CSC336 Tutorial 2 – Computer arithmetic

QUESTION 1 *Find the positive numbers in $\mathcal{R}_2(4,3)$ assuming normalized mantissa.*
ANSWER: The numbers in $\mathcal{R}_2(4,3)$ are of the form $f \times 2^e$, where f the mantissa and e the exponent. The table below indicates all possible positive (and normalized) values of the mantissa f, all possible values for the exponent e, and the numbers of the form $f \times 2^e$.
Notes:

1. (1) does not need to be stored since we assume normalized mantissa.
2. The $4th$ bit of the mantissa is used for the sign. $\mathcal{R}_2(4,3)$ also includes the respective negative numbers, as well as 0, which is handled as a special number.

3. Figure 1 shows the position of the numbers on the axis. Numbers get denser towards zero, while numbers with the same exponent are uniform.

Table 1: $f \times 2^e$

	e	$-(11)_2$	$-(10)_2$	$-(01)_2$	$-(00)_2$	$(01)_2$	$(10)_2$	$(11)_2$
		-3	-2	-1	0	1	2	3
	2^e	1/8	1/4	1/2	1	2	4	8
f								
$(.(1)000)_2$	8/16	1/16	1/8	1/4	1/2	1	2	4
$(.(1)001)_2$	9/16	9/128	9/64	9/32	9/16	9/8	9/4	9/2
$(.(1)010)_2$	10/16	5/64	5/32	5/16	5/8	5/4	5/2	5
$(.(1)011)_2$	11/16	11/128	11/64	11/32	11/16	11/8	11/4	11/2
$(.(1)100)_2$	12/16	3/32	3/16	3/8	3/4	3/2	3	6
$(.(1)101)_2$	13/16	13/128	13/64	13/32	13/16	13/8	13/4	13/2
$(.(1)110)_2$	14/16	7/64	7/32	7/16	7/8	7/4	7/2	7
$(.(1)111)_2$	15/16	15/128	15/64	15/32	15/16	15/8	15/4	15/2

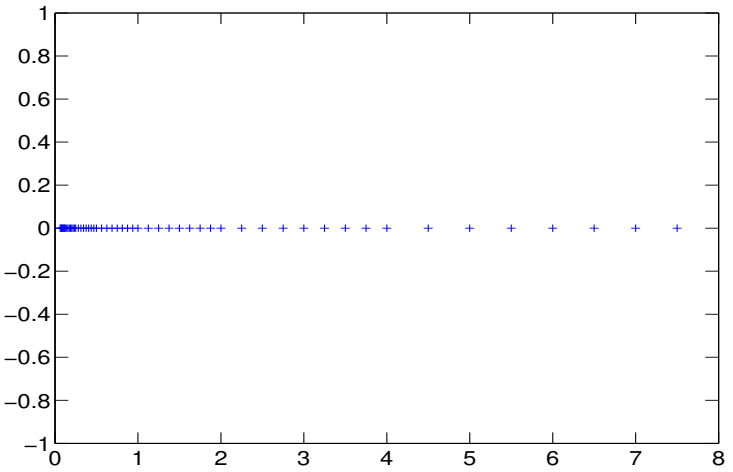

Figure 1: Positive numbers in $\mathcal{R}_2(4,3)$

QUESTION 2 *Let x be a number and let $fl(x)$ be its floating-point representation with t-digit mantissa, in base b. Prove that the following bounds for the relative round-off error $\delta = (x - fl(x))/x$ hold:*
(i) chopping and normalized: $|\delta| \leq b^{1-t}$.
(ii) traditional rounding and normalized: $|\delta| \leq \frac{1}{2} b^{1-t}$.
PROOF:
(i) Case $x > 0$, i.e. $x \geq fl(x)$, therefore, $|\delta| = \delta = (x - fl(x))/x$

$$
\begin{aligned}
|\delta| &= \frac{.d_1 d_2 \ldots d_t d_{t+1} \ldots \times b^e - .d_1 d_2 \ldots d_t \times b^e}{.d_1 d_2 \ldots d_t d_{t+1} \ldots \times b^e} \\
&= \frac{.d_{t+1} d_{t+2} \ldots \times b^{-t}}{.d_1 d_2 \ldots} \\
&\leq \frac{.d_{t+1} \ldots \times b^{-t}}{.100\ldots} \qquad \text{[Note: } \textbf{.100...} \text{ is the smallest normalized number]} \\
&\leq \frac{.aa \ldots \times b^{-t}}{.1} \qquad \text{where } a = b-1 \\
&= \frac{1 \times b^{-t}}{.1} \quad (*) \\
&= b^{1-t}
\end{aligned}
$$

