Spring 2021 Math 128B Reference Solutions

Hongli Zhao

May 31st, 2021

Abstract

Math 128B: Numerical Analysis at UC Berkeley is the second course in the Math 128 series of undergraduate-level numerical analysis. Topics of the second course include matrix algorithms and analysis, function approximations, introduction to numerical ODEs (initial-boundary value problems) and numerical PDEs, following the textbook *Numerical Analysis, 10th Edition* by Burden & Faires, with slight edits to problem statements. The Spring 2021 offering of the course was given by Prof. Olga Holtz and graduate teaching assistant Oliver Edtmair, managed and supported via online platform Piazza and Gradescope by grader Hongli Zhao. Major programming tools used are MATLAB, Julia and Python.

This document includes all published official solutions written by Hongli Zhao and advised by Prof. Olga Holtz, with revisions and suggestions given by Oliver Edtmair. All proofs and numerical algorithms may not be included, and is available upon request. Comments, errata and suggestions are welcome, please send an email to `mailto:honglizhaobob@berkeley.edu`. This is an unofficial reference and is not a publication.

1 Homework 1

1.1 Norm Verification

(a) Verify that:

$$\|x\|_3 := \left(\sum_{j=1}^{n} |x_j|^3 \right)^{1/3}, x \in \mathbb{C}^n$$

defines a norm.

(b) Find pair of constants $0 < c < C$ such that:

$$c\|x\|_1 \leq \|x\|_3 \leq C\|x\|_1, \forall x \in \mathbb{C}^n$$

as proof that $\|\cdot\|_3$ is equivalent to $\|\cdot\|_1$.

1.1.1 a

To show a function is a norm, we need to show 3 things (here $X = \mathbb{C}^n$).

1. $\|x + y\| \leq \|x\| + \|y\|$ for all $x, y \in X$.
2. $\|tx\| = |t|\|x\|$ for all $x \in X$ and $\lambda \in \mathbb{C}$.
3. $\|x\| \geq 0$ and $= 0$ if and only if $x = 0$.

More generally, we can show that $\|x\|_p$ defines a norm for $p \in [1, \infty)$.

By definition:

$$\|x\|_p = \sqrt[p]{\sum_{i=1}^{n} |x_i|^p} \geq 0$$

since we have absolute value on each x_i.

If $x = 0$, $\|0\|_p = 0$. Now suppose $\|x\|_p = 0$, we have:

$$\sum_{i=1}^{n} |x_i|^p = 0$$

since $|x_i|^p \geq 0$ for all i, this implies $|x_i| = 0$ for all i, which means $x_i = 0$ for all i. Then $x = 0$.

Take $t \in \mathbb{C}$, consider:

$$\|tx\|_p = \left(\sum_1^n |tx_i|^p\right)^{1/p} = \left(\sum_1^n |t|^p |x_i|^p\right)^{1/p} = \left(|t|^p \sum_1^n |x_i|^p\right)^{1/p}$$

$$= (|t|^p)^{1/p} \cdot \left(\sum_1^n |x_i|^p\right)^{1/p} = |t| \cdot \|x\|_p$$

Triangle inequality follows from Minkowski inequality.

1.1.2 b

There is a functional analysis fact that in a finite dimensional vector space (such as $\mathbb{C}^n$ here), all norms are equivalent (if interested, Chapter 5.1 Exercise 6 from *Real Analysis: Modern Techniques and Their Applications* by Gerald Folland (or other standard functional analysis textbooks) provides a statement).

Here we only apply this fact to our specific case $p = 3$.

First:

$$\|x\|_3^3 = \sum_1^n |x_i|^3$$

on the other hand:

$$\|x\|_1^3 = \left(\sum_1^n |x_i|\right)^3 = \sum_1^n |x_i|^3 + 3\sum_{i<j} |x_i|^2|x_j| + 6\sum_{i<j<k} |a_i||a_j||a_k|$$

which is larger than:

$$\geq \sum_1^n |x_i|^3 = \|x\|_3^3$$

therefore taking cube root on both sides, we showed:

$$\|x\|_3 \leq \|x\|_1$$

with constant 1.

For the other direction, consider:

$$\|x\|_\infty^3 = \max_i |x_i|^3 \leq \sum_1^n |x_i|^3 = \|x\|_3^3$$

on the other hand we have showed before (for all i, each $|x_i|$ is less than the maximum element, then repeating the max n times is going to be larger than adding all $|x_i|$ together):

$$\|x\|_1 \leq n\|x\|_\infty$$

or:

$$\|x\|_1^3 \leq n^3\|x\|_\infty^3$$

combining the previous inequalities:

$$\frac{1}{n^3}\|x\|_1^3 \leq \|x\|_\infty^3 \leq \|x\|_3^3$$

we take cube root, and get:

$$\frac{1}{n}\|x\|_1 \leq \|x\|_3$$

here the constant is $1/n$.

As above, we showed that $\|\cdot\|_1$ is equivalent to $\|\cdot\|_3$.

1.2 Plotting Unit Sphere for Different Norms

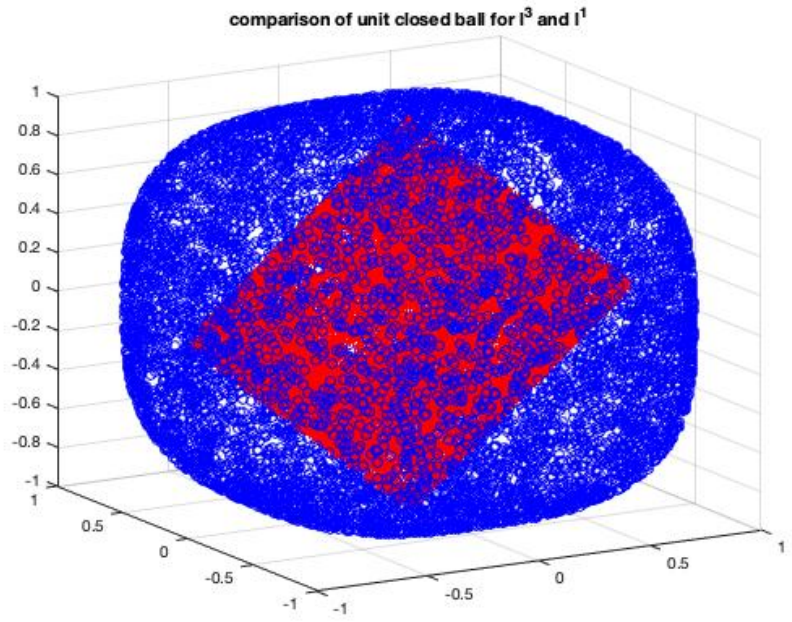
Use MATLAB to plot the unit spheres for norms $\|\cdot\|_1$ and $\|\cdot\|_3$ in $\mathbb{R}^3$.

1.2.1

There are many ways to make the plot. The MATLAB code for this particular plot is below:

```
% plotting unit ball with respect to different norms

% sample from a Gaussian
X = randn(10000, 3);

% l1 norm coeffs
l1_coef = sum(abs(X).^1, 2).^(1/1);
% l3 norm coeffs
l3_coef = sum(abs(X).^3, 2).^(1/3);
% normalize data by p-norm
X_l1_data = (bsxfun(@times, X, 1./l1_coef));
X_l3_data = (bsxfun(@times, X, 1./l3_coef));


figure;
scatter3(X_l1_data(:, 1), X_l1_data(:, 2), X_l1_data(:, 3),...
    '*', "red");
hold on;
scatter3(X_l3_data(:, 1), X_l3_data(:, 2), X_l3_data(:, 3),...
    'o', "blue"); title('comparison of unit closed ball for l^3 and l^1');
```

which yields the result:



1.3 Calculating 2-norm and ∞-norm

For each of the matrices, find ∞-norm and 2-norm.

(1)

$$\begin{pmatrix} 1 & 2 \\ 3 & 2 \end{pmatrix}$$

(2)

$$\begin{pmatrix} -2 & 3 \\ 3 & -2 \end{pmatrix}$$

(3)

$$\begin{pmatrix} 1 & 1 & 1 & 1 \\ 0 & 2 & 2 & 3 \\ 0 & 0 & 3 & 2 \\ 0 & 0 & 0 & 4 \end{pmatrix}$$

The matrix operator 2-norm is the square root of largest eigenvalue of $A^T A$ (for A real), and the l_∞-norm is given by the maximum absolute row

sum.

1.3.1

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 2 \end{pmatrix}, A^T A = \begin{pmatrix} 10 & 8 \\ 8 & 8 \end{pmatrix}$$

$$\|A\|_2 = \sqrt{\lambda_{max}(A^T A)} = \sqrt{9 + \sqrt{65}}$$

and:

$$\|A\|_\infty = \max_{1 \le i \le 2} \sum_{j=1}^{2} |a_{ij}| = 5$$

1.3.2

$$A = \begin{pmatrix} -2 & 3 \\ 3 & -2 \end{pmatrix}, A^T A = \begin{pmatrix} 13 & -12 \\ -12 & 13 \end{pmatrix}$$

$$\|A\|_2 = \sqrt{\lambda_{max}(A^T A)} = 5$$

$$\|A\|_\infty = 5$$

1.3.3

$$A = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 0 & 2 & 2 & 3 \\ 0 & 0 & 3 & 2 \\ 0 & 0 & 0 & 4 \end{pmatrix}, A^T A = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 5 & 5 & 7 \\ 1 & 5 & 14 & 13 \\ 1 & 7 & 13 & 30 \end{pmatrix}$$

$$\|A\|_2 \approx 6.2813$$

$$\|A\|_\infty = 7$$

1.4 2-norm of Symmetric Matrix

Show that if a matrix A is symmetric, then its 2-norm is equal to its spectral radius.

1.4.1

A is symmetric, it admits the following decomposition:

$$A = PDP^{-1}$$

where P is orthogonal, thus:

$$A^T A = PDP^{-1}PDP^{-1} = PD^2P^{-1}$$

then the eigenvalues of $A^T A$ are simply squares of the eigenvalues of A. Then:

$$\|A\|_2 = \sqrt{\lambda_{max}(A^T A)} = \sqrt{\lambda_{max}(A)^2} = \lambda_{max}(A) = \rho(A)$$

as desired.

2 Homework 2

2.1 MATLAB implementation of Jacobi Method

Please refer to Algorithm 7.1 and 7.2 for exact implementations. Regarding implementations:

1. Many students used for loops to find L, U, in MATLAB, we can consider using:

```
triu(A);

tril(A);
```

to get upper, lower triangular parts of A.

2. In practical implementations, we should avoid calling explicit inverse of matrices; rather, we should use a linear solver or backslash. Here is actually an interesting discussion that explores why this is the case:

```
https://arxiv.org/abs/1201.6035
```

2.2 MATLAB implementation of Gauss-Seidel Method

Gauss-Seidel usually requires fewer iterations to reach similar level of accuracy as Jacobi as it (GS) tries to incorporate new information at each equation in the linear system. If at the n-th equation we already have solutions for the previous $(n-1)$, why not use them in solving the current one?

Furthermore, the condition for convergence should be fulfilled for diagonally dominant matrices with any initial condition.

2.3 Convergence of Matrix Powers

Let A be a square matrix and let $\|\cdot\|$ be any matrix norm (not necessarily natural/induced). Prove that $\lim_{k\to\infty} \left\|A^k\right\| = 0$ if $\rho(A) < 1$.

2.3.1

Suppose $\rho(A) < 1$, we have $\lim_{n\to\infty} \lambda_i^m = 0$ for all i. Any matrix A admits a Jordan decomposition in the following form:

$$A = PJP^{-1}$$

where P is invertible, and J is a block-diagonal matrix with $k \leq n$ blocks. And each block J_i for $i \in [1, k]$ have superdiagonal be all 1's, or:

$$J_i = \begin{pmatrix} \lambda_i & 1 & & \\ & \lambda_i & \ddots & \\ & & \ddots & 1 \\ & & & \lambda_i \end{pmatrix}$$

and we can express:

$$J = \begin{pmatrix} J_1 & & & & \\ & J_2 & & & \\ & & \ddots & & \\ & & & J_{k-1} & \\ & & & & J_k \end{pmatrix}$$

Then we have:

$$A^m = (PJP^{-1}) \cdot (PJP^{-1}) \cdots (PJP^{-1}) = PJ^mP^{-1}$$

and since J is block-diagonal:

$$J^m = \begin{pmatrix} J_1^m & & & & \\ & J_2^m & & & \\ & & \ddots & & \\ & & & J_{k-1}^m & \\ & & & & J_k^m \end{pmatrix}$$

then we can investigate what each J_i^m behavior is.

For a Jordan matrix J with eigenvalue λ, we can write it as:

$$J = (\lambda I + S)$$

where S is called a "shift" matrix, in the sense that it shifts all values in a vector up by 1 place, zeroing out the last element. More explicitly suppose we are in $\mathbb{R}^d$:

$$S = \begin{pmatrix} 0 & 1 & & \\ & 0 & \ddots & \\ & & \ddots & 1 \\ & & & 0 \end{pmatrix}$$

and the shifting is in the sense that:

$$S \cdot [x_1, x_2, \cdots, x_{d-1}, x_d]^T = [x_2, x_3, \cdots, x_d, 0]^T$$

Furthermore, S and λI commutes, because I is the identity. Then we can write the powers of J as a binomial expansion:

$$J^m = (\lambda I + S)^m = \sum_0^m \binom{m}{r} \lambda^{m-r} S^r$$

By the formulation of S, we know that it is nilpotent with degree n (repeating S by n times on a vector, we get the 0 vector). Eigenvalues of nilpotent matrices can only be 0, therefore S^r will not cause the norm to grow.

Furthermore, S^d becomes 0, and d as the matrix dimension is finite. Then as $m \to \infty$, we can safely write:

$$\lim_{m\to\infty} J^m = \sum_0^{d-1} \lambda^{m-r} S^r$$

because all subsequent terms $S^d, S^{d+1}, \cdots$ will be wiped out by the fact that $S^d = 0$. Therefore this is a finite matrix sum with $\lambda^{m-r} \to 0$ (r can only grow as large as $d-1$). Thus we conclude:

$$\lim_{m\to\infty} J^m = 0$$

and this is true for all Jordan matrices with $|\lambda| < 1$. Therefore $J^m \to 0$ as $m \to \infty$.

Now we can use what we showed above and conclude:

$$\lim_{m\to\infty} \|A^m\| = \lim_{m\to\infty} \left\|PJ^mP^{-1}\right\| \leq \lim_{m\to\infty} \|P\|\left\|P^{-1}\right\|\|J^m\| = \kappa(P) \lim_{m\to\infty} \|J^m\| = 0$$

here we use the fact that $\kappa(P)$ is finite unless P is singular (which by Jordan normal form, it is not), hence the last equality to 0.

3 Homework 3

3.1 Find the first two iterations of the SOR method

Find the first two iterations of the SOR method with $\omega = 1.2$ for the following linear system:

$$10x_1 - x_2 = 9$$
$$-x_1 + 10x_2 - 2x_3 = 7$$
$$-2x_2 + 10x_3 = 6$$

check convergence of the corresponding T_ω matrix.

3.1.1

The linear system is:

$$\begin{pmatrix} 10 & -1 & 0 \\ -1 & 10 & -2 \\ 0 & -2 & 10 \end{pmatrix} x = \begin{pmatrix} 9 \\ 7 \\ 6 \end{pmatrix}$$

or $Ax = b$.

In SOR, the matrix A permits the following decomposition:

$$A = D + L + U$$

where:

$$D = \begin{pmatrix} 10 & 0 & 0 \\ 0 & 10 & 0 \\ 0 & 0 & 10 \end{pmatrix}, U = \begin{pmatrix} 0 & -1 & 0 \\ 0 & 0 & -2 \\ 0 & 0 & 0 \end{pmatrix}, L = \begin{pmatrix} 0 & 0 & 0 \\ -1 & 0 & 0 \\ 0 & -2 & 0 \end{pmatrix}$$

The exact solution for this linear system can be calculated by hand or a calculator:

$$x = \begin{pmatrix} \frac{473}{475} \\ \frac{91}{95} \\ \frac{375}{475} \end{pmatrix} \approx [0.99578947, 0.95789474, 0.79157895]^T$$

See textbook equation (7.18). The matrix T_w is given by:

$$T_w = (D - wL)^{-1}\big[(1 - w)D + wU\big]$$

letting $w = 1.2$ will yield approximately:

$$T_w = \begin{pmatrix} -0.2 & -0.12 & 0 \\ 0.024 & -0.1856 & -0.24 \\ -0.00576 & 0.044544 & -0.1424 \end{pmatrix}$$

In line with B&F Definition 7.16, it is enough to check the spectral radius of T_w, namely:

$$\rho(T_w) = \max |\lambda(T_w)| \leq \|T_w\| \approx 0.4115$$

(one can also choose to compute the eigenvalues, or check positive-definiteness. Therefore the matrix is convergent, refer to Theorem 7.17 for detailed properties.

Now we use T_w to perform two iterations of SOR, exact details are skipped, follow:

$$x^{(k)} = T_w x^{(k-1)} + c_w$$

where $c_w = w(D - wL)^{-1}b$.

$$c_w \approx \begin{pmatrix} 1.08 \\ 0.7104 \\ 0.549504 \end{pmatrix}$$

Pick initial guess $x^{(0)} = [1, 1, 1]^T$ (it is okay to choose other initial guesses, correctness was graded based on values for T_w, c_w). Two steps of SOR yields:

$$x^{(1)} = T_w x^{(0)} + c_w = \begin{pmatrix} 0.76 \\ 0.3088 \\ 0.445888 \end{pmatrix}$$

$$x^{(2)} = T_w x^{(1)} + c_w = \begin{pmatrix} 0.890944 \\ 0.5643136 \\ 0.49538714 \end{pmatrix}$$

3.2 Inequality

Let $\kappa(A)$ denote the condition number of square matrix A. Show that any singular matrix B satisfies the following inequality:

$$\kappa(A)^{-1} \leq \frac{\|A-B\|}{\|A\|}$$

3.2.1

B is singular, therefore $ker(B) \neq \emptyset$, pick any $x \in ker(B)$, assume $\|x\| = 1$ (if not, normalize by $x' = x/\|x\|$). Then we observe:

$$(A-B)x = Ax - Bx = Ax$$

Then we have:

$$\|Ax\| = \|(A-B)x\| \leq \|A-B\|\|x\| = \|A-B\|$$

As we learned in class:

$$\kappa(A) = \|A\| \cdot \left\|A^{-1}\right\|$$

It is enough to show $\|A-B\| \cdot \|A^{-1}\| \geq 1$ (rearrange the original inequality).

Consider the matrix-vector multiply:

$$A^{-1}(A-B)x = Ix - A^{-1}Bx = x$$

recall $Bx = 0$. Then we have:

$$1 = \|x\| = \left\|A^{-1}(A-B)x\right\| \leq \left\|A^{-1}\right\|\|A-B\|\|x\| = \left\|A^{-1}\right\|\|A-B\|$$

as desired.

3.3 Gaussian elimination with three-digit rounding

Use Gaussian elimination and three-digit rounding arithmetic to find an approximate solution to the following linear system:

$$0.03x_1 + 58.9x_2 = 59.2$$

$$5.31x_1 - 6.1x_2 = 47.0$$

Then use one iteration of iterative refinement. Compare both approximations to the exact solution.

3.3.1

We have the system:

$$\begin{pmatrix} 0.03 & 58.9 \\ 5.31 & -6.1 \end{pmatrix} x = \begin{pmatrix} 59.2 \\ 47.0 \end{pmatrix}$$

the system has an exact solution:

$$x = \begin{pmatrix} 10.0 \\ 1.0 \end{pmatrix}$$

the key is to understand how 3-digit rounding may give poor accuracy. Let r_1, r_2 denote row 1 and row 2 of the matrix, the Gaussian Elimination step is to add $-\frac{5.31}{0.03} r_1$ to r_2. Here we have:

$$-\frac{5.31}{0.03} \approx -177$$

since we only have 3 digits. Doing this calculation by hand:

$$-\frac{5.31}{0.03} r_1 = -5.31x_1 - 10400x_2 = -10300$$

Now add this to r_2 we have:

$$-10400x_2 = -10300$$

then solve, and round by 3 digits:

$$\tilde{x}_2 = 0.990$$

This causes problems, plug $\tilde{x}_2$ back to r_1 and solve for x_1 we have:

$$\tilde{x}_1 = \frac{59.2 - 58.9 \cdot 0.990}{0.03} \approx \frac{0.900}{0.03} = 30.0$$

The reason why this happens is because A is ill-conditioned (Definition 7.28).

$$\kappa(A)_\infty \approx 12.24$$

Now keeping 3-digit rounding, we use one step of iterative refinement. We had:

$$x^{(1)} = \tilde{x} = \begin{pmatrix} 30.0 \\ 0.990 \end{pmatrix}$$

which yields residual:

$$r^{(1)} = b - Ax^{(1)} = \begin{pmatrix} 59.2 \\ 47.0 \end{pmatrix} - \begin{pmatrix} 0.03 & 58.9 \\ 5.31 & -6.1 \end{pmatrix} \begin{pmatrix} 30.0 \\ 0.990 \end{pmatrix}$$

in more detail:

$$0.03 \times 30.0 + 58.9 \times 0.990 \approx 0.900 + 58.3 \approx 59.2$$

$$5.31 \times 30.0 - 6.1 \times 0.990 \approx 159 - 6.04 \approx 153$$

$$r^{(1)} \approx \begin{pmatrix} 59.2 \\ 47.0 \end{pmatrix} - \begin{pmatrix} 59.2 \\ 153 \end{pmatrix} = \begin{pmatrix} 0 \\ -106 \end{pmatrix}$$

Now solve for the correction:

$$A\delta^{(1)} = r^{(1)}$$

yielding:

$$\delta^{(1)} \approx \begin{pmatrix} -20.0 \\ 0.0102 \end{pmatrix}$$

Refine our solution:

$$x^{(2)} = x^{(1)} + \delta^{(1)} = \begin{pmatrix} 30.0 \\ 0.990 \end{pmatrix} + \begin{pmatrix} -20.0 \\ 0.0102 \end{pmatrix} = \begin{pmatrix} 10.0 \\ 1.00 \end{pmatrix}$$

142 which is much better!

3.4 Perturbation Estimate

Let:

$$A = \begin{pmatrix} 1 & 2 \\ 1.00001 & 2 \end{pmatrix}, b = \begin{pmatrix} 3 \\ 3.00001 \end{pmatrix}$$

The problem $Ax = b$ has exact solution $x = [1, 1]^T$. Use seven-digit rounding arithmetic to solve the perturbed system with:

$$\tilde{A} = \begin{pmatrix} 1 & 2 \\ 1.000011 & 2 \end{pmatrix}, \tilde{b} = \begin{pmatrix} 3.00001 \\ 3.00003 \end{pmatrix}$$

and compare the actual error to the Perturbation Estimate.

