Math 4610 Tasksheet 2

Jacob Fitzgerald (A02261889)

Docs

https://jfitzusu.github.io/math4610/

Code

https://github.com/jfitzusu/math4610/tree/main/Assignment02

Task 1

Code for Newton's Method:

```
import sympy

'''
Approximates the Root of a Function Using the Newton Method
f: The Function in Question. MUST BE A SYMPY EXPRESSION
x0: Initial Guess
tol: Maximum Permissible Error
maxIter: Maximum Iterations to Test Before Giving Up
-v: Verbose Mode, Tracks/Returns Intermittent Results
Returns: The Approximate Root, Convergence Status

NOTES: f'(x0) Cannot Equal 0
NOTES: FUNCTION MUST BE A SYMPY EXPRESSION
'''
def newton(f, x0, tol, maxIter=1000, v=False):
    x1 = 0
    x = sympy.symbols('x')
    derF = sympy.diff(f, x)
    error = 10 * tol
    for i in range(maxIter):
        derivative = derF.subs(x, x0)

        if derivative == 0:
            raise Exception("ERROR: Encountered Invalid Derivative {0}")

        x1 = x0 - f.subs(x, x0) / derivative
        error = abs(x1 - x0)
        if v:
            print(f'Iteration={i}. xApprox={x1}. Error={error}')

        if error <= tol:
            break

        x0 = x1

    return x1, error <= tol
```

This function approximates the root of a (mathematical) function using Newton's method. It excepts serveral parameters:

* f: A SymPy Representation of the Function
* x0: Initial Guess for the Root
* tol: Maximum Permissible Error
* maxIter: Maximum Steps to Try Before Giving Up
* v: Verbose Mode

The code returns the approximation of the root, as well as True/False depending on whether convergence was reached within the maximum number of iterations.

It should be noted that f'(xApprox) cannot equal 0, otherwise the function won't know what direction to proceed in.

Newton's method approximates the root by using the slope of the function at an initial guess (calculated with the derivative) in order to approximate the next step, hoping for convergence. The derivitive of the function is calculated programatically, using the SymPy library.

Code for Testing:

```
def testNewton():
    print("TESTING NEWTON METHOD")
    print("---------------------")
    x = sympy.symbols('x')
    expr = x * math.e ** -x

    for i in range(-3, 4):
        sol = newton(expr, i + 0.2, 0.001)
        print(f"The Approximate Root of x * e ^ -x Using {i + 0.2} as an Initial
Guess is {sol[0]}")
        print(f"CONVERGED: {sol[1]}.")
    print()
```

This code set's up a SymPy expression representing the function f(x) = x * e ^ -x. It then attempts to use Newton's Root Finding Method as describe above with incremental starting points over the interval [-2.8, 3.2] and prints the results.

This code can be found in the main directory of this project, inside the *test.py* file.

Testing Output:

```
TESTING NEWTON METHOD
---------------------
The Approximate Root of x * e ^ -x Using -2.8 as an Initial Guess is
-4.84985051763171E-9
CONVERGED: True.
The Approximate Root of x * e ^ -x Using -1.8 as an Initial Guess is
-1.57405083662271E-11
CONVERGED: True.
The Approximate Root of x * e ^ -x Using -0.8 as an Initial Guess is
-3.94254168339394E-9
CONVERGED: True.
The Approximate Root of x * e ^ -x Using 0.2 as an Initial Guess is
-3.19841468958201E-11
```

```
CONVERGED: True.
The Approximate Root of x * e ^ -x Using 1.2 as an Initial Guess is
1011.22139775427
CONVERGED: False.
The Approximate Root of x * e ^ -x Using 2.2 as an Initial Guess is
1008.70733923114
CONVERGED: False.
The Approximate Root of x * e ^ -x Using 3.2 as an Initial Guess is
1009.16089902031
CONVERGED: False.
```

As we can see from our results, our function has (at least) one root at x=0. At values of x close to 0, Newton's method will converge, while it will diverge at values greater than 1. This is due to the behavior of our original functions derivative.

