How to analyze the convergence rate of a nonlinear solver

This *Mathematica* notebook solves a nonlinear function f[x]=0 using three methods:

(a) Built-in solver

(b) Explicitly programmed classical Newton-Raphson solver

(c) Explicitly programmed Modified Newton-Raphson solver

In[1]:=
```
SetDirectory[NotebookDirectory[]]
```

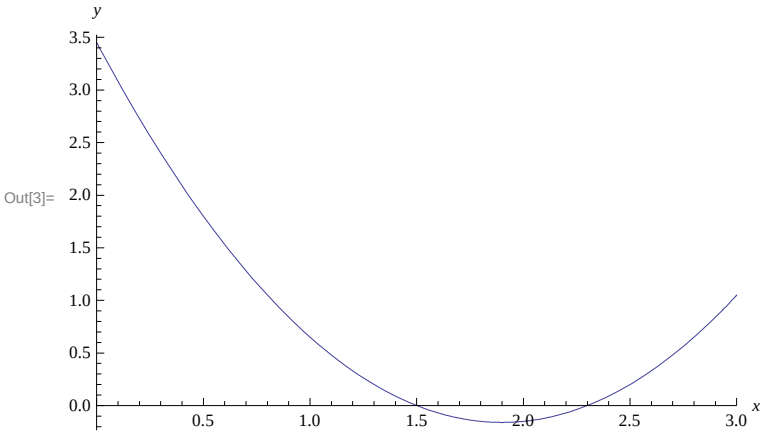
Out[1]=
```
C:\Users\Cobbie\Dropbox\ME7960_Comp_Constitutive_Modeling\Homework
```

Case study

Consider

In[2]:=
```
f[x_] := (x - 1.5) (x - 2.3)
Plot[f[x], {x, 0, 3}, PlotRange → {{0, 3}, All},
 AxesLabel → {x, y}, PlotLabel → Row[{"y=", Expand[f[x]]}]]
```

Out[3]=



By design, the exact solutions are x=1.5 and x=2.3

In[4]:=
```
xexact = x /. Solve[f[x] == 0, x]
```

Out[4]=
```
{1.5, 2.3}
```

■ Convergence analysis for the built-in nonlinear solver

The following applys the built-in solver using a starting guess of x=0.

The Reap and Sow stuff is how *Mathematica* lets us see the intermediate results, rather than just the final result.

In[5]:=
```
{res, {xlist}} = Reap[FindRoot[f[x] == 0, {x, 0},
     AccuracyGoal → Infinity, PrecisionGoal → 2000, EvaluationMonitor :→ Sow[x]]];
```

The res output is the solution and the xlist array contains the successively improved guesses

In[6]:=
```
res
```

Out[6]=
```
{x → 1.5}
```

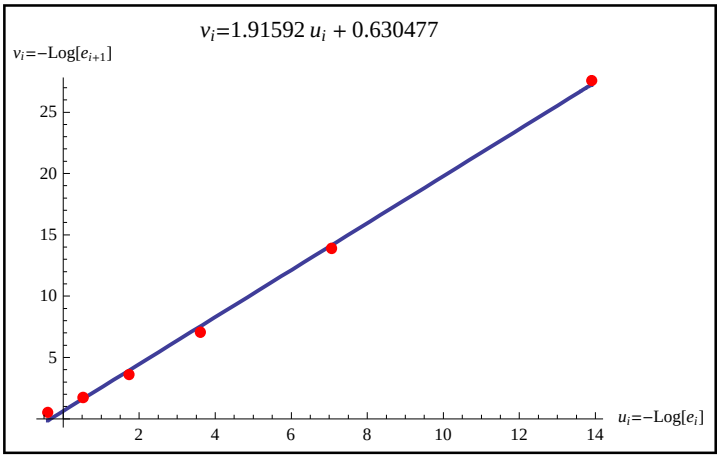
In[7]:=
```
xlist
```

Out[7]=
```
{0., 0.907895, 1.32331, 1.47293, 1.49914, 1.5, 1.5, 1.5}
```

The following table goes from i=1 to Length[xlist]-1 to drop the last error (since it typically bucks the convergence trend by essentially repeating the result right before it)