(ii) Case $x > 0, d_{t+1} < b/2$, i.e. $x \geq fl(x)$, therefore, $|\delta| = \delta = (x - fl(x))/x$

$$\begin{aligned} |\delta| &= \frac{.d_1d_2\ldots d_td_{t+1}\ldots \times b^e - .d_1d_2\ldots d_t \times b^e}{.d_1d_2\ldots d_td_{t+1}\ldots \times b^e} \\ &= \frac{.d_{t+1}d_{t+2}\ldots \times b^{-t}}{.d_1d_2\ldots} \\ &\leq \frac{.caa\ldots \times b^{-t}}{.1} \quad \text{where } c = b/2 - 1 \\ &= \frac{1}{2}b^{1-t} \quad (**) \end{aligned}$$

Case $x > 0, d_{t+1} \geq b/2$ i.e. $x < fl(x)$, therefore, $|\delta| = -\delta = -(x - fl(x))/x$

$$\begin{aligned} |\delta| &= \frac{fl(x) - x}{x} \\ &= \frac{.d_1d_2\ldots(d_t+1) \times b^e - .d_1d_2\ldots d_td_{t+1}\ldots \times b^e}{.d_1d_2\ldots d_td_{t+1}\ldots \times b^e} \\ &= \frac{b^{-t} + .d_1d_2\ldots d_t - .d_1d_2\ldots d_td_{t+1}\ldots}{.d_1d_2\ldots} \\ &\leq \frac{b^{-t} + .d_1d_2\ldots d_t - .d_1d_2\ldots d_tc}{.d_1d_2\ldots} \quad \text{where } c = b/2 \end{aligned}$$

$$\begin{aligned} &= \frac{b^{-t} - cb^{-t-1}}{.1} \\ &= b^{1-t} - \frac{b}{2}b^{-t} \\ &= b^{1-t} - \frac{1}{2}b^{1-t} \\ &= \frac{1}{2}b^{1-t} \end{aligned}$$

The proof for $x < 0$ is similar.

Notes: $\Sigma = \alpha + \alpha w + \alpha w^2 + \ldots + \alpha w^\nu = \frac{\alpha(w^{\nu+1}-1)}{w-1}, \Sigma_{[1,\infty),w<1} = \frac{\alpha}{1-w}$

Therefore:

(*) $(.aa\ldots) = \sum_{i=1}^{\infty} ab^{-i} = a\frac{b^{-1}}{1-b^{-1}} = (b-1)\frac{1}{b-1} = 1$

(**) by (*), $.0aa\ldots = \frac{1}{b}(.aa\ldots) = \frac{1}{b} = b^{-1}$

$.caa\ldots = .0aa\ldots + .c = b^{-1} + (\frac{b}{2} - 1)b^{-1} = \frac{b}{2}b^{-1} = \frac{1}{2}$

QUESTION 3 *Find ϵ_{mach} for a floating-point system with t binary digits precision and (i) traditional rounding or (ii) proper rounding. (Answer in terms of t.)*

ANSWER: Recall that ϵ_{mach} is the smallest (non-normalised) floating-point number with the property $1 + \epsilon_{mach} > 1$. The floating-point representation of "1" (with binary mantissa) is $1 = 0.100\cdots0 \times 2^1$. We construct ϵ_{mach} with the smallest possible mantissa and the smallest possible exponent, so that $1 + \epsilon_{mach} > 1$ holds.

