# Homework 5

November 27, 2020

---

## 1 Problem 1

We can determine on which parts of the $(n, x)$ plane that the given recurrence relation is stable by rearranging terms, with the following observation. In particular, we know that the following equalities hold for any recurrence relation $E_n(x)$

$$E_n(x) = E_n^{\text{true}}(x) + \epsilon_n(x) \ ,$$

and also that

$$E_{n+1}(x) = E_{n+1}^{\text{true}}(x) + \epsilon_{n+1}(x) \ .$$

In these equations, we are claiming that the value of the recursive function, for either $n$ or $n+1$, is equal to the contribution of the actual value of the function in addition to error terms which we denote with $\epsilon_n(x), \epsilon_{n+1}(x)$. From the problem, we know that $E_{n+1}(x)$ satisfies

$$E_{n+1}(x) = \frac{1}{n}\bigg(e^{-x} - xE_n(x)\bigg) \ ,$$

which from our second equation implies that

$$\frac{1}{n}\bigg(e^{-x} - xE_n(x)\bigg) = E_{n+1}^{\text{true}}(x) + \epsilon_{n+1}(x) \ .$$

Furthermore, we can obtain an expression for $\epsilon_{n+1}(x)$. We will make use of the fact that $E_{n+1}^{\text{true}}(x)$ is a recurrence relation. It satisfies the same equality that we substituted in for $E_n(x)$. We get that

$$\begin{aligned}\frac{1}{n}\bigg(e^{-x} - xE_n(x)\bigg) &= \frac{1}{n}\bigg(e^{-x} - xE_n(x)\bigg) + \epsilon_{n+1}(x) \\ &= \frac{e^{-}x}{n} - \frac{x}{n}E_n^{\text{true}}(x) + \frac{\epsilon_n(x)}{n}\end{aligned}$$

Above, observe that we substituted for $E_n(x)$ using the first equation. Altogether, eliminating terms gives an expression for $\epsilon_{n+1}(x)$, which entails that

$$\frac{-x}{n}\epsilon_n(x) = \epsilon_{n+1}(x) \ .$$

We can now check for stability by looking at the ratio

$$\left|\frac{\epsilon_{n+1}(x)}{\epsilon_n(x)}\right| .$$

We know that

$$\left|\frac{\epsilon_{n+1}(x)}{\epsilon_n(x)}\right| = \left|\frac{\frac{x}{n}\epsilon_n(x)}{\epsilon_n(x)}\right| = \frac{x}{n} \ .$$

So $\frac{x}{n} = f(x,n)$ is the function that we were looking for. Now, to check for stability, we will see whether the ratio is greater or less than 1. Directly, we know that $\frac{x}{n} < 1 \Rightarrow E_n(x)$ is stable, while on the other hand $\frac{x}{n} > 1 \Rightarrow E_n(x)$ is unstable.

From the expression for $f(x,n)$, this function, which represents the ratio between the error terms, is stable for decreasing $n$ precisely for the $x$ that allow $f(x,n) < 1$ when we increase $n$. So if we look at $n = 1, 2, \cdots$, we know that $f(x,1) = x/1, f(x,2) = x/2, \cdots$. In the $(n,x)$ plane, these $x$ need to be larger than $n$ so that $f(x,n) < 1$ consistently. On the other hand, we know that the given recurrence relation can also be stable for decreasing $n$. In this remaining case, we need the opposite to occur. Namely, we want that there are values of $x$ that are smaller than the decreasing values of $n$. Intuitively, the expression $\frac{x}{n}$ could remain stable when we obtain smaller and smaller $n$, but we need to make $x$ smaller than $n$ in order to preserve the condition $|f(x,n)| < 1$ that we have discussed at length. In short, in the $(n,x)$ plane the recurrence relation $E_n(x)$ is stable for increasing $n$ for the values of $x, n$ in the $(n,x)$ plane that lie in the region to the left of the diagonal line, ie $x = n$, and that $E_n(x)$ would be stable for decreasing $n$ for values in the $(n,x)$ plane that lie in the region to the right of the diagonal line. Geometrically, we know that lying to the left of the diagonal line implies that $n > x$, and that $n < x$ to the right of the diagonal.

