Simple Motion Programs

This section shows some very simple motion programs with their associated setup in Power PMAC. Each example in this section includes the on-line commands necessary to set up the coordinate system to run the motion program immediately following it. These examples assume that any motors utilized in the example have already been set up for proper operation.

Example 1: Basic Moves

This first example shows a simple move of an axis out and back.

```
// A typical coordinate system setup for this program is like:
// &1                    // Address C.S. 1
// #1->1000X             // Motor 1 assigned to X-axis, 1000 counts/unit

// Motion program code
open prog 1                    // Open buffer for entry
linear;                        // Linear interpolation move mode
abs;                           // Absolute move mode
ta500;                         // 1/2-second accel/decel time
ts0;                           // No S-curve accel/decel time
f5;                            // Speed of 5 axis units per time unit
X10;                           // Move X-axis to position of 10 units
dwell500;                      // Sit here for 1/2-second
X0;                            // Move X-axis to position of 0 units
dwell500;                      // Sit here for 1/2-second
close                          // Close buffer, end program entry

// To run this program with C.S. 1, issue these on-line commands:
// &1 b1r
```

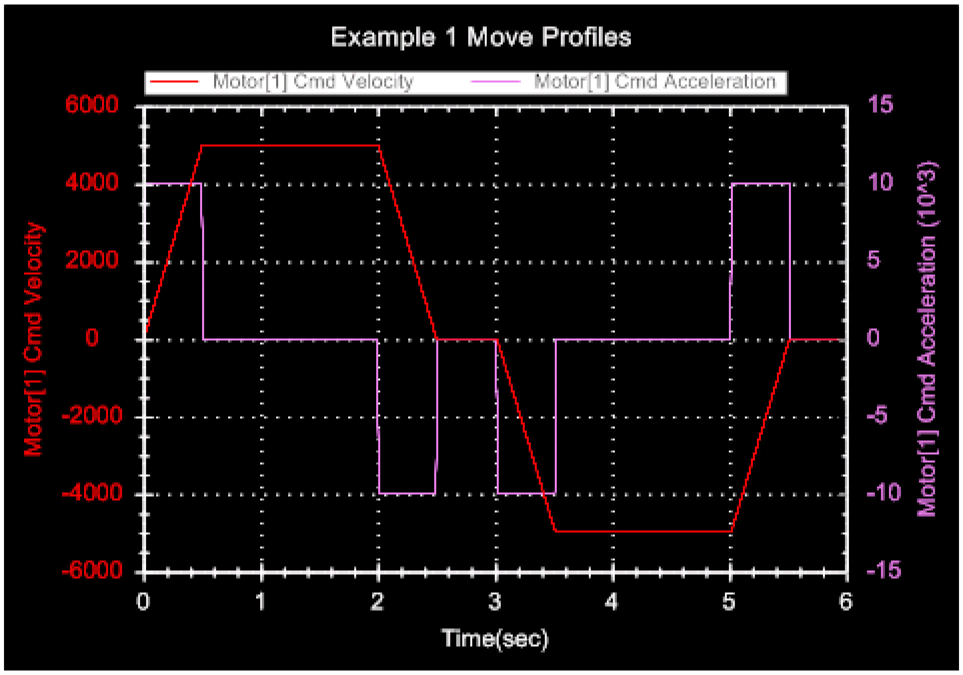
Note several things about this simple example:

- The motion program does not belong to any specific coordinate system, nor does it directly specify which motor(s) it will move. One or more coordinate systems can point to the program and run it (even simultaneously!). Which motor or motors move is dependent on which have already been assigned to the X-axis in the executing coordinate system.

- The program explicitly declares all of the factors affecting the moves. It is possible to rely on defaults for much of this, but it is better where possible to declare explicitly, both because the defaults can change, and because it makes it easier to understand what the program is intended to do. Note that these factors are modal; they do not have to be declared for each move.

- The setup assigns a motor to an axis with 1000 motor units (usually encoder counts) per axis unit. Therefore, each axis unit commanded corresponds to 1000 units of the assigned motor.

The following diagram shows the commanded trajectory generated by execution of this program, gathered from Turbo PMAC's actual execution of the program:



