CHAPTER 6

6.1 The function can be formulated as a fixed-point iteration as

$$x_{i+1} = 2\sin\left(\sqrt{x_i}\right)$$

Using an initial guess of $x_0 = 0.5$, the first iteration is

$$x_1 = 2\sin\left(\sqrt{0.5}\right) = 1.299274$$

$$\varepsilon_a = \left|\frac{1.299274 - 0.5}{1.299274}\right| \times 100\% = 61.517\%$$

The remaining iterations are summarized below. As can be seen, 7 iterations are required to drop below the specified stopping criterion of 0.01%.

i	*x*	ε_a	ratio
0	0.5		
1	1.299274	61.517%	
2	1.817148	28.499%	0.46327
3	1.950574	6.840%	0.24002
4	1.969743	0.973%	0.14227
5	1.972069	0.118%	0.12122
6	1.972344	0.014%	0.11832
7	1.972377	0.002%	0.11797

The table also includes a column showing the ratio of the relative errors between iterations:

$$\text{ratio} = \frac{\varepsilon_{a,i}}{\varepsilon_{a,i-1}}$$

As can be seen, after the first few iterations this ratio is converging on a constant value of about 0.118. Recall that the error of fixed-point iteration is

$$E_{t,i+1} = g'(\xi)E_{t,i}$$

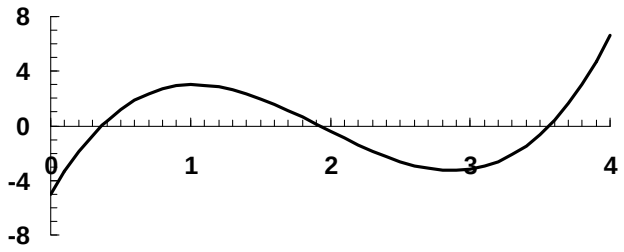
For our problem

$$g'(x) = \frac{d}{dx}2\sin\left(\sqrt{x}\right) = \frac{1}{\sqrt{x}}\cos\left(\sqrt{x}\right)$$

The value of this quantity in the vicinity of the true root (1.9724) agrees with the ratio obtained in the table confirming that the convergence is linear and conforms to the theory.

$$g'(1.9724) = \frac{1}{\sqrt{1.9724}}\cos\left(\sqrt{1.9724}\right) = 0.1179$$

6.2 (a) Graphical



Root ≈ 3.58

(b) Fixed point

The equation can be solved in numerous ways. A simple way that converges is to solve for the x that is not raised to a power to yield

$$x = \frac{5 - 2x^3 + 11.7x^2}{17.7}$$

The resulting iterations are

i	x_i	ε_a
0	**3**	
1	**3.180791**	5.68%
2	**3.333959**	4.59%
3	**3.442543**	3.15%

(c) Newton-Raphson

i	x_i	$f(x)$	$f'(x)$	ε_a
0	**3**	-3.2	1.5	
1	**5.133333**	48.09007	55.68667	41.56%
2	**4.26975**	12.95624	27.17244	20.23%
3	**3.792934**			12.57%

(d) Secant

i	x_{i-1}	$f(x_{i-1})$	x_i	$f(x_i)$	ε_a
0	3	-3.2	**4**	6.6	
1	4	6.6	**3.326531**	-1.9688531	20.25%
2	3.326531	-1.96885	**3.481273**	-0.7959153	4.44%
3	3.481273	-0.79592	**3.586275**	0.2478695	2.93%

(e) Modified secant ($\delta = 0.01$)

i	x	$f(x)$	dx	$x+dx$	$f(x+dx)$	$f'(x)$	ε_a
0	**3**	-3.2	0.03	3.03	-3.14928	1.6908	
1	**4.892595**	35.7632	0.048926	4.9415212	38.09731	47.7068	38.68%
2	**4.142949**	9.73047	0.041429	4.1843789	10.7367	24.28771	18.09%
3	**3.742316**	2.203063	0.037423	3.7797391	2.748117	14.56462	10.71%

6.3 (a) Fixed point. The equation can be solved in two ways. The way that converges is

$$x_{i+1} = \sqrt{1.8x_i + 2.5}$$

The resulting iterations are

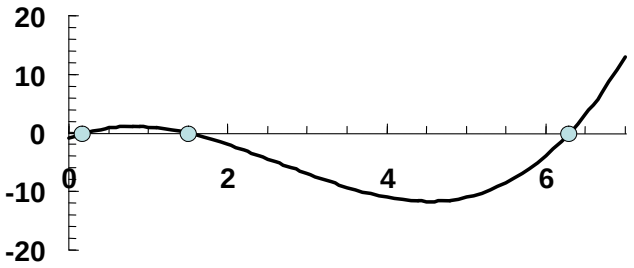
i	x_i	ε_a
0	**5**	
1	**3.391165**	47.44%
2	**2.933274**	15.61%
3	**2.789246**	5.16%
4	**2.742379**	1.71%
5	**2.726955**	0.57%
6	**2.721859**	0.19%
7	**2.720174**	0.06%
8	**2.719616**	0.02%

(b) Newton-Raphson

$$x_{i+1} = x_i - \frac{-x_i^2 + 1.8x_i + 2.5}{-2x_i + 1.8}$$

i	x_i	$f(x)$	$f'(x)$	ε_a
0	5	-13.5	-8.2	
1	3.353659	-2.71044	-4.90732	49.09%
2	2.801332	-0.30506	-3.80266	19.72%
3	2.721108	-0.00644	-3.64222	2.95%
4	2.719341	-3.1E-06	-3.63868	0.06%
5	2.719341	-7.4E-13	-3.63868	0.00%

6.4 (a) A graph of the function indicates that there are 3 real roots at approximately 0.2, 1.5 and 6.3.



(b) The Newton-Raphson method can be set up as

$$x_{i+1} = x_i - \frac{-1 + 5.5x_i - 4x_i^2 + 0.5x_i^3}{5.5 - 8x_i + 1.5x_i^2}$$

This formula can be solved iteratively to determine the three roots as summarized in the following tables:

i	x_i	$f(x)$	$f'(x)$	ε_a
0	0	-1	5.5	
1	0.181818	-0.12923	4.095041	100.000000%
2	0.213375	-0.0037	3.861294	14.789338%
3	0.214332	-3.4E-06	3.85425	0.446594%
4	0.214333	-2.8E-12	3.854244	0.000408%

i	x_i	$f(x)$	$f'(x)$	ε_a
0	2	-2	-4.5	
1	1.555556	-0.24143	-3.31481	28.571429%
2	1.482723	-0.00903	-3.06408	4.912085%
3	1.479775	-1.5E-05	-3.0536	0.199247%
4	1.479769	-4.6E-11	-3.05358	0.000342%

i	x_i	$f(x)$	$f'(x)$	ε_a
0	6	-4	11.5	
1	6.347826	0.625955	15.15974	5.479452%
2	6.306535	0.009379	14.7063	0.654728%
3	6.305898	2.22E-06	14.69934	0.010114%
4	6.305898	1.42E-13	14.69934	0.000002%

Therefore, the roots are 0.214333, 1.479769, and 6.305898.

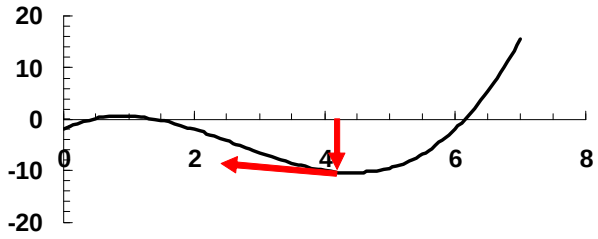
6.5 (a) The Newton-Raphson method can be set up as

$$x_{i+1} = x_i - \frac{-2 + 6x_i - 4x_i^2 + 0.5x_i^3}{6 - 8x_i + 1.5x_i^2}$$

Using an initial guess of 4.2, this formula jumps around and eventually converges on the root at 0.474572 after 9 iterations:

i	x_i	$f(x)$	$f'(x)$	ε_a
0	4.2	-10.316	-1.14	
1	-4.84912	-182.162	80.06397	186.61%
2	-2.57392	-52.4699	36.52895	88.39%
3	-1.13753	-14.737	17.04115	126.27%
4	-0.27274	-3.94412	8.293487	317.08%
5	0.20283	-0.94341	4.439071	234.47%
6	0.415354	-0.16212	2.935948	51.17%
7	0.470574	-0.01021	2.567567	11.73%
8	0.474552	-5.2E-05	2.541384	0.84%
9	0.474572	-1.4E-09	2.541249	0.00%

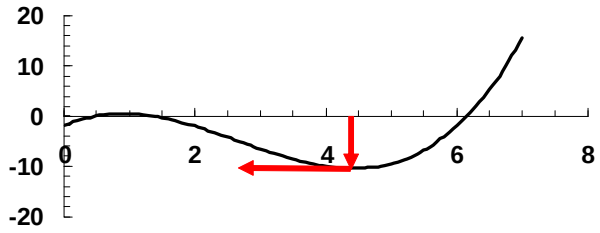
The reason for the behavior is depicted in the following plot. As can be seen, the guess of $x = 4.2$ corresponds to a small negative slope of the function. Hence, the first iteration shoots to a negative value that is far from a root.