Task 2

Code for Secant Method:

```
'''
Approximates the Root of a Function Using the Secant Method
f: The Function in Question, a Lambda
x0: Initial Guess
x1: Second Initial Guess
tol: Maximum Permissible Error
maxIter: Maximum Iterations to Test Before Giving Up
-v: Verbose Mode
Returns: The Approximate Root, Convergence Status
'''
def secant(f, x0, x1, tol, maxIter=1000, v=False):
    f0 = f(x0)
    f1 = f(x1)
    x2 = 0
    error = tol * 10

    for i in range(maxIter):
        x2 = x1 - f1 * (x1 - x0) / (f1 - f0)
        error = abs(x2 - x1)

        if v:
            print(f"Iteration={i}. xApprox={x2}. Error={error}")

        if error <= tol:
            break

        x0 = x1
        x1 = x2
        f0 = f1
        f1 = f(x1)

    return x2, error <= tol
```

This function approximates the root of a (mathematical) function using the Secant method. It excepts serveral parameters:

* f: A SymPy Representation of the Function
* x0: Initial Guess for the Root
* x1: Initial Second Guess for the Root
* tol: Maximum Permissible Error
* maxIter: Maximum Steps to Try Before Giving Up
* v: Verbose Mode

The code returns the approximation of the root, as well as True/False depending on whether convergence was reached within the maximum number of iterations.

The Secant method is very similar to Newton's method. However, it uses an approximation for the derivitive rather than the actual derivative. This method should be used when the derivitive of a function is very expensive to calculate/evaluate.

Code for Testing:

```
def testSecant():
    print("TESTING SECANT METHOD")
    print("---------------------")
    f = lambda x: x * math.e ** -x

    for i in range(-3, 4):
        sol = secant(f, i - 0.2, i, 0.001)
        print(f"The Approximate Root of x * e ^ - x Using {i - 0.2} and {i} as
Initial Values is {sol[0]}")
        print(f"CONVERGED: {sol[1]}")
    print()
```

This code sets up our initial function f(x) = x * e ^ -x as a Python lambda function. It then uses the Secant Method above to test for roots on the interval [-3.2, 3], hoping for convergence. It then reports the results.

This code can be found in the main directory of this project, inside the *test.py* file.

Testing Output:

```
TESTING SECANT METHOD
---------------------
The Approximate Root of x * e ^ - x Using -3.2 and -3 as Initial Values is
-1.2606547390883835e-06
CONVERGED: True
The Approximate Root of x * e ^ - x Using -2.2 and -2 as Initial Values is
-1.9046048655496834e-06
CONVERGED: True
The Approximate Root of x * e ^ - x Using -1.2 and -1 as Initial Values is
-4.413357882363867e-07
CONVERGED: True
The Approximate Root of x * e ^ - x Using -0.2 and 0 as Initial Values is 0.0
CONVERGED: True
The Approximate Root of x * e ^ - x Using 0.8 and 1 as Initial Values is
0.9996393159369001
CONVERGED: True
```

```
The Approximate Root of x * e ^ - x Using 1.8 and 2 as Initial Values is
701.6745834655374
CONVERGED: False
The Approximate Root of x * e ^ - x Using 2.8 and 3 as Initial Values is
702.0129256425017
CONVERGED: False
```

As we can see, our function again is seen as having (at least) one root at zero. It's interesting to note that the secant method doesn't seem to "blow up" as much when it doesn't converge, which may or may not save some calulation power if that's actually the case.

Task 3

Code for Verbose Mode:

```
Code is present in Tasks 1 and 2. It's essentially a 2 line difference for each,
so it's much simpler just to include one, complete copy of each method in the
report.
```

Example Code for Testing (Newton):

```
x = sympy.symbols('x')
expr = x * math.e ** -x
x0 = 0.5
print(newton(expr, x0, 0.0000001, v=True))
```

In this code, we simply set up our f(x) = x * e ^ -x as before, and run the newton method with the sympy expression generated, as well as an initial guess of 0.5, and an tolerance of 0.00000001. In this case, however, verbose mode is toggled, so our function will print out it's itermediate results.

