CH2120

Class 19

# mainProgram.f08

**program** mainProgram

**implicit** **none**

**call** mainBracketingMethods()

**end** **program** mainProgram

# mainBracketingMethods.f08

**subroutine** mainBracketingMethods

**implicit** **none**

! External functions

**real**, **external** :: rootFindingFunction

**logical**, **external** :: haveOppositeSigns

! Variables: User Input

**real** xLow, xHigh

**real** tolerance

! Variables: Console Output

**real** root, error

**integer** iterations

**character**(*len*=35) methodName

! Variables: Internal

**real** fLow, fHigh

! Get user input for the (1) initial bracket and (2) tolerance.

**write**(\*,\*) "Enter the lower limit of the initial bracket:"

**read**(\*,\*) xLow

**write**(\*,\*) "Enter the lower limit of the initial bracket:"

**read**(\*,\*) xHigh

**write**(\*,\*) "Enter the tolerance:"

**read**(\*,\*) tolerance

! Evaluate function at the two limits.

fLow = rootFindingFunction(xLow)

fHigh = rootFindingFunction(xHigh)

! Display the column headers of the output table.

**write**(\*,10) "|", "Method", "|", "Root", "|", "Error", "|", "Iterations", "|"

! Check: Is the lower limit of the initial bracket a root?

**if**(fLow == 0) **then**

root = xLow

error = 0.0

iterations = 0

methodName = "None"

**call** displayRoot(methodName, root, error, iterations)

**return**

**end** **if**

! Check: Is the upper limit of the initial bracket a root?

**if**(fHigh == 0) **then**

root = xHigh

error = 0.0

iterations = 0

methodName = "None"

**call** displayRoot(methodName, root, error, iterations)

**return**

**end** **if**

! Check: Does the root lie within the initial bracket?

**if**(haveOppositeSigns(fLow, fHigh) .**eqv**. .**false**.) **then**

**stop** "Error: Limits don't bracket the root."

**end** **if**

! Get and display the root as estimated by the bisection method.

methodName = "Bisection"

!! call bisectionRoot(xLow, xHigh, tolerance, root, error, iterations)

!! call displayRoot(methodName, root, error, iterations)

! Get and display the root as estimated by the false-position method.

methodName = "False-Position"

!! call falsePositionRoot(xLow, xHigh, tolerance, root, error, iterations)

!! call displayRoot(methodName, root, error, iterations)

! Get and display the root as estimated by the modified false-position method.

methodName = "Modified False-Position"

!! call modifiedFalsePositionRoot(xLow, xHigh, tolerance, root, error, iterations)

!! call displayRoot(methodName, root, error, iterations)

! Get and display the root as estimated by Dekker's method.

methodName = "Dekker"

**call** dekkerRoot(xLow, xHigh, tolerance, root, error, iterations)

**call** displayRoot(methodName, root, error, iterations)

! The format statement for the column headers of the output table.

10 **format**(a3, a25, a3, a10, a3, a10, a3, a12, a3)

**end** **subroutine** mainBracketingMethods

# dekkerRoot.f08

**subroutine** dekkerRoot(xLowInitial, xHighInitial, tolerance, root, error, iterations)

**implicit** **none**

! Declare external functions

**real**, **external** :: rootFindingFunction, bisectionRootOneStep, secantRootOneStep

**logical**, **external** :: haveOppositeSigns

! Declare arguments

!! Input arguments

**real**, **intent**(in) :: xLowInitial, xHighInitial, tolerance

!! Output arguments

**real**, **intent**(out) :: root, error

**integer**, **intent**(out) :: iterations

! Declare local variables

!! xBisection = one limit of the bisection interval {PowerPoint B}

!! xSecant = one limit of the secant interval {PowerPoint S}

!! xRoot = second limit of bisection and secant intervals {PowerPoint R}

!! also, the current estimate for the root

**real** xBisection, xSecant, xRoot

**real** fBisection, fSecant, fRoot

!! bisectionRoot = root estimated using one step of the bisection method {PowerPoint b}

!! bisectionRoot = root estimated using one step of the secant method {PowerPoint s}

**real** bisectionRoot, secantRoot

!! Function values at the initial limits (used to initialize B and R)

**real** fLowInitial, fHighInitial

! Initialize B and R such that f(R) < f(B)

! S = B for the first iteration

fLowInitial = rootFindingFunction(xLowInitial)

fHighInitial = rootFindingFunction(xHighInitial)