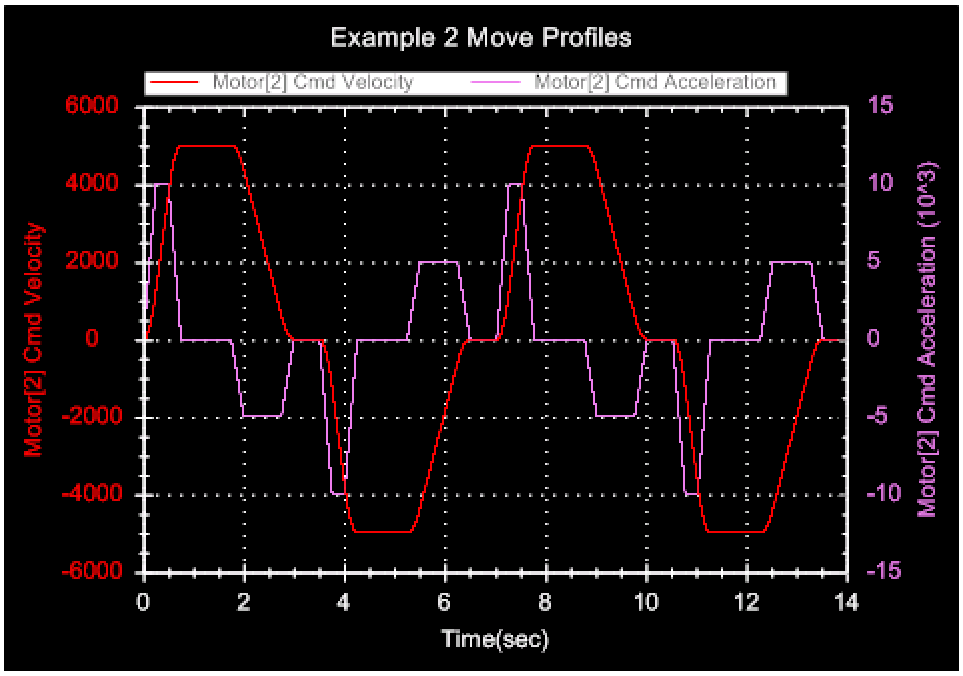
Example 2: A More Complex Move Sequence

This example introduces scaling of axes in user units, incremental end-point specification, time-specification of moves, S-curve acceleration with different acceleration and deceleration times, looping logic, variable use, and simple arithmetic.

```
// A typical coordinate system setup for this program is like:
// &1                      // Address C.S. 1
// #2->1000Y               // Motor 2 assigned to Y-axis, 1000 counts/unit

// Motion program code
open prog 2

local LoopCount;                // Loop counter index

linear;                         // Linear interpolation move mode
inc;                            // Incremental move mode
ta500;                          // 1/2-second accel time
td1000;                         // 1-second decel time
ts250;                          // 1/4-second S-curve accel/decel time
tm2000;                         // 2-second move time
Gather.Enable = 2;              // Start data gathering
LoopCount = 1;                  // Initialize counter index
while (LoopCount <= 10) {       // Loop until false (10 times)
  Y10;                          // Move distance of 10 units positive
  dwell500;                     // Stay here for 1/2-second
  Y-10;                         // Move distance of 10 units negative
  dwell500;                     // Stay here for 1/2-second
  if (LoopCount == 2) Gather.Enable = 0;  // Stop gathering on 2nd loop
  LoopCount++;                  // Increment loop counter
}                               // End of while loop
```

```
close

// To run this program with C.S. 1, issue the following on-line
commands:
// &1 b2r
```

The following diagram shows the commanded trajectory generated by the first two loops of execution of this program, gathered from Power PMAC's actual execution of the program:



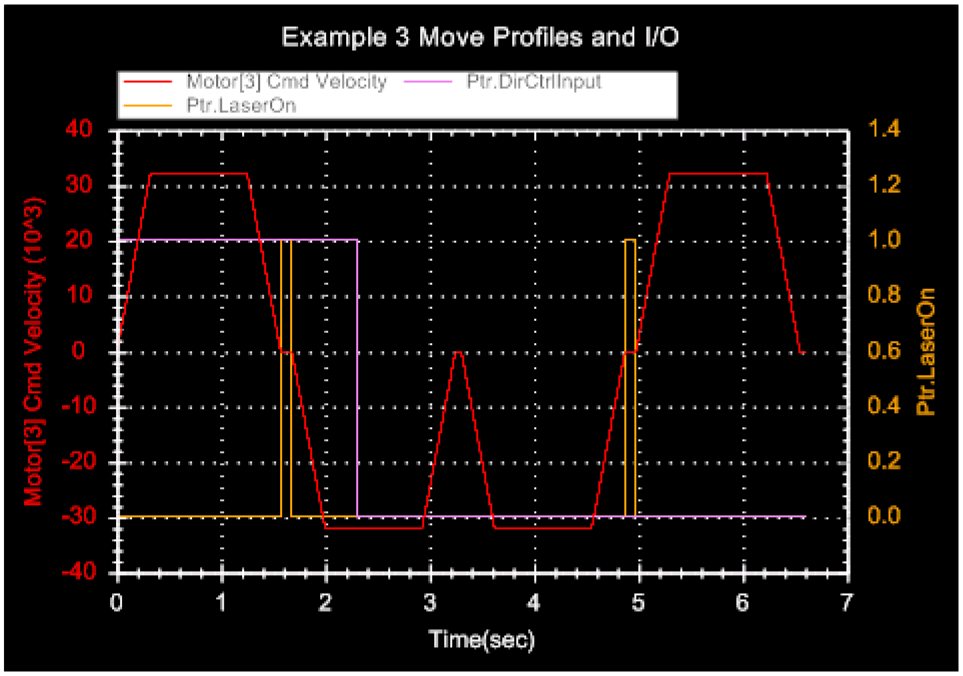
Example 3: Moves with Looping, Branching, and I/O

This next example program introduces conditional branching, calculated move distances, waiting for settling "in position", and I/O addressing.

```
// Variable declarations
ptr LoopCtrlInput->u.io:$A00000.8.1;      // IO Card 0 Point 00
ptr DirCtrlInput->u.io:$A00000.9.1;       // IO Card 0 Point 01
ptr LaserOn->u.io:$A0000C.8.1;            // IO Card 0 Point 24
csglobal MoveLength;                      // Externally settable
variable

MoveLength = 40;                          // Set value for this example

// A typical coordinate system setup for this program is something
like:
// &1
// #3->1000Z                        // Motor 3 assigned to Z-axis, 1000
counts/unit
```

```
// Motion program code
open prog 3

local ThisCs;                 // Local var for # of CS running program

ThisCs = Ldata.Coord;         // Set to # of CS running program
rapid;                              // Rapid move mode
abs;                                // Absolute end-point specification
while (LoopCtrlInput) {       // Loop unit control input false
  if (DirCtrlInput) Z(MoveLength);  // Move in positive direction
  else Z(-1.0*MoveLength);                // Move in negative direction
  while (Coord[ThisCs].InPos == 0) {}     // Wait for settled
  LaserOn = 1;                      // Turn on laser control output
  dwell100;                         // Hold for 100 msec
  LaserOn = 0;                      // Turn off laser control output
  Z0;                               // Return to home position
  dwell50;                          // Hold for 50 msec
}
close
```

The following plot shows the commanded velocity profile along with the direction-control input and the laser-control output.

