Homework 11
APPM 4600 Numerical Analysis, Fall 2025

Due date: Friday, November 21, before midnight, via Gradescope. **Instructor**: Prof. Becker
Revision date: 11/15/2025

Theme: Gaussian quadrature, linear algebra (specifically, LU and QR factorizations).

Instructions Collaboration with your fellow students is OK and in fact recommended, although direct copying is not allowed. The internet is allowed for basic tasks (e.g., looking up definitions on wikipedia) but it is not permissible to search for proofs or to *post* requests for help on forums such as `http://math.stackexchange.com/` or to look at solution manuals. Please write down the names of the students that you worked with. Please also follow our AI policy.

An arbitrary subset of these questions will be graded.

Turn in a PDF (either scanned handwritten work, or typed, or a combination of both) to **Gradescope**, using the link to Gradescope from our Canvas page. Gradescope recommends a few apps for scanning from your phone; see the Gradescope HW submission guide.

We will primarily grade your written work, and computer source code is *not* necessary (and you can use any language you want). You may include it at the end of your homework if you wish (sometimes the graders might look at it, but not always; it will be a bit easier to give partial credit if you include your code). For nicely exporting code to a PDF, see the APPM 4600 HW submission guide FAQ.

Problem 1: Gauss-Laguerre quadrature We'll estimate the integral of the Runge function over the whole real line

$$I = \int_{-\infty}^{\infty} \frac{1}{1+x^2}\,dx = 2\int_{0}^{\infty} \frac{1}{1+x^2}\,dx$$

using Gauss-Laguerre quadrature. Note that the Runge function is the probability density function (pdf) of the Cauchy distribution (up to a normalization constant), so the analytic solution is known; in fact, the antiderivative of $\frac{1}{1+x^2}$ is $\tan^{-1}(x)$, so we can use this to check our answer. Background on Gauss-Laguerre is at the end of this document—it's very similar to the Gauss-Legendre that we saw in class, and you can use libraries to find the weights and nodes for you.

a) Using a software library such as `lagpts.m` (Matlab) or `scipy.special.roots_laguerre` (Python) [see the background section at the end of this document], for various values of n, compute the nodes and weights for Gauss-Laguerre quadrature and use these to estimate I[1]. How small can you make the error (and for which value of n is this)? Can you make the error as close to machine precision as you like?

b) Now repeat the process but try the change of variables $x = e^t - 1$. How small can you make the error? Can you make the error as close to machine precision as you like? *Note:* you are welcome to simplify the integrand by hand (after the change of variables), as this may help with numerical issues.

c) Explain the results you observed in parts (a) and (b).

Problem 2: Least Squares Let $\mathbf{A} \in \mathbb{R}^{m\times n}$ with $m \geq n$, and let $\mathbf{Q} \in \mathbb{R}^{m\times n}$ and $\mathbf{R} \in \mathbb{R}^{n\times n}$ be its thin QR factoriztion, so $\mathbf{A} = \mathbf{QR}$ and $\mathbf{Q}$ has orthonormal columns and $\mathbf{R}$ is upper triangular. Let $\boldsymbol{b} \in \mathbb{R}^m$.

[1] *Hint*: $\int_0^\infty f(x)\,dx = \int_0^\infty f(x)e^x e^{-x}\,dx$. Also, you might want to check that your code is correct by integrating an easier function. I suggest integrating $\int_0^\infty \frac{1}{\sqrt{2\pi}} e^{-x^2/2}\,dx = 1/2$. You should get excellent results with fewer than 100 nodes.

a) Define $\boldsymbol{b}_\parallel = \mathbf{Q}\mathbf{Q}^\top \boldsymbol{b}$, the projection of $\boldsymbol{b}$ onto $\text{span}(\mathbf{Q}) = \text{span}(\mathbf{A})$. Define $\boldsymbol{b}_\perp = \boldsymbol{b} - \boldsymbol{b}_\parallel$, so that $\boldsymbol{b} = \boldsymbol{b}_\parallel + \boldsymbol{b}_\perp$. Show that $\boldsymbol{b}_\parallel$ and $\boldsymbol{b}_\perp$ are orthogonal.

b) Show that $\|\boldsymbol{b}\|_2^2 = \|\boldsymbol{b}_\parallel\|_2^2 + \|\boldsymbol{b}_\perp\|_2^2$.

