

EURON 2002 Summer School on Visual Servoing

Benicàssim, September 16-20, 2002

Laboratory Worknotes

Homogeneous transforms

The transformation, translation and/or rotation, of a frame F_j with respect to another frame F_i is represented by the (4x4) homogeneous transformation matrix ${}^i T_j$. Translation is defined by Cartesian coordinates, but different representations are available for rotation, e.g. Euler angles, Roll-Pitch-Yaw angles and quaternions. In the following, unless otherwise stated, orientations are given in RPY angles.

As for other conventions, lengths are expressed in meters, and angles in radians.

1. Define the world frame as $T_0 = ht$, and draw it with `plot(T0, '0')`. Axes are color-coded: x is red, y is green, z is blue. The second parameter (optional) of the plot function is a text label which is displayed next to the frame arrows.
2. Define a new frame with a pure translation
`T1 = ht('xyz', 0.5, -0.4, 0.3)`
and plot it, holding the previous one.
3. Define a new frame with a pure rotation about the **x** axis
`T2 = ht('xrot', pi/6)`
and plot it, holding the previous one.
4. Compose the previous frames into a new one
`T3 = T1 * T2`
5. Verify the non-commutativeness of frame composition
`T3b = T2 * T1`
6. The end-effector of a robot arm is located at point (-0.2, 0.3, 0.8) with orientation ($\pi/6$, $\pi/2$, $-\pi/12$). A camera is mounted on the end-effector at the relative coordinates (0.03, -0.02, 0.5) and orientation ($\pi/4$, 0, 0). Write a script named `frames.m` to create and plot the world, end-effector, and camera frames.

NOTATION: names of Matlab variables for homogeneous transforms.

- Let transform ${}^i T_j$ be stored in a variable named T_{ij}
- Let us define the following one-character labels for frames:
 - '0' is the world frame,
 - 'e' is the end-effector frame, and
 - 'c' is the camera frame.

Solution:

```
T0 = ht
clf
plot(T0, 'world')
hold on
T0e = ht('xyzrpy', -0.2, 0.3, 0.8, pi/6, pi/2, pi/12)
plot(T0e, 'end-effector')
Tec = ht('xyzrpy', 0.03, -0.02, 0.5, pi/4, 0, 0)
T0c = T0e * Tec
plot(T0c, 'camera')
axis equal
```

Polyhedra in space

A polyhedron is defined by its vertices, i.e., a set of rigidly-attached points, each point being defined by a (3×1) column vector of Cartesian coordinates, with respect to the world frame. Homogeneous coordinates are used to store the vertices. Optionally, faces can be stored, and their edges will be plotted too.

1. A single-face flat square is created with

```
p = polyhedra([ ...
    0.25 -0.25 0; ...
    0.25  0.25 0; ...
   -0.25  0.25 0; ...
   -0.25 -0.25 0] , ...
    [1 2 3 4])
```

and drawn with `plot(p)`. Write a script named `poly0.m`, which does that, and displays the frame too.

Solution:

```
T0 = ht
p = ...
clf;
plot(T0, 'world');
plot(p);
axis equal;
```

2. Several polygons are pre-defined:

- `polyhedra('tetrahedron', length);`
- `polyhedra('polygon', edges, length);`
- `polyhedra('pyramid', edges, length, height);`

3. Polyhedra can be plotted with respect to a given frame by using `plot(p,T)`, where `T` is the homogenous transformation of such frame. Write a new script named `poly1.m`, based on the previous one, which defines a new frame `T0o`, translated by (0.9, -0.7, 0.8) meters and rotated by $(-\pi/3, \pi/2, -\pi/6)$ radians with respect to the world frame, and plots a new polyhedron with respect to that new frame.

Solution:

```
poly0;
T0o = ht('xyzrpy', 0.9, -0.7, 0.8, -pi/3, pi/2, pi/6)
hold on
plot(T0o, 'object');
plot(p, T0o);
axis equal;
```

4. Now the polyhedron is located 1.5 meters away from the camera frame (as computed in exercise 6 of previous section) along the z-axis. Write a new script named `poly2.m`, to define the object-to-camera frame, compute the transformation between the object and the world frame, and display them all.

Solution:

```
frames;
p = ...
Tco = ht('xyz', 0, 0, 1.5)
T0o = T0e * Tec * Tco
hold on;
```

```
plot(T0o,'object');  
plot(p,T0o);  
axis equal;
```

Motion of a rigid body

The velocity of a body in space is described by a screw, which consists of two components:

- v_i representing the linear velocity at O_i with respect to the fixed frame F_0 ;
- ω_i representing the angular velocity of the body with respect to frame F_0 .

