COMPSCI 527 — Homework 1

Due on September 8, 2016

Motivation for this Assignment. COMPSCI 527 requires basic knowledge of linear algebra, probability, and MATLAB programming. This assignment is meant as a way for you to assess your own level of comfort with these topics (except probability, whose use will be minimal this year). The due date is before the add/drop deadline, so you can make appropriate decisions when the stakes are still comparatively low. However, the only feedback you will get before that deadline is your own assessment (it takes time to grade, and the class is large this year).

You may be rusty on some of the material, and in that case you may struggle with some of the questions below. That's OK, as struggle leads to learning. On the other hand, you may get the impression that this assignment is way over your head. In that case, you are urged to reconsider taking this class. You may come back in a later year, once you are more comfortable with the prerequisites.

Collaboration and Communication Policy. You are *urged* to work in pairs on this and future assignments. This helps you discuss concepts with a partner, make working on the assignment more pleasant, and reduce the likelihood that you will get stuck with a problem. It also helps us with grading: There are many of you, and we will be able to return your graded work earlier if we have fewer papers to read.

Pairs can change for different assignments. However, once you started to work with a person on a particular assignment, you cannot change your partner for that assignment. For instance, if your partner goes AWOL after the two of you start working on the assignment, you are on your own (please remove your partner's name from the assignment). Resist the temptation to divvy up the work: You will pay for that in the exams, as you will not have had practice on half of the material.

The only allowed communication of any kind and in any form on any of the assignments consists of (i) communication with your partner and (ii) communication with this class's teaching staff. Everything else will be considered cheating. It is OK to use the web or books or other materials to study concepts. *It is cheating to use the web or books or other materials to look for solutions, solution ideas, or "inspiration" for a solution. All suspected violations of academic integrity will be fully investigated and prosecuted as warranted.*

Late Policy. Grades are out of 100, and there is a 10-point (not 10 percent) penalty per *calendar* day of delay. For instance, if you return on Monday an assignment due the previous Thursday, the penalty is 40 points.

Formatting. You are *urged* to use LaTeX for all your homework for this class. LaTeX is the *lingua franca* of computer science, mathematics, and many of the sciences, so you may as well learn it now if you do not know it already. If you use something else, you still need to hand in PDF files for the text part of the assignment (no other format accepted), and the same formatting guidelines apply. So if you use, say, MS Word, you will end up having to simulate LaTeX with Word. This will get tricky when you do math.

This assignment comes with a LaTeX template file `hw1.tex` that helps you get started. If the file contains no formatting errors, running LaTeX on it will create a file `hw1.pdf`. Suitable MATLAB `.m` files are provided as well.

The `.tex` file has problem headers and `\newpage` statements that place page breaks between problems. Please do not change headers or page breaks. This will help us read your work more easily. You may add page breaks if this is useful.

The template file also contains some text, formulas, matrices, and code to show you how those are written in case you are not familiar with LaTeX. Please remove all this extra LaTeX input from your file, so we don't have to read our own code and text.

Make sure that your PDF file is formatted correctly and that all lines of text and code show up in their entirety.

Code. Please intersperse concise comments with your code whenever useful.

Each piece of code you submit must show up in two places: Your `.pdf` file and a separate `.m` file, as per instructions in each problem. One easy way to incorporate code in a `.pdf` file is to include it in a `verbatim` environment. In any case, the code must be typeset in a `fixed-width font` (for instance, `Courier`) in your PDF file. This is automatic if you use `verbatim`. Make sure that all your code and comments show up in their entirety. If not, use line breaks or make the font size for the code `\small` (but no smaller).

What to Submit. The name of the person or people who worked on the assignment must appear on the first page (use the `\author{}` command). If you work with someone else, *submit only one set of files to Sakai*, not two. It does not matter who submits it. As long as both names show up on the first page, we will give the same grade to both authors.

