MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

Working Paper 238 September 1982

Talking to the Puma

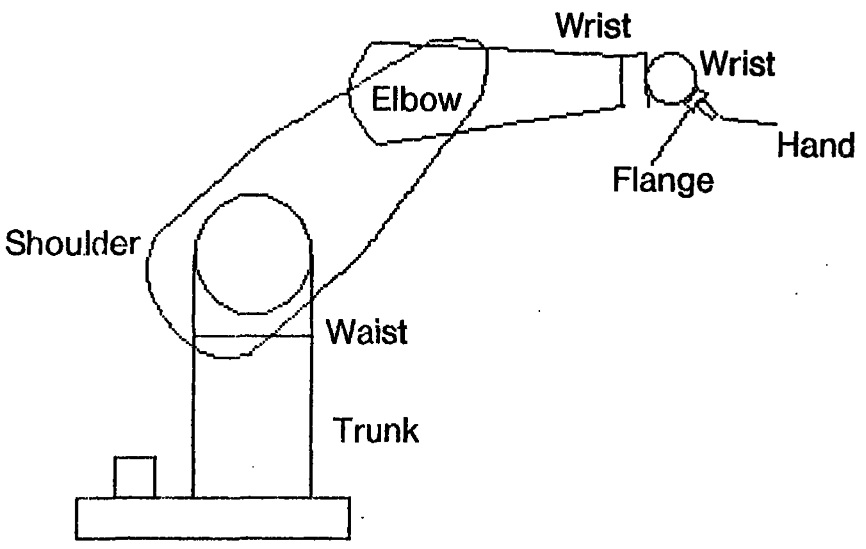
Patrick G. Sobalvarro

Abstract

The AI Lab's Unimation Puma 600 is a general-purpose industrial robot arm that has been interfaced to a Lisp Machine for use in robotics projects at the lab. It has been fitted with a force-sensing wrist. The Puma is capable of moving payloads of up to 5 pounds at up to 1 meter per second, with positioning accuracy to within a millimeter.

This paper is a primer on the control of the Puma from a Lisp Machine. The current Lisp Machine interface is preliminary; the Lisp Machine communicates with the Puma is over a serial line in Unimation's VAL language. The interface will probably change over the next year; however, the commands documented in this paper will probably remain much the same.

Figure 1. The Puma, anthropomorphically labelled.



1. Introduction: The Puma

The AI Lab's Unimation Puma 600 robot arm is a high-quality, general purpose industrial robot arm. It has been fitted with a force-sensing wrist with a sensitivity of about 0.5 ounces. Other hardware associated with the Puma includes a video frame-grabber fitted to a Lisp Machine.

The Puma is a robot arm with six *revolute axes*. This means that it has six joints, each of which rotates about a single axis. Six is a good number of degrees of freedom to have, as we'll discuss later. It happens that the geometry of the Puma lends itself to comparison with human anatomy; the Puma has a waist, a shoulder, an elbow, a wrist, another wrist, and a flange (see figure 1).

Each of these is a joint which rotates about a single axis. You can see that there is some ambiguity in this anthropomorphic naming of the joints, however; for example, while the Puma has only one joint that would be called an elbow, it has two wrists. To avoid just this sort of ambiguity, the joints are numbered as well; the waist is joint number one, the shoulder is joint number two, and so on down to the flange, which is joint number six (figure 1).

When using a manipulator, one must be able to specify the orientation in space of the tool attached to the manipulator as well as its position. For example, if the robot were drilling holes in a curved surface, it might be necessary for it to hold the drill at an orientation normal to the surface. Thus it does not suffice to be able to position the tool at locations in space; it is also necessary to have control over the orientation of the tool.

This is one of the reasons why it is good to have six revolute axes. It takes three joints to hold the tool at an arbitrary orientation in space, and it takes another three joints to place it in an arbitrary position in the space within the manipulator's reach. Why not have more than six joints? With more than six joints there are an infinite number of combinations of joint angles for most positions. While it might be useful to have many combinations of joint angles available, e.g., for the avoidance of obstacles, not much work has been done on controlling such an arm. With less than six joints there are many positions that cannot be reached. With exactly six joints there are usually eight joint-angle combinations that result in a given position and orientation [Horn & Inoue].

There are usually eight joint-angle combinations that will result in a given position and orientation because there are three sets of joints, each of which can assume two configurations that will result in the same terminal position. VAL allows one to choose which configuration each set of joints will assume; thus, the current Lisp Machine interface does the same.

2. Transformations

2.1. Euler Angles

Often when one talks about manipulator tasks one refers to postion-orientation combinations. These combinations are usually called *transformations*. A transformation is only a position-orientation combination, and does not tell which joint-angle combination will be used to attain that position; this joint-angle combination will depend on the current configuration of the robot.

Transformations can be represented in several ways. Often the representation of a transformation seen by the programmer will be a sequence of six numbers; the first three are the Cartesian coordinates (hereinafter referred to as X, Y, and Z) representing the position of the tool, and the other three are Euler angles representing its orientation in space. Although one would think that the term "Euler angle" would indicate that there is a standard interpretation for Euler angles, this is not true. In this paper we will use the interpretation that Unimation does [VAL] when we speak of Euler angles, and we will refer to them respectively as O, A, and T.

X, Y, and Z are measured in millimeters. If the robot's front is the side opposite the motor for joint 1, X is an axis through the center of the trunk, parallel to the ground, increasing to the robot's right. Y is an axis through the center of the trunk, parallel to the ground, increasing to the front of the robot. Z is an axis along the center of the trunk, perpendicular to the ground, increasing upwards. The origin is a point at the center of the trunk at the level of a vector through the center of joint 2 (see figure 2).

