In [153]:
```python
import numpy as np
import matplotlib.pyplot as plt
import scipy.optimize as so
import time as tm
```

This document was made by Scott Johnstun, Aiden Harbick, and Mark Anderson. The corresponding repository is located at [https://git.physics.byu.edu/computational-physics/NumericalDerivatives (https://git.physics.byu.edu/computational-physics/NumericalDerivatives)](https://git.physics.byu.edu/computational-physics/NumericalDerivatives).

Numerical Derivatives Homework

Problem 1: Finite Differences

Calculate the numerical first derivative of the function $y(x) = \sin x$ using the centered two-point formula:

$$\frac{dy}{dx} \approx \frac{y(x+h) - y(x-h)}{2h} \tag{1}$$

and plot the error

$$\text{Error} = y'(x) - \cos x \tag{2}$$

in the range $(-\pi, \pi)$ at 100 points. Optimize the step size, h. How accurate can you get by adjusting h?

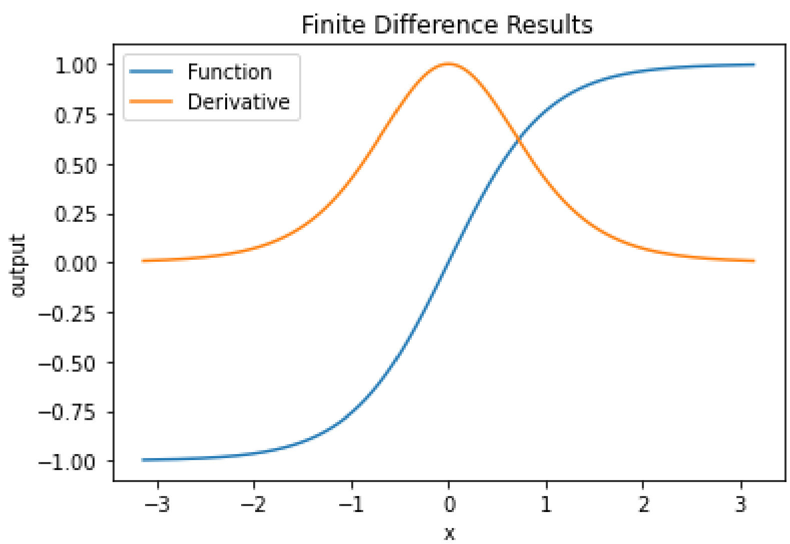
First, let's define a function to do a finite difference

In [128]:
```python
def finiteDiff(func,x,h):
    return (func(x + h) - func(x - h)) / (2*h)
```

Now let's test the function:

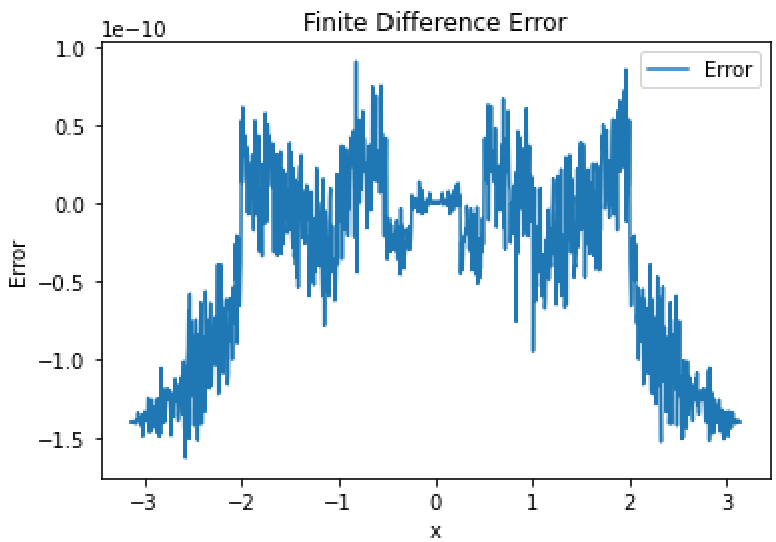
In [129]:

```
# Function to differentiate. Feel free to play with the function
func = np.tanh

# Region over which to differentiate
x = np.linspace(-np.pi,np.pi,1000)

# Step size
h = 1e-6

# Computing the derivatives
dydx = finiteDiff(func,x,h)

# Plotting results
plt.plot(x,func(x),label = "Function")
plt.plot(x,dydx,label = "Derivative")
plt.title("Finite Difference Results")
plt.xlabel("x")
plt.ylabel("output")
plt.legend();
```



Now let's use the function to test the `f(x) = sin(x)` function:

In [130]:
```
# Function to differentiate. Feel free to play with the function
func = np.sin

# Region over which to differentiate
x = np.linspace(-np.pi,np.pi,1000)

# Step size
h = 1e-6

# Computing the derivatives
dydx = finiteDiff(func,x,h)

# Plotting the error
plt.plot(x,dydx - np.cos(x),label = "Error")
plt.title("Finite Difference Error")
plt.xlabel("x")
plt.ylabel("Error")
plt.legend();


```



Optimizing the step size

Let's optimize the step size now. Since it sounds like a good way to practice, let's try using some actual optimization techniques via `scipy.optimize` .

First, let's define a function in terms of step size only. This function takes as input a step size value and returns the rms error.

In [131]:
```
# Gets the resultant rms error for step size h
def run_h(h):
    func = np.sin
    x = np.linspace(-np.pi,np.pi,1000)
    dydx = finiteDiff(func,x,h)

    difference = dydx - np.cos(x) # We will lose lots of precision in t

    dfiniteDiffdh = np.imag(finiteDiff(func,x,1j*1e-30))

    return np.mean(np.sqrt(difference**2))

# Provides the rms error and the derivative of the error with respect t
def test_h(h):

    complexStep = 1e-30

    value = run_h(h)
    dh = np.imag(run_h(h + 1j*complexStep))/complexStep

    return value, dh

```

Now we can use `scipy.optimize` to solve the optimization problem:

In [132]:
```
options = {'disp':True}

so.minimize(test_h,1,options = options, jac = True)
```

```
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 8
         Function evaluations: 18
         Gradient evaluations: 18
```

Out[132]:
```
      fun: 3.933341395515391e-11
 hess_inv: array([[4.7065467]])
      jac: array([-4.20478701e-06])
  message: 'Optimization terminated successfully.'
     nfev: 18
      nit: 8
     njev: 18
   status: 0
  success: True
        x: array([-1.95299911e-05])
```

And there you have it! It looks like a step size on the order of 10^{-5} is the optimal step size. Depending on the starting point, the ideal step size will be negative, but that is irrelevant because we are stepping both directions for the centered-difference.

We can also find the optimal step size via equation 5.7.5 in the textbook:

$$h \approx \sqrt{\frac{\epsilon_f f}{f''}} \tag{3}$$

where ϵ_f is the fractional accuracy and can be approximated as $\epsilon_f \approx \epsilon_m$ for simple functions (and we assue that $\sin x$ counts as a simple function here).

Using the values

- $\epsilon_f = 10^{-16}$
- $f(x) = \sin x$
- $f''(x) = -\sin x$

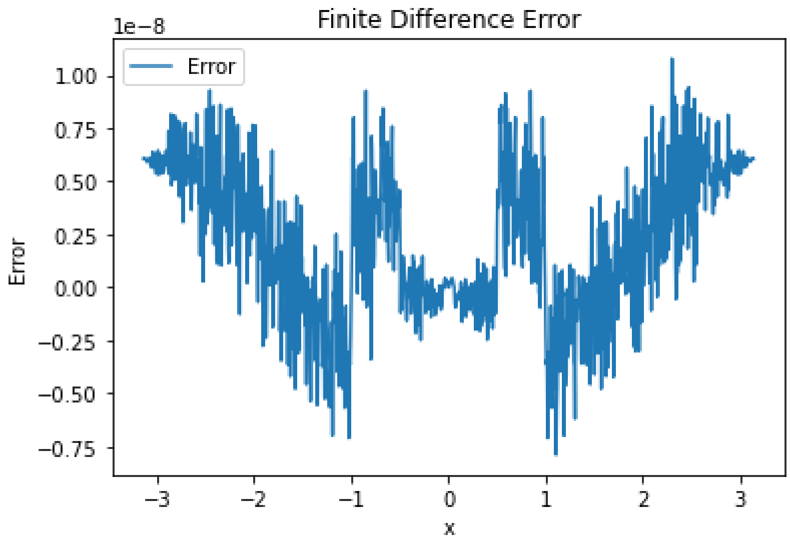
Evidently, we will end up with a negative, so let's just take the magnitude of the derivatives, which results in

