Arun Suresh

Computational Physics 1 - Midterm Exam

(1 a) The given function is

$$g(x) = \left[\cos(5\pi x) - e^{\frac{-x^2}{2}} \sin\left(\frac{\pi}{2}x\right)\right] \frac{\sin(\pi x)}{x}$$

The function was sampled with 200 data points in between -10 to 10 and is plotted below.

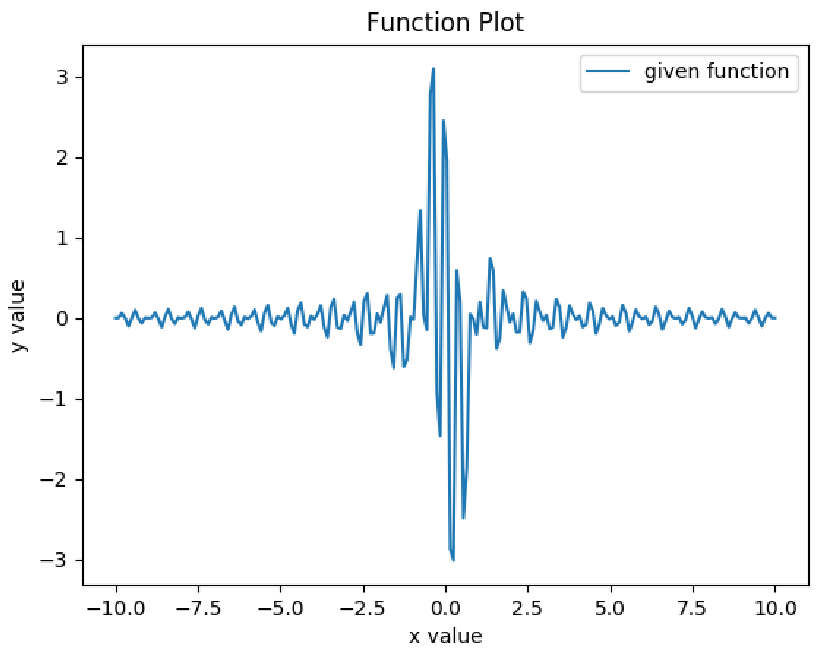


Figure 1: Function Plot

(1 b) The derivative of the above function was calculated to be

$$g'(x) = \left[\cos(5\pi x) - e^{-x^2/2} \sin\left(\frac{\pi}{2}x\right)\right]\left(- \frac{\sin(\pi x)}{x^2} + \frac{1}{x}\pi \cos(\pi x)\right) +$$

$$\frac{\sin(\pi x)}{x}(-5\pi \sin(5\pi x) + xe^{-(x^2)/2} \sin(0.5\pi x) + 0.5\pi \cos(0.5\pi x)e^{(-(x{*}{*}2)/2)})$$

This analytic solution was plotted for the same sampled data as before, and is reproduced below.

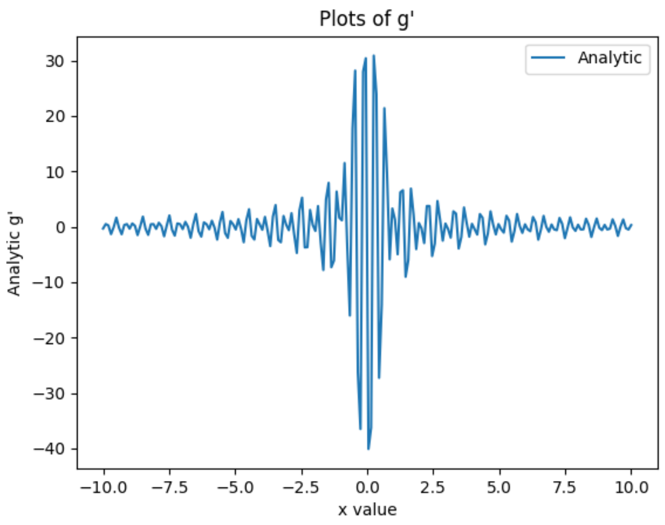


Figure 2: Analytic derivative

(1 c) The forward, backward and central difference methods were employed for the function $g(x)$ to compute the derivative. The overlapping plots of the three methods, along with the analytic solution is presented below.

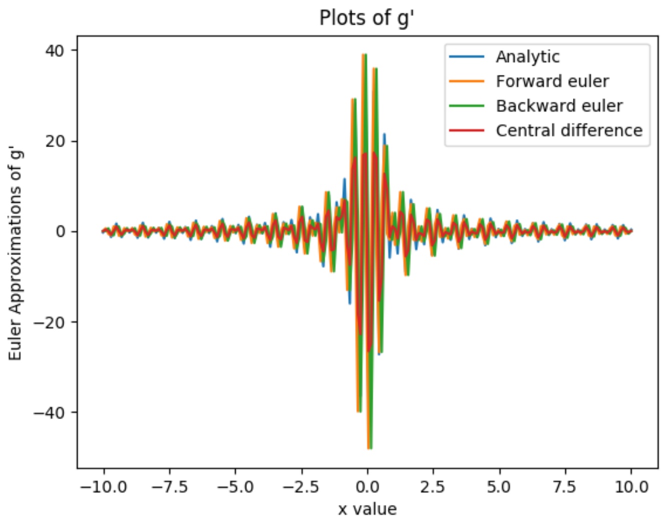
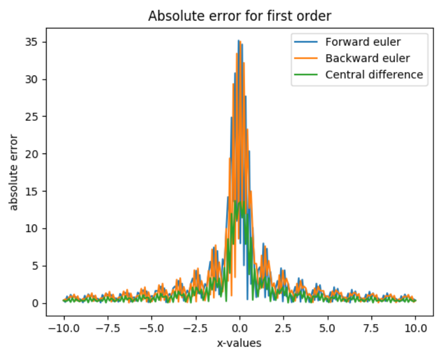


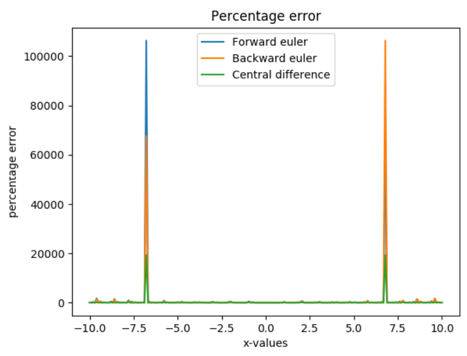
Figure 3: Euler Approximations

The backward euler, and central difference are implicit methods because the unknown variable is on both sides of the equation, and we are essentially solving an algebraic expression to obtain our desired result. However, the forward euler uses data at current position and previous position to obtain the value at the future point - and is thus an explicit method.