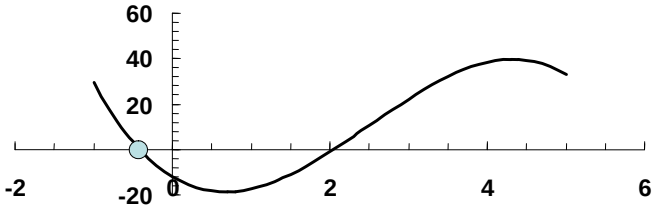
(b) Using an initial guess of 4.43, this formula jumps around even more and eventually converges on the root at 0.474572 after 25 iterations:

i	x_i	$f(x)$	$f'(x)$	ε_a
0	4.43	-10.4504	-0.00265	
1	-3939.13	-3.1E+10	23306693	100.11%
2	-2625.2	-9.1E+09	10358532	50.05%
3	-1749.25	-2.7E+09	4603793	50.08%
4	-1165.28	-8E+08	2046132	50.11%
5	-775.964	-2.4E+08	909393.5	50.17%
6	-516.423	-7E+07	404176.4	50.26%
7	-343.397	-2.1E+07	179635.5	50.39%
8	-228.048	-6139351	79839.54	50.58%
9	-151.152	-1818988	35485.77	50.87%
10	-99.8927	-538908	15772.97	51.31%
11	-65.7262	-159642	7011.7	51.98%
12	-42.9582	-47279	3117.774	53.00%
13	-27.7938	-13994.1	1387.097	54.56%
14	-17.7051	-4137.11	617.8453	56.98%
15	-11.009	-1219.99	275.8709	60.82%
16	-6.58671	-357.941	123.7708	67.14%
17	-3.69475	-103.992	56.0347	78.27%
18	-1.8389	-29.6688	25.78352	100.92%
19	-0.68821	-8.1868	12.21615	167.20%
20	-0.01805	-2.10961	6.144891	3712.744%
21	0.325261	-0.45441	3.556607	105.549%
22	0.453025	-0.05629	2.683645	28.203%
23	0.474	-0.00146	2.545015	4.425%
24	0.474572	-1.1E-06	2.541252	0.121%
25	0.474572	-5.9E-13	2.541249	0.000%

The reason for the behavior is depicted in the following plot. As can be seen, the guess of $x = 4.43$ corresponds to an even smaller near-zero negative slope. Hence, the first iteration shoots to a large positive value that is far from a root.



6.6 (a) A graph of the function indicates that the lowest real root is approximately –0.4:



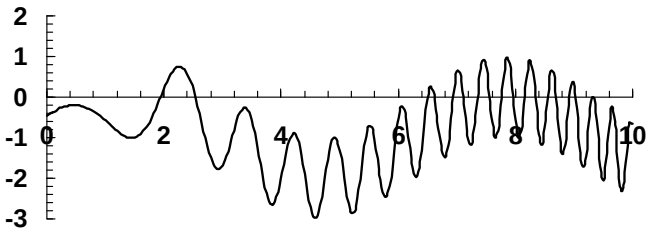
(b) The stopping criterion corresponding to 3 significant figures can be determined with Eq. 3.7 is

$$\varepsilon_s = (0.5\times 10^{2-3})\% = 0.05\%$$

Using initial guesses of $x_{i-1} = -1$ and $x_i = -0.6$, the secant method can be iterated to this level as summarized in the following table:

i	x_{i-1}	$f(x_{i-1})$	x_i	$f(x_i)$	ε_a
0	-1	29.4	**-0.6**	7.5984	
1	-0.6	7.5984	**-0.46059**	1.72547499	30.268%
2	-0.46059	1.725475	**-0.41963**	0.15922371	9.761%
3	-0.41963	0.159224	**-0.41547**	0.00396616	1.002%
4	-0.41547	0.003966	**-0.41536**	9.539E-06	0.026%

6.7 A graph of the function indicates that the first positive root occurs at about 1.9. However, the plot also indicates that there are many other positive roots.

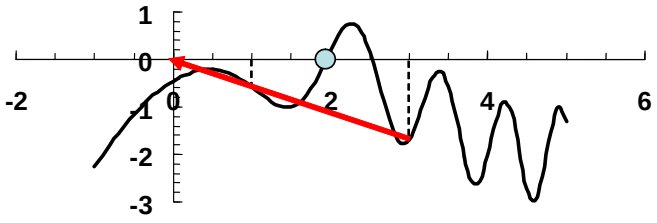


(a) For initial guesses of $x_{i-1} = 1.0$ and $x_i = 3.0$, four iterations of the secant method yields

i	x_{i-1}	$f(x_{i-1})$	x_i	$f(x_i)$	ε_a
0	1	-0.57468	**3**	-1.697951521	
1	3	-1.69795	**-0.02321**	-0.483363437	13023.081%
2	-0.02321	-0.48336	**-1.22635**	-2.744750012	98.107%
3	-1.22635	-2.74475	**0.233951**	-0.274717273	624.189%

4	0.233951	-0.27472	**0.396366**	-0.211940326	40.976%

The result jumps to a negative value due to the poor choice of initial guesses as illustrated in the following plot:

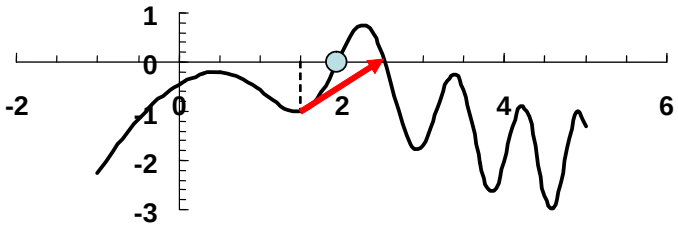


Thereafter, it seems to be converging slowly towards the lowest positive root. However, if the iterations are continued, the technique again runs into trouble when the near-zero slope at 0.5 is approached. At that point, the solution shoots far from the lowest root with the result that it eventually converges to a root at 177.26!

(b) For initial guesses of $x_{i-1} = 1.0$ and $x_i = 3.0$, four iterations of the secant method yields

i	x_{i-1}	$f(x_{i-1})$	x_i	$f(x_i)$	ε_a
0	1.5	-0.99663	**2.5**	0.1663963	
1	2.5	0.166396	**2.356929**	0.6698423	6.070%
2	2.356929	0.669842	**2.547287**	-0.0828279	7.473%
3	2.547287	-0.08283	**2.526339**	0.0314711	0.829%
4	2.526339	0.031471	**2.532107**	0.0005701	0.228%

For these guesses, the result jumps to the vicinity of the second lowest root at 2.5 as illustrated in the following plot:

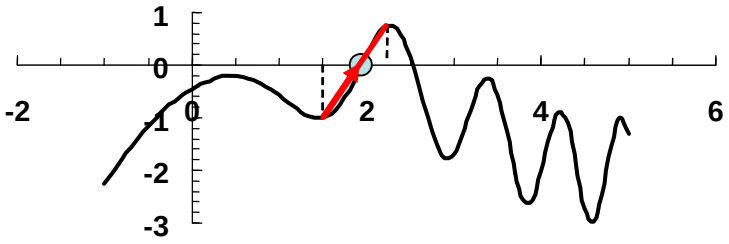


Thereafter, although the two guesses bracket the lowest root, the way that the secant method sequences the iteration results in the technique missing the lowest root.

