• Recall that we said there are a couple of approaches we could take in turning our separate codes into subprograms.

  – We could have the subprogram for each method return $x^{k+1}$, the next iterate. The input for each subprogram for each method would be different but output the same.

  – We could have each subprogram return the converged value or indicate that convergence failed. This would mean that in each subprogram we would loop over the iterations, producing a sequence of iterates and checking for convergence.

• We will choose the first approach and have our main driver routine have the loop over the number of iterations; inside this loop it will call the appropriate subprogram and the check for convergence.

• This way we will only have to program these statements once. All the function for the methods will do is generate the next iterate.
• The main driver program will contain the function defining the nonlinear equation and its derivative.

• Constructing the code in this manner allows us to write and debug the subprograms in the module and then never have to change the module when we change the nonlinear equation or the convergence tolerance.

• As we learn new methods, we can simply write a subprogram to add to the module which advances the iteration one step.

• We will use the case construct in the main or driver program to read in the appropriate input, call the appropriate subprogram, and get ready for the next iteration.
A Subprogram for Newton’s Method

• For input we need only the previous guess, say $x_k$, and the value of the function and its derivative at this point.

• For output we will determine the new iterate by our formula

$$x^{k+1} = x^k - \frac{f(x^k)}{f'(x^k)}$$

• Since we have only one output, the new iterate, we can write either a function or a subroutine. Recall that if we use a function the result will be returned in the function name unless we use the result specifier.

• Note that our formula for $x^{k+1}$ requires division by $f'(x^k)$. If this derivative is zero, or near zero, then we will get a run time error message. Consequently we will add a statement to check this against machine epsilon. This is the largest number such that when we add it to one we still get one. It is an intrinsic function in fortran given by \texttt{epsilon(x)}.
• We assume that our module uses our `common_data` module to set precision.
• We assume the module contains the `implicit none` statement.
function newton ( xk , f_at_xk, fprime_at_xk ) result(xkp1)
real(prec) :: f_at_xk, fprime_at_xk
real(prec) ::xkp1
real(prec) :: xk
real(DP) ::  x
if ( fprime_at_xk < epsilon(x) ) then
    print *, "derivative is too small, Newton fails"
    stop
end if
xkp1 = xk - f_at_xk / fprime_at_xk
end function newton

with the calling statement
\[ xkp1 = \text{newton}(xk, \text{f\_at\_xk}, \text{fprime\_at\_xk}) \]

where, of course, \( f\_at\_xk, f\_at\_xkp1 \) have been defined as

\[ f\_at\_xk = fx(xk) \]

\[ fprime\_at\_xk = fprime(xk) \]

with the functions \( fx \) for \( f(x) \) and \( fprime \) for \( f'(x) \).
A Function for the Secant Method

- For the function routine for the Secant Method, we simply need to input 2 values (say \( x_k \), \( x_{k-1} \)) and their corresponding function values and compute the new iterate from our formula

\[
x^{k+1} = x^k - \frac{f(x^k)(x^k - x^{k-1})}{f(x^k) - f(x^{k-1})}
\]

- The function statement might look like

```fortran
function secant ( xk, xkm1, f_at_xk, f_at_xkm1)
with the result returned in secant or with the result returned in xkp1
function secant ( xk, xkm1, f_at_xk, f_at_xkm1) result (xkp1)
with a calling statement
xkp1 = secant( xk, xkm1, f_at_xk, f_at_xkm1)
```
with \( f_{at\_xk}, f_{at\_xkm1} \) defined.

- Recall that the order of the arguments is up to you but they must match with the order (and number) in the calling statement for the function.

- We can simply code the above formula and the new iterate will be returned in the name of the function, \( \text{secant} \) or in \( xkp1 \) if we use the alternate form.

- Note that in this formula we are dividing by \( f(x^k) - f(x^{k-1}) \). We are looking for \( x^k \) such that \( f(x^k) \rightarrow 0 \). Usually we won’t have to check that this denominator is near zero, because we have a check on convergence for \( f(x^k) \).

- You will be asked to write a function for the secant method in Project II.
If our strategy is to just to have our function routine provide us with the next iterate, then it’s simple to write a function routine for the Bisection Method or the Regula Falsi Method.

For the Bisection Method we simply need the endpoints of the current interval (which can still be thought of \(x_{km1}, x_k\) - here \(x_{km1}\) plays the role of \(a\) and \(x_k\) as \(b\) - as in the Secant Method but for bisection \(x^{k-1} < x^k\)) and compute the next iterate as the midpoint of this interval.