Testing Output (Newton):

```
Iteration=0. xApprox=-0.500000000000000. Error=1.00000000000000
Iteration=1. xApprox=-0.166666666666667. Error=0.333333333333333
Iteration=2. xApprox=-0.0238095238095238. Error=0.142857142857143
Iteration=3. xApprox=-0.000553709856035436. Error=0.0232558139534884
Iteration=4. xApprox=-3.06424934164618E-7. Error=0.000553403431101272
Iteration=5. xApprox=-9.38962114881332E-14. Error=3.06424840268406E-7
Iteration=6. xApprox=-8.80999858950826E-27. Error=9.38962114881244E-14
(-8.80999858950826e-27, True)
```

As we can see, the function prints out it's intermediate results, showing how the approximation converges step by step. In this case, we actually converge pretty rapidly, as our initial guess is pretty close, and the derivitive is well behaved in that area as well. Once again, we find that our function has a root at 0.

Example Code for Testing (Secant):

```
print(secant(lambda x: x * math.e ** -x, 0.5, 1, 0.0000001, v=True))
```

In this example, we set up our initial function as a lambda yet again, and pass it to our secant function with initial guesses of 0.5 and 1.0. Addtionally, we set the tolerance to 0.00000001, and turn on verbose mode. As verbose mode is on, it should print out it's intermediate results.

Testing Output (Secant):

```
Iteration=0. xApprox=-1.8467422493615944. Error=2.8467422493615944
Iteration=1. xApprox=0.9132678841357524. Error=2.760010133497347
Iteration=2. xApprox=0.8295032530647222. Error=0.08376463107103027
Iteration=3. xApprox=-5.863499375307308. Error=6.69300262837203
Iteration=4. xApprox=0.8283297831858931. Error=6.691829158493201
Iteration=5. xApprox=0.8271568031264656. Error=0.0011729800594275641
Iteration=6. xApprox=-3.977541263244942. Error=4.804698066371407
Iteration=7. xApprox=0.8189863481017046. Error=4.796527611346646
Iteration=8. xApprox=0.810844076457881. Error=0.008142271643823529
Iteration=9. xApprox=-3.5878173219371723. Error=4.3986613983950535
Iteration=10. xApprox=0.7986567562963298. Error=4.386474078233502
Iteration=11. xApprox=0.7865389915716535. Error=0.012117764724676294
Iteration=12. xApprox=-3.028597785937831. Error=3.815136777509484
Iteration=13. xApprox=0.7648310945536778. Error=3.7934288804915086
Iteration=14. xApprox=0.7433810620829245. Error=0.021450032470753277
Iteration=15. xApprox=-2.3118599551443175. Error=3.055241017227242
Iteration=16. xApprox=0.6977887659573034. Error=3.009648721101621
Iteration=17. xApprox=0.6536532226579213. Error=0.044135543299382185
Iteration=18. xApprox=-1.405837772355623. Error=2.0594909950135443
Iteration=19. xApprox=0.5383792236912337. Error=1.9442169960468567
Iteration=20. xApprox=0.43736919941683333. Error=0.10101002427440037
Iteration=21. xApprox=-0.4590283713021122. Error=0.8963975707189455
Iteration=22. xApprox=0.1864267223380533. Error=0.6454550936401655
Iteration=23. xApprox=0.0730924410044726. Error=0.1133342813335807
Iteration=24. xApprox=-0.015638951891051633. Error=0.08873139289552423
Iteration=25. xApprox=0.0011761032398240508. Error=0.016815055130875684
Iteration=26. xApprox=1.826054200285295e-05. Error=0.0011578426978211978
Iteration=27. xApprox=-2.1489113119010168e-08. Error=1.828203111597196e-05
Iteration=28. xApprox=3.9240643126575127e-13. Error=2.1489505525441434e-08
(3.9240643126575127e-13, True)
```

As we can see, our function accuratley prints out the approximations at each iteration, so we can conclude that our verbose mode is working. Our end result also equals the value of the last approximation, so we shouldn't be missing any either. Once again, we get zero as an approximate root for our function. Convergence is slower than with the newton example above, but that's mainly due to our second guess being much farther off.