(c) For initial guesses of $x_{i-1} = 1.5$ and $x_i = 2.25$, four iterations of the secant method yields

i	x_{i-1}	$f(x_{i-1})$	x_i	$f(x_i)$	ε_a
0	1.5	-0.996635	**2.25**	0.753821	
1	2.25	0.753821	**1.927018**	-0.061769	16.761%
2	1.927018	-0.061769	**1.951479**	0.024147	1.253%
3	1.951479	0.024147	**1.944604**	-0.000014	0.354%
4	1.944604	-0.000014	**1.944608**	0.000000	0.000%

For this case, the secant method converges rapidly on the lowest root at 1.9446 as illustrated in the following plot:



6.8 The modified secant method locates the root to the desired accuracy after one iteration:

$x_0 = 3.5$ $\quad f(x_0) = 0.21178$
$x_0 + \delta x_0 = 3.535$ $\quad f(x_0 + \delta x_0) = 3.054461$

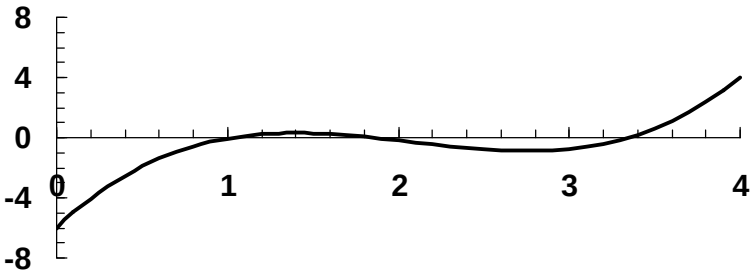
$$x_1 = 3.5 - \frac{0.035(0.21178)}{3.054461 - 0.21178} = 3.497392$$

$$\varepsilon_a = \left|\frac{3.497392 - 3.5}{3.497392}\right| \times 100\% = 0.075\%$$

Note that within 3 iterations, the root is determined to 7 significant digits as summarized below:

i	*x*	*f(x)*	*dx*	*x+dx*	*f(x+dx)*	*f'(x)*	ε_a
0	**3.5**	0.21178	0.03500	3.535	3.05446	81.2194	
1	**3.497392**	0.002822	0.03497	3.532366	2.83810	81.0683	0.075%
2	**3.497358**	3.5E-05	0.03497	3.532331	2.83521	81.0662	0.001%
3	**3.497357**	4.35E-07	0.03497	3.532331	2.83518	81.0662	0.000%

6.9 (a) Graphical



Highest real root ≈ 3.3

(b) Newton-Raphson

i	x_i	*f(x)*	*f'(x)*	ε_a
0	**3.5**	0.60625	4.5125	
1	**3.365651**	0.071249	3.468997	3.992%
2	**3.345112**	0.001549	3.318537	0.614%
3	**3.344645**	7.92E-07	3.315145	0.014%

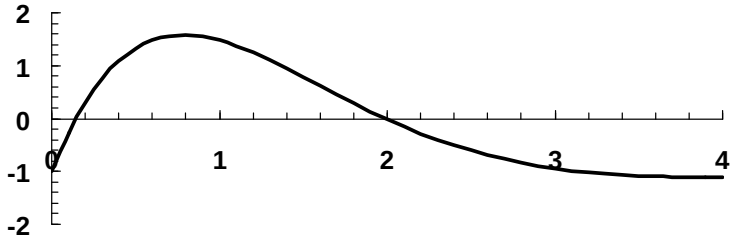
(c) Secant

i	x_{i-1}	$f(x_{i-1})$	x_i	$f(x_i)$	ε_a
0	2.5	-0.78125	**3.5**	0.60625	
1	3.5	0.60625	**3.063063**	-0.6667	14.265%
2	3.063063	-0.6667	**3.291906**	-0.16487	6.952%
3	3.291906	-0.16487	**3.367092**	0.076256	2.233%

(d) Modified secant ($\delta = 0.01$)

i	x	$x+dx$	$f(x)$	$f(x+dx)$	$f'(x)$	ε_a
0	3.5	3.535	0.60625	0.76922	4.6563	
1	3.3698	3.403498	0.085704	0.207879	3.6256	3.864%
2	3.346161	3.379623	0.005033	0.120439	3.4489	0.706%
3	3.344702	3.378149	0.000187	0.115181	3.4381	0.044%

6.10 (a) Graphical



Lowest positive real root ≈ 0.15

(b) Newton-Raphson

i	x_i	$f(x)$	$f'(x)$	ε_a
0	**0.3**	0.751414	3.910431	
1	**0.107844**	-0.22695	6.36737	178.180%
2	**0.143487**	-0.00895	5.868388	24.841%
3	**0.145012**	-1.6E-05	5.847478	1.052%

(c) Secant

i	x_{i-1}	$f(x_{i-1})$	x_i	$f(x_i)$	ε_a
0	0.5	1.32629	**0.4**	1.088279	
1	0.4	1.088279	**-0.05724**	-1.48462	798.821%
2	-0.05724	-1.48462	**0.206598**	0.334745	127.706%
3	0.206598	0.334745	**0.158055**	0.075093	30.713%
4	0.158055	0.075093	**0.144016**	-0.00585	9.748%
5	0.144016	-0.00585	**0.14503**	9E-05	0.699%

(d) Modified secant ($\delta = 0.01$)

i	x	$x+dx$	$f(x)$	$f(x+dx)$	$f'(x)$	ε_a
0	**0.3**	0.303	0.751414	0.763094	3.893468	

1	**0.107007**	0.108077	-0.23229	-0.22547	6.371687	180.357%
2	**0.143463**	0.144897	-0.00909	-0.00069	5.858881	25.412%
3	**0.145015**	0.146465	-1.2E-06	0.008464	5.83752	1.070%
4	**0.145015**	0.146465	2.12E-09	0.008465	5.837517	0.000%
5	**0.145015**	0.146465	-3.6E-12	0.008465	5.837517	0.000%

6.11 The Newton-Raphson method can be set up as

$$x_{i+1} = x_i - \frac{e^{-0.5x_i}(4 - x_i) - 2}{-e^{-0.5x_i}(3 - 0.5x_i)}$$

(a)

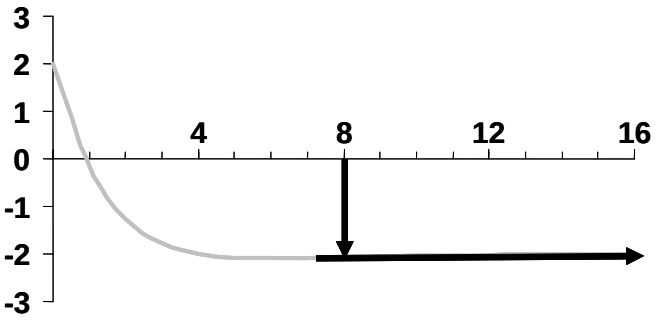
i	*x*	*f*(*x*)	*f*'(*x*)
0	2	-1.26424	-0.73576
1	0.281718	1.229743	-2.48348
2	0.776887	0.18563	-1.77093
3	0.881708	0.006579	-1.64678
4	0.885703	9.13E-06	-1.64221
5	0.885709	1.77E-11	-1.6422
6	0.885709	0	-1.6422

(b) The case does not work because the derivative is zero at $x_0 = 6$.

(c)

i	*x*	*f*(*x*)	*f*'(*x*)
0	8	-2.07326	0.018316
1	121.1963	-2	2.77E-25
2	7.21E+24	-2	0

This guess breaks down because, as depicted in the following plot, the near zero, positive slope sends the method away from the root.



6.12 The optimization problem involves determining the root of the derivative of the function. The derivative is the following function,

$$f'(x) = -12x^5 - 6x^3 + 10$$

The Newton-Raphson method is a good choice for this problem because

- The function is easy to differentiate
- It converges very rapidly

The Newton-Raphson method can be set up as

$$x_{i+1} = x_i - \frac{-12x_i^5 - 6x_i^3 + 10}{-60x_i^4 - 18x_i^2}$$

First iteration:

$$x_1 = x_0 - \frac{-12(1)^5 - 6(1)^3 + 10}{-60(1)^4 - 18(1)^2} = 0.897436$$
$$\varepsilon_a = \left|\frac{0.897436 - 1}{0.897436}\right| \times 100\% = 11.43\%$$

Second iteration:

$$x_1 = x_0 - \frac{-12(0.897436)^5 - 6(0.897436)^3 + 10}{-60(0.897436)^4 - 18(0.897436)^2} = 0.872682$$
$$\varepsilon_a = \left|\frac{0.872682 - 0.897436}{0.872682}\right| \times 100\% = 2.84\%$$

Since ε_a < 5%, the solution can be terminated.

