PHYS 581: Assignment #3

Scott Salmon UCID: 30093320

March 3rd, 2025

Question 1 (10 points)

We need to derive the n-point integration formula's for $2 \leq n \leq 10$ using Lagrange Basis Polynomials defined on an evenly spaced grid on Mathematica. We will then compare the derived formula from our program with the direct derivations completed using the Trapezoid rule (i.e. $n = 2$), Simpsons 1/3rd rule ($n = 3$), and Simpson's 3/8th rule ($n = 4$).

We were given the derivation for the Trapezoid rule as equation (1.47) of the course notes, which is as follows:

$$\int_{x_1}^{x_2} f(x)\, dx \approx \frac{h}{2}\Big[f(x_1) + f(x_2)\Big]$$

To do the derivation for Simpson's 1/3rd rule, we start with a a 3-point langrage interpolating polynomial, in which $h = (x_3 - x_1)/2$ and $x_3 = x_1 + 2h$, and $x_2 = x_1 + h$:

$$P(x) = f(x_1)\frac{(x - x_1 - h)(x - x_3)}{-h(x_1 - x_3)} + f(x_2)\frac{(x - x_1)(x - x_3)}{h(x_1 + h - x_3)} + f(x_3)\frac{(x - x_1)(x - x_1 - h)}{(x_3 - x_1)(x_3 - x_1 - h)}$$

and then we'll integrate from x_1 to x_3 (using $u = x - x_1 - h$, or $x = u + x_1 + h$)

$$\int_{x_1}^{x_3} P(x)\, dx = \int_{-h}^{h} P(u)\, du$$

$$\begin{aligned}
&= f(x_1)\int_{-h}^{h} \frac{(u + x_1 + h - x_1 - h)(u + x_1 + h - x_3)}{-h(x_1 - x_3)}\, du \\
&\quad + f(x_2)\int_{-h}^{h} \frac{(u + x_1 + h - x_1)(u + x_1 + h - x_3)}{h(x_1 + h - x_3)}\, du \\
&\quad + f(x_3)\int_{-h}^{h} \frac{(u + x_1 + h - x_1)(u + x_1 + h - x_1 - h)}{(x_3 - x_1)(x_3 - x_1 - h)}\, du
\end{aligned}$$

$$\begin{aligned}
&= f(x_1)\int_{-h}^{h} \frac{u(u + x_1 + h - x_3)}{-h(x_1 - x_3)}\, du \\
&\quad + f(x_2)\int_{-h}^{h} \frac{(u + h)(u + x_1 + h - x_3)}{h(x_1 + h - x_3)}\, du \\
&\quad + f(x_3)\int_{-h}^{h} \frac{u(u + h)}{(x_3 - x_1)(x_3 - x_1 - h)}\, du
\end{aligned}$$

$$= f(x_1)\int_{-h}^{h}\frac{u(u-h)}{2h^2}\,du + f(x_2)\int_{-h}^{h}\frac{(u+h)(u-h)}{-h^2}\,du + f(x_3)\int_{-h}^{h}\frac{u(u+h)}{2h^2}\,du$$

$$= \frac{f(x_1)}{2h^2}\int_{-h}^{h}(u^2-uh)\,du + \frac{f(x_2)}{-h^2}\int_{-h}^{h}(u^2-h^2)\,du + \frac{f(x_3)}{2h^2}\int_{-h}^{h}(u^2+uh)\,du$$

$$= \frac{f(x_1)}{2h^2}\left[\frac{u^3}{3}-\frac{u^2h}{2}\right]\Bigg|_{-h}^{h} + \frac{f(x_2)}{-h^2}\left[\frac{u^3}{3}-uh^2\right]\Bigg|_{-h}^{h} + \frac{f(x_3)}{2h^2}\left[\frac{u^3}{3}+\frac{u^2h}{2}\right]\Bigg|_{-h}^{h}$$

$$= \frac{f(x_1)}{2h^2}\left[\frac{h^3}{3}-\frac{h^3}{2}+\frac{h^3}{3}+\frac{h^3}{2}\right] + \frac{f(x_2)}{-h^2}\left[\frac{h^3}{3}-h^3+\frac{h^3}{3}-h^3\right] + \frac{f(x_3)}{2h^2}\left[\frac{h^3}{3}+\frac{h^3}{2}+\frac{h^3}{3}-\frac{h^3}{2}\right]$$