Task 4

Code for Hybrid Newton Method:

```
'''
Approximates the Root of a Function Using the Newton Method, Falling Back to
Bisection if It Doesn't Converge
f: The Function in Question. MUST BE A SYMPY EXPRESSION
a: Lower Boundary
b: Upper Boundary
tol: Maximum Permissible Error
```

```
maxIter: Maximum Iterations to Test Before Giving Up
maxTries: Maximum Times to Reduce the Problem With Bisection Before Giving Up
strictInterval: Boolean, Dictates Whether Convergence Must Happen Within the
Interval

Returns: The Approximate Root, Convergence Status

NOTES: f'(b) Cannot Equal 0
NOTES: FUNCTION MUST BE A SYMPY EXPRESSION
'''



def hybridNewton(f, a, b, tol, maxiter=1000, maxTries=10, strictInterval=False):
    error = 10 * tol
    x = sympy.symbols('x')
    derF = sympy.diff(f, x)

    for i in range(maxTries):

        # Setup for Newton Method
        x0 = b
        x1 = 0
        for j in range(maxiter):
            derivative = derF.subs(x, x0)

            if derivative == 0:
                raise Exception("ERROR: Encountered Invalid Derivative {0}")

            x1 = x0 - f.subs(x, x0) / derivative

            error = abs(x1 - x0)

            if error <= tol:
                break

            x0 = x1

        # Returns Root if Convergence Happened
        if strictInterval:
            if error < tol and a < x1 < b:
                return x1, True
        elif error < tol:
            return x1, True

        # Reduces Interval With Bisection if Convergence Failed
        fa = f.subs(x, a)
        fb = f.subs(x, b)

        for j in range(4):
            c = 0.5 * (a + b)
            fc = f.subs(x, c)

            if fa * fc < 0:
                fb = fc
                b = c
```

```
            elif fb * fc < 0:
                fa = fc
                a = c
            elif fc == 0:
                return c, True
            else:
                return c, False

        if abs(b - a) < tol:
            return b, True

    return b, False
```

This function approximates the root of a (mathematical) function using the "Hybrid Newton" method. It excepts serveral parameters:
* f: A SymPy Representation of the Function
* a: Left Side Bound for Interval Containing a Root
* b: Right Hand Bound for Interval Containing a Root
* tol: Maximum Permissible Error
* maxIter: Maximum Steps (Newton) to Try Before Giving Up
* maxTries: Maximum Steps (Bisection) to Try Before Giving Up
* strictInterval: Returns Roots Only Within the Initial Interval

This code returns the approximation of the root, as well as whether or not the approximation converged.

The "Hybrid Newton" method works by combining the bisection and Newton methods together to hopefully get more consistent results. The fucntion will first attempt to find convergence by applying the newton method to the midpoint of the interval. If it fails, four steps of the bisection method are used to reduce the interval size, and try again. This will loop untill the function either converges (during either the Newton or Bisection parts), or if it runs out of "tries", as specified in the paramters. It should be noted that if strictInterval is not set to true, the function can return values outside of the original interval.

Code for Testing:

```
def testHybridNewton():
    print("TESTING HYBRID NEWTON METHOD")
    print("----------------------------")
    x = sympy.symbols('x')
    expr = 10.14 * math.e ** (x * x) * sympy.cos(math.pi / x)
    sol = hybridNewton(expr, -3, 7, 0.001)
    print(f"The Approximate Root of 10.14 * e ^ (x ^ 2) * cos(PI/x) Within Range
[-3, 7] is {sol[0]}")
    print(f"CONVERGED: {sol[1]}.")
    print()

def allRootsHybridNewton(strict):
    print("TESTING ALL ROOTS FOR THE HYBRID NEWTON METHOD")
    print("---------------------------------------------")
    x = sympy.symbols('x')
    expr = 10.14 * math.e ** (x * x) * sympy.cos(math.pi / x)
    intervals = [(-3, -1), (-1, -0.5), (-0.5, -0.3), (-0.3, 0.2), (0.2, 0.25),
(0.25, 0.35), (0.35, 0.5), (0.5, 0.7), (0.7, 7)]
```

```
    for interval in intervals:
        sol = hybridNewton(expr, interval[0], interval[1], 0.001,
strictInterval=strict)
        print(f"The Approximate Root of 10.14 * e ^ (x ^ 2) * cos(PI/x) Within
Range [{interval[0]}, {interval[1]}] is {sol[0]}")
        print(f"CONVERGED: {sol[1]}.")
    print()
```

In this code, we test the function f(x) = 10.14 * e ^ (x ^ 2) * cos(PI/x) using our hybrid method. As we are using the newton method, a SymPy expression is used to represent the function. We first look at the overall interval [-3, 7], and then look at several intervals of note that should contain additional roots. For each interval, we run our function, and print the results. The multiple root version contains an option to enable strict interval mode.

