonPushed;   // Detect Brick right arrow button state

task main() {

  // planar manipulator variables
  float l1, l2; // length of link 1 and link 2 [mm]

  float xDesired[], yDesired[]; // desired line's (x,y) way points
  int numberOfWayPoints;        // # of points between start and end points
  int numberOfSpaces;           // # of equally spaced segments
  int maxVectorSize;            // # array elements = numberOfWayPoints + 2
  float xP, yP;                 // EE absolute position wrt x0y0 frame [mm]
  int i, j;                     // dummy index variables
  string str01, str02;          // dummy string variables to display text
  float t;                      // variable for parametric equation of line

  THETA_IN_DEGREES anglesInDegrees;  // struct defined in
  THETA angles;                      // xl320-twoLinkFunctions1_0a.h

  // Initializations
  l1 = 7 * mmPerStud;            // [mm] link 1 is 7 studs long
  l2 = 5 * mmPerStud;            // [mm] link 2 is 5 studs long

  // Define and initialize arrays that will hold waypoints
  numberOfWayPoints = 7;                  // start stud + (7 studs) + end stud
  numberOfSpaces = numberOfWayPoints + 1; // i.e. hence 8 equally spaced segments
  maxVectorSize = numberOfWayPoints + 2;  // include both start and end points
  ArrayInit(xDesired, 0, maxVectorSize);  // initialize waypoint x vector to 0
  ArrayInit(yDesired, 0, maxVectorSize);  // initialize waypoint x vector to 0
  // Initialize start and end points of line
  xDesired[0] = 6*mmPerStud;                    // [mm] starting point
  yDesired[0] = 8*mmPerStud;                    // [mm] starting point
  xDesired[numberOfWayPoints+1] = 6*mmPerStud;  // [mm] ending point
  yDesired[numberOfWayPoints+1] = -8*mmPerStud; // [mm] ending point

  // Parametric equation of line to calculate equally spaced points
  i = 1;
  while (i <= (numberOfWayPoints)) {
    t = i/(numberOfWayPoints+1);
    xDesired[i] = 6*mmPerStud; // [mm]
    yDesired[i] = (yDesired[0]*(1.0-t)) + (t*yDesired[numberOfWayPoints+1]);
    i++;
  } // end of while
  sprintf(str01, "t=%3.3f " , 1.0/(numberOfWayPoints+1) );
  sprintf(str02, " N=%d" , numberOfWayPoints);
  TextOut(0, LCD_LINE1, strcat(str01, str02));
  TextOut(0, LCD_LINE2, FormatNum("yD[0] = %3.2f mm" , yDesired[0]));
  sprintf(str01, "yD[%d]" , numberOfWayPoints + 1);
  sprintf(str02, "%3.2f mm" , yDesired[numberOfWayPoints + 1]);
  TextOut(0, LCD_LINE3, strcat(str01, str02) );
  TextOut(0, LCD_LINE5, "Cont'd ORG" );
```

Figure 1B: Listing for **xl320-line1_0.nxc**

```
  do {
    orangeButtonPushed = ButtonPressed(BTNCENTER, FALSE);
  } while(!orangeButtonPushed);

  UseRS485();
  RS485Enable();
  RS485Uart(HS_BAUD_57600, HS_MODE_8N1); //57600 baud, 8bit, 1stop, no parity

  // Prompt user to begin
  ClearScreen();
  TextOut(0, LCD_LINE1, "Start: hit ->");
  do {
     rightArrowButtonPushed = ButtonPressed(BTNRIGHT, FALSE);
  } while(!rightArrowButtonPushed);
  ClearScreen();

  // (1) go to home position
  TextOut(0, LCD_LINE1, "Homing..." );
  Wait(2000);
  goHome();
  Wait(2000);
  TextOut(0, LCD_LINE7, "Homed" );
  PlayTone(TONE_E4, 500);
  TextOut(0, LCD_LINE8, "Cont'd ORG" );
  do {
    orangeButtonPushed = ButtonPressed(BTNCENTER, FALSE);
  } while(!orangeButtonPushed);