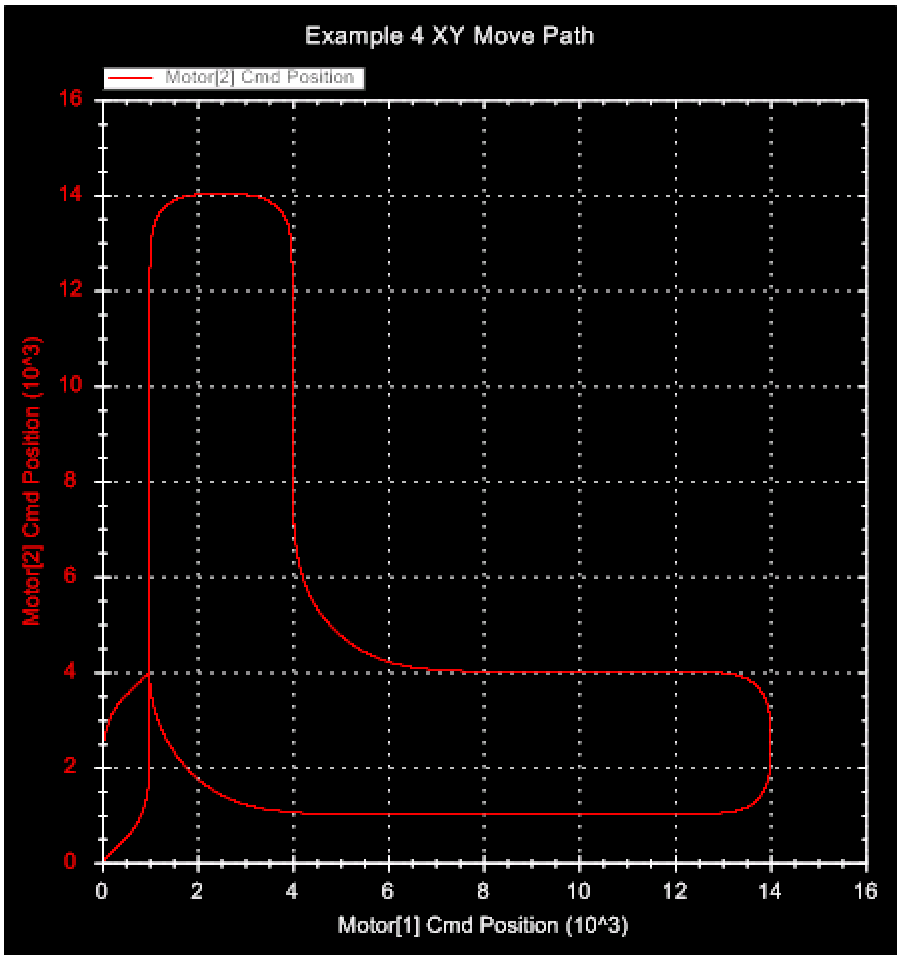


Example 4: Coordinated and Blended Moves with Linear and Circular Interpolation

This example introduces coordinated moves (multiple axes commanded on the same line of a motion program are automatically fully coordinated), blended moves (consecutive moves are automatically blended together unless specifically commanded not to), plus rapid, linear, and circular interpolation in a Cartesian system.

```
ptr DispensePumpOn->u.io:$A0000C.12.1;     //IO Card 0 IO Point 28
Coord[1].SegMoveTime = 5;      // Segmentation time of 5 msec

open prog 4
abs;                           // Absolute end-point specification
normal k-1;                    // XY-plane specification for circles
Gather.Enable = 2;             // Start data gathering
rapid x1 y4;                   // Rapid mode move to (1,4)
dwell 20;                      // Hold position for 20 msec
DispensePumpOn = 1;            // Turn on dispensing pump
dwell 50;                      // Hold position for 50 msec
f50; ta100; ts50;              // Params for linear & circle moves
linear y13;                    // Straight-line move to (1,13)
circle1 x2 y14 i1 j0;          // CW arc to (2,14) about (2,13)
linear x3;                     // Straight-line move to (3,14)
circle1 x4 y13 i0 j-1;         // CW arc to (4,13) about (3,13)
linear y7;                     // Straight-line move to (4,7)
circle2 x7 y4 i3 j0;           // CCW arc to (7,4) about (7,7)
linear x13;                    // Straight-line move to (13,4)
circle1 x14 y3 i0 j-1;         // CW arc to (14,3) about (13,3)
linear y2;                     // Straight-line move to (14,2)
circle1 x13 y1 i-1 j0;         // CW arc to (13,1) about (13,2)
linear x4;                     // Straight-line move to (4,1)
circle1 x1 y4 i0 j3;           // CW arc to (1,4) about (4,4)
dwell 0;                       // Stop blending and lookahead
DispensePumpOn = 0;            // Turn off dispensing pump
rapid x0 y0;                   // Return to home position
Gather.Enable = 0;             // Stop data gathering
close
```

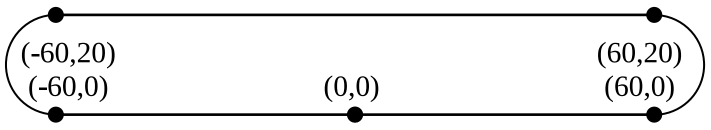
The following plot shows the XY commanded path generated by this program.