(1 d) Plotted below are the average and percent error with respect to x



(a) Absolute Error

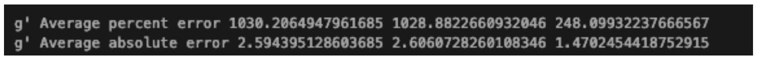


(b) Percent error

Figure 4: Error Plots

It is clear that the central difference method performs much better than both foward and backward euler methods. This is because, the signal ($g(x)$) given rapidly changes and is very sensitive to changes in x. So, the forward and backward euler methods that rely on next and previous data point overestimate and underestimate the derivative value by a huge amount. The central difference, being an average of the two, performs better. Notice however, that due to the erratic behavior of the function close to zero, the averaging out process in central difference fails to capture the peaks properly; this is where most of the errors occur.

The spikes that can be noticed in the percentage error from backward and forward euler methods, seem worrysome - and should be normalized properly, however, at the risk of losing insight. The spikes exist because at those two x values, the analytic derivative gets very close to zero, while the euler approximations aren't as close. The difference between the analytic value and the real value is on the order of 1.50, while the orignial analytic value there is 0.001743, and so upon dividing the two numbers, we have results in the order of 1500, and when multiplied by 100, we obtain the huge number that we observe. I believe these spikes are specific to the sampling of x values. Given a different initial point, we could have more spikes or less depending on how many x values are sampled where $g'(x)$ is very close to zero. We could also reduce the amount of error we have by increasing the resolution of the data (getting more data points)[1]
The average errors (for Forward Euler, backward Euler and central-difference respectively) reported are as follows



```
g' Average percent error 1030.2064947961685 1028.8822660932046 248.09932237666567
g' Average absolute error 2.594395128603685 2.6060728260108346 1.4702454418752915
```

Figure 5: Average errors

[1]we will observe this in part 1 f

(1 e)
The same steps as above was done for computing the second derivative of the function.[2]. Plotted below is the analytic second derivative.

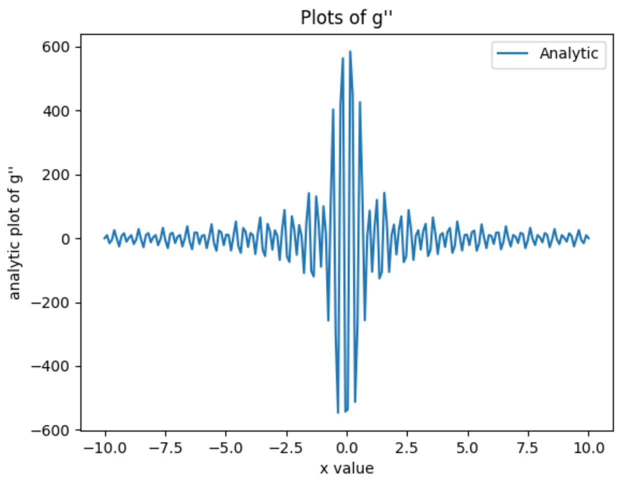


Figure 6: Analytic plot of g”

This looks promising because, it looks like a flipped and scaled up version of $g(x)$, which is what we would expect out of a second derivative of a function that has x^2's, sines and cosines as its main components. The forward, backward and central difference methods were employed for the function $g'(x)$ to compute the derivative. The overlapping plots of the three methods, along with the analytic solution is presented below (Figure 7).

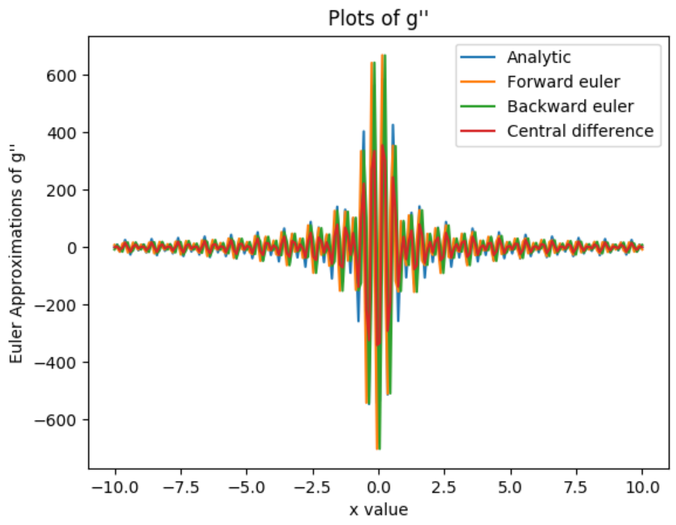
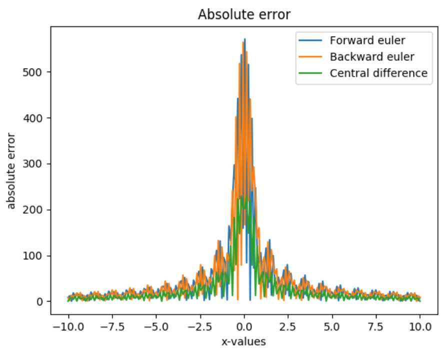


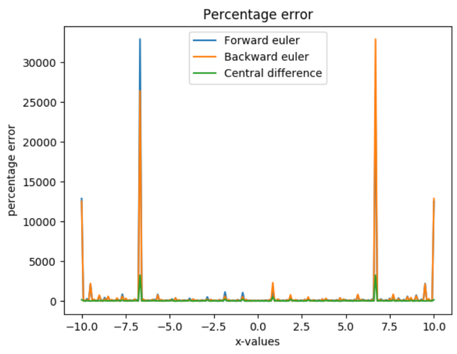
Figure 7: Euler Approximations

[2]for the sake of sanity, I refrain from reproducing the analytic second derivative

Plotted below are the average and percent error with respect to x (Figure 8)



(a) Absolute Error



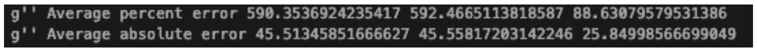
(b) Percent error

Figure 8: Error Plots

It is clear that, once again the central difference method performs much better than both foward and backward euler methods, for the same reasons as before (in fact, $g'(x)$ is even more sensitive than g)

We notice more spikes in the percentage errors here, because like we noticed $g'(x)$ varies very rapidly, and thus will have a lot of points where it plummets to zero. Forward and backward euler methods do not particularly work well in this scenario, and thus resulting in huge spikes with a comparably lesser magnitude than the ones in the case of g'.

