

University of Burgundy

Masters in Computer Vision and Robotics

Visual Servoing Module

Project Report for Visual Servoing from Lines

By:

Vamshi Kodipaka

vamshikodipaka@gmail.com

Supervisor: Dr. Omar Tahri

Dated: 24-Dec-2019



VISUAL SERVOING FROM LINES

Solution 1:

Two. Because only one degree of freedom is correspond to the depth h of the camera, other two parameters are directly considered from the image plane, which are then controlled using a single normal vector to the interpretation plane.

Solution 2:

The answer is *three*.

Solution 3:

1. Propose a restriction on the classical Plucker coordinate which explicitly splits the representation in such way that the depth becomes an independent variable. This will allow the definition of the notion of image line alignment.
2. The definition of the control law on the related sub manifold and this is prove of its stability. The use of bi-normalize Plucker coordinates yield to a partial decoupling between the rotation and translation control. The control law consist then of an analytic inversion of the motion equations and the stability proof exhibits global analytic conditions for stationarity.

VS Looping and h-representation:

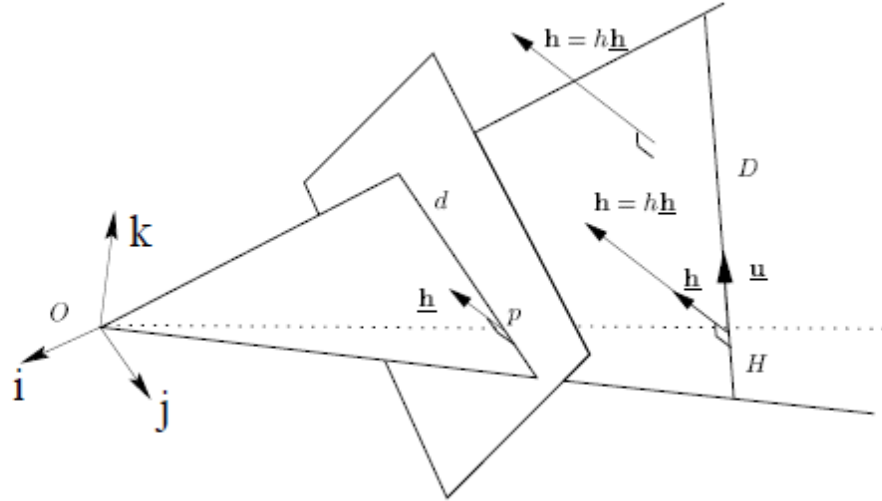


Fig.1: Geometric interpretation of the normalized Plucker coordinates of a line

The normalized Plucker coordinates are of a 3D line defined by the projective Plucker coordinates (u,v):

$$L = (u,v) \quad \dots(1)$$

$$u^T h = 0$$

$$u^T u = 1$$

..(2)

Where $\mathbf{u} = \mathbf{u}/\|\mathbf{u}\|$ and $\mathbf{h} = \mathbf{v}/\|\mathbf{u}\|$

It can be easily shown given a 3D point P when: $\mathbf{h} = \mathbf{P} \times \mathbf{u}$

Then the equivalence holds: $h = 0$ passes through the origin

Solution 4: *Implementation of VS loop using three lines*

Here I used the *RobotMotionController* Toolbox of matlab. This toolbox has the Visual Servoing toolkit which helped me to perform VS using three lines. The coded is commented everywhere necessary and cited.

Initial set parameters::

1. 'example'	Use set of canned parameters
2. 'niter',N	Maximum number of iterations
3. 'eterm',E	Terminate when norm of feature error < E
4. 'lambda',L	Control gain, +ve definite scalar/matrix
5. 'T0',T	The initial pose
6. 'Tf',T	final camera pose to determine desired
7. 'P',p	The set of world points (3xN)
8. 'planes',P	World planes holding the lines (4xN)
9. 'fps',F	Number of simulation frames per second

Steps:

1. Setting Tf to default.
2. Set camera back to its initial pose and h normal.
3. final pose is specified in terms of a camera-target pose using `getline()` : control scheme is explained in the code and Refer Page.16 from [1].
 - a. compute lines and their slope and intercept
 - b. compute rho and theta and also dimensions of the target.
 - c. initiate step. Simulate one time step useful for the termination
 - d. plot current line and desired lines
 - e. compute image plane error with respect to both normal
4. compute the velocity of camera in camera frame
5. update the camera pose using differential motion
6. update the external view camera pose
7. Repeat from step3 if elapse time is not finished.