$$h \approx \sqrt{\epsilon_f} \tag{4}$$

$$h \approx 10^{-8} \tag{5}$$

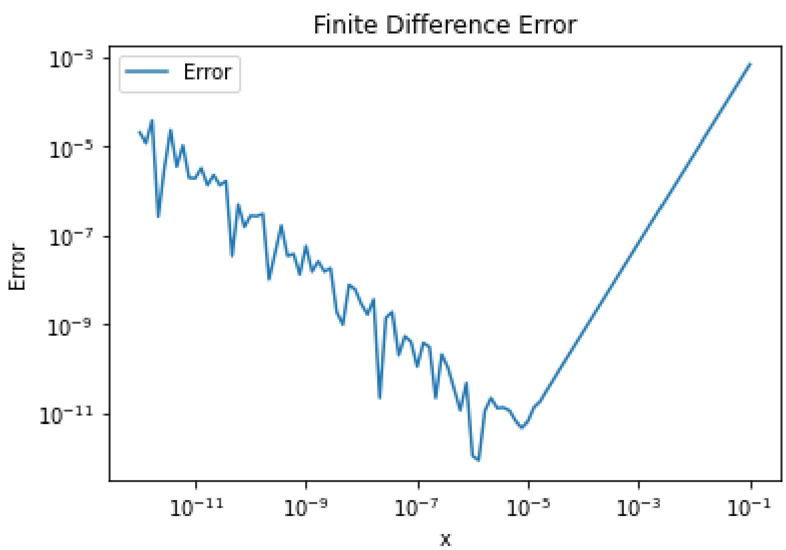
assuming that the machine accuracy truly is 10^{-16}. Let's test this out now:

In [133]:
```
func = np.sin

# Region over which to differentiate
x = np.linspace(-np.pi,np.pi,1000)

# Step size
h = 1e-8

# Computing the derivatives
dydx = finiteDiff(func,x,h)

# Plotting the error
plt.plot(x,dydx - np.cos(x),label = "Error")
plt.title("Finite Difference Error")
plt.xlabel("x")
plt.ylabel("Error")
plt.legend();
```



Now let's try plotting the error as a function of h:

In [134]:
```
# Function to differentiate
func = np.sin
trueDerivative = np.cos

# Point to evaluate at
x = 2

# Step sizes
h = np.logspace(-1,-12,num=100)

# Computing the derivatives
dydx = finiteDiff(func,x,h)

# Plotting the error
plt.loglog(h,np.abs(dydx - trueDerivative(x)),label = "Error")
plt.title("Finite Difference Error")
plt.xlabel("x")
plt.ylabel("Error")
plt.legend();
```



We see here that the optimal value is indeed around 10^{-5} on my (Mark Anderson's) computer.

Problem 2: More Sophisticated Derivatives

Part A: Implementing the Different Algorithms

Richardson Extrapolation of the finite difference formulas

In [135]:
```
def richardsonDiff(func,x, tol, maxiters = 10):
    h1 = x[1]-x[0]
    x1diff = finiteDiff(func, x, h1)

    count = 1
    change = 1

    while change > tol:
        h2 = h1/(2**count)
        x2diff = finiteDiff(func, x, h2)
        change = np.linalg.norm(x2diff-x1diff)
        if count > maxiters:
            break
        x1diff = x2diff
        count += 1

    xdiff_extrap = x2diff + (x2diff-x1diff)/3

    return xdiff_extrap

```

Automatic differentiation using dual numbers

For this method it might be good to look into the `AlgoPy` package.

In [136]:

```
# Scott codes here
class DN:
    def __init__(self, v, d):
        # value
        self.v = v
        # derivative
        self.d = d

    # Basic operations between two dual numbers
    def __add__(self, other):
        if type(other) == float or type(other) == int:
            other = DN(other, 1.)
        return DN(self.v+other.v, self.d+other.d)

    def __sub__(self, other):
        if type(other) == float or type(other) == int:
            other = DN(other, 1.)
        return DN(self.v-other.v, self.d-other.d)

    def __mul__(self, other):
        if type(other) == float or type(other) == int:
            other = DN(other, 1.)
        return DN(self.v*other.v, self.d*other.v+other.d*self.v)

    def __truediv__(self, other):
        if type(other) == float or type(other) == int:
            other = DN(other, 1.)
        return DN(self.v/other.v, (other.v*self.d-self.v*other.d)/(othe

    # these next four methods allow the computation of x & d,
    #  where x is a real number, & is the operation, and d is a dual nu
    def __radd__(self, other):
        return DN(other, 0.) + self

    def __rsub__(self, other):
        return DN(other, 0.) - self

    def __rmul__(self, other):
        return DN(other, 0.)*self

    def __rtruediv__(self, other):
        return DN(other, 0.)/self

    # computes d^p where d is a dual number and p is a real number
    def __pow__(self, p):
        return DN(self.v**p, p*self.v**(p-1)*self.d)

    # gives string form so you can print a dual number for debugging
    def __str__(self):
        return "{0} + {1}ε".format(self.v, self.d)

    # computes the sine of a dual number
    def sin(self):
        return DN(np.sin(self.v), np.cos(self.v)*self.d)

    # computes the cosine of a dual number
```

```
    def cos(self):
        return DN(np.cos(self.v), -np.sin(self.v)*self.d)

    # computes the natural log of a dual number
    def log(self):
        return DN(np.log(self.v), self.d/self.v)

    # computes the sqrt of a dual number
    def sqrt(self):
        return DN(np.sqrt(self.v), 1/2/np.sqrt(self.v)*self.d)

def autoDiff(func,x):
    dualx = np.array([DN(xi, 1) for xi in x])
    fvals = np.array(func(dualx))
    return [f.d for f in fvals]
```

Chebyshev methods

The Chebyshev polynomials are defined as:

$$T_0(x) = 1 \tag{6}$$

$$T_1(x) = x \tag{7}$$

$$T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x) \tag{8}$$

or, more compactly, as

$$T_n(x) = \cos\left(n \cos^{-1}(x)\right) \tag{9}$$

We can approximate an arbitrary function $f(x)$ as

$$f(x) \approx \left[\sum_{k=0}^{N-1} c_k T_k(x)\right] - \frac{1}{2}c_0 \tag{10}$$

where

$$c_j = \frac{2}{N}\sum_{k=0}^{N-1} f(x_k)T_j(x_k) = \frac{2}{N}\sum_{k=0}^{N-1} f\left[\cos\left(\frac{\pi\left(k+\frac{1}{2}\right)}{N}\right)\right]\cos\left(\frac{\pi j\left(k+\frac{1}{2}\right)}{N}\right) \tag{11}$$

If you want to get the derivatives of a Chebyshev-Approximated function, then you can get the coefficients that approximate the derivative as

$$c'_{i-1} = c'_{i+1} + 2ic \quad (i = m-1, m-2, \ldots, 1) \tag{12}$$

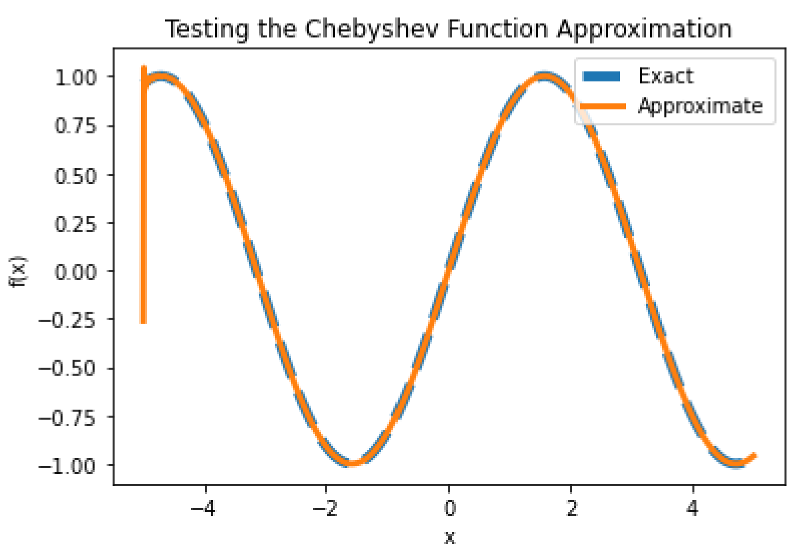
In [137]:

```
def getCoefficient(func,x,j,N):

    a = np.min(x)
    b = np.max(x)

    bma = 0.5*(b-a)
    bpa = 0.5*(b+a)

    c_j = 0

    for k in range(0,N-1):
        cosArg = np.pi*(k + 0.5)/N
        c_j = c_j + func(np.cos(cosArg)*bma + bpa)*np.cos(cosArg*j)

    return 2/N * c_j

def getChebyshevPoly(x,n):

    a = np.min(x)
    b = np.max(x)

    y = (x - 0.5*(b+a))/(0.5*(b-a))

    return np.cos(n * np.arccos(y))

def getChebyshevZeros(n):
    x_k = np.zeros(n)
    for k in range(0,n-1):
        x_k[k] = np.cos(np.pi * (k + 0.5) / n)

    return x_k

def getChebyshevApprox(func,x,N):

    f = np.zeros(len(x))

    for k in range(0,N-1):
        c = getCoefficient(func,x,k,N)
        T = getChebyshevPoly(x,k)
        f = f + c * T

    f = f - 0.5*getCoefficient(func,x,0,N)

    return f

# Section 5.9
def chebyshevDiff(func,x,N):
    a = np.min(x)
    b = np.max(x)
    m = N
    cder = np.zeros(m)
    c = getCoefficient(func,x,m-1,N)

    cder[m-1] = 0
    cder[m-2] = 2*(m-1)*c

