Assignment 4: Camera Calibration and Pose Estimation

This assignment is out of 100 points. Please turn into Moodle, a .zip file containing the following:

1. A short report containing answers to all necessary questions and optional questions of your choice. The report should contain **all images** generated by applying your code to images listed in the questions. Additionally, each question in the report should contain names of functions / scripts that you have written for that particular question, and input and output at the MATLAB prompt (as long as the output is small: if the output is an image, do not print it to the MATLAB prompt!). If there is any hand-written part that you will provide me in person, please indicate this in the report.
2. All images generated by applying your code, written out as .jpg or .png files.
3. All scripts and functions written by you for the assignment. Important: make sure the code is commented with a help block in the beginning, and with comments throughout the code.

Optional: any hand-written work that goes along with the assignment can be turned in to me in class the day the assignment is due, slid underneath my door on the due date of the assignment, or scanned in and submitted with the report. Start your assignment soon! Assignment questions begin on Page 2. Happy coding!

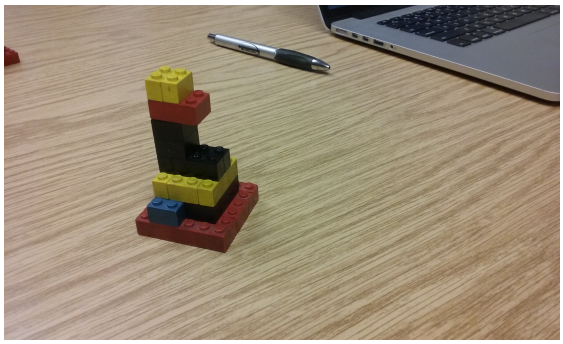
The objective of this assignment is two-fold (both related): 1) to calibrate a camera (i.e., to obtain its matrix **K**, and 2) to estimate the pose of the calibration target, i.e., to obtain **R** and **t**. In the first part of the assignment, you will write code to use an arbitrary object as a calibration target, and simultaneously estimate the pose of the object in the perspective of the camera. In the second part of the assignment, you will use a the Camera Calibration Toolbox and a checkerboard plane to calibrate your camera (however, you need not report the pose of the target).

Camera calibration and pose estimation can be done for any camera that captures color images. Apart from the traditional point-and-shoot camera, cameras on a smart phone, a laptop, and a wearable device are all good candidates for this assignment. If you have trouble getting access to a camera, please contact the instructor early.

All results in Sections 1 and 2 are to be provided on a photograph taken using your camera. Results in Section 3 are to be provided in two other photographs taken using your camera. **In this assignment, required results are provided in boldface**.

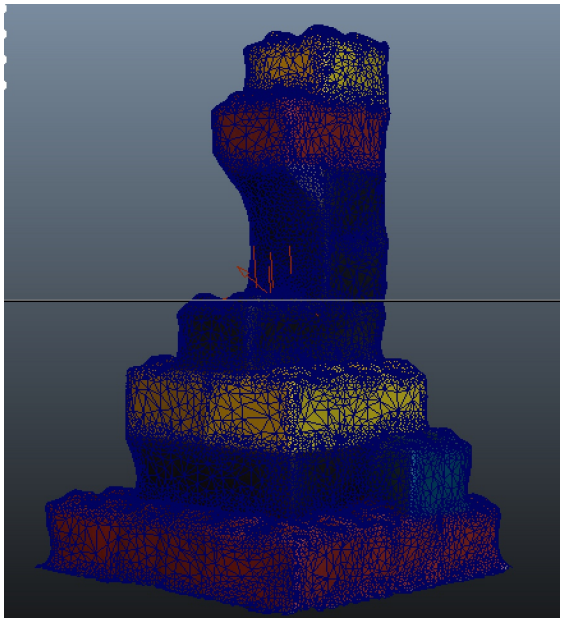
1 Camera Calibration and Pose Estimation with an Arbitrary Object

On this assignment, you will use a LEGO object provided by the instructor. The following image shows the LEGO object you will use on this assignment. Please remember to pick up a LEGO object from the instructor.