The function could simply look like

```fortran
function bisection ( a, b )
real(prec) :: a, b
real(prec) :: bisection
bisection = ( a + b ) / two
end function bisection
```
with the calling statement

\[ x_{k+1} = \text{bisection} (x_{k-1}, x_k) \]

- To implement the equation for the Regula Falsi we need the two endpoints and their function values. So the calling statement might look like

\[ x_{k+1} = \text{regula_falsi} (x_{k-1}, x_k, f_{\text{at}_{x_{k-1}}}, f_{\text{at}_{x_k}}) \]

and we simply code the formula we derived for the next iterate in regula falsi.

- However, when we get ready for the next iteration these two methods are a bit more complicated than secant or newton. Recall that to get ready for the next iteration for Newton’s Method we just set \( x_k = x_{k+1} \) and for the Secant Method we set \( x_{k-1} = x_k \) and \( x_k = x_{k+1} \).

- But recall for the Bisection and Regula Falsi we have to decide if the root is in (using this notation) \([x_{k-1}, x_k+1] \) or \([x_k+1, x_k] \). We already did this in our main program \text{Bisection.f90}.

- We can write a subprogram for this and include it in the module for the nonlinear functions.
• Since we need to output the two ends of the interval (and their function values) we need to write a subroutine instead of a function.

• The subroutine statement might look like

```plaintext
subroutine find_interval (a, b, c, f_at_a, f_at_b, f_at_c)
where we pass in \(x^{k-1}\) for \(a\), \(x^k\) for \(b\) and the new iterate \(x^{k+1}\) for \(c\) since it lies in the interval \([a, b]\).
```

• For our check we can determine the sign of \(f_{at\_a} \times f_{at\_c}\) and then either overwrite \(a\) or \(b\).

• We might as well overwrite the corresponding function values as well to save ourselves another function evaluation.
subroutine find_interval (a, b, c, f_at_a, f_at_b, f_at_c)
declarations (see code)
if (f_at_a * f_at_c == zero ) then
    print *, "Root has been found at ", c
    stop
else if (f_at_a * f_at_c < zero) then! root in (a,c)
    b = c; f_at_b = f_at_c
else
    a = c; f_at_a = f_at_c ! root in (c,b)
end if
end subroutine find_interval

- The module nonlinear_solvers in the file nonlinear_solvers.f90 contains functions for Newton’s Method and the Bisection Method as well as this subroutine. You will be adding functions for the other methods.
module nonlinear_solvers

use common_data

implicit none

contains

function newton ( xk , f_at_xk, fprime_at_xk )
:
function secant ( xk, xkm1, f_at_xk, f_at_xkm1)
:
function bisection ( a, b )
function regula_falsi ( a, b, f_at_a, f_at_b )
:
subroutine find_interval (a, b, c, f_at_a, f_at_b, f_at_c)
:
end module nonlinear_solvers
The Main or Driver Program

• In the driver program the user will be asked to indicate which method he/she wants to use. We can use a flag for this. For example, if the flag is 2 we use the Secant Method, if it is 1 we use Newton’s Method, etc. Alternately, we can use the name of the method as the selector.

• As we have seen, Fortran 90 provides a case construct to easily handle this. Of course we could also use conditionals.

• The main program will contain a function for \( f(x) \), where we are solving the nonlinear equation \( f(x) = 0 \) and for its derivative (if needed).

• The main program will have our loop over the number of iterations where we call the appropriate subprogram, check for convergence, and print the results and get ready for next iteration.

• Of course it will also contain a use nonlinear_solvers statement to give it access to the subprograms in our module as well as the usual use common_data.
Structure for Driver Program

• Set tolerance and maximum number of iterations. Ultimate, we will allow the maximum number of iterations to be different for different methods but for now we only have one set.

• Query the user to determine which method is to be used.
  – If Newton’s method used, read in one starting value.
  – If Secant method used, read in two starting values.
  – If the Bisection Method or Regula Falsi is used we read in two starting values which must bracket the root.
  – Evaluate the function at these initial points.
  – Set the max number of iterations for specified method.
• Loop over number of iterations.
  – For Newton’s Method
    * evaluate $f'(x^k)$
    * Call appropriate function to get $x^{k+1}$
  – For Secant Method, bisection or regula falsi
    * Call appropriate function to get $x^{k+1}$
• Check for convergence.
• If converged, print out results.
• If not converged, get ready for next iteration;
  – For Newton’s method set $x_k = x_{k+1}$
  – For secant method set $x_{k-1} = x_k$, $x_k = x_{k+1}$
  – For the Bisection Method or Regula Falsi call the subroutine `find_interval`
• In the same file we will include our function for $f(x)$ and its derivative (if needed).
A sample driver program can be found in `driver_nonlinear.f90` which currently just has the option to call Newton’s Method or the Bisection Method.