```

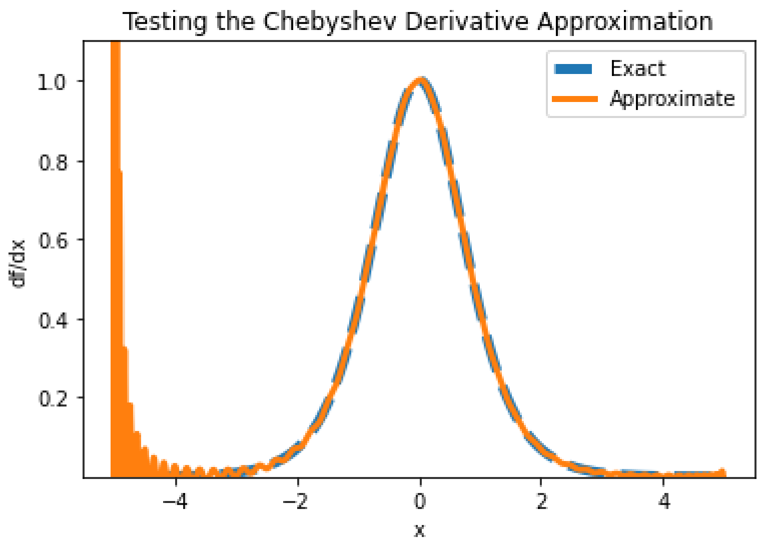
```python
    j = m-2
    while j > 0:
        cder[j-1] = cder[j+1] + 2*j * getCoefficient(func,x,j,N)
        j = j - 1

    cder = cder * 2/(b-a)

    deriv = np.zeros(len(x))
    for k in range(0,N-1):
        c = cder[k]
        T = getChebyshevPoly(x,k)
        deriv = deriv + c * T

    deriv = deriv - 0.5*cder[0]

    return deriv
```

In [138]:

```python
func = np.sin
x = np.linspace(-5,5,1000)
approx = getChebyshevApprox(func,x,100)
deriv = chebyshevDiff(func,x,100)

plt.plot(x,func(x),label = "Exact",linewidth = 5,linestyle = '--')
plt.plot(x,approx,label = "Approximate",linewidth = 3)
plt.legend()
plt.title("Testing the Chebyshev Function Approximation")
plt.xlabel("x")
plt.ylabel("f(x)");
```



In [139]:
```
func = np.tanh
x = np.linspace(-5,5,1000)
approx = getChebyshevApprox(func,x,100)
deriv = chebyshevDiff(func,x,100)
finiteDeriv = finiteDiff(func,x,1e-6)

plt.plot(x,finiteDeriv,label = "Exact",linewidth = 5,linestyle = '--')
plt.plot(x,deriv,label = "Approximate",linewidth = 3)
plt.legend()
plt.ylim(1.1*np.min(finiteDeriv),1.1*np.max(finiteDeriv))
plt.title("Testing the Chebyshev Derivative Approximation")
plt.xlabel("x")
plt.ylabel("df/dx");
```



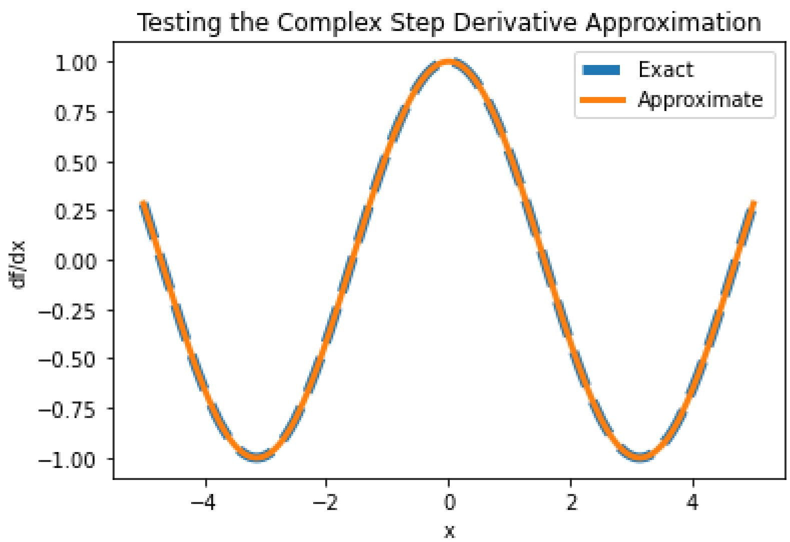
Complex Step

Just for fun, let's add in the complex step method.

In [140]:
```
def complexStep(func,x,h):
    return np.imag(func(x + 1j*h))/h
```

In [141]:

```
func = np.sin
x = np.linspace(-5,5,1000)
deriv = complexStep(func,x,1e-30)
finiteDeriv = finiteDiff(func,x,1e-6)

plt.plot(x,finiteDeriv,label = "Exact",linewidth = 5,linestyle = '--')
plt.plot(x,deriv,label = "Approximate",linewidth = 3)
plt.legend()
plt.ylim(1.1*np.min(finiteDeriv),1.1*np.max(finiteDeriv))
plt.title("Testing the Complex Step Derivative Approximation")
plt.xlabel("x")
plt.ylabel("df/dx");
```



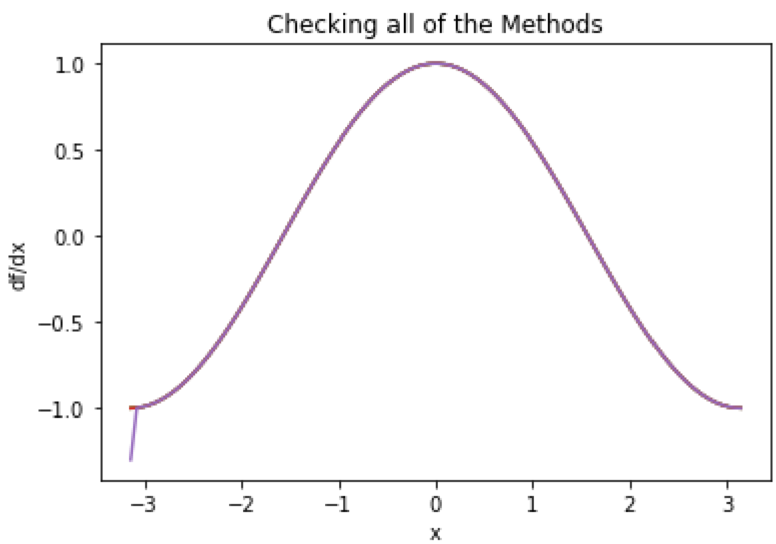
Part B: Write a Driver Function for a Common Interface

In [142]:

```python
def getDerivative(func,x, method = 'Finite Difference'):

    if method == 'Finite Difference':
        startTime = tm.time()
        deriv = finiteDiff(func,x,1e-6)
        runTime = tm.time() - startTime

    elif method == 'Complex Step':
        startTime = tm.time()
        deriv = complexStep(func,x,1e-30)
        runTime = tm.time() - startTime

    elif method == 'Richardson Extrapolation':
        startTime = tm.time()
        deriv = richardsonDiff(func,x, 1e-8)
        runTime = tm.time() - startTime

    elif method == 'Automatic Differentiation':
        startTime = tm.time()
        deriv = autoDiff(func,x)
        runTime = tm.time() - startTime

    elif method == 'Chebyshev Method':
        startTime = tm.time()
        deriv = chebyshevDiff(func,x,1000)
        runTime = tm.time() - startTime

    else:
        deriv = np.nan

    error = deriv - complexStep(func,x,1e-30)

    return deriv, error, runTime
```