Example 5: Coordinated Path Motion

Many applications require the controller to generate a precise path in a plane or in space. Power PMAC controllers have easy and powerful capabilities for doing this. The moves of axes commanded on the same program line are automatically fully coordinated, starting and stopping together and taking the prescribed path. Consecutive moves are automatically blended together (unless specifically told not to).

The most common move modes used for generating paths are linear and circle modes, so called because of the paths they create in a Cartesian system. This example uses linear and circle modes to generate the following oval shape in a Cartesian system.

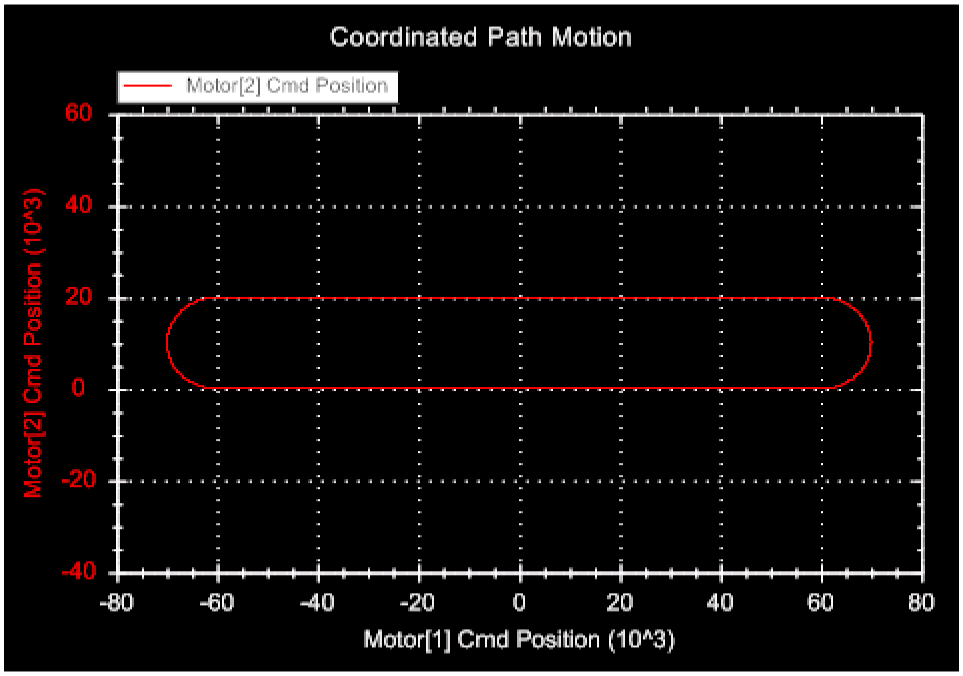


For circular interpolation it is necessary to define the "segmentation time" for the coordinate system – the period at which the exact circle calculations are done. In between these precisely calculated points, a simpler cubic-spline interpolation is done to create new commanded positions every servo cycle. Typically the exact calculations are done every 10 to 20 servo cycles, yielding a very accurate path without overloading the processor from too many trigonometric calculations. Here we set the segmentation time for C.S. 1 to 5 msec with **Coord[1].SegMoveTime** to 5.

```
// Coordinated path motion example
open prog 5
normal k-1;                   // XY plane for circles
abs;                          // Absolute endpoint specification
F200;                         // Vector speed of 200 mm/sec
ta25; ts0;                    // 25 msec accel/decel time, no S-curve
Gather.Enable=2;              // Start data gathering
linear X60 Y0;                // Straight move to (60,0)
circle2 X60 Y20 R10;          // CCW arc of radius 10 to (60,20)
linear X-60 Y20;              // Straight move to (-60,20)
circle2 X-60 Y0 R10;          // CCW arc of radius 10 to (-60,0)
linear X0 Y0;                 // Straight move to (0,0)
dwell 100;                    // Hold position for 100 msec
Gather.Enable=0;              // Stop data gathering
close
```