In[8]:=
```
(* Select which of the two exact roots has been found *)
iexact = 1;
(* Create a table of errors in the x values at each iteration,
but don't include the last one since Mathematica's final value typically
 bucks the convergence trends, probably from some unknown "clean up" step *)
ex = Table[Abs[xlist[[i]] - xexact[[iexact]]], {i, Length[xlist] - 1}];
(* Create a list of u_i=-Log[e_i] values *)
u = Table[-Log[ex[[i]]], {i, Length[ex] - 1}];
(* Create a list of v_i=-Log[e_(i+1)] values *)
v = Table[-Log[ex[[i + 1]]], {i, Length[ex] - 1}];
(* Combine the lists into (u_i,v_i) data pairs *)
uvdat = Transpose[{u, v}];
(* Perform a linear fit to the data*)
model = LinearModelFit[uvdat, ufit, ufit];
(* Create a plot showing data and the fit *)
Show[
  Plot[Normal[model], {ufit, Min[u], Max[u]}, PlotStyle → Thick],
  ListPlot[uvdat, PlotStyle → {PointSize[0.02], Red}],
  AxesLabel → {"u_i=-Log[e_i]", "v_i=-Log[e_(i+1)]"},
  PlotLabel → Row[{"v_i=", Normal[model] /. ufit → "u_i"}]] // Framed
```

Out[14]=



Based on the plot label, the slope of the fitted line is 1.9, indicating that the built-in solver is approximately second-order accurate.

■ Convergence analysis for the hand-coded “dumb” CNR solver

The following is a brute force implmentation that has no protections against division by zero from a bad first guess, nor any scaling to ensure that built-in tolerance is reasonable. Constitutive models are a good example of the danger of forgetting to scale. For a constitutive model, the stress is a function of strain. Stress typically has values on the order of 10^9, which would make the tolerance of 10^{-8} in the following code overly agressive, possibly leading to an infinite loop of trying to find a solution that can’t be found with such tight tolerances. On the other hand, if this solver

were to be applied to a problem where f[x] gives a strain value, then it might converge prematurely since strain values are often very tiny numbers. Lesson: know the limitations of the numerical solvers, and perform scaling or other protective actions accordingly.

In[15]:=
```
dumbCNRsolver[f_, xguess_] := Module[{xlist, tolerance = 1.0 × 10^-8, x},
  xlist = {xguess};
  x = xguess;
  While[Abs[f[x]] > tolerance,
   x = x - f[x]/f'[x];
   xlist = Append[xlist, x];
  ];
  Return[xlist]
 ]
```

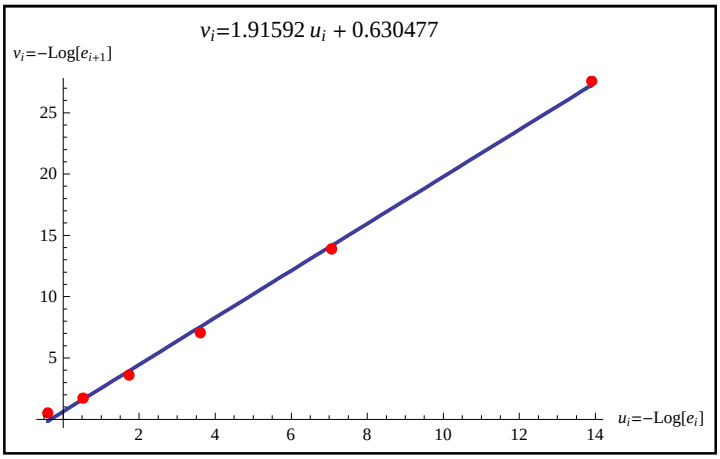
Apply the solver to get a list of the x values for each iteration:

In[16]:=
```
xlist = dumbCNRsolver[f, 0.0]
```

Out[16]=
```
{0., 0.907895, 1.32331, 1.47293, 1.49914, 1.5, 1.5}
```

In[17]:=
```
(* Select which of the two exact roots has been found *)
iexact = 1;
(* Create a table of errors in the x values at each iteration *)
ex = Table[Abs[xlist[[i]] - xexact[[iexact]]], {i, Length[xlist]}];
(* Create a list of u_i=-Log[e_i] *)
u = Table[-Log[ex[[i]]], {i, Length[ex] - 1}];
(* Create a list of v_i=-Log[e_i+1] *)
v = Table[-Log[ex[[i + 1]]], {i, Length[ex] - 1}];
(* Combine the lists into (u_i,v_i) data pairs *)
uvdat = Transpose[{u, v}];
(* Perform a linear fit to the data*)
model = LinearModelFit[uvdat, ufit, ufit];
(* Create a plot showing data and the fit *)
Show[
  Plot[Normal[model], {ufit, Min[u], Max[u]}, PlotStyle → Thick],
  ListPlot[uvdat, PlotStyle → {PointSize[0.02], Red}],
  AxesLabel → {"u_i=-Log[e_i]", "v_i=-Log[e_i+1]"},
  PlotLabel → Row[{"v_i=", Normal[model] /. ufit → "u_i"}]] // Framed
```

Out[23]=



In[24]:=
```
Export["ClassicalNewtonRaphsonConvergencePlot.png", %]
```

Out[24]=
```
ClassicalNewtonRaphsonConvergencePlot.png
```

Since the slope is approximatly 2, his Classical N-R solver has an approximately second-order rate of convergence.