3.4.1

We have exact system:

$$A = \begin{pmatrix} 1 & 2 \\ 1.00001 & 2 \end{pmatrix}, b = \begin{pmatrix} 3 \\ 3.00001 \end{pmatrix}$$

and it is perturbed with (follow the notation from B&F Theorem 7.29):

$$\delta A = \begin{pmatrix} 0 & 0 \\ 0.000001 & 0 \end{pmatrix}, \delta b = \begin{pmatrix} 0.00001 \\ 0.00002 \end{pmatrix}, \tilde{A} = A + \delta A, \tilde{b} = b + \delta b$$

Then with 7-digit rounding:

$$\tilde{x} \approx \begin{pmatrix} 1.8181820 \\ 0.5909141 \end{pmatrix}$$

Refer to Formula (7.25) for the perturbation bound (with respect to inf norm):

$$LHS = \frac{\|x - \tilde{x}\|}{\|x\|} \approx 0.82$$

$$RHS = \frac{\kappa(A)\|A\|}{\|A\| - \kappa(A)\|\delta A\|} \cdot \left(\frac{\|\delta b\|}{\|b\|} + \frac{\|\delta A\|}{\|A\|} \right) \approx 5.25$$

which is what's expected since:

$$\|\delta A\| = 10^{-6}$$

but:

$$\frac{1}{\|A^{-1}\|} \approx 5 \times 10^{-6} > \|\delta A\|$$

Full credit for this question was given if student has written the correct formula for the error upper bound and made reasonable attempt to evaluate it (exact number does not have to be meticulously matched), and has correctly computed $\delta A, \delta b$. The error bound should be on the order of $O(10^{-5})$. The perturbed solution was more rigorously checked.

4 Homework 4

4.1 Implement CG Method

Create a function with inputs matrix A, vector b, $x^{(0)}$ as initial guess and tolerance tol. The algorithm implements conjugate gradient method and should find an approximate solution to $Ax = b$ without preconditioning. CG should terminate after n steps. Run your algorithm with test linear systems.

4.1.1

Example implementation in Python and Julia are as follows, with very simple test with:

$$A = \begin{pmatrix} 4 & 1 \\ 1 & 3 \end{pmatrix}, b = \begin{pmatrix} 1 \\ 2 \end{pmatrix}$$

(it is a very good idea to try translating the code to MATLAB if you are not familiar with the algorithm, MATLAB code was intentionally not provided).

Python version (next page):

161

hw4_code

March 10, 2021

[4]:
```
# reference code for HW4 P1
import numpy as np
import numpy.linalg as la
# set random seed for generating matrices
np.random.seed(10)
```

[141]:
```
def conj_grad(A, b, x0, n=100, tol=1e-4):
    # (7.31) from B&F
    """ conjugate gradient method without preconditioning
    A   (numpy.ndarray)    (N x N) matrix
    b   (numpy.ndarray)    (N x 1) vector
    x0  (numpy.ndarray)    (N x 1) initial guess
    n   (int)              number of iterations allowed
    tol (float)            tolerance, default 1e-6
    """
    k = 0
    r_new = b - A@x0
    xk = x0
    vk = r_new
    r_old = r_new
    all_err = np.array([])
    while k <= n:
        k += 1
        if la.norm(r_new) < tol:
            return xk, all_err
        else:
            all_err = np.append(all_err, la.norm(r_new, float('inf')))
            tk = r_old.dot(r_old)/vk.dot(A@vk)
            r_new = r_new - tk*(A@vk)
            sk = r_new.dot(r_new)/r_old.dot(r_old)
            xk = xk + tk*vk
            vk = r_new + sk*vk
            r_old = r_new
    raise FloatingPointError("CG could not converge")
```

[142]:
```
A = np.array([
    [4, 1],
```

1

162

```
    [1, 3]
])
b = np.array([1,2])
x0 = np.array([2, 1])
x_exact = la.solve(A, b)
x_cg, all_err = conj_grad(A, b, x0)
```

[143]:
```
all_err
```

[143]:
```
array([8.        , 0.74924471])
```

[144]:
```
# exact
la.inv(A)@b
```

[144]:
```
array([0.09090909, 0.63636364])
```

[145]:
```
# CG should converge in 2 steps, as shown in len(all_err)
x_cg
```

[145]:
```
array([0.09090909, 0.63636364])
```

163

Untitled

March 10, 2021

[1]:
```
using LinearAlgebra
```

[7]:
```
# Conjugate gradient solver

function CG(A,b,x0,tol=1e-5,maxit=1000)
    r = b - A * x0;
    x = x0
    p = r;
    rsold = transpose(r) * r;
    if norm(rsold) < tol
        return x
    end
    for i in collect(1:length(b))
        Ap = A * p;
        alpha = rsold / (transpose(p) * Ap);
        x = x + alpha * p;
        r = r - alpha * Ap;
        rsnew = transpose(r) * r;
        if sqrt(rsnew) < tol
              break;
        end
        p = r + (rsnew / rsold) * p;
        rsold = rsnew;
    end
    return x
end
```

[7]:
```
CG (generic function with 3 methods)
```

[18]:
```
A = [4 1; 1 3];
b = [1,2];
x_exact = A\b
```

[18]:
```
2-element Array{Float64,1}:
 0.09090909090909091
 0.6363636363636364
```

[22]:
```
x0 = [2,1];
CG(A, b, x0)
```

```
164
```

```
[22]: 2-element Array{Float64,1}:
       0.09090909090909094
       0.6363636363636365
```

It is possible for CG to not converge if we set tolerance to be too low (close to machine ϵ) or $\kappa(A)$ is very large, which will cause numerical issues with floating point arithmetic (depending on specific choice of random matrices). Preconditioned CG is supposed to mitigate this issue. The correctness for this problem is graded based on correct implementation.

4.2 Rewrite CG with Preconditioning

Rewrite CG algorithm in previous section with input matrix C. And discuss good and bad choices of C.

4.2.1

Please refer to B&F Algorithm 7.5 and convert it to MATLAB code. Correctness of this question is graded based on implementation. For the preconditioner, it is okay to try a few choices and discuss (good discussion involves trying a good preconditioner, a bad conditioner, and base solution $C = I$, and observe the effects of each on condition numbers and respective approximate solutions), the most common preconditioner is the incomplete Cholesky factorization (refer to B&F last paragraph in section 7.6). The code should be at least correct when $C = I$, meaning there is no preconditioning at all.

4.3

Use the Gershgorin Circle theorem to determine bounds for the eigenvalues and the spectral radius of the following matrices.

4.3.1 a

Errata: The comments about eigenvalue locations are only empirically true. As predicted by the Theorem, the eigenvalues can be anywhere in the unioned circles. There are only upper bounds for the spectral radius $\rho(A)$. For more details, please post any questions / comments on Piazza post @50.

Follow Theorem 9.1, we first find the discs.

$$A = \begin{pmatrix} 4 & 0 & 1 & 3 \\ 0 & 4 & 2 & 1 \\ 1 & 2 & -2 & 0 \\ 3 & 1 & 0 & 4 \end{pmatrix}$$

Then respectively:

$$R_1 = \{z \in \mathbb{C} : |z - a_{11}| \leq \sum_{j=1, j\neq 1}^{n} |a_{1j}|\} = \{z \in \mathbb{C} : |z - 4| \leq 4\}$$

$$R_2 = \{z \in \mathbb{C} : |z - a_{21}| \leq \sum_{j=1, j\neq 2}^{n} |a_{2j}|\} = \{z \in \mathbb{C} : |z - 4| \leq 3\}$$

we can see $R_2 \subset R_1$.

$$R_3 = \{z \in \mathbb{C} : |z - a_{31}| \leq \sum_{j=3, j\neq 3}^{n} |a_{3j}|\} = \{z \in \mathbb{C} : |z + 2| \leq 3\}$$

$$R_4 = \{z \in \mathbb{C} : |z - a_{41}| \leq \sum_{j=1, j\neq 4}^{n} |a_{4j}|\} = \{z \in \mathbb{C} : |z - 4| \leq 4\}$$

therefore $R_2 \subset R_1 = R_4$. If we graph the discs on the complex plane, we see 2 circles centered at $z = 4$ with radius 3 and 4, and 1 circle centered at $z = -2$ with radius 3. Furthermore, there are precisely 3 eigenvalues in R_1, and 1 eigenvalue in R_3.

Finally,

$$\rho(A) = \max_{i \in \{1,2,3,4\}} |\lambda_i|$$

and we can determine a bound:

$$\rho(A) \leq 8$$

by focusing on the region as far right on the real line as possible.

4.3.2 b

$$A = \begin{pmatrix} 1 & 0 & -1 & 1 \\ 2 & 2 & -1 & 1 \\ 0 & 1 & 3 & -2 \\ 1 & 0 & 1 & 4 \end{pmatrix}$$

similar to part (a), we determine:

$$R_1 = \{z \in \mathbb{C} : |z - 1| \leq 2\}, R_2 = \{z \in \mathbb{C} : |z - 2| \leq 4\}$$

$$R_3 = \{z \in \mathbb{C} : |z - 3| \leq 3\}, R_4 = \{z \in \mathbb{C} : |z - 4| \leq 2\}$$

$$\rho(A) \leq 6$$

4.4 Matrix Product Similarity

Let A, B be $n \times n$ and nonsingular, show that AB is similar to BA.

4.4.1

Refer to Definition 9.11 in B&F, a matrix M is said to be similar to N if there is a nonsingular matrix S such that:

$$M = S^{-1}NS$$

similar matrices are important because the eigenvalues of M are preserved through this transformation by S, which is a useful property in designing many iterative methods.

In this problem, we have A, B both invertible. We can consider:

$$AB = A(BA)A^{-1}$$

or:

$$BA = A^{-1}(AB)A$$

then setting $M = BA$, $N = AB$, $S = A$ shows that AB, BA are similar.

5 Homework 5

5.1 Power Method

Create a function which inputs matrix A, initial vector $x^{(0)}$, tolerance tol and uses the power method with stopping criteria:

$$\left\|x^{(k)} - x^{(k-1)}\right\|_\infty < tol$$

to obtain an approximate eigenpair (λ, x). Test the algorithm on:

$$A = \begin{pmatrix} 2 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 2 \end{pmatrix}, x^{(0)} = \begin{pmatrix} 1 \\ -1 \\ 2 \end{pmatrix}, tol = 0.01$$

5.1.1

The matrix in our problem is symmetric, a Julia implementation of Algorithm 9.2 from textbook is as follows. If your eigenvector is not the same as displayed below, it is not incorrect as long as it is up to a scalar multiple of (the ones below).

5.2

Let:

$$A = \begin{bmatrix} 5 & 2 & 0 & 0 \\ 1 & 4 & -1 & 0 \\ 0 & -1 & 4 & 2 \\ 0 & 0 & 1 & 5 \end{bmatrix}$$

Use the Power Method, Wielandt deflation, and the Inverse Power method to approximate the eigenvalues and eigenvectors of A.

Rubrics: This question is graded based on steps, procedures (if done on paper with the aid of linear solvers or calculators), or implementations (if done entirely in MATLAB). Two main points are update rules for approximate eigenvectors and that for eigenvectors. If student used MATLAB to perform Inverse Iteration with $q = 0$, same credit was given if student did not implement Algorithm 9.3, and simply called the Power Method (implemented previously) with A^{-1}. Incomplete hand writing that does not show any steps how each vectors are obtained (namely only vectors with arbitrary

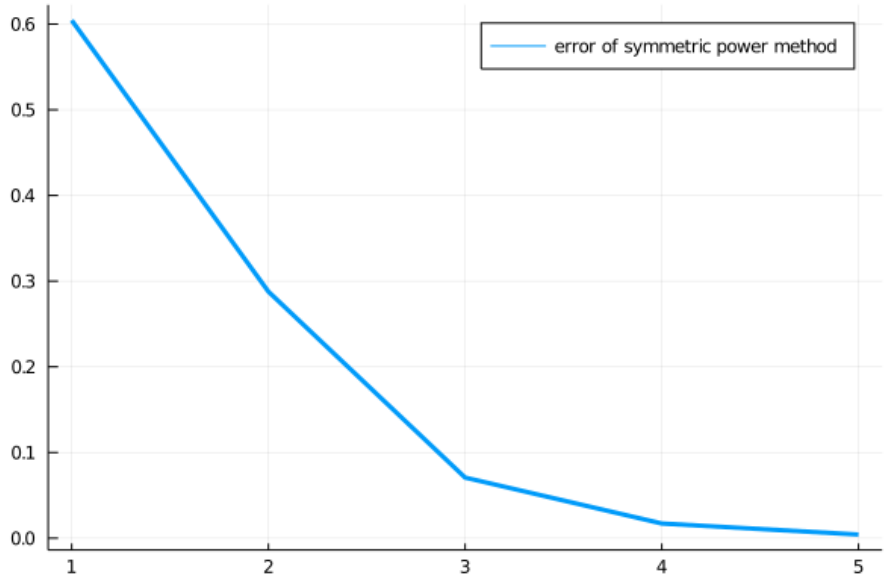


Figure 1: Error of symmetric power method (Homework 5, Problem 1)

numbers without context were written) calculations was penalized, despite final solutions are correct. For the Wielandt part, **points are not deducted** if student does not use estimated dominant eigen-pairs, and had directly calculated true eigenvalues and eigenvectors, as solution has suggested.

Follow Algorithms 9.1, 9.3, 9.4 for MATLAB solution.

5.2.1

$$A = \begin{pmatrix} 5 & 2 & 0 & 0 \\ 1 & 4 & -1 & 0 \\ 0 & -1 & 4 & 2 \\ 0 & 0 & 1 & 5 \end{pmatrix}$$

If we solve the characteristic polynomials, we will find the following eigenvalues (sorted, exact calculations omitted).

$$\lambda_1 = 5 + \sqrt{2} \approx 6.41421$$

$$\lambda_2 = 4 + \sqrt{3} \approx 5.73205$$

$$\lambda_3 = 5 - \sqrt{2} \approx 3.58579$$

$$\lambda_4 = 4 - \sqrt{3} \approx 2.26795$$

With approximate eigenvectors (corresponding to the sorted eigenvalues):

$$v_1 \approx \begin{pmatrix} -2.0 \\ -1.414 \\ 1.414 \\ 1.0 \end{pmatrix}$$

$$v_2 \approx \begin{pmatrix} 2.0 \\ 0.7321 \\ 0.7321 \\ 1.0 \end{pmatrix}$$

$$v_3 \approx \begin{pmatrix} -2.0 \\ 1.414 \\ -1.414 \\ 1.0 \end{pmatrix}$$

$$v_4 \approx \begin{pmatrix} 2.0 \\ -2.7321 \\ -2.7321 \\ 1.0 \end{pmatrix}$$

Now we run the algorithms to approximate these values.

5.2.2 Power Method

We aim to use the Power iteration to find the largest eigenvalue and associated eigenvector (λ_1, v_1). We use the steps outlined in chapter 9.3; since this algorithm is purely based on hitting the initial vector with A, it's okay to randomly pick an initial guess, so long as it is not 0. In here we choose:

$$x_0 = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

Hitting it with A a few times gives us:

$$x_1 = Ax_0 = \begin{pmatrix} 5 \\ 1 \\ 0 \\ 0 \end{pmatrix}, x_2 = Ax_1 = \begin{pmatrix} 27 \\ 9 \\ -1 \\ 0 \end{pmatrix}, x_3 = Ax_2 = \begin{pmatrix} 153 \\ 64 \\ -13 \\ -1 \end{pmatrix}$$

$$x_4 = Ax_3 = \begin{pmatrix} 893 \\ 422 \\ -118 \\ -18 \end{pmatrix}, x_5 = Ax_4 = \begin{pmatrix} 5309 \\ 2699 \\ -930 \\ -208 \end{pmatrix}, x_6 = Ax_5 = \begin{pmatrix} 31943 \\ 17035 \\ -6835 \\ -1970 \end{pmatrix}$$

Then the sequence of eigenvalue approximations is:

$$\lambda_1^{(1)} = \frac{\|x_1\|_\infty}{\|x_0\|_\infty} = \frac{5}{1} = 5, \lambda_1^{(2)} = \frac{\|x_2\|_\infty}{\|x_1\|_\infty} = \frac{27}{5} = 5.4$$

$$\lambda_1^{(3)} = \frac{\|x_3\|_\infty}{\|x_2\|_\infty} = \frac{153}{27} \approx 5.6667, \lambda_1^{(4)} = \frac{\|x_4\|_\infty}{\|x_3\|_\infty} = \frac{893}{153} \approx 5.8366$$

$$\lambda_1^{(5)} = \frac{\|x_5\|_\infty}{\|x_4\|_\infty} = \frac{5309}{893} \approx 5.94513, \lambda_1^{(6)} = \frac{\|x_6\|_\infty}{\|x_5\|_\infty} = \frac{31943}{5309} \approx 6.0168$$

More steps are omitted, if we had some patience, we see:

$$x_{10}/\|x_{10}\|_\infty \approx \begin{pmatrix} 1.0 \\ 0.58898 \\ -0.34148 \\ -0.16187 \end{pmatrix}, \lambda_1^{(10)} \approx 6.15682$$

$$x_{20}/\|x_{20}\|_\infty \approx \begin{pmatrix} 1.0 \\ 0.65843 \\ -0.55397 \\ -0.35731 \end{pmatrix}, \lambda_1^{(20)} \approx 6.30707$$

This approximates the largest eigenvalue to within $O(10^{-1})$, and it begins to converge very slowly (but it eventually will give us the eigenvalue). If we multiply the normalized vector $\widehat{x_{20}}$ by -2 to see whether we have a good approximation to v_1, we have:

$$\widehat{x_{20}} \cdot (-2) \approx \begin{pmatrix} -2.0 \\ -1.3169 \\ 1.1079 \\ 0.7146 \end{pmatrix}$$

which is not very good, but we are nearing the true values.

5.2.3 Inverse Iteration

If A is invertible, applying the power method on A^{-1} will yield the smallest eigenvalue and the corresponding eigenvector. This corresponds to setting $q = 0$. If A contains eigenvalues that are 0, this might not be a good idea; a remedy is to set q to be a very small number so that $1/q < \infty$.

In the following, we run the power method on A^{-1}. Take the starting value:

$$x^{(0)} = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}$$

exact steps are similar to the previous part. Since only the eigenvalues are modified through A^{-1}, the eigenvectors should be the same as A. For an implementation, follow Algorithm 9.3 in textbook. Through some linear system solving, we have (after obtaining the scalars, invert them):

$$\lambda_4^{(10)} \approx 2.27515, \lambda_4^{(20)} \approx 2.26802$$

which yielded approximate eigenvectors (normalized):

$$v_4^{(10)} \approx \begin{pmatrix} -0.73495 \\ 1.0 \\ 0.98990 \\ -0.36029 \end{pmatrix}, v_4^{(20)} \approx \begin{pmatrix} -0.73208 \\ 1.0 \\ 0.99990 \\ -0.36597 \end{pmatrix}$$

if we make the first entry 2 by scaling, we see:

$$\widehat{v_4}^{(20)} \approx \begin{pmatrix} 2.0 \\ -2.7319 \\ -2.7316 \\ 1.0 \end{pmatrix}$$

which is certainly very close to the true v_4.

5.2.4 Wielandt Deflation

The Wielandt Deflation fills in the rest of the eigenvalues, λ_2, λ_3. To be consistent with our calculations, we use the **estimated** dominating eigenpairs to feed into the deflation method. In realistic settings, we often do not

240 know the exact dominating eigen-pairs. But through running Power Method
241 for long enough, we will eventually get to one.

The dominant eigen-pairs found through power method after 100 iterations:

$$\tilde{\lambda}_1 = \lambda_1^{(100)} \approx 6.41429$$

with corresponding eigen-vector:

$$\tilde{v}_1 = \tilde{v}_1^{(100)} \approx \begin{pmatrix} -2.0 \\ -1.4142764 \\ 1.41441128 \\ 1.00018424 \end{pmatrix}$$

these are close enough to the true solution. Let us use this eigenpair for the first deflation (need to normalize the eigenvector). The normalized vector is:

$$\widehat{\tilde{v}}_1 \approx \begin{pmatrix} -1.0 \\ -0.707 \\ 0.707 \\ 0.500 \end{pmatrix} =: v_1$$

The procedure is described in Theorem 9.20 of our textbook. Let $i = 1$,

$$x = \frac{1}{\lambda_1 (v_1)_1}(A_{11}, A_{12}, \cdots, A_{1n})^T = \frac{1}{6.41429 \cdot (-1)} \cdot \begin{pmatrix} 5 \\ 2 \\ 0 \\ 0 \end{pmatrix} \approx \begin{pmatrix} -0.7795 \\ -0.3118 \\ 0 \\ 0 \end{pmatrix}$$

Then:

$$v_1 \cdot x^T = \begin{pmatrix} -1.0 \\ -0.707 \\ 0.707 \\ 0.500 \end{pmatrix} \cdot \begin{pmatrix} -0.7795 & -0.3118 & 0 & 0 \end{pmatrix} \approx \begin{pmatrix} 0.7795 & 0.3118 & 0 & 0 \\ 0.5512 & 0.2205 & 0 & 0 \\ -0.5513 & -0.2205 & 0 & 0 \\ -0.3898 & -0.1559 & 0 & 0 \end{pmatrix}$$

Deflate matrix A:

$$B = A - \lambda_1 v_1 \cdot x^T \approx \begin{pmatrix} O(10^{-6}) & 0 & 0 & 0 \\ -2.5357 & 2.5857 & -1.0 & 0 \\ 3.536 & 0.4144 & 4.0 & 2.0 \\ 2.5005 & 1.0 & 1.0 & 5.0 \end{pmatrix}$$

then we can get rid of the first row and first column of B to get a smaller matrix:

$$A' = \begin{pmatrix} 2.5857 & -1.0 & 0 \\ 0.4144 & 4.0 & 2.0 \\ 1.0 & 1.0 & 5.0 \end{pmatrix}$$

Now we can run Power Method again to find the second dominant eigenpair. Exact steps are left out, after 100 iterations, we obtain:

$$\lambda_2^{(100)} \approx 5.7320$$

with estimated (sub)-eigenvector (normalized):

$$w_2^{(100)} \approx \begin{pmatrix} -0.3178 \\ 1.0 \\ 0.9319 \end{pmatrix}$$

append a 0 entry, namely $w_2 = (0, -0.3178, 1.0, 0.9319)^T$, and we can recover via Equation (9.6) our v_2 approximation:

$$v_2 = (\lambda_2 - \lambda_1)w_2 + \lambda_1(x^T w_2)v_1 \approx \begin{pmatrix} 2.0 \\ 0.731986404630087 \\ 0.732253436240512 \\ 1.00018881986252 \end{pmatrix}$$

as expected. The last eigenpair can be obtained in a similar way by deflating B with the obtained pair (λ_2, v_2).

5.3

Use Householder's method to place the following matrix in tridiagonal form:

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

5.3.1

Details for the concrete meaning of these numbers were outlined and developed in Lecture 11. Follow textbook Theorem 9.22. If you were following Page 606, there was a slight typo in r, where $\alpha_{k+1,k}$ should be $A_{k+1,k}$, the subdiagonal entry from the matrix.