The average errors (for Forward Euler, backward Euler and central-difference respectively) reported are as follows



```
g'' Average percent error 590.3536924235417 592.4665113818587 88.63079579531386
g'' Average absolute error 45.51345851666627 45.55817203142246 25.84998566699049
```

Figure 9: Average errors

We notice that the average percent error has significantly gone down from the case of g', but the average absolute error has significantly gone up. This can be because of the additional scale factors that we obtain upon taking the second derivative - and so the absolute errors naturally scale up. However, the percentage errors scale down because - now we have a positive scaling to the function values (and especially to the ones close to zero), and thus division by the new analytic values do not cause huge errors to crop up - like it did in the case of g'. One other thing to notice is that in both cases - areas with most absolute errors do not have a relatively huge amount of percentage errors because the values close to the origin are more often away from zero, and thus no huge scaling occurs in that region, and we get a desirable percentage estimate.

(1 f) The number of grid points was increased to 400, and with this, I was able to obtain an increased degree of smoothing in the function. The function $g(x)$ is plotted below.

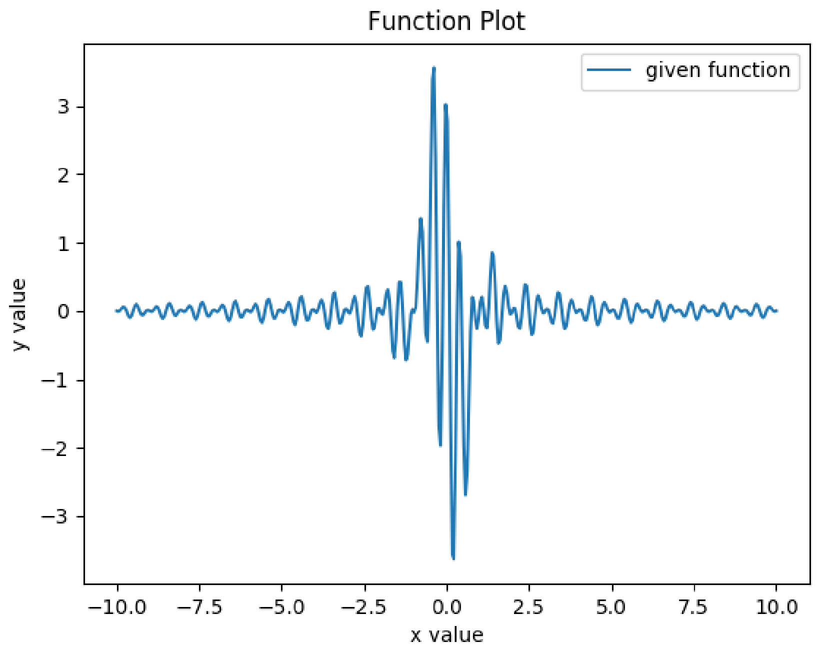


Figure 10: Function Plot

The analytic derivative was calculated, and is also plotted below

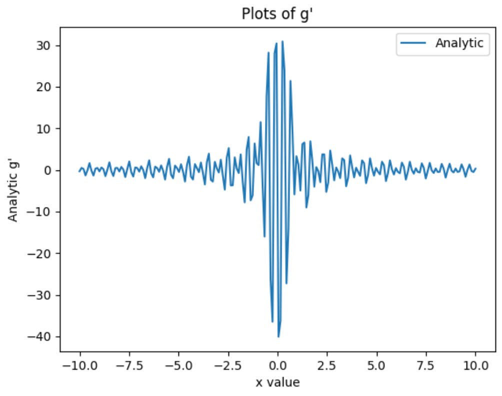


Figure 11: Analytic derivative

The forward, backward and central difference methods were employed for the function $g(x)$ to compute the derivative. The overlapping plots of the three methods, along with the analytic solution is presented below.

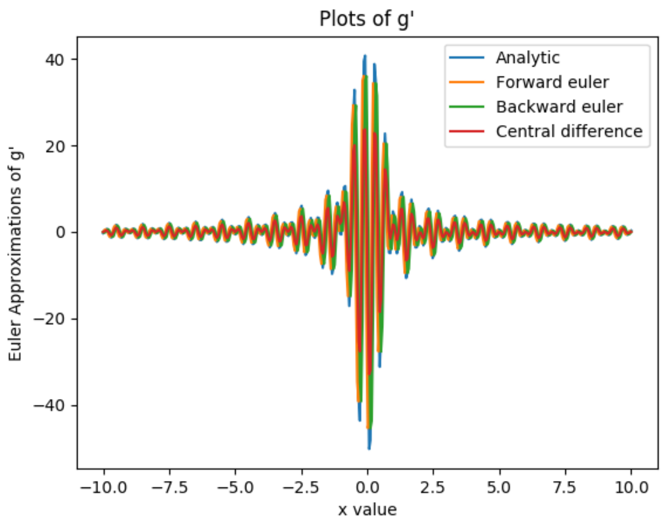
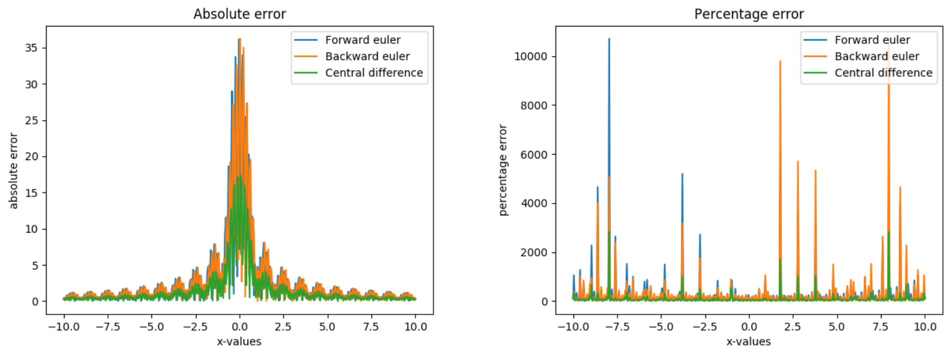