In [143]:

```
x = np.linspace(-np.pi,np.pi,100)
func = np.sin

finite_difference = getDerivative(func,x,method = 'Finite Difference')
complex_step = getDerivative(func,x,method = 'Complex Step')
richardson_extrapolation = getDerivative(func,x,method = 'Richardson Ex
automatic_differentiation = getDerivative(func,x,method = 'Automatic Di
chebyshev_method = getDerivative(func,x,method = 'Chebyshev Method')

plt.figure()
plt.plot(x,finite_difference[0])
plt.plot(x,complex_step[0])
plt.plot(x,richardson_extrapolation[0])
plt.plot(x,automatic_differentiation[0])
plt.plot(x,chebyshev_method[0])
plt.title("Checking all of the Methods")
plt.xlabel("x")
plt.ylabel("df/dx");
```



Part C: Applying Derivatives to Functions

In [144]:
```
def compareDerivatives(func,x):
    methods = ["Finite Difference",
               "Richardson Extrapolation",
               "Automatic Differentiation",
               "Chebyshev Method"]

    # Iterate over all methods
    for i in range(0,len(methods)):
        result = getDerivative(func,x,method = methods[i])
        deriv = result[0]
        error = result[1]
        runTime = result[2]
        plt.figure(2)
        plt.plot(x,deriv,label = methods[i])

        plt.figure(3)
        plt.plot(x,error,label = methods[i])

        print(np.mean(np.sqrt(error**2)),"\t (",methods[i]," Error )")

    plt.figure(1)
    plt.plot(x,func(x))
    plt.title("Function to Differentiate")
    plt.xlabel("x")
    plt.ylabel("f(x)")

    plt.figure(2)
    plt.title("Derivative Values")
    plt.xlabel("x")
    plt.ylabel("df/dx")
    plt.legend()

    plt.figure(3)
    plt.title("Errors Relative to Complex Step")
    plt.xlabel("x")
    plt.ylabel("Error")
    plt.legend()

    return
```

Test Problems

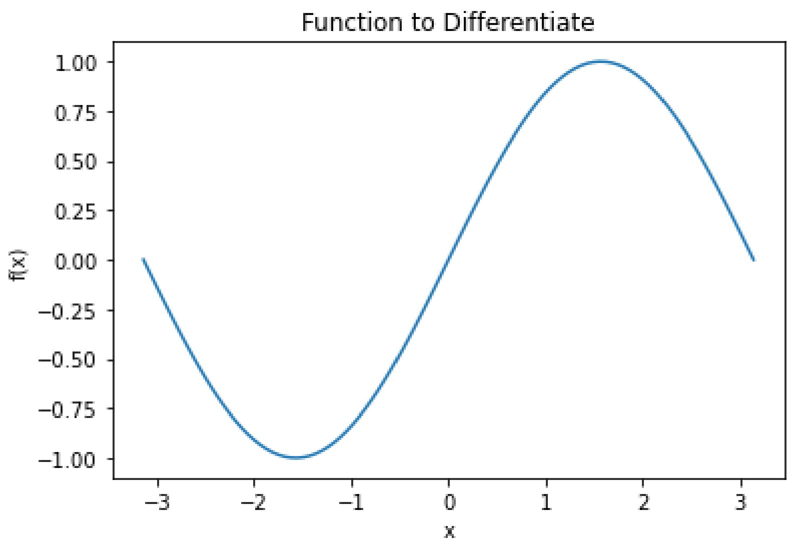
$$f(x) = \sin x \quad (x \in (-\pi, \pi)) \tag{13}$$

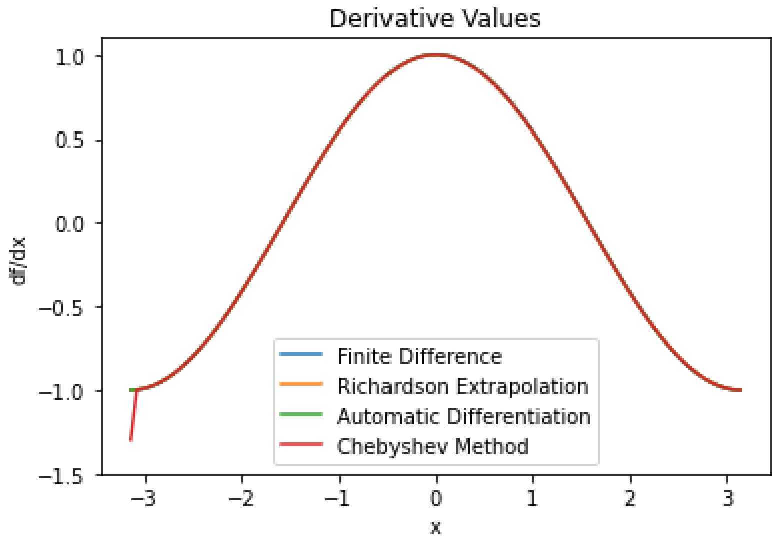
In [145]:

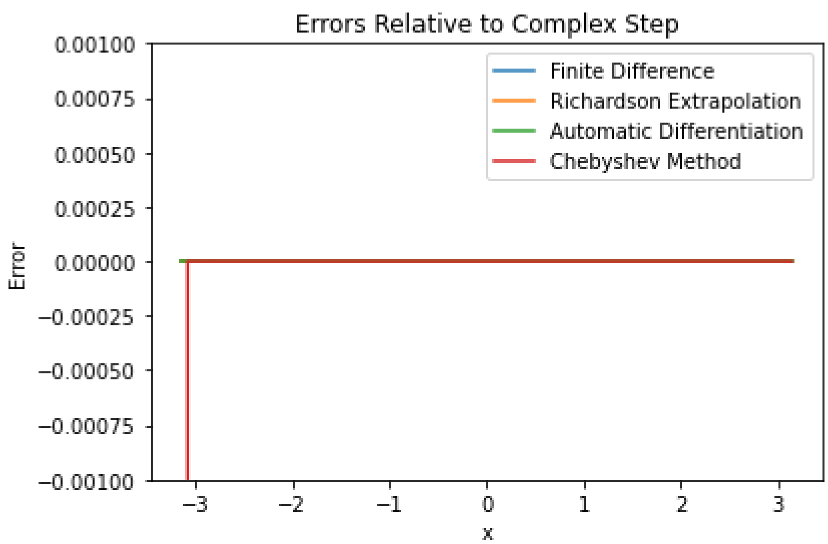
```
func = np.sin

x = np.linspace(-np.pi,np.pi,100)
compareDerivatives(func,x)

#--- Adjusting the y-axis limits so we can see everything well
plt.figure(2)
plt.ylim(-1.5,1.1)

plt.figure(3)
plt.ylim(-0.001,0.001)
```

```
5.91431633528705e-11      ( Finite Difference  Error )
9.916715365543282e-13     ( Richardson Extrapolation  Error )
1.8318679906315084e-17    ( Automatic Differentiation  Error )
0.0029755801612904266     ( Chebyshev Method  Error )
```

Out[145]: (-0.001, 0.001)





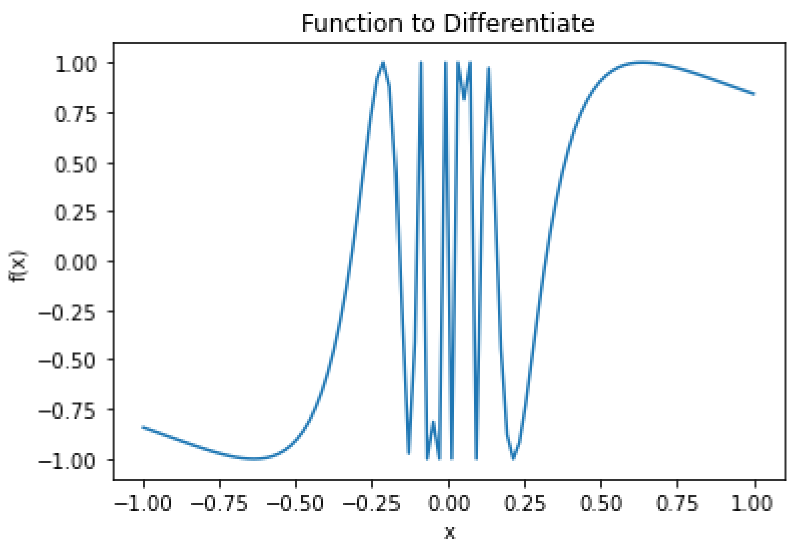


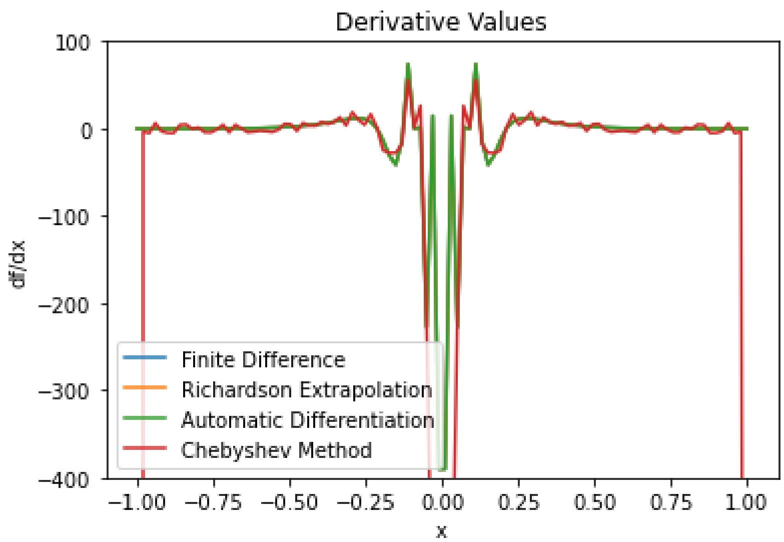
$$f(x) = \sin\left(\frac{1}{x}\right) \quad (x \in [-1, 1]) \tag{14}$$