O, A, and T often give the newcomer to robotics trouble, but once their meaning is understood, debugging manipulator programs becomes much easier. To understand what O, A, and T represent, let's consider the reference frame of the tool. If the tool attached to the Puma is a two-fingered hand, the reference frame of the tool has its origin at the intersection of an axis Z_T along the axis of the flange, an axis Y_T through the center of the flange and parallel to a line between the two fingertips, and an axis X_T through the center of the flange and perpendicular to Y_T and Z_T (see figure 3).

O, A, and T are rotations of the reference frame of the tool with respect to the world reference frame. O is a rotation about the world Z-axis. It increases in the counter-clockwise direction when looking from positive Z. A is a rotation about the new (after performing the rotation specified by O) X-axis. A increases in the counter-clockwise direction when looking from positive X. T is a rotation about the new Y-axis (after performing the rotations specified by O and A). It increases in the *clockwise* direction when looking from positive Y (see figure 4).

When O, A, and T are all zero, the hand points along the negative Z-axis, rotated so that a line between the two fingers is parallel to the world X-axis; thus the fingers are in a horizontal plane. When O, A, and T are 90, —90, and 0 degrees,

Figure 2. The Puma, showing X, Y, and Z axes.

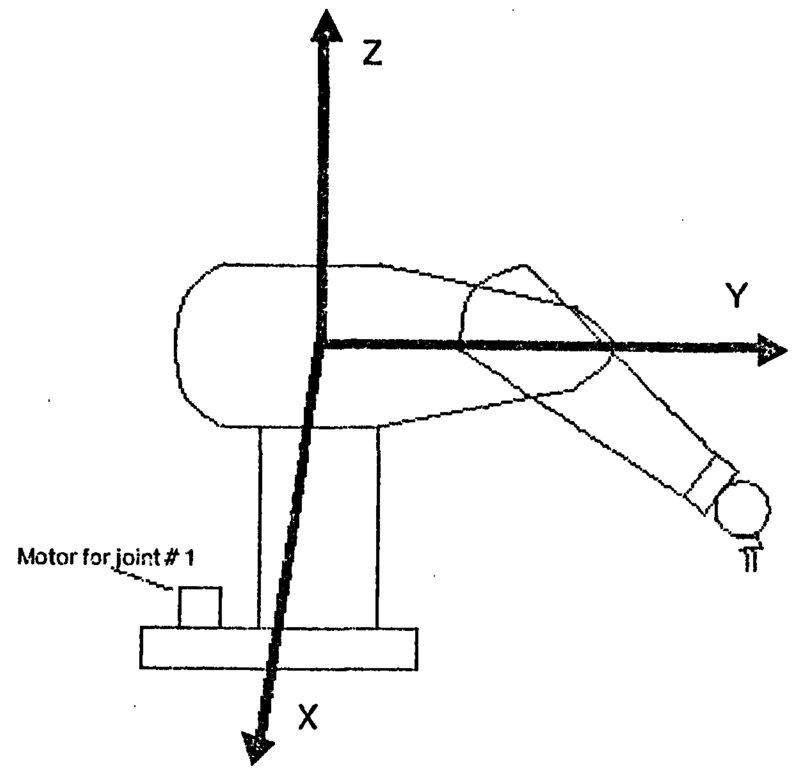
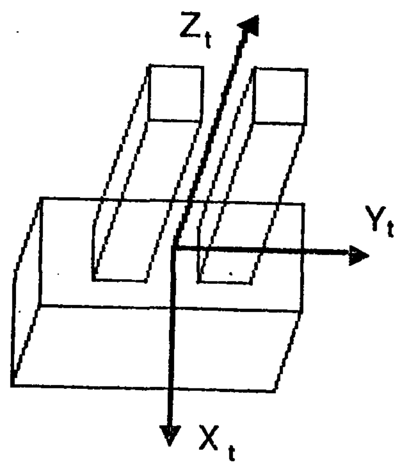


Figure 3. The reference frame of the tool.

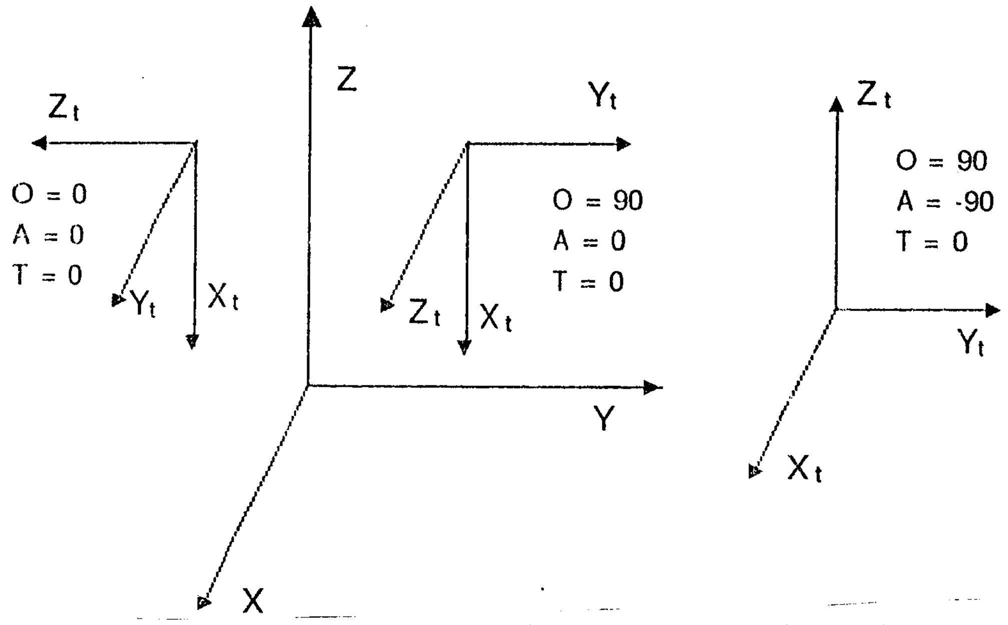


X_T, Y_T, and Z_T are parallel to X, Y, and Z, and increase in the same directions as do X, Y, and Z; the hand points straight up, with the flange rotated so that a line between the fingertips is along the Y-axis.