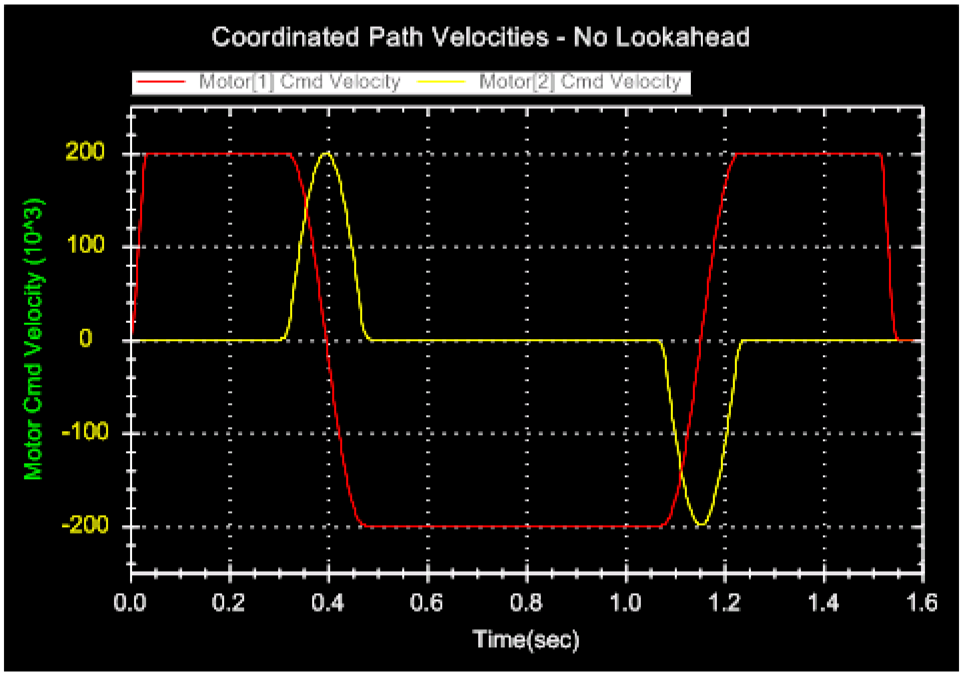
Note that the circular arc moves are programmed here by specifying the radius of the arc. It is also possible to program them by specifying a vector to the center of the arc with I, J, and K components.

The following diagram shows the commanded path generated by execution of this program, gathered from Turbo PMAC's actual execution of the program:

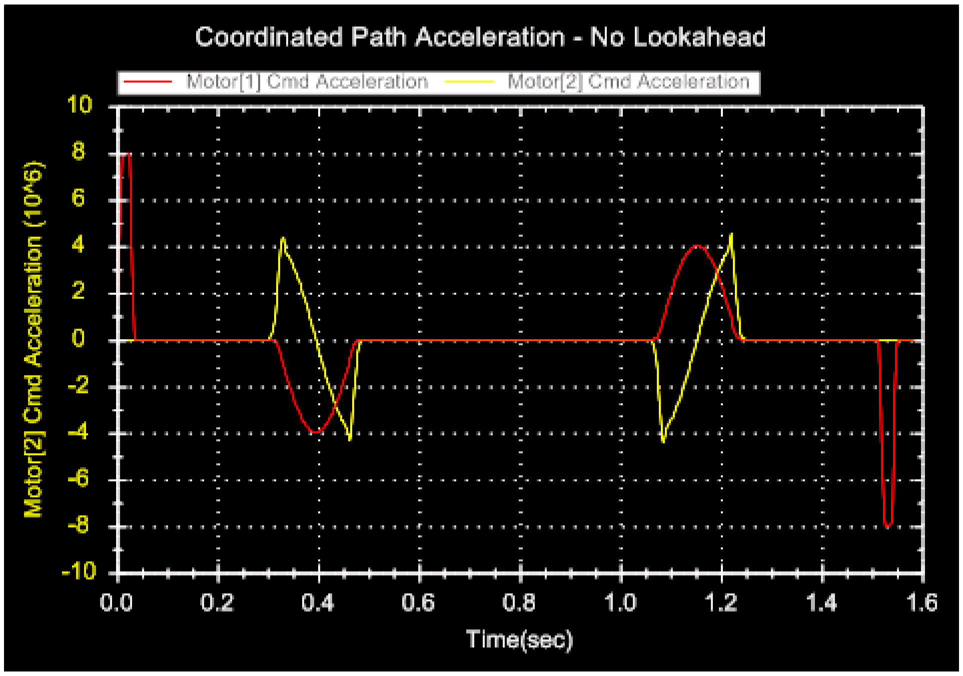


Programmed Move Profiles

This first diagram shows the commanded axis velocity profiles, with a 25-msec acceleration time and without lookahead, gathered from actual execution on Power PMAC:



The next diagram shows the commanded acceleration profiles for the same move sequence:



Buffered Lookahead for Dynamic Limiting

Many times, some parts of the path, such as tight arcs or sharp corners may have to be executed more slowly than other parts because of acceleration constraints. On many systems, figuring out where the system needs to move slowly and how to tell the controller where to move more slowly is so difficult that the whole path is executed slowly.

However, Power PMAC's buffered lookahead feature permits the controller to automatically identify problem areas in the path, compute the highest speed at these areas that do not violate constraints, and to compute the optimum deceleration into these areas and acceleration out of them. The user simply needs to tell Power PMAC what the constraints are and to enable the feature. The problem areas do not need to be identified in the motion program. (While the lookahead algorithm also checks for violations of position and velocity constraints, it is usually the acceleration constraints that are key.)