To find $P^{(1)}$, we compute the following:

$$A^{(1)} = A = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{pmatrix}$$

$$\alpha_1 = -sgn(A^{(1)}_{2,1})\sqrt{\left(\sum_{j=2}^{4}(A^{(1)}_{j1})^2\right)} = -\sqrt{1^2+1^2+1^2} = -\sqrt{3}$$

$$r_1 = \sqrt{\frac{1}{2}\left(\alpha_1^2 - \alpha \cdot A^{(1)}_{2,1}\right)} = \sqrt{\frac{1}{2}(3+\sqrt{3}\cdot 1)} = \sqrt{\frac{1}{2}(3+\sqrt{3})}$$

$$w^{(1)} = \begin{pmatrix} 0 & \frac{A^{(1)}_{2,1}-\alpha_1}{2r_1} & \frac{A^{(1)}_{3,1}}{2r_1} & \frac{A^{(1)}_{4,1}}{2r_1} \end{pmatrix}^T = (0, \frac{1+\sqrt{3}}{\sqrt{6+2\sqrt{3}}}, \frac{1}{\sqrt{6+2\sqrt{3}}}, \frac{1}{\sqrt{6+2\sqrt{3}}})^T$$

$$P^{(1)} = I - 2w^{(1)}(w^{(1)})^T = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & -\frac{2(1+\sqrt{3})^2}{6+2\sqrt{3}}+1 & -\frac{2(1+\sqrt{3})}{6+2\sqrt{3}} & -\frac{2(1+\sqrt{3})}{6+2\sqrt{3}} \\ 0 & -\frac{2(1+\sqrt{3})}{6+2\sqrt{3}} & 1-\frac{2}{6+2\sqrt{3}} & -\frac{2}{6+2\sqrt{3}} \\ 0 & -\frac{2(1+\sqrt{3})}{6+2\sqrt{3}} & -\frac{2}{6+2\sqrt{3}} & 1-\frac{2}{6+2\sqrt{3}} \end{pmatrix}$$

Then the first Householder transformation yields:

$$A^{(2)} = P^{(1)} \cdot A \cdot P^{(1)} = \begin{pmatrix} 1 & -\sqrt{3} & 0 & 0 \\ -\sqrt{3} & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

which is in tridiagonal form, therefore we stop.

5.4

Modify Householder's algorithm to find a Hessenberg matrix similar to the matrix:

$$A = \begin{bmatrix} 4 & 1 & 1 & 1 \\ 1 & 4 & 0 & 0 \\ 1 & 1 & 4 & 0 \\ 1 & 1 & 1 & 4 \end{bmatrix}$$

5.4.1

Would like to run Householder to reduce:

$$A = \begin{pmatrix} 4 & 1 & 1 & 1 \\ 1 & 4 & 0 & 0 \\ 1 & 1 & 4 & 0 \\ 1 & 1 & 1 & 4 \end{pmatrix}$$

to upper Hessenberg form. Following Algorithm 9.5 plus the modifications.

$$k = 1, A^{(1)} = A$$

$$q_1 = \sum_{j=2}^{4} (A_{j1})^2 = 1^2 + 1^2 + 1^2 = 3$$

$$\alpha_1 = -\frac{\sqrt{q_1} \cdot A_{21}}{|A_{21}|} = -\sqrt{3}$$

$$2r_1^2 = RSQ = \alpha_1^2 - \alpha_1 \cdot A_{21} = 3 + \sqrt{3}$$

$$v_1 = 0, v_2 = A_{21} - \alpha_1 = 1 + \sqrt{3}, v_3 = A_{31} = 1, v_4 = A_{41} = 1$$

(Step 6 is modified as the following):

$$u_1 = \frac{1}{RSQ}\sum_{i=2}^{4} A_{1i}v_i = \frac{1}{3+\sqrt{3}}(1\cdot(1+\sqrt{3})+1\cdot 1+1\cdot 1) = 1$$

$$u_2 = \frac{1}{RSQ}\sum_{i=2}^{4} A_{2i}v_i = \frac{4\cdot(1+\sqrt{3})}{3+\sqrt{3}} = \frac{4+4\sqrt{3}}{3+\sqrt{3}}$$

$$u_3 = \frac{1}{RSQ}\sum_{i=2}^{4} A_{3i}v_i = \frac{1+\sqrt{3}+4\cdot 1}{3+\sqrt{3}} = \frac{5+\sqrt{3}}{3+\sqrt{3}}$$

$$u_4 = \frac{1}{RSQ}\sum_{i=2}^{4} A_{4i}v_i = \frac{(1+\sqrt{3}+1+4)}{3+\sqrt{3}} = \frac{6+\sqrt{3}}{3+\sqrt{3}}$$

$$y_1 = \frac{1}{RSQ}\sum_{i=2}^{4} A_{i1}v_i = \frac{1+\sqrt{3}+1+1}{3+\sqrt{3}} = 1$$

$$y_2 = \frac{1}{RSQ}\sum_{i=2}^{4} A_{i2}v_i = \frac{4\cdot(1+\sqrt{3})+1+1}{3+\sqrt{3}} = \frac{6+4\sqrt{3}}{3+\sqrt{3}}$$

$$y_3 = \frac{1}{RSQ}\sum_{i=2}^{4} A_{i3}v_i = \frac{4+1}{3+\sqrt{3}} = \frac{5}{3+\sqrt{3}}$$

$$y_4 = \frac{1}{RSQ}\sum_{i=2}^{4} A_{i4}v_i = \frac{}{3+\sqrt{3}} = \frac{4}{3+\sqrt{3}}$$

Now step 7:

$$PROD = \sum_{j=2}^{4} v_j u_j = \frac{(1+\sqrt{3})(4+4\sqrt{3})}{3+\sqrt{3}} + \frac{5+\sqrt{3}}{3+\sqrt{3}} + \frac{6+\sqrt{3}}{3+\sqrt{3}} = \frac{\sqrt{3}+17}{2}$$

step 8:

$$z_1 = u_1 - \frac{PROD}{RSQ}v_1 = 1$$

$$z_2 = u_2 - \frac{PROD}{RSQ}v_2 = -\frac{1+3\sqrt{3}}{2}$$

$$z_3 = u_3 - \frac{PROD}{RSQ}v_3 = -2+\frac{5}{6}\sqrt{3}$$

$$z_4 = u_4 - \frac{PROD}{RSQ} v_4 = -3 + \frac{7}{6}\sqrt{3}$$

step 9:

$$A_{12}^{(2)} = A_{12} - z_1 v_2 = 1 - 1 - \sqrt{3} = -\sqrt{3}$$

$$A_{21}^{(2)} = A_{21} - y_1 v_2 = 1 - 1 - \sqrt{3} = -\sqrt{3}$$

$$A_{22}^{(2)} = A_{22} - z_2 v_2 - y_2 v_2 = 4 + \frac{(1+3\sqrt{3})(1+\sqrt{3})}{2} - \frac{(6+3\sqrt{3})(1+\sqrt{3})}{3+\sqrt{3}} = 5$$

$$A_{32}^{(2)} = A_{32} - z_3 v_2 - y_2 z_3 = 1 - (-2 + \frac{5}{6}\sqrt{3})(1+\sqrt{3}) - \frac{6+4\sqrt{3}}{3+\sqrt{3}} \approx -0.2113$$

$$A_{42}^{(2)} = A_{42} - z_4 v_2 - y_2 v_4 = 1 - (-3 + \frac{7}{6}\sqrt{3})(1+\sqrt{3}) - \frac{6+4\sqrt{3}}{3+\sqrt{3}} \approx 0.9434$$

$$A_{13}^{(2)} = A_{13} - z_1 v_3 = 1 - 1 = 0$$

$$A_{31}^{(2)} = A_{31} - y_1 v_3 = 1 - 1 = 0$$

$$A_{23}^{(2)} = A_{23} - z_2 v_3 - y_3 v_2 = 0 + \frac{(1+3\sqrt{3})}{2} - \frac{5(1+\sqrt{3})}{3+\sqrt{3}} \approx 0.2113$$

$$A_{33}^{(2)} = A_{33} - z_3 v_3 - y_3 v_3 = 4 - (-2 + \frac{5}{6}\sqrt{3}) - \frac{5}{3+\sqrt{3}} \approx 3.5$$

$$A_{43}^{(2)} = A_{43} - z_4 v_3 - y_3 v_4 = 1 - (-3 + \frac{7}{6}\sqrt{3}) - \frac{5}{3+\sqrt{3}} \approx 0.9226$$

$$A_{14}^{(2)} = A_{14} - z_1 v_4 = 1 - 1 = 0$$

$$A_{41}^{(2)} = A_{41} - y_1 v_4 = 1 - 1 = 0$$

$$A_{24}^{(2)} = A_{24} - z_2 v_4 - y_4 v_2 = \frac{1+3\sqrt{3}}{2} - \frac{4(1+\sqrt{3})}{3+\sqrt{3}} = \approx 0.7887$$

$$A_{34}^{(2)} = A_{34} - z_3 v_4 - y_4 v_3 = 2 - \frac{5}{6}\sqrt{3} - \frac{4}{3+\sqrt{3}} \approx -0.2887$$

$$A_{44}^{(2)} = A_{44} - z_4 v_4 - y_4 v_4 = 3.50$$

One step of Householder transformation should yield:

$$A^{(2)} = \begin{pmatrix} 4 & -\sqrt{3} & 0 & 0 \\ -\sqrt{3} & 5 & 0.2113 & 0.7887 \\ 0 & -0.2113 & 3.50 & -0.2887 \\ 0 & 0.9434 & 0.9226 & 3.50 \end{pmatrix}$$

Another step should yield the end result, close to:

$$\begin{pmatrix} 4 & -1.732 & 0 & 0 \\ -1.732 & 5 & -0.817 & 0 \\ 0 & 0.817 & 3.50 & 0.289 \\ 0 & 0 & -0.289 & 3.50 \end{pmatrix}$$

6 Midterm Exam

6.1 Limit of Matrix Power

Determine, with proof the limit:

$$\lim_{n\to\infty} A^n v$$

where:

$$A = \begin{pmatrix} 2/3 & -3 & 12 \\ 0 & 1/2 & 1/2 \\ 0 & 1/3 & 1/2 \end{pmatrix}, v = \begin{pmatrix} 1 \\ -1 \\ 1 \end{pmatrix}$$

6.1.1

$$A = \begin{pmatrix} \frac{2}{3} & -3 & 12 \\ 0 & \frac{1}{2} & \frac{1}{2} \\ 0 & \frac{1}{3} & \frac{1}{2} \end{pmatrix}$$

It is enough to show that A is a convergent matrix, we do so by computing eigenvalues by hand:

$$A - \lambda I = \begin{pmatrix} \frac{2}{3} - \lambda & -3 & 12 \\ 0 & \frac{1}{2} - \lambda & \frac{1}{2} \\ 0 & \frac{1}{3} & \frac{1}{2} - \lambda \end{pmatrix}$$

then set:

$$\det(A - \lambda I) = 0$$

and solve:

$$(\frac{2}{3} - \lambda)\left[(\lambda - \frac{1}{2})^2 - \frac{1}{6}\right] = 0$$

after some computations, we conclude the our eigenvalues are:

$$\lambda_1 = \frac{2}{3}, \lambda_2 = \frac{1}{2} + \frac{1}{\sqrt{6}}, \lambda_3 = \frac{1}{2} - \frac{1}{\sqrt{6}}$$

Since electronics were prohibited, let's bound $\frac{1}{\sqrt{6}}$, we have $\frac{1}{\sqrt{6}} < \frac{1}{2}$ since $\sqrt{6} > 2$ then we have:

$$\frac{1}{2} + \frac{1}{\sqrt{6}} < 1/2 + 1/2 = 1$$

and certainly:

$$0 < \frac{1}{2} - \frac{1}{\sqrt{6}} < 1$$

In short, $\rho(A) < 1$: we have by Theorem 7.17 of BF that A is convergent, as a consequence:

$$\lim_{n\to\infty} A^n v = 0$$

6.2 l_2-norm and condition number

Determine, with proof, the l_2-norm and 2-condition number of the matrix:

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

6.2.1

First l_2 norm:

$$\|A\|_2^2 = \lambda_{max}(A^T A)$$

the largest eigenvalue of $A^T A$. We have:

$$B = A^T A = \begin{pmatrix} 5 & 2 \\ 2 & 1 \end{pmatrix}$$

and compute:

$$\det(B - \lambda I) = (\lambda - 5)(\lambda - 1) - 4 = 0$$

and solve, we have:

$$\lambda = 3 \pm 2\sqrt{2}$$

then the largest eigenvalue is $\lambda = 3 + 2\sqrt{2}$. Then:

$$\|A\|_2 = \sqrt{3 + 2\sqrt{2}}$$

The condition number is the two norm sense is:

$$\kappa(A) = \|A\|_2 \cdot \left\|A^{-1}\right\|_2$$

A is a nice symmetric matrix, we can write its eigendecomposition as:

$$A = PDP^{-1}$$

where P is orthogonal. Then we have:

$$A^{-1} = PD^{-1}P^{-1}$$

therefore nothing changed except for inverse eigenvalues.

We have:

$$\left\|A^{-1}\right\|_2^2 = \lambda_{\max}(A^{-T}A^{-1}) = \lambda_{\max}((AA^T)^{-1})$$

We have:

$$AA^T = A^TA = \begin{pmatrix} 5 & 2 \\ 2 & 1 \end{pmatrix}$$

whose inverse is:

$$(AA^T)^{-1} = \begin{pmatrix} 1 & -2 \\ -2 & 5 \end{pmatrix}$$

since this is a 2 by 2 matrix, in Math 54 or Math 110 we might have seen a handy trick to compute an inverse quickly (swap a, d and swap b, c then negate), and the determinant of AA^T happens to be 1.

We now find the eigenvalues of this new matrix by computing:

$$\det\left(B^{-1} - \lambda I\right) = 0$$

and solve:

$$(\lambda - 5)(\lambda - 1) = 4$$

which is the same as before, then we can conclude:

$$\kappa(A) = \|A\|_2 \cdot \left\|A^{-1}\right\|_2 = 3 + 2\sqrt{2}$$

6.3 Jacobi Method Computations

Let initial vector be $x^{(0)} = 0$, find two iterations of the Jacobi Method for the following linear system:

$$-2x_1 + x_2 + \frac{1}{2}x_3 = 4$$

$$x_1 - 2x_2 - \frac{1}{2}x_3 = 2$$

$$x_2 + 2x_3 = 0$$

6.3.1

Note: In class and also in BF, we defined the strictly lower, strictly upper, triangular part to be $-L, -U$.

In this problem we have:

$$A = \begin{pmatrix} -2 & 1 & \frac{1}{2} \\ 1 & -2 & -\frac{1}{2} \\ 0 & 1 & 2 \end{pmatrix}, b = \begin{pmatrix} 4 \\ 2 \\ 0 \end{pmatrix}$$

with initial vector $x^{(0)} = 0$.

By (7.6) in BF, the Jacobi Matrix Update Rule is:

$$x^{(i+1)} = D^{-1}(L+U)x^{(i)} + D^{-1}b$$

where $D + L + U = A$ are diagonal, lower triangular part, upper triangular part. There is no complicated inverse, and:

$$D^{-1} = diag([-\frac{1}{2}, -\frac{1}{2}, \frac{1}{2}])$$

And:

$$c = D^{-1}b = \begin{pmatrix} -1/2 & 0 & 0 \\ 0 & -1/2 & 0 \\ 0 & 0 & 1/2 \end{pmatrix} \begin{pmatrix} 4 \\ 2 \\ 0 \end{pmatrix} = \begin{pmatrix} -2 \\ -1 \\ 0 \end{pmatrix}$$

$$T = -D^{-1}(L+U) = -\begin{pmatrix} -1/2 & 0 & 0 \\ 0 & -1/2 & 0 \\ 0 & 0 & 1/2 \end{pmatrix} \left[\begin{pmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} + \begin{pmatrix} 0 & 1 & 1/2 \\ 0 & 0 & -1/2 \\ 0 & 0 & 0 \end{pmatrix} \right]$$

$$T = \begin{pmatrix} 0 & 1/2 & 1/4 \\ 1/2 & 0 & -1/4 \\ 0 & -1/2 & 0 \end{pmatrix}$$

Using T and c, two iterations are:

$$x^{(1)} = \begin{pmatrix} -2 \\ -1 \\ 0 \end{pmatrix}$$

$$x^{(2)} = \begin{pmatrix} -5/2 \\ -2 \\ 1/2 \end{pmatrix}$$

6.4 Householder Matrix Eigenvalues

Find eigenvalues of a real $n \times n$ Householder matrix $I - ww^T$ where $\|w\|_2 = 1$.

6.4.1

The householder matrix:

$$H = I - 2ww^T$$

where we have $\|w\|_2 = \sqrt{w^T w} = 1$. Here w is a column vector, ww^T means vector outer product with itself, and is a matrix. It doesn't make much sense to talk about $I - 2w^T w$.

ww^T has a nice property that it is symmetric, and $I - 2ww^T$ will be symmetric as well. The eigenvalues of real symmetric matrices are real. In matrix notation, we can see:

$$(I - 2ww^T)^T = I^T - 2(ww^T)^T = I - 2ww^T$$

and also since w is a unit vector, H is orthogonal:

$$(I - 2ww^T)(I - 2ww^T) = I - 4ww^T + 4ww^T ww^T$$

$$= I - 4ww^T + 4w(w^T w)w^T = I$$

In Math 110 we may have proved in one of the homework assignments that orthogonal matrices only have eigenvalues with magnitude 1. But H is

symmetric, thus all eigenvalues of H are real. Then we know that the only possible eigenvalues of H here are ± 1 (should be proved). Indeed:

$$(I - 2ww^T)w = w - 2w(w^T w) = w - 2w = -w = (-1) \cdot w$$

and take any unit vector v such that $w \perp v$, we have:

$$(I - 2ww^T)v = Iv - 2ww^T v = Iv = v = (1) \cdot v$$

7 Homework 6

7.1 QR Decomposition

Create a function that implements the QR decomposition with shifts. Use the algorithm to determine, within 10^{-5} all eigenvalues of the matrix:

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

7.1.1

A Julia implementation is attached below for reference (Following Algorithm 9.6 in book). The eigenvalues should be close to:

$$\lambda_1 \approx 3.912, \lambda_2 \approx 2.130, \lambda_3 \approx -2.041$$

276

qr_shift

March 24, 2021

[57]:
```
using LinearAlgebra
```

[99]:
```
# alg. 9.6, with slight modifications
function qrws(A::SymTridiagonal{Float64,Array{Float64,1}}, tol=1e-5, maxit=1000)
    a = A.dv;
    d = copy(a);
    b = A.ev;
    n = length(diag);
    k = 1;
    shift = 0;
    all_lams = zeros(Float64, 0);
    while k <= maxit
        # step 3
        if abs(b[n]) <= tol
            all_lams = append!(all_lams, a[n] + shift);
            n = n - 1;
        end
        # step 4
        if abs(b[2]) <= tol
            all_lams = append!(all_lams, a[1] + shift);
            n = n - 1;
            a[1] = a[2];
            for j in collect(2:1:n)
                a[j] = a[j+1];
                b[j] = b[j+1];
            end
        end
        # step 5
        if n == 0
           break
        end
        # step 6
        if n == 1
            lambda = a[1] + shift;
            all_lams = append!(all_lams, lambda);
        end
        # step 7
```

277

```
    for j in collect(3:1:(n-1))
        if abs(b[j]) <= tol
            # split, no output, this solution is for simplicity
            break
        end
    end
    # step 8
    _b = -(a[n-1] + a[n]);
    _c = a[n]*a[n-1] - (b[n])^2;
    _d = sqrt((_b^2 - 4*_c));
    if _b > 0
        m1 = -2*_c/(_b+_d);
        m2 = -(_b+_d)/2;
    else
        m1 = (_d-_b)/2;
        m2 = 2*_c/(_d-_b);
    end
    # step 10
    if n == 2
        lambda1 = m1 + shift;
        lambda2 = m2 + shift;
        all_lams = append!(all_lams, [lambda1, lambda2]);
        break
    end
    #  step 11 choosing
    if abs(m1 - a[n]) <= abs(m2 - a[n])
        sig = m1;
    else
        sig = m2;
    end
    # step 12
    shift = shift + sig;

    # step 13
    for j in collect(1:n)
        d[j] = a[j] - sig;
    end
    # step 14 and 15
    x = zeros(Float64, n); y = zeros(Float64, n);
    c = zeros(Float64, n); sigmas = zeros(Float64, n);
    z = zeros(Float64, n); q = zeros(Float64, n);
    r = zeros(Float64, n);
    x[1] = d[1];
    y[1] = b[1]; # A[2,1]
    for j in collect(2:n)
        z[j-1] = sqrt((x[j-1])^2 + (b[j])^2);
        c[j] = x[j-1]/z[j-1];
```

278

```
            # println(z)
            sigmas[j] = b[j]/z[j-1];
            # there was a typo in B&F |
            q[j-1] = c[j]*y[j-1] + sigmas[j]*d[j];
            x[j] = -sigmas[j]*y[j-1] + c[j]*d[j];
            if j != n
                r[j-1] = sigmas[j]*b[j+1];
                y[j] = c[j]*b[j+1];
            end
        end
            # steps 16-18
        z[n] = x[n];
        a[1] = sigmas[2]*q[1] + c[2]*z[1];
        b[2] = sigmas[2]*z[2];

        for j in collect(2:1:(n-1))
            a[j] = sigmas[j+1]*q[j] + c[j]*c[j+1]*z[j];
            b[j+1] = sigmas[j+1]*z[j+1];
        end
        # step 18
        a[n] = c[n]*z[n];
        k = k + 1;
    end
end
```

[99]: qrws (generic function with 9 methods)

7.2 Find Singular Values

Determine singular values of the following matrices:

$$\begin{pmatrix} 2 & 1 \\ 1 & 1 \\ 0 & 1 \end{pmatrix}, \begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix}$$

7.2.1 a

We can compute the singular values by hand using Definition 9.27.

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

$$B = A^T A = \begin{pmatrix} 5 & 3 \\ 3 & 3 \end{pmatrix}$$

then we solve $\det(B - \lambda I) = 0$, or:

$$\det\left[\begin{pmatrix} 5-\lambda & 3 \\ 3 & 3-\lambda \end{pmatrix}\right] = \lambda^2 - 8\lambda - 6 = 0$$

$$\lambda_1 = 4 + \sqrt{10}, \lambda_2 = 4 - \sqrt{10}$$

Then the two singular values are:

$$\sigma_1 = \sqrt{4 + \sqrt{10}}, \sigma_2 = \sqrt{4 - \sqrt{10}}$$

7.2.2 b

$$A = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix}$$

Then:

$$B = A^T A = \begin{pmatrix} 2 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 2 \end{pmatrix}$$

then solve $\det(B - \lambda I) = 0$, or:

$$-(\lambda - 2)^3 + 3\lambda - 4 = 0$$

then:

$$\lambda_1 = 4, \lambda_2 = \lambda_3 = 1$$

A has full rank, then we have 3 singular values:

$$\sigma_1 = 2, \sigma_2 = \sigma_3 = 1$$

7.3 l_2-condition number

Show that if A is a matrix with singular values $s_1, s_2, \cdots, s_n$, then its l_2-condition number is equal to:

$$\kappa(A) = \frac{s_1}{s_n}$$

7.3.1

By definition of two norm condition number:

$$\kappa(A) = \|A\|_2 \left\|A^{-1}\right\|$$

where A is nonsingular and admits the SVD:

$$A = U\Sigma V^*$$

where U, V are both unitary.

7.3.2 preservation of norm by unitary matrix

Let U be unitary, x be a vector, then:

$$\|Ux\|_2^2 = x^*U^*Ux = x^*x = \|x\|_2^2$$

therefore norm is preserved by unitary transformations.

7.3.3 two norm of A

$$\|A\|_2 = \sup_{\|x\|_2=1} \|Ax\|_2 = \sup_{\|x\|=1} \|U\Sigma V^*x\|_2 = \sup_{\|x\|=1} \|\Sigma V^*x\|_2 = \sup_{\|x\|=1} \|\Sigma y\|_2$$

renaming $y = V^*x$.

$$= \sup_{\|x\|=1} \left(\sum_1^n s_i^2 |y_i|^2 \right)^{1/2} = \sup_{\|y\|=1} \left(\sum_1^n s_i^2 |y_i|^2 \right)^{1/2} = s_1$$

where the last equality comes from $\|x\| = \|V^*x\| = \|y\|$. (WLOG assume the singular values are sorted) The maximum is attainable at $y = (1, 0, \cdots, 0)$.