$$= \frac{f(x_1)}{2h^2}\left(\frac{2}{3}h^3\right) + \frac{f(x_2)}{-h^2}\left(\frac{-4}{3}h^3\right) + \frac{f(x_3)}{2h^2}\left(\frac{2}{3}h^3\right)$$

$$\therefore \quad \int_{x_1}^{x_3} f(x)\,dx \approx \frac{h}{3}\Big[f(x_1)+4f(x_2)+f(x_3)\Big]$$

which aligns with equation (1.48) of the course notes (i.e. Simpson's 1/3rd rule). This procedure can be completed to find equation (1.49) for Simpson's 3/8th rule, but I do not want to do all that work again so I will explicitly state that equation:

$$\int_{x_1}^{x_4} f(x)\,dx \approx \frac{3h}{8}\Big[f(x_1)+3f(x_2)+3f(x_3)+f(x_4)\Big]$$

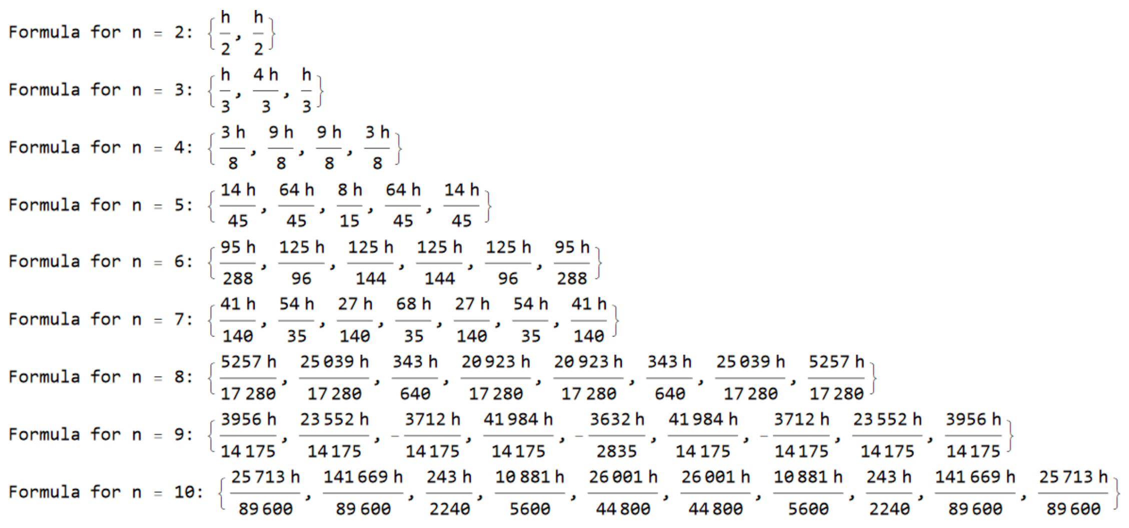
Now, using equation (1.50) from the course notes, we know that the following is true:

$$\int_{x_1}^{x_N} f(x)\,dx \approx \sum_{i=1}^{N} w_i f_i$$

where w_i depends on the integration rules determined previously. So for example, the Trapezoid rule will have $w_i = h/2$ for $i = 1, N$ and $w_i = h$ for $i = 2, ..., N-1$. Therefore, if we integrate a Lagrange Basis Polynomial, we'll be able to extract the weights of each formula and compare these to the predicted weights from the aforementioned rules.

To solve the question for $n = 4, 5, 6, 7, 8, 9, 10$, I created a function in Mathematica that takes in an n-value and automatically spits out the weights for the specified rule. The function dynamically makes an xPoints array that is n points long separated in an equally spaced fashion (spacing denoted as h). I then create a set of Lagrange Basis Polynomials and (applying equation 1.50) I integrate each of the polynomials from x_1 to x_N to find the specified weights for each point.

An image of the Mathematica output is shown below, and note the equivalence to the Trapezoid, Simpson's 1/3rd and Simpson's 3/th rules for $n = 2, 3, 4$.