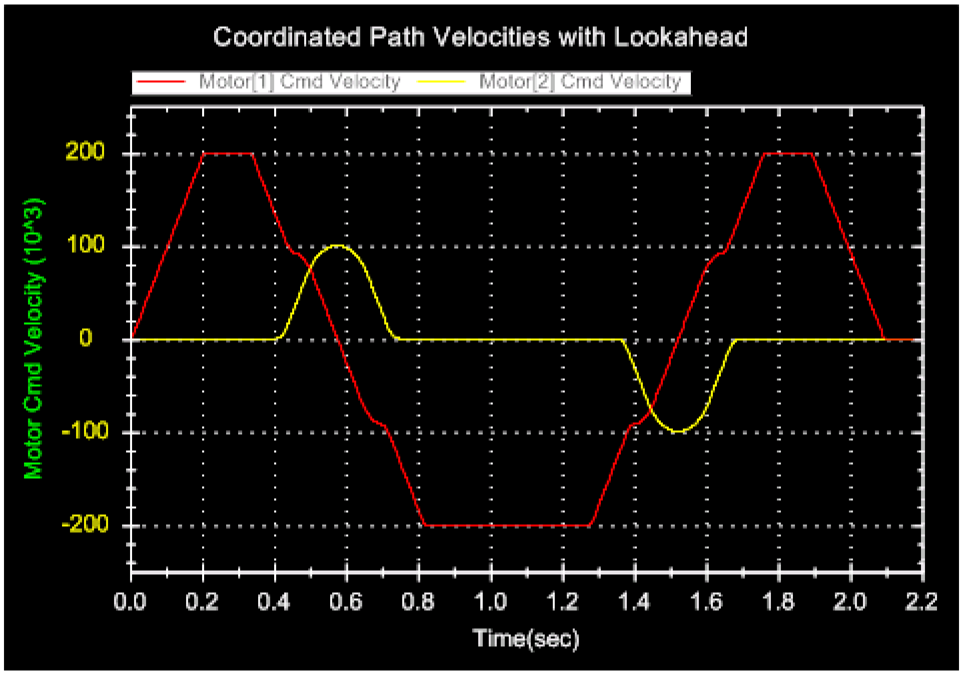
In this example, the X and Y-axes are limited to an acceleration of 1000 mm/sec^2. To execute the arc moves, which have a radius of 10 mm, at the programmed speed of 200 mm/sec would require a centripetal acceleration of 200*200/10 = 4000 mm/sec^2. Note that simply programming a lower speed for these moves would not necessarily solve the problem, because it does not ensure that the deceleration into the arc and the acceleration out of it are handled properly.

For robust acceleration control, the algorithm must look ahead at least the worst case stopping time, which is the maximum velocity divided by the maximum acceleration. In this example, we have a maximum velocity of 500 mm/sec (even though the program only asks for 200), so our worst case stopping time is 500/1000 = 0.5 sec, or 500 msec to decelerate from full speed.

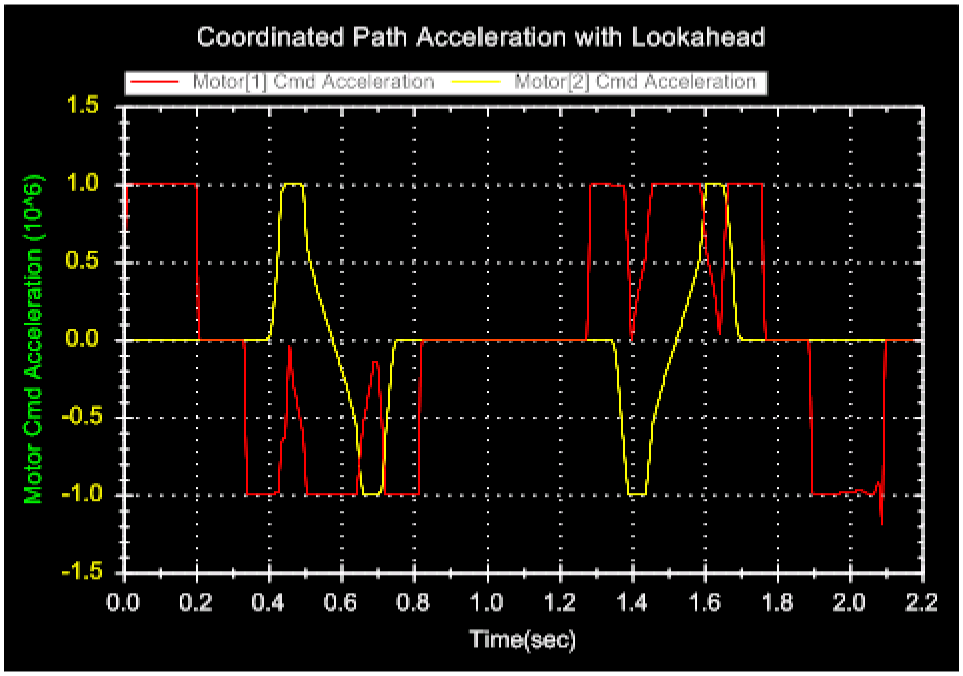
The following commands can be used to set up the buffered lookahead for this example:

```
// Setup for buffered lookahead
Motor[1].MaxSpeed=500;        // 500 motor units per msec
Motor[1].InvAmax=1.0;         // 1 motor unit per msec per msec
Motor[2].MaxSpeed=500;        // 500 motor units per msec
Motor[2].InvAmax=1.0;         // 1 motor unit per msec per msec
Coord[1].SegMoveTime=5;       // Segmentation time of 5 msec
&1 define lookahead 3000      // Set up buffer with 3K segments
Coord[1].LHDistance=200;      // Look ahead 200 segments in path
```

These next two diagrams show the profiles when lookahead has been applied. First we see the commanded velocities. Note that the acceleration ramps to and from a stop have been significantly stretched out compared to the above velocity profiles. Note also that the arc moves have been slowed down, not because their programmed velocities exceeded the lookahead velocity limits, but because their centripetal acceleration at the programmed speed is above the lookahead acceleration limit. Furthermore, the linear moves going into the arcs start slowing down well before the beginning of the arc, and the linear moves out of the arc accelerate for some distance from the end of the arcs.