7.3.4 two norm of A^{-1}

A is invertible by assumption, then:

$$A^{-1} = (U\Sigma V^*)^{-1} = V\Sigma^{-1}U^*$$

where Σ^{-1} is the diagonal matrix containing the inverse of the singular values of Σ. Then by a similar argument as before, we conclude:

$$\left\|A^{-1}\right\|_2 = 1/s_n$$

because $s_n < s_{n-1} < \cdots < s_1$.

7.3.5 conclusion

As above, we conclude:

$$\kappa(A) = \|A\|_2\left\|A^{-1}\right\|_2 = s_1/s_n$$

as desired.

8 Homework 7

8.1 Least Squares Polynomial Approximation

Find the least squares approximation of degree 2 and 3 to:

$$f(x) = e^x$$

on the interval $[-1, 1]$.

8.1.1

$$f(x) = e^x$$

we would like to find respectively degree 2 and 3 least squares polynomial approximation on $[-1, 1]$. The goal of least squares is to minimize:

$$\int_{-1}^{1} (f(x) - P_n(x))^2 dx$$

where:

$$P_2(x) = a_2x^2 + a_1x + a_0$$
$$P_3(x) = a_3x^3 + a_2x^2 + a_1x + a_0$$

using (8.6) from textbook, we need to solve the normal equations for $j = 0, 1, \cdots, n$:

$$\sum_{k=0}^{2} a_k \int_{-1}^{1} x^{j+k} dx = \int_{-1}^{1} x^j f(x) dx$$

For $n = 2$, we have the following:

$$\begin{cases} a_0 \int_{-1}^{1} 1dx + a_1 \int_{-1}^{1} xdx + a_2 \int_{-1}^{1} x^2 dx = \int_{-1}^{1} e^x dx \\ a_0 \int_{-1}^{1} xdx + a_1 \int_{-1}^{1} x^2 dx + a_2 \int_{-1}^{1} x^3 dx = \int_{-1}^{1} xe^x dx \\ a_0 \int_{-1}^{1} x^2 dx + a_1 \int_{-1}^{1} x^3 dx + a_2 \int_{-1}^{1} x^4 dx = \int_{-1}^{1} x^2 e^x dx \end{cases}$$

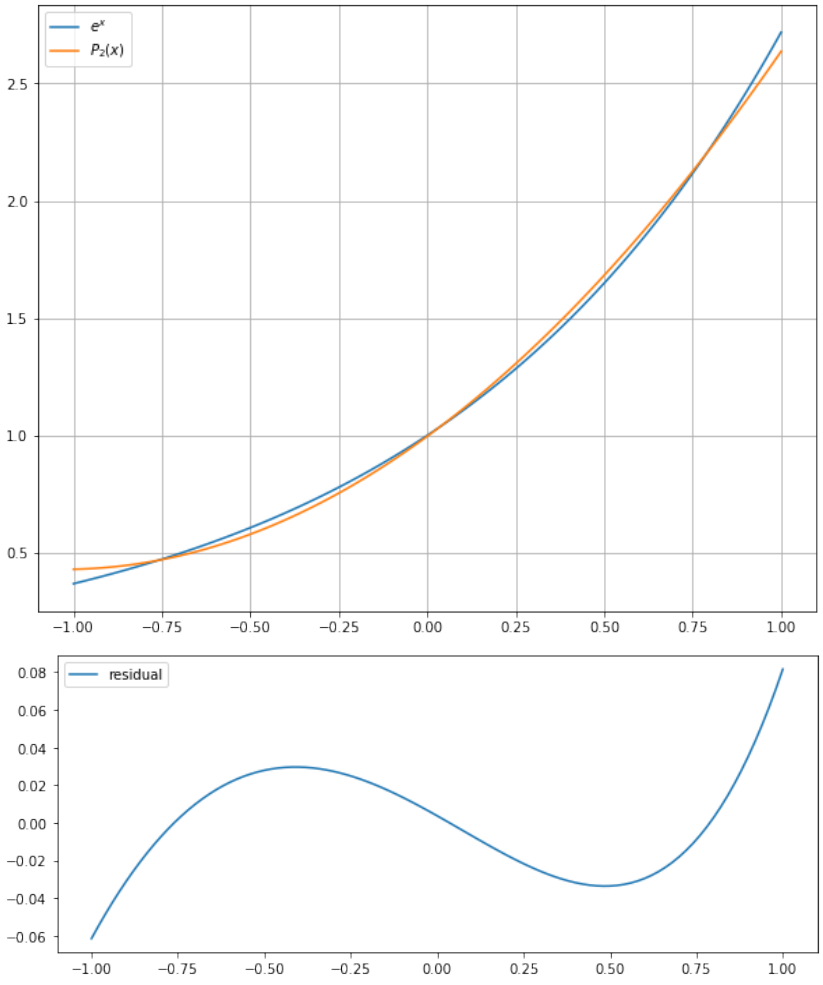
which yields the following linear system:

$$\begin{pmatrix} 2 & 0 & 2/3 \\ 0 & 2/3 & 0 \\ 2/3 & 0 & 2/5 \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \end{pmatrix} = \begin{pmatrix} e - \frac{1}{e} \\ \frac{2}{e} \\ e - \frac{5}{e} \end{pmatrix}$$

Using a linear solver, we obtain:

$$a \approx \begin{pmatrix} 0.53672153 \\ 1.10363832 \\ 0.99629402 \end{pmatrix}$$

299 The following plots show a degree-2 approximation would yield a degree-3 residual, as expected.



300

Now for $n = 3$:

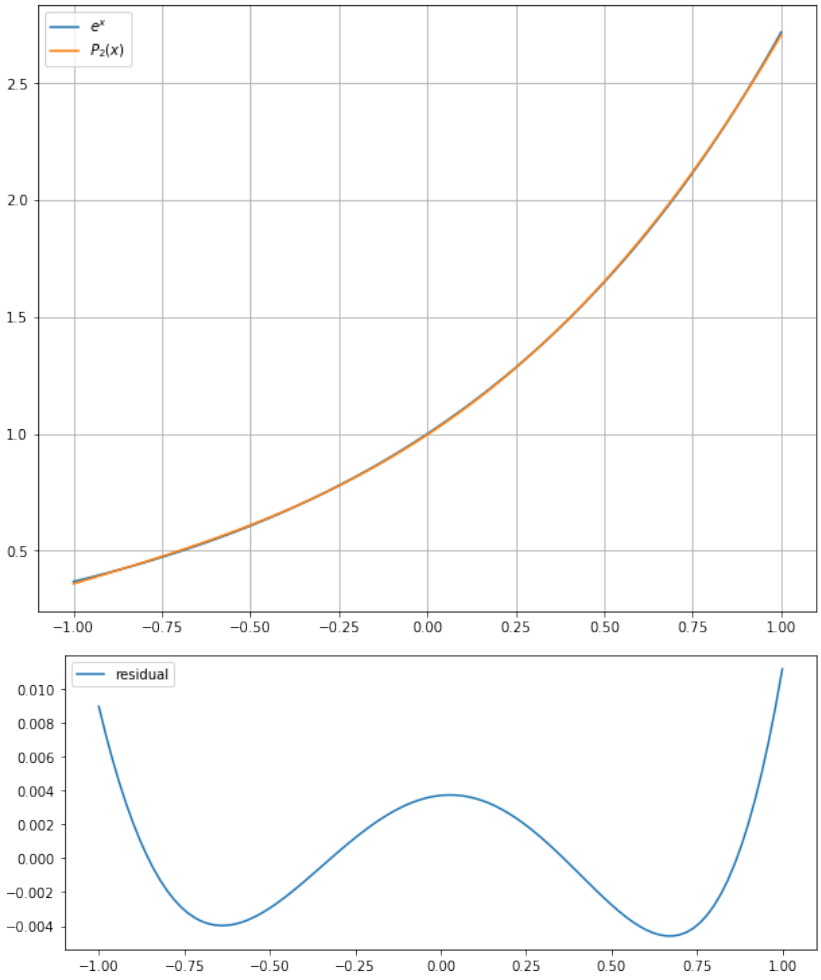
$$\begin{cases} a_0 \int_{-1}^{1} 1dx + a_1 \int_{-1}^{1} xdx + a_2 \int_{-1}^{1} x^2dx + a_3 \int_{-1}^{1} x^3dx = \int_{-1}^{1} e^x dx \\ a_0 \int_{-1}^{1} xdx + a_1 \int_{-1}^{1} x^2dx + a_2 \int_{-1}^{1} x^3dx + a_3 \int_{-1}^{1} x^4dx = \int_{-1}^{1} xe^x dx \\ a_0 \int_{-1}^{1} x^2dx + a_1 \int_{-1}^{1} x^3dx + a_2 \int_{-1}^{1} x^4dx + a_3 \int_{-1}^{1} x^5dx = \int_{-1}^{1} x^2e^x dx \\ a_0 \int_{-1}^{1} x^3dx + a_1 \int_{-1}^{1} x^4dx + a_2 \int_{-1}^{1} x^5dx + a_3 \int_{-1}^{1} x^6dx = \int_{-1}^{1} x^3e^x dx \end{cases}$$

which gives the system:

$$\begin{pmatrix} 2 & 0 & 2/3 & 0 \\ 0 & 2/3 & 0 & 2/5 \\ 2/3 & 0 & 2/5 & 0 \\ 0 & 2/5 & 0 & 2/7 \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{pmatrix} = \begin{pmatrix} e - 1/e \\ 2/e \\ e - 5/e \\ 16/e - 2e \end{pmatrix}$$

$$a \approx \begin{pmatrix} 0.99629402 \\ 0.99795487 \\ 0.53672153 \\ 0.17613908 \end{pmatrix}$$

The following figures show that for degree 3 approximation, we have degree 4 residual.



8.2 Chebyshev Polynomial Identity

Show that:

$$T_i(x)T_j(x) = \frac{1}{2}(T_{i+j}(x) + T_{i-j}(x)), \forall i > j$$

8.2.1

WLOG assume $i \geq j$, both $i+j, i-j \geq 0$. Would like to show for Chebyshev Polynomials:

$$T_i \cdot T_j = \frac{1}{2}(T_{i+j} + T_{i-j})$$

We use a trigonometric identity:

$$\cos(a+b) = \cos(a)\cos(b) - \sin(a)\sin(b)$$

$$\cos(a-b) = \cos(a)\cos(b) + \sin(a)\sin(b)$$

therefore we have:

$$\cos(a)\cos(b) = \frac{1}{2}(\cos(a+b) + \cos(a-b))$$

By definition of Chebyshev Polynomial:

$$T_i(x) \cdot T_j(x) = \cos(i \arccos(x)) \cdot \cos(j \arccos(x))$$

$$= \frac{1}{2}\Big[\cos((i+j)\arccos(x) + \cos((i-j)\arccos(x)))\Big] = \frac{1}{2}(T_{i+j}(x) + T_{i-j}(x))$$

as desired.

8.2.2 other properties of Chebyshev Polynomials

The following can be proved using the definition, recurrence relation, and previous properties (let us know if you attempted the proofs!).

(1)

$$T_n(1) = 1$$

(2)

$$T_n(x) = 2^{n-1}x^n + O(x^{n-1})$$

(3)

$$T_n(x) = \begin{cases} \cos(n \cdot \arccos(x)), \text{ if } |x| \leq 1 \\ \cosh(n \cdot \operatorname{arccosh}(x)), \text{ if } |x| \geq 1 \end{cases}$$

(4)

$$\text{if } |x| \leq 1, |T_n(x)| \leq 1$$

(5)

$$T_n(x) = \frac{1}{2}\big[(x + \sqrt{x^2-1})^m + (x + \sqrt{x^2-1})^{-m}\big], \text{ for} |x| > 1$$

(6)

$$T_n(1+\epsilon) \geq \frac{1}{2}(1 + n\sqrt{2\epsilon}), \text{ for } \epsilon > 0$$

8.3 Laguerre Polynomial Recurrence Relation

Derive the 3-term recurrence relation for the Laguerre polynomials L_n, which are orthogonal with respect to the weight function $w(x) = e^{-x}$ on the interval $(0, \infty)$. Plot the polynomials L_0, L_1, L_2, L_3 on an interval containing all of their respect zeros, discuss interlacing of roots.

8.3.1

The derivation of Laguerre Polynomial recurrence is graded, but lightly. The proof follows from plugging in B_k, C_k and verifying the general formula for (normalized) Laguerre polynomials (cannot use the recurrence in Theorem 8.7 directly without proof).

The general formula for Laguerre polynomials is:

$$L_k(x) = \sum_{j=0}^{k} \binom{k}{j} \frac{(-1)^j}{j!} x^j$$

yielding the recurrence:

$$(k+1)L_{k+1}(x) = (2k+1-x)L_k(x) - kL_{k-1}(x)$$

Computations:

Let:

$$L_0(x) = 1, w(x) = e^{-x}$$

we use Theorem 8.7 from textbook to construct polynomials of higher degrees with domain $(0, \infty)$.

$$B_1 = \frac{\int_0^\infty x e^{-x} dx}{\int_0^\infty e^{-x} dx} = \frac{1}{1} = 1$$

$$L_1(x) = x - 1$$

For $n = 2$:

$$B_2 = \frac{\int_0^\infty x e^{-x}(x-1)^2 dx}{\int_0^\infty e^{-x}(x-1)^2 dx} = \frac{3}{1} = 3$$

$$C_2 = \frac{\int_0^\infty x e^{-x}(x-1) dx}{\int_0^\infty e^{-x} dx} = \frac{1}{1} = 1$$

$$L_2(x) = (x-3)(x-1) - 1 = x^2 - 4x + 2$$

For $n = 3$:

$$B_3 = \frac{\int_0^\infty xe^{-x}(x^2 - 4x + 2)^2 dx}{\int_0^\infty e^{-x}(x^2 - 4x + 2)^2 dx} = \frac{20}{4} = 5$$

$$C_3 = \frac{\int_0^\infty xe^{-x}(x^2 - 4x + 2)(x - 1) dx}{\int_0^\infty e^{-x}(x - 1)^2 dx} = \frac{4}{1} = 4$$

$$L_3(x) = (x - 5) \cdot (x^2 - 4x + 2) - 4(x - 1) = x^3 - 9x^2 + 18x - 6$$

Numerical solutions:

L_0 has no zeros.

For L_1, $x_1^{(1)} = 1$.

For L_2, $x_2^{(1)} \approx 0.58579, x_2^{(2)} \approx 3.41421$.

For L_3, $x_3^{(1)} \approx 0.41577, x_3^{(2)} \approx 2.29428, x_3^{(3)} \approx 6.28995$. The following plot shows interlacing of zeros. In fact, this is a general property for orthogonal sequence of functions (Stieltjes interlacing property). Python code for the plot is as follows.

328

p3LaguerrePlots

March 26, 2021

[1]:
```
# P3 plotting
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
from numpy.polynomial.polynomial import Polynomial
```

[2]:
```
# generate polynomials
coef0 = [1]
coef1 = [-1, 1]
coef2 = [2, -4, 1]
coef3 = [-6, 18, -9, 1]
L0 = Polynomial(coef0)
L1 = Polynomial(coef1)
L2 = Polynomial(coef2)
L3 = Polynomial(coef3)
```

[5]:
```
# find zeros
L0.roots() # has no zeros
```

[5]:
```
array([], dtype=float64)
```

[6]:
```
L1.roots()
```

[6]:
```
array([1.])
```

[7]:
```
L2.roots()
```

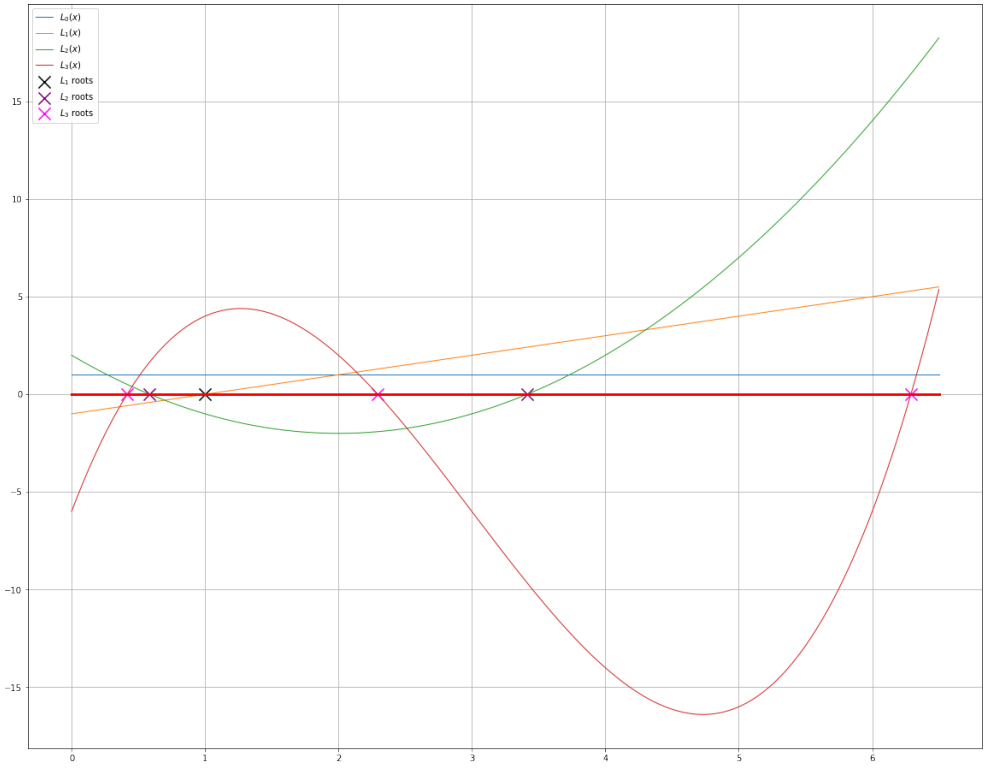
[7]:
```
array([0.58578644, 3.41421356])
```

[8]:
```
L3.roots()
```

[8]:
```
array([0.41577456, 2.29428036, 6.28994508])
```

[25]:
```
# save roots
r1 = L1.roots(); r2 = L2.roots(); r3 = L3.roots();
```

[47]:
```
# plot
X = np.linspace(0, 6.5, 1001)
plt.figure(1, figsize=(20,16));
plt.plot(X, L0(X), lw=1, label="$L_0(x)$");
plt.scatter(r1, [0], marker="x", s=200, color='black', label="$L_1$ roots");
plt.plot(X, L1(X), lw=1, label="$L_1(x)$");
plt.scatter(r2, [0]*len(r2), marker="x", s=200, color='purple', label="$L_2$␣
↪roots");
plt.plot(X, L2(X), lw=1, label="$L_2(x)$");
plt.scatter(r3, [0]*len(r3), marker="x", s=200, color='magenta', label="$L_3$␣
↪roots");
plt.plot(X, L3(X), lw=1, label="$L_3(x)$");
plt.plot(X, [0]*len(X), lw=3, color='red');
plt.grid(); plt.legend();
```



[ ]:

8.4 Padé Approximation

Determine the Padé approximation of degree 6 with $n = 2, m = 4$ to:

$$f(x) = \sin(x)$$

8.4.1

The method is outlined from Page 536 to Page 538 of Textbook.

We would like to approximate:

$$f(x) = \sin(x) = \sum_{0}^{\infty} (-1)^i \frac{x^{2i+1}}{(2i+1)!}$$

by solving for suitable coefficients such that (for $n = 2, m = 4$):

$$p(x) = p_2 x^2 + p_1 x + p_0$$

$$q(x) = q_4 x^4 + q_3 x^3 + q_2 x^2 + q_1 x + q_0$$

such that:

$$\sum_{i=0}^{k} a_i q_{k-i} = p_k, k = 0, 1, \cdots, 6$$

or:

$$(0 \cdot 1 + x + 0 \cdot x^2 - \frac{x^3}{6} + 0 \cdot x^4 + \frac{x^5}{120} + 0 \cdot x^6)(1 + q_1 x + q_2 x^2 + q_3 x^3 + q_4 x^4)$$

$$= p_0 + p_1 x + p_2 x^2 + 0 \cdot x^3 + 0 \cdot x^4 + 0 \cdot x^5 + 0 \cdot x^6$$

We would like all coefficients for up to x^6 vanish, solve:

$$x^0 : 0 = p_0$$

$$x : 1 = p_1$$

$$x^2 : q_1 = p_2$$

$$x^3 : q_2 - \frac{1}{6} = 0$$

$$x^4 : q_3 - \frac{1}{6} q_1 = 0$$

$$x^5 : q_4 - \frac{1}{6}q_2 + \frac{1}{120} = 0$$
$$x^6 : -\frac{1}{6}q_3 + \frac{1}{120}q_1 = 0$$

solving this system yields:

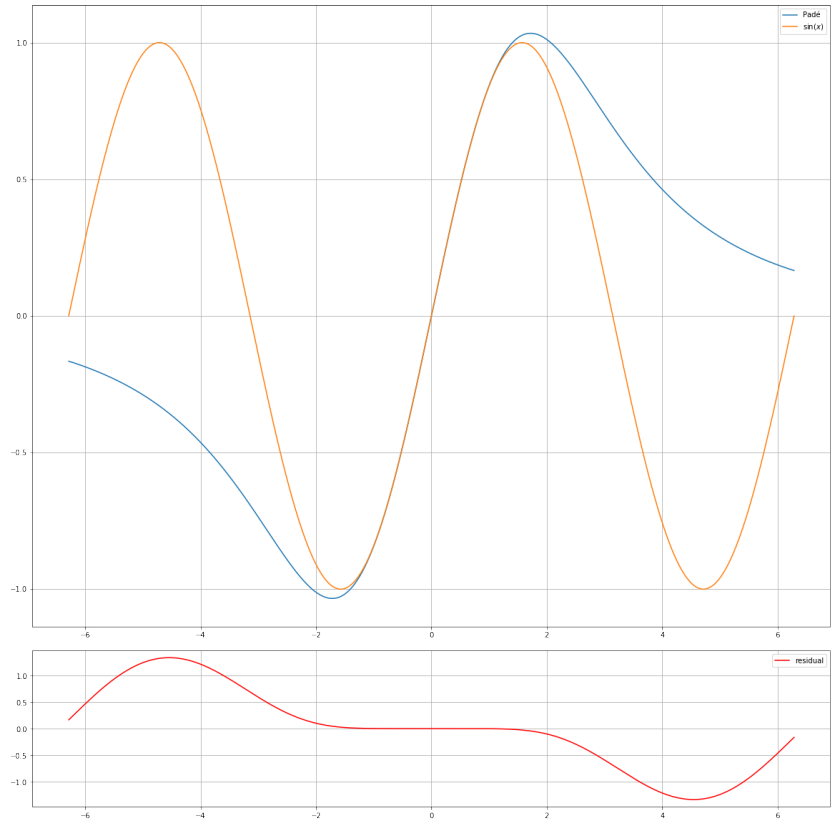
$$\begin{cases} p_0 = 0 \\ p_1 = 1 \\ p_2 = 0 \\ q_0 = 1 \\ q_1 = 0 \\ q_2 = \frac{1}{6} \\ q_3 = 0 \\ q_4 = \frac{7}{360} \end{cases}$$

$$p(x) = x, q(x) = \frac{7}{360}x^4 + \frac{1}{6}x^2 + 1$$

thus the rational approximant is:

$$r(x) = \frac{x}{\frac{7}{360}x^4 + \frac{1}{6}x^2 + 1}$$

The following page contains a plot to verify our intuition that it should approximate $f(x) = \sin(x)$ well around $x = 0$.