The LEGO object comes with a 3D model created using photogrammetry with Autodesk 123D Catch (now replaced by ReCap). The figure on the following page shows the 3D model of the LEGO object used in this assignment.

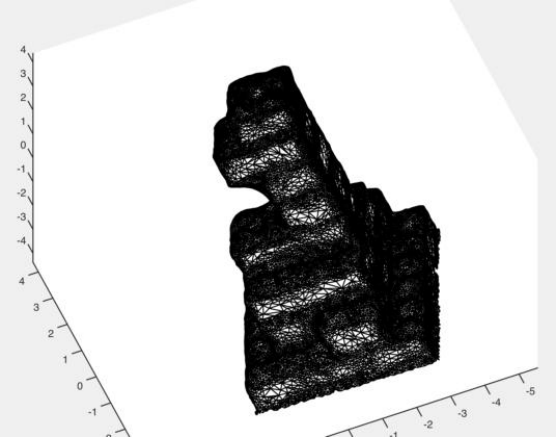
A variety of files for the 3D model are stored in the directory 'dalekosaur' in the .zip file provided for this assignment.



You can view a mesh corresponding to the 3D model in MATLAB. A mesh is a collection of 3D points connected to each other by means of faces. The 3D points and the faces corresponding to the 3D model are stored in the file 'dalekosaur/object.mat', and can be loaded into MATLAB by calling `load dalekosaur/object.mat`. This will load a variable `Xo` for the 3D points of the mesh and a variable `Faces` for the faces joining the 3D points. `Xo` is of size $3 \times N_{\text{mesh}}$ where N_{mesh} is the number of 3D points in the mesh, and 3 represents the number of dimensions, i.e., 3D. `Faces` is of size $N_{\text{faces}} \times 3$, where N_{faces} is the number of faces, and 3 here stands for the fact each face is a triangle, i.e., each face connects three vertices. You can view the 3D model in MATLAB by calling

```
patch('vertices', Xo', 'faces', Faces, 'facecolor', 'w', 'edgecolor', 'k');
axis vis3d;
axis equal;
xlabel('Xo-axis'); ylabel('Yo-axis'); zlabel('Zo-axis');
```

The `patch` function allows you to draw a mesh corresponding to a set of vertices and a set of faces, with a desired color for the faces (white or 'w' in this case), and a desired color for the edges (black or 'k' in this case). You can use the rotation option in the figure window to rotate the 3D model around and see it from multiple viewpoints. The command `axis vis3D` forces the 3D axes to be rigid, and `axis equal` forces the ticks along X, Y, and Z to be all the same length. The following figure shows the output of the `patch` function after moving the mesh around.
In this assignment, you will take a photograph of the LEGO object, and use the 3D model of the LEGO object together with the photograph to calibrate your camera and estimate the



pose of the object. The following sub-sections elucidate the different steps that you will take to obtain your results.

1.1 MARK POINTS

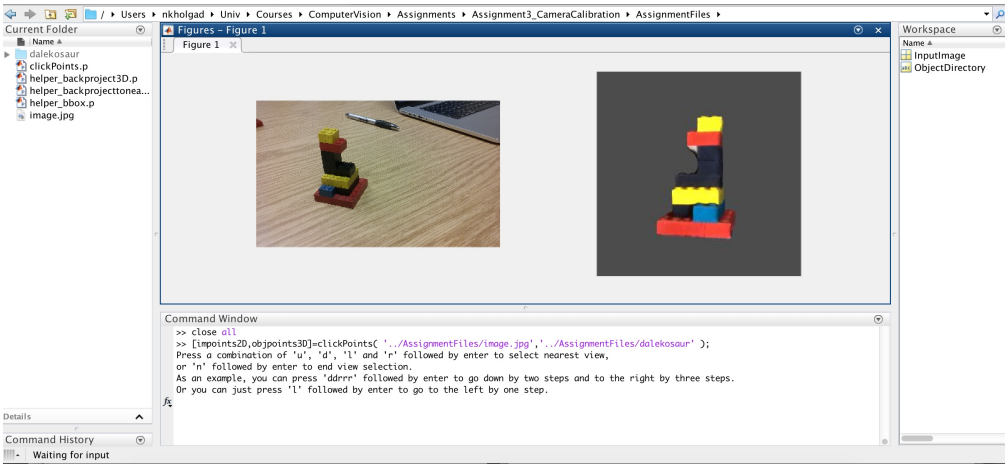
[5 Points] The assignment directory contains a variety of code files with the extension .p. The .p file is a semi-secure encryption of a .m file, and can be called in the same way. You can also do a help on a .p file the same way you would do it on a .m file (performing help on the non-helper functions in the assignment directory is highly recommended). Use the function `clickPoints` stored in 'clickPoints.p' to mark points between the image and the object. Assuming you have read in your image to a variable `InputImage`, you can call `clickPoints` as follows:

```
[impoints, objpoints3D] = clickPoints( InputImage, ObjectDirectory );
```