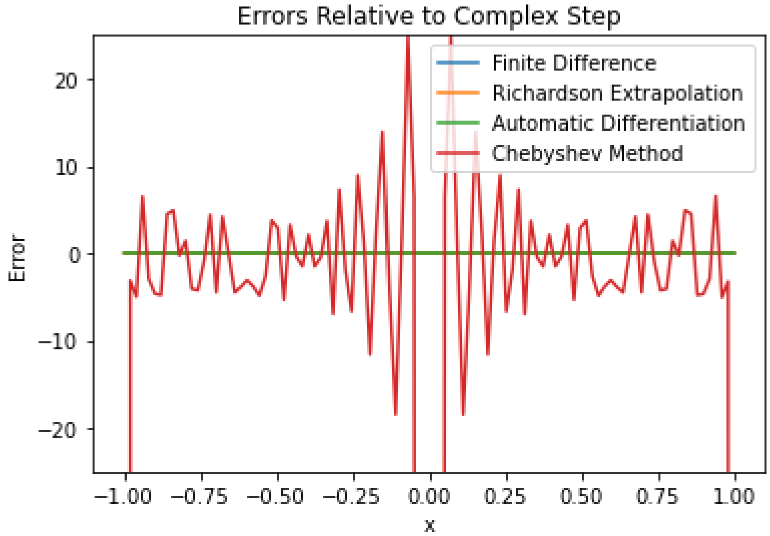
In [146]:
```
def func(x):
    return np.sin(1/x)

x = np.linspace(-1.0,1.0,100)
compareDerivatives(func,x)

#--- Adjusting the y-axis limits so we can see everything well
plt.figure(2)
plt.ylim(-400,100)

plt.figure(3)
plt.ylim(-25,25)
```

```
6.605948626013238e-05     ( Finite Difference  Error )
9.239995089414587e-05     ( Richardson Extrapolation  Error )
1.600390826805853e-15     ( Automatic Differentiation  Error )
2098.520093468335         ( Chebyshev Method  Error )
```

Out[146]: (-25.0, 25.0)





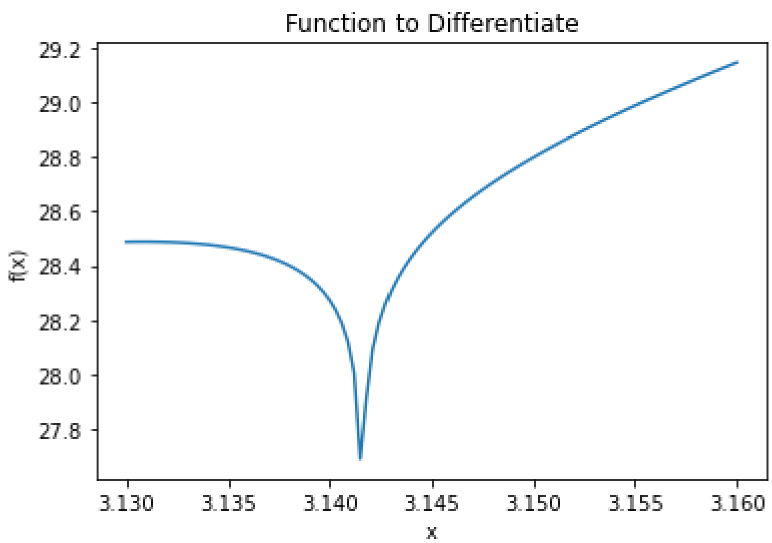


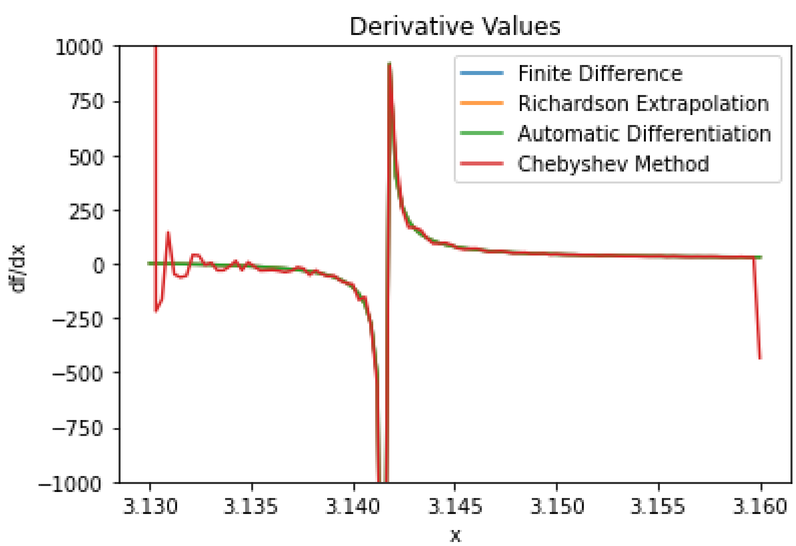
$$f(x) = 3x^2 + \frac{1}{\pi^2}\ln\left[(\pi - x)^2\right] \quad (x \in [3.13, 3.16]) \tag{15}$$

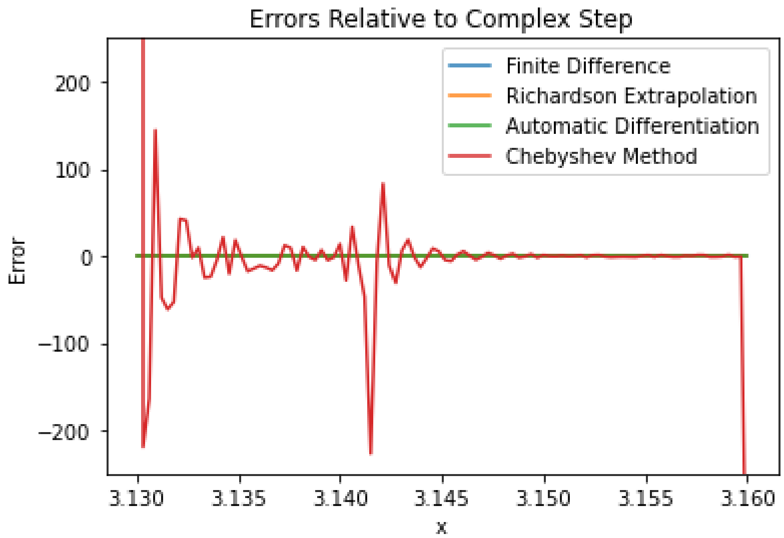
In [147]:
```python
def func(x):
    return 3*x**2 + 1/(np.pi**2) * np.log((np.pi - x)**2)

x = np.linspace(3.13,3.16,100)
compareDerivatives(func,x)

#--- Adjusting the y-axis limits so we can see everything well
plt.figure(2)
plt.ylim(-1000,1000)

plt.figure(3)
plt.ylim(-250,250)
```

```
0.0015328363930446765     ( Finite Difference  Error )
7.177737668784267e-08     ( Richardson Extrapolation  Error )
9.061640326990528e-15     ( Automatic Differentiation  Error )
4580585.623917096         ( Chebyshev Method  Error )
```

Out[147]: (-250.0, 250.0)







