Selected Solutions for Exercises in

Numerical Methods with MATLAB: Implementations and Applications

Gerald W. Recktenwald

Chapter 5

Unavoidable Errors in Computing

5–3 Convert the following numbers to normalized floating-point values with eight-bit mantissas: 0.4, 0.5, 1.5.

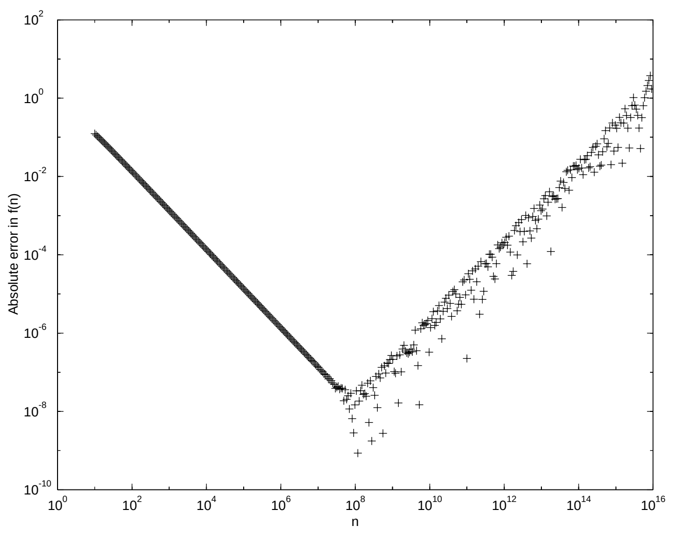
Partial Solution: $0.4 = 0.4 \times 10^0$ is already is already in normalized format. Converting 0.4 to a binary number using Algorithm 5.1 gives $(0\,1100\,1100\,1100\,\ldots)_2$. To eight bits we get $0.4 = (01100110)_2$. Converting $(01100110)_2$ back to base 10 with Equation (5.2) gives

$$\begin{aligned}&0\times 2^{-1}+1\times 2^{-2}+1\times 2^{-3}+0\times 2^{-4}+0\times 2^{-5}+1\times 2^{-6}+1\times 2^{-7}+0\times 2^{-8}\\&\quad=\frac{1}{4}+\frac{1}{8}+\frac{1}{64}+\frac{1}{128}=0.398437500\end{aligned}$$

5–9 Modify the `epprox` function in Listing 5.2 on page 207 so that vectors of `n` and `abs(f-e)` are saved inside the loop. (*Hint*: Create the `nn` and `err` vectors, but do not modify the loop index `n`.) Remove (i.e., comment out) the `fprintf` statements, and increase the resolution of the loop by changing the `for` statement to `for n=logspace(1,16,400)`. Plot the variation of the absolute error with `n`. Do not connect the points with line segments. Rather, use a symbol, such as `'+'` for each point. Explain the behavior of the error for $n > 10^7$.

Partial Solution: The `epproxPlot` function, listed below, contains the necessary modifications. Running `epproxPlot` produces the plot on the following page. Explanation of the plot is left as an exercise to the reader.

```
function epproxPlot
% epproxPlot  Plot error in approximating e = exp(1) with the f(n)
%             f(n) = lim_{n->infinity} (1 + 1/n)^n
%
% Synopsis:  y = epproxPlot
%
% Input:     none
%
% Output:    Plot comparing exp(1) and f(n) = (1 + 1/n)^n as n -> infinity

%  Ref:  N.J. Higham, Accuracy and Stability of Numerical Algorithms,
%        1996, SIAM, section 1.11

e = exp(1);
% fprintf('\n    n          f(n)             error\n');
k=0;
for n=logspace(1,16,400)
   f = (1+1/n)^n;
   % fprintf('%9.1e  %14.10f  %14.10f\n',n,f,abs(f-e));
   k = k + 1;
   nn(k) = n;
   err(k) = abs(f-e);
end
loglog(nn,err,'+');   xlabel('n');   ylabel('Absolute error in f(n)');
[minerr,imin] = min(err);
fprintf('\nMinimum error = %11.3e occurs for n = %24.16e in this sample\n',...
   minerr,nn(imin));
```



Plot of solution to **Exercise 5–9**.