# 2 Problem 2

## 2.1 Part A

From **302**, we know that Dawson's Integral is given by

$$F(x) = e^{-x^2}\int_{t=0}^{x} e^{t^2}\mathrm{d}t \ .$$

For reference, $F(1.1) = e^{-1.1^2}\int_{t=0}^{1.1} e^{t^2}\mathrm{d}t \approx 0.526109\cdots$, and that $F(0.1) \approx 0.101348\cdots$. To get the algorithm that we want, we will make use of the pseudocode for Lentz's Algorithm that is given on **208**. We can implement this code with the following

```
static double _kf_gammaq(double s, double z)
{
    int j;
    double C, D, f;
    f = 1. + z - s; C = f; D = 0.;
    for (j = 1; j < 100; ++j) {
        double a = j * (s - j), b = (j<<1) + 1 + z - s, d;
        D = b + a * D;
        if (D < pow(10,-30)) D = pow(10,-30);
        C = b + a / C;
        if (C < pow(10,-30)) C = pow(10,-30);
        D = 1. / D;
        d = C * D;
        f *= d;
        if (fabs(d - 1.) < pow(10,-5)) break;
    }
    return (s * log(z) - z - kf_lgamma(s) - log(f));
}
```

More or less, the code that I have included in C above takes into account the necessary steps that we must take. Also, as mentioned on **208**, observe that I have set the values **tiny** and **eps** equal to what is recommended. In addition to these recommended values, observe in the code that I left $f$ blank; depending on the numerical situation that we are faced with, the user must decide what the initial term of the continued fraction would be, all of which depends on the value of $x$ that they are evaluating in $F(x)$.

Now, we will call the function in order to evaluate the given continued fraction. Also, observe that the function call for kflgamma($s$) that we included in the last line of the method above is defined as

```
double kf_lgamma(double z)
{
    double x = 0;
    x += 0.1659470187408462e-06 / (z+7);
    x += 0.9934937113930748e-05 / (z+6);
    x -= 0.1385710331296526     / (z+5);
    x += 12.50734324009056      / (z+4);
    x -= 176.6150291498386      / (z+3);
    x += 771.3234287757674      / (z+2);
    x -= 1259.139216722289      / (z+1);
    x += 676.5203681218835      / z;
    x += 0.9999999999995183;
    return log(x) - 5.58106146679532777 - z + (z-0.5) * log(z+6.5);
}
```

From the code that I have included, it is clear that we have to hard code some of the other inputs that we need before evaluating the continued fraction. In this particular case, we know that applying the Modified Lentz Algorithm to the given continued fraction can produce a desired error of $10^{-7}$ by running this algorithm in XCode directly. However, before running the function that I have included at the top of this page, we have to take into account the case that we have. To properly evaluate Dawson's Integral, it is important to recognize the **tolerance** that we want to achieve.

More specifically, in each case for $F(1.1)$ and $F(0.1)$, it is clear that we can obtain the continued fractions for the quantities that would give us the desired precision of the algorithm. In each of the evaluations $F(1.1)$ and $F(0.1)$, the rational function approximation that satisfies the prescribed algorithmic precision of $10^{-7}$ are the following continued fractions

$$1+\cfrac{1}{-2+\cfrac{1}{-9+\cfrac{1}{-13+\cfrac{1}{-3+\cfrac{1}{-2+\cdots}}}}} \approx 0.526108936\cdots$$

,

for $F(1.1)$, and

$$0+\cfrac{1}{10+\cfrac{1}{-8+\cfrac{1}{2+\cfrac{1}{13+\cdots}}}} \approx 0.10134856\cdots$$

for $F(0.1)$. As mentioned in the directions for **A**, observe that

$$\left| e^{-1.1^2} \int_{t=0}^{1.1} e^{t^2} \mathrm{d}t - 0.526108936\cdots \right| \approx 10^{-7}, \text{ and also that}$$
$$\left| e^{-0.1^2} \int_{t=0}^{0.1} e^{t^2} \mathrm{d}t - 0.10134856\cdots \right| \approx 10^{-7} \ .$$

So with the pseudocode from **208** and some diligence, we were able to evaluate the continued fraction for Dawson's Integral. In addition to listing the continued fractions that we obtained from the numerical values for $F(1.1)$ and $F(0.1)$, we observe that the **code** that I have provided above depends on the initial term of the continued fraction expansion. Namely, from the code in C that I have included, the initial value of $f$ and $b_0$ that we are defining depends on the value of $x$ that we are substituting for in the continued fraction expansion for $F(x)$ that we are given.