(i) traditional rounding: Consider the operations $1+2^{-t}$ and $1+2^{-t-1}$ (mantissa shown in binary):

	$0.100\cdots000 \times 2^1$		$0.100\cdots000 \times 2^1$
+	$0.000\cdots010 \times 2^1$	+	$0.000\cdots001 \times 2^1$ where we (temporarily) show $t+2$ bits
=	$0.100\cdots010 \times 2^1$	=	$0.100\cdots001 \times 2^1$

in the mantissa.

Note that $0.100\cdots010 \times 2^1$ is rounded up to $0.100\cdots100 \times 2^1 = 0.100\cdots1 \times 2^1 > 1$ (the $t+1$st digit is $b/2$, so round-up), while $0.100\cdots001 \times 2^1$ is rounded down to $0.100\cdots0 \times 2^1 = 1$ (the $t+1$st digit is 0, so less than $b/2$, so round-down). Thus, $\epsilon_{mach} = 0.000\cdots010 \times 2^1 = 0.000\cdots1 \times 2^0 = 2^{-t}$.

(ii) proper rounding: Consider the operations $1 + 2^{-t+1}$ and $1 + 2^{-t}$ (mantissa shown in binary):

	$0.100\cdots000 \times 2^1$		$0.100\cdots000 \times 2^1$
+	$0.000\cdots100 \times 2^1$	+	$0.000\cdots010 \times 2^1$ where we (temporarily) show $t+2$ bits
=	$0.100\cdots100 \times 2^1$	=	$0.100\cdots010 \times 2^1$

in the mantissa.

Note that $0.100\cdots100 \times 2^1 = 0.100\cdots1 \times 2^1 > 1$ (remains as is, no round-up or down), while $0.100\cdots010 \times 2^1$ is rounded down to $0.100\cdots0 \times 2^1 = 1$ (the $t+1$st digit is $b/2$; if rounded-up we end with an odd digit, so it is rounded-down). Thus, $\epsilon_{mach} = 0.000\cdots100 \times 2^1 = 0.000\cdots10 \times 2^0 = 2^{-t+1}$.

QUESTION 4 *Find ϵ_{mach} for a binary floating-point system with proper rounding, without knowing the number t of digits in the mantissa. (Essentially, find t.)*

ANSWER: Consider the code (written in MATLAB for convenience)

```
x = 1; i = 0;
while (1+x > 1)
    x = x/2; i = i+1;
end
me = 2*x; fprintf(' machine epsilon = %22.15e t = %d\n', me, i)
```

This code goes through the numbers $x = 1, (0.1)_2 = 2^{-1}, (0.01)_2 = 2^{-2}, \ldots$, up to the point that $1 + x = 1$, where $x = 2^{-i}$, and outputs $2x$, i.e. 2^{-i+1}, which is the smallest x for which $1 + x > 1$. Thus $\epsilon_{mach} = 2^{-i+1}$. The code also outputs (the last) i.

In a previous discussion we showed that ϵ_{mach} for proper rounding is 2^{-t+1}, where t is the number of digits in the mantissa. Thus we can deduce that the number t of digits in the mantissa is i as output by the code.

If we run this code in MATLAB we get as output

```
machine epsilon = 2.220446049250313e-016 t = 53.
```

Note that this corresponds to double precision IEEE standard (as given in the notes). In general, MATLAB uses double precision.

Note also, that MATLAB has a predefined variable `eps` which gives the value of the machine epsilon, so we need not run the code to find it. We can, though, check that MATLAB's `eps` and `me` from the above code have the same value, 2^{-52}.

In C, C++, fortran or other languages, we can run a similar code using either floats (reals) or doubles, and find ϵ_{mach} for single or double precision, respectively. (If traditional rounding is used, we need to adjust the code accordingly.)