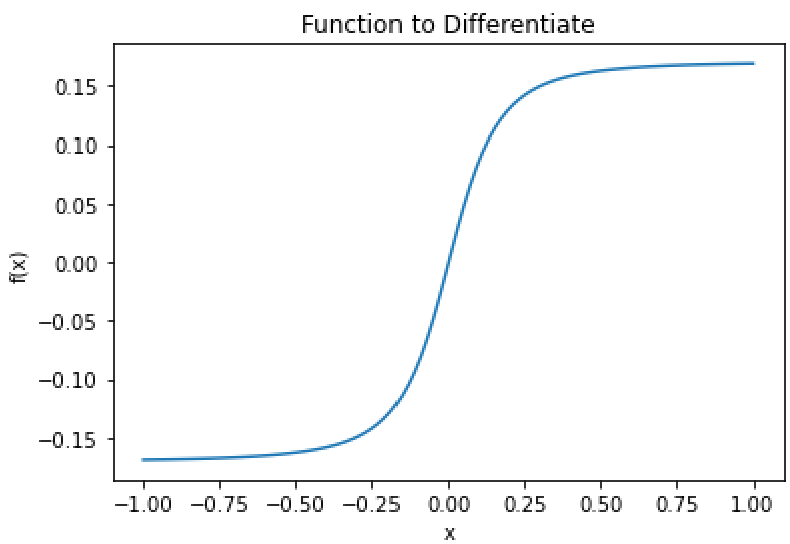
$$f(x) = \sin(\sin((\sin(\ldots\sin(x))))) \quad (x \in [-1, 1]) \tag{16}$$

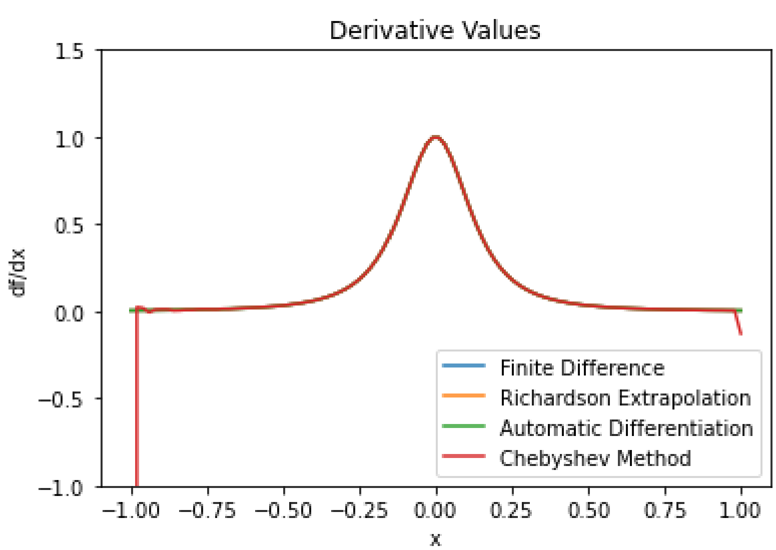
$$100 \text{ iterations} \tag{17}$$

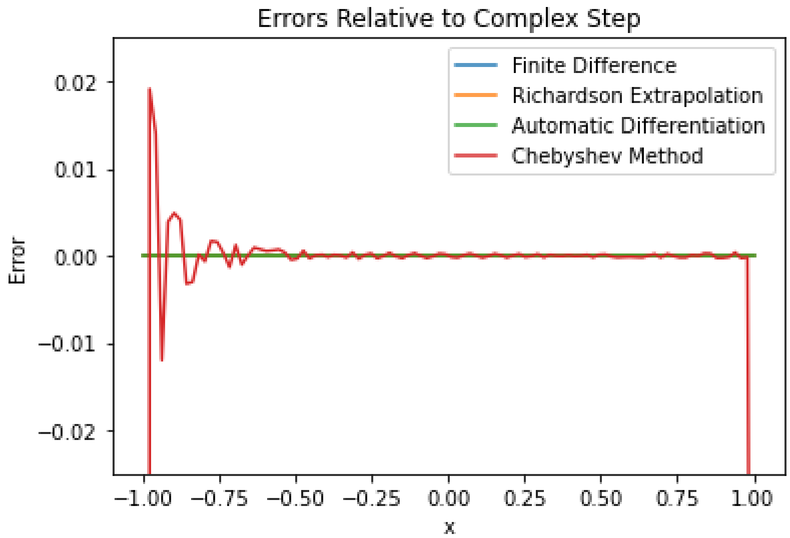
In [148]:
```
def func(x):
    value = x
    for i in range(0,100):
        value = np.sin(value)

    return value

x = np.linspace(-1,1,100)
compareDerivatives(func,x)

#--- Adjusting the y-axis limits so we can see everything well
plt.figure(2)
plt.ylim(-1,1.5)

plt.figure(3)
plt.ylim(-0.025,0.025)
```

```
2.501244360068877e-11     ( Finite Difference  Error )
3.900585401547807e-12     ( Richardson Extrapolation  Error )
9.429089453671934e-17     ( Automatic Differentiation  Error )
407.25783174617135        ( Chebyshev Method  Error )
```

Out[148]: (-0.025, 0.025)







$$f(x) = \frac{1}{|y|} \quad (x \in [-3, 3]) \tag{18}$$

where

$$y \in R^2 \text{satisfies} \begin{pmatrix} 1 & x \\ 2 & x^2 \end{pmatrix} y = \begin{pmatrix} 1 \\ 1 \end{pmatrix} \tag{19}$$

We can invert the matrix to find that

$$y = \frac{1}{x^2 - 2x} \begin{pmatrix} x^2 & -x \\ -2 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \begin{pmatrix} \frac{x-1}{x-2} \\ \frac{-1}{x-2} \end{pmatrix} \tag{20}$$

which means that

$$||y||(x) = \frac{1}{x(x-2)} \sqrt{x^2(x-1)^2 + 1} \tag{21}$$

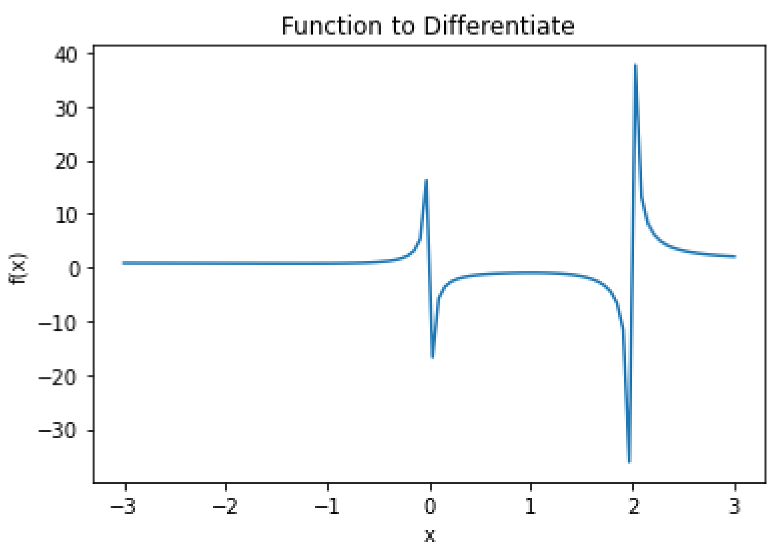
In [149]:

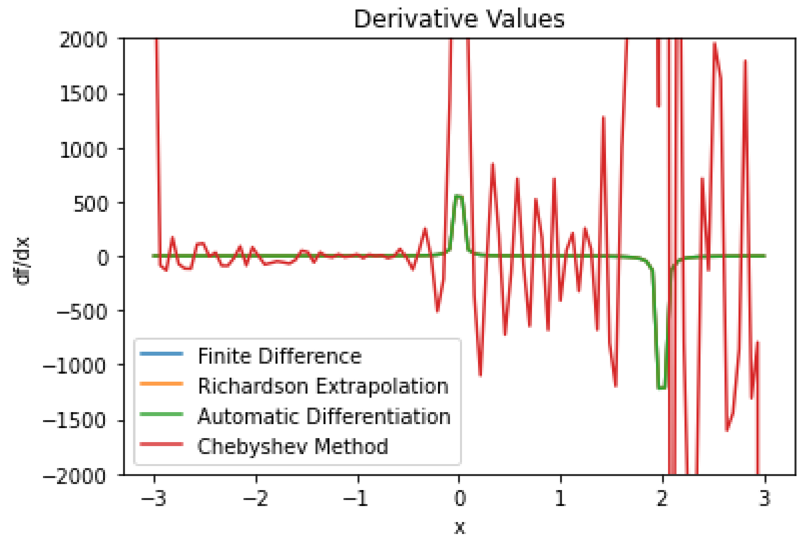
```
def func(x):
    return 1/x/(-2.+x)*np.sqrt(x**2*(-1+x)**2 + 1)

x = np.linspace(-3, 3, 100)
compareDerivatives(func, x)

#--- Adjusting the y-axis limits so we can see everything well
plt.figure(2)
plt.ylim(-2000,2000)

plt.figure(3)
plt.ylim(-2000,2000)
```