6.13 Newton-Raphson is the best choice because:

- You know that the solution will converge. Thus, divergence is not an issue.
- Newton-Raphson is generally considered the fastest method
- You only require one guess
- The function is easily differentiable

To set up the Newton-Raphson first formulate the function as a roots problem and then differentiate it

$$f(x) = e^{0.5x} - 5 + 5x$$
$$f'(x) = 0.5e^{0.5x} + 5$$

These can be substituted into the Newton-Raphson formula

$$x_{i+1} = x_i - \frac{e^{0.5x_i} - 5 + 5x_i}{0.5e^{0.5x_i} + 5}$$

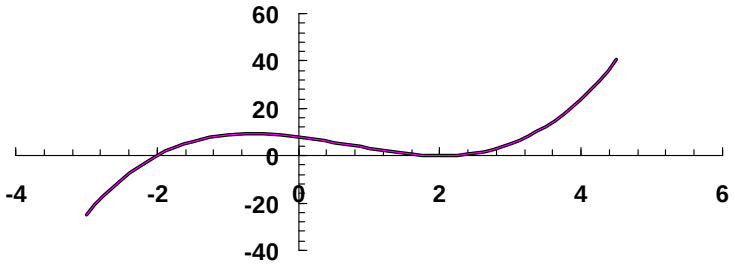
First iteration:

$$x_1 = 0.7 - \frac{e^{0.5(0.7)} - 5 + 5(0.7)}{0.5e^{0.5(0.7)} + 5} = 0.7 - \frac{-0.08093}{5.7095} = 0.714175$$

$$\varepsilon_a = \left|\frac{0.714175 - 0.7}{0.714175}\right| \times 100\% = 1.98\%$$

Therefore, only one iteration is required.

6.14 As indicated by the following plot, a double root is located at $x = 2$.



(a) The standard Newton-Raphson method can be set up as

$$x_{i+1} = x_i - \frac{x_i^3 - 2x_i^2 - 4x_i + 8}{3x_i^2 - 4x_i - 4}$$

As expected, this method converges slowly as summarized in the following table:

i	x_i	*f*(*x*)	*f*'(*x*)	ε_a
0	**1.2**	2.048	-4.48	
1	**1.657143**	0.429901	-2.3902	27.586%
2	**1.837002**	0.101942	-1.22428	9.791%
3	**1.92027**	0.024921	-0.61877	4.336%
4	**1.960544**	0.006166	-0.31097	2.054%
5	**1.980371**	0.001534	-0.15588	1.001%
6	**1.99021**	0.000382	-0.07803	0.494%
7	**1.995111**	9.55E-05	-0.03904	0.246%
8	**1.997557**	2.39E-05	-0.01953	0.122%
9	**1.998779**	5.96E-06	-0.00976	0.061%
10	**1.99939**	1.49E-06	-0.00488	0.031%
11	**1.999695**	3.73E-07	-0.00244	0.015%
12	**1.999847**	9.31E-08	-0.00122	0.008%

(b) The modified Newton-Raphson method (Eq. 6.12) can be set up for the double root ($m = 2$) as

$$x_{i+1} = x_i - 2\frac{x_i^3 - 2x_i^2 - 4x_i + 8}{3x_i^2 - 4x_i - 4}$$

This method converges much quicker than the standard approach in **(a)** as summarized in the following table:

i	x_i	*f*(*x*)	*f*'(*x*)	ε_a
0	**1.2**	2.048	-4.48	
1	**2.114286**	0.053738	0.953469	43.243%
2	**2.001566**	9.81E-06	0.012532	5.632%

3	**2**	3.75E-13	2.45E-06	0.078%

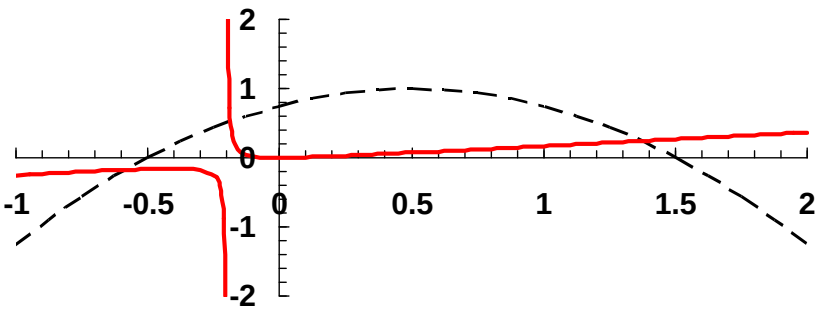
(c) The modified Newton-Raphson method (Eq. 6.16) can be set up for the double root ($m = 2$) as

$$x_{i+1} = x_i - \frac{\left(x_i^3 - 2x_i^2 - 4x_i + 8\right)\left(3x_i^2 - 4x_i - 4\right)}{\left(3x_i^2 - 4x_i - 4\right)^2 - \left(x_i^3 - 2x_i^2 - 4x_i + 8\right)\left(6x_i - 4\right)}$$

This method also converges much quicker than the standard approach in **(a)** as summarized in the following table:

i	x_i	*f*(*x*)	*f*'(*x*)	*f*"(*x*)	ε_a
0	**1.2**	2.048	-4.48	3.2	
1	**1.878788**	0.056989	-0.92562	7.272727	36.129%
2	**1.998048**	1.52E-05	-0.01561	7.988287	5.969%
3	**2**	9.09E-13	-3.8E-06	7.999997	0.098%

6.15 The functions can be plotted (y versus x). The plot indicates that there are three real roots at about (–0.6, –0.18), (–0.19, 0.6), and (1.37, 0.24).



(a) There are numerous ways to set this problem up as a fixed-point iteration. One way that converges is to solve the first equation for x and the second for y,

$$x = \sqrt{x + 0.75 - y}$$

$$y = \frac{x^2}{1 + 5x}$$

Using initial values of $x = y = 1.2$, the first iteration can be computed as:

$$x = \sqrt{1.2 + 0.75 - 1.2} = 0.866025$$

$$y = \frac{(0.866025)^2}{1 + 5(0.866025)} = 0.14071$$

Second iteration

$$x = \sqrt{0.866025 + 0.75 - 0.14071} = 1.214626$$

$$y = \frac{(1.214626)^2}{1 + 5(1.214626)} = 0.20858$$

Third iteration

$$x = \sqrt{1.214626 + 0.75 - 0.20858} = 1.325159$$
$$y = \frac{(1.325159)^2}{1 + 5(1.325159)} = 0.230277$$

Thus, the computation is converging on the root at $x = 1.372065$ and $y = 0.239502$.

Note that some other configurations are convergent and others are divergent. This exercise is intended to illustrate that although it may sometimes work, fixed-point iteration does not represent a practical general-purpose approach for solving systems of nonlinear equations.

(b) The equations to be solved are

$$u(x, y) = -x^2 + x + 0.75 - y$$
$$v(x, y) = x^2 - y - 5xy$$

The partial derivatives can be computed and evaluated at the initial guesses ($x = 1.2$, $y = 1.2$) as

$$\frac{\partial u}{\partial x} = -2x + 1 = -1.4 \qquad \frac{\partial u}{\partial y} = -1$$
$$\frac{\partial v}{\partial x} = 2x - 5y = -3.6 \qquad \frac{\partial v}{\partial y} = -1 - 5x = -7$$

The determinant of the Jacobian can be computed as

$$-1.4(-7) - (-1)(-3.6) = 6.2$$

The values of the function at the initial guesses can be computed as

$$u(1.2, 1.2) = -(1.2)^2 + 1.2 + 0.75 - 1.2 = -0.69$$
$$v(1.2, 1.2) = (1.2)^2 - 1.2 - 5(1.2)(1.2) = -6.96$$

These values can be substituted into Eq. (6.24) to give

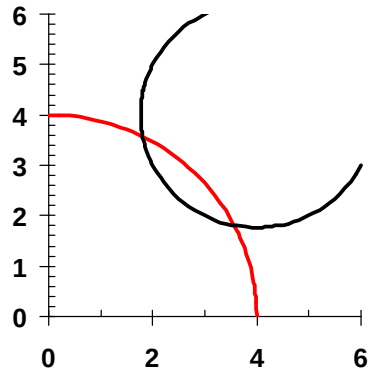
$$x = 1.2 - \frac{-0.69(-7) - (-6.96)(-1)}{6.2} = 1.543548$$
$$y = 1.2 - \frac{-6.96(-1.4) - (-0.69)(-3.6)}{6.2} = 0.0290325$$

The remaining iterations are summarized below:

i	x_i	y_i	ε_a
0	**1.2**	**1.2**	
1	**1.543548**	**0.0290325**	4033%

2	**1.394123**	**0.2228721**	86.97%
3	**1.372455**	**0.2392925**	6.86%
4	**1.372066**	**0.2395019**	0.0874%
5	**1.372065**	**0.2395019**	1.87×10^{-5}%

6.16 The functions can be plotted (y versus x). The plot indicates that there are two roots at about (1.8, 3.6) and (3.6, 1.8).