QUESTION 5 *A computer system uses decimal floating-point arithmetic with 3 digits mantissa (plus one digit for the mantissa sign) and 1 digit exponent (plus one digit for the exponent sign). All computer operations return the correctly rounded result (i.e. the number closest to the correct answer in the representation being used). Compute the results of the following arithmetic operations on the above computer system. Indicate when overflow or underflow occurs. Let* $a = 0.697$ *and* $b = 0.699$.

$$
\begin{aligned}
(336 + 1.42) + 0.1 &= 337.42 + 0.1 && \rightarrow 337 + 0.1 \\
&= 337.1 && \rightarrow 337 = .337 \times 10^3 \\
336 + (1.42 + 0.1) &= 336 + 1.52 && \rightarrow 336 + 1.52 \\
&= 337.52 && \rightarrow 338 = .338 \times 10^3 \\
336 \cdot 10^7 &= 336 \cdot 10^7 && \rightarrow .336 \times 10^{10} \text{ overflow} \\
-900 \cdot 10^6 - 100 \cdot 10^6 &= -1000 \cdot 10^6 && \rightarrow -.100 \times 10^{10} \text{ overflo} \\
10^8 + 10^5 &= 1001 \cdot 10^5 = .1001 \cdot 10^9 && \rightarrow .100 \times 10^9 \ (= 10^8) \\
(a + b)/2 = (0.697 + 0.699)/2 &= 1.396/2 && \rightarrow 1.40/2 = 0.7 = 0.700 \\
a + (b - a)/2 = 0.697 + (0.699 - 0.697)/2 &= 0.697 + 0.002/2 && \rightarrow 0.697 + (0.2 \times 10^{-2})/2 \\
&= 0.697 + 0.001 = 0.698 && \rightarrow 0.698 \times 10^0
\end{aligned}
$$

Notes:

For each individual mathematical operation, we first compute the correct mathematical result (indicated by the $=$ sign), then show the computational result as represented in the particular computer system given (indicated by the $\rightarrow$ sign), by applying rounding (whenever needed).

In showing each computational result we take into account that the respective mathematical result is written by possibly shifting the mantissa digits and adjusting the exponent accordingly (though, in some obvious cases, we do not explicitly give this presentation of the result).

We can also use the fl notation to show the computed result for the above operations. This notation is more formal, but possibly a bit cumbersome. For example, for the operations $(336 + 1.42) + 0.1$, we have

$$
\begin{aligned}
fl(fl(fl(336) + fl(1.42)) + fl(0.1)) &= fl(fl(336 + 1.42) + 0.1) && \text{representation} \\
&= fl(fl(337.42) + 0.1) && \text{operation} \\
&= fl(337 + 0.1) && \text{representation} \\
&= fl(337.1) && \text{operation} \\
&= 337 = .337 \times 10^3 && \text{representation}
\end{aligned}
$$

Note that the exact mathematical result is 337.42 (which, if rounded, gives 337).

If we change 336 to 336.1, and do the operations $(336.1 + 1.42) + 0.1$, we have

$$
\begin{aligned}
fl(fl(fl(336.1) + fl(1.42)) + fl(0.1)) &= fl(fl(336 + 1.42) + 0.1) && \text{representation} \\
&= fl(fl(337.42) + 0.1) && \text{operation} \\
&= fl(337 + 0.1) && \text{representation} \\
&= fl(337.1) && \text{operation} \\
&= 337 = .337 \times 10^3 && \text{representation}
\end{aligned}
$$

Note that the exact mathematical result is 337.52 (which, if rounded, gives 338).

By comparing the two examples ($(336 + 1.42) + 0.1$ and $(336.1 + 1.42) + 0.1$), we see that it is not enough to do all operations mathematically, then take the floating-point respresentation of the final result. We need to take the floating-point respresentation of each individual number or result, before each operation takes place. In the $(336 + 1.42) + 0.1$ case, there is no initial representation error (no need to round), while, in the $(336.1 + 1.42) + 0.1$ case, there is some initial rounding error.

QUESTION 6 *Examples of catastrophic cancellation and how to avoid it.*

(a) Solve the quadratic equation $x^2-56x+1=0$, using 5-decimal-digits floating-point arithmetic.

$$\begin{aligned} x_1 &= 28+\sqrt{783} = 28+27.982 = 55.982 = 0.55982\times 10^2 \\ x_2 &= 28-\sqrt{783} = 28-27.982 = .018 = .18\times 10^{-1} \\ x_{1(exact)} &= 55.982137\ldots \\ x_{2(exact)} &= .0178628\ldots, \quad x_2 \text{ only 2 digits correct} \end{aligned}$$

The root x_2 is only 2 digits correct due to cancellation between 28 and $\sqrt{783}$.
A better way to compute the root x_2 is to first compute x_1 (which does not suffer from cancellation) and then let $x_2 = 1/x_1 = .017863$. Thus we have 5 digits correct.