From the calculations of the 2 continued fractions that we have encountered for $F(1.1)$ and $F(0.1)$, we can compare our results against another method. From the **Simpy** documentation, we can try out a method, called 'continued fraction reduce' that allows us to determine accurate numerical expressions for $F(1.1)$ and $F(0.1)$ by instantiating an array which includes the terms that appear in the continued fraction expansion, as shown above. In the case of $F(1.1)$, we can compare the algorithmic efficiency of this second method provided by the Simpy Documentation. We get that

```
>>> continued_fraction_reduce([1,-2,-9,-13,-3,-2])
937/1781
```

Numerically, $\frac{937}{1781} \approx 0.5261089275\cdots$. From this numerical value that is output from 'continued fraction reduce' it is clear that this method is a little less algorithmically precise in comparison to the numerical result that we obtained with the Modified Lentz Algorithm above. Similarly, for $F(0.1)$, repeating the same code gives that

```
>>> continued_fraction_reduce([0,10,-8,2,13])
203/2003
```

From this output, $\frac{203}{2003} \approx 0.101347978\cdots$. In this case again, we observe that the Modified Lentz Algorithm is slightly preferable over the Simpy Method. All of this is in line with general remarks in the book, around **206**, which claim that the Modified Lentz Algorithm is the best choice anyways.

## 2.2 Part B

It's optional for a reason!

# 3 Problem 3

## 3.1 Part A

### 3.1.1 Determining the Number of Terms for Absolute Error of $10^{-9}$

From Python, we can directly form the Chebyshev polynomial. In particular, the 'chebyfit' method allows us to sample a user-defined number of points spaced in $[\frac{-1}{2}, 2]$, and we can also see the error. From this method, we see that

```
>>> poly , err = chebyfit(lambda x: 1/x - 1/(x*(1+x)**(2/3)) , [-1/2,2] , 23 , error = True)
>>> nprint(err)
2.28939e-9
```

and also,

```
>>> poly , err = chebyfit(lambda x: 1/x - 1/(x*(1+x)**(2/3)) , [-1/2,2] , 24 , error = True)
>>> nprint(err)
9.5152e-10
```

From this output, we see that we would need between 22 and 23 terms while forming the Chebyshev approximation. With these number of terms, we are able to achieve the specified error of $10^{-9}$. In short, the number of points that we need to use to obtain the desired error of $10^{-9}$ is 23 terms.

Now, we can evaluate the polynomial at 100 equally spaced points. In Python, we can evaluate the polynomial that we obtained in the previous code. We know that the polynomials that we got for $N = 23$ and $N = 24$ points are the following.

**23 points sampled**

```
>>> poly , err = chebyfit(lambda x: 1/x - 1/(x*(1+x)**(2/3)) , [-1/2,2],23, error = True)
>>> print(np.poly1d(poly))
           22             21            20           19           18
2.994e-05 x  - 0.0005469 x  + 0.004549 x  - 0.02276 x  + 0.07634 x
           17          16          15          14          13
 - 0.1809 x  + 0.3115 x  - 0.3981 x  + 0.3908 x  - 0.3261 x
           12          11          10          9          8          7
 + 0.2854 x  - 0.2969 x  + 0.3261 x  - 0.3435 x + 0.3525 x - 0.3639 x
           6         5          4          3          2
 + 0.3798 x - 0.399 x + 0.4225 x - 0.4527 x + 0.4938 x - 0.5556 x + 0.6667
```