This code can be found in the main directory inside the *test.py* file.

Testing Output:

```
TESTING HYBRID NEWTON METHOD
----------------------------
The Approximate Root of 10.14 * e ^ (x ^ 2) * cos(PI/x) Within Range [-3, 7] is
2.00000000664768
CONVERGED: True.

TESTING ALL ROOTS FOR THE HYBRID NEWTON METHOD
----------------------------------------------
The Approximate Root of 10.14 * e ^ (x ^ 2) * cos(PI/x) Within Range [-3, -1] is
0.285714201641803
CONVERGED: True.
The Approximate Root of 10.14 * e ^ (x ^ 2) * cos(PI/x) Within Range [-1, -0.5]
is 0.666666666665263
CONVERGED: True.
The Approximate Root of 10.14 * e ^ (x ^ 2) * cos(PI/x) Within Range [-0.5, -0.3]
is -0.285714285513292
CONVERGED: True.
The Approximate Root of 10.14 * e ^ (x ^ 2) * cos(PI/x) Within Range [-0.3, 0.2]
is -2.00000090705241
CONVERGED: True.
The Approximate Root of 10.14 * e ^ (x ^ 2) * cos(PI/x) Within Range [0.2, 0.25]
is -2.00000063850729
CONVERGED: True.
The Approximate Root of 10.14 * e ^ (x ^ 2) * cos(PI/x) Within Range [0.25, 0.35]
is 0.399999996895097
CONVERGED: True.
The Approximate Root of 10.14 * e ^ (x ^ 2) * cos(PI/x) Within Range [0.35, 0.5]
is -0.666666666665263
CONVERGED: True.
The Approximate Root of 10.14 * e ^ (x ^ 2) * cos(PI/x) Within Range [0.5, 0.7]
is 0.666666613719453
CONVERGED: True.
The Approximate Root of 10.14 * e ^ (x ^ 2) * cos(PI/x) Within Range [0.7, 7] is
2.00000000664768
CONVERGED: True.
```

As we can see from this result, strictInterval mode was not toggled on. Our function returns a root for every possible sub interval, but the root is not always within the subinterval. This is due to the unbounded behavior of the Newton method, compared to the bounded behavior of the bisection method.

Due to it's periodic nature our function techincally has infinite roots, so defining an interval for each is actually impossible, but we can observe a variety of different roots still.

Strict Output:

```
TESTING ALL ROOTS FOR THE HYBRID NEWTON METHOD
-----------------------------------------------
The Approximate Root of 10.14 * e ^ (x ^ 2) * cos(PI/x) Within Range [-3, -1] is
-1.99951171875
CONVERGED: True.
The Approximate Root of 10.14 * e ^ (x ^ 2) * cos(PI/x) Within Range [-1, -0.5]
is -0.666666666664977
CONVERGED: True.
The Approximate Root of 10.14 * e ^ (x ^ 2) * cos(PI/x) Within Range [-0.5, -0.3]
is -0.399999999959820
CONVERGED: True.
The Approximate Root of 10.14 * e ^ (x ^ 2) * cos(PI/x) Within Range [-0.3, 0.2]
is -0.285714284192987
CONVERGED: True.
The Approximate Root of 10.14 * e ^ (x ^ 2) * cos(PI/x) Within Range [0.2, 0.25]
is 0.222222207910603
CONVERGED: True.
The Approximate Root of 10.14 * e ^ (x ^ 2) * cos(PI/x) Within Range [0.25, 0.35]
is 0.285714285286432
CONVERGED: True.
The Approximate Root of 10.14 * e ^ (x ^ 2) * cos(PI/x) Within Range [0.35, 0.5]
is 0.399999984493125
CONVERGED: True.
The Approximate Root of 10.14 * e ^ (x ^ 2) * cos(PI/x) Within Range [0.5, 0.7]
is 0.666666613719453
CONVERGED: True.
The Approximate Root of 10.14 * e ^ (x ^ 2) * cos(PI/x) Within Range [0.7, 7] is
2.00000000664768
CONVERGED: True.
```

In this case, we simply set strict to true. As you can see, the function now only returns roots within the interval. We can see even more roots using this method.