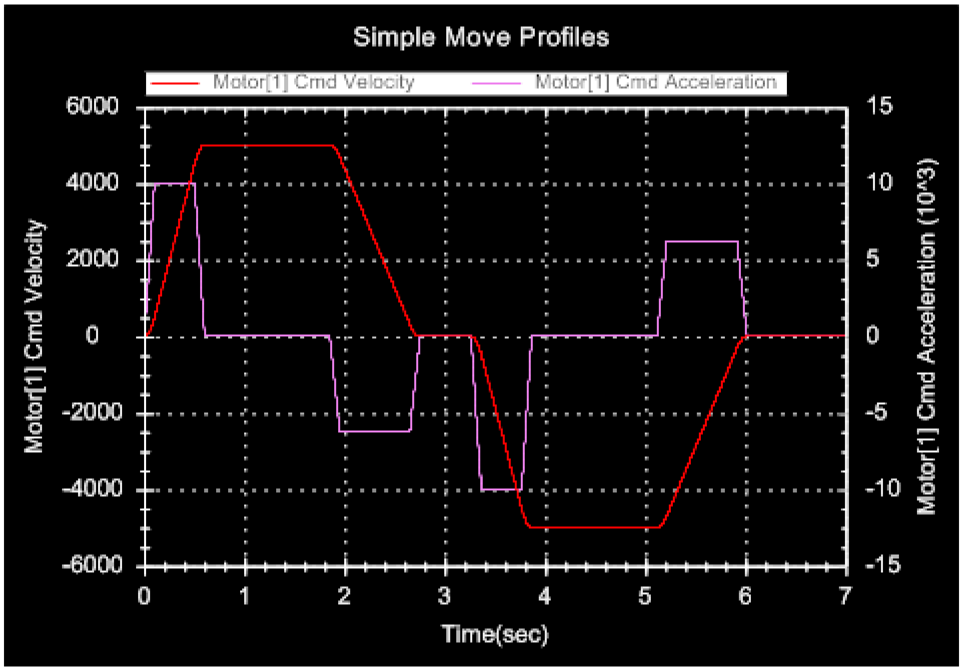


Finally we see the commanded accelerations with lookahead applied. We see that the magnitude of acceleration has been dramatically reduced compared to the plot without limits, above. Note that the acceleration-limiting calculations are approximations, permitting very momentary slight excursions past the specified limits (+/-1,000,000 counts/sec^2) to preserve smoothness in the trajectories.



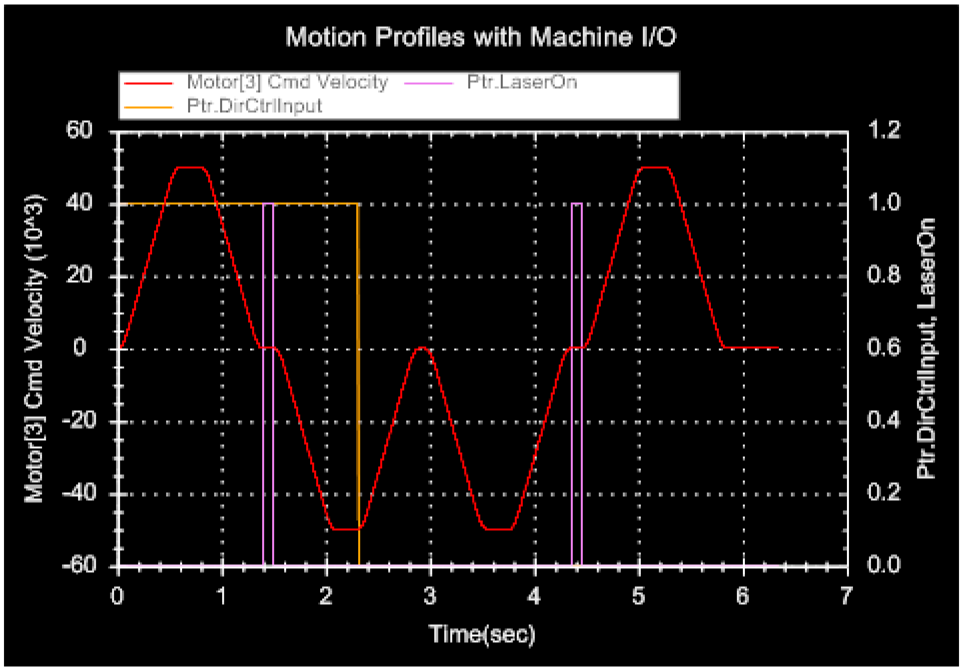
A Simple Move

```
/****************************************/
open prog 1                     // Open buffer for entry
linear;                         // Linear interpolation move mode
abs;                            // Absolute move specification mode
ta500;                          // 0.5 second acceleration time
td800;                          // 0.8 second deceleration time
ts100;                          // 0.1 second S-curve accel/decel time
F5;                             // Speed of 5 axis units per time unit
Gather.Enable = 2;              // Turn on data gathering
X10;                            // Move X-axis to position of 10 units
dwell 500;                      // Sit here for 0.5 seconds
X0;                             // Move X-axis to position of 0 units
dwell 500;                      // Sit here for 0.5 seconds
close
/****************************************/
```