  // (2) move to start of line
  ClearScreen();
  TextOut(0, LCD_LINE1, "Going (xP0,yP0):" );
  Wait(2000);
  xP = xDesired[0];
  yP = yDesired[0];
  sprintf(str01, "(%3.2f," , xP/mmPerStud);
  sprintf(str02, "%3.2f) stud" , yP/mmPerStud);
  TextOut(0, LCD_LINE2, strcat(str01, str02));
  anglesInDegrees = twoLinkInverseKinematics(l1, l2, xP, yP);
  rotateMotorAbsolutely(anglesInDegrees.theta1InDegrees, anglesInDegrees.theta2InDegrees);
  TextOut(0, LCD_LINE7, "Now @ line start" );
  TextOut(0, LCD_LINE8, "Start: hit ->");
  do {
     rightArrowButtonPushed = ButtonPressed(BTNRIGHT, FALSE);
  } while(!rightArrowButtonPushed);

  // (3) Iterate thru waypoints, calculate IK, and command motor angles
  for(i=1; i <= numberOfWayPoints+1; i++) {
      xP = xDesired[i];
      yP = yDesired[i];
      anglesInDegrees = twoLinkInverseKinematics(l1, l2, xP, yP);
      sprintf(str01, "(%3.2f," , xP/mmPerStud);
      sprintf(str02, "%3.2f) stud" , yP/mmPerStud);
      TextOut(0, LCD_LINE2, strcat(str01, str02));
      // Actuate the XL-320 motors
      rotateMotorAbsolutely(anglesInDegrees.theta1InDegrees, anglesInDegrees.theta2InDegrees);
  }; // end for-loop

  // (4) Lastly, since trajectory done, go home
  Wait(3000);
  TextOut(0, LCD_LINE7, "Line done" );
  TextOut(0, LCD_LINE8, "Go Home: Hit ORG" );
  do {
    orangeButtonPushed = ButtonPressed(BTNCENTER, FALSE);
  } while(!orangeButtonPushed);
  ClearScreen();
  goHome();
  TextOut(0, LCD_LINE4, "Homed. Exiting" );
  Wait(3000);
  PlaySound(SOUND_DOUBLE_BEEP);
} // end main
```

Figure 1B continued: Listing for **xl320-line1_0.nxc**

Recall that this LEGO-based 2-link planar manipulator employs Beam 9 and Beam 7 parts for links 1 and 2 respectively. However, their mounting points on the XL-320 servos define the lengths of links 1 and 2 to be 7 and 5 studs long (see **Figure 1D**). This was defined as:

```
l1 = 7 * mmPerStud;             // [mm] link 1 is 7 studs long
l2 = 5 * mmPerStud;             // [mm] link 2 is 5 studs long
```

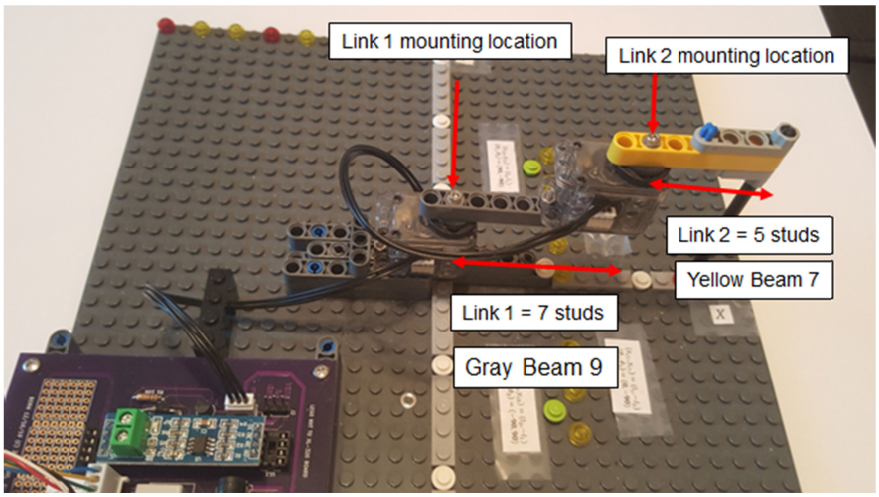


Figure 1D: One can visually count that the lengths of links 1 and 2 are 7 and 5 studs respectively. In LEGO, studs are 8 millimeters apart.

Also recall (see **Figure 1E**), 9 yellow 1-stud bricks were placed on the 32 x 32 LEGO baseplate. The first one (start point) was mounted on baseplate stud location (6, 8). The ninth one (end point) was affixed at stud location (6, -8). One observes that to be equally spaced apart, the remaining 7 bricks should be mounted 2 studs apart.

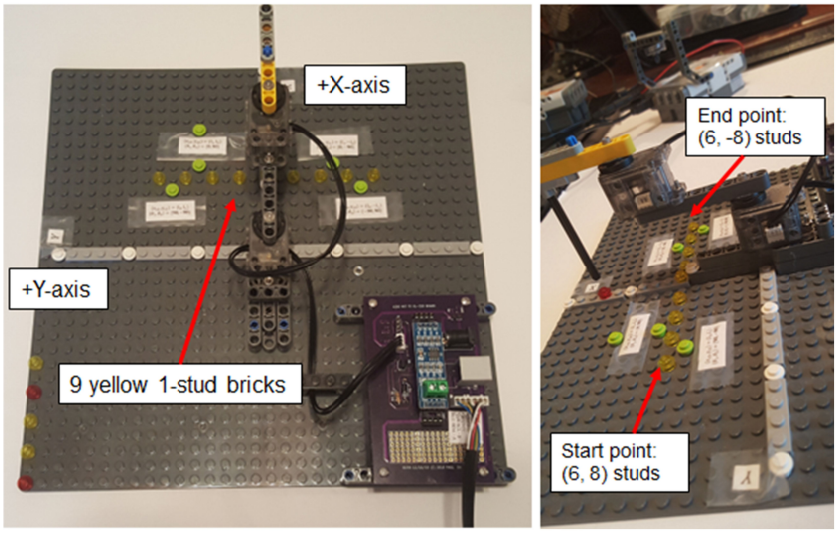


Figure 1E: There are 9 yellow 1-studs: 1 start point + 1 end point + 7 way points

Referencing the yellow-highlighted lines, this program begins with setting the desired the number of waypoints (e.g. `numberOfWayPoints = 7`) and then using `ArrayInit` to properly size the vectors that will hold these way point values and initialize them to zero.

The desired line trajectory will be parallel to the Y-axis. Thus the way point vector for x values will remain constant. Equations (1) and (2) is thus implemented on the y values and stored in the vector `yDesired[i]`.

After the way point vectors have been calculated, The Brick's Port 4 is set for 57,600 baud 8N1. The first step is to home the manipulator by calling the `goHome()` function. The second step is to move the end-effector to the line's starting position. The values of this position are stored in `xDesired[0]` and `yDesired[0]`. These values are passed to the function `twoLinkInverseKinematics` which returns the corresponding joint angles in the structure `anglesInDegrees`. These angles are then fed into `rotateMotorAbsolutely` to command the two XL-320 to those angles. This process is of calling `twoLinkInverseKinematics` and `rotateMotorAbsolutely` is used for the third step – which iterates thru the remaining way points, including the end of the line. The fourth and final step is to home the manipulator once the line trajectory is completed.

Compiling requires that the H-files are in the same directory as **`xl320-line1_0.nxc`**. These H-Files are: **`xl320-twoLinkFunctions1_0a.h`**, `xl320-defines1_0a.h` and `xl320-functions1_0d.h`. Execute the file and observe how the end-effector moves along the line.

Congratulations! You parameterized a line with equally spaced way points to move the manipulator's end-effector in a line.

Exercises

1.1 In **`xl320-line1_0.nxc`** change the value of `numberOfWayPoints` using the table below. Compile, execute and fill your observations

`numberOfWayPoints`	t when $i = 1$	$l(t\|_{i=1})$ [stud] value	What path does the end-effector make? How well does it "stay" on the desired line?
1	0.5	0	
2			
3			
5			
15			
31			

Concept 2: Velocities versus Positions – Reducing Delays

One key observation from Exercise 1.1 is that the end-effector's motion is not smooth. Rather it's a go-stop-go motion. The core reason stems from position, rather than velocity, commands. The program **xl320-line1_0.nxc** forces the end-effector to visit each way point. This leads to the discrete, instead of a continuous, motion profile. Close examination of the `rotateMotorAbsolutely` function in **xl320-twoLinkFunctions1_0a.h** shows:

```
XL320_servo(ID_MOTOR01, theta01InCounts, 200); // motor position at speed 200
// Wait(1500); // Uncomment and change value e.g. 1500 ms if troubleshooting
XL320_servo(ID_MOTOR02, theta02InCounts, 200); // motor position at speed 200
Wait(1000); // Uncomment to see impact on communications
PlayTone(TONE_B3,50);
```

The `Wait(1000)` statement creates a 1 second delay. At 57,600 baud, bits are transmitted to through to a XL-320 servo at 17.36 microseconds (or 0.139 milliseconds per byte). There function doesn't perform error checking or confirms successful transmission of the byte packet. Thus the `Wait` statement gives some time margin for the XL-320 servos to receive, process, and execute the commanded motion. This time margin can be changed to reduce the go-stop-go phenomena.

Exercises

2.1 Set `numberOfWayPoints` in **xl320-line1_0.nxc** and the `Wait` statement in **xl320-twoLinkFunctions1_0a.h** using the table below.. Compile, execute, and fill your observations

`numberOfWayPoints`	`Wait`	What path does the end-effector make? How well does it "stay" on the desired line?
15	50	
7	50	
5	50	
1	50	
15	200	
7	200	
5	200	
1	200	

Concept 3: Velocities versus Positions – On-the-Fly Velocity Commands

Many smart servos have the ability to change velocities "on-the-fly". This means that the servo's velocity changes, regardless if it reached the commanded position or not, as soon as it receives a command. One may have observed that in Exercise 2.1. For example, if the Wait time is short (e.g. 50 milliseconds) and the number of way points was small (e.g. 5), the end-effector did not visit all the way points. Rather, the 2-link manipulator traced a curvilinear path. This is because the servos processed the motion command quickly and before it could reach that way point, received the command to the next way point.

Again, close examination of the `rotateMotorAbsolutely` function in **`xl320-twoLinkFunctions1_0a.h`** shows:

```
XL320_servo(ID_MOTOR01, theta01InCounts, 200); // motor position at speed 200
// Wait(1500); // Uncomment and change value e.g. 1500 ms if troubleshooting
XL320_servo(ID_MOTOR02, theta02InCounts, 200); // motor position at speed 200
Wait(1000); // Uncomment to see impact on communications
PlayTone(TONE_B3,50);
```

The yellow highlight commands each XL-320 at 200 counts/minute. The XL-320 e-manual says that 0.111 RPM per counts/minute. Hence 200 counts/minute is 22.2 RPM. So, in addition to adjusting the `Wait` statement, one can assign different velocity values in `XL320_servo`.

Exercises

3.1 Set `numberOfWayPoints to 7` in **`xl320-line1_0.nxc`** and the `Wait` statement and velocities in **`xl320-twoLinkFunctions1_0a.h`** using the table below.. Compile, execute, and fill your observations

`numberOfWayPoints`	`Wait`	Velocity [counts/min]	What path does the end-effector make? How well does it "stay" on the desired line?
15	50	200	
15	50	400	
15	100	200	
15	100	400	
5	50	200	
5	50	400	
5	100	200	
5	100	400	

Summary Conclusions:

1. Using way points to characterize points on a line is a simply and intuitive way to move an end-effector on a path
2. While intuitive, the way point approach is naïve; it's a position-control approach which yields a go-stop-go motion
3. Velocity-controlled approaches leverage a motor's "on-the-fly" motion profile. This can yield a smooth motion rather than a go-stop-go one.

There are many methods to implement the last point. The most popular ones define motor velocities based on a polynomial. Examples include cubic and quintic polynomials and B-splines. These methods still use way points, but adjust the velocities as the end-effector passes over them.