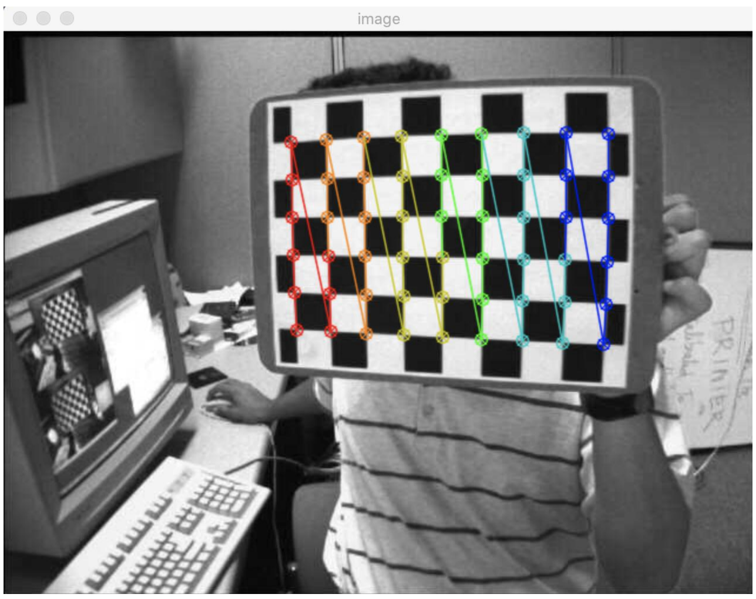
Part 1

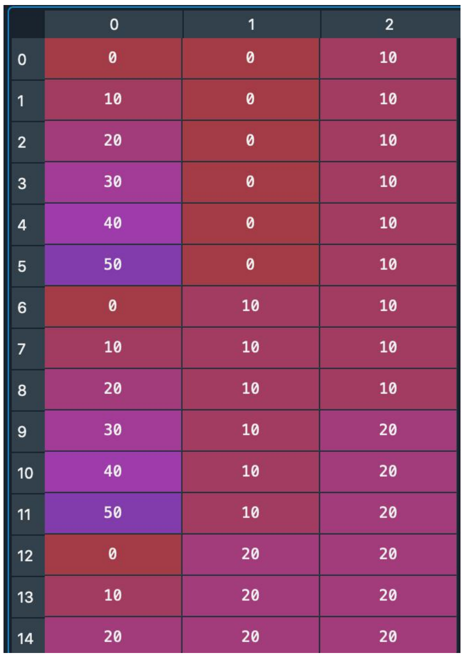
Problem: feature points extraction

Solution:

Given a calibration target, use cvFindChessboardCorners to find chessboard corners. Then find subpixel for the corners as feature points. Draw the subpixels on the chessboard. Save the feature points as image points (2D) and generate object points (3D) as a mesh grid. Manipulate the z-axis of object points such that object points are on different plane for non-planar calibration. Also export corresponding points (3D-2D) to a file.

Result:





	0	1	2
0	0	0	10
1	10	0	10
2	20	0	10
3	30	0	10
4	40	0	10
5	50	0	10
6	0	10	10
7	10	10	10
8	20	10	10
9	30	10	20
10	40	10	20
11	50	10	20
12	0	20	20
13	10	20	20
14	20	20	20

Part 2:

Problem: non-planar calibration

Solution:

Read data and split as object points and image points.

Convert object points to 3DH (add a column of ones).

Construct matrix A and solve for M using SVD.

Compute parameters from M.

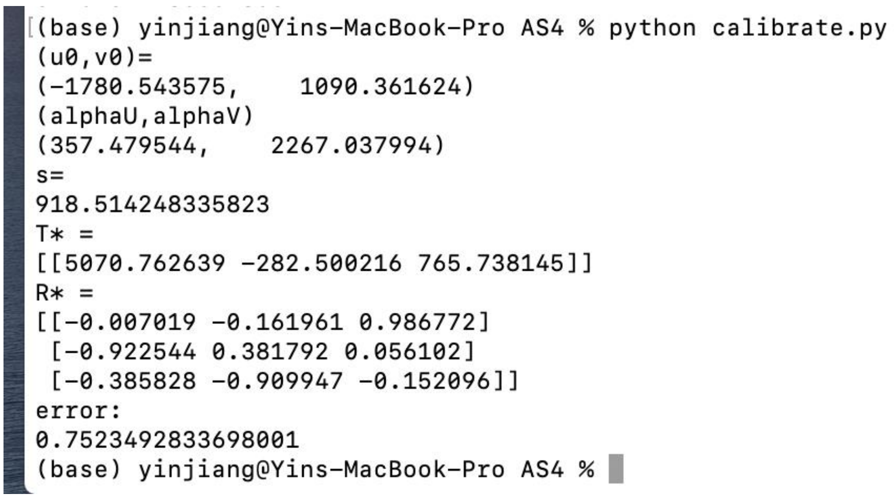
Report parameters.