c) For any $\boldsymbol{x} \in \mathbb{R}^n$, show that $\|\mathbf{Q}\boldsymbol{x}\|_2 = \|\boldsymbol{x}\|_2$.

d) For any $\boldsymbol{x} \in \mathbb{R}^n$, show that $\|\mathbf{A}\boldsymbol{x} - \boldsymbol{b}\|_2^2 = \|\mathbf{R}\boldsymbol{x} - \mathbf{Q}^\top \boldsymbol{b}\|_2^2 + \|\boldsymbol{b}_\perp\|_2^2$.

e) Assuming $\mathbf{A}$ is full rank, show that $\|\boldsymbol{b} - \mathbf{Q}\mathbf{Q}^\top \boldsymbol{b}\|_2 = \min_{\boldsymbol{x}\in\mathbb{R}^n} \|\mathbf{A}\boldsymbol{x} - \boldsymbol{b}\|_2$.

Problem 3: Gaussian elimination

a) Implement your own code (in Matlab or Python) to perform Gaussian elimination and then back-substitution to solve a square system of equations $A\mathbf{x} = \mathbf{b}$.[2] Include your code in your homework. For simplicity, you do *not* have to do any kind of pivoting (we'll optimistically assume that we never encounter a zero pivot).

b) Let $n = 10$ and define the $n \times n$ matrix $\mathbf{A} = [A_{ij}]$ as

$$A_{ij} = \cos\left(\frac{(i-1)(j-\frac{1}{2})\pi}{n}\right) \quad \forall i, j = 1, \ldots, n$$

so it should look like

```
1.0000    1.0000    1.0000    1.0000    1.0000    1.0000    1.0000    1.0000    1.0000    1.0000
0.9877    0.8910    0.7071    0.4540    0.1564   -0.1564   -0.4540   -0.7071   -0.8910   -0.9877
0.9511    0.5878    0.0000   -0.5878   -0.9511   -0.9511   -0.5878   -0.0000    0.5878    0.9511
0.8910    0.1564   -0.7071   -0.9877   -0.4540    0.4540    0.9877    0.7071   -0.1564   -0.8910
0.8090   -0.3090   -1.0000   -0.3090    0.8090    0.8090   -0.3090   -1.0000   -0.3090    0.8090
0.7071   -0.7071   -0.7071    0.7071    0.7071   -0.7071   -0.7071    0.7071    0.7071   -0.7071
0.5878   -0.9511   -0.0000    0.9511   -0.5878   -0.5878    0.9511    0.0000   -0.9511    0.5878
0.4540   -0.9877    0.7071    0.1564   -0.8910    0.8910   -0.1564   -0.7071    0.9877   -0.4540
0.3090   -0.8090    1.0000   -0.8090    0.3090    0.3090   -0.8090    1.0000   -0.8090    0.3090
0.1564   -0.4540    0.7071   -0.8910    0.9877   -0.9877    0.8910   -0.7071    0.4540   -0.1564
```

This is a version of the Discrete Cosine Transform (the DCT-II). Let $\boldsymbol{x} = (1, 2, \ldots, n)^T$, and define $\boldsymbol{b} = \mathbf{A}\boldsymbol{x}$. We know $\boldsymbol{x}$ and we'll keep this in mind for checking the error, but now pretend we don't know $\boldsymbol{x}$ and we'll solve for $\boldsymbol{x}$.

Apply your Gaussian elimination code to solve $\boldsymbol{b} = \mathbf{A}\boldsymbol{x}$ for $\boldsymbol{x}$, and report the augmented matrix (with 2 decimal places) after it has become upper triangular, and also the entrywise error in your final solution for $\boldsymbol{x}$.

c) Let $n = 1000$ and $\mathbf{A}$ be a $n \times n$ matrix with entries from the standard normal distribution[3], and again define $\boldsymbol{x} = (1, 2, \ldots, n)^\top$ and define $\boldsymbol{b} = \mathbf{A}\boldsymbol{x}$. How long does it take your Gaussian elimination code to solve this system? And what's the error (in terms of Euclidean norm, `norm` in Matlab and `numpy.linalg.norm` in Python)?