(b) Consider computing $y=\sqrt{x+\delta}-\sqrt{x}$, when $|\delta| \ll x, x>0$.
Here we have again subtraction of nearly equal numbers. To avoid subtraction of nearly equal numbers, compute y by $\dfrac{x+\delta-x}{\sqrt{x+\delta}+\sqrt{x}} = \dfrac{\delta}{\sqrt{x+\delta}+\sqrt{x}}$.

(c) Consider computing $y=\cos(x+\delta)-\cos(x)$, when $|\delta| \ll x$.
Again we have subtraction of nearly equal numbers. To avoid subtraction of nearly equal numbers, compute y by $y=-2\sin\frac{\delta}{2}\sin(x+\frac{\delta}{2})$.

QUESTION 7 *Study the condition number of the function* $f(x)=\frac{10}{1-x^2}$.

The condition number of $f(x)$ is $\kappa_f = |\dfrac{xf'(x)}{f(x)}| = |\dfrac{x\frac{-2x\cdot 10}{(1-x^2)^2}}{\frac{10}{1-x^2}}| = |\dfrac{-2x^2}{1-x^2}|$

When is κ_f large? We need to check the cases that the numerator in κ_f is large and the denominator is small.

Study the case when the numerator is large, that is, $|x|$ is large.

We have $\dfrac{-2x^2}{1-x^2} \rightarrow_{x\rightarrow\pm\infty} = \dfrac{-4x}{-2x} = 2$ (by de l'Hospital's rule), thus $\kappa_f \rightarrow 2$ which is small.

Study the case when the denominator is small, that is, $|x| \rightarrow 1$.

We have $\dfrac{-2x^2}{1-x^2} \rightarrow_{|x|\rightarrow 1} \pm\infty$

So $\kappa_f \rightarrow \infty$, thus the condition number will be very large if $|x| \rightarrow 1$.

See Figure 2, where the solid line is $\kappa_f = |\frac{-2x^2}{1-x^2}|$ versus x, and the dashed line is $\frac{-2x^2}{1-x^2}$ versus x.

```
x = linspace(-2, 2, 1000);
kf = abs(2*x.^2./(1-x.^2));
kf1 = -2*x.^2./(1-x.^2);
plot(x, kf, 'r-', x, kf1, 'g--');
hax = gca; set(hax, 'FontSize', 16, 'TickLength', [0.02 0.05])
print -depsc testcondf.m1.eps
```


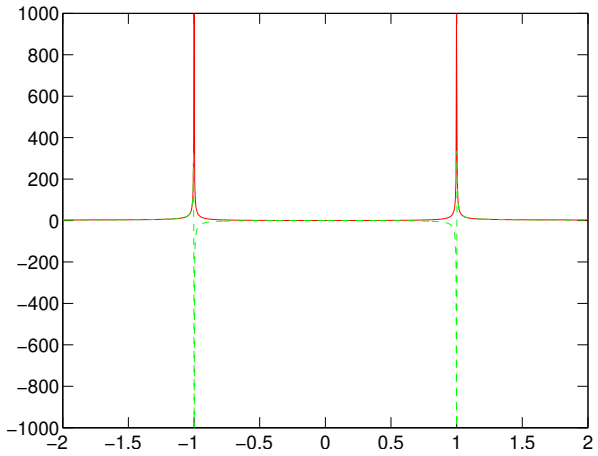

Figure 2: Condition number of $f(x)=\frac{10}{1-x^2}$

Now consider $f(x)=\dfrac{10}{1-x^2}$ again, and give examples of what it practically means to have a large condition number when $|x| \rightarrow 1$.