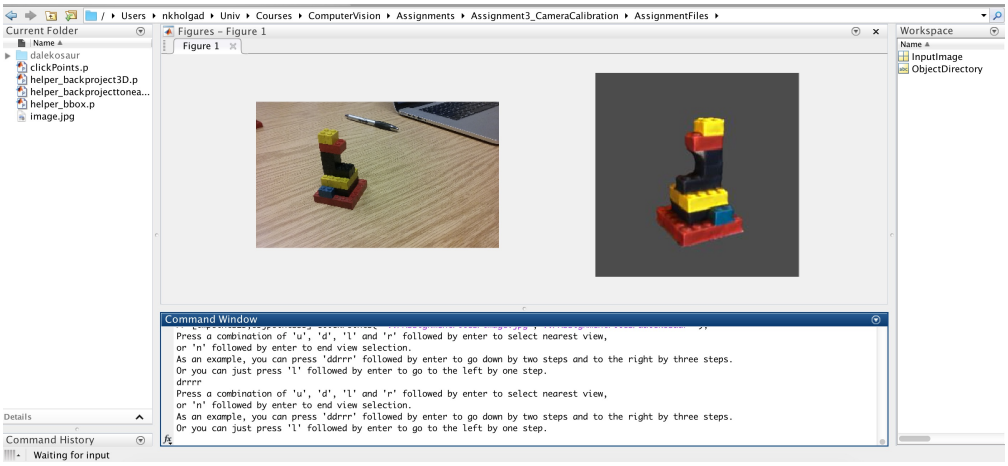
Here `ObjectDirectory` is the full path to the directory containing the object files. In the case of this assignment, `ObjectDirectory` should point to 'dalekosaur'. If you are in the assignment directory, you can set `ObjectDirectory` as:

```
ObjectDirectory = 'dalekosaur';
```

Calling the `clickPoints` function brings up the following window:
At the top, you will have a panel containing the image on the left and the object rendered from a standard viewpoint on the right. A render is a synthetic image created for a particular viewpoint of an object from a 3D model: it almost looks like the real object, but it still looks 'fake'. At the bottom, you have a prompt that allows you to rotate the object on the right by using the 'u', 'd', 'l', and 'r' keys on your keyboard. The key 'u' rotates the object up, 'd' rotates it down, 'l' rotates it to the left, and 'r' rotates it to the right. As you move the object, you will see it rendered from a new viewpoint corresponding to how you have moved the object.

