**PSYCH-GA.2211/NEURL-GA.2201 – Fall 2025**
**Mathematical Tools for Neural and Cognitive Science**

# Homework 2

Due: 10 Oct 2025
(late homeworks penalized 10% per day)

See the course web site for submission details. Reminder: rather than using the functions `pinv()` and `norm()`, use the linear algebra tools we learned in class. Please: don't wait until the day before the due date... start *now*!

1. **Trichromacy.** Load the file `colMatch.mat` in your MATLAB environment (or use `scipy.io.loadmat` for Python). This file contains matrices and vectors related to the color matching experiment presented in class. The variable `P` is an $N \times 3$ matrix containing wavelength spectra for three "primary" lights, to be used in a color-matching experiment. For these problems $N = 31$, corresponding to samples of the visible wavelength spectrum from 400 nm to 700 nm in increments of 10 nm.

   The function `humanColorMatcher.p` simulates a normal human observer in a color matching experiment. For Python, download the file and use `from trichromacy import human_color_matcher`. The input variable `light` should contain the wavelength spectrum of a test light (a 31-dimensional column vector). The input variable `primaries` should contain the wavelength spectra of a set of primary lights (typically, a $31 \times 3$ matrix, as for matrix `P` described above). The function returns a 3-vector containing the observer's "knob settings" - the intensities of each of the primaries that, when mixed together, appear identical to the test light. The function can also be called with more than one test light (by passing a matrix whose columns contain 31-dimensional test lights), in which case it returns a matrix whose columns are the knob settings corresponding to each test light.

   (a) Create a test light with an arbitrary wavelength spectrum, by generating a random column vector with 31 positive components (use `rand` in MATLAB or `np.random.rand` in Python). Use `humanColorMatcher` to "run an experiment", asking the "human" to set the intensities of the three primaries in `P` to match the appearance of the test light. Compute the 31-dimensional wavelength spectrum of this combination of primaries, plot it together with the original light spectrum, and explain why the two spectra are so different, even though they appear the same to the human.

   (b) Now characterize the human observer as a linear system that maps 31-dimensional lights to 3-dimensional knob settings. Specifically, run a set of experiments to estimate the contents of a $3 \times 31$ color-matching matrix `M` that can predict the human responses. Verify on a few random test lights that this matrix exactly predicts the responses of the function `humanColorMatcher`.

   (c) Your colleague down the hall, Dr. Evoprimary, proudly tells you about a new color-matching configuration using primaries derived from pigments that were most prevalent in the environment of our evolutionary ancestors. You respond that this is a beautiful concept, but seems unlikely to offer new insights into human trichromacy, since you can precisely predict the color matches that will be obtained with these new primaries. A

wager is made (winner gets a box of Rafetto's ravioli). Dr. E allows you to measure the wavelength spectra of the new primaries (stored in variable `eP`). Derive (write math, and explain logic) an expression for the color-matching matrix (3x31, maps a light to 3 knob settings) that predicts matches that would be obtained in Dr E's lab. Compute this matrix, making use of `humanColorMatcher.p` with your own primaries, `P`. Check, for a random test light, that the predicted mixture of `eP` primaries matches (produces the same knob settings when tested with *your* primaries, `P`). If it does not, explain why.

(d) The variable `Cones` contains (in the rows) approximate spectral sensitivities of the three color photoreceptors (cones) in the human eye: `Cones(1,:)` is for the L (long-wavelength, or *red*) cones, `Cones(2,:)` the M (green) cones, and `Cones(3,:)` the S (blue) cones (for Python users, the indexing starts from `0`). Applying the matrix `Cones` to any light $\vec{l}$ yields a 3-vector containing the average number of photons absorbed by that cone (try `plot(Cones')` to visualize them!). Verify that the cones provide a physiological explanation for the matching experiment, in that the cone absorptions are equal for any pair of lights that are perceptually matched. First, do this informally, by checking that randomly generated lights and their corresponding 3-primary matching lights produce equal cone absorptions. Then, provide an extended comment with a more mathematical demonstration (an informal proof) using concepts from linear algebra. [Hints for two possible approaches: (1) write math/code that computes cone responses for any test light and then computes the weighted combination of primaries that would produce the same cone responses - show that this is equivalent to the color-matching matrix; (2) convince yourself (and explain why) it is sufficient to show that the color matching matrix `M` and `Cones` have the same nullspace. Then use SVD to demonstrate that this is true!]

2. **2D polynomial regression.** Load the file `regress2.mat` into your MATLAB environment. The matrix $D$ contains 3 columns of data, which we'll refer to as $X$, $Y$, and $Z$ respectively. The corresponding elements of these vectors, $(X_k, Y_k, Z_k)$, represent 3D data points. X and Y are uniformly distributed on a square grid.

