Exam-Style Problems

1. This problem takes you through the computation of the set of all least-squares solutions to the following linear system:

\[
\begin{align*}
3x + 4y &= 2 \\
3x + 4y &= 3
\end{align*}
\]

and the solutions to a related optimization problem. All the answers to the questions in this problem are numerical. They refer only to the data given in the problem, and no more general answers are required. You may leave your answers in the form of fractions, with expressions like the following:

\[
\frac{\sqrt{3}}{2} \begin{bmatrix} 2 \\ -5 \end{bmatrix},
\]

but please simplify. If you cannot answer a question, write out a symbolic placeholder, and refer to that in later parts if necessary. For instance, if you don’t know what \( b \) is in the first question, write

\[
b = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}.
\]

(a) What are \( A \) and \( b \) if we write the system in this problem in the following form?

\[
Ax = b
\]

(b) What is the rank of \( A \)?

(c) Give a unit column vector \( r \) that spans the row space of \( A \).

(d) Give a unit column vector \( n \) that spans the null space of \( A \).

(e) Write the matrix \( V \) in the SVD \( A = U \Sigma V^T \) of \( A \).

(f) Compute the matrices \( U \) and \( \Sigma \) in the SVD of \( A \). [Hint: compute \( U \Sigma \) first.]

(g) Compute the pseudo-inverse \( A^\dagger \) of \( A \).

(h) Find the minimum-norm solution \( x^* \) of the system \( Ax = b \).

(i) Give an expression for the set \( S \) of all least-squares solutions of the system \( Ax = b \).

(j) Find all the solutions to

\[
\hat{x} = \arg \min_{\|x\|=1} \|Ax\|.
\]
2. The Lucas-Kanade algorithm initialized with displacement $d_0 = 0$ is used to track point features from image $I(x)$ to image $J(x)$. Both images are gray (no color), that is, they are functions from $\mathbb{R}^2$ to $\mathbb{R}$. The residual

$$
\epsilon(x_I, d) = \sum_x [J(x + d) - I(x)]^2 w(x - x_I)
$$

at image point $x_I$ in $I(x)$ with displacement $d$ is defined over a tiny, $2 \times 2$ tracking window with uniform weight:

$$
w(x) = \begin{cases} 
1 & \text{if } x = (0, 0) \text{ or } (0, 1) \text{ or } (1, 0) \text{ or } (1, 1) \\
0 & \text{otherwise}
\end{cases}
$$

A threshold $\sigma_0 = 0.1$ on the smaller singular value of the matrix

$$
A_I(x_I) = \sum_x \nabla I(x) |\nabla I(x)|^T w(x - x_I)
$$

is used as the only criterion for choosing good features to track.

The gradient

$$
\nabla I(x) = \begin{bmatrix}
\frac{\partial I}{\partial x_1} \\
\frac{\partial I}{\partial x_2}
\end{bmatrix}
$$

of image $I(x)$ at a particular image location has components

$$
G_1 = \frac{\partial I}{\partial x_1} = \begin{bmatrix} 1 & -1 \\ 2 & 1 \end{bmatrix} \quad \text{and} \quad G_2 = \frac{\partial I}{\partial x_2} = \begin{bmatrix} 2 & -2 \\ 4 & 2 \end{bmatrix}.
$$

for the four pixels $x$ within the tracking window. Note that $G_1$ and $G_2$ are miniature images. For instance, the gradient in the upper-left pixel of the tracking window is

$$
\begin{bmatrix} G_1(1, 1) \\ G_2(1, 1) \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}.
$$

Is the feature at this image location a good feature to track? Why or why not? [Hint: Do not over-compute.]

3. The figure below shows a top view of two three-dimensional reference systems. The system with axes $wX_1, wX_2, wX_3$ is the world reference system, and its third axis $wX_3$ points out of the page. The system with axes $cX_1, cX_2, cX_3$ is the camera reference system. Its origin has coordinates $(3, 1, 0)$ in the world reference system, and its third axis $cX_3$ points out of the page as well, and is parallel to the $wX_3$ axis.

A point with coordinates $wX$ in the world reference system has coordinates in the camera reference system given by the vector

$$
cX = R(wX - t).
$$

where $R$ is a $3 \times 3$ matrix and $t$ is a $3 \times 1$ vector.
In the questions below, a “numerical expression” can be any expression that evaluates to a number or an array of numbers. So it is OK to use fractions, square roots, and so forth in your expressions.

(a) Write numerical expressions for \( R \) and \( t \).

(b) Write a symbolic (no numbers) expression for \( \mathbf{w}^T \mathbf{X} \) as a function of \( \mathbf{X} \) involving \( R \) and \( t \).