Project object points to image points using M.

Compare and report error.

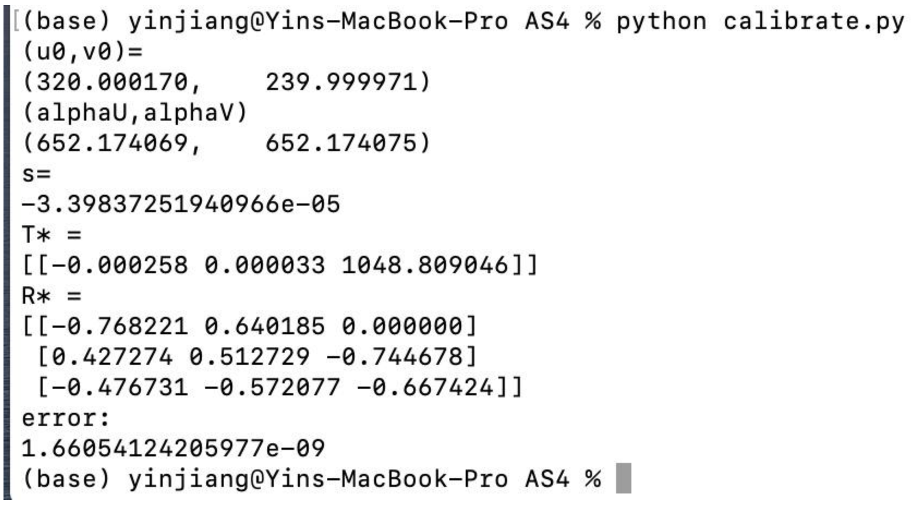
Result:

Using data points from part 1



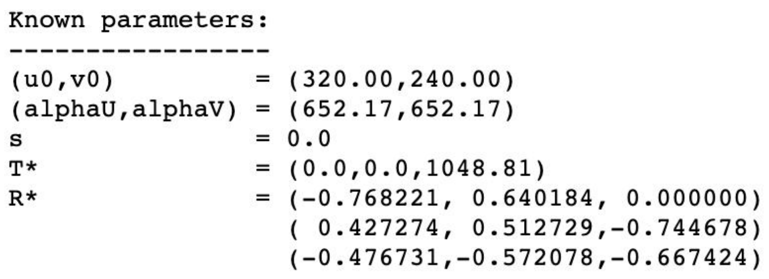
```
(base) yinjiang@Yins-MacBook-Pro AS4 % python calibrate.py
(u0,v0)=
(-1780.543575,    1090.361624)
(alphaU,alphaV)
(357.479544,    2267.037994)
s=
918.514248335823
T* =
[[5070.762639 -282.500216 765.738145]]
R* =
[[-0.007019 -0.161961 0.986772]
 [-0.922544 0.381792 0.056102]
 [-0.385828 -0.909947 -0.152096]]
error:
0.7523492833698001
(base) yinjiang@Yins-MacBook-Pro AS4 %
```

Using test data provided on website



```
(base) yinjiang@Yins-MacBook-Pro AS4 % python calibrate.py
(u0,v0)=
(320.000170,    239.999971)
(alphaU,alphaV)
(652.174069,    652.174075)
s=
-3.39837251940966e-05
T* =
[[-0.000258 0.000033 1048.809046]]
R* =
[[-0.768221 0.640185 0.000000]
 [0.427274 0.512729 -0.744678]
 [-0.476731 -0.572077 -0.667424]]
error:
1.66054124205977e-09
(base) yinjiang@Yins-MacBook-Pro AS4 %
```

Comparing with known parameters, the calibration is correct.