Task 5

Code for Hybrid Secant Method:

```
'''
Approximates the Root of a Function Using the Secant Method, Falling Back to
Bisection if It Doesn't Converge
f: The Function in Question, a Lambda
a: Lower Boundary
b: Upper Boundary
tol: Maximum Permissible Error
maxIter: Maximum Iterations to Test Before Giving Up
```

```
maxTries: Maximum Times to Reduce the Problem With Bisection Before Giving Up
strictInterval: Boolean, Dictates Whether Convergence Must Happen Within the
Interval

Returns: The Approximate Root, Convergence Status
'''



def hybridSecant(f, a, b, tol, maxiter=1000, maxTries=10, strictInterval=False):
    error = 10 * tol

    for i in range(maxTries):

        # Setup for Secant Method
        x0 = b
        x1 = 10 * tol + x0
        x2 = 0

        f0 = f(x0)
        f1 = f(x1)
        for j in range(maxiter):

            x2 = x1 - f1 * (x1 - x0) / (f1 - f0)

            error = abs(x2 - x1)

            if error <= tol:
                break

            x0 = x1
            x1 = x2
            f0 = f1
            f1 = f(x1)

        # Returns Result if Convergence Happened
        if strictInterval:
            if error < tol and a < x1 < b:
                return x1, True
        elif error < tol:
            return x1, True

        # Reduces Interval Using Bisection if Failed to Converge
        fa = f(a)
        fb = f(b)

        for j in range(4):
            c = 0.5 * (a + b)
            fc = f(c)

            if fa * fc < 0:
                fb = fc
                b = c
            elif fb * fc < 0:
                fa = fc
                a = c
```

```
            elif fc == 0:
                return c, True
            else:
                return c, False

        # Returns Result if Bisection Managed to Converge, Otherwise Loops Back
to Secant Method
        if abs(b - a) < tol:
            return b, True

    return b, False
```

This function approximates the root of a (mathematical) function using the "Hybrid Secant" method. It excepts serveral parameters:

* f: A Lambda Representation of the Function
* a: Left Side Bound for Interval Containing a Root
* b: Right Hand Bound for Interval Containing a Root
* tol: Maximum Permissible Error
* maxIter: Maximum Steps (Newton) to Try Before Giving Up
* maxTries: Maximum Steps (Bisection) to Try Before Giving Up
* strictInterval: Returns Roots Only Within the Initial Interval

This code returns the approximation of the root, as well as whether or not the approximation converged.

The "Hybrid Secant" method works exactly the same as the "Hybrid Newton" method, simply using the secant method in place of the newton. It should be used in the same cases that the secant method should be used (complex derivatives).

Code for Testing:

```
def testHybridSecant():
    print("TESTING HYBRID SECANT METHOD")
    print("----------------------------")
    expr = lambda x: 10.14 * math.e ** (x * x) * math.cos(math.pi / x)
    sol = hybridSecant(expr, -3, 7, 0.001)
    print(f"The Approximate Root of 10.14 * e ^ (x ^ 2) * cos(PI/x) Within Range
[-3, 7] is {sol[0]}")
    print(f"CONVERGED: {sol[1]}.")
    print()

def allRootsHybridSecant(strict):
    print("TESTING ALL ROOTS FOR HYBRID SECANT METHOD")
    print("-----------------------------------------")
    expr = lambda x: 10.14 * math.e ** (x * x) * math.cos(math.pi / x)
    intervals = [(-3, -1), (-1, -0.5), (-0.5, -0.3), (-0.3, 0.2), (0.2, 0.25),
(0.25, 0.35), (0.35, 0.5), (0.5, 0.7), (0.7, 7)]
    for interval in intervals:
        sol = hybridSecant(expr, interval[0], interval[1], 0.001,
strictInterval=strict)
        print(f"The Approximate Root of 10.14 * e ^ (x ^ 2) * cos(PI/x) Within
Range [{interval[0]}, {interval[1]}] is {sol[0]}")
        print(f"CONVERGED: {sol[1]}.")
    print()
```

In this code, we test the function f(x) = 10.14 * e ^ (x ^ 2) * cos(PI/x) using our hybrid method. As we are using the secant method, a lamdba expression is used to represent the function. We first look at the overall interval [-3, 7], and then look at several intervals of note that should contain additional roots. For each interval, we run our function, and print the results. The multiple root version contains an option to enable strict interval mode.