To implement the Newton-Raphson method, the equations to be solved are

$$u(x, y) = 5 - (x-4)^2 - (y-4)^2$$
$$v(x, y) = 16 - x^2 - y^2$$

The partial derivatives can be computed and evaluated at the first set of initial guesses (x = 1.8, y = 3.6) as

$$\frac{\partial u}{\partial x} = -2(x-4) = 4.4 \qquad \frac{\partial u}{\partial y} = -2(y-4) = 0.8$$
$$\frac{\partial v}{\partial x} = -2x = -3.6 \qquad \frac{\partial v}{\partial y} = -2y = -7.2$$

The determinant of the Jacobian can be computed as

$$4.4(-7.2) - 0.8(-3.6) = -28.8$$

The values of the function at the initial guesses can be computed as

$$u(1.8, 3.6) = 5 - (1.8-4)^2 - (3.6-4)^2 = 0$$
$$v(1.8, 3.6) = 16 - (1.8)^2 - (3.6)^2 = -0.2$$

These values can be substituted into Eq. (6.24) to give

$$x = 1.8 - \frac{0(-7.2) - (-0.2)(0.8)}{-28.8} = 1.805556$$
$$y = 3.6 - \frac{-0.2(4.4) - 0(-3.6)}{-28.8} = 3.569444$$

The remaining iterations are summarized below:

i	x_i	y_i	ε_a
0	**1.8**	**3.6**	
1	**1.805556**	**3.569444**	0.856%
2	**1.805829**	**3.569171**	0.0151%
3	**1.805829**	**3.569171**	2.35×10^{-6}%

For the second set of initial guesses ($x = 3.6$, $y = 1.8$), the partial derivatives can be computed and evaluated as

$$\frac{\partial u}{\partial x} = -2(x-4) = 0.8 \qquad \frac{\partial u}{\partial y} = -2(y-4) = 4.4$$

$$\frac{\partial v}{\partial x} = -2x = -7.2 \qquad \frac{\partial v}{\partial y} = -2y = -3.6$$

The determinant of the Jacobian can be computed as

$$0.8(-3.6) - 4.4(-7.2) = 28.8$$

The values of the function at the initial guesses can be computed as

$$u(1.8, 3.6) = 5 - (3.6-4)^2 - (1.8-4)^2 = 0$$
$$v(1.8, 3.6) = 16 - (3.6)^2 - (1.8)^2 = -0.2$$

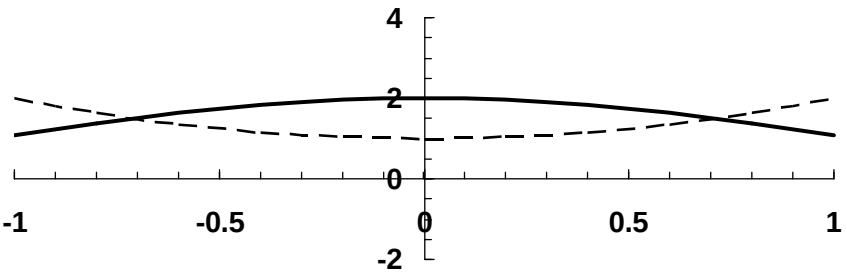
These values can be substituted into Eq. (6.24) to give

$$x = 3.6 - \frac{0(-3.6) - (-0.2)(4.4)}{28.8} = 3.569444$$
$$y = 1.8 - \frac{-0.2(0.8) - 0(-7.2)}{28.8} = 1.805556$$

The remaining iterations are summarized below:

i	x_i	y_i	ε_a
0	**3.6**	**1.8**	
1	**3.569444**	**1.805556**	0.856%
2	**3.569171**	**1.805829**	0.0151%
3	**3.569171**	**1.805829**	2.35×10^{-6}%

6.17 The functions can be plotted (y versus x). The plot indicates that there are two roots at about (–0.7, 1.5) and (0.7, 1.5).



To implement the Newton-Raphson method, the equations to be solved are

$$u(x, y) = x^2 + 1 - y$$
$$v(x, y) = 2\cos x - y$$

We will solve for the positive root. The partial derivatives can be computed and evaluated at the initial guesses ($x = 0.7$, $y = 1.5$) as

$$\frac{\partial u}{\partial x} = 2x = 1.4 \qquad \frac{\partial u}{\partial y} = -1$$
$$\frac{\partial v}{\partial x} = -2\sin(0.7) = -1.288435 \qquad \frac{\partial v}{\partial y} = -1$$

The determinant of the Jacobian can be computed as

$$1.4(-1) - (-1)(-1.288435) = -2.688435$$

The values of the function at the initial guesses can be computed as

$$u(0.7, 1.5) = (0.7)^2 + 1 - 1.5 = -0.01$$
$$v(0.7, 1.5) = 2\cos(0.7) - 1.5 = 0.0296844$$

These values can be substituted into Eq. (6.24) to give

$$x = 0.7 - \frac{-0.01(-1) - 0.0296844(-1)}{-2.688435} = 0.7147611$$
$$y = 1.5 - \frac{0.0296844(1.4) - (-0.01)(-1.288435)}{-2.688435} = 1.510666$$

The remaining iterations are summarized below:

i	x_i	y_i	ε_a
0	**0.7**	**1.5**	
1	**0.7147611**	**1.510666**	2.065%
2	**0.7146211**	**1.510683**	0.0196%
3	**0.7146211**	**1.510683**	1.76×10^{-6}%

6.18 The function to be evaluated is

$$f(c) = \frac{W}{V} - \frac{Q}{V}c - k\sqrt{c} = 1 - 0.1c - 0.25\sqrt{c}$$

Using an initial guess of $x_0 = 4$ and $\delta = 0.5$, the three iterations can be summarized as

i	*x*	*x+dx*	*f(x)*	*f(x+dx)*	*f'(x)*	ε_a
0	**4**	6	0.1	-0.21237	-0.15619	
1	**4.640261**	6.960392	-0.00256	-0.3556	-0.15217	13.798%
2	**4.623452**	6.935178	9.94E-05	-0.35189	-0.15226	0.364%
3	**4.624105**	6.936158	-3.8E-06	-0.35203	-0.15226	0.014%

Therefore, the root is estimated as $c = 4.624105$. This result can be checked by substituting it into the function to yield,

$$f(c) = 1 - 0.1(4.624105) - 0.25\sqrt{4.624105} = -3.8 \times 10^{-6}$$

6.19 Convergence can be evaluated in two ways. First, we can calculate the derivative of the right-hand side and determine whether it is greater than one. Second, we can develop a graphical representation as in Fig. 6.3.

For the first formulation, the derivative can be evaluated as

$$g'(c) = -\frac{2Q(W - Qc)}{(kV)^2} = -3.2 + 0.32c$$

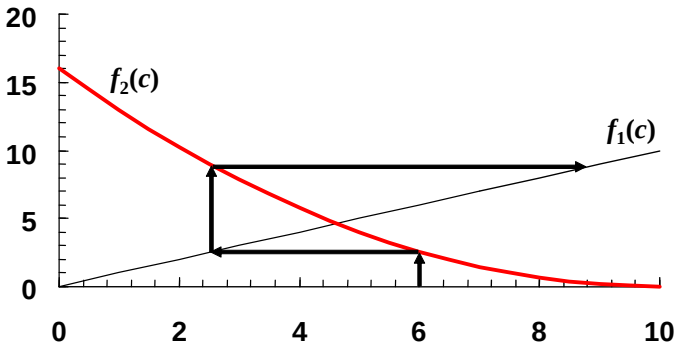
Between, $c = 2$ and 6, this ranges from –2.56 and –1.28. Therefore, because $|g'(c)| > 1$, we would expect that fixed-point iteration would be divergent.