8.5 Cont'd Fractions for Rational Functions

Express the following rational functions as continued fractions:

$$f_1(x) = \frac{4x^2 + 3x - 7}{2x^3 + x^2 - x + 5}, f_2(x) = \frac{2x^3 + x^2 + 3x - 1}{3x^3 + x^2 - x + 1}$$

8.5.1

Continued fraction is of the form:

$$(a_0x + b_0) + \cfrac{c_1}{(a_1x + b_1) + \cfrac{c_2}{(a_2x+b_2)+\cfrac{c_3}{(a_3x+b_3)+\ddots}}}$$

whose primary use is to reduce the number of flops or express irrational numbers.

Each step of continued fraction is done by doing polynomial long division. The choice of scaling in this solution is purely for convenience of computations. Student answer is considered correct up to scaling of both numerator and denominator with suitable constants.

8.6 a

$$\frac{4x^2+3x-7}{2x^3+x^2-x+5} = \frac{1}{\frac{2x^3+x^2-x+5}{4x^2+3x-7}} = \frac{8}{\frac{16x^3+8x^2-8x+40}{4x^2+3x-7}}$$
$$= \frac{8}{(4x-1)+\frac{23x+33}{4x^2+3x-7}} = \frac{8}{(4x-1)+\frac{529}{(92x-63)-\frac{1624}{23x+33}}}$$

8.7 b

$$\frac{2x^3+x^2+3x-1}{3x^3+x^2-x+1} = \frac{1}{3}\cdot\frac{6x^3+3x^2+9x-3}{3x^3+x^2-x+1} = \frac{1}{3}\left(2+\frac{x^2+11x-5}{3x^3+x^2-x+1}\right)$$
$$= \frac{2}{3}+\frac{x^2+11x-5}{9x^3+3x^2-3x+3} = \frac{2}{3}+\frac{1}{\frac{9x^3+3x^2-3x+3}{x^2+11x-5}}$$
$$= \frac{2}{3}+\frac{1}{(9x-96)+\frac{1098x-477}{x^2+11x-5}} = \frac{2}{3}+\frac{1}{(9x-96)+\frac{1}{\frac{x^2+11x-5}{1098x-477}}}$$
$$= \frac{2}{3}+\frac{1}{(9x-96)+\frac{1098}{\frac{1098x^2+12078x-5490}{1098x-477}}} = \frac{2}{3}+\frac{1}{(9x-96)+\frac{1098}{(x+\frac{12555}{1098})-\frac{\frac{39285}{1098}}{1098x-477}}}$$

9 Homework 8

9.1 Trigonometric Approximation

Find the continuous least squares trigonometric polynomial S_n for:

$$f(x) = e^x$$

on the interval $[-\pi, \pi]$.

9.1.1

Find continuous least squares trigonometric polynomial approximation for $f(x) = e^x$ on $[-\pi, \pi]$.

Remark The root for this kind of approximation is the Stone-Wierstrass approximation, a special case shows that trigonometric polynomials are *dense* in the space of continuous functions. An intuitive understanding of *dense*, without reproducing the exact definition is to consider the analogy that $\mathbb{Q}$ is dense in $\mathbb{R}$. For any arbitrary real number, we can find an arbitrarily good approximation of that real number in $\mathbb{Q}$. Similarly, for any continuous function, we can find an arbitrarily good approximation of that function using trig polynomials.

Using Textbook notation in Sect. 8.5, the general form is:

$$S_n(x) = \frac{1}{2}a_0 + a_n \cos nx + \sum_{k=1}^{n-1}(a_k \cos kx + b_k \sin kx)$$

Doing the computations yields:

$$a_0 = \frac{1}{\pi}\int_{[-\pi,\pi]} e^x dx = \frac{1}{\pi}(e^\pi - e^{-\pi})$$

$$a_k = \frac{1}{\pi}\int_{[-\pi,\pi]} f(x)\cos kx dx = \frac{1}{\pi}\int_{[-\pi,\pi]} e^x \cos(kx) dx$$

$$= \frac{1}{\pi}\cdot \frac{e^x(k\sin(kx) + \cos(kx))}{k^2+1}\Big|_{-\pi}^{\pi} = \frac{(-1)^k \cdot k}{(k^2+1)\pi}(e^\pi - e^{-\pi})$$

because $\sin(-k\pi) = \sin(k\pi) = 0, k \in \mathbb{N}$, $\cos(k\pi) = \cos(-k\pi) = \pm 1$.

$$b_k = \frac{1}{\pi}\int_{[-\pi,\pi]} f(x)\sin kx dx = \frac{1}{\pi}\int_{[-\pi,\pi]} e^x \sin kx dx = \frac{1}{\pi}\left(\frac{e^x(\sin(kx) - k\cos(kx))}{k^2+1}\Big|_{-\pi}^{\pi}\right)$$

$$= \frac{(-1)^k \cdot k}{(k^2+1)\pi}(e^{-\pi} - e^{\pi})$$

9.2 Degree-4 Discrete Least Squares Trig Approximation

Determine the discrete least squares trigonometric polynomial S_4 for $f(x) = e^x$ on $[-\pi, \pi]$, and compute the error $E_4(x)$ of this approximation.

9.2.1

Theorem 8.13 in Textbook gives the explicit formula for computing the approximation. The Python code below is used to generate and plot these polynomials.

With regression methods, m controls the number of data points we are fitting on. For a fixed n, increasing m will keep decreasing error for some time. If m is too large, however, we start to see *overfitting*. The code file is also uploaded to Resources if you would like to run it with different parameter choices.

The coefficients are given explicitly:

$$a_k = \sum_{0}^{2m-1} y_j \cos(kx_j), k \in \{0, 1, 2, \cdots, n\}$$

$$b_k = \sum_{0}^{2m-1} y_j \sin(kx_j), k \in \{0, 1, 2, \cdots, n-1\}$$

The coefficients for $n = 4, m = 2n - 1 = 7$ found using the code on the following page are:

```
a_coef = [ 5.82532985, -2.15048477,
-0.05134437,  0.77977417, -1.07202048]

b_coef = [ 3.55350809, -2.69329961,  1.82801647]
```

With total squared error: 42.3402344921022.

379

Untitled

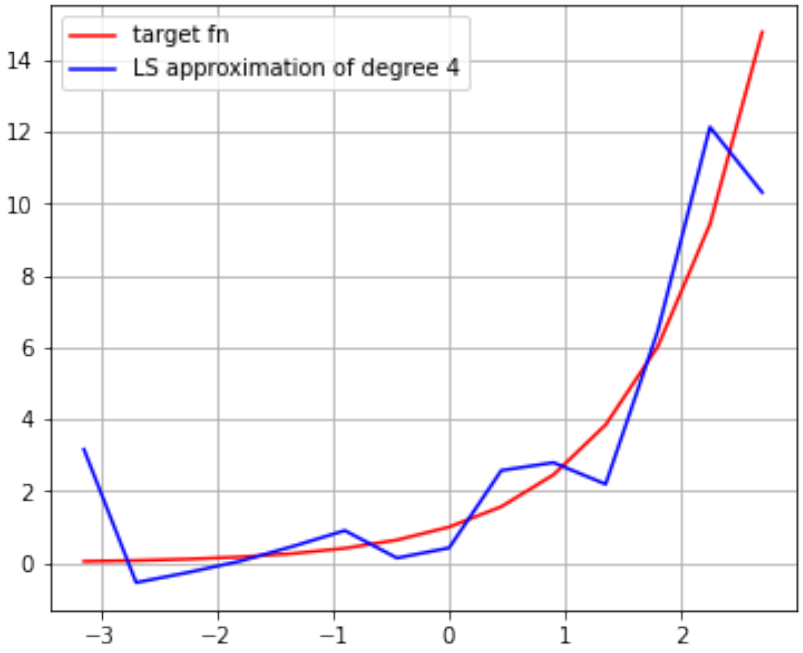
April 25, 2021

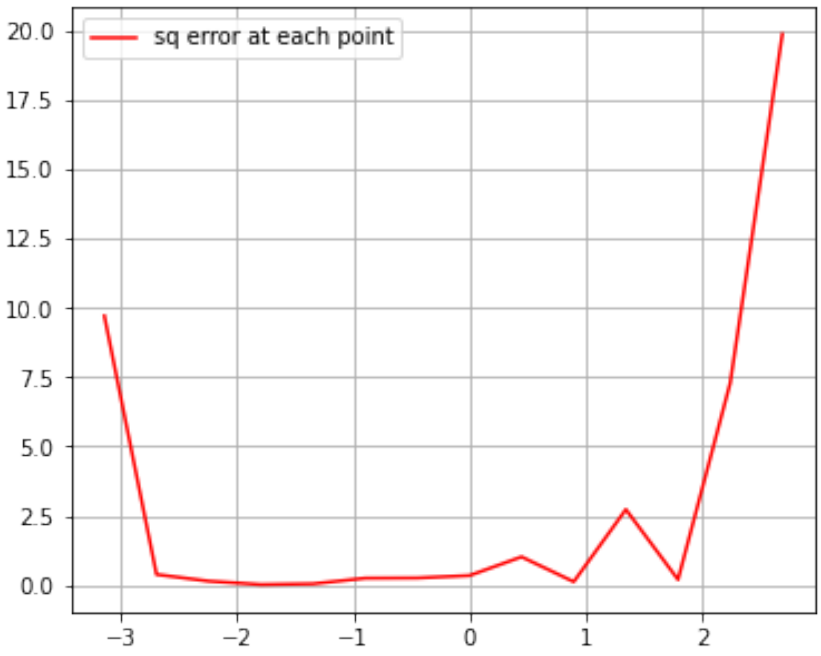
[140]:
```
# Prob 2, HW8
import numpy as np
import scipy as sp
import matplotlib.pyplot as plt
%matplotlib inline
# interactive plot
from ipywidgets import interact, interactive, fixed, interact_manual
import ipywidgets as widgets

##
def compute_coefs(target_fn=np.exp, n=4, m=None):
    """ compute coefficients a_k, b_k, stored
    as vectors, via Thm 8.13 in Burden/Faires.
    For a degree n of Discrete Trig Polynomial
    Approximation to target_fn, using m queries.
    """
    if m is None:
        # textbook's choice, also seems to be a popular choice
        # in student solutions
        m = 2 * n - 1
    left = -np.pi
    right = np.pi
    # assign grid
    grid = np.linspace(left, right, 2*m+1)
    # query indices
    idx = np.arange(2*m+1)
    assert grid[0] == left and grid[::-1][0] == right
    query_points = target_fn(grid)
    # get coefs
    # preallocate
    # a1, a2, ... an-1 (a0, an computed separately)
    a_coef = np.array([(1/m)*np.sum([np.multiply(query_points[0:2*m], np.
↪cos((k)*grid[0:2*m]))]) for
                       k in np.arange(n-1)+1])
    # b1, b2, ... bn-1
    b_coef = np.array([(1/m)*np.sum([np.multiply(query_points, np.
↪sin((k)*grid))]) for k in np.arange(n-1)+1])
```

```
    a0 = (1/m) * np.sum(query_points[0:2*m])
    an = (1/m) * np.sum(np.multiply(query_points[0:2*m], np.cos(n * grid[0:
↪2*m])))
    a_coef = np.append(a0, a_coef)
    a_coef = np.append(a_coef, an)
    return grid, a_coef, b_coef

def approximate(target_fn=np.exp, n=4, m=None, verbose=True):
    """ uses a degree n discrete least sq. trig polynomial
    to approximate target_fn, with 2*m queries. Returns sq error
    on each query point.

    If verbose, plot approximate and error.
    """
    # find coefs
    grid, a_coef, b_coef = compute_coefs(target_fn, n, m)
    a_coef_short = a_coef[1:len(a_coef)-1] # a1, a2, ... an-1 only
    assert len(a_coef_short) == len(b_coef)
    # assemble polynomial
    S_n = lambda x: (a_coef[0]/2) + (a_coef[::-1][0]*np.cos(n*x)) + \
                np.sum([a_coef_short[k]*np.cos((k+1)*x) + \
                        b_coef[k]*np.sin((k+1)*x) for k in np.arange(n-1)])
    # evaluate at gri point
    approximated = np.array([S_n(x_j) for x_j in grid[0:len(grid)-1]])
    # compute final sq error
    query = target_fn(grid)[0:len(grid)-1]
    plot_grid = grid[0:len(grid)-1]
    sq_error = np.sum((query - approximated)**2)
    if verbose:
        # plot
        plt.figure(1, figsize=(6,5));
        plt.plot(plot_grid, query, color='red', \
                 label="target fn");
        plt.plot(plot_grid, approximated, color='blue', \
                 label='LS approximation of degree {}'.format(n));
        plt.grid(); plt.legend();
        # plot error
        plt.figure(2, figsize=(6,5));
        plt.plot(plot_grid, (query-approximated)**2, color='red', \
                 label="sq error at each point");
        plt.grid(); plt.legend();
    return sq_error
error = approximate(n=4, verbose=True)
print("====== total squared error = {}".format(error))
```

```
====== total squared error = 42.34023449210218
```

381


target fn
LS approximation of degree 4
14
12
10
8
6
4
2
0
−3
−2
−1
0
1
2


sq error at each point
20.0
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0
−3
−2
−1
0
1
2

382

[148]:
```
interact(lambda n, m: print("==== using n = {}, m = {}, final sq. error = ".
↪format(n, m), \
                            approximate(np.exp, n, m, verbose=True)), \
        n=(1,20), m=(2,100));
```

```
interactive(children=(IntSlider(value=10, description='n', max=20, min=1),␣
↪IntSlider(value=51, description='m'…
```

[ ]:

9.3 Trigonometric interpolating polynomial

Determine the trigonometric interpolating polynomial of degree 4 for $f(x) = x(\pi - x)$ on $[-\pi, \pi]$ using direct calculation and FFT, respectively.

9.3.1 direct calculation

Direct calculation used the Python code from Problem 2, with:

$$f(x) = -x^2 + \pi x, x \in [-\pi, \pi]$$

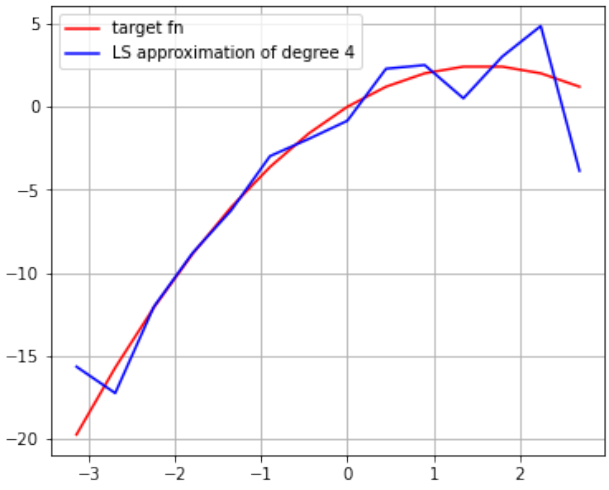
To replicate the results, open

`hw8_p2_slider.ipynb`

and run:

```
approximate(target_fn=lambda x: x*(np.pi-x), n=4, verbose=True)
```

Figure 2: 4-th Degree approximation of $x(\pi - x)$



9.3.2 Fast Fourier Transform

The illustration in Section 8.6 and Algorithm 8.3 provide an explanation of the implementation. For a demo, we use $8 = 2 \cdot 2^2$ data points on $[-\pi, \pi]$, namely:

$$x_j = -\pi + \frac{j\pi}{4}, j = 0, 1, 2, \cdots, 7$$

We seek the approximation:

$$S_4(x) = \frac{1}{2}a_0 + \frac{1}{2}a_4\cos(4x) + \sum_{k=1}^{3}(a_k\cos(kx) + b_k\sin(kx))$$

By formula, the FFT is defined as:

$$\frac{1}{4}\sum_{j=0}^{7} c_k e^{ikx}$$

where:

$$c_k = \sum_{j=0}^{7} y_j e^{ik\pi j/4}$$

Then direct calculation gives:

$$c_0 = \sum_{0}^{7} y_i$$

$$c_1 = y_0 + (\frac{i+1}{\sqrt{2}})y_1 + iy_2 + (\frac{i-1}{\sqrt{2}})y_3 - y_4 - (\frac{i+1}{\sqrt{2}})y_5 - iy_6 - (\frac{i-1}{\sqrt{2}})y_7$$

$$c_2 = y_0 + iy_1 - y_2 - iy_3 + y_4 + iy_5 - y_6 - iy_7$$

$$c_3 = y_0 + (\frac{i-1}{\sqrt{2}})y_1 - iy_2 + (\frac{i+1}{\sqrt{2}})y_3 - y_4 - (\frac{i-1}{\sqrt{2}})y_5 + iy_6 - (\frac{i+1}{\sqrt{2}})y_7$$

$$c_4 = y_0 - y_1 + y_2 - y_3 + y_4 - y_5 + y_6 - y_7$$

$$c_5 = y_0 - (\frac{i+1}{\sqrt{2}})y_1 + iy_2 - (\frac{i-1}{\sqrt{2}})y_3 - y_4 + (\frac{i+1}{\sqrt{2}})y_5 - iy_6 + (\frac{i-1}{\sqrt{2}})y_7$$

$$c_6 = y_0 - iy_1 - y_2 + iy_3 + y_4 - iy_5 - y_6 + iy_7$$

$$c_7 = y_0 - (\frac{i-1}{\sqrt{2}})y_1 - iy_2 - (\frac{i+1}{\sqrt{2}})y_3 - y_4 + (\frac{i-1}{\sqrt{2}})y_5 + iy_6 + (\frac{i+1}{\sqrt{2}})y_7$$

The above expressions are from Page 559 of textbook. Page 560 has a further explanation of reducing flops. If we implement a double for loop to compute the c_k, there is no advantage; the written part is only a demonstration.

With the help of a calculator we obtain (rounded to 2 decimals):

$$c_0 = -37.01, c_1 = -26.72 - 23.83i, c_2 = -14.8 - 13.57i$$

$$c_3 = -12.76 - 4.09i, c_4 = -12.34, c_5 = -12.76 + 4.09i$$

$$c_6 = -14.8 + 9.87i, c_7 = -26.72 + 23.83i$$

Now we can set up the coefficients as:

$$a_k = \text{Re}(\frac{1}{4}c_k e^{-ik\pi})$$

$$b_k = \text{Im}(\frac{1}{4}c_k e^{-ik\pi})$$

for $k = 0, 1, 2, 3, 4$.

Now we recover:

$$a_0 = -9.25275413, a_1 = 6.67951825, a_2 = -3.70110165$$

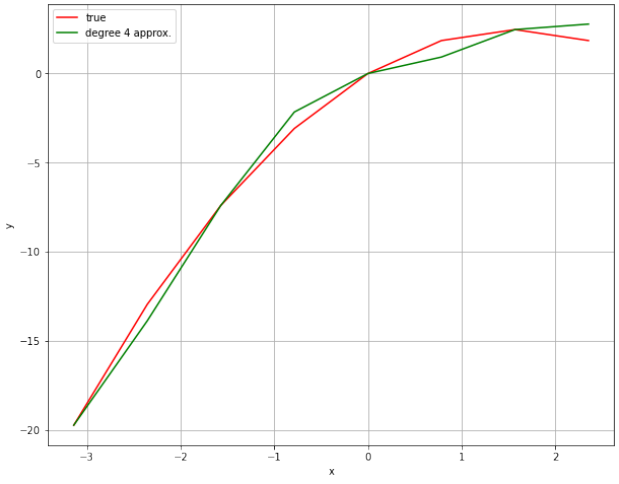
$$a_3 = 3.19008615, a_4 = -3.08425138$$

$$b_0 = 0, b_1 = 5.9568332, b_2 = -3.39267651$$

$$b_3 = 1.022031, b_4 = 0$$

The following is a plot of the approximation using these coefficients.

Figure 3: 4-th Degree approximation of $x(\pi - x)$ via FFT



9.4 Equivalent Matrix Vector Multiply

Show that $c_0, \cdots, c_{2m-1}$ in B& F Algorithm 8.3 is given by:

$$\begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ \vdots \\ c_{2m-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \cdots & \omega^{2m-1} \\ 1 & \omega^2 & \omega^4 & \cdots & \omega^{4m-2} \\ \vdots & \cdots & \cdots & \ddots & \vdots \\ 1 & \omega^{2m-1} & \omega^{4m-2} & \cdots & \omega^{(2m-1)^2} \end{pmatrix} \cdot \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{2m-1} \end{pmatrix}$$

where $\omega = e^{\pi i/m}$. Also, find the eigenvalues of this matrix.

9.4.1 Matrix Vector Equivalence

Refer to equation (8.28) in Textbook:

$$c_k = \sum_{j=0}^{2m-1} y_j e^{ik\pi j/m}, k = 0, 1, 2, \cdots, 2m-1$$

Now substitute in $\omega = e^{i\pi/m}$, then:

$$c_k = \sum_{j=0}^{2m-1} y_j \omega^{jk}$$

For each $k \in \{0, 1, 2, \cdots, 2m-1\}$, this can be interpreted as an inner product:

$$c_k = \begin{pmatrix} 1 & \omega^k & \omega^{2k} \cdots \omega^{(2m-1)k} \end{pmatrix} \cdot \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{2m-1} \end{pmatrix}$$

Making this substitution for every k yields the desired matrix-vector product, of dimension $(2m)^2$.

9.4.2 Eigenvalues

Let:

$$A = \begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \cdots & \omega^{2m-1} \\ 1 & \omega^2 & \omega^4 & \cdots & \omega^{2(2m-1)} \\ \vdots & \ddots & \cdots & \vdots & \vdots \\ 1 & \omega^{2m-1} & \omega^{(2m-1)2} & \cdots & \omega^{(2m-1)(2m-1)} \end{pmatrix}$$

A is the DFT matrix (the discrete Fourier transform, represented as a linear transformation; FFT refers to a computation of DFT that saves flops, hence "fast"), refer to this paper to explore its properties and eigen-structure.

The matrix:

$$U = \frac{1}{\sqrt{2m}} A$$

can be proved to be a unitary matrix. Without the normalization factor $1/\sqrt{2m}$, we can show that the columns of A are orthogonal.

Comment: The proof of orthogonality is not required. If we did decide to write a proof, we need to be careful with the dot product because columns of A are complex numbers:

$$x \cdot y = \sum_{i=0}^{2m-1} x_i \overline{y_i}$$

Since U is unitary, its determinant is 1. The eigenvalues are $\pm 1, \pm i$. This means that the eigenvalues of A are:

$$\pm\sqrt{2m}, \pm i\sqrt{2m}$$

10 Homework 9

10.1 Fixed-Point Iteration for Nonlinear System

Use fixed point iteration to find all solutions to the nonlinear system outlined below, to accuracy within $O(10^{-5})$, and use l_∞-norm.