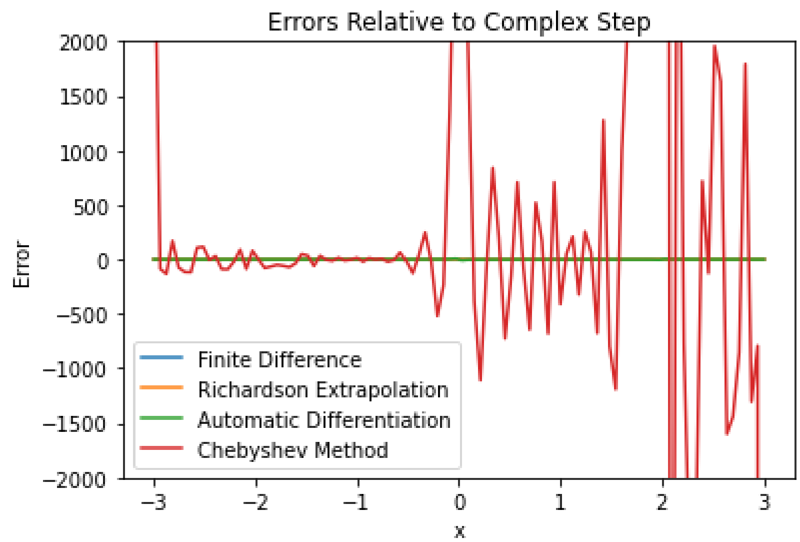
```
4.0057703411819705e-08    ( Finite Difference  Error )
1.5802988547093667e-10    ( Richardson Extrapolation  Error )
0.6029198086126173        ( Automatic Differentiation  Error )

<ipython-input-149-333c5cb4df45>:2: RuntimeWarning: divide by zero encoun
tered in true_divide
  return 1/x/(-2.+x)*np.sqrt(x**2*(-1+x)**2 + 1)

6366.9264499453675        ( Chebyshev Method  Error )
```

Out[149]:

```
(-2000.0, 2000.0)
```







Part D: Design a function to outperform other methods

Scott's Function

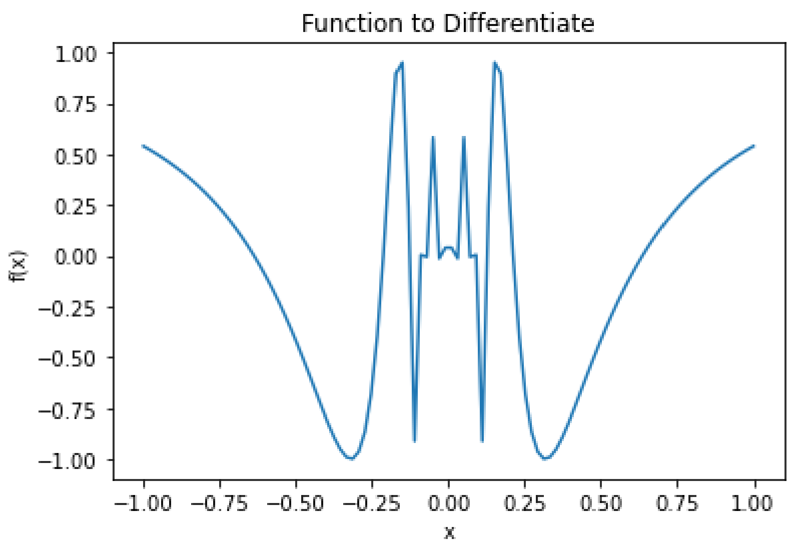
Good for Automatic Differentiation, bad for other methods:

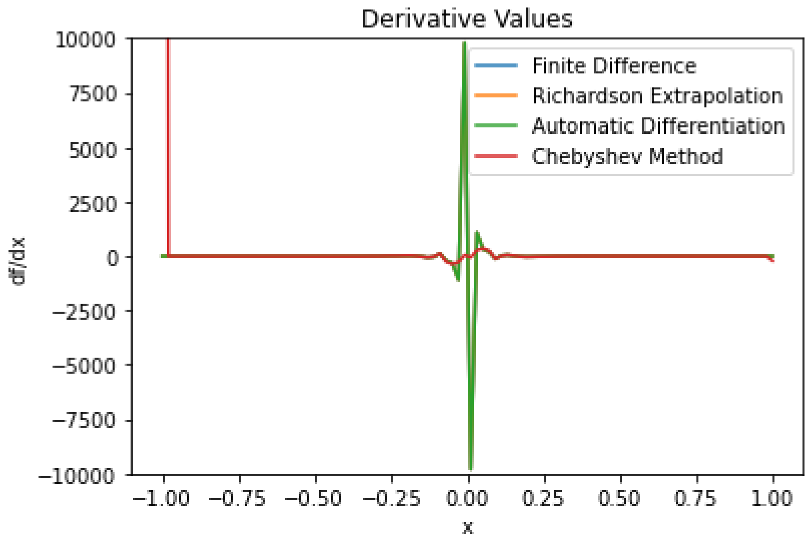
$$f(x) = \cos(1/x) \tag{22}$$

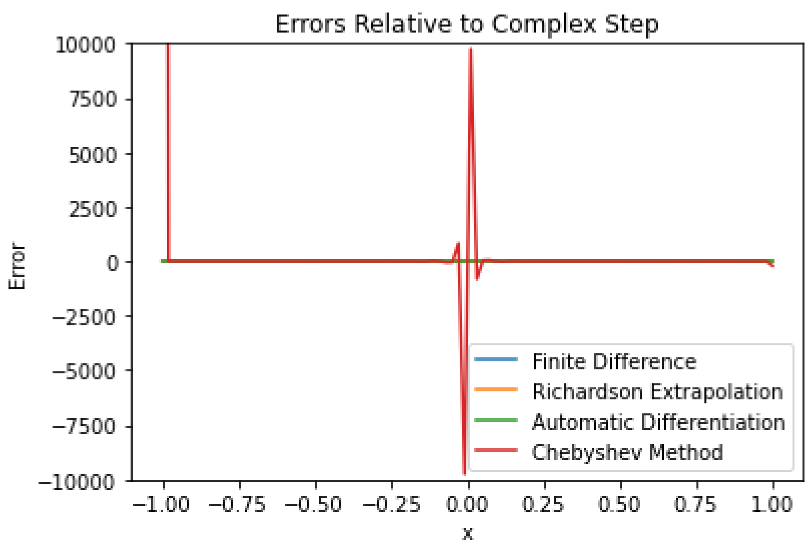
In [150]:
```
def func(x):
    return np.cos(1/x)

x = np.linspace(-1,1,100)
compareDerivatives(func,x)

#--- Adjusting the y-axis limits so we can see everything well
plt.figure(2)
plt.ylim(-10000,10000)

plt.figure(3)
plt.ylim(-10000,10000)
```

```
0.0031458683495831584    ( Finite Difference  Error )
0.0005674026927829124    ( Richardson Extrapolation  Error )
2.255973186038318e-14    ( Automatic Differentiation  Error )
1521.8579451967132       ( Chebyshev Method  Error )
```

Out[150]: (-10000.0, 10000.0)