Let $x=1+\delta$, for $\delta=-10^{-1},\ldots,-10^{-3},10^{-3},\ldots,10^{-1}$. Let $y=x+10^{-5}$. Compute $\kappa_f(x)$, and $\dfrac{f(x)-f(y)}{f(x)}$. In this way, we perturb x a little (absolute perturbation 10^{-5}, relative perturbation $10^{-5}/x$), and study the relative perturbation $\dfrac{f(x)-f(y)}{f(x)}$ in $f(x)$.

Theoretically we expect $\dfrac{\frac{f(x)-f(y)}{f(x)}}{\frac{x-y}{x}} \approx \kappa_f(x)$. That is, when x is close to 1, since $\kappa_f(x)$ is large, for a small relative perturbation $10^{-5}/x$ in x, the relative perturbation $\dfrac{f(x)-f(y)}{f(x)}$ is expected to be large.

Figure 3 as well as the output of the code verifies that $\dfrac{\frac{f(x)-f(y)}{f(x)}}{\frac{x-y}{x}}$ behaves approximately as κ_f. The solid line is $\kappa_f(x)$ versus x, and the dashed line is $\dfrac{\frac{f(x)-f(y)}{f(x)}}{\frac{x-y}{x}}$ versus x. Notice that we study only the $[1-10^{-1},1+10^{-1}]$ range of x (but we expect similar behaviour everywhere).

```
d = [-10.^[-1:-1:-3] 10.^[-3:-1]]; e = 1e-5;
fprintf(' errx      errf      rerrx     rerrf   rerrf/rerrx condf\r
x = 1+d; x2 = x.^2;
y = x+e; y2 = y.^2;
f = 10./(1-x2);
g = 10./(1-y2);
errx = x - y; rerrx = errx./x;
errf = f - g; rerrf = errf./f;
kf = abs(2*x2./(1-x2));
ratio = abs(rerrf./rerrx);
plot(1+d, kf, 'r-', 1+d, ratio, 'b--')
hax = gca; set(hax, 'FontSize', 16, 'TickLength', [0.02 0.05])
axis tight
print -depsc testcondf.m2.eps
for i = 1:length(d);
    fprintf('%9.2e %9.2e %9.2e %9.2e %9.2e %9.2e\n', ...
    errx(i), errf(i), rerrx(i), rerrf(i), rerrf(i)/rerrx(i), kf(i))
end
```



```
 errx      errf      rerrx     rerrf   rerrf/rerrx condf
-1.00e-05 -4.99e-03 -1.11e-05 -9.47e-05  8.53e+00  8.53e+00
-1.00e-05 -5.00e-01 -1.01e-05 -9.96e-04  9.86e+01  9.85e+01
-1.00e-05 -5.05e+01 -1.00e-05 -1.01e-02  1.01e+03  9.99e+02
-1.00e-05 -4.95e+01 -9.99e-06  9.91e-03 -9.92e+02  1.00e+03
-1.00e-05 -4.99e-01 -9.90e-06  1.00e-03 -1.01e+02  1.02e+02
-1.00e-05 -4.99e-03 -9.09e-06  1.05e-04 -1.15e+01  1.15e+01
```