5–19 Implement the combined tolerance $|x_k - x_{k-1}| < \max\left[\Delta_a,\ \delta_r|x_{k-1}|\right]$ in the `newtsqrt` function in Listing 5.3. Note that Δ_a and δ_r should be *separate* inputs to your `newtsqrt` function. As a diagnostic, print the number of iterations necessary to achieve convergence for each argument of `newtsqrt`. Based on the results of Example 5.6, what are good default values for Δ_a and δ_r? If $\Delta_a = \delta_r$, is the convergence behavior significantly different than the results of Example 5.6? Is the use of an absolute tolerance helpful for this algorithm?

Partial Solution: The MATLAB part of the solution is implemented in the `newtsqrt` and `testConvSqrt` functions listed below. The `newtsqrt` function is a modified version of the `newtsqrtBlank.m` file in the `errors` directory of the NMM Toolbox. The modified file is saved as `newtsqrt.m`. The `testConvSqrt` function is a modified version of `testsqrt` function in Listing 5.4.

The inputs to `newtsqrt` are `x`, the argument of the square root function (`newtsqrt` returns an approximation to $\sqrt{\texttt{x}}$), `deltaA`, the absolute tolerance, `deltaR`, the relative tolerance, and `maxit`, the maximum number of iterations. Note that the `newtsqrt` function has two input tolerances, whereas the `newtsqrt` function in Listing 5.3 has only one tolerance (`delta`). The default tolerances in `testConvSqrt` are

$$\texttt{deltaA} = 5\varepsilon_m = 1.11 \times 10^{-15}$$

$$\texttt{deltaR} = 5 \times 10^{-6}$$

The absolute tolerance is designed prevent unnecessary iterations: values of $\texttt{r} - \texttt{rold}$ less than ε_m are meaningless due to roundoff. The relative tolerance corresponds to a change in the estimate of the root of less than 0.05 percent (0.0005) on subsequent iterations.

```
function testConvSqrt(deltaA,deltaR)
% testConvSqrt  Test alternative convergence criteria for the newtsqrt function
%
% Synopsis:  testConvSqrt
%            testConvSqrt(deltaA)
%            testConvSqrt(deltaA,deltaR)
%
% input:  deltaA = (optional) absolute convergence criterion
%                  Default:  deltaA = 5e-9
%         deltaR = (optional) relative convergence criterion
%                  Default:  deltaR = 5e-6

if nargin<1,  deltaA = 5*eps;  end
if nargin<2,  deltaR = 5e-6;  end

xtest = [4  0.04  4e-4  4e-6  4e-8  4e-10  4e-12];   % arguments to test

fprintf('\n Combined Convergence Criterion\n');
fprintf('\t\t deltaA = %12.4e\t\t deltaR = %12.4e\n\n',deltaA,deltaR)
fprintf('    x         sqrt(x)   newtsqrt(x)    error       relerr     it\n');

for x=xtest         %  repeat for each column in xtest
   r = sqrt(x);
   [rn,it] = newtsqrt(x,deltaA,deltaR);
   err = abs(rn - r);
   relerr = err/r;
   fprintf('%10.3e  %10.3e  %10.3e  %10.3e  %10.3e  %4d\n',x,r,rn,err,relerr,it)
end
```

```
function [r,iter] = newtsqrt(x,deltaA,deltaR,maxit)
% newtsqrt   Use Newton's method to compute the square root of a number
%
% Synopsis:  r = newtsqrt(x,delta,maxit)
%            [r,it] = newtsqrt(x,delta,maxit)
%
% Input:  x     = number for which the square root is desired
%         delta = (optional) convergence tolerance.  Default: delta = 5e-9
%         maxit = (optional) maximum number of iterations.  Default: maxit = 25
%
% Output:    r = square root of x to within combined tolerance
%                 abs(r-rold) < max( deltaA, deltaR*abs(rold) )
%                where r is the current guess at the root and rold
%                is the guess from the previous iteration
%            it = (optional) number of iterations to reach convergence

if x<0,  error('Negative input to newtsqrt not allowed');  end
if x==0,      r=x;  return;   end
if nargin<2,  deltaA = 5*eps;  end
if nargin<3,  deltaR = 5e-6;  end
if nargin<4,  maxit=25;       end

r = x/2;  rold = x;   %  Initialize, make sure convergence test fails on first try
it = 0;
while abs(r-rold) > max( [deltaA, deltaR*abs(rold)] ) & it<maxit  %  Convergence test
   rold = r;                        %  Save old value for next convergence test
   r = 0.5*(rold + x/rold);         %  Update the guess
   it = it + 1;
end
if nargout>1,  iter=it;  end
```

The reader is left with the task of running `testConvSqrt` and discussing the results. What happens if the call to `testConvSqrt` is `testConvSqrt(5e-9)`? Which tolerance, the absolute or the relative (or both, or neither) is more effective at detecting convergence?