Motion with Related Machine I/O

```
/****************************************/
// Variable declarations
ptr LoopCtrlInput->u.io:$A00000.8.1;      // IO Card 0 Point 00
ptr DirCtrlInput->u.io:$A00000.9.1;       // IO Card 0 Point 01
ptr LaserOn->u.io:$A0000C.8.1;            // IO Card 0 Point 24
csglobal MoveLength;                      // Externally set variable
MoveLength = 40;                          // Set value for this example

open prog 3
local ThisCs;                       // Local var for # of CS
ThisCs = Ldata.Coord;               // Set to # of CS running program
rapid; abs;                         // Rapid move mode, end-point spec
while (LoopCtrlInput) {             // Loop until control input false
  if (DirCtrlInput) Z(MoveLength);  // Move in positive direction
  else Z(-1.0*MoveLength);          // Move in negative direction
  while (!(Coord[ThisCs].InPos)) {} // Wait for settled
  LaserOn = 1;                      // Turn on laser control output
  dwell100;                         // Hold for 100 msec
  LaserOn = 0;                      // Turn off laser control output
  Z0;                               // Return to home position
  dwell50;                          // Hold for 50 msec
}
close
/****************************************/
```



Interactive Jog Control PLC Program

```
/****************************************/
// Interactive motor jog PLC program
// A single motor selected out of #1-#8 can be jogged in "continuous"
// mode, in either the positive or negative direction.
// In continuous mode, the motor will be commanded to jog its
// "deceleration distance" from the present position each scan as long
// as the button is pressed.
// This distance is calculated as V^2/A/2 or V^2xTa/2.
// With S-curve accel specified (JogTs != 0), this distance is
// increased somewhat.

// Declarations of input variables -- application specific
ptr MotorSelectSw->u.io:$A00000.8.3;
ptr JogMinusButton->u.io:$A00000.12.1;
ptr JogPlusButton->u.io:$A00000.13.1;
global JogDecelDist = 500;
/****************************************/
open plc 1

local JogMotor;                     // # of motor selected for jogging
local JogMotorStatus;               // Present state of selected motor

if (JogMotorStatus == 0) {          // No motor jogging?
  JogMotor = MotorSelectSw + 1;     // Read switch to select motor
  Ldata.Motor = JogMotor;           // Specify motor for jog commands
}
if (JogMinusButton && JogMotorStatus <= 0) {
  jog:(-JogDecelDist);
  JogMotorStatus = -1;
}
else {
  if (JogPlusButton && JogMotorStatus >= 0) {
    jog:(JogDecelDist);
    JogMotorStatus = 1;
  }
  else JogMotorStatus = 0;
}
close
/****************************************/
```

SCARA Robot Kinematics

This example shows the forward and inverse-kinematic subroutines for a 4-axis (“shoulder”, “elbow”, “wrist”, and “vertical” SCARA robot. These subroutines permit the user to program robot motion in Cartesian coordinates, with Power PMAC automatically computing the required joint positions to obtain the desired tool-tip path.

```
//====================================================================
// SCARA robot forward kinematic subroutine
// Global variable declarations and values
global Len1 = 400;                  // Shoulder-to-elbow length 400 mm
global Len2 = 300;                  // Elbow-to-wrist length 300 mm
global Zofs = 250;                  // Vertical axis offset 250 mm
global SumLenSqrd;                  // L1^2 + L2^2
global ProdOfLens;                  // 2 * L1 * L2
global DifLenSqrd;                  // L1^2 - L2^2
//====================================================================
open forward (1)                    // Forward kinematics for C.S.1
local SplusE;                       // Sum of shoulder and elbow angles
if (Coord[1].HomeComplete) {        // All motors referenced?
  SplusE = KinPosMotor1 + KinPosMotor2;
  KinPosAxisX = Len1 * cosd(KinPosMotor1) + Len2 * cosd(SplusE);
  KinPosAxisY = Len1 * sind(KinPosMotor1) + Len2 * sind(SplusE);
  KinPosAxisC = SplusE + KinPosMotor3;
  KinPosAxisZ = KinPosMotor4 + Zofs;
  KinAxisUsed = $1C4;               // Mask for C,X,Y,Z values returned
  // Compute intermediate constants for inverse kinematics
  SumLenSqrd = Len1 * Len1 + Len2 * Len2;
  ProdOfLens = 2 * Len1 * Len2;
  DifLenSqrd = Len1 * Len1 - Len2 * Len2;
}
else abort1;                        // Not referenced; stop
close
//====================================================================
// SCARA robot inverse kinematic subroutine
open inverse (1)                    // Inverse kinematics for C.S.1
local X2Y2, Ecos, SplusQ, Qangle;   // Intermediate variables
X2Y2 = KinPosAxisX * KinPosAxisX + KinPosAxisY * KinPosAxisY;
Ecos = (X2Y2 - SumLenSqrd) / ProdOfLens;
if (abs(Ecos) < 0.996) {            // Valid solution w/ 5-deg margin?
  KinPosMotor2 = cosd(Ecos);
  SplusQ = atan2d(KinPosAxisY, KinPosAxisX);
  Qangle = acosd(X2Y2 + DifLenSqrd) / (2 * Len1 * sqrt(SumLenSqrd));
  KinPosMotor1 = SplusQ - Qangle;
  KinPosMotor3 = KinPosAxisC - KinPosMotor1 - KinPosMotor1;
  KinPosMotor4 = KinPosAxisZ - Zofs;
 }
 else abort;                        // No valid solution; stop
close
//====================================================================
```