The dumbCNRsolver has exactly the same results as the built-in solver, which is no surprise since the *Mathematica* documentation indicates that the built-in solver is a classical NR solver. By having written one ourselves, we have conducted a prudent verification test that *Mathematica*'s algorithms are indeed working as advertised in their documentation!

■ Convergence analysis for the hand-coded modified NR solver

The modified NR solver is the same except that the slope is not re-evaluated at each iteration. Just the slope at the first guess is used.

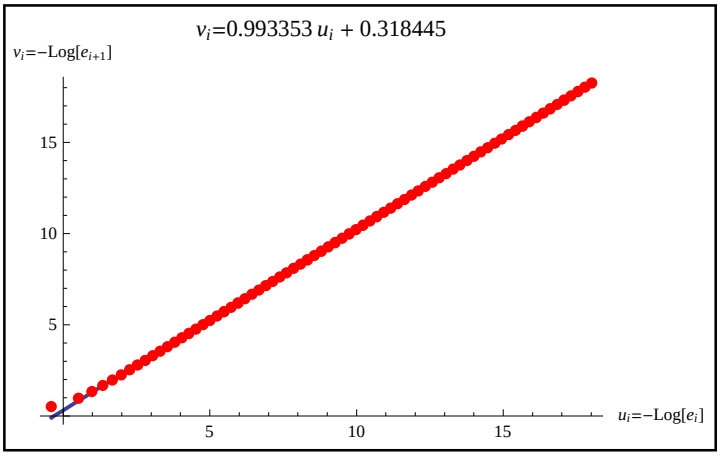
In[25]:=
```
dumbMNRsolver[f_, xguess_] := Module[{xlist, tolerance = 1.0 × 10^-8, x, fprime},
  xlist = {xguess};
  x = xguess;
  fprime = f'[xguess];
  While[Abs[f[x]] > tolerance,
   x = x - f[x]/fprime;
   xlist = Append[xlist, x];
  ];
  Return[xlist]
 ]
```

In[26]:=
```
xlist = dumbMNRsolver[f, 0.0]
```

Out[26]=
```
{0., 0.907895, 1.12481, 1.24084, 1.31308, 1.36162, 1.39579, 1.42059, 1.43897,
 1.4528, 1.46332, 1.4714, 1.47763, 1.48247, 1.48624, 1.48919, 1.4915, 1.49331,
 1.49473, 1.49584, 1.49672, 1.49742, 1.49796, 1.49839, 1.49873, 1.499, 1.49921,
 1.49938, 1.49951, 1.49961, 1.49969, 1.49976, 1.49981, 1.49985, 1.49988, 1.49991,
 1.49993, 1.49994, 1.49995, 1.49996, 1.49997, 1.49998, 1.49998, 1.49999, 1.49999,
 1.49999, 1.49999, 1.49999, 1.5, 1.5, 1.5, 1.5, 1.5, 1.5, 1.5, 1.5, 1.5, 1.5,
 1.5, 1.5, 1.5, 1.5, 1.5, 1.5, 1.5, 1.5, 1.5, 1.5, 1.5, 1.5, 1.5, 1.5, 1.5, 1.5}
```

In[27]:=
```
(* Select which of the two exact roots has been found *)
iexact = 1;
(* Create a table of errors in the x values at each iteration *)
ex = Table[Abs[xlist[[i]] - xexact[[iexact]]], {i, Length[xlist]}];
(* Create a list of u_i=-Log[e_i] *)
u = Table[-Log[ex[[i]]], {i, Length[ex] - 1}];
(* Create a list of v_i=-Log[e_i+1] *)
v = Table[-Log[ex[[i + 1]]], {i, Length[ex] - 1}];
(* Combine the lists into (u_i,v_i) data pairs *)
uvdat = Transpose[{u, v}];
(* Perform a linear fit to the data*)
model = LinearModelFit[uvdat, ufit, ufit];
(* Create a plot showing data and the fit *)
Show[
  Plot[Normal[model], {ufit, Min[u], Max[u]}, PlotStyle → Thick],
  ListPlot[uvdat, PlotStyle → {PointSize[0.02], Red}],
  AxesLabel → {"u_i=-Log[e_i]", "v_i=-Log[e_i+1]"},
  PlotLabel → Row[{"v_i=", Normal[model] /. ufit → "u_i"}]] // Framed
```

Out[33]=



In[34]:=
```
Export["ModifiedNewtonRaphsonConvergencePlot.png", %]
```

Out[34]= ModifiedNewtonRaphsonConvergencePlot.png

This modified NR has a slope of about 1.0 in its convergence plot, so it is approximately first-order.

