Homework 7 Solutions

CAS CS 132: Geometric Algorithms

Due: **Thursday November 2, 2023 at 11:59PM**

Submission Instructions

- Make the answer in your solution to each problem abundantly clear (e.g., put a box around your answer or used a colored font if there is a lot of text which is not part of the answer).
- Choose the correct pages corresponding to each problem in Gradescope. Note that Gradescope registers your submission as soon as you submit it, so you don't need to rush to choose corresponding pages. **For multipart questions, please make sure each part is accounted for.**

Graders have license to dock points if either of the above instructions are not properly followed.

Note. Solutions written here may be lengthy because they are expository, and may not reflect that amount of detail that you were expected to write in your own solutions.

Practice Problems

The following list of problems comes from *Linear Algebra and its Application 5th Ed* by David C. Lay, Steven R. Lay, and Judi J. McDonald. They may be useful for solidifying your understanding of the material and for studying in general. **They are optional, so please don't submit anything for them**.

- 2.5.1-3, 2.5.14, 2.5.26
- Example 8 (useful for Problem 2), 2.7.2, 2.7.3-8, 2.7.12, 2.7.15, 2.7.16. 2.7.17

1 Sparse Factorizations

In this problem, you will use the provided code in `sparse.py` to benchmark LU factorization and matrix inversion.

A. (2 points) The provided code includes a function `test_matrix` which builds a matrix given a positive integer parameter. Write down the matrix returned by this function applied to 4.

B. (8 points) There is a function in the `scipy` library called `lu` which returns the LU factorization of a matrix. Use this to construct the LU factoration of $A =$ `test_matrix(10 ** 3)`. This function returns a 3-tuple of matrices. The 2nd and 3rd entries are L and U, respectively, where $A = LU$.

   Then use the function `np.linalg.inv` to construct the inverse A^{-1} of this matrix.

   Finally, you can use the expression `len(np.nonzero(mat)[0])` to determine the number of nonzero entries in a 2D numpy array `mat`. **Write down the number of nonzero entires of L, U, and A^{-1}, where A is the matrix `test_matrix(10 ** 3)`, as well as the lines of code you used.**

C. (5 points) Uncomment the code at the end of the file `sparse.py` and run it. Write down the printed values and discuss what they mean. In particular, is the amount of time that it takes to factor versus invert consistent with our discussion in lecture?[1]

[1] You'll note that solving systems with the matrix inverse is faster than with the LU factorization. Even though the two have roughly the same theoretical guarantees, matrix-vector multiplication implementations tend to be better optimized.

Solution.

A.

$$\begin{bmatrix} 4 & -1 & -1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 4 & 0 & -1 & 0 & 0 & 0 & 0 \\ -1 & 0 & 4 & -1 & -1 & 0 & 0 & 0 \\ 0 & -1 & -1 & 4 & 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 & 4 & -1 & -1 & 0 \\ 0 & 0 & 0 & -1 & -1 & 4 & 0 & -1 \\ 0 & 0 & 0 & 0 & -1 & 0 & 4 & -1 \\ 0 & 0 & 0 & 0 & 0 & -1 & -1 & 4 \end{bmatrix}$$

B. The code

```
a = test_matrix(10 ** 3)
p, l, u = lu(a)
a_inv = np.linalg.inv(a)
print(f'# entries in L: {len(np.nonzero(l)[0])}')
print(f'# entries in U: {len(np.nonzero(u)[0])}')
print(f'# entries in A inverse: {len(np.nonzero(a_inv)[0])}')
```

gives us

```
# entries in L: 5997
# entries in U: 5997
# entries in A inverse: 3792976
```

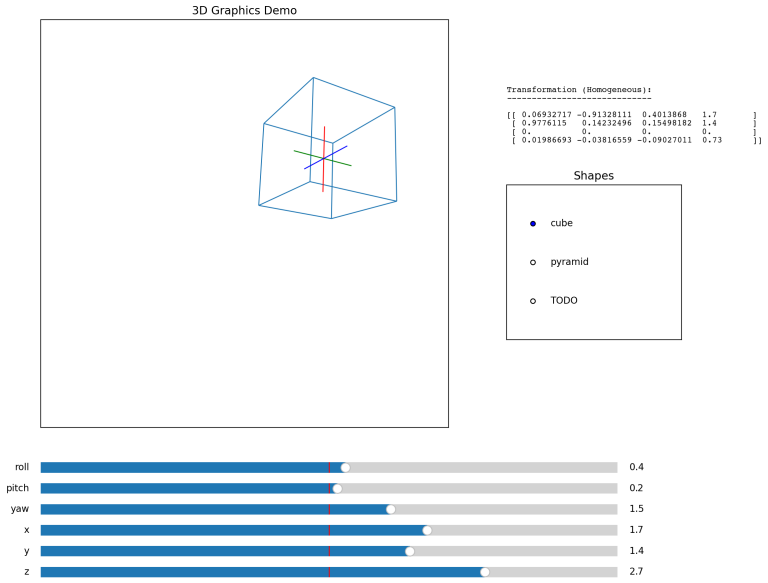
C. Runnning the given code gives us

```
building test matrix...
factoring m...
time: 9.96743106842041 seconds
solving with LU...
time: 7.4140448570251465 seconds
inverting m...
time: 35.37879991531372 seconds
solving with inv...
time: 2.086169958114624 seconds
```

This is relatively consistent with what we discussed in class. The time it takes to invert a matrix should be roughly 3 times longer than factoring.