Figure 12: Euler Approximations

Plotted below are the average and percent error with respect to x

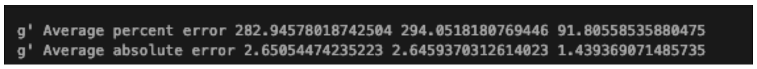


(a) Absolute Error (b) Percent error

Figure 13: Error Plots

Notice that the number of spikes in percent error has gone up significantly. This is because, with a finer grid spacing, we are hitting a lot of x values where $g'(x)$ is very close to zero. However, the most important thing to notice is that the percentage error has significantly gone down. This is simply because as we get closer to points where g' is zero, we have closer and closer approximations, and thus the approximated solution is much closer to the analytic solution, and thus dividing by the analytic solution does not create as big an impact as it did with coarser grid spacing. Notice also that the magnitude of the absolute error has gone down significantly as well. This is justified by the very same reason. Needless to say, the central difference method outperforms the other two methods in this setting as well.

The average errors (for Forward Euler, backward Euler and central-difference respectively) reported are as follows



```
g' Average percent error 282.94578018742504 294.0518180769446 91.80558535880475
g' Average absolute error 2.65054474235223 2.6459370312614023 1.439369071485735
```

Figure 14: Average errors

Notice that the average percentage error has gone down by a significant amout. This is once again apparent from the reasons we discussed above. It is however interesting to see a slight increase in the absolute errors pertaining to forward and backward euler methods. This could be because of various reasons, one in particular being that we sample x-values where a lot more of them have $g'(x)$ very close to zero, with euler methods producing non-zero solutions.

The same steps as above was done for computing the second derivative of the function. The forward, backward and central difference methods were employed for the function $g'(x)$ to compute the derivative. The overlapping plots of the three methods, along with the analytic solution is presented below.

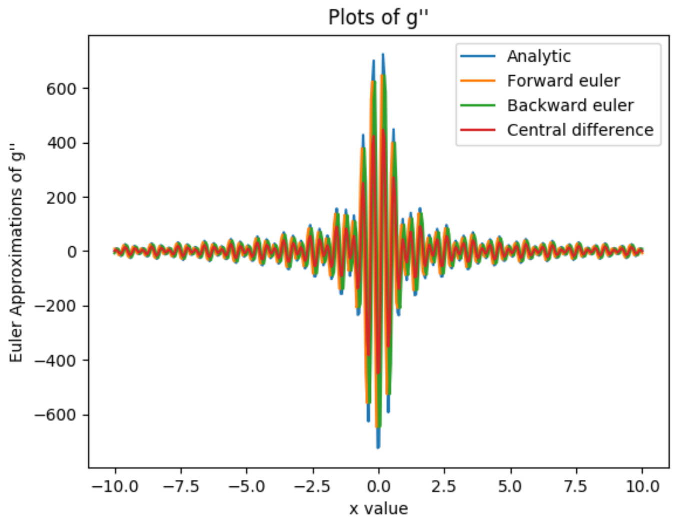
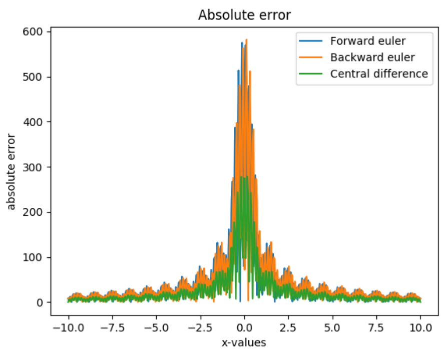
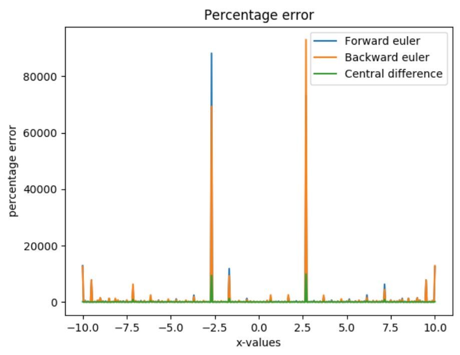


Figure 15: Euler Approximations

Plotted below are the average and percent error with respect to x



(a) Absolute Error



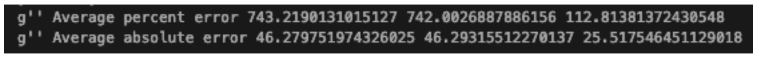
(b) Percent error

Figure 16: Error Plots

It is clear that, once again the central difference method performs much better than both foward and backward euler methods, for the same reasons as before

We notice more spikes in the percentage errors here again, because like we noticed $g'(x)$ varies very rapidly, and thus will have a lot of points where it plummets to zero. In addition, the additional grid spacing are not helping because we sample more points that are very close to zero. Forward and backward euler methods do not particularly work well in this scenario, and thus resulting in huge spikes with a comparably lesser magnitude than the ones in the case of g'. The lesser magnitude can once again be attributed to there being a more refined data in this scenario.