Submit exactly the following files to Sakai: `hw1.pdf` plus one MATLAB file for every function you are asked to code. Do not rename the files. If you skip all the work in a file, submit the unmodified file anyway. For instance, if you do not implement the function `showBits.m`, submit the file `showBits.m` all the same, without changing it.

All files must be submitted as separate attachments. Do *not* use the Sakai drop box. Be careful to include all the required files. Here is the file checklist for this assignment:

`hw1.pdf`, `embed.m`, `showBits.m`, `retrieve.m`

Keep in mind that your code must also show up in `hw1.pdf` .

Linear Algebra

The document on *Linear Transformations* on the syllabus page will not be covered in class, and is a succinct summary of some of the required linear algebra concepts. You may want to study that before you do the problems below if you are rusty on the concepts. No explanations are needed in your answers, unless explicitly requested.

1. Are the vectors

$$\begin{bmatrix} 2 \\ 4 \\ 3 \end{bmatrix} \quad , \quad \begin{bmatrix} 1 \\ 2 \\ 0 \end{bmatrix} \quad , \quad \begin{bmatrix} 0 \\ 0 \\ 3 \end{bmatrix}$$

linearly independent?

2. List *all* the possible sizes of the set S of solutions to a linear system

$$A\mathbf{x} = \mathbf{b}$$

where A is a 3×2 matrix. The size (or cardinality) of a set is the number of its elements. Your choices are integers or expressions of the form ∞^k, which represents the size of a k-dimensional space of real numbers. For instance, the plane has ∞^2 points.

3. Give an expression for the set S of *all* solutions to the following linear system:

$$\begin{bmatrix} 2 & 2 \\ 1 & 1 \end{bmatrix} \mathbf{x} = \begin{bmatrix} 4 \\ 3 \end{bmatrix} .$$

4. Give a simple parametric expression for the set S of *all* solutions to the following linear system:

$$\begin{bmatrix} 2 & 2 \\ 1 & 1 \end{bmatrix} \mathbf{x} = \begin{bmatrix} 4 \\ 2 \end{bmatrix} .$$

A parametric expression defines a set in terms of one or more independent variables called parameters. For instance, a parametric expression for the plane $x_2 = 4$ in $\mathbb{R}^3$ is as follows:

$$\left\{ \mathbf{x} \;\middle|\; \mathbf{x} = \begin{bmatrix} 0 \\ 4 \\ 0 \end{bmatrix} + \alpha \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} + \beta \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \text{ for } \alpha, \beta \in \mathbb{R} \right\} .$$

As α and β vary, all and only the points on that plane are described. Different parametric expressions are possible for the same set.

5. Use the Gram-Schmidt procedure to find orthonormal vectors $\mathbf{q}_i$ that span the same space as the vectors

$$\mathbf{a}_1 = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} \quad , \quad \mathbf{a}_2 = \begin{bmatrix} 0 \\ 2 \\ 2 \end{bmatrix} \quad , \quad \mathbf{a}_3 = \begin{bmatrix} 3 \\ 0 \\ 0 \end{bmatrix} .$$

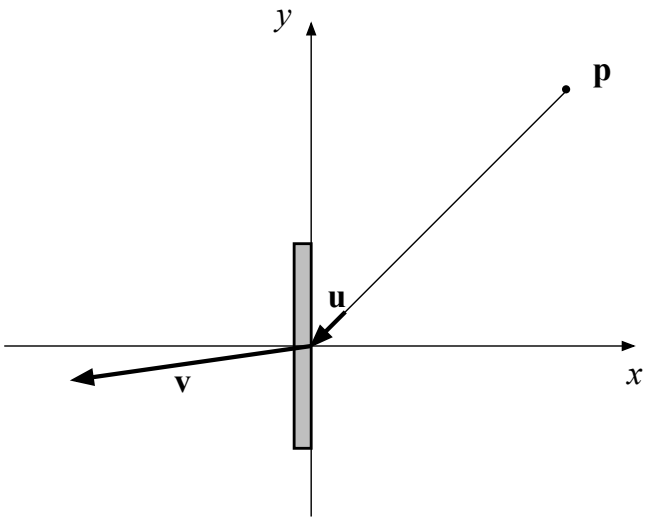
"Orthonormal" means orthogonal and of unit norm. When using Gram-Schmidt, incorporate the vectors $\mathbf{a}_1$, $\mathbf{a}_2$, $\mathbf{a}_3$ in this order.