5–23 Use the `sinser` function in Listing 5.5 to evaluate $\sin(x)$, for $x = \pi/6$, $5\pi/6$, $11\pi/6$, and $17\pi/6$. Use the periodicity of the sine function to modify the `sinser` function to avoid the undesirable behavior demonstrated by this sequence of arguments. (*Hint*: The modification does not involve any change in the statements inside the `for...end` loop.)

Partial Solution: The output of running `sinser` for $x = 17\pi/16$ is

```
>> sinser(17*pi/6)
Series approximation to sin(8.901179)

  k      term        ssum
  1    8.901e+00    8.90117919
  3   -1.175e+02  -108.64036196
  5    4.656e+02  357.00627682
  7   -8.784e+02  -521.41383256
  9    9.666e+02  445.22638523
 11   -6.963e+02  -251.02690828
 13    3.536e+02  102.59385039
 15   -1.334e+02  -30.82387869
 17    3.886e+01    8.03942600
 19   -9.003e+00   -0.96401885
 21    1.698e+00    0.73443795
 23   -2.659e-01    0.46848851
 25    3.512e-02    0.50360758
 27   -3.964e-03    0.49964387
 29    3.868e-04    0.50003063

Truncation error after 15 terms is 3.06329e-05

ans =
    0.5000
```

Due to the large terms in the numerator, the value of the series first grows before converging toward the exact value of 0.5. Since the large terms alternate in sign, there is a loss of precision in the least significant digits when the large terms with opposite sign are added together to produce a small result. Thus, even if more terms are included in the series, it is not possible to obtain arbitrarily small errors. Evaluating the series for the sequence $x = \pi/6$, $5\pi/6$, $11\pi/6$, $17\pi/6, \ldots$ requires an increasing number of terms, even though $|\sin(x)| = 1/2$ for each x in the sequence.

The results of running `sinser` for $x = \pi/6$, $5\pi/6$, $11\pi/6$, and $17\pi/6$ are summarized in the table below.

x	Terms	Absolute Error
$\pi/6$	6	3.56×10^{-14}
$5\pi/6$	10	1.16×10^{-11}
$11\pi/6$	15	4.40×10^{-11}
$17\pi/6$	15	3.06×10^{-05}

For smaller values of x, the series converges with fewer terms. The error in the approximation to $\sin(x)$ grows as $|x|$ increases.

To reduce the effect of roundoff for large x we exploit the periodicity of $\sin(x)$: the value of $\sin(x)$ is the same for $x \pm 2n\pi$ for $n = 1, 2, \ldots$. Thus, whenever $|x| > 2\pi$ we can remove an integer factor of 2π before evaluating the series. Removing an integer factor of 2π is easily achieved with the built-in `rem` function. `rem(a,b)` returns the remainder (fractional part) of a/b. To apply this to the `sinser` code, insert

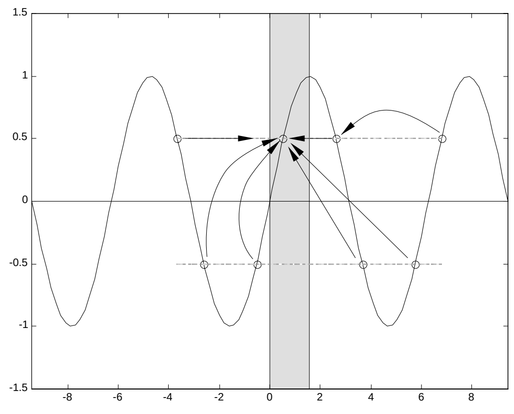
```
x = rem(x,2*pi);
```

before the evaluation of the series. This fix is implemented in `sinserMod` (not listed here). Running `sinserMod` for $x = 11\pi/6$ gives

```
>> s = sinserMod(17*pi/6)
Series approximation to sin(2.617994)

  k       term          ssum
  1    2.618e+00    2.61799388
  3   -2.991e+00   -0.37258065
  5    1.025e+00    0.65227309
  7   -1.672e-01    0.48502935
  9    1.592e-02    0.50094978
 11   -9.920e-04    0.49995780
 13    4.358e-05    0.50000139
 15   -1.422e-06    0.49999996
 17    3.584e-08    0.50000000
 19   -7.183e-10    0.50000000

Truncation error after 10 terms is 1.15644e-11

s =
    0.5000
```

The `sinserMod` function obtains a more accurate result than `sinser`, and it does so by evaluating fewer terms.