2.2. Rotation Matrices

Transformations can also be represented as matrices. The orientation of the hand

Figure 4. $O = 90$, $A = -90$, $T = 0$.



can be represented by a 3×3 rotation matrix, which is the product of three simpler matrices, each of which represents a rotation of the reference frame about some axis. The position of the hand can be represented as a 3×1 vector; for ease of manipulation, we can combine the two (position and orientation) as a special case of a three-dimensional homogeneous transformation.

In such a transformation, the rotation matrix occupies the upper left-hand corner of the matrix, and the top three members of the fourth column are X, Y, and Z, which are defined as in the Euler angle representation. The bottom row of the matrix is always (0 0 0 1).[1] So the entire rotation matrix looks like:

$$\begin{pmatrix} m_{11} & m_{12} & m_{13} & X \\ m_{21} & m_{22} & m_{23} & Y \\ m_{31} & m_{32} & m_{33} & Z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

2.3. Transformations and the Lisp Machine Interface

By now you may be wondering what type of transformations we use on the Lisp Machine when talking to the Puma. Do we use Euler angles or rotation matrices? The somewhat surprising answer is that we use both.

Euler angles are the only representation for transformations that VAL understands, but they are somewhat clumsy to manipulate. The rotation matrix form of transformation, on the other hand, allows simple composition of transformations; composition of transformations is matrix multiplication.

[1]The bottom row in a three-dimensional homogeneous transformation usually represents a perspective transformation, with the final element being a scale factor. These are more often of use in computer graphics than in robotics, however.

The solution we have adopted is to use rotation matrices for computation on the Lisp Machine and translate them to Euler angle form when they are passed to the Puma.[2] Transformations that have already been defined are cached locally on the Puma. When they are modified, they are marked as no longer valid, so that they will be updated the next time the software refers to them. What this means for the programmer is that she or he can manipulate and modify rotation-matrix type transformations through the functions provided for doing this, and use them without worrying about whether they have been updated or not; the interface will take care of this automatically.[3]

[2]This may seem slow, but the time taken for the translation step is negligible compared to the time spent in transmitting the transformation to the Puma over a serial line.

[3]While the rotation-matrix transformation is preferred for manipulation, commands are provided that allow the manipulation of the Euler-angle transformation.

3. Using the Puma

3.1. Starting Up

To use the Puma, you must first find the Lisp Machine that is hooked up to it and log in on it. At the current time, this is CADR-25, but this may well change in the future. To load up the interface software, do:

```
(load "<puma>psys.lisp")
```

Then do

```
(make-system 'puma)
```

This will load up the Puma interface, and redefine system-R (for Robot) to select the Puma interaction window. The default Puma interaction window is a frame with two panes. The upper pane is a window on the Lisp Machine's communications with the Puma. The window is provided because communicating with the Puma in VAL over a serial line is a shaky business at best, and, while the current interface is fairly robust, line noise or a dropped character will be easier to recognize with a communications window.[4] The lower pane is a Lisp Listener.

The symbol :*PUMA* has been bound to an object of flavor PUMA; you can think of this object as the Puma's avatar in the Lisp Machine world. You communicate with the Puma by sending messages to this object.

Now that you have loaded up the interface software and selected the Puma interaction window, you are ready to initialize the Puma. Do not turn the Puma 11/02 on before initializing the Puma, or you will cause the interface software to miss part of the initialization dialogue and become confused. If the Puma was already on, you should turn it off at this point.

Now do

```
(send *PUMA* ':INITIALIZE)
```

The message

```
Make sure the Puma 11/02 is connected to your Lisp Machine's
serial line.
Then turn it on.
```

will be displayed.

As soon as you turn on the Puma, the communications pane will show VAL announcing its presence, and the Lisp Machine will tell the Puma controller to initialize itself. When this is done, the Lisp Listener will display the message:

```
Now turn on the arm power and press the COMP button
on the manual control.
Is this done yet? (Y or N)
```

[4]Of course, there is a way to disable this feature.

Once you have pressed the COMP button, type a "Y" or a space to the query. You will notice some activity in the communications window. The Lisp Machine will cause the Puma to be calibrated, and finally the message

```
The Puma has been calibrated and should be ready to receive
commands now.
```

will be displayed. At this point you are in a normal lisp listener, and can use the commands described in the following sections.

If for some reason you should need to type directly to the VAL interface (for example, if a character is dropped, and the interface software becomes confused), you can use the function `(PUMA-TALK-DIRECT)`, which selects the communications pane. You can end direct typein by typing the `END` character.

If at some point you should wish to go through the initialization sequence again (for example, if you power-cycle the Puma controller), you can do:

```
(send :*PUMA* ':INITIALIZE)
```

3.2. Some Cautionary Notes on the Use of the Puma

- Never leave the room while the Puma is executing a program. While the Puma has many safety features designed to keep it from hurting itself or others, there is still the danger that something may go wrong and you will need to disable power manually.
- Do not come within one meter of the Puma while it is executing a program. The table around the Puma has been designed so that the Puma cannot reach you, unless you lean over on it. If you must approach the Puma while it is in the course of moving, **hit the arm power off button (the red button on the front of the controller) first.** This will immediately disable power to the Puma. While the Puma's payload is only five pounds, the manipulator itself is fairly heavy and moves very quickly. An unexpected bug in your program, a power glitch in the controller — any one of these could cause it to do something unplanned, and very quickly.
- To avoid damage to the Puma, be very careful when using the manual control. There is rarely any reason to set the speed knob on the manual control to more than 50. Do not touch the FREE button, or you may seriously damage the arm.
- Never try to determine how strong the arm is by trying to make it lift something heavier than five pounds, or by pushing against it when the power is on. Even when the Puma is not moving, its motors are servoing to keep it in position. If you push against it, they will work harder and harder. If we are lucky, you will blow a fuse. If we are unlucky, you will destroy the power amplifiers.
- When you are finished using the Puma, turn the arm power off (there is a big red button on the Puma 11/02 provided for this) **first**; then turn the controller off.