   (a) plot $Z$ as a function of $X$ and $Y$ (using `surf` in `MatLab`, or `plot_surface` in python). To use the mouse to rotate the 3D space and view the data from different angles, execute `rotate3d on` in `MatLab`, or placing `%matplotlib widget` at the top of your Juypiter notebook. Note: you'll need to reshape the three column vectors into square matrices.

   (b) Fit the $Z$ values with polynomials in $X$ and $Y$, up to order 3: $p_0(X,Y) = \beta_0$, $p_1(X,Y) = \beta_0 + \beta_1 X + \beta_2 Y$, $p_2(X,Y) = \beta_0 + \beta_1 X + \beta_2 Y + \beta_3 X^2 + \beta_4 XY + \beta_5 Y^2$, etc. Compute this using `svd` and basic linear algebra manipulations that you've learned in class!

   (c) For each of the polynomials, (a) plot the fitted surface (use `surf`) and data points (use `plot3`) in the same figure, and rotate it around to convince yourself that the fit is reasonable. (b) compute the error for each element of $Z$, plot a histogram of these values, and compute the mean of the squared errors. How does the error behave as you increase the order of the polynomial? Which polynomial do you think gives the "best" fit? Explain.

3. **Trimmed regression.** One of the limitations of least-squares regression is sensitivity to outliers. A common solution is to iteratively discard the bad points. Load the file `regress3.mat`. First solve the standard regression problem, using a constant and a linear term. Then write a loop that locates the data point with the largest magnitude error (use function `max`), eliminates that row from the data vector and regressor matrix, and then re-solves the regression

problem. Note that you can also do this by including a diagonal weight matrix, and zeroing the entry corresponding to the outlier. On each iteration, plot the points and the best-fitting regression line, plot a histogram (use `histogram` in MATLAB or `plt.hist` in Python) of the squared errors over all points, and record the average over these errors (i.e., on the nth iteration, save the average error in the nth element of the vector). You may want put the two plots in separate figure windows, and you may want to pause after each iteration (use `pause` in MATLAB or `time.sleep` in Python). Run the loop until half of the data points have been discarded.

After running, plot the vector of average errors, as a function of iteration, in a third figure. Based on this, and the histograms you observed, which iteration gave the "best" fit? To visualize this solution, plot the corresponding regression line, and *all* of the data points, labeling the discarded data points with a different plot symbol. Did you make a good choice?

4. **Constrained Least Squares Optimization.** Load the file `constrainedLS.mat` into a MATLAB or Jupyter notebook. This contains an $N \times 2$ data matrix, `data`, whose columns correspond to horizontal and vertical coordinates of a set of 2D data points, $\vec{d}_n$ (note that each $\vec{d}_n$ is a column vector but is a row of the matrix `data`). It also contains a 2-vector `w`. Consider a constrained optimization problem:

$$\min_{\vec{\beta}} \sum_n \left(\vec{\beta}^T \vec{d}_n\right)^2, \quad \text{s.t.} \quad \vec{\beta}^T \vec{w} = 1.$$

There is a family of possible vectors $\vec{\beta}$ that satisfy the *constraint* $\vec{\beta}^T \vec{w} = 1$. Geometrically, any $\vec{\beta}$ whose arrow-tip lies on a specific line perpendicular to $\vec{w}$ will satisfy the constraint. The perpendicular distance of this constraint line from the origin will be $1/||\vec{w}||$ from the origin (think about the dot product, draw the vector $\vec{w}$ and the constraint line to prove this to yourself). Thus, this is a new contrained optimization that is a bit like total least squares, except that $\beta$ is forced to satisfy a linear constraint, rather than forced to be a unit vector.

(a) Rewrite the optimization problem in matrix form. Then rewrite the problem in terms of a new optimization variable, $\tilde{\beta}$ (i.e. 'beta tilde', a linear transformation of $\vec{\beta}$), such that the quantity to be minimized is now $||\tilde{\beta}||^2$. Note: you must also rewrite the constraint in terms of $\tilde{\beta}$.

(b) The transformed problem is one that you should be able to solve. In particular, you must find the shortest vector $\tilde{\beta}$ that lies on the constraint line. Compute the solution for $\tilde{\beta}$, and plot the solution vector, the constraint line and the data points after transforming them in the same way that $\beta$ was transformed to $\tilde{\beta}$.

(c) Transform the solution back into the original space (i.e., solve for $\vec{\beta}$). Plot $\vec{\beta}$, the original constraint line, and the original data points. Is the optimal vector $\vec{\beta}$ perpendicular to the constraint line? On the same graph, plot the total least squares solution (i.e., the vector that minimizes the same objective function, but that is constrained to be a unit vector). Are the two solutions the same?