Formula for n = 2: $\left\{\frac{h}{2}, \frac{h}{2}\right\}$

Formula for n = 3: $\left\{\frac{h}{3}, \frac{4 h}{3}, \frac{h}{3}\right\}$

Formula for n = 4: $\left\{\frac{3 h}{8}, \frac{9 h}{8}, \frac{9 h}{8}, \frac{3 h}{8}\right\}$

Formula for n = 5: $\left\{\frac{14 h}{45}, \frac{64 h}{45}, \frac{8 h}{15}, \frac{64 h}{45}, \frac{14 h}{45}\right\}$

Formula for n = 6: $\left\{\frac{95 h}{288}, \frac{125 h}{96}, \frac{125 h}{144}, \frac{125 h}{144}, \frac{125 h}{96}, \frac{95 h}{288}\right\}$

Formula for n = 7: $\left\{\frac{41 h}{140}, \frac{54 h}{35}, \frac{27 h}{140}, \frac{68 h}{35}, \frac{27 h}{140}, \frac{54 h}{35}, \frac{41 h}{140}\right\}$

Formula for n = 8: $\left\{\frac{5257 h}{17280}, \frac{25039 h}{17280}, \frac{343 h}{640}, \frac{20923 h}{17280}, \frac{20923 h}{17280}, \frac{343 h}{640}, \frac{25039 h}{17280}, \frac{5257 h}{17280}\right\}$

Formula for n = 9: $\left\{\frac{3956 h}{14175}, \frac{23552 h}{14175}, -\frac{3712 h}{14175}, \frac{41984 h}{14175}, -\frac{3632 h}{2835}, \frac{41984 h}{14175}, -\frac{3712 h}{14175}, \frac{23552 h}{14175}, \frac{3956 h}{14175}\right\}$

Formula for n = 10: $\left\{\frac{25713 h}{89600}, \frac{141669 h}{89600}, \frac{243 h}{2240}, \frac{10881 h}{5600}, \frac{26001 h}{44800}, \frac{26001 h}{44800}, \frac{10881 h}{5600}, \frac{243 h}{2240}, \frac{141669 h}{89600}, \frac{25713 h}{89600}\right\}$

The solution for this can be found in the attached Mathematica workbook file titled "*Assignment 3 - Mathematica Workbook.nb*".

Question 2 (90 points)

Question 2a (10 points)

To do this, I first had to create a function that depicted the recursive Hermite polynomial given to us in the assignment outline:

$$H_{n+1}(x) = 2xH_n(x) - 2nH_{n-1}(x)$$

with base cases $H_{-1} = 0$, $H_0 = 1$, and $H_1 = 2x$. In my code, I did not include the $n = -1$ base case as the input n is never negative. I did however set the base cases for $n = 0, 1$. The function then has a for loop that begins at $i = 2$ and runs until $i \leq n$. Each loop is simply applying that equation noted above. At the end of the loop, the function returns the Hermite polynomial which will be at the desired n-index.

Next, I made a xmax finder function. This function takes in the j and k values, and sets the initial guess for x to be 5. This is lower than all of the xmax values for $j, k \leq 10$. The program then has an infinite loop that constantly calls the aforementioned Hermite polynomial function with the current x-value and the j and k values, and then compares the integrand to the set machine precision threshold (I used 2.22×10^{-16}, as that is the estimated machine precision value for a double). If the integrand is less than this value, the x-value is increased by 0.1 and the program runs again and this is repeated until the threshold is met, at which point the function returns xmax.

Finally, for my main function, I created some print statements and ran the xmax program for all possible j and k values, and then outputted an easy to read table to view the results.

All of this can be found in the (commented) C file named "*question2a.c*".

Question 2b (20 points)

This question is largely an extension of the previous question. I reused the xmax function and the recursive hermite polynomial functions defined in the question2a.c files. I then made a function named "compute_weights", which is simply a "dictionary" function that takes in a desired n-rule value and a step size (dx) variable and applies the weights found from Mathematica at the end of question 1.

The more interesting function I had to define was my actual numerical method function titled "compute_integral". This function takes in j and k values (for the Hermite Polynomial calculations), the desired n-point integration rule, the desired gridsize (denoted as m), and the xmax value (determined using the xmax function).

The general idea that I used for this integration may not be the way fully intended, but the way that made sense to me was dividing the grid into n sized chunks, and applying the n-point integration rule to each chunk. So for example, say $n = 4$ and $m = 1000$. Then I applied the $n = 4$ integration rule on the 1000/4 sub-intervals, and then added each sub-integral together.