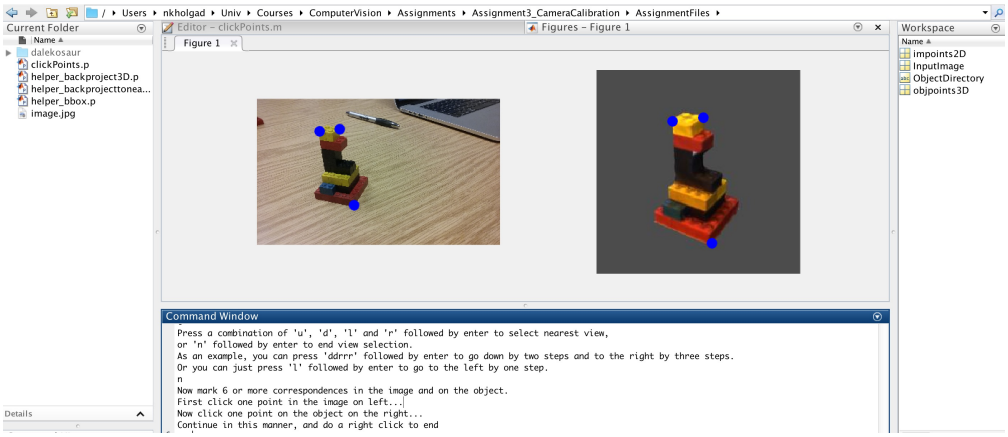


The prompt allows you to provide several rotation steps in one go so as to save time. For instance, if you type ‘drrrr’ and hit Enter, you will rotate the object down by one step, and to the right by four steps. Here is what the panel would look like once you enter ‘drrrr’ for the image shown earlier in the write up:



Continue using combinations of ‘u’, ‘d’, ‘l’, and ‘r’ with Enter to get the object to match the image viewpoint as closely as it can. Then press ‘n’ to go to the next step.

In the next step, the function prompts you to click points in the image and on the object. For the code to work correctly, first click a point in the image, then click a point on the object. Continue clicking image-object pairs till you have as many as you would like (>=6), and then right click to quit. The following figure shows three pairs.
The function will provide the image points in `impoints2D` and object points in `objpoints3D`.



You can plot the image points by calling

```
figure;
imshow(I); hold on;
plot( impoints2D(:,1), impoints2D(:,2), 'b.');,
```

and the object points by calling

```
figure;
patch('vertices', Xo', 'faces', Faces, 'facecolor', 'w', 'edgecolor', 'k');
axis vis3d;
axis equal;
plot3( objpoints3D(:,1), objpoints3D(:,2), objpoints3D(:,3), 'b.'  );
```

Provide results of the points plotted on the image and on the mesh of the object in your report.

1.2 Estimating the Camera Projection Matrix **M**

[20 Points] Use the method discussed in class to estimate the camera projection matrix **M** by creating a design matrix **P** based on the object points, and solving a least-squares system of equations **Pq** = **r**. **Provide a function** `estimateCameraProjectionMatrix` **that takes in** `impoints2D` **and** `objpoints3D` **and returns** `M`**, with the following header:**

```
M=estimateCameraProjectionMatrix( impoints2D,objpoints3D );
```

Also provide the matrix M that you obtain as a result. Do not use RANSAC, as you will have too few points for a reliable RANSAC run.

1.3 GETTING THE INTRINSIC PARAMETERS MATRIX **K**, AND THE POSE **R** AND **t** FROM **M**

[20 Points] Provide code that computes the matrix of intrinsic parameters K, and the extrinsic parameters, i.e., the rotation R and translation t of the object from the camera projection matrix M. Use the method discussed in class of analyzing $\mathbf{K}\mathbf{K}^T = \lambda^2\mathbf{C}$. Provide values of **K**, **R**, and **t**.

1.4 VERIFYING YOUR RESULT

[15 Points] Given the matrices **K**, **R**, and **t**, use equations for 3D transforms together with equations for camera projection to re-project the clicked points, i.e., the points in `objpoints3D` to 2D to obtain `imgpoints2D_estim`. **Use the** `plot` **function in matlab to plot** `imgpoints2D` **as blue dots and** `imgpoints2D_estim` **as red circles onto your image. Include the visual result of the plot in your report.** As we saw in class, **compute the sum-squared distance between the actual image points and the estimated points, and include the value of the sum-squared distance in your report.** The accuracy of your result will be determined by how small this value is.