A even better solution involves recognizing that $\sin(x)$ for all x can be constructed from a table of $\sin(x)$ values for $0 < x < \pi/2$. The diagram below indicates how values of $\sin(x)$ versus x can be used to generate $\sin(x)$ values for any x.



Exercise 5–23. All values of $\sin(x)$ can be generated from $\sin(x)$ versus x in $0 \le x \le \pi/2$.

The following table gives additional clues on how to implement a more efficient and more accurate `sinser`

Range of x	Simplification
$\frac{\pi}{2} < x < \pi$	$\sin(x) = \sin(\pi - x)$
$\pi < x < \frac{3\pi}{2}$	$\sin(x) = -\sin(x - \pi) = \sin(x - \pi)$
$\frac{3\pi}{2} < x < 2\pi$	$\sin(x) = -\sin(2\pi - x) = \sin(x - 2\pi)$

The range of $x < 0$ is handled with the identity $\sin(x) = -\sin(x)$.

The following MATLAB code provides additional hints on the implementation.

```
x = rem(x,2*pi);
if x==0
  ssum = 0;    return;
elseif x==pi/2
  ssum = sign(xin);    return;
elseif  x>pi/2 & x<pi
  x = pi - x;
...
```

The goal of the more sophisticated code is to use the absolute minimum number of terms in the series approximation. Reducing the number of terms makes the computation more efficient, but more importantly, it reduces the roundoff.

The improved code is contained in the `sinserMod2` function, which is not listed here. The `demoSinserMod2` function compares the results of `sinserMod` and `sinserMod2` for $x = \pi/6$, $5\pi/6$, $11\pi/6$, and $17\pi/6$. Running `demoSinserMod2` gives

```
>> demoSinserMod2

           -- sinserMod --      -- sinserMod2 --
  x*6/pi     n        err        n2        err2
  -17.00    10     1.156e-11     6     3.558e-14
  -11.00    15    -4.402e-11     6    -3.558e-14
   -5.00    10     1.156e-11     6     3.564e-14
   -1.00     6     3.564e-14     6     3.564e-14
    1.00     6    -3.564e-14     6    -3.564e-14
    5.00    10    -1.156e-11     6    -3.564e-14
   11.00    15     4.402e-11     6     3.558e-14
   17.00    10    -1.156e-11     6    -3.558e-14
```

For the same tolerance (5×10^{-9}) `sinserMod2` obtains a solution that is as or more accurate than `sinserMod`, and it does so in fewer iterations. For $x = \pm 5\pi/6$ the two functions are identical. Complete implementation and testing of `sinserMod` and `sinserMod2` is left to the reader.

5–27 Derive the Taylor series expansions $P_1(x)$, $P_2(x)$, and $P_3(x)$ in Example 5.9 on page 222.

Partial Solution: The Taylor polynomial is given by Equation (5.15)

$$P_n(x) = f(x_0) + (x - x_0)\,\left.\frac{df}{dx}\right|_{x=x_0} + \frac{(x-x_0)^2}{2}\,\left.\frac{d^2f}{dx^2}\right|_{x=x_0} + \cdots + \frac{(x-x_0)^n}{n!}\,\left.\frac{d^nf}{dx^n}\right|_{x=x_0}$$

Given $f(x) = 1/(1-x)$ we compute

$$\frac{df}{dx} = \frac{+1}{(1-x)^2} \qquad \frac{d^2}{dx^2} = \frac{+2}{(1-x)^3} \qquad \frac{d^3}{dx^3} = \frac{+6}{(1-x)^4} \qquad \frac{d^n}{dx^n} = \frac{n!}{(1-x)^{n+1}}$$

Thus

$$P_1(x) = f(x_0) + (x - x_0)\frac{1}{(1-x_0)^2}$$