CP.1.4.7, Sauer3

Consider the function

$$f(x) = e^{\sin^3 x} + x^6 - 2x^4 - x^3 - 1$$

on the interval $[-2, 2]$. Plot the function on the interval, and find all three roots to six correct decimal places. Determine which roots converge quadratically, and find the multiplicity of the roots that converge linearly.

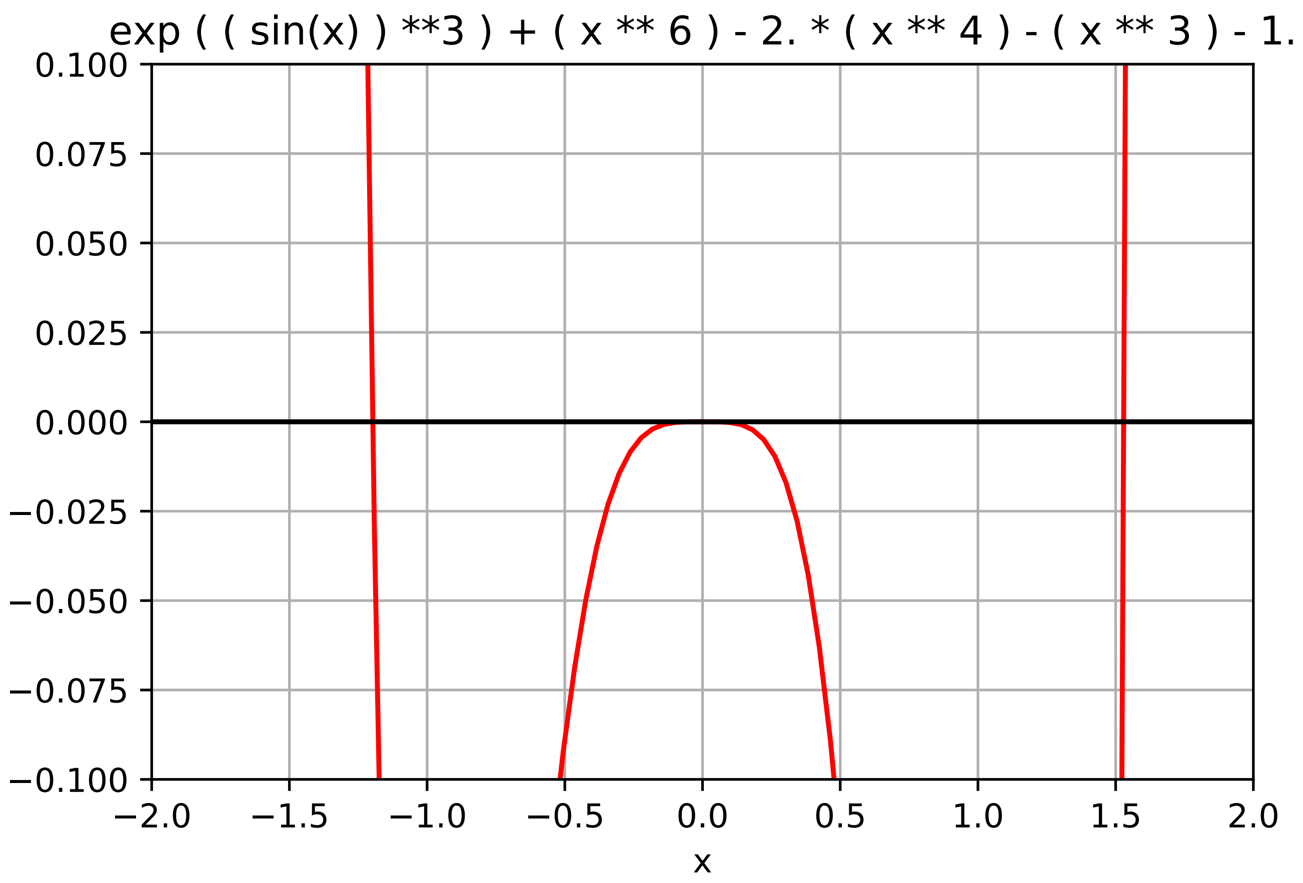
CP.1.4.7, Sauer3, solution, Langou

Colab Notebook: `https://colab.research.google.com/drive/1-akjROUL9xvwEIYW5sZDOQOx8mhapqjJ`

```
from math import exp
from math import sin
from math import cos
import scipy.optimize
import numpy as np
import matplotlib.pyplot as plt
```

```
f = lambda x : exp ( ( sin(x) ) **3 ) + ( x ** 6 ) \
                    - 2. * ( x ** 4 ) - ( x ** 3 ) - 1.
```

```
fnp = lambda x : np.exp ( ( np.sin(x) ) **3 ) + ( x ** 6 )  \
    - 2. * ( x ** 4 ) - ( x ** 3 ) - 1.
xx = np.linspace(-2, 2, 100)
yy = fnp(xx)
plt.plot(xx, yy, '-r', label='forward error bound');
xx = np.linspace(-2, 2, 10)
yy = np.zeros([10])
plt.plot(xx, yy, '-k', label='forward error bound');
plt.title('exp ( ( sin(x) ) **3 ) + ( x ** 6 )'+\
                 '- 2. * ( x ** 4 ) - ( x ** 3 ) - 1.')
plt.xlabel('x')
plt.xlim([-2., 2.])
plt.ylim([-0.1, 0.1])
plt.grid()
plt.show()
```



```
dfdx = lambda x : ( 3 * ( cos(x) ) * ( sin(x) ) **2 ) \
       * exp ( ( sin(x) ) **3 ) \
           + 6 * ( x ** 5 ) − 8. * ( x ** 3 ) − 3. * ( x ** 2 )
```

```
# getting the root close to 1.5
x0 = 1.5

x_fsolve = scipy.optimize.fsolve( f, x0 )[0]
print( "  ", f"{x_fsolve:.16f}" )

r = x_fsolve

x = x0
for i in range(0,4):
  x = x − f(x) / dfdx(x)
  true_fwd_rel_error = abs( r − x ) / abs( r )
  print( f"{i+1:2d}", f"{x:.16f}", f"{true_fwd_rel_error:.2e}" )

# we observe quadratic convergence
```

```
   1.5301335081665943
 1 1.5332246760734796 2.02e−03
 2 1.5301622402064223 1.88e−05
 3 1.5301335106759133 1.64e−09
 4 1.5301335081666154 1.38e−14
```

```
# getting the root close to -1.3
x0 = -1.3

x_fsolve = scipy.optimize.fsolve( f, x0 )[0]
print( "  ", f"{x_fsolve:.16f}" )

r = x_fsolve

x = x0
for i in range(0,5):
  x = x - f(x) / dfdx(x)
  true_fwd_rel_error = abs( r - x ) / abs( r )
  print( f"{i+1:2d}", f"{x:.16f}", f"{true_fwd_rel_error:.2e}" )

# we observe quadratic convergence
```

```
   -1.1976237221335699
 1 -1.2239067821719753 2.19e-02
 2 -1.1998686893584445 1.87e-03
 3 -1.1976417926765932 1.51e-05
 4 -1.1976237233157172 9.87e-10
 5 -1.1976237221335699 0.00e+00