413 **10.1.1**

The system:

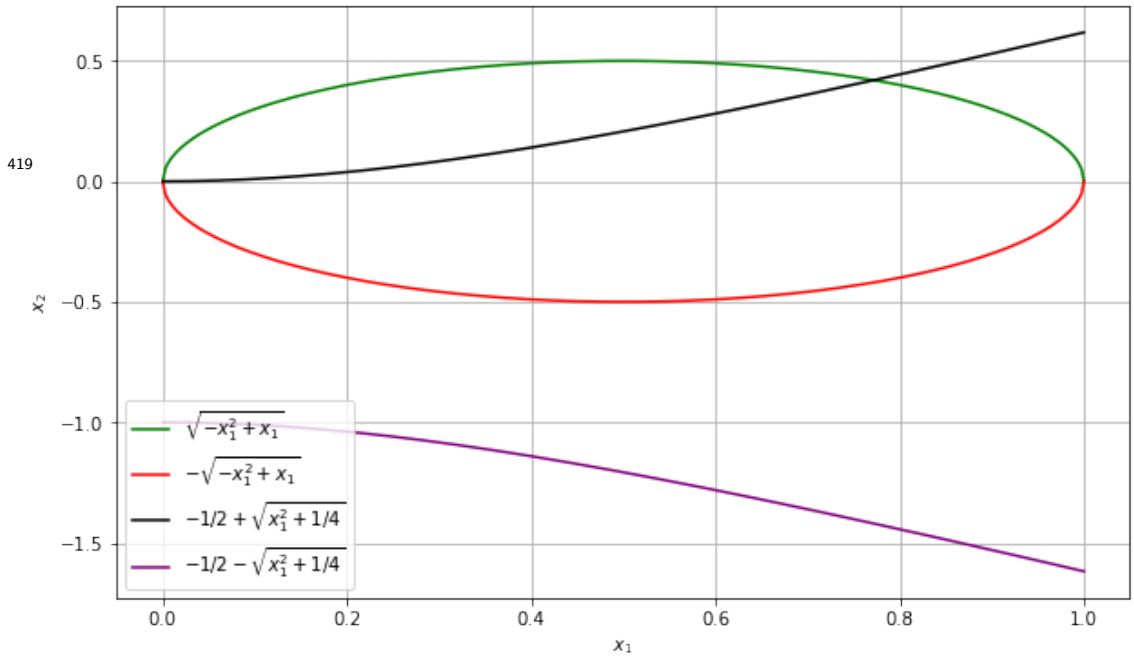
$$x_1^2 + x_2^2 - x_1 = 0$$
$$x_1^2 - x_2^2 - x_2 = 0$$

414 has 2 real solutions.

By rearranging:

$$x_2 = \pm\sqrt{-x_1^2 + x_1}$$
$$x_2 = -\frac{1}{2} \pm \sqrt{x_1^2 + \frac{1}{4}}$$

415 Then we can plot x_2 against x_1 and look at the intersections, as shown
416 below (we need to restrict the domain to $x_1 \in [0, 1]$, otherwise, complex
417 solutions will arise for the circle part, but we are operating in $\mathbb{R}^2$; having
418 more than 2 solutions is not incorrect, only the 2 real solutions are graded):



[370]:
```python
# p1 fixed point theorem
def rhs(x):
    """ right hand side, takes in a vector of dimension 2 x 1.
    returns 2 x 1 output G(x).
    """
    assert x.shape[0] == 2 and x.shape[1] == 1
    x1, x2 = x[0], x[1]
    G1 = np.sqrt(x2**2+x2)
    G2 = np.sqrt(-x1**2+x1)
    Gx = np.array([G1, G2])
    Gx = Gx.reshape(len(x), -1)
    assert Gx.shape[0] == 2 and Gx.shape[1] == 1
    return Gx

def fixed_point_iteration(G, x0=None, tol=1e-5):
    """ fixed point iteration. G is right hand side, with
    random starting point (by default) drawn from [0, 1]x[-1/2, 1/2],
    convergence criteria is inf norm. """
    if x0 is None:
        x0 = np.random.uniform([0, -1/2], [1, 1/2], size=(1, 2))
    x0 = np.array(x0)
    # reshape
    x0 = x0.reshape(-1, 1)
    n = x0.shape[0]
    # history
```

```
    all_x = []
    all_x.append(x0)
    x_prev = 0
    x_curr = x0
420 while np.linalg.norm(x_curr - x_prev, ord=float('inf')) > tol:
        x_prev = x_curr
        x_curr = G(x_curr)
        all_x.append(x_curr)
    len_history = len(all_x)
    history = np.array(all_x).reshape(-1, len_history)
    last_x = history[:, -1]
    return last_x, history
```

[371]:
```
last, _ = fixed_point_iteration(rhs) # fixed point 1
print(last)
```

```
[0.7726402  0.41964714]
```

[ ]:

[ ]:

[ ]:

[ ]:

Another solution is found by setting $\mathbf{x_0} = (\epsilon, \epsilon)^T$ where $\epsilon \approx 0$; this will yield $\mathbf{x} = (0, 0)^T$.

Rearrange and solve for x_1, x_2, using the appropriate branch of the x_1 circle and the x_2 parabola:

$$x_1 = \sqrt{x_2^2 + x_2}$$

$$x_2 = \sqrt{-x_1^2 + x_1}$$

Now let $D = \{(x_1, x_2) : x_1 \in [0, 1], x_2 \in [-1/2, 1/2]\}$ be a rectangle, where our solutions lie:

$$\mathbf{G}(\mathbf{x}) = \begin{pmatrix} \sqrt{x_2^2 + x_2} \\ \sqrt{-x_1^2 + x_1} \end{pmatrix}$$

then compute the Jacobian matrix:

$$\mathbf{J} = \begin{pmatrix} \frac{\partial G_1}{\partial x_1} & \frac{\partial G_1}{\partial x_2} \\ \frac{\partial G_2}{\partial x_1} & \frac{\partial G_2}{\partial x_2} \end{pmatrix} = \begin{pmatrix} 0 & \frac{1}{2}(x_2^2 + x_2)^{-1/2} \cdot (2x_2 + 1) \\ -\frac{1}{2}(-x_1^2 + x_1)^{-1/2} \cdot (2x_1 - 1) & 0 \end{pmatrix}$$

due to the $-1/2$ power, we see that it is not bounded by some constant less than 1 in the original D, however this can be mitigated and solved by restricting the domain away from $x_1 = 0$; in the smaller domain, the fixed point is unique and independent of starting points

Comment: many students concluded at this point that there does not exist any solutions / the fixed point iteration does not converge. Theorem 10.6 has two parts. One, a fixed point *exists* if $\mathbf{G}$ is a continuous, injective map into D. Part two is a stronger condition about whether we will always converge to a unique fixed point regardless of choice of starting points.

10.2 Continuous Differentiability

Let A be $\mathbb{R}^{n\times n}$ and define the operator:

$$F : \mathbb{R}^n \to \mathbb{R}^n$$

with:

$$F(x) = Ax$$

Show that F is continuously differentiable on $\mathbb{R}^n$. Find the Jacobian of F at an arbitrary point $x \in \mathbb{R}^n$.

10.2.1

F is a linear mapping:

$$F : \mathbb{R}^n \to \mathbb{R}^n$$

with $F(x) = Ax$ for A a constant $\mathbb{R}^{n\times n}$ matrix. We need to show two parts. F is differentiable, and the partial derivatives are continuous.

Recall definition of differentiability in $\mathbb{R}^n \supset U \to \mathbb{R}^n$. F is said to be *differentiable* if there exists some linear mapping D_f such that:

$$\lim_{\|h\|\to 0} \frac{\|F(\mathbf{x_0}+h) - F(\mathbf{x_0}) - L(h)\|}{\|h\|} = 0$$

Refer to standard texts in Advanced Calculus for more details. Most often in proofs, we do not know what D_f is *a priori*. We must resort to some way of guessing. One result from calculus gives:

If all partial derivatives exist and are continuous at $\mathbf{x_0} \in U \subset \mathbb{R}^n$, then F is differentiable at $\mathbf{x_0}$, and the matrix representation of L is uniquely given by the Jacobian (Munkres).

Therefore one good starting point is computing the Jacobian. Explicitly we can write:

$$F(\mathbf{x}) = A\mathbf{x} = \begin{pmatrix} \sum_{j=1}^n A_{1j}x_j \\ \sum_{j=1}^n A_{2j}x_j \\ \vdots \\ \sum_{j=1}^n A_{nj}x_j \end{pmatrix}$$

then:

$$\frac{\partial \mathbf{F}_i}{\partial x_j} = A_{ij}$$

since each entry of $\mathbf{F} = F(\mathbf{x})$ is a linear combination of all entries in $\mathbf{x} = (x_1, x_2, \cdots, x_n)^T$, only x_j will contribute to the derivative. Furthermore, linear and constant functions in x_j are continuous.

Then we have found a candidate, if we let $L(h) = Ah$, and now we have for all $\epsilon > 0$, let $\|h\| < \delta = \epsilon$, we conclude:

$$\frac{\|F(\mathbf{x_0} + h) - F(\mathbf{x_0}) - Ah\|}{\|h\|} = \frac{\|A\mathbf{x_0} + Ah - A\mathbf{x_0} - Ah\|}{\|h\|} = 0 < \epsilon$$

We conclude that F is continuously differentiable, and the Jacobian is A. The Jacobian is the matrix representation of directional derivatives for all $\mathbf{x} \in U$ (the converse is not true, namely, all directional derivatives existing does not imply differentiability).

10.3 Newton's Method for Nonlinear System

Solve the following outline nonlinear system using Newton's Method, compute the first 2 iterations using $x^{(0)} = 0$.

10.3.1

The nonlinear system at hand is:

$$\begin{cases} x_1^2 + x_2 - 37 = 0 \\ x_1 - x_2^2 - 5 = 0 \\ x_1 + x_2 + x_3 - 3 = 0 \end{cases}$$

Newton's method is introduced in Section 10.2 of Textbook, where we use the following iteration:

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \mathbf{J}(\mathbf{x}^{(k)})^{-1} \cdot \mathbf{F}(\mathbf{x}^{(k)})$$

where $\mathbf{J}$ is the Jacobian matrix, $\mathbf{F}$ comes from the system:

$$\mathbf{F}(\mathbf{x}) = \mathbf{0}$$

Below is an implementation of Newton's method in Julia.

Comment: Many students approached the implementation of $\mathbf{J}^{-1}$ by directly inverting the Jacobian matrix. In realistic settings, direct inversion

tends to be avoided because of the cost (this Wikipedia page gives a reminder). A more preferred implementation (also used in Algorithm 10.1 of Textbook) is by solving the linear system for $\mathbf{y}$:

$$\mathbf{J}(\mathbf{x}^{(k)})(\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}) = \mathbf{J}(\mathbf{x}^{(k)})\mathbf{y} = -\mathbf{F}(\mathbf{x}^{(k)})$$

456 Oftentimes we do not have access to $\mathbf{J}$ in closed form. Numerical Jacobian
457 is usually found using finite differencing.

Here we have:

$$\mathbf{F}(\mathbf{x}) = \begin{pmatrix} x_1^2 + x_2 - 37 \\ x_1 - x_2^2 - 5 \\ x_1 + x_2 + x_3 - 3 \end{pmatrix}$$

and the Jacobian:

$$\mathbf{J}(\mathbf{x}) = \begin{pmatrix} 2x_1 & 1 & 0 \\ 1 & -2x_2 & 0 \\ 1 & 1 & 1 \end{pmatrix}$$

458 as expected from what we may know in Math 128A (Newton's method for
459 1D); this method converges roughly in second-order.

hw9_p3_newtons

April 28, 2021

[190]:
```
# P3 Newton's Method
using LinearAlgebra
function newtons(x0, F, J, random_start, tol=1e-10, maxit=100, verbose=false)
    #==
    Implementation of Newton's method assuming direct access to J
    ==#
    if random_start == true
        # generate random start
        x0 = rand(Float64, (length(x0), 1));
    end
    N = length(x0);
    all_x = zeros(N, maxit);
    all_x[:, 1] = x0;
    num_it = 1;
    all_err = [];
    x_curr = x0;
    while (num_it <= maxit)
        i = num_it;
        x_prev = x_curr;
        # solve for y
        y = -J(x_curr)\F(x_curr);
        # update x_curr
        x_curr = x_curr + y;
        # store history
        all_x[:, i] = x_curr;
        # check convergence
        err = norm(y);
        all_err = append!(all_err, err);
        if err < tol
            println("===== converged");
            break
        end
        num_it += 1;
    end
    return all_x[:, 1:num_it-1], all_err
end
```

```
# P3 functions
function F(x)
    #== helper function, takes in x of size 3 by 1==#
    x1 = x[1];
    x2 = x[2];
    x3 = x[3];
    return [x1^2+x2-37; x1-x2^2-5; sum(x)-3];
end

function J(x)
    #== helper function, takes in x of size 3 by 1 and
    computes Jacobian of size 3 by 3 ==#
    x1, x2, x3 = x;
    N = length(x);
    Jx = zeros(N, N);
    Jx[1, 1] = 2*x1;
    Jx[1, 2] = 1;
    Jx[2, 1] = 1;
    Jx[2, 2] = -2*x2;
    Jx[3, 1] = 1;
    Jx[3, 2] = 1;
    Jx[3, 3] = 1;
    return Jx;
end
```

[190]:
```
J (generic function with 1 method)
```

[191]:
```
x0 = [100;1000;13210];
all_x, all_err = newtons(x0, F, J, true, 1e-10);
```

```
===== converged
```

[192]:
```
all_x
```

[192]:
```
3×10 Matrix{Float64}:
  49.2311   24.8454   13.0171    7.77926  …   5.99749   5.99996   6.0   6.0
  28.0316   14.3698    7.46384   3.9181       1.03049   1.00043   1.0   1.0
 -74.2627  -36.2152  -17.4809   -8.69736     -4.02798  -4.0004   -4.0  -4.0
```

[193]:
```
all_err
```

[193]:
```
11-element Vector{Any}:
 93.51644466724949
 47.21145333109397
 23.207141626254703
 10.823973300783608
  4.240792760066012
  1.2817266334696407
```

0.3461529384813618
0.040870629089773
0.0005898078630249815
1.2286133417257642e-7
4.8255478180902664e-15

10.4 Nonlinear System with 6 Solutions

Solve the following outlined nonlinear system using Broyden's method. Show that the solution is symmetric around (x_1, x_2, x_3) hyperplane.

10.4.1

We have the system:

$$\begin{cases} 4x_1 - x_2 + x_3 = x_1x_4 \\ -x_1 + 3x_2 - 2x_3 = x_2x_4 \\ x_1 - 2x_2 + 3x_3 = x_3x_4 \\ x_1^2 + x_2^2 + x_3^2 = 1 \end{cases}$$

10.4.2 Symmetry

To avoid confusion with what is known and what is unknown (the dummy variables in the system of equations), let's rename our solution to be $(a_1, a_2, a_3, a_4)^T$ which are all constants (known).

Plug in $(-a_1, -a_2, -a_3, a_4)$, we verify all equations (notice and make sense of the sign changes):

$$-4a_1 + a_2 - a_3 = -(4a_1 - a_2 + a_3) = -a_1a_4 = -(a_1)a_4$$

$$-(-a_1)+3(-a_2)-2(-a_3) = a_1-3a_2+2a_3 = -(-a_1+3a_2-2a_3) = -(a_2a_4) = (-a_2)a_4$$

$$(-a_1)-2(-a_2)+3(-a_3) = -a_1+2a_2-3a_3 = -(a_1-2a_2+3a_3) = -(a_3a_4) = (-a_3)a_4$$

$$(-a_1)^2 + (-a_2)^2 + (-a_3)^2 = a_1^2 + a_2^2 + a_3^2 = 1$$

therefore we see that it is indeed a solution.

10.4.3 Broyden's method

We only need to apply Broyden's method 3 times because the reflection around the x_4-axis is also a solution.

A Julia implementation of Algorithm 10.2 (Broyden's method) is as follows, with:

$$\mathbf{F}(\mathbf{x}) = \begin{pmatrix} 4x_1 - x_2 + x_3 - x_1x_4 \\ -x_1 + 3x_2 - 2x_3 - x_2x_4 \\ x_1 - 2x_2 + 3x_3 - x_3x_4 \\ x_1^2 + x_2^2 + x_3^2 - 1 \end{pmatrix}$$

and the Jacobian:

$$\mathbf{J}(\mathbf{x}) = \begin{pmatrix} 4 - x_4 & -1 & 1 & -x_1 \\ -1 & 3 - x_4 & -2 & -x_2 \\ 1 & -2 & 3 - x_4 & -x_3 \\ 2x_1 & 2x_2 & 2x_3 & 0 \end{pmatrix}$$

The solutions are on the following page, negating x_1, x_2, x_3 of each solutions yield new solutions.

In [28]:
```
# P4, implementation of Broyden's method
using LinearAlgebra

function broydens(x0, F, J, random_start, tol=1e-8, maxit=100)
    #== Alg. 10.2 in Text, assuming access to J. ==#
    if random_start == true
        # generate random start
        x0 = rand(Float64, (length(x0), 1));
    end
    N = length(x0);
    all_x = zeros(N, maxit);
    all_x[:, 1] = x0;
    all_err = [];
    x_curr = x0;
    v = F(x_curr);
    A = inv(J(x_curr));
    s = -A*v;
    x_curr = x_curr + s;
    num_it = 2;
    all_x[:, 2] = x_curr;
    while (num_it <= maxit)
        w = v;
        v = F(x_curr);
        y = v - w;
        z = -A*y;
        p = -transpose(s)*z;
        u = transpose(transpose(s)*A);
        A = A + (1/p)*(s+z)*transpose(u);
        s = -A*v;
        # update x
        x_curr = x_curr + s;
        # save x
        num_it += 1;
        i = num_it;
        all_x[:, i] = x_curr;
        err = norm(s);
        all_err = append!(all_err, err);
        if err < tol
            println("===== converged. ");
            all_err = append!(all_err, err);
            break
        end
    end
    return all_x[:, 1:num_it], all_err
end

# ==== functions in Problem 4
function F(x)
    #== input: 4x1 vector ==#
    x1, x2, x3, x4 = x;
    return [4*x1-x2+x3-x1*x4; -x1+3*x2-2*x3-x2*x4; x1-2*x2+3*x3-x3*x4; x1^2+x2^2
end

function J(x)
    #=== Jacobian: 4 x 4, input 4 x 1. ===#
    x1, x2, x3, x4 = x;
    Jx = zeros(4, 4);
    Jx[1, 1] = 4-x4;
    Jx[1, 2] = -1;
```

```
    Jx[1, 3] = 1;
    Jx[1, 4] = -x1;
    Jx[2, 1] = -1;
    Jx[2, 2] = 3-x4;
    Jx[2, 3] = -2;
    Jx[2, 4] = -x2;
    Jx[3, 1] = 1;
    Jx[3, 2] = -2;
    Jx[3, 3] = 3-x4;
    Jx[3, 4] = -x3;
    Jx[4, 1] = 2*x1;
    Jx[4, 2] = 2*x2;
    Jx[4, 3] = 2*x3;
    Jx[4, 4] = 0;
    return Jx;
end
```

Out[28]:
```
J (generic function with 1 method)
```

In [29]:
```
x_history, errors = broydens([1;1;1;1], F, J, true);
```

```
===== converged.
```

In [30]:
```
x_history
```

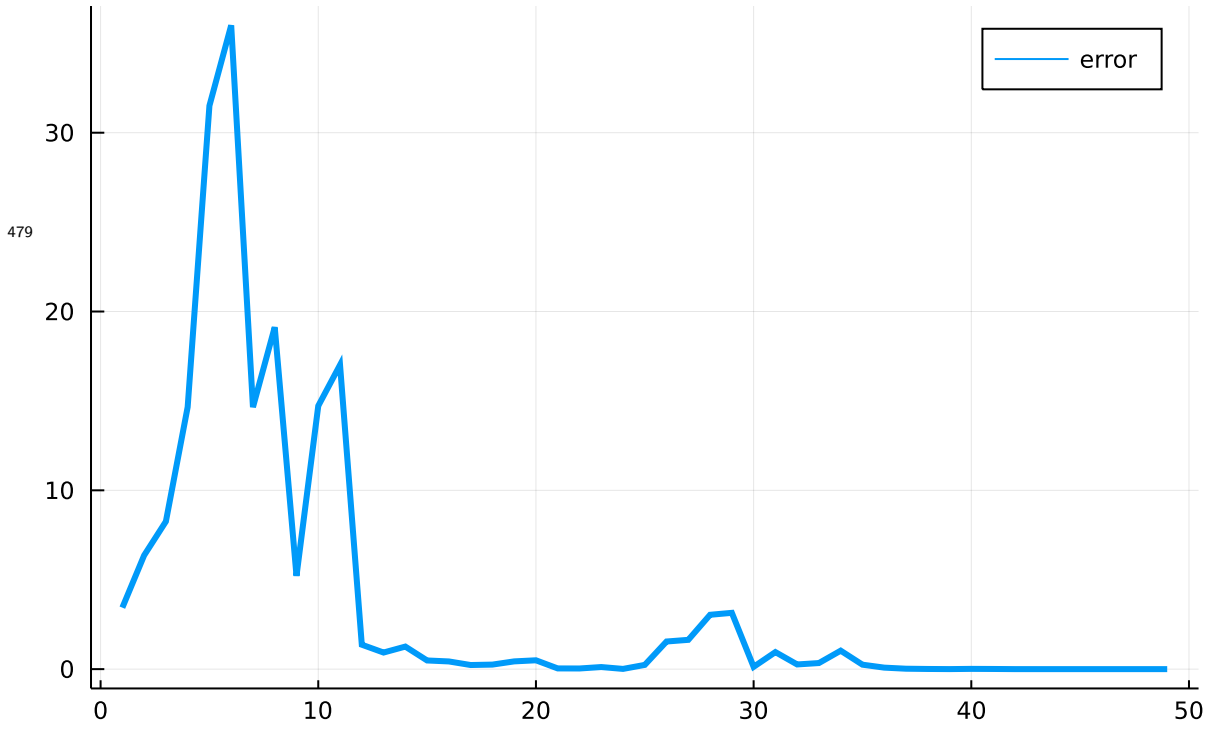
Out[30]:
```
4×50 Matrix{Float64}:
 0.625478    1.08672     0.898825  -1.1153   …  -5.58668e-11   1.54809e-14
 0.0375685   0.455422    1.00727    2.88394      -0.707107     -0.707107
 0.0267659  -0.00577733  0.385281   3.34746      -0.707107     -0.707107
 0.147917    6.10327     2.73689   -2.17375       1.0           1.0
```

In [31]:
```
last_sol = x_history[:,end]
```

Out[31]:
```
4-element Vector{Float64}:
  1.5480908578395253e-14
 -0.7071067811865902
 -0.7071067811866502
  1.0000000000003826
```

In [45]:
```
using Plots
plot(1:length(errors), errors, title="Error Plot (Broydens)",
label="error", lw=3)
```

Out[45]:



In [60]:
```
# sol2
x_history2, errors2 = broydens([1;0;0;0], F, J, false);
```

```
===== converged.
```