4. Some Simple Programs

4.1. Pick and Place

Since a transformation defines most of a state of the manipulatoı,[5] we can write simple manipulator programs by telling the robot to move from one transformation to another. One of the most elementary manipulator tasks, called pick and place, involves getting the robot to approach an object, grasp it, move to another position, and place the object there.

Place a bolt on a sheet of styrofoam on the table before the Puma, and, using the manual control,[6] move the Puma to a position where it can grasp the bolt by closing its hand. Don't forget to press the COMP button on the manual control when you've finished; otherwise computer control is disabled and you will get an error the next time you give a motion command. Then do

```
(setq 'bolt-place (send :*PUMA* ':HERE))
```

which asks the Puma to return a transformation which represents its current position. Now move the Puma to the location at which you wish to place the bolt, and set `destination` to that location:

```
(setq 'destination (send :*PUMA* ':HERE))
```

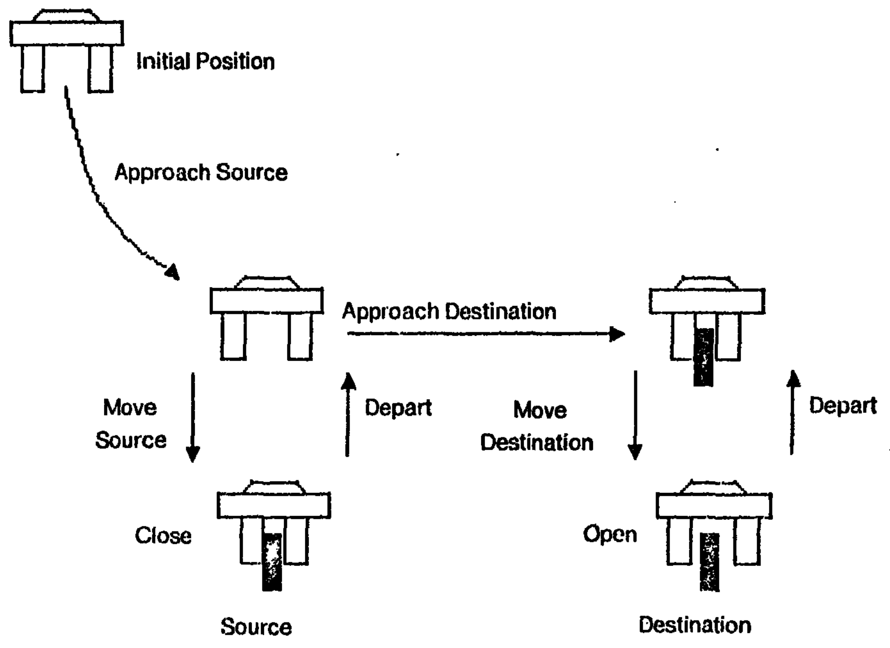
Here is a function that will cause the Puma to pick up the bolt and place it at the new location. The operation is illustrated in figure 5.

```
(defun pick-and-place (source-trans destination-trans)
  ;; move to a point 50mm over the source
  (send :*PUMA* ':APPROACH source-trans 50.)
  ;; actually move to the source
  (send :*PUMA* ':MOVE source-trans)
  ;; grasp the bolt
  (send :*PUMA* ':CLOSE)
  ;; move up 50mm
  (send :*PUMA* ':DEPART 50.)
  ;; move to a point 50mm over the destination
  (send :*PUMA* ':APPROACH destination-trans 50.)
  ;; actually move to the destination
  (send :*PUMA* ':MOVE destination-trans)
  ;; let the bolt go
  (send :*PUMA* ':OPEN)
  ;; move to someplace safe
  (send :*PUMA* ':DEPART 50.))
```

[5]As we mentioned above, a transformation does not define a complete manipulator state, as all the joint angles aren't specified, and any one of 8 combinations may be used, depending on the configuration of the ar ı. Usually, however, this doesn't matter.

[6]For an explanation of the use of the manual control, see the Unimation VAL primer [VAL].

Figure 5. Pick and place.



4.2. Some Interesting Things to do with Transformations

Of course, in order to do anything interesting with a manipulator, it is important to have some ability to generate and manipulate transforms under software control. Here we'll present some simple primitives for manipulating transformations.

We've already seen that it is possible to create a transformation by sending the Puma a `:HERE` message. `NULL-TRANSFORM` is a function that returns the null transform; that is, a transformation whose matrix part is the identity matrix. Because of the way in which O, A, and T are defined, the Euler angles corresponding to the null transformation are 90, —90, and 0 respectively.

```
(COPY-TRANSFORM transform)
```

is a function that returns a transform that is a copy of `transform`; the Euler and rotation matrix parts are both copied, along with the value of the `EULER-UP-TO-DATE?` slot (see the section on writing your own transformation-backing functions for an explanation of how the interface software uses this slot, if you are interested; otherwise, you can safely ignore it).

