Numerical Analysis (CS 450)

Homework Set 2

Due: February 26, 2014 · Out: February 12, 2014

Remarks:

- Use a fixed random seed whenever this homework set asks for a random matrix:

  ```
  import numpy as np
  np.random.seed(12) # or some such
  ```

- Whenever a problem asks to 'print' something, please provide the output in the PDF you turn in.

  Make sure your program prints labels for each number being printed, so that the output can be understood without examining your code.

  Also note that we do not want pages upon pages of output—in general no more than a page or three per problem. Unless the problem specifically asks for matrix or vector entries, we usually do not want to see those. Use norms and similar ways of summarizing information.

- Unless a problem specifies otherwise, relative errors are preferable to absolute errors.

Problem 1: Develop the Cholesky factorization (25 points)

(a) Suppose that the matrix $\mathbf{A}$ has a factorization of the form $\mathbf{A} = \mathbf{B}\mathbf{B}^T$, with $\mathbf{B}$ nonsingular. Show that $\mathbf{A}$ must be

(i) symmetric, and

(ii) positive definite, i.e. that

$$\mathbf{x}^T\mathbf{A}\mathbf{x} > 0 \quad \text{for } \mathbf{x} \neq \mathbf{0}. \tag{1}$$

Matrices of with both of these properties are often called "*SPD*", for *symmetric positive definite*, and they are an important special class of matrices. This problem develops a special form of LU factorization for them.

(b) A symmetric and positive definite matrix $\mathbf{A}$ has an LU factorization which can be written as $\mathbf{A} = \mathbf{L}\mathbf{L}^T$ where $\mathbf{L}$ is lower triangular and has positive diagonal entries. In other words, for SPD matrices the factors $\mathbf{L}$ and $\mathbf{U}$ of the LU decomposition can be made to be transposes of each other. This factorization is known as the *Cholesky* factorization. It can be computed by equating the corresponding entries of $\mathbf{A}$ and $\mathbf{L}\mathbf{L}^T$.

Considering a 3×3 symmetric positive definite matrix as shown in (2),

$$\begin{bmatrix} a_{11} & a_{21} & a_{31} \\ a_{21} & a_{22} & a_{32} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{bmatrix} \begin{bmatrix} l_{11} & l_{21} & l_{31} \\ 0 & l_{22} & l_{32} \\ 0 & 0 & l_{33} \end{bmatrix} \tag{2}$$

Proceed as follows:

(continued on next page...)

(i) Compute the matrix product $\mathbf{L}\mathbf{L}^T$ symbolically. (Symmetry helps. :)

(ii) Equate the entries of $\mathbf{L}\mathbf{L}^T$ with those of $\mathbf{A}$.

(iii) Find an ordering so that each entry l_{ij} can be computed by only referring to already computed previous values.

If you find that you have a choice in ordering, finish computing each row of $\mathbf{L}$ before moving on to the next one.

Derive expressions for the matrix elements of the lower triangular factor $\mathbf{L}$.

(c) Generalizing part (b), find an algorithm that computes the Cholesky factor $\mathbf{L}$ for a given symmetric, positive-definite matrix $\mathbf{A}$ of size $n \times n$.

Hints:

(i) Start with l_{11} and proceed through rows.

(ii) Finish computing each row of $\mathbf{L}$ before moving on to the next one.

(iii) Only one half of the matrix (lower half) $\mathbf{A}$ needs to be accessed.

(d) Implement the algorithm devised in part (c) and run it on three random SPD matrices of size 20×20. Here is some code to make a random SPD matrix:

```
import numpy as np

def rand_spd(n):
    A = np.random.rand(n,n)
    return np.dot(A, A.T)
```

For each test case, print

- the relative error $\|\mathbf{L}\mathbf{L}^T - \mathbf{A}\|_2 / \|\mathbf{A}\|_2$, where $\mathbf{L}$ is your computed Cholesky factor.
- the condition number of the matrix, as found by `numpy.linalg.cond`.

Note that there is no need for pivoting.

(e) Compute the determinant of three random SPD matrices (of sizes 5, 10 and 100, computed as above) using their Cholesky factors and compare the results with the determinant obtained using the `numpy.linalg.det()` function. For each example, print the following:

- the size of the matrix n,
- the value from `numpy.linalg.det()`,
- your computed determinant,
- and the relative error between the two,assuming result from `numpy.linalg.det()` to be the 'true' value.

