2.1 #4

Each of the functions in (a) – (f) has a single root r = 0, and satisfies f(-1)\*f(1) < 0. Implement and run regula falsi on each of them, starting with the interval [-1, 1] and terminating when |Xn – r| ≤ tol, for the specific tolerance level. State the observed order of convergence and provide enough convergence history to justify your claim.

*# MTH451 - HW2 - 2.1 ex4*

*#-------------------------------------------------*

*# The following functions corr. to parts (a) - (f)*

**def** fA(x):

**if** x < 0:

y = x\*\*2

**else**:

y = -x\*\*2 / (2\*x + 1)

**return** y

**def** fB(x):

y = -2\*x / (x+3)

**return** y

**def** fC(x):

y = -999\*x / (x+1000)

**return** y

**def** fD(x):

y = (x\*\*2 - 2\*x) / (x\*\*2 + 2)

**return** y

**def** fE(x):

y = (x\*\*3 - 2\*x) / (x\*\*3 + 2)

**return** y

**def** fF(x):

p = 3/2; q = 1/2

y = (q \* abs(x)\*\*p - x) / (q \* abs(x)\*\*p + 1)

**return** y

*#---------------------------------------------------*

*# Check if there's a root in current interval*

**def** rootCheck(a, b, f):

**if** f(a) \* f(b) < 0:

**return** 1

**else**:

**return** 0

*# Run Regula Falsi over current interval [a, x] or [x, b]*

**def** falsi(f, tol):

a = -1; b = 1

r = 0

n = 0

x = (a\*f(b) - b\*f(a)) / (f(b) - f(a))

**while** abs(a - r) > tol **and** abs(b - r) > tol **and** n <= 100:

**if** rootCheck(a, b, f) == 1:

b = x

**else**:

a = x

print("f(a)\*f(b) > 0; bad search interval. Now using [x,b]:")

print(a, b)

n += 1

x = (a\*f(b) - b\*f(a)) / (f(b) - f(a))

*#Tolerance levels*

tol1 = 10\*\*(-12)

tol2 = 2\*\*(-52)

*# Call functions*

print("Part A convergence:")

falsi(fA, tol1)

print("**\n** Part B convergence:")

falsi(fB, tol1)

print("**\n** Part C convergence:")

falsi(fC, tol1)

print("**\n** Part D convergence:")

falsi(fD, tol2)

print("**\n** Part E convergence:")

falsi(fE, tol2)

print("**\n** Part F convergence:")

falsi(fF, tol2)

Using the output of this code, provided on the following page, it appears:

1. Converges sublinearly
2. Converges linearly
3. Converges linearly (but faster than b)
4. Converges quadratically
5. Converges cubically
6. Converges superlinearly