(c) Let \( \mathbf{i}_0 \) be the unit vector that points along the \( X_1 \) axis of the world reference system, expressed in camera coordinates. Write a numerical expression for \( \mathbf{i}_0 \).

### Tracking

4. The MATLAB function `lucas_kanade` provided with this assignment is a simple implementation of the Lucas-Kanade tracker. It takes two gray-level images \( \mathbf{I} \) and \( \mathbf{J} \), a \( 2 \times n \) array \( \mathbf{xI} \) of image positions in \( \mathbf{I} \), a positive scalar \( \sigma \), and an optional \( 2 \times n \) array \( \mathbf{displ} \). The first row in \( \mathbf{xI} \) and \( \mathbf{displ} \) denotes row coordinates, the second row denotes column coordinates. The function makes a square tracking window of size \( 2h + 1 \) where \( h = \lceil 2.5 \sigma \rceil \) and whose pixels are weighted by a Gaussian function with spread parameter \( \sigma \). The optional argument \( \mathbf{displ} \) contains initial estimates of the displacement. If this argument is not provided, the function sets \( \mathbf{displ} \) to all zeros. The output from the function is also called \( \mathbf{displ} \), and contains displacements computed by the Lucas-Kanade algorithm. The coordinates \( \mathbf{xJ} \) of the points corresponding to those in \( \mathbf{xI} \) can be computed as

\[
\mathbf{xJ} = \mathbf{xI} + \mathbf{displ}
\]

Tracking may fail for some features. In that case, `lucas_kanade` prints out a warning and sets the corresponding column of \( \mathbf{displ} \) to NaNs.

Functions `gauss` and `grad` are provided as well. The function `gauss` computes a one-dimensional truncated gaussian with the given parameters and the function `grad` computes the gradient of an image. It returns a cell array \( \mathbf{g} \) such that \( \mathbf{g}{1} \) is the derivative in the row direction and \( \mathbf{g}{2} \) is the derivative in the column direction.

The file `data.mat` provided with this assignment contains two images \( \mathbf{I} \) and \( \mathbf{J} \). One is a stock photograph. The other was obtained from the first artificially, by warping it differently in different parts of the image. This affords a luxury that is usually absent in actual computer vision work, namely, the true motion at every pixel in the image. This true motion field is also provided in `data.mat` as an array \( \mathbf{D} \) of dimensions \( \lbrack \text{rows}, \ \text{cols}, \ 2 \rbrack \) where \( \text{rows}, \text{cols} \) is the size of both \( \mathbf{I} \) and \( \mathbf{J} \). Pixel \( \mathbf{xI}(\mathbf{a} \ 2 \times 1 \ \text{vector}) \) in \( \mathbf{I} \) corresponds to the pixel in \( \mathbf{J} \) with coordinates

\[
\mathbf{xJ} = \mathbf{xI} + \text{squeeze}(\mathbf{D}(\mathbf{xI}(1), \ \mathbf{xI}(2), :));
\]
(a) Before we track, we need to select image points for tracking. In this context, we call a point and a surrounding window to be tracked a feature. Use the function `grad` and the MATLAB built-in function `conv2` with the ‘same’ option to write a function with header

```
function lambdaMin = smallEigenvalue(I, sigma)
```

that computes an image with the smallest eigenvalue \( \lambda_{\text{min}}(x) \) of the Gramian \( A(x) \) at every pixel in image \( I \). Recall that the Gramian is defined as follows:

\[
A(x) = \sum_{z \in \mathbb{Z}^2} w(z) \nabla I(z + x)[\nabla I(z + x)]^T
\]

where \( w \) is a Gaussian window. The second argument \( \sigma \) to `smallEigenvalue` is the parameter \( \sigma \) used in the definition of \( w(z) \).

**[Eigenvalues and singular values]** For a symmetric, positive semi-definite matrix like \( A(x) \), eigenvalues and singular value are the same. We talk about eigenvalues here because you know a formula for computing eigenvalues for small matrices. More on this below.

If you are surprised that `conv2` shows up here, note that the expression for \( A(x) \) above is a convolution. Specifically, the product

\[
P(x) = \nabla I(x)[\nabla I(x)]^T = \begin{bmatrix} a(x) & d(x) \\ d(x) & b(x) \end{bmatrix}
\]

is a \( 2 \times 2 \) symmetric matrix. So your code will first call `grad` to compute the two components of \( \nabla I \) everywhere in the image, then use matrix operations to compute the three images \( a(x), b(x), d(x) \) in \( P(x) \). The three distinct entries of

\[
A(x) = \begin{bmatrix} p(x) & r(x) \\ r(x) & q(x) \end{bmatrix}
\]

are the convolutions of the three distinct entries of \( P(x) \) with the Gaussian kernel \( w \). **Make sure you use the separability of the Gaussian to make these convolutions efficient.**

Finally, you can compute the image \( \lambda_{\text{min}} \) by recalling that the smaller eigenvalue of \( A(x) \) is

\[
\lambda_{\text{min}}(x) = p(x) + q(x) - \sqrt{(p(x) - q(x))^2 + 4r^2(x)}.
\]

Use matrix operations to do this computation at once over the entire image, without explicit for loops.