Hint: for timings, in Matlab you can use `tic` and `toc`; in Python, you can import the `time` module and use `time.perf_counter()`, or

```
from timeit import time
%time [your command here]
```

which is an IPython line magic (so works in IPython and Jupyter). Fun fact: IPython was created as a side project by CU Boulder student Fernando Pérez in 2001.

[2] For Python users, be aware that `numpy` distinguishes between a pure vector of length n (which has "shape" `(n,)`) and a row vector of length n (shape `(1,n)`) and a column vector of length n (shape `(n,1)`). Often a pure vector is treated like a column vector, but not always. You can convert between the different shapes using things like `numpy.reshape`, `numpy.atleast_2d`, and `.T` (to transpose things). Other helpful `numpy` things may include `numpy.hstack` or `numpy.concatenate`, and `@` for matrix multiplication.

[3] in Matlab, `randn(n)`; and in Python, `from numpy.random import default_rng` then `rng = default_rng()` then `A = rng.standard_normal((n,n))`

d) Solve the same system of equation as in (c) but use Matlab's/Python's default solvers, and report the time. For Matlab, this is "backslash" `A\b` aka `mldivide`, and in Python, this is `scipy.linalg.solve`. Also report the error, and comment on the differences between (d) and (c).

Problem 4: LU factorization verse the inverse We're interested in solving the square $n \times n$ system of equations $\mathbf{A}\boldsymbol{x} = \boldsymbol{b}$ using standard techniques; we'll do some programming, but it will be high-level programming. *See the end of this PDF for some background on pivoted LU factorization*

a) Write high-level code that solves $\mathbf{A}\boldsymbol{x} = \boldsymbol{b}$ using these three methods:

i. (Highest level) Use "backslash" `A\b` aka `mldivide` (Matlab) or `scipy.linalg.solve` (Python)

ii. (LU with pivoting) Perform an LU factorization and then solve via backsubstitution, relying on Matlab or Python libraries for everything. *In particular, for Matlab, use either `[L,U,P] = lu(A)` or `[L,U,p] = lu(A,'vector')` (slightly faster), and figure out how to use the permutation matrix; then you can call something like `U\y` and Matlab will know that `U` is upper triangular and use backsubstitution. For Python, use either `scipy.linalg.lu(A,permute_l=False)` followed by some `scipy.linalg.solve_triangular` (for the forward/backward substitution; sometimes with the flag `lower=True`) and applying the inverse permutation to $\boldsymbol{b}$; or use `scipy.linalg.lu_factor(A)` followed by `scipy.linalg.lu_solve`. Note that neither Matlab nor `scipy` has an LU factorization function that doesn't pivot; if it doesn't appear to pivot, it means that it really has pivoted but just incorporated the permutation into* $\mathbf{L}$.

iii. (Inverse) Solve via an explicit inverse of $\mathbf{A}$, using `inv` (Matlab) or `scipy.linalg.inv` (Python).

Show your code (each method might only be one or two lines of code). *It's a good idea to check your code on a small problem with a known answer to make sure all these methods work.*

b) Record the timing and error for each of these methods for a range of sizes n between 1000 and 5000, and plot both the error and the timing; is the timing $O(n^2)$ or $O(n^3)$ or $O(n^4)$? For a given n, generate $A, \boldsymbol{x}$ and $\boldsymbol{b}$ as we did in Problem 2c/d. Comment on your findings.

Hint: for timings, in Matlab you can use `tic` and `toc`; in Python, you can import the `time` module and use `time.perf_counter()`.

Background on using the LU factorization

To recall the (pivoted) `LU`-factorization, we write

$$\mathbf{A}\boldsymbol{x} = \boldsymbol{b}, \quad \mathbf{A} = \underbrace{\mathbf{PLU}}_{\text{pivoted LU factorization}}, \quad \text{so} \quad \mathbf{PL}\underbrace{\mathbf{U}\boldsymbol{x}}_{\boldsymbol{y}} = \boldsymbol{b}$$

where $\mathbf{P}$ is a permutation matrix, either given as an explicit matrix (and it is orthogonal, so $\mathbf{P}^\top = \mathbf{P}^{-1}$ hence $\mathbf{LU}\boldsymbol{x} = \mathbf{P}^\top\boldsymbol{b}$) or as a permutation vector `p` of the indices so we can apply $\mathbf{P}^\top\boldsymbol{b}$ as `b(p)` (Matlab) or `b[p]` (Python). Then we first solve $\mathbf{L}\boldsymbol{y} = \mathbf{P}^\top\boldsymbol{b}$ using *forward-substitution*, and then solve $\mathbf{U}\boldsymbol{x} = \boldsymbol{y}$ using *backward-substitution*. Be careful: Matlab and Python have different conventions for returning $\mathbf{P}$ — Matlab actually returns $\mathbf{P}^{-1}$ while Python returns $\mathbf{P}$.