Additionally, visualize the projection of the transformed mesh on the image. Use equations for 3D transforms together with equations for camera projection to obtain a 2D point for every 3D point in the mesh, i.e., in `Xo`. Assuming that the 2D points obtained as a result are stored to the matrix `x`, where `x` is of size $2xN_{\text{mesh}}$, then you can display the mesh superposed onto the image using the following syntax:

```
figure;
imshow(I); hold on; % 'hold on' holds the image to draw more content
patch('vertices', x', 'faces', Faces, 'facecolor', 'n', 'edgecolor', 'b');
```

The command to display the 2D projection of the mesh is similar to the patch command for displaying the 3D mesh. The only difference is that we are displaying the edges in blue (i.e., 'b') and we are selecting no color for the faces (i.e., 'n'), so that the faces end up being transparent and you can see through the mesh. **Include a snapshot of the mesh superposed onto the image in your report.**

2 CAMERA CALIBRATION USING A PLANAR CHECKERBOARD

[20 Points] In this section, you will use the Camera Calibration Toolbox in MATLAB.
`https://www.mathworks.com/help/vision/ug/single-camera-calibrator-app.html`
Print the checkerboard provided in this assignment onto a sheet of paper, and attach it to a static object such as a wall or a door. Then take images of the printed checkerboard from a variety of different viewpoints and distances.

This method only provides *intrinsic parameters*. At the end of the calibration, you will have the following variables in your workspace (apart from several others which you need not

worry about):

`fc`: the focal length of the camera,
`cc`: the location of the camera center (also called the principal point), and
`alpha_c`: the skew coefficient, i.e., the value of $\mathbf{K}(2,2)$.

Put these values together into a matrix $K_{checker}$. How similar or different are K using the LEGO object, and $K_{checker}$ using the planar checkerboard? What may be the reason for the differences if any?

3 MOVING OBJECTS AROUND IN AN IMAGE

[20 Points] Calibrating a camera allows you to perform fancy object insertions such as the in the following paper:

`http://kevinkarsch.com/publications/sa11-lowres.pdf`

The paper provides a method to perform photorealistic insertions of 3D models of objects into images.

In this assignment, we will use MATLAB to insert a mesh corresponding to the 3D model of the LEGO object into various locations in an image. Here you will use two new images taken by your camera (not the same as the one used in the camera calibration exercise in Section 1). **Provide the following results**

1. **Three results per image of inserting the 3D model of the LEGO object using K.**
2. **The same three results per image of inserting the 3D model using K_{center}.**

To insert a mesh into the new image, transform the 3D points of the mesh `Xo` to a new mesh `X_transformed` using 3D rotations and translations of your choice.

As discussed in class, you have the choice of performing your rotations and translations in object coordinates or in camera coordinates. The only difference is that to perform transforms in camera coordinates, you first transform `Xo` by **R** and **t** estimated in Section 1, and then perform your rotations and translations, and to perform transforms in object coordinates, you first perform your rotations and translations on `Xo` and then transform by **R** and **t**. The two methods are not the same, i.e., using the same rotation and translation values provides different results in either case due to the order in which operations are performed. (Hint: it is easier to perform rotations in the object coordinate system and translations in the camera coordinate system.) **Explain the operations you perform in your report.**