Your solution should be exact, meaning that numerical values should not be approximated. For instance, $1/\sqrt{2}$ should stay that, or perhaps be written as $\sqrt{2}/2$ (your choice), but it should not be approximated with, say, 0.7071. No need to show your calculations, just the result.

6. What are the dimensions of null space, row space, left null space, and range of the following matrix?

$$A = \begin{bmatrix} 1 & 0 & 3 \\ 0 & 2 & 0 \\ 1 & 2 & 0 \end{bmatrix}$$

[Hint: The columns of A are from the previous problem.]



7. You are practicing tennis by bouncing a ball against a wall. In describing the situation, we ignore the height off the ground, so we can model everything in two dimensions, as shown above. Place the origin of a two-dimensional Cartesian reference system at the point where the ball hits the wall, which is built along the y axis. You stand somewhere in the $x > 0$ half-plane as you hit the ball with your racket. That is, your coordinates are $\mathbf{p} = (x, y)$ with $x > 0$. Let $\mathbf{u}$ be the (two-dimensional) velocity vector of the ball as it hits the wall. Since we assume that the ball approaches the wall along a straight line and at constant velocity (gravity does not matter in the (x, y) plane), the direction of the velocity vector $\mathbf{u}$ is the same as that of the reverse $-\mathbf{p}$ of the position vector, while the magnitude of $\mathbf{u}$ depends on how hard you hit the ball.

The wall is made of a very strange mesh-like material: If the ball hits with velocity vector $\mathbf{u}$, then the ball leaves the wall with velocity vector

$$\mathbf{v} = A\mathbf{u} \quad \text{where} \quad A = \begin{bmatrix} 3 & 5\sqrt{3} \\ 5\sqrt{3} & -7 \end{bmatrix} .$$

If you try out some input velocity vectors $\mathbf{u}$ in the expression above, you will see that in some cases the ball goes *through* the wall. For instance, in the Figure above, your position $\mathbf{p}$ is along the line

$$y = x .$$

If you hit the ball with velocity

$$\mathbf{u} = -\begin{bmatrix} 1 \\ 1 \end{bmatrix} ,$$

then the ball goes through the wall in this case, because it exits with velocity

$$\mathbf{v} = -\begin{bmatrix} 5\sqrt{3}+3 \\ 5\sqrt{3}-7 \end{bmatrix} \approx -\begin{bmatrix} 11.66 \\ 1.66 \end{bmatrix} ,$$

whose x component is still negative. The exit velocity $\mathbf{v}$ in this case is much faster and closer to the x axis than the approach velocity $\mathbf{u}$. For some other approach velocities, the ball will bounce back into the $x > 0$ half-plane. Note that the x component of $\mathbf{u}$ is always negative, because you are in the $x > 0$ half-plane and hit the ball towards the wall. The sign of the y component of $\mathbf{u}$, on the other hand, can change depending on where you stand.

For the **questions on the next page**, and where appropriate, give both exact answers, in the sense defined in problem 5, and numerical approximations rounded to two digits after the period, as done for $\mathbf{v}$ in the example above. If you are confident of your answers, you need not give any explanation. If you are unsure, a brief and crystal-clear account of your reasoning may help you earn some partial credit.

(a) On what line through the origin do you need to be so that the ball comes exactly back at you after it hits the wall? Give the line in the form of an equation, either parametric or implicit, as you prefer. A parametric equation has the form

$$\mathbf{p} = \alpha\mathbf{q}$$

for fixed vector $\mathbf{q}$ and variable α, and an implicit equation has the form

$$y = \beta x$$

for fixed β and variables x, y. [Hint: Think eigenvalues/eigenvectors.]