The average errors (for Forward Euler, backward Euler and central-difference respectively) reported are as follows

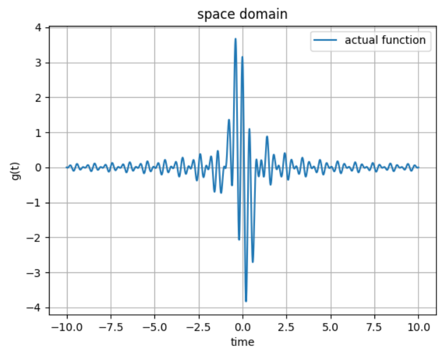


```
g'' Average percent error 743.2190131015127 742.0026887886156 112.81381372430548
g'' Average absolute error 46.279751974326025 46.29315512270137 25.517546451129018
```

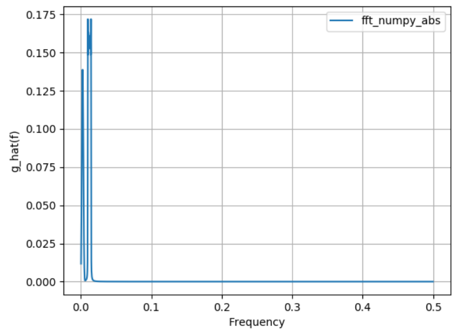
Figure 17: Average errors

We notice that the average percent error has significantly gone up from the case of g'' when $n = 200$, and this is because like we noted - g'' behaves very violently, and with more grid points, we have a lot more data points were the function becomes very close to zero which is undesirable for forward and backward euler. The absolute errors have also gone up because of the additional scale factors that we obtain upon taking the second derivative - and so the absolute errors naturally scale up. The huge spikes that we notice in the percentage error is because of our sampled x value being very close to zero (this time in the order of 10^{-3}), whereas in the previous case, we simply jumped over this data point, and picked the two neighbors of it, thus avoiding this problem (by accident). However, negating the huge spikes (throwing out the anamolies from the data) in both cases, we have the average percent error in $n = 400$ case being lower than in the case of $n = 200$ as expected.

(2 a) In order to compute the fourier transform of g, the number of x values being considered had to be changed. It was changed to $1024 = 2^{10}$. This sampling will allow for an efficient fft. The FFT library provided by numpy was used for this purpose. The negative frequencies are filtered out in the process as they are not of physical significance. The unscaled FT plot (in both time and frequency domain) is reproduced below. Upon



(a) Space Domain

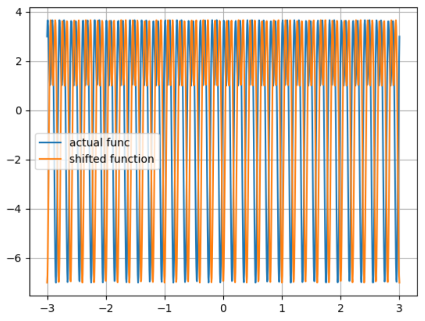


(b) Frequency Domain

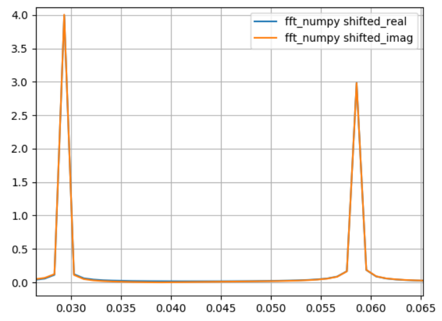
Figure 18: FFT Plots

rescaling we should be able to obtain the correct frequencies from the FFT plot. Since the original frequencies are hard to compute for the given function, it becomes hard to justify the frequencies in the FFT plot.

For a proof of concept, consider the simple function $4\sin(10\pi x) + 3\cos(20\pi x)$ and a shift of it $4\sin(10\pi(x-0.25)) + 3\cos(20\pi(x-0.25))$. The two functions are plotted below along with their FFT. The sampled number of data points were 1024. Notice that in the FFT plot here,



(a) Space Domain



(b) Frequency Domain

Figure 19: Proof of concept

the peaks occur at $\simeq 0.0299$ and $\simeq 0.0598$, and we have $\frac{10\pi}{1024} = 0.03067$ and $\frac{20\pi}{1024} = 0.06135$. The little error we see can be attributed to numerical errors that are transferred while travelling to the frequency domain. This proves that rescaling does return the desired frequencies.

(2 b) The first derivative of the function was calculated using spectral methods, and the output is plotted along with the analytic derivative. The plot is reproduced below.

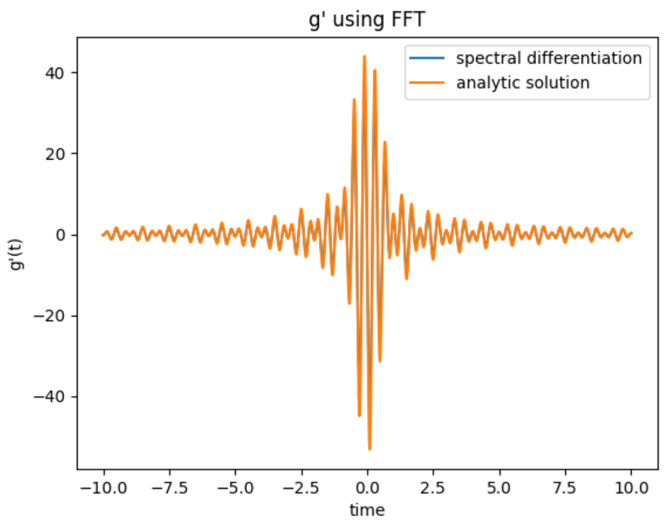


Figure 20: Spectral method and analytic derivative

It is apparent from the plot that the error was very close to zero (that is to say, the spectral method produced the exact analytic derivative) with very small numerical errors.

(2 c) The plot obtained through the spectral method was then plotted alongside the Euler approximations obtained in 1 c. The plot is reproduced below.

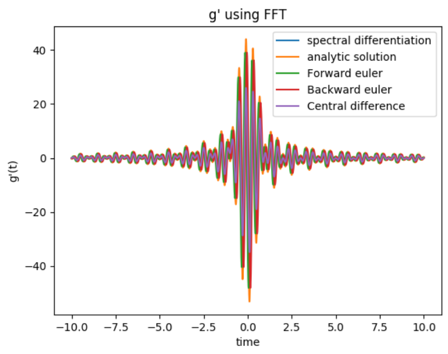


Figure 21: Spectral method vs Euler approximations

Upon computing the error, the maximum absolute error that was reported was $33.30, 32.56$ and 9.11 for Forward Euler, Backward Euler and Central difference respectively. Once again, we see the central difference outperforming the forward and backward euler.

(3 a) Given $f(x) = ae^{bx}$. Upon fitting the function with an exponential fit, we obtain the values of a and b as $a = 3.64404617$ and $b = 0.54550424$. The fitted function along with the data is plotted below.