**24 points sampled**

```
>>> poly , err = chebyfit(lambda x: 1/x - 1/(x*(1+x)**(2/3)) , [-1/2,2],24, error = True)
>>> print(np.poly1d(poly))
            23             22           21           20           19
-1.985e-05 x  + 0.0003775 x  - 0.00328 x  + 0.01722 x  - 0.06084 x
           18         17          16          15          14          13
 + 0.1527 x  - 0.28 x  + 0.3821 x  - 0.3976 x  + 0.3393 x  - 0.2846 x
           12          11          10          9          8          7
 + 0.2826 x  - 0.3114 x  + 0.3331 x  - 0.3422 x + 0.3508 x - 0.3638 x
         6         5          4          3          2
 + 0.38 x - 0.399 x + 0.4225 x - 0.4527 x + 0.4938 x - 0.5556 x + 0.6667
```

### 3.1.2 Determining the Accuracy of our 23 degree polynomial by evaluating it at 100 equally spaced points

Now, we will create the 100 equally spaced points that we will use to study the error. Depending on our implementation, we can either use linspace or take the maximum of all the errors, as mentioned in the documentation for Simpy. So it is possible that

```
>>> np.linspace(-1/2,2,num=100)
```

or with the polynomial objects that we have created, we can evaluate the polynomial that we obtained when sampling $N = 24$ points. We get that this error is

```
>>> error = lambda x: abs(1/x-1/(x*(1+x)**(2/3)) - polyval(poly,x))
>>> nprint(max([error(1+n/100.) for n in range(100)]),12)
2.39500030919e-10
```

Therefore, in the case of this function, sampling more points of the $N = 24$ points (the degree 23 polynomial) returns a smaller error, but the magnitude of this error is not much different than what we obtained in the earlier cases. This is significant because it reflects how taking the $N = 24$ polynomial that we obtained, even when evaluated at 100 equally points, cannot improve too much upon the original error that we listed earlier in this first part of **3**.

## 3.2 Part B

We will plot the coefficients $c_i$ versus $i$. From the plot, observe that we calculated the coefficients of the Chebyshev series by using a slightly different method in Python. The 'poly2cheb' method returns the coefficients of the Chebyshev series from which we want to construct the plot. So we run the code for the number of points that we got in **A**. In the code **below**, observe that I used an alternative command that gives the same result, it is also listed on the Python documentation page.

For the **24 points sampled** case, we will obtain the coefficients of the Chebyshev series. **Directly** from the Scipy page, the coefficients of the Chebyshev Series are given by the array

```
>>> p = P.Polynomial([-1.98e-05, 0.0003775, -0.00328 , 0.01722, -0.06084 , 0.1527 , -0.28 , 0.3821 ,
>>> c = p.convert(kind=P.Chebyshev)
>>> c
Chebyshev([-7.51316576e-01,  1.56038712e+00, -1.27641520e+00,  1.14648576e+00,
       -7.99011734e-01,  6.45724708e-01, -3.90537165e-01,  2.97867509e-01,
       -1.58843568e-01,  1.17510264e-01, -5.48683901e-02,  3.93999411e-02,
       -1.57132830e-02,  1.09199254e-02, -3.60746975e-03,  2.42944939e-03,
       -6.37946701e-04,  4.16821885e-04, -8.15122604e-05,  5.17163992e-05,
       -6.69193268e-06,  4.12685871e-06, -2.64930725e-07,  1.58953667e-07], domain=[-1.,  1.], windo
```