Hints:

- $\det(\mathbf{A}\mathbf{B}) = \det(\mathbf{A})\det(\mathbf{B})$,
- the determinant of a triangular matrix is the product of its diagonal entries.

(continued on next page...)

Problem 2: Transform matrices (15 points)

Consider the vector

$$\mathbf{a} = \begin{bmatrix} 3 \\ 5 \\ 12 \end{bmatrix} \tag{3}$$

(a) Calculate a 3×3 elementary (Gaussian) elimination matrix to eliminate the third component of $\mathbf{a}$ using the first component.

(b) Calculate a 3×3 Householder transformation matrix that eliminates only the third component of $\mathbf{a}$. Also specify α and the Householder vector $\mathbf{v}$.

(c) Calculate a 3×3 Givens rotation matrix that annihilates the third component of $\mathbf{a}$ using the second component.

In each case, your answer should consist of:

- the 3×3 transformation matrix $\mathbf{M}$,
- the product $\mathbf{Ma}$,
- and any extra information requested above.

Problem 3: Least squares fitting with Gram-Schmidt and QR (20 points)

(a) Write a program that implements QR factorization using the modified Gram-Schmidt procedure.

(b) Write a program that implements QR factorization using Householder reflectors.

(c) Compute the QR factorization for three random matrices (of sizes 5×5, 10×10, and 100×80) using both the algorithms.

For each test case, print

- the matrix shape,
- the relative error $\|\mathbf{QR} - \mathbf{A}\|_2 / \|\mathbf{A}\|_2$, where $\mathbf{Q}$ and $\mathbf{R}$ are your computed QR factors, and
- the condition number of the matrix, as found by `numpy.linalg.cond`.

(d) Use each of above two algorithms and `numpy.linalg.lstsq()` to solve a least squares data fitting problem.

The file `Price_of_Gasoline.txt` contains prices of gasoline over a period of 345 months. Construct least squares problems to fit the data to polynomials of degree 1 to 5.

For each polynomial degree, and for each method, provide the following, clearly identifying degree and method used:

- the approximate polynomial, in the format:

```
a * x^2 + b *x + c
a = 0.324234
b = 1.34234
c = 3.2374
```


(continued on next page...)

- the relative residual $\|Ax - b\|_2 / \|b\|$ of the least squares problem solved.

For each method, also provide a plot of the data and the five fitted polynomials (one plot per method).

Provide an evaluation:

- Do the methods differ in the relative error obtained? If so, which one is more accurate?
- Do the polynomial degrees differ in the relative error obtained? If so, which one provides the best approximant? Why?

Hint: You may use the following code to read the data into an array.

```
import numpy as np
v = np.genfromtxt("Price_of_Gasoline.txt", delimiter="\n")
```

Problem 4: Eigenvalue finding (15 points)

In each of the following, consider the iteration converged if the 2-norm of the relative difference of subsequent iterates has changed by 10^{-12} or less.

(a) Implement inverse iteration with a shift to compute the eigenvalue nearest to 2, and a corresponding (2-)normalized eigenvector, of the matrix

$$\mathbf{A} = \begin{bmatrix} 6 & 2 & 1 \\ 2 & 3 & 1 \\ 1 & 1 & 1 \end{bmatrix}. \tag{4}$$

Run your program using random starting vectors with 10 different (but deterministic) random seeds.

Print the final estimate of the eigenvalue, the eigenvector, and the number of iterations.

(b) Write a program to implement Rayleigh quotient iteration for computing an eigenvalue and corresponding (2-)normalized eigenvector of a matrix. Test your program on the matrix given in (4) using the same random starting vectors as in the previous part.

Print the final estimate of the eigenvalue, the eigenvector, and the number of iterations.

(c) In comparing the eigenvalue estimates produced by the two previous parts, what do you observe?

(d) Use `scipy.linalg.eig()` to compute all of the eigenvalues and eigenvectors of the matrix, and compare the results with those obtained in parts (a) and (b).

Modify your code for (a) and (b) so that it finds the eigenvalue output by `eig` which is closest to your approximate value, and using the value from `eig` as the 'true' value, outputs the relative error in the eigenvalue and the relative error of the eigenvector in the 2-norm.

(e) Using $[1, 4, 2]^T$ as the starting vector, print and compare the computed eigenvalue and the computed eigenvector of inverse iteration and Rayleigh quotient iteration. You may assume either value to be the actual value. Also print and compare the number of iterations each method takes for convergence. Which one converges more rapidly?