In the main function, I have a bunch of nested loops that test every combination of j and k values, with every n-point integration rule and the various gridsizes. The program takes my laptop roughly 25 seconds to run, which is actually impressive considering how fast C is. It fortunately ran much faster on my desktop CPU. The results are very close to the exact solutions, and I am satisfied that my approach to the integration (whether the intended one or not) works.

For the questions for about the ideal gridsize, I would have to say that it depends. For lower j,k values, often the 100 point gridsize was more then adequate to yield a result that is nearly exactly the same. For example, for conditions $i = j = 1$, $n = 2$ and $m = 100$, the amount of error is 4.44×10^{-16}. The error was comparable regardless of the gridsize for this index.

For higher j,k values on the other hand, the bigger grid size improved results. For conditions $i = j = 8$, $n = 6$, and $m = 100$ the error was 1.22×10^{4}, or 12200 - which is obviously a terrible result. However, for $m = 1000$, the error falls to 7.45×10^{-9}, which is much much better. I would say that a gridsize of at least $m = 1000$ yielded good results across the board, and much more than that was not really necessary.

Similarily, the n-rules also changed the results. In general, the higher n-rules did produce *better* results, but it became fairly negligible after $n = 4$ or so. Interestingly, the higher n-rules performed better with bigger gridsizes, which makes sense based on the integration approach I took to divide the integral into sub-integrals. If I was building a program that was using n-point rules as a numerical method to solve a problem, anything Simpson's 3/8th rule is likely not necessary as it didn't substantially improve the results as much as it increased the complexity required to write such a program.

All of this can be found in the (commented) C file named "*question2b.c*".

Question 2c (60 points)

- Question 2c i) and ii)

First, to solve the first portion of this problem, I needed to use Mathematica to obtain the roots of the degree m Hermite and Chebyshev polynomials. This was fairly trivial as I just used the NSolve function built into Mathematica and evaluated at $x = 0$.

The much more difficult part of this problem was finding the Gaussian quadrature weights. I ended up implementing Equation (1.97) from the course notes, as shown below:

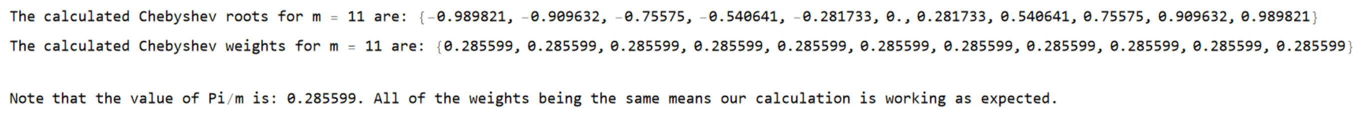
$$w_j = -\frac{k_{N+1}h_N}{k_N \phi'_N(x_j)\phi_{N+1}(x_j)}$$

where k_N represents the leading coefficient of the highest degree polynomial of the Nth polynomial, h represents the normalization constant of the polynomials, x_j represents the root points of the polynomial, phi_N+1 represents the $N+1$th orthonomial polynomial and phi'_N represents the derivative of the Nth polynomial.

For Chebyshev polynomials, Dr. Feder gave us values for $\frac{k_{N+1}}{k_N}$ and h_N, which I am very thankful for because I used his values to reverse-engineer what everything in the formula was meaning. We know that $h_N = \pi/2$ from Equation (1.109) in the course notes, and we know that $\frac{k_{N+1}}{k_N} = 2$.

Next, I used the built in function in Mathematica for Chebyshev polynomials and defined a set of polynomials of order $m+1$. I then evaluated these polynomials with the roots from the Chebyshev polynomials of order m. Next, I defined another set of Chebyshev polynomials or order m, and immediately took the derivative with respect to x and again evaluated this set with the roots of the regular order m polynomials.