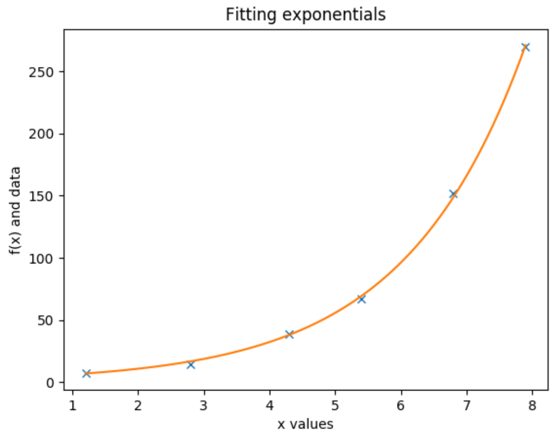


Figure 22: Exponential fitting

(3 b) The data was then interpolated, using a cubic spline interpolation to the given points and the values returned were as follows: $f(1.3) = 6.9971847$, $f(2.9) = 15.3595104$, $f(4.4) = 40.8406263$, $f(5.5) = 70.876742$ and $f(6.9) = 160.54950$

A cubic interpolating spline s is a cubic polynomial that has the property that $s(x_m) = f(x_m)$, with the additional condition that $s''(x_0) = s''(x_m) = 0$. This helps in smoothing of the spline, thereby getting a better estimate than regular polynomial interpolation which might have differentiability issues at the data points. We construct the spline by solving the following system

$s_i = a_i(x - x_i) + b_i(x - x_i)^2 + c_i(x - x_i) + d_i$.
By fixing the $4m$ coefficients (a_i, b_i, c_i, d_i as $i : 0 \to m - 1$), we fix the spline. This is a linear system solved in the following fashion with $4m$ conditions being

$s''(x_0) = s''(x_m) = 0$
$s_i(x_i) = y_i$
$s_{m-1} = y_m$
$s_i(x_{i+1}) = s_{i+1}(x_{i+1})$
$s'_i(x_{i+1}) = s'_{i+1}(x_{i+1})$
$s''_i(x_{i+1}) = s''_{i+1}x_{i+1}$

I chose the cubic spline interpolation, because it is the highest order spline interpolation provided by the interp1d function in numpy, and is very easy to impliment. The given function being non-polynomial, I realized that the highest order polynomial approximation available will minimize the errors. Plotted below is the interpolated spline.

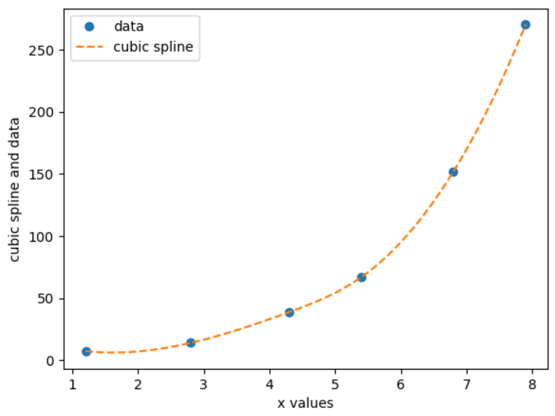


Figure 23: Cubic Spline interpolation

(1 c) The average absolute error between the fitted function and the interpolating function was reported to be 1.8387 and the average percentage error between the two was reported to be 5.1770.

Interpolation is a method of constructing new data points within the range of a set of known data points, whereas fitting data refers to the process of finding a curve that best represents the trend of the data (or in other words, best ”fits” the data). In other words, given some data - interpolation is the process of constructing a curve through the data points with the implicit assumption that the data points are accurate. Curve fitting, on the other hand applied to data that contains noise, will return a smooth curve that approximates the data and its trend.