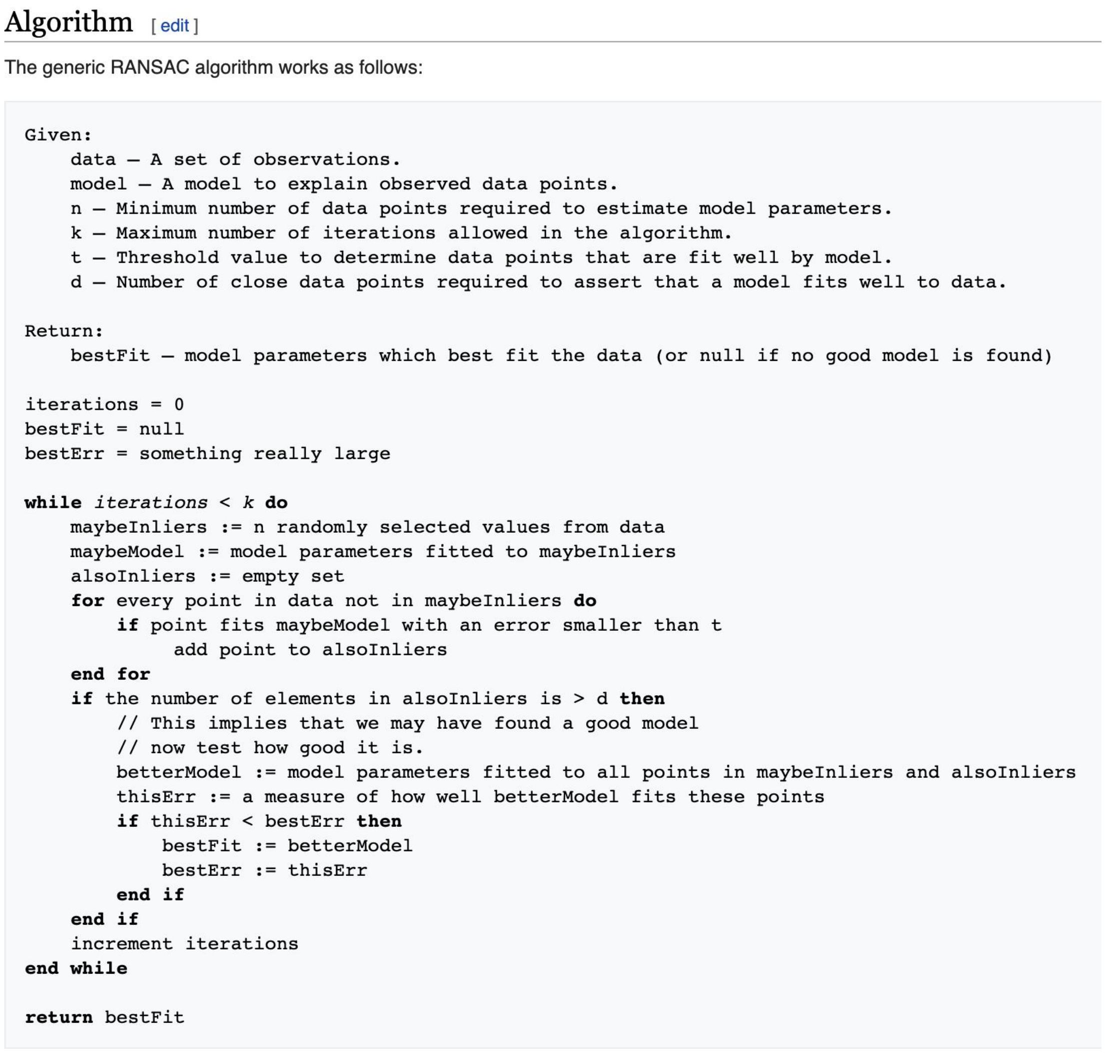
```
Known parameters:
-----------------
(u0,v0)         = (320.00,240.00)
(alphaU,alphaV) = (652.17,652.17)
s               = 0.0
T*              = (0.0,0.0,1048.81)
R*              = (-0.768221, 0.640184, 0.000000)
                  ( 0.427274, 0.512729,-0.744678)
                  (-0.476731,-0.572078,-0.667424)
```

Part 3

Problem: RANSAC implementation

Solution:

RANSAC algorithm from wiki



Algorithm [edit]