8. Remember Image Jacobian condition is inherited from Visual Servo class

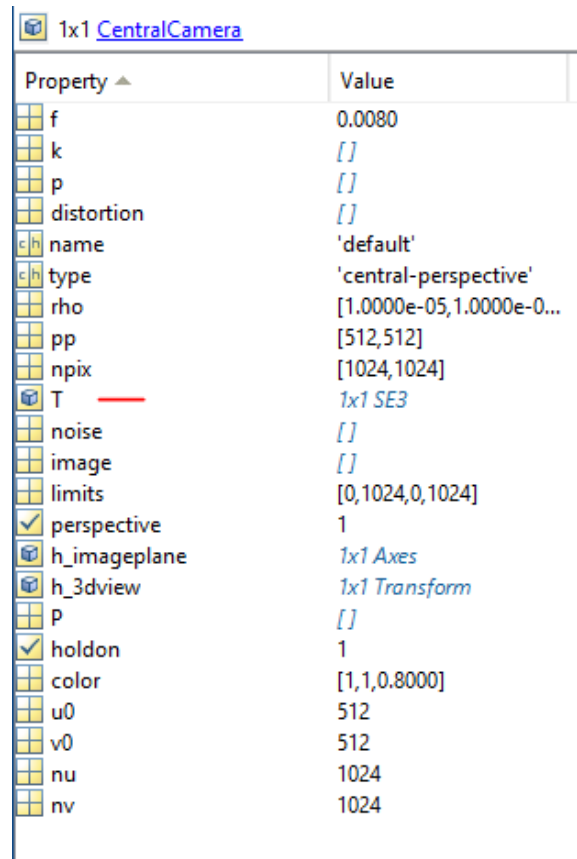
Simulation:

Firstly, root to examples folder> Open IBVS_1.m in matlab.
Then, in command line, do the following:

1. **cam = CentralCamera('default');**

This selects the default camera model for simulation.

Output: principal point not specified, setting it to centre of image plane. (We can also specify the principal point of the camera which computes the relative pose then after)



The image shows a screenshot of the MATLAB CentralCamera object inspector. The title bar indicates it is a 1x1 CentralCamera object. The table below lists the properties and their corresponding values.

Property	Value
f	0.0080
k	[]
p	[]
distortion	[]
name	'default'
type	'central-perspective'
rho	[1.0000e-05, 1.0000e-0...]
pp	[512, 512]
npix	[1024, 1024]
T	1x1 SE3
noise	[]
image	[]
limits	[0, 1024, 0, 1024]
perspective	1
h_imageplane	1x1 Axes
h_3dview	1x1 Transform
P	[]
holdon	1
color	[1, 1, 0.8000]
u0	512
v0	512
nu	1024
nv	1024

Fig.2: Gives the properties and objects defined in CentralCamera

cam.T	
Property ▲	Value
t	[6.8472e-04;6.8472e-04;0.9973]
n	[1.0000;0.0052;1.3067e-15]
o	[-0.0052;1.0000;-4.1763e-16]
a	[-1.3088e-15;4.1086e-16;1]

Fig.3: Gives the properties of cam.T object

2. `ibvs = IBVS_l(cam, 'example');`

This selects the camera cam of type

Output: canned example, line-based IBVS with three lines
draw image plane in existing axes

Axes with properties:

```

        XLim: [0 1024]
        YLim: [0 1024]
        XScale: 'linear'
        YScale: 'linear'
        GridLineStyle: '-'
        Position: [0.13 0.11 0.33466 0.815]
        Units: 'normalized'

```

Fig.4: Displays the axes parameters

3. `ibvs.run()`

This runs ibvs matlab script which *has desired settings*

Output: *completed on error tolerance*

1x1 IBVS_1	
Property ▲	Value
lambda	0.0800
eterm	0.0100
f_star_retinal	[0.5236,-1.5708,2.6180;0.2500,0....
f_star	[0.5236,-1.5708,2.6180;899.4050...
planes	4x3 double
P	[1,-0.5000,-0.5000;0,0.8660,-0.8...
uv_star	[]
Tcam	4x4 double
camera	1x1 CentralCamera
Tf	4x4 double
T0	4x4 double
pf	[]
history	1x57 struct
niter	[]
fps	5
verbose	0
arglist	1x1 cell
axis	[]
movie	[]
anim	[]
type	'line'