Those components can be concatenated to form the kinematic screw vector V_i , also called *twist* or *spatial velocity*.

1. Write a script named `screw.m`, which computes the motion of each point of a tetrahedron located at the origin of the world frame, when the screw $(-0.05, 0, 0, 0, 0, 0.5)$ is applied to it, and displays the result graphically. Use the following statements:

```
clf;
T0 = ht;
p = polyhedra('tetrahedron',0.5);
plot(T0);
plot(p,T0);
vf = vfield(p,[-0.01 0 0]',[0 0 0.3]');
plotscrew(p,vf);
axis equal;
```

2. Run CameraMotion simulation model. Initially, the camera is frame is located at $(0, 0, -1.5)$ meters. The screw consists of just a constant linear velocity of 0.3 m/s along the z-axis. Since the default simulation time is 10 seconds, the final camera frame will be located at $(0, 0, 1.5)$. The camera frame is a Simulink signal, which is stored in Matlab workspace, in a variable called `T0c`. It holds a structure with various information, including obviously the signal values, which are given by `T0c.signals.values`. Write a script named `cam_traj.m` to plot the trajectory, using the following statements:

```
n = size(T0c.signals.values,3);
clf;
plot(ht(T0c.signals.values(:,:,1)),'start');
hold on;
for i=2:5:n-1
    plot(ht(T0c.signals.values(:,:,i)));
end
plot(ht(T0c.signals.values(:,:,end)),'end');
axis equal;
axis off;
```

3. A small camera can be added to the plot with the following statements:

```
c = camera;
plot(c,ht(T0c.signals.values(:,:,1)));
plot(c,ht(T0c.signals.values(:,:,end)));
```

4. Change the value of the screw to $[0.2 \ -0.4 \ 0 \ 0 \ 0 \ 0]'$, rerun and display the resulting trajectory.
5. Try now a pure rotation with screw $[0 \ 0 \ 0 \ 0 \ 0 \ 4.5\pi/180]'$.
6. Command a trajectory with translation and rotation, using the screw $[1.5\pi/10 \ 0 \ 0 \ 0 \ -\pi/10 \ 0]'$. What motion is going to be executed? Recall that the screw is expressed *in the current camera frame*.
7. Experiment yourself with more complicated screws and trajectories!

Camera viewing

The image of an object observed by a camera is built upon the perspective projection of the points onto the image plane, i.e. a simple pin-hole camera model. Intrinsic parameters (F_u , F_v , u_0 , v_0) and image resolution (H_{res} , V_{res}) need to be input to the model.

In order to generate an image, both camera and object frames need to be provided, as well as the camera intrinsic parameters, and the polyhedral object model. Camera parameters can be included in the definition function call:

```
c = camera(F_u, F_v, u_0, v_0, hres, vres)
```

1. Based upon the data computed by script `poly2.m`, write a new script named `view1.m`, which shows the object as seen by the camera, with the following statements:

```
c = camera(1000,1000,256,256,512,512);  
figure (2); % figure 1 displays the 3D setup  
view(c,T0c,p,T0o);
```

2. Open `ObjectImaging` simulation model. Since the pose of the camera might change, and the output signal will be also called `T0c`, rename the input variable holding the initial camera pose to:

```
T0c_0 = T0c;
```

before starting the simulation. While running, the newly opened window is the object image, as seen by the camera.

3. Initially, the camera slightly approaches the object. Change the screw in order to obtain other trajectories. The camera trajectory can be displayed with the previous script `cam_traj`, or using the function `trajectory(c, T0c)`, while the trajectories of image points are plotted with `plotimg(c,img)`.

The visual servoing browser (vsbrowser)

The purpose of this tool is to manually move a camera around an object, and visualize simultaneously the camera image and the poses of both the camera and the object in 3D space. In this way, it is easy to define the starting and desired positions of the camera in a visual servoing task.

When the command `vsbrowser` is issued, a file dialog appears, where a MAT-file needs to be input. This file must contain the following variables:

- `c` camera object
- `T0c` pose of the camera in the world frame
- `p` polyhedra object
- `T0o` pose of the object in the world frame

For example, use the predefined file `vsdemo.mat`. The user interface allows the motion of the camera in discrete steps, translating / rotating the camera along / around the Cartesian axes of either the camera frame or the object frame. Steps are measured in meters or radians, and are fully editable.

The current setup (variables `c`, `T0c`, `p`, and `T0o`, though only `T0c` changes actually) can be saved at any moment, or discarded by loading another setup from disk, or closing the window.