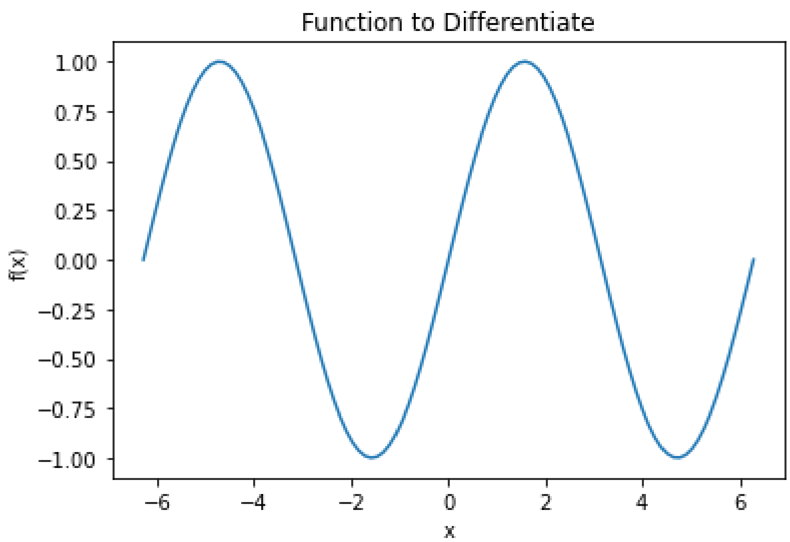
Mark's Function

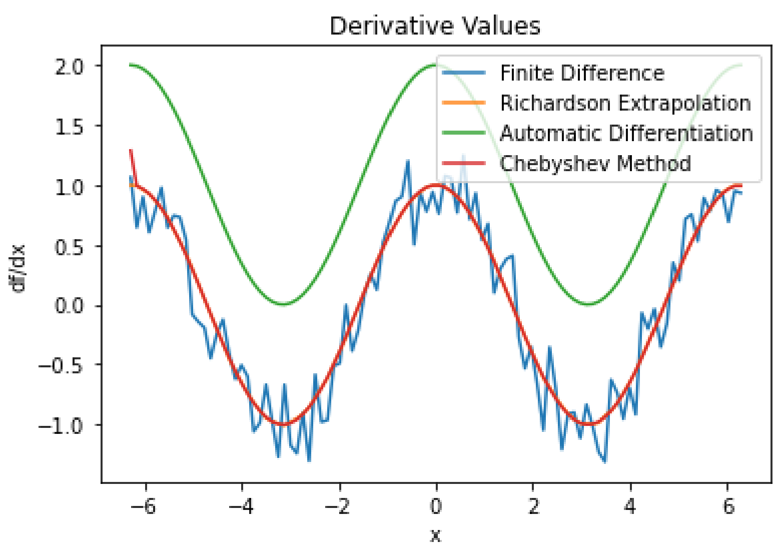
Let's try adding random noise to a function to make it no longer analytic:

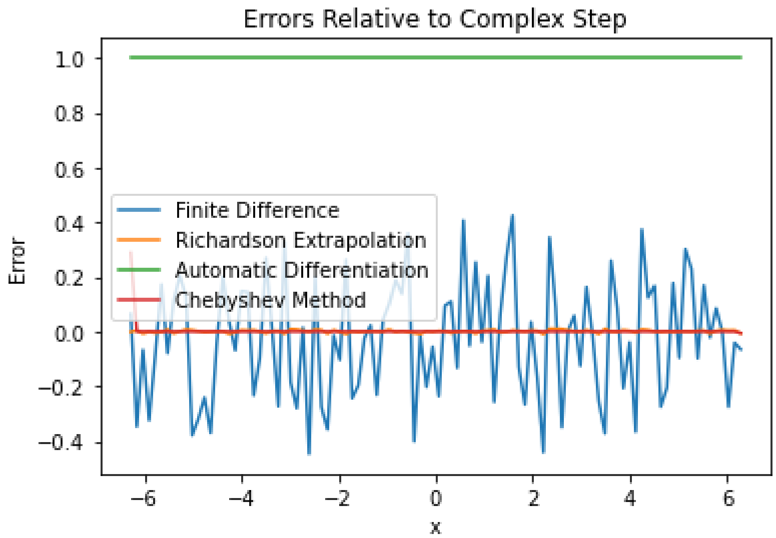
$$f(x) = \sin x + \text{Noise} \tag{23}$$

In [151]:
```
def func(x):
    return np.sin(x) + np.random.rand(np.size(x))/(1E6)

x = np.linspace(-2*np.pi,2*np.pi,100)
compareDerivatives(func,x)
```

```
0.18135052890589778      ( Finite Difference  Error )
0.0038453902207276043    ( Richardson Extrapolation  Error )
1.0      ( Automatic Differentiation  Error )
0.002979253838536221     ( Chebyshev Method  Error )
```







Aiden's Function

We see that automatic differention fails to noise added to a function, and the chebyshev method doesn't like $\sin\left(\frac{1}{x}\right)$, so if we combine the two, richardson extrapolation should outperform the two:

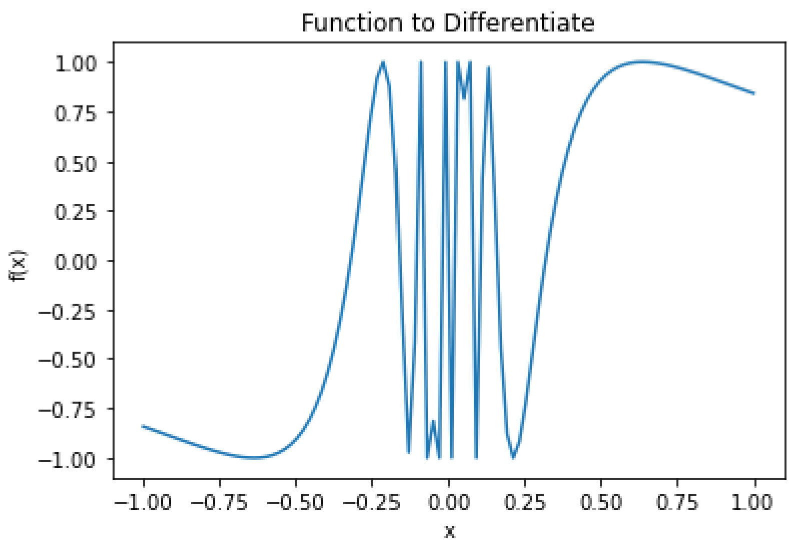
$$f(x) = \sin\left(\frac{1}{x}\right) + \text{Noise} \tag{24}$$

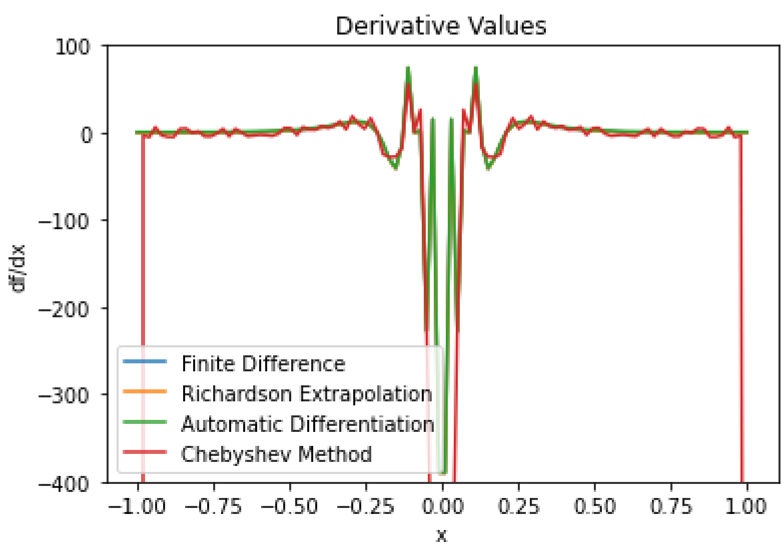
In [152]:

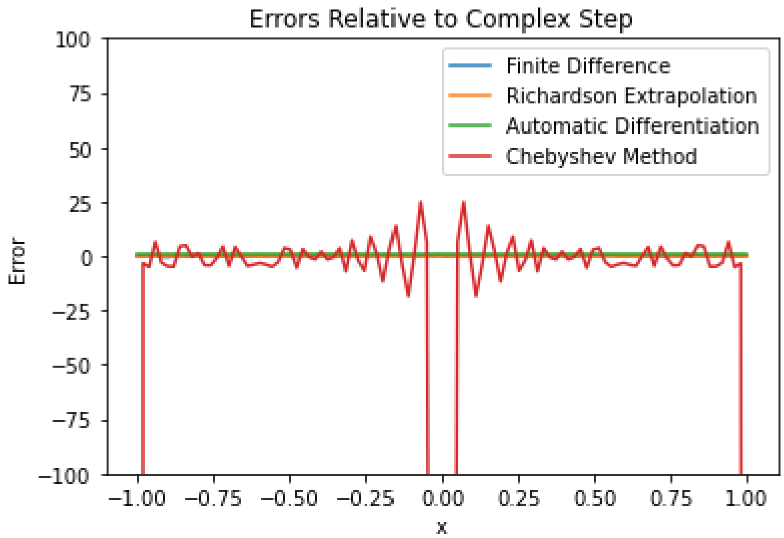
```python
def func(x):
    return np.sin(1/x) + np.random.rand(np.size(x))/(1E6)

x = np.linspace(-1,1,100)
compareDerivatives(func,x)

plt.figure(2)
plt.ylim(-400,100)

plt.figure(3)
plt.ylim(-100,100)
```

```
0.15231883273618027        ( Finite Difference  Error )
0.023639945227963758       ( Richardson Extrapolation  Error )
1.0000000000000004         ( Automatic Differentiation  Error )
2098.520774213828          ( Chebyshev Method  Error )
```

Out[152]: (-100.0, 100.0)







All three methods don't do that great with this function, but the error on Richardson Extrapolation is the lowest