`SHIFT-TRANSFORM` is a primitive that allows one to modify the Cartesian part of a transformation (X, Y, and Z).

```
(SHIFT-TRANSFORM transform X-offset Y-offset Z-offset)
```

adds the offsets to the respective coordinates in the transformation. It actually modifies the transformation it is given. If you would prefer to make a new transformation, use

```
(SHIFT-TRANSFORM (COPY-TRANSFORM transform) X-offset
                 Y-offset Z-offset)
```

Here is a program that allows one to pick up several bolts arranged at regular intervals along a straight line parallel to the X-axis in front of the manipulator and place them in a bin. First the user uses the :HERE method to define a transformation for the location of the first bolt (the one furthest to the robot's left), and a transformation for the location of the bin to place the bolts in.

```
(defun pick-in-line (how-many interval start bin)
  (dotimes (i how-many)
    (pick-and-place start bin)
    (SHIFT-TRANSFORM start interval 0 0)))
```

4.2.1. Compound Transformations

COMPOUND-TRANSFORMS does something more complex than SHIFT does. In pick-and-place, we used transformations to define locations. Sometimes we may want to be able to describe a location in terms of another location; for example, we might be writing a program for a robot that performs the same task for several objects of the same sort on a table. Then we might want to describe the locations of interest on each of those objects in terms of the location of the object itself, so that we don't have to define a whole set of transformations for each one. For example, if a robot had to insert two brushes in each of several electric motors on a table before it, we could define a transformation for one corner of each motor, and then define two transformations describing the location where the brushes had to be inserted with respect to the corner of a motor.

```
(COMPOUND-TRANSFORMS transform1 &REST transforms)
```

composes the transformations and returns a new transformation made from the result of the operation. It does this by performing a matrix multiplication of the transformations, in the order they are given to it (recall that rotations in three-space, unlike rotations in two-space, do not commute). Thus

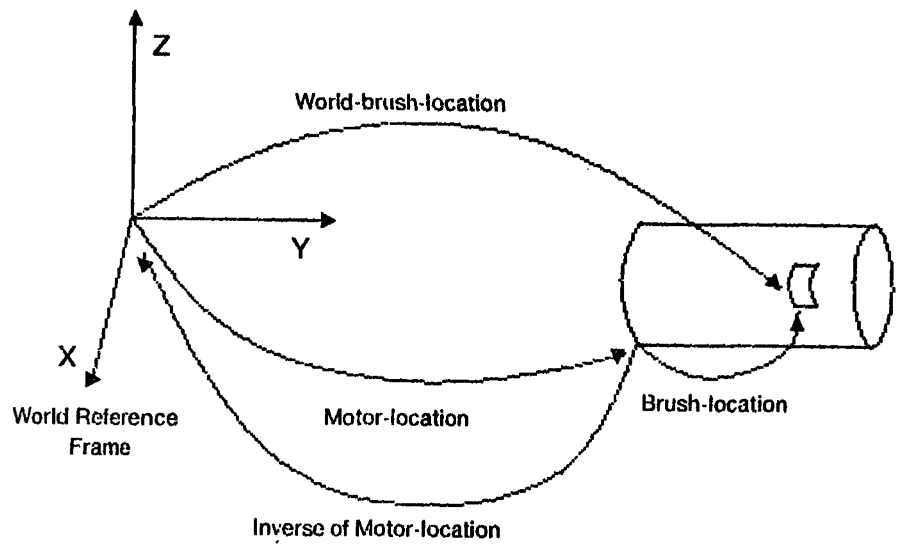
```
(COMPOUND-TRANSFORMS motor-location brush-location)
```

would do what we wanted above, provided that brush-location was defined as a transformation with motor-location as the origin (see figure 6). We will use this primitive to write a program to unload pallets in the next section.

We can use the primitive INVERT-TRANSFORM to define a transformation with respect to another transformation. INVERT-TRANSFORM returns the matrix inverse of a transformation. Since composition of transformations is just matrix multiplication, all we need do if we have a transformation world-brush-location for the location of the brush with respect to the world origin and we want one with respect to the location of the motor is to "compound" it with the inverse of the location of the motor. That is,

```
(COMPOUND-TRANSFORMS (INVERT-TRANSFORM motor-location)
                     world-brush-location)
```

Figure 6. The location of the brush with respect to the location of motor.



Inverse of Motor-location * World-brush-location = Brush-location

will give us a transformation for the position of the brush with respect to the motor (see figure 6).

4.3. Unloading a Pallet

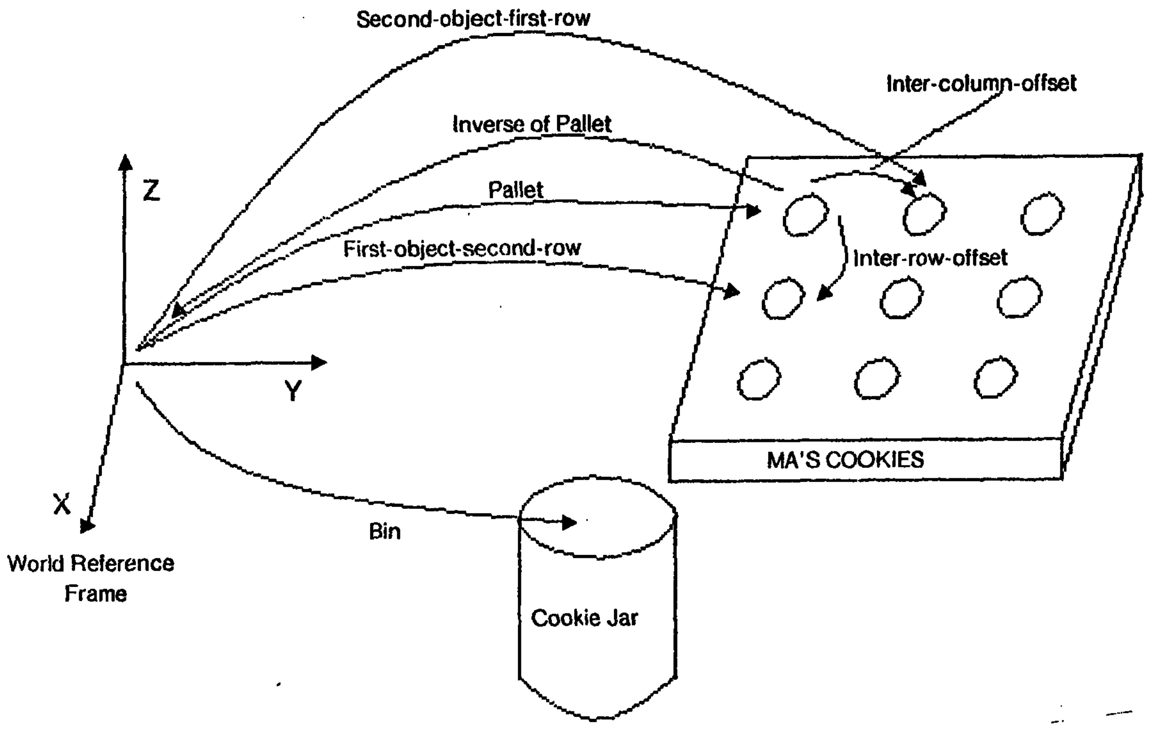
Another common manipulator task is unloading a pallet. An exposition of how a program like this might be coded will be useful for demonstrating how we can use the functions described in the previous section.