APPENDIX 1

4(a)

```python
import numpy as np
import matplotlib.pyplot as plt
import timeit

# ------------------------- Function definitons --------------------------------


def forwardeuler(f, x, h):
    return (f(x+h) - f(x))/h


def backwardeuler(f, x, h):
    return (f(x) - f(x-h))/h


def centralDifference(f, x, h):
    return (f(x+h) - f(x-h))/(2*h)


def g(x):
    return (np.cos(5*np.pi*x) - np.e**(-(x**2)/2)*np.sin(0.5*np.pi*x))*(np.sin(np.pi*x)/x)


def gpr(x):
    m = np.cos(5*np.pi*x) - np.e**(-(x**2)/2)*np.sin(0.5*np.pi*x)
    n = np.sin(np.pi*x)/x
    mpr = -5*np.pi*np.sin(5*np.pi*x) - (np.e**(-(x**2)/2)*(-x)*np.sin(0.5*
                                                                        np.pi*x) + 0.5*np.pi*np.cos(0.5*np.pi*x)*np.e**(-(x**2)/2))
    npr = -(np.sin(np.pi*x)/x**2) + (1/x)*(np.pi*np.cos(np.pi*x))
    return (m*npr + n*mpr)


def gdpr(x):
    return -(np.pi**2*np.sin(np.pi*x)*(np.cos(5*np.pi*x) - np.e**(-x**2/2)*np.sin(1.5708*x)))/x + 2*np.pi*np.cos(np.pi*x)*((np.e**(-x**2/2)*x*np.sin(1.5708*x) - 1.5708*np.e**(-x**2/2)*np.cos(1.5708*x) - 5*np.pi*np.sin(5*np.pi*x))/x - (np.cos(5*np.pi*x) - np.e**(-x**2/2)*np.sin(1.5708*x))/x**2) + np.sin(np.pi*x)*((2.4674*np.e**(-x**2/2)*np.sin(1.5708*x) + (np.e**(-x**2/2) - np.e**(-x**2/2)*x**2)*np.sin(1.5708*x) + 3.14159*np.e**(-x**2/2)*x*np.cos(1.5708*x) - 25*np.pi**2*np.cos(5*np.pi*x))/x - (2*(np.e**(-x**2/2)*x * np.sin(1.5708*x) - 1.5708*np.e**(-x**2/2)*np.cos(1.5708*x) - 5*np.pi*np.sin(5*np.pi*x)))/x**2 + (2*(np.cos(5*np.pi*x) - np.e**(-x**2/2)*np.sin(1.5708*x)))/x**3)
# ------------------------------------------------------------------------------
```

```python


h = 0.1
x = np.linspace(-10, 10, num=200)
y = g(x)
ypr = gpr(x)
ydpr = gdpr(x)

# --------------------------------------------------------------------------------
yprfor = []  # forward euler for g'
yprbac = []  # backward euler for g'
yprcdf = []  # central difference for g'

for i in range(len(x)):
    yprfor.append(forwardeuler(g, x[i], h))
    yprbac.append(backwardeuler(g, x[i], h))
    yprcdf.append(centralDifference(g, x[i], h))
# ---------------------------------- 1A ------------------------------------------
plt.title("Function Plot")
plt.plot(x, y, label="given function")
plt.xlabel("x value")
plt.ylabel("y value")
plt.legend()
plt.show()
# -------------------------------------------------------------------------------


# ---------------------------------- 1B -----------------------------------------

plt.title("Plots of g'")
plt.plot(x, ypr, label="Analytic")  # Plotting analytic solution
# -------------------------------------------------------------------------------


# ---------------------------------- 1C -----------------------------------------
plt.plot(x, yprfor, label="Forward euler")
plt.plot(x, yprbac, label="Backward euler")
plt.plot(x, yprcdf, label="Central difference")
plt.xlabel("x value")
plt.ylabel("Euler Approximations of g'")
plt.legend()
plt.show()
# -------------------------------------------------------------------------------


# ---------------------------------- 1D
```

```
    ----------------------------------------

pererrorfor = []  # for percent error in forward euler
pererrorbac = []  # for percent error in backward eulder
pererrorcdf = []  # for percent error in central difference
abserrorfor = []  # for absolute error in forward euler
abserrorbac = []  # for absolute error in backward euler
abserrorcdf = []  # for absolute error in central difference

# computing absolute error at every data point. = |analytic - approx|
for i in range(len(x)):
    abserrorfor.append(np.abs((ypr[i] - yprfor[i])))
    abserrorbac.append(np.abs((ypr[i] - yprbac[i])))
    abserrorcdf.append(np.abs((ypr[i] - yprcdf[i])))

# computing % error at every data point. = 100*(|analytic - approx|/
    analytic)
for i in range(len(x)):
    pererrorfor.append(100*np.abs((ypr[i] - yprfor[i])/(ypr[i])))
    pererrorbac.append(100*np.abs((ypr[i] - yprbac[i])/(ypr[i])))
    pererrorcdf.append(100*np.abs((ypr[i] - yprcdf[i])/(ypr[i])))

# plotting absolute and percent error
plt.title("Absolute error")
plt.plot(x, abserrorfor, label="Forward euler")
plt.plot(x, abserrorbac, label="Backward euler")
plt.plot(x, abserrorcdf, label="Central difference")
plt.xlabel("x-values")
plt.ylabel("absolute error")
plt.legend()
plt.show()

plt.title("Percentage error")
plt.plot(x, pererrorfor, label="Forward euler")
plt.plot(x, pererrorbac, label="Backward euler")
plt.plot(x, pererrorcdf, label="Central difference")
plt.xlabel("x-values")
plt.ylabel("percentage error")
plt.legend()
plt.show()

# computing the mean error in each of the arrays
forpererror = np.mean(pererrorfor)
bacpererror = np.mean(pererrorbac)
cdfpererror = np.mean(pererrorcdf)
forabserror = np.mean(abserrorfor)
bacabserror = np.mean(abserrorbac)
cdfabserror = np.mean(abserrorcdf)

print("g' Average percent error", forpererror, bacpererror, cdfpererror)
print("g' Average absolute error", forabserror, bacabserror, cdfabserror)
#
    ------------------------------------------------------------------------------
```

```python

# ------------------------------ 1E
   ------------------------------------------
ydprfor = []  # forward euler for g''
ydprbac = []  # backward euler for g''
ydprcdf = []  # central difference for g''

plt.title("Plots of g''")
plt.plot(x, ydpr, label="Analytic")  # plotting the analytic g''

# plotting forward, backward and central difference computations for g''
for i in range(len(x)):
    ydprfor.append(forwardeuler(gpr, x[i], h))
    ydprbac.append(backwardeuler(gpr, x[i], h))
    ydprcdf.append(centralDifference(gpr, x[i], h))


plt.plot(x, ydprfor, label="Forward euler")
plt.plot(x, ydprbac, label="Backward euler")
plt.plot(x, ydprcdf, label="Central difference")
plt.xlabel("x value")
plt.ylabel("Euler Approximations of g''")
plt.legend()
plt.show()

pererrordfor = []  # for percent error in forward euler
pererrordbac = []  # for percent error in backward euler
pererrordcdf = []  # for percent error in centralDifference
abserrordfor = []  # for absolute error in forward euler
abserrordbac = []  # for absolute error in backward euler
abserrordcdf = []  # for absolute error in central difference

# computing absolute error for each data point
for i in range(len(x)):
    abserrordfor.append(np.abs((ydpr[i] - ydprfor[i])))
    abserrordbac.append(np.abs((ydpr[i] - ydprbac[i])))
    abserrordcdf.append(np.abs((ydpr[i] - ydprcdf[i])))

# computing percent error for each data point
for i in range(len(x)):
    pererrordfor.append(100*np.abs((ydpr[i] - ydprfor[i])/(ydpr[i])))
    pererrordbac.append(100*np.abs((ydpr[i] - ydprbac[i])/(ydpr[i])))
    pererrordcdf.append(100*np.abs((ydpr[i] - ydprcdf[i])/(ydpr[i])))

# plotting the errors
plt.title("Absolute error")
plt.plot(x, abserrordfor, label="Forward euler")
plt.plot(x, abserrordbac, label="Backward euler")
plt.plot(x, abserrordcdf, label="Central difference")
plt.xlabel("x-values")
plt.ylabel("absolute error")
plt.legend()
plt.show()

plt.title("Percentage error")
plt.plot(x, pererrordfor, label="Forward euler")
plt.plot(x, pererrordbac, label="Backward euler")
plt.plot(x, pererrordcdf, label="Central difference")
plt.xlabel("x-values")
plt.ylabel("percentage error")
plt.legend()
plt.show()

# computing average percent error and absolute error
dforpererror = np.mean(pererrordfor)
dbacpererror = np.mean(pererrordbac)
dcdfpererror = np.mean(pererrordcdf)
dforabserror = np.mean(abserrordfor)
dbacabserror = np.mean(abserrordbac)
dcdfabserror = np.mean(abserrordcdf)
#
    ------------------------------------------------------------------------------

# ---------------------------- Print statements
    --------------------------------
# printing all average errors to console.
print("g'' Average percent error", dforpererror, dbacpererror,
    dcdfpererror)
print("g'' Average absolute error", dforabserror, dbacabserror,
    dcdfabserror)
#
    ------------------------------------------------------------------------------



# ---------------------------- Fourier stuff
    -----------------------------------
# -------------------------------- 2A
    -------------------------------------------
N = 4096  # 2^11
x = np.linspace(-10, 10, num=N)  # redefining axis for fourier
    transformation
f1 = np.array(g(x))  # defining the function in time domain
freqs = np.fft.fftfreq(N)  # defining the frequency axis
mask = freqs > 0  # creating a mask to omit negative frequencies
yf1 = np.array(np.fft.fft(f1))  # performing fft using numpy
reyf1 = 2*(yf1.real/N)  # getting real part of the output after rescaling
imyf1 = 2*(yf1.imag/N)  # getting imaginary part of the output after
    rescaling
fft_fin1 = 2*np.abs(yf1/N)  # getting the rescaled absolute value


# Plotting

plt.title("space domain")
plt.plot(x, f1, label="actual function")
plt.xlabel("time")
```

```python
plt.ylabel("g(t)")
plt.legend()
plt.grid()
plt.show()


plt.title("Frequency Domain")
#plt.plot(freqs[mask], reyf1[mask], label="fft_numpy_real")
#plt.plot(freqs[mask], imyf1[mask], label="fft_numpy_imag")
plt.plot(freqs[mask], fft_fin1[mask], label="fft_numpy_abs")
plt.legend()
plt.xlabel("Frequency")
plt.ylabel("g_hat(f)")
plt.grid()
plt.show()
# ------------------------------------------------------------------------------

# ----------------------------------- 2B,C ---------------------------------------
N = 1024  # 2^10
xmin = -10
xmax = 10
step = (xmax-xmin)/(N)
xdata = np.linspace(xmin, xmax, N)  # redefining x in order to do FFT properly
v = g(xdata)
vhat = np.fft.fft(v)  # fft of v
what = 1j*np.zeros(N)  # set up for taking derivative
what[0:int(N/2)] = 1j*np.arange(0, N/2, 1)
what[int(N/2)+1:] = 1j*np.arange(-N/2 + 1, 0, 1)
what = what*vhat
w = np.real(np.fft.ifft(what/4))  # converting back to time domain**
# -------------------------------------------------------------------------------

plt.title("g' using FFT")
plt.plot(xdata, w, label="spectral differentiation")  # FFT output
plt.plot(x, ypr, label="analytic solution")
# Comment the next three lines to compare analytic and spectral solutions
plt.plot(x, yprfor, label="Forward euler")
plt.plot(x, yprbac, label="Backward euler")
plt.plot(x, yprcdf, label="Central difference")
plt.xlabel("time")
plt.ylabel("g'(t)")
plt.legend()
plt.show()  # change num to 4096 in line 39 to make this plot to work.
# ------------------------------------------------------------------------------

# ** We divide by 4 because xdata contains 4096 entries, whereas we are only really calculating 1024.
#   So we need to scale our output down by a factor of 4 in order to
```

```
    normalize

# comment the fft plot above and change num in line 39 to 1024 to make
    this work.

# we do this because, when we are not plotting, we only want to focus on
    the
# positive real frequencies, so we need to turn the mask back on to
    compare
fftforerror = []
fftbacerror = []
fftcdferror = []

print(len(w), len(ypr))

for i in range(len(w)):
    fftforerror.append(np.abs(w[i] - yprfor[i]))
    fftbacerror.append(np.abs(w[i] - yprbac[i]))
    fftcdferror.append(np.abs(w[i] - yprcdf[i]))

print("Spectral vs. Euler errors", np.amax(
    fftforerror), np.amax(fftbacerror), np.amax(fftcdferror))
```