We will look at the code in sections. The first section is to read in the initial values.

```fortran
print *, "What nonlinear solver do you want to use?"
p
print *, "Enter 1 for Newton’s Method ", &
"Enter 3 for Bisection Method 
read *, flag_method

select case ( flag_method )
  case ( 1 )
    print *, "Solving f(x)=0 using Newton Method"
    print *, " Enter an initial guess 
    read *, xk
    f_at_xk = fx (xk)

  case ( 3)
```
print *, "Solving f(x)=0 using Bisection Method"
print *, " Enter endpoints of an interval bracketing root (separated by comma) "
read *, xkm1, xk
f_at_xk = fx (xk)
f_at_xkm1 = fx(xkm1)

if ( f_at_xk * f_at_xkm1 > 0 ) then
    print *, " initial interval does not bracket the root"
    stop
end if

end select
• The next section is the portion of the iteration loop where we call the appropriate function to get \( x_{kp1} \).

\[
\text{do } n = 1, \text{ max\_iterations}
\]

\[
\begin{align*}
\text{select case ( flag\_method) } \\
\text{ case (1) } ! \text{ Newton's Method} \\
& \quad f\text{prime\_at\_xk} = f\text{prime} ( x_k ) \\
& \quad x_{kp1} = \text{newton} ( x_k, f\text{\_at\_xk}, f\text{prime\_at\_xk} ) \\
\text{ case (3) } ! \text{ Bisection Method} \\
& \quad x_{kp1} = \text{bisection} ( x_{km1}, x_k )
\end{align*}
\]

\[
\text{end select}
\]

\[
\text{f\_at\_xkp1} = \text{fx} ( x_{kp1} )
\]

\[
\text{print *, "At iteration number ", n, "the approximation is ", xkp1}
\]
The last section of the code is checking for convergence and getting ready for the next iteration.

```Fortran
if ( abs ( xkp1 - xk ) / abs (xkp1) <= tolerance ) then
    print *, " method has converged in ", n, " iterations ",
    stop
else  ! method not converged so get ready for next iteration

    select case ( flag_method )
    case (1)
        xk = xkp1
        f_at_xk = f_at_xkp1
    case (3 )
        call find_interval (xkm1, xk, xkp1, f_at_xkm1, f_at_xk, f_at_xkp1)
    end select

end if
```
• At the end of class you can compile the module for the nonlinear solvers in the file `nonlinear_solvers.f90`.

• Then compile and run the driver program `driver_nonlinear.f90`; be sure to include the object file `nonlinear_solvers.o` on the command line. Try both Newton’s Method and the Bisection Method; the current function in the code is $x^2 - 5$. 
Another Method for Solving $f(x) = 0$

- If we have written our programs intelligently, then it should be relatively easy to implement a new method for finding a root of $f(x) = 0$.

- We consider a method called fixed point iteration and see how we would have to change our program to give the option to implement it.

- A fixed point of the iteration $x^{k+1} = g(x^k)$ is a point $x^*$ such that $x^* = g(x^*)$.

- Now what does this have to do with finding a root of $f(x)$?
  - Oftentimes the problem of finding a root of $f(x)$ can be recast into a fixed point problem.
  - For example, finding $x$ such that $f(x) = 0$ is the same as finding $x$ such that $x = x + f(x) = g(x)$ which can, for appropriate $g(x)$ be approximated by the iteration $x^{k+1} = g(x^k)$. There are other ways for finding $g(x)$.  

– For example, if \( f(x) = x^2 - x - 2 = 0 \) we could have
\[
* \quad g(x) = x^2 - 2 \text{ since } x^2 - x - 2 = 0 \implies x = x^2 - 2.
\]
\[
* \quad g(x) = \sqrt{x + 2} \text{ since } x^2 - x - 2 = 0 \implies x^2 = x + 2.
\]
\[
* \quad g(x) = 1 + \frac{2}{x} \text{ since } x^2 - x - 2 = 0 \implies x - 1 - 2/x = 0 \implies x = 1 + 2/x.
\]

– One can determine analytically the conditions on \( g(x) \) which will guarantee convergence of the fixed point iteration. It can be shown that if \( g'(x^*) > 1 \) then the iteration diverges; so we need \( g'(x^*) < 1 \) if there is any hope of the series converging. As in other nonlinear methods, the choice of the initial guess is critical.

– In the following figure you can see an example of a convergent sequence for finding the fixed point of \( g(x) = 1 + 2/x \). Here we have chosen \( x = 1 \) as the initial guess. The red lines are \( y = x \) and \( y = g(x) \).
What do we need to implement fixed point iteration?