COMMENTARY: This solver required many more function evaluations and many more steps to converge, but that alone does not imply that it is necessarily more expensive computationally. This solver could be the least expensive choice for functions that are so complicated that their derivative is highly expensive. An example from constitutive modeling might be a function that outputs a Hencky strain tensor as a function of, say, a controlled component of stress. Evaluating the derivative of Hencky strain is extremely expensive, so the Modified NR solver might be cheaper despite needing more evaluations.

■ Convergence analysis for a hand-coded secant solver

The following approximates the slope using secants, which is useful when it isn't possible to evaluate derivatives

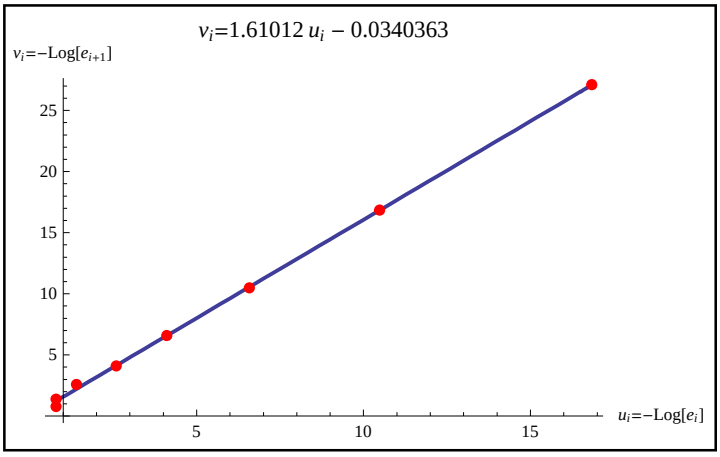
In[35]:=
```
dumbSECANTsolver[f_, xguess1_, xguess2_] :=
 Module[{xlist, tolerance = 1.0 × 10^-8, x1, x2, f1, f2, fprime},
  xlist = {x1};
  x1 = xguess1;
  x2 = xguess2;
  While[Abs[f[x1]] > tolerance,
   f1 = f[x1];
   f2 = f[x2];
   fprime = (f2 - f1)/(x2 - x1);
   x2 = x1;
   x1 = x1 - f1/fprime;
   xlist = Append[xlist, x1];
  ];
  Return[xlist]
 ]
```

In[36]:=
```
xlist = dumbSECANTsolver[f, 0.0, 0.5]
```

Out[36]=
```
{1.04545, 1.04545, 1.25248, 1.4251, 1.48348, 1.49861, 1.49997, 1.5, 1.5}
```

In[37]:=
```
(* Select which of the two exact roots has been found *)
iexact = 1;
(* Create a table of errors in the x values at each iteration *)
ex = Table[Abs[xlist[[i]] - xexact[[iexact]]], {i, Length[xlist]}];
(* Create a list of u_i=-Log[e_i] *)
u = Table[-Log[ex[[i]]], {i, Length[ex] - 1}];
(* Create a list of v_i=-Log[e_i+1] *)
v = Table[-Log[ex[[i + 1]]], {i, Length[ex] - 1}];
(* Combine the lists into (u_i,v_i) data pairs *)
uvdat = Transpose[{u, v}];
(* Perform a linear fit to the data*)
model = LinearModelFit[uvdat, ufit, ufit];
(* Create a plot showing data and the fit *)
Show[
  Plot[Normal[model], {ufit, Min[u], Max[u]}, PlotStyle → Thick],
  ListPlot[uvdat, PlotStyle → {PointSize[0.02], Red}],
  AxesLabel → {"u_i=-Log[e_i]", "v_i=-Log[e_i+1]"},
  PlotLabel → Row[{"v_i=", Normal[model] /. ufit → "u_i"}]] // Framed
```

Out[43]=



In[44]:=
```
Export["SecantSolverConvergencePlot.png", %]
```

Out[44]=
```
SecantSolverConvergencePlot.png
```

With a convergence plot slope of 1.6, this secant solver is somewhere between 1st and 2nd order accurate.