4(b)

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit
from scipy import interpolate

xfine = np.linspace(1.2, 7.9, num=500)
x = np.array([1.2, 2.8, 4.3, 5.4, 6.8, 7.9])
f = np.array([7.5, 14.1, 38.9, 67.0, 151.8, 270.1])


def curve(a, b, x):
    return a*np.exp(b*x)


# ------------------------------- 1A --------------------------------------------

[res, data] = curve_fit(lambda t, a, b: a*np.exp(b*t), x, f, p0=(4, 0.01))
print(res)
y = curve(res[0], res[1], xfine)
plt.title("Fitting exponentials")
plt.plot(x, f, 'x', label="data")
plt.plot(xfine, y, label="fitted function")
plt.xlabel("x values")
plt.ylabel("f(x) and data")
plt.show()
# --------------------------------------------------------------------------------

# -----------------------------Interpolation-----------------------------------

# directly plotting the cubic spline function using interp1d
fcubic = interpolate.interp1d(x, f, kind="cubic")

# -------------------------------------------------------------------------------

print("f(1.3)", fcubic(1.3), "f(2.9)", fcubic(2.9), "f(4.4)",
      fcubic(4.4), "f(5.5)", fcubic(5.5), "f(6.9)", fcubic(6.9))
# -------------------------------Plotting the Data-----------------------------
plt.plot(x, f, 'o', label="data")
plt.plot(xfine, fcubic(xfine), '--', label="cubic spline")
plt.xlabel("x values")
plt.ylabel("cubic spline and data")
plt.legend()
plt.show()

# ----------------------------------- 3 C ---------------------------------------
```

```python
datapts = [1.3, 2.9, 4.4, 5.5, 6.9]
abserror = []
pererror = []

for i in range(5):
    abserror.append(np.abs(fcubic(datapts[i]) - curve(res[0], res[1],
    datapts[i])))
    pererror.append(
        100*np.abs(fcubic(datapts[i]) - curve(res[0], res[1], datapts[i]))
    /curve(res[0], res[1], datapts[i]))

print("avg abs error:", np.mean(abserror))
print("avg % error", np.mean(pererror))
```