(b) If you are on the line you found above, how much faster or slower does the ball come back at you, compared to how fast you hit it?

(c) Suppose that the relationship between $\mathbf{u}$ and $\mathbf{v}$ is the same regardless of what side of the wall you stand on. If you are on the line you found above but on the other side of the wall (*i.e.*, on the $x < 0$ half-plane), will the ball come back at you or go through the wall?

Coding

The exercise below will challenge you to think about arrays, images, bits, and data format conversion. If you don't know MATLAB but know some other programming language, it won't take you long to learn from the tutorials on the MathWorks Tutorials site. If you do not know how to program at all, you do not have the prerequisites for this course.

MATLAB started out as a matrix manipulation language, so it is particularly good at expressing matrix operations. Any operator that applies to a scalar can typically be applied to a matrix as well, and operates on each entry separately. For instance, if `X` is a matrix, then the instruction

```
S = sin(X);
```

will output a matrix `S` of the same size as `X` that contains the sines of all the entries of `X`.

MATLAB also has extensive indexing constructs. One that is often overlooked is the use of a logical predicate as an index. For example,

```
A(A > 200) = 200;
```

will replace all elements in array `A` whose value is greater than 200 with 200. The expression `A > 200` denotes an array of logical values with the same dimensions as `A`, and that logical array is then used to select all entries of `A` where the predicate is true.

Keep these observations in mind and try to avoid using iterative constructs like `for` or `while` loops explicitly when processing image pixels in your code. It is OK to use a `for` loop to loop over a small number of possibilities, but it is very inefficient in MATLAB to loop over the (typically very many) pixels of an image. **You will be penalized 3 points for every construct that explicitly iterates over an image in your code for this problem.** This problem is worth 30 points out of 100.

[Hint: In case you are not familiar with all the relevant MATLAB functions, here is a list of all the ones I used for my solution sample:

> `bin2dec`, `char`, `dec2bin`, `double`, `error`, `false`, `find`, `floor`, `imresize`, `isa`, `isempty`, `length`, `logical`, `mod`, `prod`, `reshape`, `size`, `uint8`, `zeros`

Of course, you may end up using a different set. I did use a `for` loop, but over just three values of its loop variable.]

8. For our purposes, a gray-level image is a two-dimensional array of `uint8` items, so each pixel can take one of 256 values. A color image is a three-dimensional array where the three dimensions represent the values of red, green, and blue for each pixel (for instance, `img(30, 50, 2)` is the green value (`2`) of the pixel at row 30, column 50). A color image with three identical red, green, blue sub-arrays looks the same and uses three times as much storage as a gray-level image.

Steganography is the study of how to embed a secret message in some data record. We can secretly embed a string in a gray-level image by observing that two pixels that differ by just one gray level (say, one has value 120 and the other has value 121) are virtually indistinguishable to the human eye. If we want to embed, say, the string `'banana'` into an image, we can convert the string to the sequence of its 8-bit ASCII codes represented in binary. The MATLAB function `dec2bin` does a good part of the job, but outputs an array of characters. To use the result as an index, convert it to a vector of logical values, call it `bits`. Since each ASCII code has 8 bits, there is a total of 48 bits for `'banana'`. Let us also append the bits `00000000` to denote the end of the string, for a total of 56 bits. Characters are represented left-to-right (so the first one is `b`), and so are the bits for each character (so the bits for `b` are `01100010`, which represent the ASCII value 98). [Warning: `dec2bin` needs a second argument to ensure that the output always has eight bits.] Here is the encoding of `'banana'`:

```
01100010011000010110111001100001011011100110000100000000
```

It is convenient for the vector `bits` to have as many entries as the image has pixels, so it can be used as a logical index into the image. To this end, the vector is padded with enough zeros appended at the end, or the string is properly truncated. In the example, if the image has n pixels and $n > 56$, then the vector `bits` is padded with $n - 56$ zeros (logical `false` values). If the image has fewer than 56 pixels, then the input string `s` is truncated (at its tail end) so that it has as many characters as possible without exceeding 56 bits, including the 8-bit end-of-string code. Ignore images with 7 or fewer pixels.