Fig.4: Gives the properties and objects defined in ibvs_1

ibvs.camera.h_imageplane	
Property ▲	Value
CameraPosition	[512,512,7.2408e+03]
CameraPositionMode	'auto'
CameraTarget	[512,512,0]
CameraTargetMode	'auto'
CameraUpVector	[0,-1,0]
CameraUpVectorMode	'auto'
CameraViewAngle	8.0894
CameraViewAngleMode	'auto'
View	[0,90]
Projection	'orthographic'
LabelFontSizeMultiplier	1.1000
AmbientLightColor	[1,1,1]
DataAspectRatio	[1,1,1]
DataAspectRatioMode	'manual'
PlotBoxAspectRatio	[512,512,1]
PlotBoxAspectRatioMode	'auto'

Fig.5: Gives the properties and objects defined in *ibvs.camera.h_imageplane*(normal considered)

Simulation Plots: Considering 3-Lines

Plot-1: Initial stage

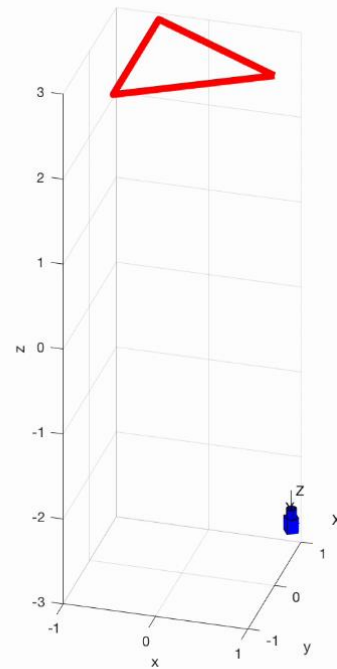
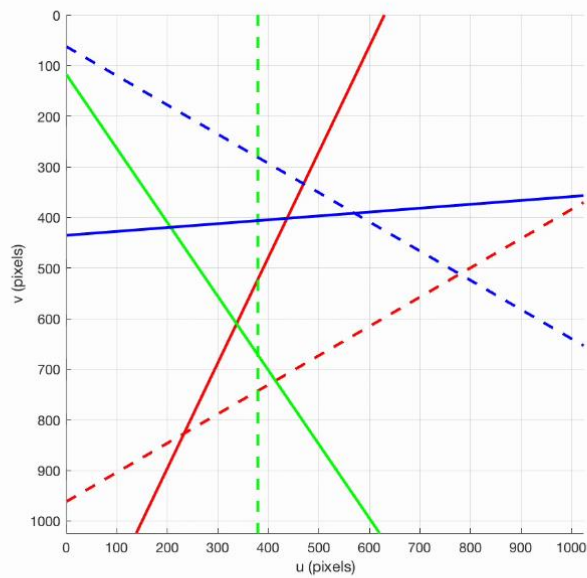


Fig.6: Pose of the lines w.r.t to camera (on left: 6a) and camera pose in (3D on right: 6b). Thick colored lines represents the initial pose and dotted color lines are the desired pose of lines. Red frame is the plane in 3D space.

Plot-2: Intermediate stage

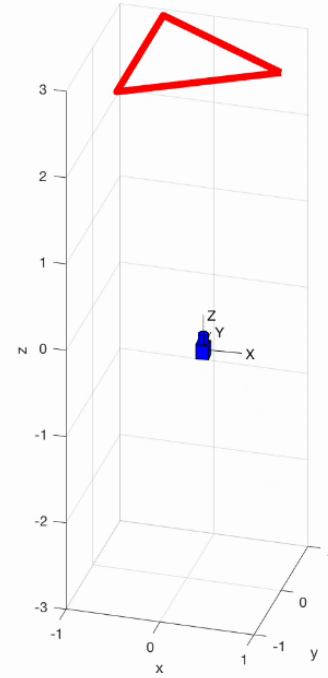
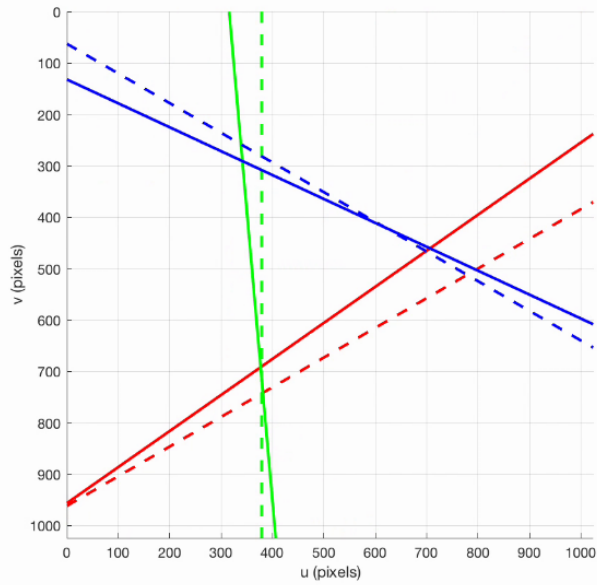