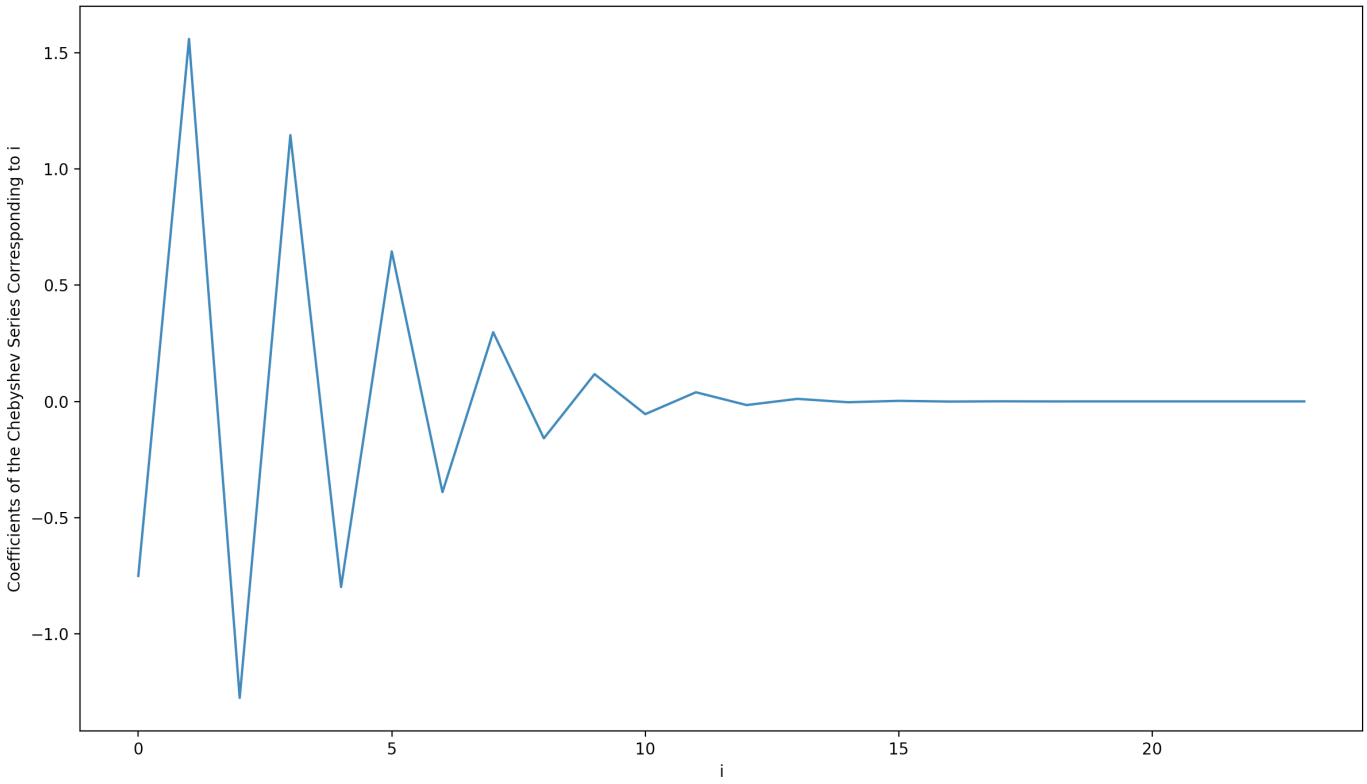
With these coefficients $c$, the polynomial that we are looking for is precisely

```
>>> p_2 = np.poly1d([-7.51316576e-01,  1.56038712e+00, -1.27641520e+00,  1.14648576e+00,
...        -7.99011734e-01,  6.45724708e-01, -3.90537165e-01,  2.97867509e-01,
...        -1.58843568e-01,  1.17510264e-01, -5.48683901e-02,  3.93999411e-02,
...        -1.57132830e-02,  1.09199254e-02, -3.60746975e-03,  2.42944939e-03,
...        -6.37946701e-04,  4.16821885e-04, -8.15122604e-05,  5.17163992e-05,
...        -6.69193268e-06,  4.12685871e-06, -2.64930725e-07,  1.58953667e-07])
```

Explicitly, the polynomial is

```
>>> print(np.poly1d(p_2))
          23         22          21          20          19           18
-0.7513 x  + 1.56 x  - 1.276 x  + 1.146 x  - 0.799 x  + 0.6457 x
           17          16          15          14           13
 - 0.3905 x  + 0.2979 x  - 0.1588 x  + 0.1175 x  - 0.05487 x
           12           11           10           9           8
 + 0.0394 x  - 0.01571 x  + 0.01092 x  - 0.003607 x + 0.002429 x
              7             6             5             4             3
 - 0.0006379 x + 0.0004168 x - 8.151e-05 x + 5.172e-05 x - 6.692e-06 x
              2
 + 4.127e-06 x - 2.649e-07 x + 1.59e-07
```

So with the 'poly2cheb' method in Python, we have obtained the coefficients of the Chebyshev series from the polynomials that I have listed. The plot is included below, for $i = 0$ up to $i = 23$.