In [61]:
```
x_history2
```

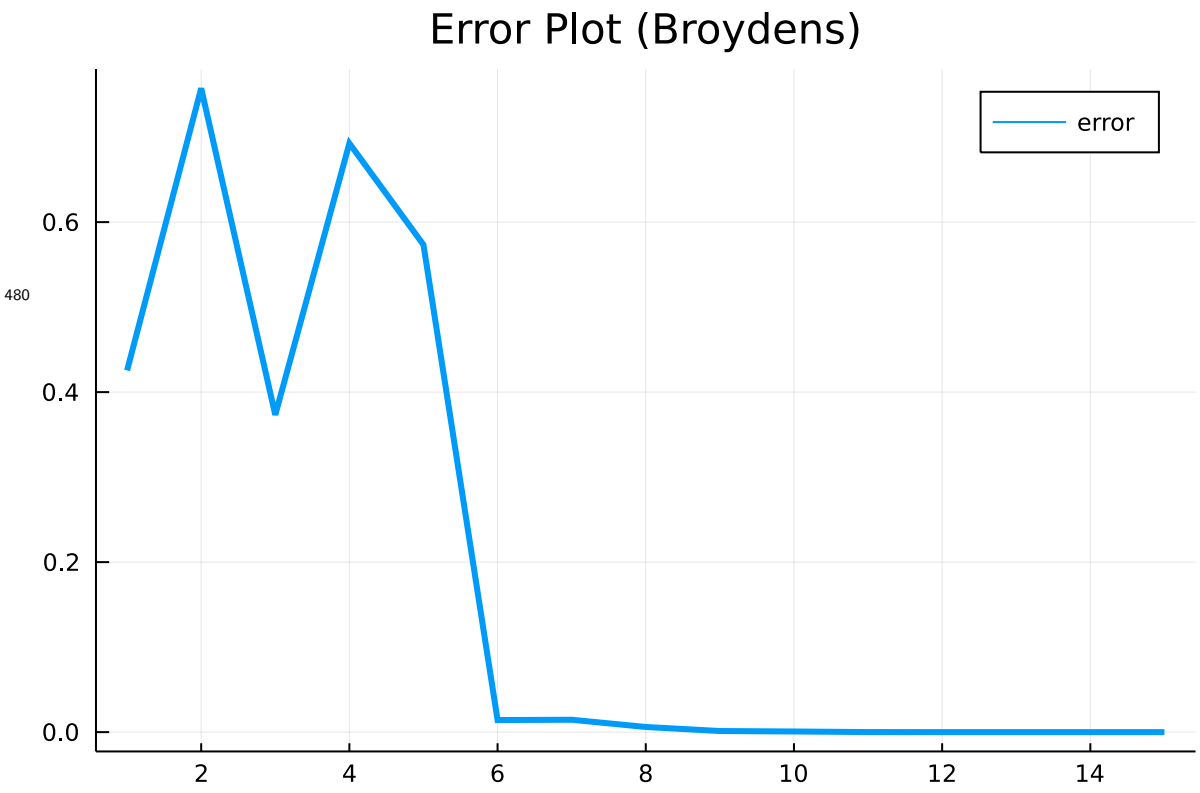
Out[61]:
```
4×16 Matrix{Float64}:
 1.0    1.0    0.964129   0.783924  …   0.816497   0.816497   0.816497
 0.0    0.2    0.321963   0.471895      0.408248   0.408248   0.408248
 0.0   -0.2   -0.321963  -0.471895     -0.408248  -0.408248  -0.408248
 0.0    3.6    3.21259    2.50816       3.0        3.0        3.0
```

In [62]:
```
last_sol = x_history2[:,end]
```

Out[62]:
```
4-element Vector{Float64}:
  0.816496580928769
  0.4082482907394642
 -0.4082482901875766
  2.999999999999557
```

In [63]:
```
plot(1:length(errors2), errors2, title="Error Plot (Broydens)",
label="error", lw=3)
```

Out[63]:



In [140…
```
# sol3
x_history3, errors3 = broydens([0;0.1;10;10], F, J, false);
```

```
===== converged.
```

In [141…
```
x_history3
```

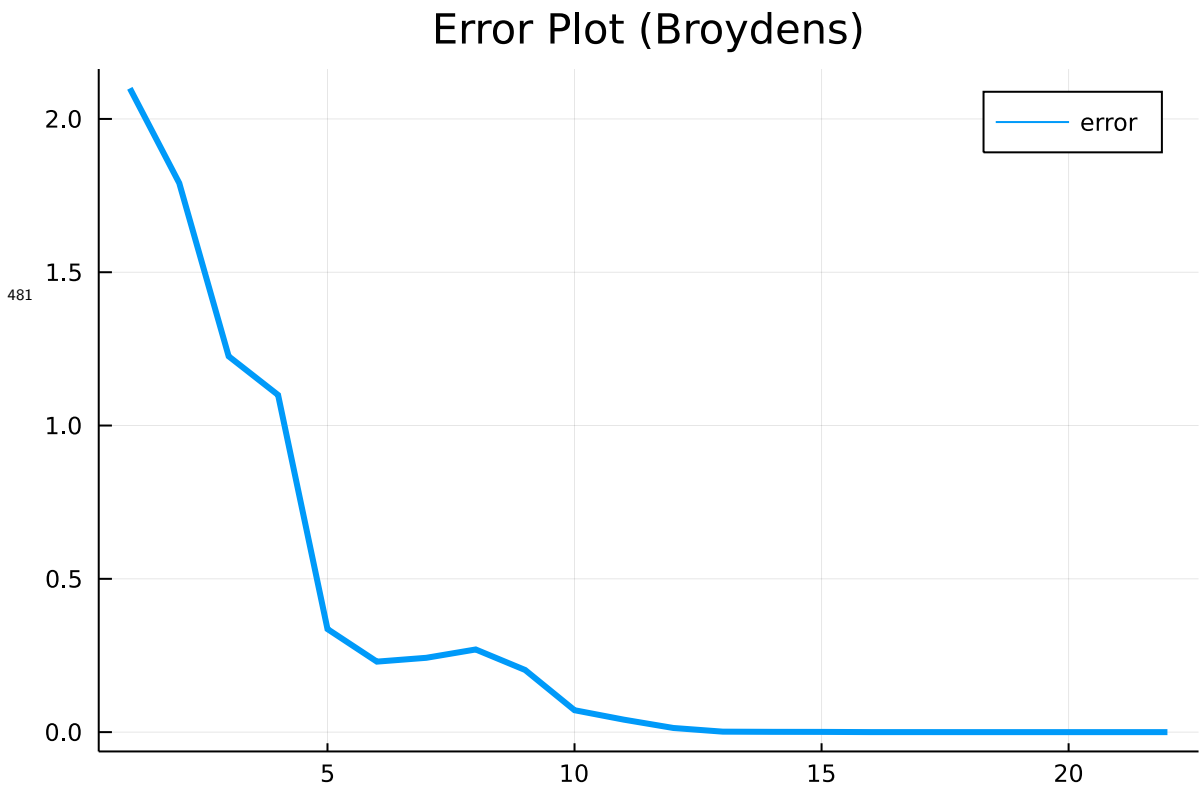
Out[141…
```
4×23 Matrix{Float64}:
  0.0   1.10445   1.64186   1.92365  …   0.57735   0.57735   0.57735
  0.1  -1.56061  -2.07763  -2.08219     -0.57735  -0.57735  -0.57735
 10.0   5.06611   3.18996   1.42295      0.57735   0.57735   0.57735
 10.0   6.87629   6.29913   6.36767      6.0       6.0       6.0
```

In [142…
```
last_sol = x_history3[:,end]
```

Out[142…
```
4-element Vector{Float64}:
  0.5773502691959138
 -0.5773502691895835
  0.5773502691900773
  5.999999999995131
```

In [143…
```
plot(1:length(errors3), errors3, title="Error Plot (Broydens)",
label="error", lw=3)
```

Out[143…


Error Plot (Broydens)
error
2.0
1.5
481
1.0
0.5
0.0
5
10
15
20

11 Course Project / Homework 10

Note: Only partial solutions were published.

11.1 Nonlinear Shooting Algorithm

We would like to solve the following boundary value problem:

$$y''(x) = -(y'(x))^2 - y(x) + \ln x, x \in [1, 2]$$

$$y(1) = 0, y(2) = \ln 2$$

using nonlinear shooting with Newton's method; the method is described in Section 11.1 and 11.2 from textbook.

With uniqueness assumption (Theorem 11.1), the main idea is to convert the boundary value problem to an initial value problem, and perturbing the initial condition (the initial slope y'). The uniqueness constraint is important, but not a requirement for convergence of the numerical method. This paper has done numerical studies of the solution behavior. Different initial conditions can yield drastically different solutions for badly behaved systems.

11.1.1 Derivation

We follow Textbook (11.9)-(11.13), and repeatedly solve two associated initial value problem.

At step k, to advance from $t_k \mapsto t_{k+1}$, we solve:

$$\begin{cases} y''(x, t_k) = f(x, y, y'), y(1, t_k) = 0, y'(1, t_k) = t_k \\ z''(x, t_k) = \frac{\partial f}{\partial y}(x, y, y')z(x, t_k) + \frac{\partial f}{\partial y'}(x, y, y')z'(x, t), z(1, t_k) = 0, z'(1, t_k) = 1 \end{cases}$$

then Newton's method gives us the next step:

$$t_{k+1} = t_k - \frac{y(2, t_k) - \ln 2}{z(2, t_k)}$$

Here in our problem, $f(x, y, y') = -(y')^2 - y + \ln x$. Then:

$$\frac{\partial f}{\partial y}(x, y, y') = -1$$

$$\frac{\partial f}{\partial y'}(x, y, y') = -2y'$$

At the beginning of the iteration, we initialize:

$$t_0 = \frac{\ln 2 - 0}{2 - 1} = \ln 2$$

The main routine is to treat t_k as a parameter, and solve the two initial value problem over $x \in [1, 2]$ for each k. Until the solution error at $x = 2$ (the right boundary) is satisfactory, we update $t_k \mapsto t_{k+1}$ via Newton's method.

The two second order ODE (in x) can be re-expressed as systems of first order ODEs via the following:

$$w_1 = y$$
$$w_2 = y'$$
$$w_3 = z$$
$$w_4 = z'$$

Then we have:

$$\mathbf{w}'(x) = \frac{d}{dx}\begin{pmatrix} w_1(x) \\ w_2(x) \\ w_3(x) \\ w_4(x) \end{pmatrix} = \begin{pmatrix} w_2(x) \\ -w_2(x)^2 - w_1(x) + \ln x \\ w_4(x) \\ -w_3(x) - 2w_2(x) \cdot w_4(x) \end{pmatrix}$$

with the initial condition:

$$\mathbf{w}(1) = \begin{pmatrix} 0 \\ t_k \\ 0 \\ 1 \end{pmatrix}$$

which can be solved by any standard numerical integrator, such as Runge-Kutta.

11.1.2 Example Pseudocode

In order to update t_{k+1}, we need to rely on $y(2, t_k), z(2, t_k)$, which are only obtained through integration over the domain $[1, 2]$ using parameter t_k.

The algorithm is inherently sequential, as is the case in many finite difference methods. When we consider implicit methods, we will see that there is a (interesting) tradeoff between runtime and space:

(1) Solving the entire solution step by step, one point at a time; and:

(2) Solving the entire solution at once, holding many grid points at the same time.

The pseudocode is as follows:

```
initialize h, tol, t0, w0
% checks whether numerical approximation
at right boundary is close enough to ln2
while (|w(1) - ln2| > tol)
    w <- integrate over x=1 to x=2 using standard integrator
    tk <- tk-1 - (w(1) - ln2) / w(3)
    % update initial slope for next iteration
    w(2) <- tk
end while
```

11.1.3 Python Implementation

In the implementation, we write our own RK4 integrator as many of us did. Student will receive full point as well if `ode45` was used (but not for solving the BVP itself). All codes are in the appendix at the end of this document. The numerical experiments were done using $h = 0.1, 0.05$ and $h = 0.01$. Approximate solutions are nearly indistinguishable from the exact solution; the error plots are at least on the order of $O(10^{-5})$, which is to be expected from the truncation error analysis of RK4.

11.2 Finite Difference Method with Newton's Iteration

Please refer to section 11.3, 11.4 from Textbook for a derivation. In finite difference methods, the main idea is to assign an equally (or can be adaptive) spaced grid, and express all derivatives in terms of values on the grid via a truncated Taylor expansion. This idea is generalizable to multiple dimensions for partial derivatives. Suppose we have the function and h as cell size:

$$f(x_1, x_2, \cdots, x_n), \text{ with appropriate smoothness properties}$$

then the following can serve as example approximations.

Forward differencing:

$$\frac{\partial f}{\partial x_k} \approx \frac{1}{h}\Big(f(x_1, x_2, \cdots, x_k + h, \cdots, x_n) - f(x_1, x_2, \cdots, x_k, \cdots, x_n)\Big)$$

Centered differencing:

$$\frac{\partial^2 f}{\partial x_k^2} \approx \frac{1}{h^2}\Big(f(x_1, x_2, \cdots, x_k + h, \cdots, x_n) - 2f(x_1, x_2, \cdots, x_k, \cdots, x_n) + f(x_1, x_2, \cdots, x_k - h, \cdots, x_n)\Big)$$

We solve the BVP by following derivation (11.20) from Textbook, which yields an implicit method. Implicit schemes are generally more stable than explicit counterparts, but are more computationally demanding (as we will need to store the entire grid in memory, along with requisite Jacobian matrices), in the sense that the initial error does not blow up to infinity; precise reasons for why they are stable can be derived through Von Neumann Stability Analysis.

Our problem was:

$$y''(x) = f(x, y, y') = -(y'(x))^2 - y(x) + \ln x, x \in [1, 2]$$

$$y(1) = 0, y(2) = \ln 2$$

As before:

$$f_y(x, y, y') = -1$$

$$f_{y'}(x, y, y') = -2y'$$

Detailed derivation can be found from textbook. For Newton's iteration, we follow Algorithm 11.4. At step 0, we initialize a linear approximation; this choice of initial vector serves as a good guess for Newton's method:

$$\mathbf{w^{(0)}} = \begin{pmatrix} \alpha \\ \alpha + \frac{\beta-\alpha}{b-a}h \\ \alpha + 2\frac{\beta-\alpha}{b-a}h \\ \vdots \\ \alpha + N\frac{\beta-\alpha}{b-a}h \\ \beta \end{pmatrix}$$

And the tridiagonal Jacobian matrix for each iteration is:

$$\mathbf{J}(\mathbf{w^{(k)}}) = \begin{cases} -1 + \frac{1}{2}hf_{y'}(x_i, w_i, w_i'), \text{ upper diagonal} \\ 2 + h^2 f_y(x_i, w_i, w_i'), \text{ diagonal} \\ -1 - \frac{1}{2}hf_{y'}(x_i, w_i, w_i'), \text{ lower diagonal} \end{cases}$$

$$= \begin{cases} -1 - hw_i', \text{ upper diagonal} \\ 2 - h^2, \text{ diagonal} \\ -1 + hw_i', \text{ lower diagonal} \end{cases}$$

where w_i' is approximated using centered differencing, namely:

$$w_i' = \frac{w_{i+1} - w_{i-1}}{2h}$$

541

hw10_p1p2_code

July 2, 2021

[2]:
```
import numpy as np
import scipy as sp
import matplotlib.pyplot as plt
%matplotlib inline
```

1 Problem 1: Nonlinear Shooting with Newton's Method

[72]:
```
# Problem 1: Nonlinear shooting with Newton's Method

def prob1_rhs(w, x):
    """ computes right hand side (see solution derivation). """
    w1, w2, w3, w4 = w[0], w[1], w[2], w[3]
    return np.array([w2, (-w2**2)-w1+np.log(x), w4, -w3-2*w2*w4])


def rk4(w0, h, rhs, x_start=1, x_end=2):
    """ standard Runge-Kutta integrator.

    Input:
        w0                initial condition
        h                 stepsize
        rhs               function to compute rhs, takes in w
        x_start, x_end    specify domain for integration
    Output:
        w_history         full dynamics of w from x_start to x_end
        x_grid            grid generated during integration
    """
    # number of grid points
    N = int((x_end - x_start) / h)
    x_grid = np.linspace(x_start, x_end, N+1)
    w_history = np.zeros([len(w0), N+1])
    w_history[:, 0] = w0
    for i in range(N):
        k1 = h * rhs(w_history[:, i], x_grid[i])
        k2 = h * rhs(w_history[:, i] + k1/2, x_grid[i] + h/2)
        k3 = h * rhs(w_history[:, i] + k2/2, x_grid[i] + h/2)
        k4 = h * rhs(w_history[:, i] + k3, x_grid[i] + h)
```

542

```
        k = (1/6) * (k1 + 2*k2 + 2*k3 + k4)
        # advance one step
        w_history[:, i+1] = w_history[:, i] + k
    return w_history, x_grid


def nonlinear_shooting(a, b, h, alpha=0, beta=np.log(2), tol=1e-8):
    """ nonlinear shooting algorithm for souling two point BVP.
    Reinterpreted version of alg. 11.2 from Textbook.

    Input:
        a, b                spatial domain x_start, x_end
        alpha, beta         boundary conditions for y
        h                   step size for integrator
        tol                 tolerance, default to 1e-08

    Output:

    """
    # initialize params
    t0 = ( beta - alpha ) / (b - a)
    w0 = np.array([0, t0, 0, 1])
    w_k = w0
    t_k = t0
    t_history = []
    t_history.append(t_k)
    # loop, breaks loop when error is small
    while True:
        # shoot
        w_k_history, x_grid = rk4(w0, h, prob1_rhs, x_start=1, x_end=2)
        # take approximated right end point, compare with beta
        y_approx = w_k_history[0, w_k_history.shape[1]-1]
        z_approx = w_k_history[2, w_k_history.shape[1]-1]
        end_error = y_approx - beta
        if abs(end_error) < tol:
            print("> solution converged")
            break
        # otherwise adjust initial slope for re-shooting
        t_k = t_k - (end_error / z_approx)
        t_history.append(t_k)
        w0[1] = t_k
    # after looping, return best w_k, and adjustment history
    final_w = w_k_history

    return x_grid, final_w, t_history
```

[73]:
```
# computations
```

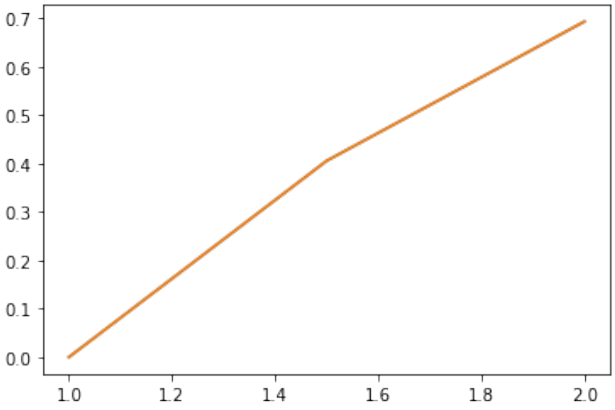
543

[81]:
```
x_grid, w, test2 = nonlinear_shooting(1, 2, 0.5, alpha=0, beta=np.log(2),␣
↪tol=1e-8)
```

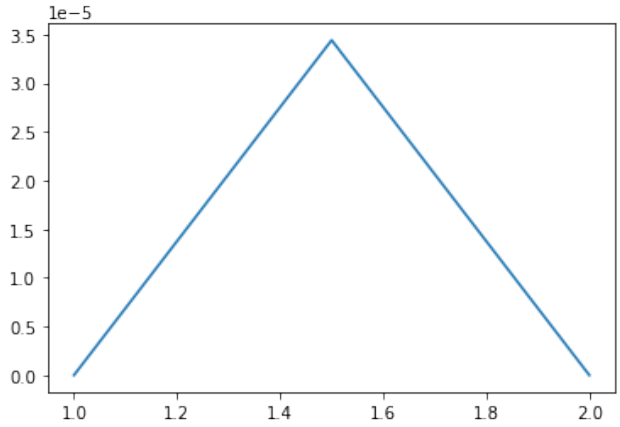
```
> solution converged
```

[84]:
```
# plotting
y_dynamics = w[0, :]
plt.plot(x_grid, y_dynamics, x_grid, np.log(x_grid))
plt.figure(2)
plt.plot(x_grid, abs(y_dynamics-np.log(x_grid)))
```

[84]:
```
[<matplotlib.lines.Line2D at 0x7f8e4ab92fa0>]
```



544



2 Problem 2: Finite Difference Method for Nonlinear BVP

[128]:
```
def prob2_rhs(x, y, y_p):
    """ right hand side for y'' = f(x, y, y'). """
    return -(y_p**2) - y + np.log(x)

def J(w, h):
    """ Helper function to generate Jacobian matrix. """
    # scipy.sparse.diags
    N = len(w)-2 # exclude boundary
    # compute w_prime using centered difference
    print(w[1+1:(N+1)+1])
    print( w[1-1:(N+1)-1])
    w_prime = (1/(2*h)) * ( w[1+1:(N+1)+1] - w[1-1:(N+1)-1] )
    upp_diag = -1 - h*w_prime
    low_diag = -1 + h*w_prime
    diag = (2 - h**2) * np.ones(N)
    J = sp.sparse.diags()
    return J


def fdfbvp_solver(a, b, alpha, beta, h, tol=1e-8):
    """ finite difference BVP solver using Newton's Iteration.
        Generates an equally spaced grid with step size h
```

545

```
    Input:
        a, b                    x_start, x_end
        alpha, beta             boundary conditions
        h                       step size
        tol                     vector error tolerance default 1e-8
    Output:
        y_approx                numerical solution on grid points
    """
    # number of grid points
    N = int(((b-a)/h))-1
    # initialize w
    w = np.zeros(N+2) # w0, w1, ..., wn, wn+1
    w[0] = alpha
    slope = (beta-alpha)/(b-a)
    w[1:N+2] = alpha + np.arange(1, N+2) * slope * h




    return w
```

[129]:
```
fdfbvp_solver(a=1, b=2, alpha=0, beta=np.log(2), h=0.1, tol=1e-8)
```

[129]:
```
array([0.        , 0.06931472, 0.13862944, 0.20794415, 0.27725887,
       0.34657359, 0.41588831, 0.48520303, 0.55451774, 0.62383246,
       0.69314718])
```

[130]:
```
test = np.arange(10)
J(test, 1)
```

```
[2 3 4 5 6 7 8 9]
[0 1 2 3 4 5 6 7]
```

[ ]:
```
test[1:10]
```

We used step sizes $h = 0.1, 0.05, 0.01$. We expect this method to be at least second order.

11.3 Further Problems

Due to time constraint, solutions to the rest of the project were not published. One may attempt the following problems, please send an email to any of the course staff members if interested.

11.3.1 Implementation of Poisson Equation Finite-Difference Method and Solve B& F Page 742, Problem 3

11.3.2 Implementation of Crank-Nicholson Method, and Solve B& F Page 754, Problem 2

11.3.3 Implementation of the Wave Equation using Finite-Difference Method, and Solve B& F Page 763, Problem 2

12 Final Exam, Spring 2021

Abstract

This document serves as reference solutions for Math 128B, Spring 2021 by Prof.Olga Holtz. The final was held online via the Gradescope platform, Thursday, May 13th, 2021, from 3:10 pm to 6:25 pm. Please find solutions after the final exam.

564

MATH 128B, Spring 2021, final exam.

All the necessary work to justify an answer and all the necessary steps of a proof must be shown clearly to obtain full credit. Partial credit **may** be given but only for significant progress towards a solution. Cross out all work you do not wish considered. Books and notes are allowed. Electronic computing devices are not allowed during the test.

1. (10pts.) Define the following matrix function on the space of $n \times n$ complex matrices:

$$\|A\|_{sum} := \sum_{i=1}^{n} \sum_{j=1}^{n} |a_{ij}|.$$

Show that $\|\cdot\|_{sum}$ is a matrix norm.