**Part A convergence:**

-1 0.5000000000000001

-1 0.33333333333333337

-1 0.25000000000000006

-1 0.2

-1 0.16666666666666669

-1 0.14285714285714288

-1 0.12500000000000003

-1 0.11111111111111113

-1 0.1

*. . .*

*Lines 11-90 Omitted*

*. . .*

-1 0.011111111111111106

-1 0.010989010989010985

-1 0.0108695652173913

-1 0.010752688172043008

-1 0.010638297872340424

-1 0.010526315789473682

-1 0.010416666666666664

-1 0.010309278350515462

-1 0.010204081632653059

-1 0.010101010101010098

-1 0.009999999999999998

-1 0.009900990099009901

-1 0.00980392156862745

**Part D convergence:**

-1 0.5000000000000001

-1 0.12500000000000003

-1 0.007812500000000002

-1 3.051757812500001e-05

-1 4.656612873077459e-10

-1 1.0842021719806335e-19

**Part B convergence:**

-1 0.3333333333333333

-1 0.11111111111111112

-1 0.037037037037037035

-1 0.01234567901234568

-1 0.004115226337448561

-1 0.0013717421124828533

-1 0.0004572473708276177

-1 0.00015241579027587258

-1 5.080526342529086e-05

-1 1.6935087808430286e-05

-1 5.645029269476762e-06

-1 1.8816764231589204e-06

-1 6.272254743863068e-07

-1 2.0907515812876892e-07

-1 6.96917193762563e-08

-1 2.3230573125418767e-08

-1 7.74352437513959e-09

-1 2.5811747917131966e-09

-1 8.60391597237732e-10

-1 2.8679719907924403e-10

-1 9.559906635974801e-11

-1 3.1866355453249343e-11

-1 1.0622118484416448e-11

-1 3.5407061614721493e-12

-1 1.1802353871573828e-12

-1 3.9341179571912764e-13

**Part E convergence:**

-1 0.5000000000000001

-1 0.06250000000000006

-1 0.00012207031249999997

-1 9.094947017737554e-13

-1 -0.0

**Part C convergence:**

-1 0.0010000000000000217

-1 1.000000000000059e-06

-1 1.0000000000002633e-09

-1 1.0000000000001274e-12

-1 1.0000000000001076e-15

**Part F convergence:**

-1 0.5000000000000001

-1 0.17677669529663698

-1 0.03716272234383505

-1 0.0035820470437682456

-1 0.00010719312502278412

-1 5.549072545898414e-07

-1 2.066810257414788e-10

-1 1.4856650839741543e-15

-1 2.8631979895300236e-23

2.2 Ex. 3

Let f(x) = cos2x and r = π/2. Run Newton’s method with x0 = 1. What is the observed order of convergence for xn -> r? If it is not quadratic, what conditions in Th. 2.15 concerning f and r are violated?

**import** **math**

**def** f(x):

y = math.cos(x)\*\*2

**return** y

**def** fprime(x):

*#y = -2 \* math.cos(x) \* math.sin(x)*

y = -math.sin(2\*x)

**return** y

*#Generates an array, called x, and fills it with iterations of Xn*

**def** newton():

n = 0

x = []; x.append(1)

**while**(abs(x[n] - r) > 10\*\*(-5)):

x\_next = x[n] - f(x[n]) / fprime(x[n])

x.append(x\_next)

n += 1

print("Xn:")

**for** i **in** x:

print(i)

**return** x

*#This function takes in array x and prints the error for (X\_n - r)/(X\_n-1 -r)*

**def** linErrChk():

n = 1

errors = []

**while**(n < len(x)):

error\_next = (x[n]-r)/(x[n-1]-r)

errors.append(error\_next)

n += 1

print("**\n** Errors (Xn - r)/(Xn-1 - r):")

**for** i **in** errors:

print(i)

*#global variables*

r = math.pi / 2

*#Call function*

newton()

linErrChk()

Based on the following convergence and error data, xn clearly converges sublinearly.

The condition violated is that f’(r) = 0.

**Xn:**

1

1.3210463079671655

1.4485841371633315

1.5099962837595349

1.540433820114687

1.555619740282817

1.563208616193309

1.567002544304102

1.5688994446500681

1.5698478868600323

1.5703221069696576

1.5705592169000513

1.5706777718496958

1.5707370493225739

1.5707666880587698

1.5707815074268376

1.5707889171108678

**Errors (Xn - r)/(Xn-1 - r):**

0.37842297990928364

2.6790976643902473

0.376689898531897

2.70353627462436

0.37326643923112507

2.730031531958215

0.3683387197981441

2.747388743604927

0.36447355936647585

2.7517418523759307

0.3634281377007364

2.751938027300025

0.36338031704994794

2.7519383938828326

0.3633802276327299

2.751938393884109

0.3633802276324186

2.751938393884109

0.3633802276324186

2.751938393884109

0.3633802276324186

2.751938393884109

0.3633802276324186

2.751938393884109

0.3633802276324186

2.751938393884109

0.3633802276324186

2.751938393884109

0.3633802276324186

2.751938393884109

**2.2 Ex. 6**

(Cycling) Let f(x) = x3 – 2x + 2. Recall that this function has a single root ≈ -1.769292354238631.

1. Suppose x0­­ = 0. Show that the Newton iterates are given by xn = (1 – ( -1)n )/2. In other words, they are 0, 1, 0, 1, . . .
2. Implement and run Newton’s method for this problem, starting with x0 = 0.1 and x0 = 0.2. Comment on the results in each case.