1. Use the data computed in script `view1.m` to define a setup named `v1_setup.mat`, and visualize it using the browser.

Solution:

```
view1;  
save v1_setup c T0c p T0o  
(load v1_setup in browser)
```

2. Based on `vsdemo.mat`, move the camera and save different setups changing its position and / or orientation. Name the resulting files as `pose2`, `pose3`, etc. Predefined files `pose0` and `pose1` contain two setups for testing purposes. All these poses will be used in next section for visual servoing simulations.

A first visual servoing model

In this section a predefined Simulink model of a visual servoing task is used. The objective is to learn how to set up and run one such simulation. In the next section, the code corresponding to the Jacobian matrix and control law will need to be developed.

Open model IBVS. It is a simple 2D-point visual servoing task, where the camera has to reach a desired position.

Block colors use a simple code according to its main purpose:

- Red blocks are inputs.
- Green blocks are outputs.
- Yellow blocks contain simulation code corresponding to the *physical* part of the task.
- Magenta blocks are the simulation code of the *computing* part of the task, i.e. the code which should be run on the computer in a real-world task.

Let us now define the data for a task going from `pose1` to `pose0`, as defined in the previous section, with the following commands:

```
load pose0
T0c_x = T0c;          % desired camera pose
load pose1
T0c_0 = T0c;          % initial camera pose
lambda = 0.125;       % gain of the control law
```

Run the simulation. Besides displaying the camera image at runtime, data is stored in the output variables `img`, `img_x`, `cVc`, and `T0c`.

Consequently, all the information of a visual servoing task can be stored with the single Matlab command:

```
save <filename> T0c_0 T0c_x c T0o p lambda img img_x cVc T0c tout
```

Results can be graphically analyzed by means of the following functions:

- `plotimg(c,img {,img_x})` displays the trajectory of points in the image plane
- `trajectory(c,T0c)` displays the 3D trajectory of the camera
- `ploterr(img,img_x)` displays the error of the feature vector ($s-s^*$)
- `plotscrew(cVc)` plots the kinematic screw (translation and rotation)

Perform other simulations using the camera poses defined in the previous section.

Several assumptions have been made in this model:

- The simulation is continuous, though it can be turn to discrete easily by changing the *Sample time* parameter of the block *Camera view* to any positive value in seconds (-1 returns to continuous).
- Depth of points is directly extracted from their 3D coordinates.
- Real (not estimated) camera intrinsic parameters are used in the computation of the Jacobian.
- The target image is generated from the desired camera pose.

Coding the visual servoing control law

Now, the code corresponding to the Jacobian matrix and control law has to be developed. Open model `IBVShandcoded`. The only difference with that of the previous section is the substitution of the blocks which compute the Jacobian and the control law, by a single block named “Hand-coded control law”. This block is just an envelope for the user function `ContrLaw.m` which has to contain the Matlab code to compute the kinematic screw of the camera.

Starting from the skeleton of this function `ContrLaw_skeleton.m`, complete the necessary code and test the simulation model with the task defined in the previous section. Hopefully, the results must be 100% coincident.

Modifications to the visual servoing loop

Based on the first visual servoing model IBVS, this section introduces some variations on the model structure, in order to simulate more advanced setups:

- **IBVSeq**: computes the interaction (Jacobian) matrix with the target (*equilibrium*) instead of the current data.
- **IBVStreck**: adds motion to the object, and a proportional-integral control law. Take into account that the camera starts at the equilibrium point, i.e., $T0c_0 = T0c_x$.
- **IBVSendeff**: introduces the end-effector frame, different of the camera frame. You should define the following variables: Tec , $T0e_0$, $T0e_x$. Sample definitions are given in MAT-file `task_endeff`. Output variables are now: eVe , $T0e$. Some scripts need to be modified to plot the camera frame appropriately.
- **IBVSstereo**: based on the previous one, uses a multicamera model to simulate a stereo setup. As in the previous setup, some new variables need to be defined, namely: $Telc$, $Terc$, lc , rc . Sample definitions are given in MAT-file `task_stereo`.

Visual servoing with 3D features

This section introduces 3D information in the visual servoing model. Such information is obtained from a CAD model of the observed object, and a 3D reconstruction algorithm (Dementhon's). Two kind of features are used:

- PBVS: the feature is the pose of the object, as obtained by the reconstruction algorithm.
- IBVS3Dp: the features are the 3D coordinates of the points in the object, obtained from its pose and the coordinates of the CAD model.