The second way to assess divergence is to create a plot of

$$f_1(c) = c$$

$$f_2(c) = \left(\frac{W - Qc}{kV}\right)^2$$

The result also indicates divergence:



For the second formulation, the derivative can be evaluated as

$$g'(c) = -\frac{kV}{2Q\sqrt{c}} = -\frac{1.25}{\sqrt{c}}$$

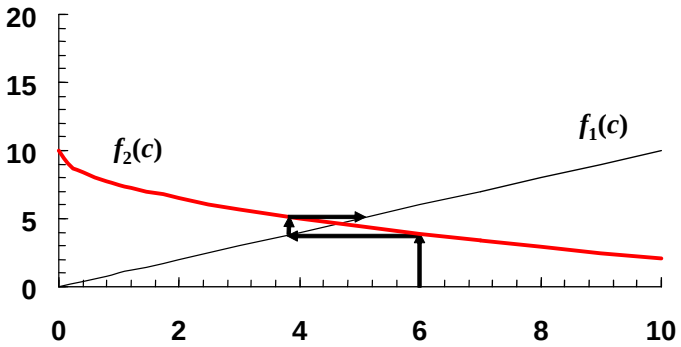
Between, c = 2 and 6, this ranges from –0.883 and –0.51. Therefore, because $|g'(c)| < 1$, we would expect that fixed-point iteration would be convergent.

The second way to assess divergence is to create a plot of

$$f_1(c) = c$$

$$f_2(c) = \frac{W - kV\sqrt{c}}{Q}$$

The result also indicates convergence:



Here are the results of using fixed-point iteration to determine the root for the second formulation.

i	x_i	ε_a	$E_{t,i}/E_{t,i-1}$
0	4		
1	5	20.0%	-0.60236
2	4.40983	13.4%	-0.56994
3	4.750101	7.2%	-0.58819
4	4.551318	4.4%	-0.57739
5	4.666545	2.5%	-0.5836
6	4.599453	1.5%	-0.57997
7	4.638416	0.8%	-0.58207
8	4.615754	0.5%	-0.58084
9	4.628923	0.3%	-0.58156
10	4.621267	0.2%	-0.58114

Notice that we have included the true error and the ratio of the true errors between iterations. The latter should be equal to $|g'(c)|$, which at the root is equal to

$$g'(4.624081) = -\frac{1.25}{\sqrt{4.624081}} = -0.5813$$

Thus, the computation verifies the theoretical result that was derived in Box 6.1 (p. 147).

6.20 Here is a VBA program to implement the Newton-Raphson algorithm and solve Example 6.3.

```
Option Explicit

Sub NewtRaph()
Dim imax As Integer, iter As Integer
Dim x0 As Double, es As Double, ea As Double
x0 = 0#
es = 0.01
imax = 20
MsgBox "Root: " & NewtR(x0, es, imax, iter, ea)
MsgBox "Iterations: " & iter
MsgBox "Estimated error: " & ea
End Sub

Function df(x)
df = -Exp(-x) - 1#
End Function

Function f(x)
f = Exp(-x) - x
End Function

Function NewtR(x0, es, imax, iter, ea)
Dim xr As Double, xrold As Double
xr = x0
iter = 0
Do
  xrold = xr
  xr = xr - f(xr) / df(xr)
  iter = iter + 1
  If (xr <> 0) Then
    ea = Abs((xr - xrold) / xr) * 100
  End If
  If ea < es Or iter >= imax Then Exit Do
Loop
NewtR = xr
End Function
```

When this program is run, it yields a root of 0.5671433 after 4 iterations. The approximate error at this point is 2.21×10^{-5}%.

6.21 Here is a VBA program to implement the secant algorithm and solve Example 6.6.

```
Option Explicit

Sub SecMain()
Dim imax As Integer, iter As Integer
Dim x0 As Double, x1 As Double, xr As Double
Dim es As Double, ea As Double
x0 = 0
x1 = 1
es = 0.01
imax = 20
MsgBox "Root: " & Secant(x0, x1, xr, es, imax, iter, ea)
MsgBox "Iterations: " & iter
MsgBox "Estimated error: " & ea
End Sub

Function f(x)
f = Exp(-x) - x
End Function
```

```
Function Secant(x0, x1, xr, es, imax, iter, ea)
xr = x1
iter = 0
Do
  xr = x1 - f(x1) * (x0 - x1) / (f(x0) - f(x1))
  iter = iter + 1
  If (xr <> 0) Then
    ea = Abs((xr - x1) / xr) * 100
  End If
  If ea < es Or iter >= imax Then Exit Do
  x0 = x1
  x1 = xr
Loop
Secant = xr
End Function
```

When this program is run, it yields a root of 0.5671433 after 4 iterations. The approximate error at this point is 4.77×10^{-3} %.

For MATLAB users, here is an M-file to solve the same problem:

```
function root = secant(func,xrold,xr,es,maxit)
% secant(func,xrold,xr,es,maxit):
%   uses secant method to find the root of a function
% input:
%   func = name of function
%   xrold, xr = initial guesses
%   es = (optional) stopping criterion (%)
%   maxit = (optional) maximum allowable iterations
% output:
%   root = real root

% if necessary, assign default values
if nargin<5, maxit=50; end     %if maxit blank set to 50
if nargin<4, es=0.001; end     %if es blank set to 0.001
% Secant method
iter = 0;
while (1)
  xrn = xr - func(xr)*(xrold - xr)/(func(xrold) - func(xr));
  iter = iter + 1;
  if xrn ~= 0, ea = abs((xrn - xr)/xrn) * 100; end
  if ea <= es | iter >= maxit, break, end
  xrold = xr;
  xr = xrn;
end
root = xrn;
```

```
>> secant(inline('exp(-x)-x'),0,1)

ans =
    0.5671
```

6.22 Here is a VBA program to implement the modified secant algorithm and solve Example 6.8.

```
Option Explicit

Sub SecMod()
Dim imax As Integer, iter As Integer
Dim x As Double, es As Double, ea As Double
```

```
x = 1
es = 0.01
imax = 20
MsgBox "Root: " & ModSecant(x, es, imax, iter, ea)
MsgBox "Iterations: " & iter
MsgBox "Estimated error: " & ea
End Sub

Function f(x)
f = Exp(-x) - x
End Function

Function ModSecant(x, es, imax, iter, ea)
Dim xr As Double, xrold As Double, fr As Double
Const del As Double = 0.01
xr = x
iter = 0
Do
  xrold = xr
  fr = f(xr)
  xr = xr - fr * del * xr / (f(xr + del * xr) - fr)
  iter = iter + 1
  If (xr <> 0) Then
    ea = Abs((xr - xrold) / xr) * 100
  End If
  If ea < es Or iter >= imax Then Exit Do
Loop
ModSecant = xr
End Function
```

When this program is run, it yields a root of 0.5671433 after 4 iterations. The approximate error at this point is 2.36×10^{-5} %.