This code can be found in the main directory inside the *test.py* file.

Testing Output:

```
TESTING HYBRID SECANT METHOD
----------------------------
The Approximate Root of 10.14 * e ^ (x ^ 2) * cos(PI/x) Within Range [-3, 7] is
2.0010210919301907
CONVERGED: True.

TESTING ALL ROOTS FOR THE HYBRID NEWTON METHOD
----------------------------------------------
The Approximate Root of 10.14 * e ^ (x ^ 2) * cos(PI/x) Within Range [-3, -1] is
-1.99951171875
CONVERGED: True.
The Approximate Root of 10.14 * e ^ (x ^ 2) * cos(PI/x) Within Range [-1, -0.5]
is -0.666666666664977
CONVERGED: True.
The Approximate Root of 10.14 * e ^ (x ^ 2) * cos(PI/x) Within Range [-0.5, -0.3]
is -0.399999999959820
CONVERGED: True.
The Approximate Root of 10.14 * e ^ (x ^ 2) * cos(PI/x) Within Range [-0.3, 0.2]
is -0.285714284192987
CONVERGED: True.
The Approximate Root of 10.14 * e ^ (x ^ 2) * cos(PI/x) Within Range [0.2, 0.25]
is 0.222222207910603
CONVERGED: True.
The Approximate Root of 10.14 * e ^ (x ^ 2) * cos(PI/x) Within Range [0.25, 0.35]
is 0.285714285286432
CONVERGED: True.
The Approximate Root of 10.14 * e ^ (x ^ 2) * cos(PI/x) Within Range [0.35, 0.5]
is 0.399999984493125
CONVERGED: True.
The Approximate Root of 10.14 * e ^ (x ^ 2) * cos(PI/x) Within Range [0.5, 0.7]
is 0.666666613719453
CONVERGED: True.
The Approximate Root of 10.14 * e ^ (x ^ 2) * cos(PI/x) Within Range [0.7, 7] is
2.00000000664768
CONVERGED: True.
```

In this case, we get the same results as with our hybrid newton function. It's apparent that strictInterval is off, as the results kind of end up all over the place. This can be fixed, as seen below.

Strict Output:

```
TESTING ALL ROOTS FOR HYBRID SECANT METHOD
```

```
----------------------------------------
The Approximate Root of 10.14 * e ^ (x ^ 2) * cos(PI/x) Within Range [-3, -1] is
-1.9998876271473107
CONVERGED: True.
The Approximate Root of 10.14 * e ^ (x ^ 2) * cos(PI/x) Within Range [-1, -0.5]
is -0.6666758963111825
CONVERGED: True.
The Approximate Root of 10.14 * e ^ (x ^ 2) * cos(PI/x) Within Range [-0.5, -0.3]
is -0.39921875
CONVERGED: True.
The Approximate Root of 10.14 * e ^ (x ^ 2) * cos(PI/x) Within Range [-0.3, 0.2]
is -0.28506357128807963
CONVERGED: True.
The Approximate Root of 10.14 * e ^ (x ^ 2) * cos(PI/x) Within Range [0.2, 0.25]
is 0.222265625
CONVERGED: True.
The Approximate Root of 10.14 * e ^ (x ^ 2) * cos(PI/x) Within Range [0.25, 0.35]
is 0.2855840834490296
CONVERGED: True.
The Approximate Root of 10.14 * e ^ (x ^ 2) * cos(PI/x) Within Range [0.35, 0.5]
is 0.39969147979083564
CONVERGED: True.
The Approximate Root of 10.14 * e ^ (x ^ 2) * cos(PI/x) Within Range [0.5, 0.7]
is 0.6658546859864347
CONVERGED: True.
The Approximate Root of 10.14 * e ^ (x ^ 2) * cos(PI/x) Within Range [0.7, 7] is
2.00102109193019O7
CONVERGED: True.


Process finished with exit code 0
```

In this case, we simply set strict to true. As you can see, the function now only returns roots within the interval. We can see even more roots using this method.