565

2. (10pts.) Show that the following formula describes a family of orthonormal polynomials for the weight function $w(x) \equiv 1$ on the interval $[-1, 1]$:

$$P_n(x) := \frac{(n+\frac{1}{2})^{1/2}}{2^n n!} \frac{d^n}{dx^n}(x^2-1)^n.$$

566

3. (14pts.) Consider the trigonometric sum $U_n(x) := \sum'^{n}_{j=-n} c_j e^{ijx}$, where

$$\sum'^{n}_{j=-n} a_j := \sum_{j=-n}^{n} a_j - \frac{1}{2}\left(a_n + a_{-n}\right).$$

Show that the solution to the interpolation problem $U_n(x_k) = f(x_k)$, $k = -n, \ldots, 0, \ldots, n$, for the equally spaced points $x_k := k\pi/n$ satisfies

(a) $c_j = \dfrac{1}{2n}\sum'^{n}_{k=-n} f(x_k) e^{-ijx_k}, \quad j = -n, \ldots, n;$ (b) $\dfrac{1}{2n}\sum'^{n}_{k=-n} |f(x_k)|^2 = \sum'^{n}_{j=-n} |c_j|^2.$

567

4. (10pts.) Suppose that p, q, and r are continuous functions and, moreover, that $q(x) \geq 0$ on $[a, b]$. Let $h := (b-a)/(N+1)$ for some positive integer N, and consider the finite-difference method applied to the linear boundary value problem

$$y'' = p(x)y' + q(x)y + r(x) \quad \text{for } x \in [a, b], \quad y(a) = \alpha,\ y(b) = \beta.$$

Use Gershgorin's theorem to prove that the resulting tridiagonal linear system (11.19) has a unique solution provided $h < 2/L$ where $L := \max_{x\in[a,b]} |p(x)|$.

568

5. (10pts.) Suppose the $n \times n$ matrix A has eigenvalues λ_j, $j = 1, \ldots, n$, such that

$$|\lambda_1| > |\lambda_2| > \cdots > |\lambda_n|,$$

and suppose the power method is applied to an initial vector of the form

$$x^{(0)} = \alpha_2 v^{(2)} + \cdots + \alpha_n v^{(n)}$$

where $\alpha_2 \neq 0$ and where $v^{(j)}$ denotes an eigenvector corresponding to eigenvalue λ_j, for each j. Determine, with proof, the limit of the sequence $\{\mu^{(m)}\}$ described in Algorithm 9.1.

569

6. (6pts.) Let $x \in \mathbb{R}^n$, $x \neq 0$, $n \geq 2$. Let P be a Householder matrix such that $Px = \pm\|x\|_2 e_1$. Let $G_{k,\ell}$ denote a Givens rotation matrix which is equal to the identity except for the rotation submatrix in rows and columns k, ℓ. Suppose $G_{1,2}$, ..., $G_{n-1,n}$ are such Givens rotations that

$$Qx = Px \qquad \text{for } Q := G_{1,2}G_{2,3}\cdots G_{n-1,n}.$$

True or false: $P = Q$? Justify your answer.

Begin Solution:

12.1 Matrix Norm

Following Definition 7.8 in textbook, a matrix norm on the space of $n \times n$ complex matrices need to satisfy:

(1) (Positivity) $\|A\| \geq 0$, and $\|A\| = 0$ if and only if $A = 0$.

(2) (Homogeneous) $\|\alpha A\| = |\alpha| \cdot \|A\|$ for all $\alpha \in \mathbb{C}$.

(3) (Triangle inequality) $\|A + B\| \leq \|A\| + \|B\|$.

(4) (Sub-multiplicative) $\|AB\| \leq \|A\| \cdot \|B\|$.

The norm in question is in fact the p-norm ($p \in [1, \infty]$), defined as the following:

$$\|A\|_p = \left(\sum_{i=1}^{n} \sum_{j=1}^{n} |A_{ij}|^p \right)^{1/p}$$

We show the above properties for the p-norm.

(1) Positivity is clear from the property of absolute value. $|A_{ij}| \geq 0$ for all i, j. Furthermore, the double sum equals to 0 if and only if every entry of A is 0.

(2) Homogeneity also comes from properties of absolute value. Take $\alpha \in \mathbb{C}$, we have:

$$\|\alpha A\|_p^p = \sum_{i=1}^{n} \sum_{j=1}^{n} |\alpha A_{ij}|^p = \sum_{i=1}^{n} \sum_{j=1}^{n} |\alpha|^p \cdot |A_{ij}|^p = |\alpha|^p \sum_{i=1}^{n} \sum_{j=1}^{n} |A_{ij}|^p = |\alpha|^p \cdot \|A\|_p^p$$

(3) Triangle inequality: Take $A, B \in \mathbb{C}^{n \times n}$, then:

$$\|A + B\|_p^p = \sum_{i=1}^{n} \sum_{j=1}^{n} |A_{ij} + B_{ij}|^p \leq \sum_{i=1}^{n} \sum_{j=1}^{n} (|A_{ij}| + |B_{ij}|)^p$$

and we can conclude the inequality by noting:

$$(|A_{ij}| + |B_{ij}|)^p \leq |A_{ij}|^p + |B_{ij}|^p$$

for all i, j. However, this may not be immediately obvious. For $p = 1$, it is triangle inequality of absolute values. For other p's, one can attempt to prove:

$$(1 + t)^p \leq 1 + t^p$$

where we take $|A_{ij}| = 1$, $t = \frac{|B_{ij}|}{|A_{ij}|}$. We can assume $|A_{ij}| = 1$ without loss of generality, because:

$$(a+b)^p = a^p(1+\frac{b}{a})^p$$

and scaling is proved from above. The rest of the proof is generally a fact and can be found in standard textbooks in real analysis or numerical analysis.

(4) Take $A, B \in \mathbb{C}^{n\times n}$, then:

$$\|AB\|_p^p = \sum_{i=1}^n \sum_{j=1}^n \left| \sum_{k=1}^n A_{ik}B_{kj} \right|^p \leq \sum_{i=1}^n \sum_{j=1}^n \sum_{k=1}^n |A_{ik}|^p |B_{kj}|^p$$

since this triple sum is finite, we can exchange summation orders:

$$= \sum_{i=1}^n \sum_{k=1}^n \sum_{j=1}^n |A_{ik}|^p |B_{kj}|^p$$

$$= \sum_{i=1}^n \sum_{k=1}^n |A_{ik}|^p \sum_{j=1}^n |B_{kj}|^p \leq \left(\sum_{i=1}^n \sum_{k=1}^n |A_{ik}|^p \right) \cdot \left(\sum_{k=1}^n \sum_{j=1}^n |B_{kj}|^p \right) = \|A\|_p^p \cdot \|B\|_p^p$$

12.2 Orthonormal Polynomials

The closed form for Legendre polynomials are defined as:

$$P_n(x) = \frac{(n+\frac{1}{2})^{1/2}}{2^n n!}\frac{d^n}{dx^n}(x^2-1)^n$$

We follow Definition 8.5 about orthonormal polynomials with $w = 1$, on the interval $[-1, 1]$, using integration by parts. Without loss of generality assume $m \geq n$. For simplicity, write $c_k = \frac{(k+1/2)^{1/2}}{2^k k!}$.

$$\int_{-1}^{1} P_m(x)P_n(x)dx = \int_{-1}^{1} c_m c_n \left(\frac{d^m}{dx^m}(x^2-1)^m\right)\left(\frac{d^n}{dx^n}(x^2-1)^n\right)dx$$

a common solution is to use integration by parts to compute:

$$\int_{-1}^{1}\left(\frac{d^m}{dx^m}(x^2-1)^m\right)\left(\frac{d^n}{dx^n}(x^2-1)^n\right)dx = \left(\frac{d^n}{dx^n}(x^2-1)^n\right)\cdot\left(\frac{d^{m-1}}{dx^{m-1}}(x^2-1)^m\right)\Bigg|_{-1}^{1}$$

$$-\int_{-1}^{1}\left(\frac{d^{m-1}}{dx^{m-1}}(x^2-1)^m\right)\cdot\left(\frac{d^{n+1}}{dx^{n+1}}(x^2-1)^n\right)dx$$

We have that the boundary term vanishes, by considering the latter factor:

$$\frac{d^{m-1}}{dx^{m-1}}(x^2-1)^m = \frac{d^{m-1}}{dx^{m-1}}(x+1)^m(x-1)^m$$

by repeatedly applying product rule, one can show inductively that the final expression involves a sum of products of $(x+1)^k$ and $(x-1)^k$ for $k \geq 1, k \in \mathbb{N}$. And hence at either $x = -1$ or $x = 1$, the expression vanishes identically.

Proceeding with integration by parts on:

$$\int_{-1}^{1}\left(\frac{d^m}{dx^m}(x^2-1)^m\right)\left(\frac{d^n}{dx^n}(x^2-1)^n\right)dx = -\int_{-1}^{1}\left(\frac{d^{m-1}}{dx^{m-1}}(x^2-1)^m\right)\left(\frac{d^{n+1}}{dx^{n+1}}(x^2-1)^n\right)dx$$

for $(m-1)$ more times, we eventually obtain:

$$= (-1)^m\int_{-1}^{1}(x^2-1)^m\left(\frac{d^{n+m}}{dx^{n+m}}(x^2-1)^n\right)dx$$

now since $m \geq n$ by assumption, then $n+m \geq 2n$.

If $m \neq n$, $n+m \geq 2n+1$, and $(x^2-1)^n$ is a degree $2n$ polynomial, hence the derivative:

$$\frac{d^{n+m}}{dx^{n+m}}(x^2-1)^n = 0$$

this shows:

$$\int_{-1}^{1} P_m(x) \cdot P_n(x) dx = 0$$

If $m = n$, we have:

$$\int_{-1}^{1} P_n^2(x) dx = (-1)^n (2n)! c_n^2 \int_{-1}^{1} (x^2-1)^n dx$$

the factorial term is from:

$$\frac{d^{2n}}{dx^{2n}}\left[x^{2n} + O(x^{2n-1})\right] = (2n)!$$

There are certain ways to compute the integral $\int_{-1}^{1}(x^2-1)^n dx$, based on the difficulty and student responses, students received full credit if:

(1) Student looked up what is referred to as "Gamma function" from other standard textbooks.

(2) Student used integration by parts to directly evaluate:

$$\int_{-1}^{1}(x^2-1)^n dx = (-1)^n \int_{-1}^{1}(1-x^2)^n dx$$

recursively, while making note that boundary terms vanish on $[-1, 1]$.

(3) Student left the un-simplied expression as-is, while documenting fully correct other coefficients.

(4) Student did not choose to evaluate $m = n$ by integration by parts but used instead the recurrence relation of Legendre polynomials.

In the end:

$$(-1)^n \int_{-1}^{1}(x^2-1)dx = \frac{n!}{(n+\frac{1}{2})(n-\frac{1}{2})(n-\frac{3}{2})\cdots\frac{1}{2}}$$

And:

$$\int_{-1}^{1} P_n^2(x) dx = (2n)! \cdot \frac{(n+\frac{1}{2})}{2^{2n}(n!)^2} \cdot \frac{n!}{(n+\frac{1}{2})(n-\frac{1}{2})(n-\frac{3}{2})\cdots\frac{1}{2}}$$

$$= \frac{(2n)!}{2^{2n}n!(n-\frac{1}{2})(n-\frac{3}{2})\cdot\frac{1}{2}} = 1$$

the last equality comes from distributing $2^n = 2\cdot 2\cdot 2\cdots 2$ to each term $n-\frac{1}{2}$, $n-\frac{3}{2}\cdots$, $\frac{1}{2}$ to yield $(2n-1)(2n-3)(2n-5)\cdots 1$. The even terms comes from $2^n n!$ in a similar manner.

12.3 Trigonometric Interpolation

12.3.1 (a) Solve for c_j

Comment: Relies on the fact that $\{e^{ijk\pi/n}\}$ forms an orthogonal set.

We directly expand the sum.

$$\sum_{k=-n}^{n}{}' f(x_k)e^{-ijx_k}$$

here j is a fixed index, and is not meant to be confused with a dummy index for expressing summation, here we substitute in:

$$f(x_k) = \sum_{l=-n}^{n}{}' c_l e^{ilx_k}$$

and we have:

$$\sum_{k=-n}^{n}{}' \left(\sum_{l=-n}^{n}{}' c_l e^{ilx_k} \right) e^{-ijx_k} = \sum_{k=-n}^{n}{}' \left(\sum_{l=-n}^{n}{}' c_l e^{i(l-j)x_k} \right)$$

now we observe that $e^{i(l-j)x_k} = \delta_{l,j}$ for x_k equally spaced, where $\delta_{l,j} = 1$ only when $l = j$, by orthogonality. Hence only c_j will survive in the summation, and we have:

$$= \sum_{k=-n}^{n}{}' c_j = (2n + 1 - 1)c_j = 2nc_j$$

since c_j does not depend on k (the sum has $(2n+1)$ terms, namely $-n, -n+1, \cdots, -1$ are n terms; $1, 2, \cdots, n$ are n terms; $k = 0$ is an additional term that some students missed). The endpoints $\pm n$ agree for all nodes x_k.

Now divide both sides by $2n$, we yield the desired result.

12.3.2 (b) Parseval's Theorem

The problem is showing the discrete version of Parseval's theorem, which states that (discrete) Fourier transform is a unitary transformation (in the sense that it preserves the squared norm).

This is done by directly making use of any / all of the following:

(1) Part (a).

(2) Orthogonal set $\{e^{ijx_k}\}$.

(3) Complex exponential has unit absolute value.
(4) $|z|^2 = z \cdot \overline{z}$ where $\overline{z}$ denotes the complex conjugate.

$$\sum_{k=-n}^{n}{}' |f(x_k)|^2 = \sum_{k=-n}^{n}{}' f(x_k) \cdot \overline{f(x_k)}$$

$$= \sum_{k=-n}^{n}{}' \left(\sum_{j=-n}^{n}{}' c_j e^{ijkx_k} \right) \cdot \left(\sum_{l=-n}^{n}{}' \overline{c_j} \cdot e^{-ilx_k} \right)$$

with similar reasoning, only the diagonal terms will survive from the inner distribution:

$$= \sum_{k=-n}^{n}{}' \sum_{j=l=-n}^{n}{}' c_j \cdot \overline{c_j} = \sum_{k=-n}^{n}{}' \sum_{j=-n}^{n}{}' |c_j|^2$$

exchanging the summation and noting that c_j has no dependence on k yields:

$$= \sum_{j=-n}^{n}{}' (2n+1-1)|c_j|^2 = 2n \sum_{j=-n}^{n}{}' |c_j|^2$$

as desired.

12.4 Unique Numerical Solution for Linear BVP

From (11.19) in textbook, we remember the linear system was formulated by using centered differencing to replace $y'' = \frac{d^2}{dx^2}y \approx \frac{1}{h^2}(y_{i+1} - 2y_i + y_{i-1})$ and $y' = \frac{d}{dx}y \approx \frac{1}{2h}(y_{i+1} - y_{i-1})$, yielding the following tridiagonal system:

$$\begin{pmatrix} 2+h^2q(x_1) & -1+\frac{h}{2}p(x_1) & 0 & \cdots & 0 \\ -1-\frac{h}{2}p(x_2) & 2+h^2q(x_2) & -1+\frac{h}{2}p(x_2) & \cdots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & -1+\frac{h}{2}p(x_{N-1}) \\ 0 & \cdots & \cdots & -1-\frac{h}{2}p(x_N) & 2+h^2q(x_N) \end{pmatrix} \mathbf{y} = \mathbf{b}$$

the problem amounts to showing that the tridiagonal system is nonsingular; under the condition that:

$$h < \frac{2}{L} = \frac{2}{\max_{x\in[a,b]}|p(x)|}, q(x) \geq 0$$

Furthermore, a matrix is nonsingular if and only 0 is not an eigenvalue; we have thus converted this problem into an eigenvalue estimation problem, where we exploit Gershgorin's Theorem.

We have an estimate of the size:

$$\left|\frac{h}{2}p(x)\right| \leq \frac{h}{2}L < \frac{L}{2}\cdot\frac{2}{L} = 1$$

therefore:

$$\left|-1-\frac{h}{2}p(x)\right| = \left|-(1+\frac{h}{2}p(x))\right| = 1+\frac{h}{2}p(x)$$

because $\frac{h}{2}p(x)$ has magnitude strictly less than 1, the quantity is guaranteed to be negative. Similarly:

$$\left|-1+\frac{h}{2}p(x)\right| = 1-\frac{h}{2}p(x)$$

Gershgorin's theorem states that:

$$\lambda \in \bigcup_{i=1}^{N} R_i$$

where:

$$R_i = \left\{ z \in \mathbb{C}, |z - A_{ii}| \leq \sum_{j=1, j\neq i}^{N} |A_{ij}| \right\}$$

where A is our tridiagonal matrix.

For all $i = 2, 3, \cdots, N-1$:

$$R_i = \{z \in \mathbb{C}, \left| z - \left(2 + h^2 q(x_i)\right) \right| \leq \left| -1 - \frac{h}{2} p(x_i) \right| + \left| -1 + \frac{h}{2} p(x_i) \right| \}$$

By above arguments, we know that:

$$\left| -1 - \frac{h}{2} p(x_i) \right| + \left| -1 + \frac{h}{2} p(x_i) \right| = 1 + 1 = 2$$

Then the circles became:

$$R_i = \{ z \in \mathbb{C} : \left| z - (2 + h^2 q(x_i)) \right| \leq 2 \}$$

we solve for z in R_i for all $i \in \{2, 3, \cdots, N-1\}$:

$$0 \leq h^2 q(x_i) \leq z \leq 4 + h^2 q(x_i)$$

At the boundaries, namely $i = 1$ or $i = N$, we have:

$$R_i = \{ z \in \mathbb{C} : \left| z - (2 + h^2 q(x_i)) \right| \leq 1 \pm \frac{h}{2} p(x) \}$$

By triangle inequality:

$$\left| -1 \pm \frac{h}{2} p(x) \right| \leq 1 + \frac{h}{2} |p(x)| < 2$$

then we solve for $z \in R_1, R_N$:

$$0 \leq h^2 q(x_i) < z < 4 + h^2 q(x_i)$$

Comment: One can stop here, conclude that the tridiagonal matrix is weakly diagonally dominant and refer to Theorem 6.31 to conclude that the tridiagonal system is nonsingular. More precisely, Gersgorin's theorem may not yield the desired tight bound even $q = 0$ is allowed. Student solutions that directly applied Gersgorin's theorem without coping with this subtlety are (tentatively) given full credit.

12.5 Power Method

We follow the inductive definition of (9.3) inductively from textbook, where:

$$y^{(m)} = Ax^{(m-1)}$$

which is normalized with respect to $\|\cdot\|_\infty$ after each step:

$$x^{(m)} = \frac{y^{(m)}}{\|y^{(m)}\|_\infty}$$

yielding (inductively, student solution can direct refer to this expression without establishing $\mu^{(1)}$):

$$\mu^{(m)} = \frac{\sum_{i=1}^n \alpha_i \lambda_i^m v_{p_{m-1}}^{(i)}}{\sum_{i=1}^n \alpha_i \lambda_i^{m-1} v_{p_{m-1}}^{(i)}}$$

where p_{m-1} is the index such that $x_{p_{m-1}}^{(m-1)} = \|x\|_\infty$. Regardless of the specific p, $v_{p_{m-1}}^{(i)}$ is a scalar.

Recall that this is done by iteratively considering:

$$Ax^{(0)} = A(\sum_{i=1}^n \alpha_i v^{(i)}) = \sum_{i=1}^n \alpha_i A v^{(i)} = \sum_{i=1}^n \alpha_i \lambda_i v^{(i)}$$

where $v^{(i)}$ is an eigenvector with eigenvalue λ_i, hence the last equality.

In our case, $\alpha_1 = 0$, then we can rewrite:

$$\mu^{(m)} = \frac{\alpha_2 \lambda_2^m v_{p_{m-1}}^{(2)} + \sum_{i=3}^n \alpha_i \lambda_i^m v_{p_{m-1}}^{(i)}}{\alpha_2 \lambda_2^{m-1} v_{p_{m-1}}^{(2)} + \sum_{i=3}^n \alpha_i \lambda_i^{m-1} v_{p_{m-1}}^{(i)}} = \lambda_2 \left(\frac{\alpha_2 v_{p_{m-1}}^{(2)} + \sum_{i=3}^n \alpha_i \left(\frac{\lambda_i}{\lambda_2}\right)^m v_{p_{m-1}}^{(i)}}{\alpha_2 v_{p_{m-1}}^{(2)} + \sum_{i=3}^n \alpha_i \left(\frac{\lambda_i}{\lambda_2}\right)^{m-1} v_{p_{m-1}}^{(i)}} \right)$$

Since the eigenvalues are sorted, $\left|\frac{\lambda_i}{\lambda_2}\right| < 1$, we have for all $i > 2$:

$$\lim_{m\to\infty} \left(\frac{\lambda_i}{\lambda_2}\right)^m = 0$$

Taking the limit of $\mu^{(m)}$ yields:

$$\lim_{m\to\infty} \mu^{(m)} = \lambda_2$$

12.6 Householder Transformation and Givens Rotation Equivalence

Intuitively we would expect the answer to be no. A givens rotation:

$$R(\theta) = \begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix}$$

rotates any vector $v \in \mathbb{R}^2$ counterclockwise by θ. In 3 dimensions (x, y, z) coordinates, we can choose to embed this rotation for any pair of indices. We can choose to rotate on the plane $(x, y), (x, z)$ and (y, z) by θ degree, counterclockwise. On the other hand, householder reflection $P = I - 2uu^T$, applied on some vector v, where $u = e_1$ is a reflection of v in the (hyper)plane through 0 and perpendicular to e_1. If e_1 is unit vector on the x-axis in $\mathbb{R}^2$, then in 2D specifically, P would be a reflection across the y-axis. In the simplest 2D case, we see intuitively that there are at least 2 ways to rotate v using Givens rotation such that it matches Pv eventually (in higher dimensions, there are many more ways to rotate than to reflect); namely clockwise ($\theta < 0$) and counterclockwise ($\theta > 0$). The rotation matrix is not equal to a reflection matrix.

Counterexample (2D) Let $v = [1, 0]^T = e_1$, $u = e_2 = [0, 1]^T$, and then:

$$P = I - 2uu^T = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

this can be considered as a reflection across the x-axis, trivially:

$$Pv = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

The same can be achieved by the counterclockwise (Givens) rotation in the (x, y) plane by 0 or 2π degrees, let:

$$R(2\pi) = \begin{pmatrix} \cos(2\pi) & -\sin(2\pi) \\ \sin(2\pi) & \cos(2\pi) \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

$$Rv = Iv = v$$

but $R \neq P$. In both of these cases $Rv = Pv = \|v\|_2 e_1$.

13 Concluding Remarks

I have always had a special feeling towards numerical analysis. Using algorithms to perform mathematical and statistical computations to test ideas, and observing our pure mathematicians making correct predictions based on rigorous theory - the whole process is simply fascinating to me, and not all feelings can be best captured by words, needless to mention not all feelings require any (pretentious) explanations. I would like to thank Prof. Olga Holtz for teaching a great course from which I had the wonderful opportunity to write up these solutions and receive feedbacks, and her mentorship to undergraduate research outside of class.

This document is hopefully dedicated to anyone who enjoys the subject of numerical analysis. If you are a young professional who shares a similar passion, please check out Berkeley courses Math 228A, Math 228B, Math 221, STAT 243 from the Math and Stats departments; Math 202A, Math 202B, Math 222A, Math 222B for rigorous theoretical backgrounds, and COMPSCI 289A, COMPSCI 267 at EECS, as I am sure you will not regret.

14 Referenece

Burden, R. L., Faires, J. D., & Burden, A. M. (2016). *Numerical analysis.*

Demmel, J. (1997). *Applied Numerical Linear Algebra.* Society for Industrial and Applied Mathematics.