2 3D Graphics (Programming)

This week you will be building an interactive matplotlib widget which can manipulate 2D renderings of 3D wireframes. Once you are done, it should look something like this, and you should be able to rotate and translate a wireframe object on screen using provided sliders.



You are given starter code in the file `hw07prog.py`. **Don't change the name of this file when you submit.** Also don't change the names of any functions included in the starter code. **The only changes you should make are to fill in the provided TODO items.** At a high level, the code works as follows.

- Wireframes are represented as lists of line segments, which are represented as 2-element lists of 3-tuples of floats (in other words, points in $\mathbb{R}^3$). You are provided with two simple wireframes, `cube` and `pyramid`. You will have an opportunity, for a small amount of extra credit, to build your own wireframe. You can use the structure of these two examples as a guide.

- These wireframes are transformed into matrices whose columns are homogeneous coordinates of the endpoints of each segment in the wireframe. This is done by the function `shape_to_hom_matrix`, which is implemented for you. At this point, red, green, and blue guide axes are included in the wireframe. So if the wireframe has 12 line segments, then this function returns a $4 \times (2 * (12 + 3)) = 4 \times 30$ matrix. In later steps we will call this matrix W. These matrices are collected in a dictionary called `shape_matrices` which is used by the latter parts of the program. **You**

do not need to change any part of this code (that is, the code under the header "SHAPES") except for potentially the extra credit wireframe.

- The next step is to build the transformation that will ultimately augment the wireframe (in other words, the transform that will be applied to the matrix W). There are a number of basic matrix constructions you have to fill in:
  - `perspective`
  - `hom_rotate_x`
  - `hom_rotate_y`
  - `hom_rotate_z`
  - `translate`

  Once you fill in these functions, you will combine them into a single product of matrices which represent the transformation to the wireframe that will be rendered on screen. **You will implement this in the function `full_transform_matrix`.** Since this will be a multiplication, you are **required** to use the function `numpy.linalg.multi_dot`. Please see the NumPy documentation for information on how to use this function.

  The matrix returned by this function should rotate and then translate the given wireframe, and then apply the perspective projection. It is important to remember that **the order of multiplications matters**. As a guide, when you finish, the roll axis should be red, and when you translate your shape, rotation should keep its center fixed. I will call the matrix returned by `full_transform_matrix` T in latter steps.

- The last step it to construct a collection of 2D line segments from the matrix TW. **You will be implementing this in `matrix_to_projection`.** Because of the perpsective projection which occurs in T, the columns of TW should have the form $[x\ y\ 0\ h]^T$. You need to convert each column to a point in $\mathbb{R}^2$ of the form $(x/h, y/h)$. You then need combine **every two** points into a 2-element list representing a line segment. Remember that they points are transformed versions of the endpoints of the given wireframe, and we build the matrix in first step by combining every endpoint into a single matrix. All of these line segments should be collected into a single list which is then returned. See the provided example in the docstring.

  As usual there is a long way and a short way to do this. If you want to try to give a more concise implementation, look at the function in the NumPy documentation called `numpy.linalg.apply_along_axis`.

- The rest of the code passes the output of the above two steps into matplotlib to be displayed, and connects all the radio buttons and sliders. **You do not need to change any part of this code** but I recommend looking through it if you're interested in how to build matplotlib widgets.

All together, there are 7 functions to fill in:

- (6 points) `perspective`
- (6 points) `hom_rotate_x`
- (6 points) `hom_rotate_y`
- (6 points) `hom_rotate_z`
- (6 points) `translate`
- (15 points) `full_transform_matrix`
- (20 points) `matrix_to_projection`

Extra credit (5 points). Build a wireframe (of some reasonable complexity) which you can use in the program. In order to get any credit you must include **in the analytical part of your submission** an image of your rendered wireframe produced by matplotlib, translated and rotates some amount (like in the example image above). In particular, you have to first complete the required parts of the assignment. The credit you recieve will be determined in part by how complex the wireframe is. Also, please state along with your image if you are comfortable making the image public (in case I collected the results on a public-facing webpage).

You will upload a single file `hw07prog.py` to Gradescope, where you can verify that it passes some (but not all) autograder tests. **Please test your system early.** There may be system dependent issues that we'd like to address as early as possible.