Finally, I brought it all together and evaluated Equation 1.97. The results are shown below:



```
The calculated Chebyshev roots for m = 11 are: {-0.989821, -0.909632, -0.75575, -0.540641, -0.281733, 0., 0.281733, 0.540641, 0.75575, 0.909632, 0.989821}
The calculated Chebyshev weights for m = 11 are: {0.285599, 0.285599, 0.285599, 0.285599, 0.285599, 0.285599, 0.285599, 0.285599, 0.285599, 0.285599, 0.285599}

Note that the value of Pi/m is: 0.285599. All of the weights being the same means our calculation is working as expected.
```

As I mentioned in the screenshot of the Mathematica output, the expected result of the Chebyshev weights was $\frac{\pi}{m}$, so all of the results being constant and equal to this value is very exciting and means that Equation (1.97) is implemented correctly.

Now, I applied the same process for the Hermite polynomials. I did the exact same process as I did with the Chebyshev polynomials to find $\phi_{N+1}(x_j)$ and $\phi'_N(x_j)$, however unlike before I was not given values for $\frac{k_{N+1}}{k_N}$ and h_N so I had to figure those out myself.

Finding the normalization constant didn't end up being too difficult, as I realized that Hermite polynomials have the following property:

$$I_{jk} = \int_{-\infty}^{\infty} e^{-x^2} H_j(x) H_k(x) = \begin{cases} 2^j j! \sqrt{\pi} & j = k \\ 0 & j \neq k \\ \end{cases}$$

Therefore, we know that $h_N = 2^j j! \sqrt{\pi}$. To find then ratio of $\frac{k_{N+1}}{k_N}$, the easiest way is visualize the recursive formula of the Hermite polynomial:

$$H_{n+1}(x) = 2xH_n(x) - 2nH_{n-1}(x)$$

From this, we can see that the highest degree term in H_N will be:

$$H_N(x) = 2^N x^N + ...$$

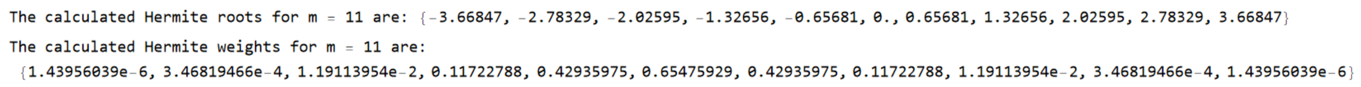
And likewise, the highest degree term in H_{N+1} will be:

$$H_{N+1}(x) = 2^{N+1} x^{N+1} + ...$$

So the ratio of the leading coefficients will simply be:

$$\frac{k_{N+1}}{k_N} = \frac{2^{N+1}}{2^N} = 2$$

Yay! Okay, now I brought everything together and evaluated Equation 1.97. The results are shown below. I also made a custom function that will output the weight values in scientific notation (as I had to use this in the next portion).



```
The calculated Hermite roots for m = 11 are: {-3.66847, -2.78329, -2.02595, -1.32656, -0.65681, 0., 0.65681, 1.32656, 2.02595, 2.78329, 3.66847}
The calculated Hermite weights for m = 11 are:
 {1.43956039e-6, 3.46819466e-4, 1.19113954e-2, 0.11722788, 0.42935975, 0.65475929, 0.42935975, 0.11722788, 1.19113954e-2, 3.46819466e-4, 1.43956039e-6}
```

I was pretty satisfied with these values, and when I compared these results with other methods such as "Gauss–Hermite quadrature" (more information can be found at https://en.wikipedia.org/wiki/Gauss%E2%80%93Hermite_quadrature) and they yielded identical results so I was ready to move onto the C code in part iii.

The solution for this can be found in the attached Mathematica workbook file titled "*Assignment 3 - Mathematica Workbook.nb*".

- Question 2c iii)

Back in C, I again reused the Hermite polynomial function I defined back in 2a. Next I implemented a simple integrand function that takes in a "j" ,"k" and "x" value. It then returns $H_j(x) \cdot H_k(x)$. Note that for the longest time, I kept the e^{-x^2} term in this function and didn't understand why my function was not working, but I eventually realized that the weights calculate already includes this factor so I was double dipping by including it here again.

Next, I defined the actual integration function, titled "gaussian_quadrature" that takes in "j" ,"k" values. This function uses the weights and roots determined from part ii, and then applies equation the summation depicted in (1.50) to integrate the function. It then returns the integral.

Finally, in the main function, I created some print statements to tabulate the results, and some for loops that ran the program for all possible j and k values, whilst comparing to the analytical solution exactly as was done in 2b.

All of this can be found in the (commented) C file named "*question2c.c*".