So this plot captures the type of relationship that we would expect. That is, the coefficients $c_i$ of the Chebyshev Series decrease for larger $i$, which is brought across by the values of successively higher coefficients $c_i$ that we have plotted above. As mentioned yesterday, we would expect this relationship between the $c_i$ and $i$ because as we continue adding more terms to the polynomials that we are using, our error is decreasing so the coefficients $c_i$ in front of the polynomial terms for higher $i$ will be smaller so that the smaller errors that we are obtaining for $f(x)$ continue decreasing. Simply put, their contribution will be smaller as not to offset the decreasing error that we obtained from using a higher number of points that we sampled from $\mathbf{A}$.

### 3.3 Part C

We can compute the derivative with a function from Python called 'chebder'. When supplied with an array of the coefficients of a Chebyshev series, the function is able to compute the coefficients of the derivative of the Chebyshev series. From **B**, we have the coefficients of the the Chebyshev series. With this function, we call the following in Terminal, setting $c$ equal to the list of the coefficients of the Chebyshev Series that we obtained in **B**.

```
>>> c = .....
>>> C.chebder(c)
array([ 1.19910919e+01, -2.03248216e+01,  2.08614096e+01, -1.52191608e+01,
        1.39824950e+01, -8.82706689e+00,  7.52524793e+00, -4.14062091e+00,
        3.35510281e+00, -1.59912382e+00,  1.23991805e+00, -5.01756015e-01,
        3.73119349e-01, -1.24637223e-01,  8.92012889e-02, -2.36280701e-02,
        1.63178072e-02, -3.21377563e-03,  2.14586310e-03, -2.79334259e-04,
        1.80639935e-04, -1.16569519e-05,  7.31186868e-06])
```

With this method, we have now obtained the coefficients of the Chebyshev Derivative. With our work from **A**, we can now compare the behavior of this polynomial to that of the analytic derivative. But for convenience, we know that the derivative of the Chebyshev series can also be obtained by differentiating the poly1d object $p_2$ from **B**. We have that

```
>>> print(np.poly1d(p3))
         22          21         20          19          18          17
-17.28 x  + 34.33 x  - 26.8 x  + 22.93 x  - 15.18 x  + 11.62 x
           16          15          14          13           12           11
 - 6.639 x  + 4.766 x  - 2.383 x  + 1.645 x  - 0.7133 x  + 0.4728 x
            10          9           8           7            6
 - 0.1728 x  + 0.1092 x - 0.03247 x + 0.01944 x - 0.004466 x
              5             4             3             2
 + 0.002501 x - 0.0004076 x + 0.0002069 x - 2.008e-05 x + 8.254e-06 x - 2.649e-07
```

So this gives an expression for the derivative on $[-\frac{1}{2}, 2]$, which answers part of the **first** question. Analytically, we know that the derivative is

$$f'(x) = \frac{-1}{x^2} - \frac{\frac{-1}{x^2}(1+x)^{\frac{2}{3}} - \frac{1}{x}\frac{2}{3}(1+x)^{\frac{1}{3}}}{(1+x)^{\frac{4}{3}}}.$$

Again for convenience (and to be careful and sure!), Python gives that the derivative of $f$ is

```
>>> f = 1/x -1/(x*(1+x)**(2/3))
>>> fprime = f.diff(x)
>>> fprime
0.666666666666667*(x + 1)**(-1.66666666666667)/x + (x + 1)**(-0.666666666666667)/x**2 - 1/x**2
```

Running similar code to what I have included for **A** implies that the error is of the form

```
>>> error = lambda x: abs(fprime - polyval(p3,x))
```

which on $[-\frac{1}{2}, 2]$ attains a maximum of $\approx 2 \times 10^{-1}$, which is the desired relative error. Furthermore, it is important to note that this error . an change depending on the number of points that we use to evaluate both the Chebyshev and Analytic derivatives. From the case that I have presented, it is then clear that numerically, the

Chebyshev Approximation is helpful to capture the behavior of our function $f(x)$. However, the advantages of the Chebyshev polynomials only come to a point. There is already some error apparent in the results that I presented in **A**, and continuing to differentiate and/or integrate the Chebyshev polynomial that we have previously obtained would only magnify certain errors. Therefore, we will obtain decreased accuracy of the derivative of $f(x)$ because near the origin, the function is discontinuous and sampling a higher number of points reflects the rapid change of behavior of $f(x)$ for $x \approx 0$. In comparison to the results that we have included in **A** and **B**, we observe that our approximation, from the analytic derivative that we calculated and have compared against the original function $f(x)$, is not as good, but that it can still capture some qualities of the function's behavior, but at least not like the previous approximations that we have used for $f(x)$ in **3**.