**What To Hand In:** Show your code for `smallEigenvalue` and the picture resulting from the call

```
showLambda(lambdaMin)
```

where `lambdaMin` is the image `smallEigenvalue` returns when called with the provided image \( I \) and with \( \sigma = 2 \).

(b) Use your function `smallEigenvalue` to write a function with header

```
function [pos, lambda] = goodFeatures(I, sigma, lambdaThresh, nMax, minDist, margin)
```

that computes a \( 2 \times n \) matrix `pos` of good features to track and a \( 1 \times n \) vector `lambda` of the corresponding values from `lambdaMin`. The number \( n \) of features cannot be greater than \( n_{\text{Max}} \), and their `lambda` values cannot be smaller than the threshold `lambdaThresh`. In addition, no two feature points can be closer to each other than \( \text{minDist} \) pixels (in Euclidean distance), and no feature point can be closer than \( \text{margin} \) pixels to the image boundaries. A method for computing good features is described in the class notes.

**What To Hand In:** Show your code and a figure with image \( I \) and the features found by `goodFeatures.m` superimposed on the image. Show a green dot for each feature (use `plot` with option ‘MarkerSize’, 16). Also report the number \( n \) of features found. Use the following parameters:

```
lambdaThresh = 15;
nMax = Inf;
minDist = 20;
margin = 80;
```

and \( \sigma \) as before.
The function `lucas_kanade` finds a local minimum of the Sum of Squared Differences (SSD) between two windows,

\[ e(d) = \epsilon(x_I, d) = \sum_{z \in \mathbb{Z}^2} w(z)[I(z + x_I) - J(z + x_I + d)]^2. \]

However, `lucas_kanade` never computes the SSD itself, just its derivatives.

Write a function with header

```matlab
function [e, dSpan] = ssdGraph(I, J, xI, sigma, h)
```

that computes an array \( e \) of \((2h + 1) \times (2h + 1)\) values of \( e(d) \) for \( d \) that takes integer values on the grid defined by

\[-h \leq d_1 \leq h \text{ and } -h \leq d_2 \leq h.\]

The parameter \( xI \) is a \( 2 \times 1 \) vector, and \( I, J, \sigma \) are as before. The output \( dSpan \) is the vector \(-h:h\).

**What To Hand In:** Show your code for `ssdGraph` and a mesh plot of \( e \) obtained with the following calls:

```matlab
mesh(dSpan, dSpan, e)
axis ij
xlabel('column displacement')
ylabel('row displacement')
```

The SSD array \( e \) should be computed with \( xI \) set to the first position (best \( \lambda \) value) reported by your function `goodFeatures`. Use \( h = 20 \) and the usual value for \( \sigma \). Before printing the figure, please rotate the mesh to give a good view of the plot, if needed. Also report the value of \( xI \) you used.

**(d)** If you run `lucas_kanade` on the array \( xI \) of features you found with `goodFeatures`, you will get many failure warnings. The main reason for this, although not the only one, is that the true displacements between \( I \) and \( J \) are larger than the algorithm can handle.

Address this problem by writing a function with header

```matlab
function d = scaleTrack(I, J, xI, sigma)
```

that runs `lucas_kanade` on a Gaussian pyramid, as explained in the class notes. Feel free to use the function `gaussianPyramid` from the supplementary materials for the September 7 lecture on the class syllabus page. Only use 4 pyramid levels (it’s OK to let `gaussianPyramid` compute more levels than necessary) and a sampling factor of 1/2.

**What To Hand In:** Show your code for `scaleTrack`, and three figures. The first one is similar to the one you made for part (b) of this problem. However, you should show green dots for the points that your function can track and red dots for the ones it cannot. The second figure should be similar to the first, but with the results of tracking (for the green dots only, of course) superimposed on image \( J \). The third figure should be a scatterplot of the displacements in \( d \), obtained with the call

```matlab
scatterplot(d)
```

where the function `scatterplot` is provided.

**Note:** A small number of features may still be lost, for reasons that have nothing to do with motion size. In particular, the interaction of Newton-Raphson and sub-pixel interpolation can cause the method to oscillate with very small steps once close to convergence, for reasons that would take very long to explain. In this case, the iteration limit may be exceeded. It’s OK if this happens for a handful of features.

**(e)** Use `ssdGraph` to make a plot similar to the one you made earlier, but for one of the features that were *not* tracked (your choice).

**What To Hand In:** Show the plot and report the coordinates of the feature.