Our function will require as arguments the number of columns, the number of rows, and four transformations: the location of the first object in the first row on the pallet, the location of the second object in the first row, the location of the first object in the second row, and a location at which to place the objects unloaded from the pallet.

```
(defun unload-pallet (number-of-rows number-of-columns pallet bin
                                     second-object-first-row
                                     first-object-second-row)
  ;; get the position of the first objects in row and column
  (let* ((pallet-to-base (INVERT-TRANSFORM pallet))
         ;; with respect to the first object on the pallet
         (inter-column-offset
          (COMPOUND-TRANSFORMS pallet-to-base
                               second-object-first-row))
         (inter-row-offset
          (COMPOUND-TRANSFORMS pallet-to-base
                               first-object-second-row)))
    (do ((rows-left number-of-rows (1- rows-left))
         ;; add the offset to the row after each column
         (row (NULL-TRANSFORM)
              (COMPOUND-TRANSFORMS row inter-row-offset)))
        ((zerop rows-left))
      (do ((columns-left number-of-columns (1- columns-left))
           ;; add the offset to the row
           (column (NULL-TRANSFORM)
                   (COMPOUND-TRANSFORMS column
                                        inter-column-offset)))
          ((zerop columns-left))
        ;; the object is at corner * row * column
        (let ((object (COMPOUND-TRANSFORMS pallet row column)))
          ;; this is what causes the action to actually be executed
          (pick-and-place object bin))))))
```

The operation of the program is depicted in figure 7. Note that for the first bolt, row and column are set to the null transformation, so the result of composing them with pallet is a transformation equal to pallet, which is correct.

Figure 7. Unloading a pallet.



5. The Tool Transformation

You may have already realized from our discussion of transformations that a transformation can serve as a frame of reference as well as a position and orientation of the tool. It is precisely this property of transformations that we are taking advantage of when we speak of a transformation relative to another transformation, as in the case of `row` and `column` in the previous section.

The reference frame of the tool (described in section 2.1) is used by certain commands, such as the `:APPROACH` and `:DEPART` methods used in the previous section. These commands result in motion along the axis of the tool; that is, the Z_T-axis. The integer one specifies as the second argument to `:APPROACH` is a negative offset in millimeters along the Z_T-axis from the transformation specified. That is, if 50. were specified as the second argument to `:APPROACH`, the tool would actually be moved to a position shifted along the Z_T-axis in the negative direction by 50. millimeters from the transformation specified.

When the Puma moves to the position and orientation described by a transformation, the thing that assumes that position and orientation is the tool. By default, the tool is located at the center of the flange. What if we affixed a different sort of tool to the arm, not the two-fingered hand we've been using, but, for example, a screwdriver. Then we might want to cause all motions to refer, not to the center of the flange, but to a point 100mm out from the flange in the positive Z_T direction. There is something called the tool transformation that makes this possible.

The tool transformation is a transformation that gives a position and orientation of the tool with respect to the center of the flange. The reason the tool is located at the center of the flange by default is because the tool transformation is the null transformation by default. One can set the tool transformation by doing:

```
        (send :*PUMA* ':TOOL transform)
```

where `transform` is the new tool transformation.

If we wanted to cause all motions to refer to a point 100mm from the flange in the positive Z_T direction, we could do:

```
(send :*PUMA* ':TOOL (SHIFT-TRANSFORM (NULL-TRANSFORM) 0 0 100.))
```

6. Writing Functions for Working with Transformations

- **Note:** The functions documented in this section will change as soon as the interface software ceases to use VAL for communication with the Puma. At that time, a revised version of this memo will be made available.

The functions provided with the interface software for working with transformations are very simple, and may not provide all the capabilities required by some users. For example, the `unload-pallet` program of the last section creates three transformations for every bolt it moves; users whose programs manipulate transformations extensively might want to reuse transformations rather than create new ones. One might also want to be able to manipulate the rotation matrix directly. The purpose of this section is to provide you with the information you will need to do this successfully.

Recall that transformations on the Lisp Machine are cached on the Puma and updated only when necessary, so as to minimize time spent in transferring this information over the serial line. There are functions provided for updating the Euler angle part of a transformation from the rotation matrix part and vice-versa, for asserting that the transformation is not up to date on the Puma, for reading a transformation from the Puma, and for writing a transformation to the Puma.