- We need to write a routine which advances the iteration, i.e., finds \( x^{k+1} \) from the previous iterates. Since we are using the iteration \( x^{k+1} = g(x^k) \) all we need is \( g(x) \) evaluated at \( x^k \). The function \( g(x) \) will have to be given to us and we will have to write a function subprogram for it just like we did for \( f(x) \) and \( f'(x) \). Since finding a fixed point of \( x = g(x) \) is equivalent to finding a root of \( f(x) = 0 \), then we can still use our same checks for convergence.
Rates of Convergence

• We now have 5 different methods for finding a root of \( f(x) = 0 \). From our experience we have seen that when convergent, Newton’s method and the secant method take fewer iterations so we say that they converge faster.

• How can we quantify the speed of convergence?

• If the error at the \( k \)th step is given by \( e^k \) then we say the method converges (theoretically) at a rate \( r \) given by

\[
\lim_{k \to \infty} \frac{|e^{k+1}|}{|e^k|r} = C'
\]

for a constant \( C' \).

• For example, if \( r = 1 \) this says that asymptotically (i.e., as \( k \to \infty \))

\[
|e^{k+1}| = C'|e^k|
\]

or in words, that the error at the \( k+1 \) iteration is approximately a constant (\(< 1\)) times the error at the previous iteration. When \( r = 1 \) we say the rate of
convergence is linear.

- If $r = 2$ this says that asymptotically (i.e., as $k \to \infty$)
  \[ |e^{k+1}| = C|e^k|^2 \]
  or in words, that the error at the $k+1$ iteration is approximately a constant times the square of the error at the previous iteration. So if the error at one step is $10^{-2}$ then the error at the next step is a constant times $10^{-4}$. When $r = 2$ we say the theoretical rate is quadratic.

- Clearly the higher $r$ is, the faster it will converge.

- What is the numerical rate of convergence? This is the rate of convergence we are seeing in our numerical calculations. As $k \to \infty$ this rate should approach the theoretical rate. Note that the only way we can determine this if to solve the problem using a nonlinear function whose exact roots are known.
• How do we determine the numerical rate of convergence?

• Suppose we have calculated three approximations $x^k$, $x^{k+1}$, and $x^{k+2}$ and their corresponding errors $|e^k|$, $|e^{k+1}|$, and $|e^{k+2}|$. Then we have

$$|e^{k+1}| \approx C|e^k|r \quad |e^{k+2}| \approx C|e^{k+1}|r$$

Solving for $C$ (and using “=” in the above relations) we get

$$C = \frac{|e^{k+1}|}{|e^k|r} = \frac{|e^{k+2}|}{|e^{k+1}|r}$$

Simplifying the above expression so that we can solve for $r$ we get

$$\frac{|e^{k+1}|}{|e^{k+2}|} = \frac{|e^k|r}{|e^{k+1}|r} = \left(\frac{|e^k|}{|e^{k+1}|}\right)^r$$

and taking the natural log of both sides and using the properties of ln

$$\ln \left(\frac{|e^{k+1}|}{|e^{k+2}|}\right) = r \ln \left(\frac{|e^k|}{|e^{k+1}|}\right)$$

Thus the numerical rate of convergence is given by

$$r = \ln \left|\left(\frac{e^{k+1}}{e^{k+2}}\right)\right| / \ln \left|\left(\frac{e^k}{e^{k+1}}\right)\right|$$
Below is a table of results from using Newton’s Method for finding the root $x = 1$ of the function $f(x) = (x - 1)(x^4 - 2x + 4)$. Note that the numerical rate of approximation is approaching 2 as $k \to \infty$. It can be show theoretically that under certain conditions, Newton’s method is quadratically convergent. In each case, the numerical rate is calculated by using the errors at that iteration and the previous two.

| $k$ | $x^k$  | $|1 - x^k|$ | numerical rate |
|-----|--------|-------------|----------------|
| 0   | 2.0    | 1.0         |                |
| 1   | 1.65217391304348 | 0.65217391304348 | 0.65217391304348 |
| 2   | 1.36663567397384  | 0.36663567397384  | 1.34           |
| 3   | 1.14212179445774  | 0.14212179445774  | 1.65           |
| 4   | 1.02009352623064  | 0.02009352623064  | 2.06           |
| 5   | 1.00029368161975  | 0.00029368161975  | 2.16           |
| 6   | 1.000000005757804 | 0.000000005757804 | 2.03           |
Classwork

• Write a short code to read in three errors and compute the numerical rate of convergence. Make sure your code is working by using the errors generated in the table above for Newton’s method.

• Write a function called `regula_falsi` to find $x^{k+1}$ given $x^k$ and $x^{k-1}$ and their corresponding function values. Here $[x^{k-1}, x^k]$ is an interval bracketing the root.

• Modify the driver program to include the option to use the regula falsi routine that you wrote. Use a flag of 4 for this method. Remember in the `case` construct you can use syntax such as `case (3, 4)` if you want the statements to be executed for both cases 3 and 4.

• Note that this is required for Project II so you can just turn it in then.
• See Project II