(continued on next page...)

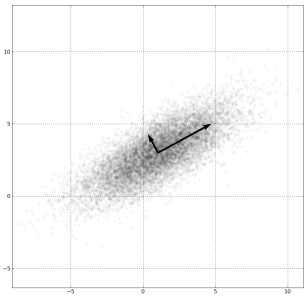

Figure 1: Principal components of a two-dimensional data set.

Problem 5: Principal component analysis (25 points)

Principal component analysis[1] is a statistical procedure that can be used to find the 'main axes' of a data set. Here, data set refers to a set of points in n-dimensional space, where the idea is that most of the data lies 'around' these main axes. See Figure 1 for an intuitive idea of what this means.

In the language of statistics, this helps identify variables that vary together, or, literally, are *correlated*. It also provides so-called 'principal components', which are orthogonal to one another and uncorrelated.

For a data set $(\mathbf{x}_i)_{i=1}^N \subset \mathbb{R}^n$, PCA can be performed as follows:

(1) Compute the empirical mean:

$$\mathbf{u} := \frac{1}{N} \sum_{i=1}^{N} \mathbf{x}_i.$$

(2) Remove the mean:

$$\widetilde{\mathbf{x}}_i := \mathbf{x}_i - \mathbf{u}, \qquad i = 1, \ldots, N.$$

(3) Assemble the 'data matrix':

$$\mathbf{Y} := \frac{1}{\sqrt{N-1}} \begin{bmatrix} \widetilde{\mathbf{x}}_1 & \widetilde{\mathbf{x}}_2 & \cdots & \widetilde{\mathbf{x}}_N \end{bmatrix}$$

(4) Compute the SVD (using `numpy.linalg.svd`) of $\mathbf{Y}$

$$\mathbf{U}\mathbf{\Sigma}\mathbf{V}^T = \mathbf{Y}$$

The principal components as shown in Figure (1) can now be computed as the columns of $\mathbf{U}\mathbf{\Sigma}$. See this tutorial[2] and the Wikipedia article linked above for more.

[1] `https://en.wikipedia.org/wiki/Principal_component_analysis`
[2] `http://www.snl.salk.edu/~shlens/pca.pdf`

(continued on next page...)

(a) Plot the 2D data set generated by this function:

```
def make_data(dims=2, npts=3000):
    np.random.seed(13)

    mix_mat = np.random.randn(dims, dims)
    mean = np.random.randn(dims)

    return np.dot(
            mix_mat,
            np.random.randn(dims, npts)) + mean[:, np.newaxis]
```

(b) Compute the principal components of this same data set and draw them into the same plot as the result from (a), centered at the mean. The figure you obtain should look similar to Figure 1.

Hints:

- Use `matplotlib.pyplot.gca().set_aspect("equal")` to make sure Matplotlib does not distort angles in your figure.
- Use `matplotlib.pyplot.arrow()` to draw arrows into the figure. Draw one for each principal component.

(c) From $\mathbf{U}$, $\mathbf{\Sigma}$, and $\mathbf{V}^T$ as returned by `numpy`, reconstruct the matrix $\widetilde{\mathbf{Y}}$:

$$\widetilde{\mathbf{Y}} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T$$

Print the relative error $\|\widetilde{\mathbf{Y}} - \mathbf{Y}\|_2 / \|\mathbf{Y}\|_2$. (This should come out to about machine precision.)

(d) Now set the bottom right nonzero entry of $\mathbf{\Sigma}$ to zero. Call the result $\mathbf{\Sigma}'$ and reconstruct another matrix $\mathbf{Y}'$:

$$\mathbf{Y}' = \mathbf{U}\mathbf{\Sigma}'\mathbf{V}^T$$

Next, undo the transformations done to $\mathbf{Y}'$ to get back to plain data (undo scaling by $1/\sqrt{N-1}$, re-add mean) and make a new plot with the principal components and the data set obtained from $\mathbf{Y}'$.

What have you just accomplished? How could this be useful? (*Hint:* Consider a scenario where some component of the data comes from noise. Also consider how much memory is required to represent the 'reduced' data set.)

For further study: Note that the only thing restricting this example to two dimensions is the plotting. Everything else in this problem applies cleanly in any number of dimensions. So go forth and analyze high-dimensional data sets. :)

There's data everywhere–here's just one example: `http://www.baseball-reference.com/`.