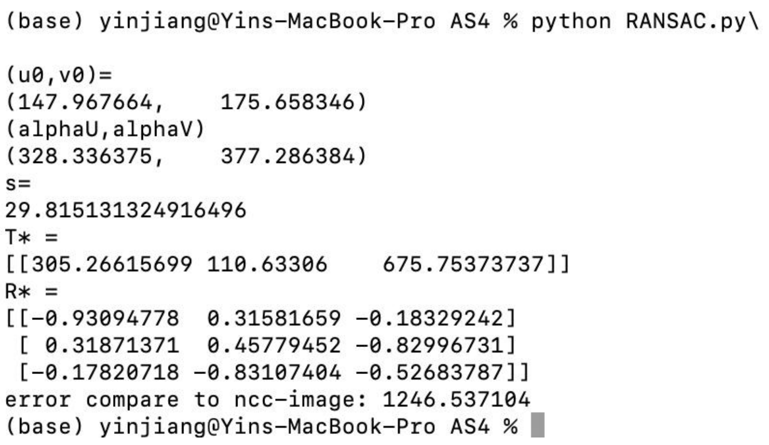
The generic RANSAC algorithm works as follows:

```
Given:
    data – A set of observations.
    model – A model to explain observed data points.
    n – Minimum number of data points required to estimate model parameters.
    k – Maximum number of iterations allowed in the algorithm.
    t – Threshold value to determine data points that are fit well by model.
    d – Number of close data points required to assert that a model fits well to data.

Return:
    bestFit – model parameters which best fit the data (or null if no good model is found)

iterations = 0
bestFit = null
bestErr = something really large

while iterations < k do
    maybeInliers := n randomly selected values from data
    maybeModel := model parameters fitted to maybeInliers
    alsoInliers := empty set
    for every point in data not in maybeInliers do
        if point fits maybeModel with an error smaller than t
             add point to alsoInliers
    end for
    if the number of elements in alsoInliers is > d then
        // This implies that we may have found a good model
        // now test how good it is.
        betterModel := model parameters fitted to all points in maybeInliers and alsoInliers
        thisErr := a measure of how well betterModel fits these points
        if thisErr < bestErr then
            bestFit := betterModel
            bestErr := thisErr
        end if
    end if
    increment iterations
end while

return bestFit
```

Parameters (n, d, kmax) are set in RANSAC.config

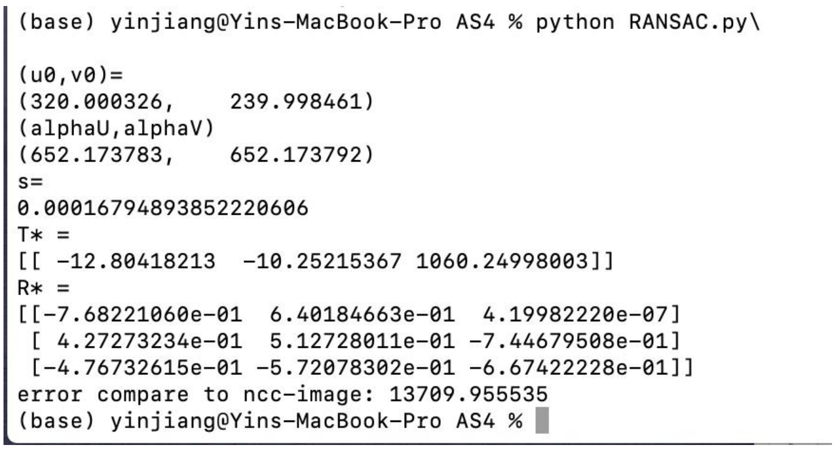
Result:

Testing with noise image 0



```
(base) yinjiang@Yins-MacBook-Pro AS4 % python RANSAC.py\

(u0,v0)=
(147.967664,    175.658346)
(alphaU,alphaV)
(328.336375,    377.286384)
s=
29.815131324916496
T* =
[[305.26615699 110.63306    675.75373737]]
R* =
[[-0.93094778  0.31581659 -0.18329242]
 [ 0.31871371  0.45779452 -0.82996731]
 [-0.17820718 -0.83107404 -0.52683787]]
error compare to ncc-image: 1246.537104
(base) yinjiang@Yins-MacBook-Pro AS4 %
```

Testing with noise image 1



```
(base) yinjiang@Yins-MacBook-Pro AS4 % python RANSAC.py\

(u0,v0)=
(320.000326,    239.998461)
(alphaU,alphaV)
(652.173783,    652.173792)
s=
0.00016794893852220606
T* =
[[ -12.80418213  -10.25215367 1060.24998003]]
R* =
[[-7.68221060e-01  6.40184663e-01  4.19982220e-07]
 [ 4.27273234e-01  5.12728011e-01 -7.44679508e-01]
 [-4.76732615e-01 -5.72078302e-01 -6.67422228e-01]]
error compare to ncc-image: 13709.955535
(base) yinjiang@Yins-MacBook-Pro AS4 %
```

Discussion:

RANSAC works well with noise-1-image, but poor with noise-0-image.

We can compute deviation between noise-image and original image. And we found that moise-1 has a large deviation which means more outliers. Whereas noise-0 has a small deviation which means little outliers can be found by RANSAC. Then RANSAC does not work well with noise-0.