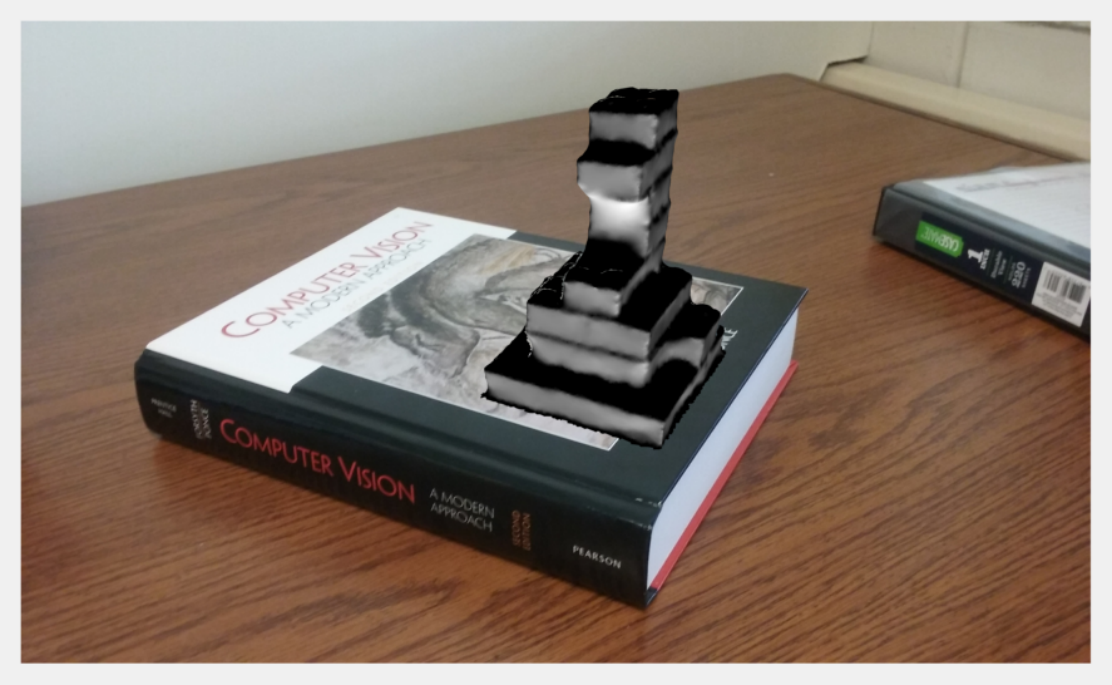
Use camera projection to project the 3D transformed mesh `X_transformed` to the 2D points `x_projected`, and use the `patch` function to display your `x_projected` on your images of choice. As an example, the following image shows the object rotated to the face the left, and

placed so as to appear to be in contact with the surface of the book:

One issue is while you can sort of see the mesh, it is so dense, that it is hard to visually see any detail. Humans see due to light bouncing off objects. In the set of functions provided, you will see a function called 'displayLit' that allows you to provide the `x_projected`, `X_transformed`, `Faces`, and a fake light to create a synthetic version of the object lit using the fake light. Call the function as follows:

```
pointsInFront=isinfront(X_transformed,Faces); % TAKES A LONG TIME TO RUN
imshow(newimage); hold on;
displayLit(x_projected,X_transformed,Faces,lightdirectionvector,pointsInFront
```

The first line provides `pointsInFront` which is a vector of size $1 \times N_{\text{mesh}}$ with true for whichever points in `X_transformed` are seen by the camera. So in the image above, the points at the back of the LEGO object will have false in `pointsInFront`. THE FUNCTION `isinfront` CAN TAKE NEARLY 5-10 MINUTES TO RUN. The next line brings up the image, and holds it so that you can display additional content. The final line takes in the projected mesh, transformed mesh, faces, a light direction vector, and `pointsInFront` and creates a lit version of the object. The light direction vector tells you where light is coming from. For instance, if light is coming from the bottom upwards (i.e., in +Y direction, as if one had held a torch light under the dalekosaur's head), you would specify `lightDirectionVector=[0,-1,0]`, and the result looks as follows:



A good way to specify the light direction is to figure out the general direction in which you see shadows in your image. Typically lights are on the top, i.e., light shines from top down. Also, you may want light to shine directly into the picture, i.e., along the positive Z direction. Play around with different values to get a result that looks good. **Provide results with the object lit using a fake light in your report.**
While performing these re-orientations of the object, it is best to rotate or translate the object through small angles and distances till you get a feel for where the object is moving.

In creating the results, consider how you believe objects should be oriented in the real world, or how they should interact with objects such as tables, floors, chairs, and people's hands and fingers. **Is there a difference in perceptual quality between the images created using K and $K_{checker}$. What do you think may influence your perception of these results?**