- Question 2c iv)

Back to Mathematica. Now we had to build a script that solves the problem in part i) but instead of solving the roots/weights directly like we did before, we were to implement the $\mathcal{J}$ matrix defined as Equation (1.100) in the course notes. I'm not going to lie to you, this question was quite daunting to me, as Equation (1.100) in the form its shown in the notes was quite intimidating, so I did some research on my own on what this matrix even was.

The Jacobi Matrix, as shown below:

$$J = \begin{bmatrix} a_0 & b_1 & 0 & \cdots & 0 \\ b_1 & a_1 & b_2 & \cdots & 0 \\ 0 & b_2 & a_2 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & b_{n-1} \\ 0 & 0 & 0 & b_{n-1} & a_{n-1} \end{bmatrix}$$

is a "symmetric linear operator acting on sequences which is given by an infinite tridiagonal matrix". Note that this is completely different from the *Jacobian Matrix* that we learn about in vector calculus.

Supposedly, the most common use for this operator to to specify systems of orthonormal polynomials - which is perfect for our purposes as Chebyshev and Hermite polynomials are both sets of orthonormal polynomials!

To build a Jacobi matrix that will solve the problem that we're setting out to solve, we need to determine appropiate a_n and b_n values to use in our Jacobi operator. Once complete, we can apply the Golub-Welsch algorithm, which tells us that:

- The **eigenvalues** of this matrix are *exactly* the roots of the n-th orthogonal polynomial
- And the **eigenvectors** are used to determine the corresponding weights.

The Golub-Welsch algorithm uses a very specific Jacobi Matrix of the form:

$$\mathcal{J} = \begin{bmatrix} a_0 & \sqrt{b_1} & 0 & \cdots & 0 \\ \sqrt{b_1} & a_1 & \sqrt{b_2} & \cdots & 0 \\ 0 & \sqrt{b_2} & a_2 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \sqrt{b_{n-1}} \\ 0 & 0 & 0 & \sqrt{b_{n-1}} & a_{n-1} \end{bmatrix}$$

In order to apply this algorithm, we need to observe the three-term recurrence relation that depicts many families of orthogonal polynomials:

$$p_{n+1}(x) = (x - a_n)p_n(x) - b_n p_{n-1}(x)$$

If we compare this equation to the given recurrence relations for Hermite and Chebyshev polynomials, we'll be able to find our sets of a_n and b_n values. We'll first observe the Hermite polynomial case:

$$H_{n+1}(x) = 2xH_n(x) - 2nH_{n-1}(x)$$

Unfortunately, this case is non-symmetric (coefficient in front of the x value of the H_n term is a problem), and we have to normalize this recurrence to match the three term recurrence relation observed above to be able to use the Golub-Welsch algorithm. Lets define a new set of polynomials, simply denoted as p_n, in which the following is true:

$$p_n(x) = \frac{H_n(x)}{2^n} \qquad \longrightarrow \qquad p_{n-1}(x) = \frac{H_{n-1}(x)}{2^{n-1}} \;\; , \;\; p_{n+1}(x) = \frac{H_{n+1}(x)}{2^{n+1}}$$

Now, substitute this new set of polynomials into the Hermite relation:

$$\left[2^{n+1}p_{n+1}(x)\right] = 2x\left[2^{n}p_{n}(x)\right] - 2n\left[2^{n-1}p_{n-1}(x)\right]$$

$$p_{n+1}(x) = \frac{2^{n+1}xp_n(x)}{2^{n+1}} - \frac{2^n np_{n-1}(x)}{2^{n+1}}$$

$$p_{n+1}(x) = xp_n(x) - \frac{n}{2}p_{n-1}(x)$$

Now, looking at the relation, we can see that all of the a_n values will be equal to 0, and all of the b_n values are equal to n/2. Therefore, if we apply it to the specific Jacobi matrix, we need to make the diagonal values all equal to 0 and the sub diagonal values all equal to $\sqrt{n/2}$. This will allow us to use the Golub-Welsch algorithm to find the roots and weights of the Hermite polynomials. Now lets take a look at the Chebyshev recurrence relation from all the way back in Assignment 1:

$$T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x)$$

Again, this relation is not normalized, but if we normalize it with a new set of polynomials the exact same way we normalized the Hermite polynomials, we find:

$$p_{n+1}(x) = xp_n(x) - \frac{1}{4}p_{n-1}(x)$$

which gives the familiar result of 0 for all a_n values along the diagonal, and the constant value of 1/2 (as $\sqrt{1/4} = 1/2$) along the sub-diagonals for the Chebyshev version of the specialized Jacobi Matrix. Note that this is only true for $n \geq 2$. As is pointed out in Equation (1.109) and (1.114) in the course notes, the Chebyshev polynomials have a special normalization at their lowest possible case that is twice as much as for their regular case. Therefore, in the case of $n = 1$, the relation is instead:

$$p_2(x) = xp_1(x) - \frac{1}{2}p_0(x)$$

This is due to the fact that T_0 is not orthogonal to the rest of the Chebyshev Polynomials if you apply the same normalization that you apply for T_n for $n \geq 1$, and therefore the recurrence relation needs to be adjusted as shown.

Finally, now that all of that preamble is out of the way, I just had to build the final script which was not too difficult. I defined a HermiteJ function that created an array that was m long filled with only zeros for the diagonal, another array that was $m - 1$ long filled with $\sqrt{j/2}$ values, and then I put these arrays together in the DiagonalMatrix built in function and assigned them accordingly. I made an identical ChebyshevJ function that used 1/2 for the sub-diagonals (with $1/\sqrt{2}$ for the first element of the sub-diagonals for the special first case).

Next, I defined the QuadratureRules function that used the built in Eigensystem function to extract the roots from the Hermite Jacobi matrix (remember that according to the algorithm, the roots are the eigenvalues, and we need the eigenvectors to compute the weights). Once these values are extracted, the weights were calculated using the following equation:

$$w_j = h_0 \left(\phi_1^{(j)} \right)^2$$

where h_0 is the normalization at $n = 0$ and $\phi_1^{(j)}$ is the first element in the eigenvector $\phi^{(j)}$.

For the Hermite Polynomials, we know that the normalization is equivalent to:

$$h_n = \sqrt{\pi} 2^n n!$$

$$h_0 = \sqrt{\pi} 2^0 0!$$

$$h_0 = \sqrt{\pi}$$

And for Chebyshev Polynomials (using Equation 1.84):

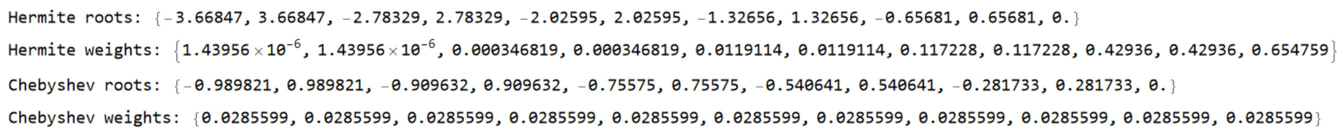
$$h_0 = \int_{-1}^{1} \frac{T_0^2}{\sqrt{1-x^2}} dx$$

$$h_0 = \int_{-1}^{1} \frac{dx}{\sqrt{1-x^2}}$$

$$h_0 = \pi$$

So to find the weights, we simply multiply the first element in the extracted eigenvectors from the Eigensystem function by $\sqrt{pi}$ for the Hermites and π for the Chebyshevs respectively.

I then made some print statements that output the results, and they matched the output from part i and ii meaning that the script is working correctly. An image of these results are shown below:



```
Hermite roots: {-3.66847, 3.66847, -2.78329, 2.78329, -2.02595, 2.02595, -1.32656, 1.32656, -0.65681, 0.65681, 0.}
Hermite weights: {1.43956×10^-6, 1.43956×10^-6, 0.000346819, 0.000346819, 0.0119114, 0.0119114, 0.117228, 0.117228, 0.42936, 0.42936, 0.654759}
Chebyshev roots: {-0.989821, 0.989821, -0.909632, 0.909632, -0.75575, 0.75575, -0.540641, 0.540641, -0.281733, 0.281733, 0.}
Chebyshev weights: {0.0285599, 0.0285599, 0.0285599, 0.0285599, 0.0285599, 0.0285599, 0.0285599, 0.0285599, 0.0285599, 0.0285599, 0.0285599}
```

The solution for this can be found in the attached Mathematica workbook file titled "*Assignment 3 - Mathematica Workbook.nb*"