Transformations on the Lisp Machine are structures. If you `describe` a transformation, you will see something like this:

```
#<TRANSFORM 31751106> is a TRANSFORM
   EULER:                          #<ART-FLOAT-6 31751156>
   EULER-UP-TO-DATE?:              NIL
   MATRIX:                         #<ART-FLOAT-4-4 31751114>
   UP-TO-DATE-ON-PUMA?:            NIL
   INTERNED-ON-PUMA-NAME:          G0259

This is a ART-Q type array.
It is 6 long.
```

As you can see, the Euler and matrix parts are both arrays of type `art-float`. If you wish to modify the matrix part, or replace it, you can use the accessor macro `MATRIX`:

```
      (setf (MATRIX transform) my-matrix)
```

After modifying or setting the transformation's matrix part, you should set `EULER-UP-TO-DATE?` to `NIL`, so that the next time the transformation is used on the Puma the Euler part will be updated from the matrix part, and then, in turn, the copy on the Puma will be updated from the Euler part. It is not necessary to set `UP-TO-DATE-ON-PUMA?` to `NIL` if you have set `EULER-UP-TO-DATE?` to `NIL`, because the code assumes that if the Euler part is not up to date, the cached version on the Puma will not to be up to date.

```
      (UPDATE-TRANSFORM-EULER transform)
```

updates the Euler part of a transformation from the matrix part.

```
        (UPDATE-TRANSFORM-MATRIX transform)
```

updates the matrix part of a transformation from the Euler part. Both set `EULER-UP-TO-DATE?` to `T`.

As mentioned above, there is usually no need to cache a transformation on the Puma explicitly, because the various methods defined in the interface check if this is needed and do so if necessary. It may be useful, however, in demo programs and the like, to define a set of transformations, cache them all sequentially, and then run the program, so as not to have to wait while the transformations are cached during the first execution of the program.

```
        (send :*PUMA* ':CACHE-TRANSFORM transform)
```

will update the Euler part of a transform if necessary, then cache the transformation on the Puma (if it is not up to date there), creating a name for it if necessary, and finally set `UP-TO-DATE-ON-PUMA?` to `T`.

```
        (send :*PUMA* ':UPDATE-TRANSFORM-FROM-CACHED transform)
```

will update the Euler part of a transformation from the cached values on the Puma. If the transformation has no `INTERNED-ON-PUMA-NAME`, one will be created, as in the `:CACHE-TRANSFORM` method. If the `INTERNED-ON-PUMA-NAME` is not recognized by the Puma, the transformation will be given the value of the null transformation.[7] After the Euler part has been updated from the Puma, the matrix part will be updated from the Euler part and `EULER-UP-TO-DATE?` will be set to `T`.

The method `:HERE` uses this method, but user programs will probably never need it unless the user has been working in VAL and wishes to upload a set of saved transformations from the Puma to the Lisp Machine. In this case, the right thing to do is use the transformation constructor macro `MAKE-TRANSFORM`, specifying the name of your transformation as the initial value of `:INTERNED-ON-PUMA-NAME`:

```
        (MAKE-TRANSFORM ':INTERNED-ON-PUMA-NAME my-transform-name)
```

Then a call to the `:UPDATE-TRANSFORM-FROM-CACHED` will retrieve the value of your transformation from the Puma.

[7]This is a side effect of using the VAL `POINT` command.

7. A Glossary of Puma Interface Functions and Methods

7.1. Functions for Creating and Manipulating Transformations

`COMPOUND-TRANSFORMS transform1 &REST transforms`

Creates and returns a new transform from the product of the transformations given it as arguments. The matrix multiplications are performed in the order that the transformations are specified. See section 4.2.1 for a lengthier explanation.

`COPY-TRANSFORM transform`

Returns a new transformation that is a copy of `transform`. The contents of the `EULER`, `MATRIX`, and `EULER-UP-TO-DATE?` slots are copied. Of course, this does not create a transform with the same `EULER` and `MATRIX` parts; rather, it copies the arrays.

`INVERT-TRANSFORM transform`

Creates and returns a transformation whose matrix part is the inverse of the matrix part of `transform`. If `transform` represents the position and orientation of reference frame F with respect to some reference frame R, then the inverse of `transform` represents the position and orientation of R with respect to F. For a description of how this is useful, see section 4.2.1.

`NULL-TRANSFORM`

Returns a transformation whose matrix part is the identity matrix. Thus composing this transformation with any other transformation will return a copy of the other transformation. The Euler part of a null transformation (when up to date) is 90, —90, and 0, for O, A, and T respectively. If it were possible to position the terminal device at the location and orientation represented by the null transformation, the hand would be positioned so that the center of the flange was at the world coordinate system origin (at the intersection of the axes of joints 1 and 2), pointing straight up, with a line between the fingers of the hand parallel to the Y-axis.

`SHIFT-TRANSFORM transform X-offset Y-offset Z-offset`

Modifies `transform` by adding the offsets `X-offset`, `Y-offset`, and `Z-offset` to X, Y, and Z.

`UPDATE-TRANSFORM-EULER transform`

Calculates the Euler part of `transform` from the matrix part and stores it in the Euler part. Sets `EULER-UP-TO-DATE?` to `T`. You probably will not have need to call this function unless you are writing your own functions for dealing with transformations. See section 6.

`UPDATE-TRANSFORM-MATRIX` **`transform`**

Calculates the matrix part of **`transform`** from the Euler part and stores it in the matrix part. Sets `EULER-UP-TO-DATE?` to `T`. You probably will not have need to call this function unless you are writing your own functions for dealing with transformations. See section 6.

7.2. Arm Motion Methods

Most of the arm motion methods are exactly like their VAL equivalents.