# 4 Code, etc

## 4.1 Code for 2A

```
double kf_lgamma(double z)
{
    double x = 0;
    x += 0.1659470187408462e-06 / (z+7);
    x += 0.9934937113930748e-05 / (z+6);
    x -= 0.1385710331296526     / (z+5);
    x += 12.50734324009056      / (z+4);
    x -= 176.6150291498386      / (z+3);
    x += 771.3234287757674      / (z+2);
    x -= 1259.139216722289      / (z+1);
    x += 676.5203681218835      / z;
    x += 0.9999999999995183;
    return log(x) - 5.58106146679532777 - z + (z-0.5) * log(z+6.5);
}


static double kf_gammaq(double s, double z)
{
    int j;
    double C, D, f;
    f = pow(10,-30); C = f; D = 0.;
    // Modified Lentz's algorithm for computing continued fraction
    // See Numerical Recipes in C, 2nd edition, section 5.2
    for (j = 1; j < 100; ++j) {
        double a = j * (s - j), b = (j<<1) + 1 + z - s, d;
        D = b + a * D;
        if (D < pow(10,-30)) D = pow(10,-30);
        C = b + a / C;
        if (C < pow(10,-30)) C = pow(10,-30);
        D = 1. / D;
        d = C * D;
        f *= d;
        if (fabs(d - 1.) < pow(10,-5)) break;
    }
    return (s * log(z) - z - kf_lgamma(s) - log(f));

}
```

```
int main(int argc, const char * argv[]) {
    // insert code here...
    int continued_fraction_evaluation= kf_gammaq(2,10);
}
```

### 4.2 Code for Making the Plot for 3B

```
>>> import matplotlib.pyplot as plt
>>> xs = [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23]
>>> ys = [-7.51316576e-01,  1.56038712e+00, -1.27641520e+00,  1.14648576e+00,
...        -7.99011734e-01,  6.45724708e-01, -3.90537165e-01,  2.97867509e-01,
...        -1.58843568e-01,  1.17510264e-01, -5.48683901e-02,  3.93999411e-02,
...        -1.57132830e-02,  1.09199254e-02, -3.60746975e-03,  2.42944939e-03,
...        -6.37946701e-04,  4.16821885e-04, -8.15122604e-05,  5.17163992e-05,
...        -6.69193268e-06,  4.12685871e-06, -2.64930725e-07,  1.58953667e-07]
>>> plt.xlabel('i')
Text(0.5,0,'i')
>>> plt.ylabel('Coefficients of the Chebyshev Series Corresponding to i')
Text(0,0.5,'Coefficients of the Chebyshev Series Corresponding to i')
>>> plt.plot(xs,ys)
[<matplotlib.lines.Line2D object at 0x11b92cc88>]
>>> plt.show()
```

### 4.3 Code for 3C

```
>>> p3 = np.polyder(p_2)
>>> p3
poly1d([-1.72802812e+01,  3.43285166e+01, -2.68047192e+01,  2.29297152e+01,
       -1.51812229e+01,  1.16230447e+01, -6.63913180e+00,  4.76588014e+00,
       -2.38265352e+00,  1.64514370e+00, -7.13289071e-01,  4.72799293e-01,
       -1.72846113e-01,  1.09199254e-01, -3.24672278e-02,  1.94355951e-02,
       -4.46562691e-03,  2.50093131e-03, -4.07561302e-04,  2.06865597e-04,
       -2.00757980e-05,  8.25371742e-06, -2.64930725e-07])
>>> print(np.poly1d(p3))
```

## 5 References

[1] Github Repository for the Lentz Modified Algorithm code, from *https://github.com/vibansal/crisp/blob/master/FET/l*