*#Below, we use the same Newton Implementation as in 2.2 ex.3*

*#---------------------------------*

**import** **math**

**def** f(x):

y = x\*\*3 - 2\*x + 2

**return** y

**def** fprime(x):

y = 3 \* x\*\*2 - 2

**return** y

*#Generates an array, called x, and fills it with iterations of Xn*

**def** newton(x0):

n = 0

nmax = 30

x = []; x.append(x0)

**while**(n < nmax):

x\_next = x[n] - f(x[n]) / fprime(x[n])

x.append(x\_next)

n += 1

**for** i **in** x:

print(i)

*#Initial Conditions*

x0\_a = 0.1

x0\_b = 0.2

*#global variables*

r = -1.769292354238631

print("Conv. for x0 = 0:")

x = newton(0)

print("**\n** Conv. for x0 = 0.1:")

x = newton(x0\_a)

print("**\n** Conv. for x0 = 0.2:")

newton(x0\_b)

Based on the data on the following page, it appears that for x0 = 0.1, the Newton iterates end up being given by xn = (1 – ( -1)n )/2 after the 16th iteration. For x0 = 0.2, the sequence converges to our root, r!

**Conv. for x0 = 0:**

0

1.0

0.0

1.0

0.0

1.0

0.0

1.0

0.0

1.0

0.0

1.0

0.0

1.0

0.0

1.0

0.0

1.0

0.0

1.0

0.0

1.0

0.0

1.0

0.0

1.0

0.0

1.0

0.0

1.0

0.0

**Conv. for x0 = 0.1:**

0.1

1.0142131979695432

0.07965576631987636

1.0090987403727651

0.05222652653371296

1.0039651847274838

0.02332943565497303

1.0008043531824031

0.004806794546775572

1.0000345480457813

0.0002072524744249904

1.0000000644214841

3.8652878042721994e-07

1.0000000000002243

1.3455903058456897e-12

1.0

0.0

1.0

0.0

1.0

0.0

1.0

0.0

1.0

0.0

1.0

0.0

1.0

0.0

1.0

0.0

**Conv. for x0 = 0.2:**

0.2

1.0553191489361704

0.2614390358394386

1.0943270305859323

0.38993146764411135

1.218649426200272

0.6596487069392855

2.052900431402751

1.437865220849364

0.9388626077809332

-0.5351654221916866

2.0218769578189595

1.415712077796703

0.9158011541665463

-0.8988034917994772

-8.150746134217721

-5.499048198867083

-3.771219375974292

-2.686974730056577

-2.0752826652454

-1.8200484597910154

-1.7710479662411496

-1.7692945635490052

-1.7692923542421366

-1.7692923542386314

-1.7692923542386314

-1.7692923542386314

-1.7692923542386314

-1.7692923542386314

-1.7692923542386314

-1.7692923542386314

**2.2 Ex. 11**

Use Newton’s method to approx. the root r = 1.2599271053493255045907517342129722238, starting with the initial guess x0 = 1. Simplify the iteration by hand before implementing it. How many iterations are required before |xn – r| is on the order of machine precision?

*#Below, we use the same Newton Implementation as in 2.2 ex.3*

*#---------------------------------*

**import** **math**

**def** f(x, p, q):

y = x\*\*3 - p\*x - q

**return** y

**def** fprime(x, p):

y = 3 \* x\*\*2 - p

**return** y

*#Generates an array, called x, and fills it with iterations of Xn*

**def** newton(x0):

n = 0

nmax = 100

x = []; x.append(x0)

**while**(n <= nmax):

x\_next = x[n] - f(x[n], p, q) / fprime(x[n], p)

x.append(x\_next)

n += 1

**for** i **in** x:

print("**%4.38e**"%(i))

*#global variables*

p = 1.1444091796875 \* 10\*\*(-5) *#p = 3\* 2\*\*(-18)*

q = 2

r = 1.2599271053493255045907517342129722238

*#Call function*

newton(1)

**Convergence for x0 = 1:**

1.00000000000000000000000000000000000000e+00

1.33333841961575672030448913574218750000e+00

1.26389213072394812620302673167316243052e+00

1.25993652251729626456722144212108105421e+00

1.25992407774502246375902814179426059127e+00

1.25992407762209923660634558473248034716e+00

1.25992407762209923660634558473248034716e+00

1.25992407762209923660634558473248034716e+00

1.25992407762209923660634558473248034716e+00

1.25992407762209923660634558473248034716e+00. . .

After 6 iterations, the convergence is on the order of machine precision.