`:ALIGN`

The orientation of the tool is changed so that the Z_T-axis is aligned with the nearest world coordinate axis. Thus if the tool had been pointing more or less in the negative Z direction (towards the ground), the execution of an `:ALIGN` method would make it point straight down. This can be useful if one is defining a group of transformations using the manual control.

`:APPROACH` **`transform offset`**
`:APPROACH-STRAIGHT` **`transform offset`**

Like the `:MOVE` and `:MOVE-STRAIGHT` methods (see below), except that, rather than causing the manipulator to move the tool to the location defined by **`transform`**, they cause it to move to that orientation and a position displaced in a negative direction along the Z_T-axis (the axis of the tool) by **`offset`** millimeters. `:APPROACH` and `:APPROACH-STRAIGHT` are useful for approaching an object to be grasped without knocking it or objects close to it over.

`:CLOSE`

Causes the hand to be closed immediately.

`:DEPART` **`offset`**
`:DEPART-STRAIGHT` **`offset`**

The counterparts to the `:APPROACH` and `:APPROACH-STRAIGHT` methods, these cause the manipulator to move `offset` millimeters in a negative direction along the Zt-axis. `:DEPART` does this using joint-interpolated motion; `:DEPART-STRAIGHT` uses straight-line motion.

`:DRAW` **`dx dy dz`**

Moves the tool along a straight line to a position displaced from the current position (in world coordinates) by **`dx`**, **`dy`**, and **`dz`**. The attitude of the tool is maintained constant throughout the motion.

`:DRIVE` **`joint charge speed`**

`joint` must be an integer between 1 and 6; joint number `joint` will be rotated through an angle of **change** degrees at a speed of **speed/100** times the monitor speed.

`:MOVE` **transform**

Causes the roo move to the location and position defined by **transform**, using a joint-interpolated trajectory. Some positions will be unattainable in certain configurations; as of now the software does not know to assume another configuration automatically in these cases. However, if a change in configuration has been requested, the change will take place during the execution of the next `:MOVE` method.

`:MOVE-OPEN` **transform**
`:MOVE-CLOSE` **transform**

These do the same thing as `:MOVE`, but, at some time during the motion, the hand will be opened or closed.

`:MOVE-STRAIGHT` **transform**

Causes the robot to move to the location and position defined by **transform**, but attempts to use a Cartesian trajectory, making any requested changes in tool attitude smoothly. If a change in configuration has been requested by the user, it will *not* take place during the execution of a `:MOVE-STRAIGHT` method.

`:MOVE-STRAIGHT-OPEN` **transform**
`:MOVE-STRAIGHT-CLOSE` **transform**

These do the same thing as `:MOVE-STRAIGHT`, but, at some time during the motion, the hand will be opened or closed.

`:OPEN`

Causes the hand to be opened immediately.

`:READY`

Causes the arm to assume the ready position, in which it is pointing straight up, all joints aligned. This method always succeeds, regardless of the position or configuration of the robot.

7.3. Methods that Deal with Transformations and Locations

`:TOOL` **transform**

The tool transformation is set to **transform**. When the Puma is told to move to some transformation (i.e., assume a position and orientation), what is being positioned is the terminal device, which is at a point at the center of the flange compounded with the tool transform, which is initially the null transformation. The `:TOOL` method allows you to set the tool transformation to account for the size and orientation of a terminal device.

`:BASE dx dy dz zrot`

Translate the world reference frame by **dx**, **dy**, and **dz**, and rotate it about the *Z*-axis by **zrot**. All motion commands are affected by the new base, as well as information that the robot returns about its state (such as the transformation returned by `:HERE`). Note that to reset the base one must negate the original arguments, not give arguments of 0.

`:HERE`

Returns a transformation that describes the current position and orientation of the terminal device. The transformation returned is affected both by the setting of the tool transformation and any translation of the base.

7.4. Methods to Control Configuration and Speed

`:SPEED speed`

The monitor speed is set to **speed**, which must be an integer. The monitor speed can be between 1 and 300 (decimal), although it is recommended that it not be set to more than 100. The precise meaning of the monitor speed depends on whether the motion being performed is straight-line or joint-interpolated. The initial monitor speed setting is 100.

`:ABOVE`
`:BELOW`

The configuration of the robot is set to cause the "elbow" (joint 3) to point either upward or downward. The actual change in configuration will take place during the next joint-interpolated motion command. The default state is `:ABOVE`.

`:FLIP`
`:NOFLIP`

The configuration of the robot is set so that the range of angles assumed by joint 3 is constrained to be positive (`:NOFLIP`) or negative (`:FLIP`). The actual change in configuration will take place during the next joint-interpolated motion command. The default state is `:NOFLIP`.

`:LEFTY`
`:RIGHTY`

The configuration of the robot is set to that it will resemble either a left arm or a right arm. This makes it possible to reach positions that would normally require joint 1 to move out of range. The actual change in configuration will take place during the next joint-interpolated motion command. The default state is `:RIGHTY`.

7.5. Miscellaneous Methods

`:CALIBRATE`

Causes the arm to be calibrated. Whenever the Puma 11/02 is turned on, the position and orientation of the arm is not precisely known until it has been calibrated.

`:INITIALIZE`

This method should be called just before powering up the arm. It goes through the initialization dialogue with the arm, and calls the `:CALIBRATE` method.

Bibliography

[Horn&Inoue]

Berthold K. P. Horn and Hirochika Inoue, *Kinematics of the MIT-AI-VICARM Manipulator*, MIT Artificial Intelligence Laboratory Working Paper 69, May 1974

[Paul]

Richard P. Paul, *Robot Manipulators: Mathematics, Programming, and Control*, MIT Press, Cambridge, Massachusetts, 1981

[VAL]

User's Guide to VAL, Unimation Inc., Danbury, Connecticut, June 1980