For MATLAB users, here is an M-file to solve the same problem:

```
function root = modsec(func,xr,delta,es,maxit)
% modsec(func,xr,delta,es,maxit):
%   uses the modified secant method
%   to find the root of a function
% input:
%   func = name of function
%   xr = initial guess
%   delta = perturbation fraction
%   es = (optional) stopping criterion (%)
%   maxit = (optional) maximum allowable iterations
% output:
%   root = real root

% if necessary, assign default values
if nargin<5, maxit=50; end     %if maxit blank set to 50
if nargin<4, es=0.001; end     %if es blank set to 0.001
if nargin<3, delta=1E-5; end   %if delta blank set to 0.00001
% Secant method
iter = 0;
while (1)
  xrold = xr;
  xr = xr - delta*xr*func(xr)/(func(xr+delta*xr)-func(xr));
  iter = iter + 1;
  if xr ~= 0, ea = abs((xr - xrold)/xr) * 100; end
  if ea <= es | iter >= maxit, break, end
end
```

```
root = xr;
```

```
>> modsec(inline('exp(-x)-x'),1,.01)

ans =
    0.5671
```

6.23 [First printing errata: p. 163, Fig. 6.12. Line 8 of pseudocode should be: *IF Sgn(fa) = Sgn(fb) THEN*]**.**

Here is a VBA program to implement Brent's method and solve Prob. 6.6.

```
Option Explicit

Function fzerosimp(xl, xu)
Dim fa As Double, fb As Double, fc As Double
Dim c As Double, s As Double, d As Double
Dim mflag As Boolean
Dim tol As Double
Dim a As Double, b As Double, e As Double
Dim p As Double, q As Double, r As Double
Dim m As Double
Const eps As Double = 2.22044604925031E-16
tol = 0.000001
a = xl: b = xu: fa = f(a): fb = f(b)
c = a: fc = fa: d = b - c: e = d
Do
  If fb = 0 Then Exit Do
  If Sgn(fa) = Sgn(fb) Then             'If necessary, rearrange points
    a = c: fa = fc: d = b - c: e = d
  End If
  If Abs(fa) < Abs(fb) Then
    c = b: b = a: a = c
    fc = fb: fb = fa: fa = fc
  End If
  m = 0.5 * (a - b)      'Termination test and possible exit
  tol = 2 * eps * max(Abs(b), 1)
  If Abs(m) <= tol Or fb = 0# Then
    Exit Do
  End If
  'Choose open methods or bisection
  If Abs(e) >= tol And Abs(fc) > Abs(fb) Then
    s = fb / fc
    If a = c Then                               'Secant method
      p = 2 * m * s
      q = 1 - s
    Else                     'Inverse quadratic interpolation
      q = fc / fa: r = fb / fa
      p = s * (2 * m * q * (q - r) - (b - c) * (r - 1))
      q = (q - 1) * (r - 1) * (s - 1)
    End If
    If p > 0 Then q = -q Else p = -p
    If 2 * p < 3 * m * q - Abs(tol * q) And p < Abs(0.5 * e * q) Then
      e = d: d = p / q
    Else
      d = m: e = m
    End If
  Else                                              'Bisection
    d = m: e = m
  End If
```

```
  c = b: fc = fb
  If Abs(d) > tol Then b = b + d Else b = b - Sgn(b - a) * tol
  fb = f(b)
Loop
fzerosimp = b
End Function

Function max(x, y)
max = x
If y > x Then max = y
End Function

Function f(x)
f = -12 - 21 * x + 18 * x ^ 2 - 2.4 * x ^ 3
End Function
```

When this program is run, it yields a root of -0.4153607.

For MATLAB users, here is an M-file to solve the same problem:

```
function b = fzerosimp(func,xl,xu,varargin)
% Brent fzero roots zero
% b = fzerosimp(func,xl,xu,p1,p2,...)
%   uses Brent's method to find the root of a function
% input:
%   func = name of function
%   xl, xu = lower and upper guesses
%   p1,p2,... = additional parameters used by func
% output:
%   b = real root
tol = 0.000001;
a = xl; b = xu; fa = func(a,varargin{:}); fb = func(b,varargin{:});
c = a; fc = fa; d = b - c; e = d;
while (1)
  if fb == 0, break, end
  if sign(fa) == sign(fb)              %If necessary, rearrange points
    a = c; fa = fc; d = b - c; e = d;
  end
  if abs(fa) < abs(fb)
    c = b; b = a; a = c;
    fc = fb; fb = fa; fa = fc;
  end
  m = 0.5 * (a - b);      %Termination test and possible exit
  tol = 2 * eps * max(abs(b), 1);
  if abs(m) <= tol | fb == 0.
    break
  end
  %Choose open methods or bisection
  if abs(e) >= tol & abs(fc) > abs(fb)
    s = fb / fc;
    if a == c                              %Secant method
      p = 2 * m * s; q = 1 - s;
    else                    %Inverse quadratic interpolation
      q = fc / fa; r = fb / fa;
      p = s * (2 * m * q * (q - r) - (b - c) * (r - 1));
      q = (q - 1) * (r - 1) * (s - 1);
    end
    if p > 0, q = -q; else p = -p; end;
    if 2 * p < 3 * m * q - abs(tol * q) & p < abs(0.5 * e * q)
      e = d; d = p / q;
    else
```

```
      d = m; e = m;
    end
  else                                              %Bisection
    d = m; e = m;
  end
  c = b; fc = fb;
  if abs(d) > tol, b = b + d; else b = b - sign(b - a) * tol; end
  fb = func(b,varargin{:});
end
```

```
>> format long
>> fcd=@(x) -12 - 21 * x + 18 * x ^ 2 - 2.4 * x ^ 3;
>> root =fzerosimp(fcd,-1,0)

root =

  -0.415360709034517
```

6.24 Here is a VBA program to implement the 2 equation Newton-Raphson method and solve Example 6.12.

```
Option Explicit

Sub NewtRaphSyst()
Dim imax As Integer, iter As Integer
Dim x0 As Double, y0 As Double, xr As Double
Dim yr As Double, es As Double, ea As Double
x0 = 1.5
y0 = 3.5
es = 0.01
imax = 20
Call NR2Eqs(x0, y0, xr, yr, es, imax, iter, ea)
MsgBox "x, y = " & xr & ", " & yr
MsgBox "Iterations: " & iter
MsgBox "Estimated error: " & ea
End Sub

Sub NR2Eqs(x0, y0, xr, yr, es, imax, iter, ea)
Dim J As Double, eay As Double
iter = 0
Do
  J = dudx(x0, y0) * dvdy(x0, y0) - dudy(x0, y0) * dvdx(x0, y0)
  xr = x0 - (u(x0, y0) * dvdy(x0, y0) - v(x0, y0) * dudy(x0, y0)) / J
  yr = y0 - (v(x0, y0) * dudx(x0, y0) - u(x0, y0) * dvdx(x0, y0)) / J
  iter = iter + 1
  If (xr <> 0) Then
    ea = Abs((xr - x0) / xr) * 100
  End If
  If (xr <> 0) Then
    eay = Abs((yr - y0) / yr) * 100
  End If
  If eay > ea Then ea = eay
  If ea < es Or iter >= imax Then Exit Do
  x0 = xr
  y0 = yr
Loop
End Sub
Function u(x, y)
u = x ^ 2 + x * y - 10
End Function
Function v(x, y)
```

```
v = y + 3 * x * y ^ 2 - 57
End Function
Function dudx(x, y)
dudx = 2 * x + y
End Function
Function dudy(x, y)
dudy = x
End Function
Function dvdx(x, y)
dvdx = 3 * y ^ 2
End Function
Function dvdy(x, y)
dvdy = 1 + 6 * x * y
End Function
```

Its application yields roots of $x = 2$ and $y = 3$ after 4 iterations. The approximate error at this point is 1.96×10^{-5}%.

6.25 The program from Prob. 6.24 can be set up to solve Prob. 6.15, by changing the functions to

```
Function u(x, y)
u = y + x ^ 2 - 0.75 - x
End Function
Function v(x, y)
v = x ^ 2 - 5 * x * y - y
End Function
Function dudx(x, y)
dudx = 2 * x - 1
End Function
Function dudy(x, y)
dudy = 1
End Function
Function dvdx(x, y)
dvdx = 2 * x ^ 2 - 5 * y
End Function
Function dvdy(x, y)
dvdy = -5 * x
End Function
```

Using a stopping criterion of 0.01%, the program yields $x = 1.3720655$ and $y = 0.2395017$ after 6 iterations with an approximate error of 1.89×10^{-3}%.

The program from Prob. 6.24 can be set up to solve Prob. 6.16, by changing the functions to

```
Function u(x, y)
u = (x - 4) ^ 2 + (y - 4) ^ 2 - 5
End Function
Function v(x, y)
v = x ^ 2 + y ^ 2 - 16
End Function
Function dudx(x, y)
dudx = 2 * (x - 4)
End Function
Function dudy(x, y)
dudy = 2 * (y - 4)
End Function
Function dvdx(x, y)
dvdx = 2 * x
End Function
Function dvdy(x, y)
```

```
dvdy = 2 * y
End Function
```

Using a stopping criterion of 0.01% and initial guesses of 1.8 and 3.6, the program yields $x = 1.805829$ and $y = 3.569171$ after 3 iterations with an approximate error of 2.35×10^{-6}.