Fig.7: Pose of the lines w.r.t to camera (on left: *7a*) after t -instances and camera pose in (3D on right: *7b*) with little camera motion. Red frame is the plane in 3D space is still fixed.

Since the control scheme imposes a behavior of s **here** rho-theta parameters which is here expressed in the Cartesian space, it allows the camera to follow theoretically an optimal trajectory in that space but generally not in the image space.

Plot-3: Final stage

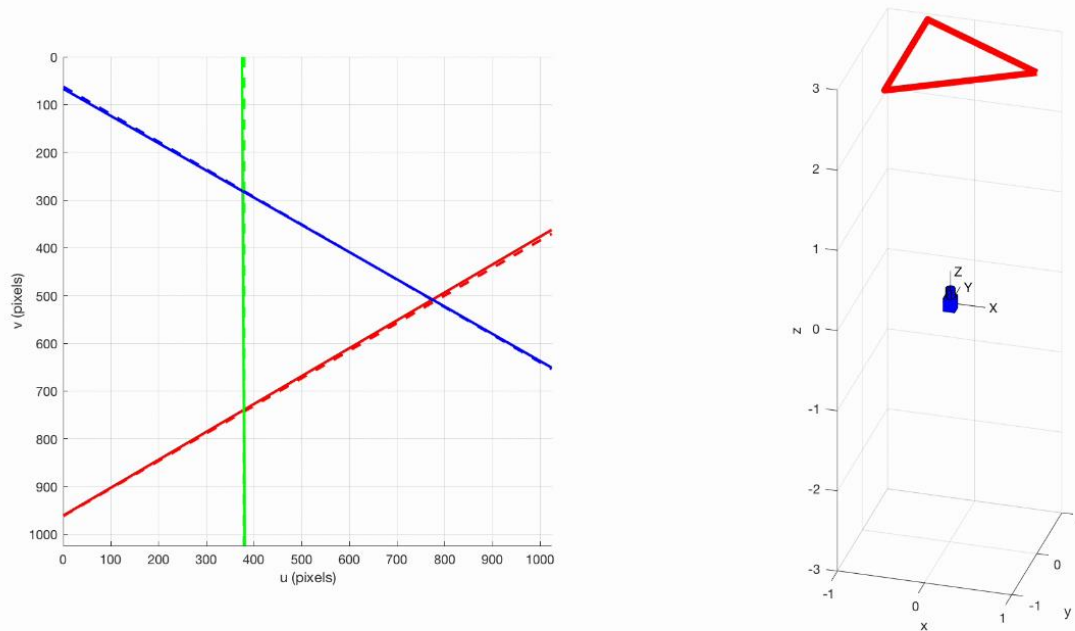


Fig.8: Final Pose of the lines w.r.t desired pose (on left: 8a) after n-instances and camera pose in (3D on right: 8b) is final position in 3D space. Observe the camera movement in 6b and 8b.

Thus lines based Visual Servoing is done using three lines as described above. See the video output (additional reference). This can be extended to four lines format. For flexibility I used the toolbox which then after I explored that I can also perform VS from sphere, this toolbox has builtin tools which I observed while implementing the project.

References

- [1] "Visual Servoing from Lines" - Nicolas Andreff, Bernard Espiau et Radu Horaud
- [2] My Class Notes, Lecture notes and slides of Prof. Omar Tahri
- [3] Github: <https://github.com/petercorke/machinevision-toolbox-matlab>

THE END