Homework 7
APPM/MATH 4650 Fall '20 Numerical Analysis

Due date: Saturday, October 24, before midnight, via Gradescope. **Instructor**: Prof. Becker
Theme: Gaussian quadrature and other numerical integration

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.

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 except for when we *explicitly* ask for it (and you can use any language you want). If not specifically requested as part of a problem, 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).

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.

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. 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: High-dimensional integration If z is a continuous random variable with probability density function (pdf) given by $p(z)$, then we define its *mean* or *expected value* as

$$\mathbb{E}[z] = \int_{-\infty}^{\infty} z \cdot p(z)\,dz.$$

That is, the expectation value $\mathbb{E}$ is an integral[1], and expectation values arise a lot: not just for means, but also for variances (since $\mathbb{V}\text{ar}[z] = \mathbb{E}[z^2] - \mathbb{E}[z]^2$) and other quantities. If

[1] for discrete random variables, it is a sum not an integral, though you can interpret this sum as a special kind of integral with respect to a different *measure*; e.g., using dirac delta functions.

$z = f(\boldsymbol{x})$ is a function of a *multivariate* random variable $\boldsymbol{x} \in \mathbb{R}^d$, then the expected value is a multidimensional integral

$$\mathbb{E}[f(\boldsymbol{x})] = \int_{\mathbb{R}^d} f(\boldsymbol{x}) \cdot p(\boldsymbol{x}) d\boldsymbol{x}.$$

Because it is common to have multivariate random variables of very large dimension, this means multidimensional integration frequently arises in probability and statistics.

For this problem, use the $d \times d$ symmetric matrix A known as the "Hilbert matrix" defined as $A_{i,j} = 1/(i+j-1)$ for $i, j = 1, \ldots, d$. You can quickly create this in Matlab using `hilb(d)` or in Python using `scipy.linalg.hilbert(d)`. Using this matrix, for $\boldsymbol{x} \in \mathbb{R}^d$, we define

$$f(\boldsymbol{x}) = \boldsymbol{x}^\top A \boldsymbol{x} \tag{1}$$

and choose $\boldsymbol{x} \sim \mathcal{N}(0, I_{d\times d})$ meaning that $\boldsymbol{x}$ is a d-dimensional standard Gaussian random variable, with independent components[2]. That is, if we write the components of $\boldsymbol{x}$ as $\boldsymbol{x} = (x_1, \ldots, x_d)$, then each x_i is a standard normal random variable (mean 0, variance 1) and x_i and x_j are independent if $i \neq j$. The pdf of a standard normal random variable is $p_{\mathrm{N}}(x) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2}$. Thus the pdf of $\boldsymbol{x}$ can be written as

$$p(\boldsymbol{x}) = \prod_{i=1}^{d} p_{\mathrm{N}}(x_i) = \prod_{i=1}^{d} \frac{1}{\sqrt{2\pi}} e^{-x_i^2/2} = \frac{1}{(2\pi)^{d/2}} \exp\left(\sum_{i=1}^{d} -x_i^2/2\right).$$

a) Write a code that numerically integrates an arbitrary function in dimension d, for d at least up to 4. Your code can rely on other packages for 1D integration (e.g., in Matlab[3], either `quad` or `integral` — although `quad` is being deprecated in favor of `integral`, I find `quad` is better about being fast if we ask it for low precision — and `scipy.integrate.quad` in Python[4]). However, you may not rely on other packages for 2D, 3D, or other high-dimensional integration (so do not use `integral2` and `integral3` in Matlab, and do not use `scipy.integrate`'s `dblquad`, `tplquad` or `nquad`... except for checking your answers if you want). **Include your code in your homework writeup**. [5]

b) Approximate $\mathbb{E}[f(\boldsymbol{x})]$ using f as defined in Eq. (1) and for $\boldsymbol{x} \sim (0, I_{d\times d})$ for $d = 1, 2, 3, 4$. Although this should be an integral over all of $\mathbb{R}^d$, you may integrate over just the hypercube $[-5, 5]^d$ since $p(\boldsymbol{x})$ is negligible outside this region. To help debug the code, the true integral (over all of $\mathbb{R}^d$) is 1 for $d = 1$, $1.3\overline{3}$ for $d = 2$ and $1.53\overline{3}$ for $d = 3$.

   i. What is the value of the integral for $d = 4$? You should have at least 2 correct digits.

   ii. Report the tolerance settings of your 1D integration routine, and then report how long the integral took for each of the dimensions $d = 1, 2, 3, 4$. You can report for larger dimensions if you were able to get the code to work.

c) Sometimes we use quadrature methods to evaluate expectations, but we can also use expectations to evaluate integrals for us: this is called Monte Carlo estimation. Repeat your estimate of $\mathbb{E}[f(\boldsymbol{x})]$ for $d = 1, 2, 3, 4$ by computing the *sample mean* of $f(\boldsymbol{x})$ over n different independent realizations of $\boldsymbol{x}$. You can draw a realization of $\boldsymbol{x}$ using `randn(d,1)` in Matlab, or `numpy.random.randn(d)` in Python[6]. Run $n = 10^6$ realizations to form your Monte Carlo estimator, and report both your *runtime* (in seconds) and *error*, for $d = 4$, $d = 12$ and $d = 100$. For $d = 4$, you can use the answer you found via quadrature as the "true answer" for calculating the error; for $d = 12$, the true value is `2.224352838648`, and for $d = 100$ it is `3.284342189302`.

[2] With this setup, finding $\mathbb{E}[\boldsymbol{x}^\top A \boldsymbol{x}]$ is related to a so-called *Hutchinson* estimator for A.

[3] in Matlab, using `arrayfun` is often convenient, since both `quad` and `integral` will ask to evaluate your function at multiple points simultaneously

[4] in Python, using `np.vectorize` is often convenient, since `quad` will ask to evaluate your function at multiple points simultaneously

[5] You can hardcode f to take 1, 2, 3 or 4 inputs, but if you wanted to make a code that works in any dimensions [not required], you may find it useful to use `varargin` in Matlab or `*args` in Python.

[6] though the recommended way is `from numpy.random import default_rng` then `rng = default_rng()` then `rng.standard_normal(d)`.

d) Make some informed comments about the difference between the quadrature and Monte Carlo methods

Optional problem (Not graded) Estimate

$$I = \lim_{\epsilon \to 0} \int_{\epsilon}^{1} x^{-1} \cos(x^{-1} \ln(x))\, dx$$

as accurately as possible using a numerical scheme, and describe your method. In previous years, students received 1 point of credit for every accurate digit of their solution.

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

$$I(f) = \int_0^{\infty} f(x) e^{-x}\, dx \tag{2}$$

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).