Background on Gauss-Laguerre quadrature

The Laguerre polynomials $\{L_n\}_{n=1}^\infty$ are defined, up to a scaling constant, such that L_n is a degree n polynomial and that

$$\int_0^\infty L_n(x)L_m(x)e^{-x}\,dx = 0 \quad \forall n, m \in \mathbb{N}, n \neq m.$$

In other words, the polynomials are *orthogonal* on the domain $[0,\infty)$ with the weight function $w(x) = e^{-x}$ (we cannot have $w(x) = 1$ because the integral over $[0,\infty)$ of any polynomials, other than the 0 polynomial, is infinite). To resolve the scaling ambiguity, some scaling convention is used, such as requiring them to be monic (i.e., leading coefficient is 1), or that $\int_0^\infty L_n(x)e^{-x}\,dx = 1$; the scaling does not concern us since we'll only care about their roots. These polynomials arise in many areas of math, and are the solutions to the 2nd order linear ODE $xy'' + (1-x)y' + ny = 0$, and arise as part of the solution to the Schrödinger equation of the one-electron atom in quantum mechanics.

We'll use the fact that L_n has n simple real roots in $(0,\infty)$. We use these as the nodes x_i. Our goal is to approximate an integral of the form

$$I(f) = \int_0^\infty f(x)e^{-x}\,dx \tag{1}$$

which we will do by interpolating f with a degree $n-1$ polynomial on n nodes; this polynomial p is unique, and is the Lagrange interpolating polynomial we've seen before. Then we approximate I by $I_n(f) = \int_0^\infty p(x)e^{-x}$; this integral I_n is tractable since we can write $p(x)$ as a weighted sum of monomials, and for any monomial x^k, we can determine $\int_0^\infty x^k e^{-x}\,dx$ by doing integration by parts k times (though there are shortcuts/formulas; we don't actually do integration by parts in practice when numerically evaluating this). It follows that if f itself is a degree $n-1$ or less polynomial, then f is its own interpolatory polynomial of degree $n-1$ or less, and so $I(f) = I_n(f)$. Now suppose f is a polynomial of degree between n and $2n-1$. Divide f by the Laguerre polynomial L_n, so we can write $f(x) = Q(x)L_n(x) + R(x)$ using polynomial long division, where R has degree less than L_n (so strictly less than n) and Q has degree between 0 and $n-1$. Then

$$\begin{aligned}
\int_0^\infty f(x)e^{-x}\,dx &= \underbrace{\int_0^\infty Q(x)L_n(x)e^{-x}\,dx}_{=0 \text{ by orthogonality}} + \int_0^\infty R(x)e^{-x}\,dx \\
&= 0 + I(R) \\
&= I_n(R) \text{ since } R \text{ has degree } n-1 \text{ or less} \\
&= I_n(f) \text{ since } f(x_i) = Q(x_i)\cdot \underbrace{L_n(x_i)}_{=0} + R(x_i) = R(x_i)
\end{aligned}$$

Matlab doesn't have a builtin routine to compute the weights and zeros for Gauss-Laguerre quadrature, but the well-respected `Chebfun` package does: see `lagpts.m`, which relies on `gammaratio.m`. You can download these directly from Matlab using the following commands:

```
websave('lagpts.m','https://github.com/chebfun/chebfun/raw/master/lagpts.m')
websave('gammaratio.m','https://github.com/chebfun/chebfun/raw/master/gammaratio
    .m');
```

Python's `scipy` package has routines for the weights and zeros. Use `scipy.special.roots_laguerre` or `numpy.polynomial.laguerre.laggauss` (which looks like it may not be as tested as the `scipy` version).