```

```
# getting the root 0.
# 0.01 is close to 0 so we start with 0.01

x0 = 0.01

x_fsolve = scipy.optimize.fsolve( f, x0 )[0]
print( "  ", f"{x_fsolve:.16f}" )

# we note that even scipy.optimize.fsolve struggle to find the root 0
# on my laptop, it returns 0.0001110501047940 (which is quite far from 0)
#
# scipy.optimize.fsolve actually throws a warning:
#    The iteration is not making good progress, as measured by
#    the improvement from the last ten iterations.
```

```
   0.0001110501047940
/usr/local/lib/python3.7/dist-packages/scipy/optimize/minpack.py:162:
  RuntimeWarning: The iteration is not making good progress, as
  measured by the improvement from the last ten iterations.
  warnings.warn(msg, RuntimeWarning)
```

```
# (1) because the root is 0, we do not look at relative error here,
# we look at ``absolute`` error
# (2) also we do not take the solution from scipy.optimize.fsolve because
# (2a) scipy.optimize.fsolve struggles to find a solution and (2b) we know
# that the root is zero.

x = x0
for i in range(0,20):
  x = x - f(x) / dfdx(x)
  print( f"{i+1:2d}", f"{x:.16f}", f"{abs(x):.2e}" )

# With Newton's method, we observe super slow convergence
# (certainly not quadratic) and then actually we stagnate.
```

```
 1 0.0075014642430662 7.50e-03
 2 0.0056269359153278 5.63e-03
 3 0.0042206791139435 4.22e-03
 4 0.0031657803707797 3.17e-03
 5 0.0023744896563440 2.37e-03
 6 0.0017809538737954 1.78e-03
 7 0.0013357654152987 1.34e-03
 8 0.0010018576175091 1.00e-03
 9 0.0007514247915753 7.51e-04
10 0.0005635244314221 5.64e-04
11 0.0004226407785347 4.23e-04
12 0.0003171387831389 3.17e-04
13 0.0002383965438105 2.38e-04
14 0.0001789924949967 1.79e-04
15 0.0001330150215249 1.33e-04
16 0.0000976355466433 9.76e-05
17 0.0000827253589515 8.27e-05
18 0.0000827253589515 8.27e-05
19 0.0000827253589515 8.27e-05
20 0.0000827253589515 8.27e-05
```

```
# Note that while the ``x`` values found by either our Newton's method
# or scipy.optimize.fsolve are far away from zero. (So the forward error
# is large.) The ``f(x)`` are very small. (So the backward error
# is small.) This is typical from an ill-conditioned problem. We ask
# python to find x such that f(x) = 0, and, in the computer f(x) is 0.
# (Or very close to.)
print( f"{x_fsolve:.16f}", f"{f( x_fsolve ):+.2e}",  )
print( f"{x:.16f}", f"{f( x ):+.2e}",  )
```

```
0.0001110501047940 -1.11e-16
0.0000827253589515 +0.00e+00
```