Using a stopping criterion of 0.01% and initial guesses of 3.6 and 1.8, the program yields $x = 3.569171$ and $y = 1.805829$ after 3 iterations with an approximate error of 2.35×10^{-6}.

6.26 Determining the square root of a number can be formulated as a roots problem:

$$x = \sqrt{a}$$
$$x^2 = a$$
$$f(x) = x^2 - a = 0 \qquad (1)$$

The derivative of this function is

$$f'(x) = 2x \qquad (2)$$

Substituting (1) and (2) into the Newton Raphson formula (Eq. 6.6) gives

$$x = x - \frac{x^2 - a}{2x}$$

Combining terms yields the "divide and average" method,

$$x = \frac{2x(x) - x^2 + a}{2} = \frac{x^2 + a/x}{2}$$

6.27 (a) The formula for Newton-Raphson is

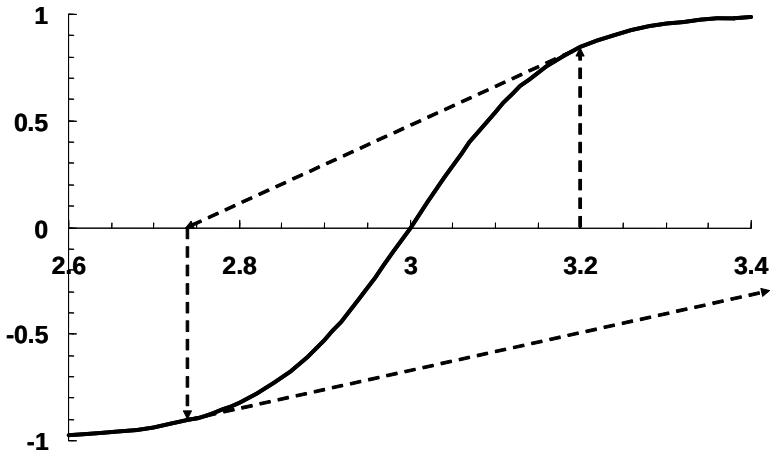
$$x_{i+1} = x_i - \frac{\tanh\left(x_i^2 - 9\right)}{2x_i \operatorname{sech}^2\left(x_i^2 - 9\right)}$$

Using an initial guess of 3.2, the iterations proceed as

iteration	x_i	$f(x_i)$	$f'(x_i)$	ε_a
0	3.2	0.845456	1.825311	
1	2.736816	–0.906910	0.971640	16.924%
2	3.670197	0.999738	0.003844	25.431%
3	–256.413			101.431%

Note that on the fourth iteration, the computation should go unstable.

(b) The solution diverges from its real root of $x = 3$. Due to the concavity of the slope, the next iteration will always diverge. The following graph illustrates how the divergence evolves.



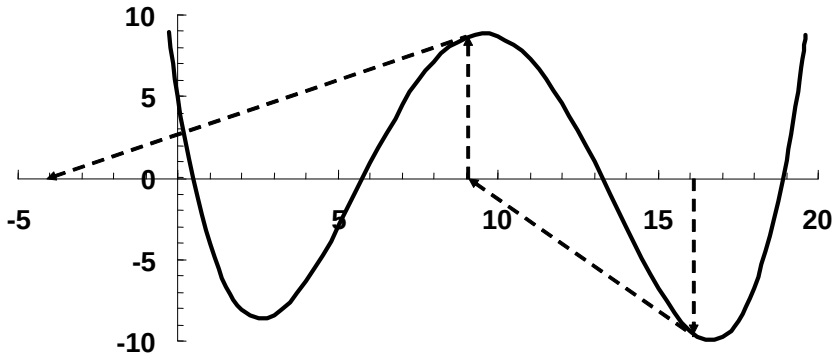
6.28 The formula for Newton-Raphson is

$$x_{i+1} = x_i - \frac{0.0074x_i^4 - 0.284x_i^3 + 3.355x_i^2 - 12.183x_i + 5}{0.0296x_i^3 - 0.852x_i^2 + 6.71x_i - 12.183}$$

Using an initial guess of 16.15, the iterations proceed as

iteration	x_i	$f(x_i)$	$f'(x_i)$	ε_a
0	16.15	–9.57445	–1.35368	
1	9.077102	8.678763	0.662596	77.920%
2	–4.02101	128.6318	–54.864	325.742%
3	–1.67645	36.24995	–25.966	139.852%
4	–0.2804	8.686147	–14.1321	497.887%
5	0.334244	1.292213	–10.0343	183.890%
6	0.463023	0.050416	–9.25584	27.813%
7	0.46847	8.81E–05	–9.22351	1.163%
8	0.46848	2.7E–10	–9.22345	0.002%

As depicted below, the iterations involve regions of the curve that have flat slopes. Hence, the solution is cast far from the roots in the vicinity of the original guess.



6.29

$$f(x) = \pm\sqrt{16-(x+1)^2} + 2$$

$$x_{i+1} = x_i - \frac{f(x_i)(x_{i-1} - x_i)}{f(x_{i-1}) - f(x_i)}$$

1st iteration

$$x_{i-1} = 0.5 \Rightarrow f(x_{i-1}) = -1.708$$

$$x_i = 3 \Rightarrow f(x_i) = 2$$

$$x_{i+1} = 3 - \frac{2(0.5-3)}{(-1.708-2)} = 1.6516$$

2nd iteration

$$x_i = 1.6516 \Rightarrow f(x_i) = -0.9948$$

$$x_{i\ -1} = 0.5 \Rightarrow f(x_{i-1}) = -1.46$$

$$x_{i+1} = 1.6516 - \frac{-0.9948(0.5-1.6516)}{(-1.46--0.9948)} = 4.1142$$

The solution diverges because the secant created by the two *x*-values yields a solution outside the function's domain.

6.30 The equation to be solved is

$$f(h) = \pi R h^2 - \left(\frac{\pi}{3}\right)h^3 - V$$

Because this equation is easy to differentiate, the Newton-Raphson is the best choice to achieve results efficiently. It can be formulated as

$$x_{i+1} = x_i - \frac{\pi R x_i^2 - \left(\frac{\pi}{3}\right)x_i^3 - V}{2\pi R x_i - \pi x_i^2}$$

or substituting the parameter values,

$$x_{i+1} = x_i - \frac{\pi(3) x_i^2 - \left(\frac{\pi}{3}\right)x_i^3 - 30}{2\pi(3) x_i - \pi x_i^2}$$

The iterations can be summarized as

iteration	x_i	$f(x_i)$	$f'(x_i)$	$\lvert\varepsilon_a\rvert$
0	3	26.54867	28.27433	
1	2.061033	0.866921	25.50452	45.558%
2	2.027042	0.003449	25.30035	1.677%
3	2.026906	5.68E-08	25.29952	0.007%

Thus, after only three iterations, the root is determined to be 2.026906 with an approximate relative error of 0.007%.

6.31 The equation can be rearranged so it can be solved with fixed-point iteration as

$$H_{i+1} = \frac{(Qn)^{3/5}(B + 2H_{i+1})^{2/5}}{B\sqrt{S}}$$

Substituting the parameters gives,

$$H_{i+1} = 1.132687(20 + 2H_{i+1})^{2/5}$$

This formula can be applied iteratively to solve for H. For example, using and initioal guess of $H_0 = 0$, the first iteration gives

$$H_1 = 1.132687(20 + 2(0))^{2/5} = 3.754238$$

Subsequent iterations yield

i	H	ε_a
0	0	
1	3.754238	100.000%
2	4.264777	11.971%
3	4.327407	1.447%
4	4.334997	0.175%
5	4.335915	0.021%
6	4.336027	0.003%
7	4.33604	0.000%
8	4.336042	0.000%
9	4.336042	0.000%

Thus, the process converges on a depth of 4.336042.

We can prove that the scheme converges for all initial guesses greater than or equal to zero by differentiating the equation to give

$$g' = \frac{0.906149}{(20 + 2H)^{3/5}}$$

This function will always be less than zero for $H \geq 0$. For example, if $H = 0$, $g' = 0.15017$. Because H is in the denominator, all values greater than zero yield even smaller values. Thus, the convergence criterion that $|g'| < 1$ always holds.