**if**(*abs*(fLowInitial) < *abs*(fHighInitial)) **then**

xRoot = xLowInitial

xBisection = xHighInitial

xSecant = xHighInitial

**else**

xRoot = xHighInitial

xBisection = xLowInitial

xSecant = xLowInitial

**end** **if**

fBisection = rootFindingFunction(xBisection)

fSecant = rootFindingFunction(xSecant)

fRoot = rootFindingFunction(xRoot)

! Initialize error as half the length of the bisection interval

error = *abs*(xRoot - xBisection) / 2

iterations = 0

**do** **while**(error > tolerance)

iterations = iterations + 1

! Evaluate roots b and s

bisectionRoot = bisectionRootOneStep(xBisection, xRoot)

secantRoot = secantRootOneStep(xSecant, xRoot)

! Case 1: s lies between b and R

**if**((secantRoot < *max*(xRoot, bisectionRoot)) .**and**. (secantRoot > *min*(xRoot, bisectionRoot))) **then**

**write**(\*,\*) iterations, ": Using the secant method..."

! Move S to R

xSecant = xRoot

fSecant = fRoot

! Move R to s

xRoot = secantRoot

fRoot = rootFindingFunction(xRoot)

**if**(haveOppositeSigns(fBisection, fRoot) .**eqv**. .**false**.) **then**

! Move B to S

xBisection = xSecant

fBisection = fSecant

**end** **if**

! Case 2: s lies outside b and R

**else**

**write**(\*,\*) iterations, ": Using the bisection method..."

! Move S to R

xSecant = xRoot

fSecant = fRoot

! Move R to b

xRoot = bisectionRoot

fRoot = rootFindingFunction(xRoot)

**if**(haveOppositeSigns(fBisection, fRoot) .**eqv**. .**false**.) **then**

! Move B to S

xBisection = xSecant

fBisection = fSecant

**end** **if**

**end** **if**

error = *abs*(xRoot - xBisection) / 2

**end** **do**

root = xRoot

**end** **subroutine** dekkerRoot

# bisectionRootOneStep.f08

**real** **function** bisectionRootOneStep(xLow, xHigh)

! Return midpoint of the input interval [xLow, xHigh] as the estimate for the root using a single iteration of the bisection method.

**implicit** **none**

**real**, **intent**(in) :: xLow, xHigh

bisectionRootOneStep = (xLow + xHigh) / 2

**end** **function** bisectionRootOneStep

# secantRootOneStep.f08

**real** **function** secantRootOneStep(x1, x2)

! Return x-intercept of the secant formed with points x1 and x2 as the estimate for the root using a single iteration of the secant method.

**implicit** **none**

**real**, **external** :: rootFindingFunction

**real**, **intent**(in) :: x1, x2

**real** f1, f2

**real** slope

f1 = rootFindingFunction(x1)

f2 = rootFindingFunction(x2)

slope = (f1 - f2) / (x1 - x2)

secantRootOneStep = x2 - (f2 / slope)

**end** **function** secantRootOneStep

# rootFindingFunction.f08

**real** **function** rootFindingFunction(x)

! Return LHS of equation f(x) = 0

**implicit** **none**

**real**, **intent**(in) :: x

rootFindingFunction = (1 / (x - 3)) - 6

**end** **function** rootFindingFunction

# swapNumbers.f08

**subroutine** swapNumbers(num1, num2)

**implicit** **none**

**real**, **intent**(inout) :: num1, num2

**real** swapper

swapper = num1

num1 = num2

num2 = swapper

**end** **subroutine** swapNumbers

# displayRoot.f08

**subroutine** displayRoot(methodName, root, error, iterations)

**implicit** **none**

**character**(*len*=25), **intent**(in) :: methodName

**real**, **intent**(in) :: root, error

**integer**, **intent**(in) :: iterations

**write**(\*,10) "|", methodName, "|", root, "|", error, "|", iterations, "|"

10 **format**(a3, a25, a3, f10.4, a3, f10.4, a3, i12, a3)

**end** **subroutine** displayRoot

# Output

## Output 1 [(1 / (x - 3)) – 6]

Enter lower limit of the initial interval:

3.1

Enter upper limit of the initial interval:

4

1 : Using the secant method...

2 : Using the secant method...

3 : Using the bisection method...

4 : Using the bisection method...

5 : Using the secant method...

6 : Using the secant method...

7 : Using the secant method...

8 : Using the secant method...

9 : Using the bisection method...

10 : Using the bisection method...

|Dekker's Method | 3.1667 | 0.00000083 | 10 |