To embed string `s` into image `img` we replace the least-significant bit of every pixel with a bit from `bits`, in the order in which they appear (left-to-right). Pixels are numbered one image *column* at a time, so they are considered in the order in which they occur in the vector `img(:)`. The resulting image will be visually indistinguishable from the original, and the string `s` can be retrieved from the image by the reverse operations: Extract all the least-significant bits from the image, package them into groups of eight, and convert each group to an ASCII character.

Programming Note: Make sure all your images are of type `uint8` for input, output, or display. Some numerical operations may be safer as `double`. If you want to make a `uint8` image with all zeros, you can say (assuming `rows` and `cols` are properly defined)

```
img = zeros(rows, cols, 'uint8');
```

An Aside on Image Files: The images that come with this assignment are stored in the PNG format, which is uncompressed. If images were compressed in a lossy format such as JPEG, compression would interfere with the least-significant bits and would destroy the secret message. Thus, if you want to store your images to disk, make sure you use an uncompressed format such as PNG (by specifying a `.png` file extension), or a lossless compression format such as GIF.

(a) Write a MATLAB function with header

```
function img = embed(s, img)
```

that embeds string `s` into gray-level image `img`. The provided template file `embed.m` contains a check that `img` is gray-level, and issues an error otherwise (the function `grayImageSize` used in `embed.m` is provided as well).

Turn in your code, both in your PDF and in a separate file `embed.m`. In your PDF, also show side-to-side in a figure the input and output image obtained when running your code on the image `eye.png` provided with the assignment and with `s` equal to `'banana'`. The two images should be visually indistinguishable from each other.

(b) For debugging and confirmation purposes it may be useful to show in a visual way which bits are embedded in which pixel in a given image. Write a MATLAB function with header

```
function cimg = showBits(img, color)
```

that takes a gray-level image `img` with a string embedded in it and optionally a `color` argument described below. The function makes a color image `cimg` with the same number of rows and columns as `img`. For instance, if `img` has size `[100 150]`, then `cimg` has size `[100 150 3]`. The output image looks like the input image at all pixels where a zero bit is embedded (thus, each pixel value there is replicated in the red, green, and blue components of `cimg`). All the pixels where a one bit is embedded are displayed in the color given as argument, or with a default color otherwise.

The argument `color` is an array of three `uint8` integers that represent the red, green, and blue components of a single pixel color. The file `showBits.m` provided with this assignment contains a default assignment to `color` in case this argument is omitted or set to the empty array. The resulting image `cimg` shows the locations of all the 'on' bits in orange (default color).

Turn in your code, both in your PDF and in a separate file `showBits.m`. In your PDF, also include a figure that shows the following two images side-to-side: Left: The color image resulting from running your `showBits` function on the image read from `drawing.png`, with the default color; Right: The upper-left 56×56 pixel square region of the color image `showBits(embed('banana', eye))`, where `eye` is the image read from `eye.png` (only few pixels will be orange here).

The image `drawing.png` does *not* embed a string of characters, but rather a set of bits that will itself show up as an intelligible drawing when displayed with `showBits`. Thus, the first image tests your `showBits`, the second tests your `embed`.

Programming Note: Displaying a 56×56 image directly would result in a blurred display, for reasons we will learn in class later. To avoid this blur, and if the 56×56 image is called, say, `corner`, display `big` instead, obtained as follows:

```
big = imresize(corner, [512 512], 'nearest')
```

(c) Write a MATLAB function with header

```
function s = retrieve(img)
```

that retrieves the string embedded in the given image. The function should return the empty string if it does not find the end-of-string code `00000000`.

Turn in your code, both in your PDF and in a separate file `retrieve.m`. In your PDF, also show the string retrieved from the image `string.png` provided. Embed the string in a `verbatim` environment in your `.tex` file, so that LaTeX does not garble it with its own formatting.