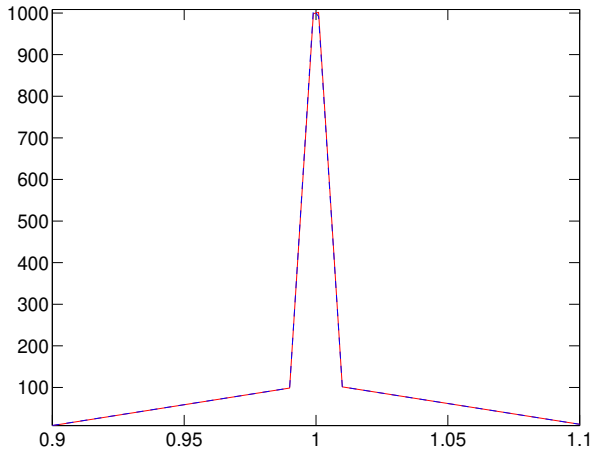

Figure 3: $\frac{\frac{f(x)-f(y)}{f(x)}}{\frac{x-y}{x}} \approx \kappa_f(x)$

QUESTION 8 *Study three ways to approximate e^x. Consider the Taylor's series $e^x = 1 + x + \frac{x^2}{2!} + \cdots$, and compute the above sum from left to right up to saturation (i.e. up to the point where the sum does not change), using three techniques:*
(a) compute the sum from left to right up to saturation as given,
(b) compute the sum from left to right up to saturation using the fact that each term added is the previous term times $\frac{x}{k}$, and
(c) do as in (b), but using the absolute value of x, then, if $x \geq 0$ return the result as is, while if $x < 0$ return the inverse of the result (taking into account that $e^{-x} = \frac{1}{e^x}$.
Compute and tabulate the error in each case and discuss advantages and disadvantages of each approximation. Assume that the function $\exp(x)$ returns the exact value of e^x.
ANSWER: Here is the code for the three approximations to e^x, the main script and the results.

```
% function [y, k] = exp0(x)
% e^x ~ 1 +x +x^2/2! +x^3/3! +
%
function [y, k] = exp0(x)
sump = 0;
sumn = 1;
k = 0;

while sump ~= sumn
    sump = sumn;
    k = k+1;
    term = x^k/factorial(k);
    sumn = sumn + term;
end
y = sumn;
```

```
% function [y, k] = exp1(x)
% e^x ~ 1 +x +x^2/2! +x^3/3! + ..
% using nested multiplication
function [y, k] = exp1(x)
sump = 0;
sumn = 1;
k = 0;
term = 1;
while sump ~= sumn
    sump = sumn;
    k = k+1;
    term = term * x / k;
    sumn = sumn + term;
end
y = sumn;
```

Note the difference between how `term` is computed in the two functions. The function to the left (`exp0`) requires exponention and a call to `factorial`, while the one to the right (`exp1`), requires only multiplication instead.


```
% function [y, k] = exp2(x)
% e^x ~ 1 +x +x^2/2! +x^3/3! + ...
% using nested mult. if x >= 0,
% and e^x ~ 1/e^{-x} if x < 0
function [y, k] = exp0(x)
sump = 0;
sumn = 1;
k = 0;
term = 1;
ax = abs(x);
while sump ~= sumn
    sump = sumn;
    k = k+1;
    term = term * ax / k;
    sumn = sumn + term;
end
if (x >= 0) y = sumn;
else y = 1/sumn; end
```

```
% script for testing various
% approximations to e^x, x = -22:22
k = 0;
for x = -22:22
    k = k + 1;
    err0(k) = (exp(x)-exp0(x))/exp(x);
    err1(k) = (exp(x)-exp1(x))/exp(x);
    err2(k) = (exp(x)-exp2(x))/exp(x);
    fprintf('%4d %10.2e %10.2e %10.2e\n
        x, err0(k), err1(k), err2(k));
end
```

The function to the left (`exp2`) is the same as `exp1`, except that `exp2` works on $|x|$ instead of x, and the case $x < 0$ is handled by $1/e^{|x|}$.


```
-22  1.2e+02  8.1e+00  1.9e-16
-21 -3.5e+01  5.2e+00 -2.7e-16
-20 -1.0e+00 -1.7e+00 -2.0e-16
-19  5.4e-01 -5.8e-01  1.5e-16
-18 -4.9e-02 -1.0e-02  2.2e-16
-17 -1.0e-03  4.1e-03  8.0e-16
-16 -2.9e-04 -5.9e-05  0.0e+00
-15 -1.0e-05 -2.3e-05  3.5e-16
-14  8.6e-06 -4.3e-07  3.8e-16
-13  1.3e-06 -1.6e-06  0.0e+00
-12 -6.1e-08 -1.6e-07 -4.1e-16
-11 -7.6e-08 -2.8e-08  0.0e+00
-10  7.2e-09  3.1e-09 -3.0e-16
 -9  5.5e-10  1.4e-10  2.2e-16
 -8  1.5e-10  6.6e-12 -1.6e-16
 -7 -1.3e-11  1.8e-11 -8.3e-16
 -6  7.3e-13 -7.1e-13  0.0e+00
 -5 -2.1e-13  2.1e-13 -3.9e-16
 -4 -1.4e-14 -1.7e-14  3.8e-16
 -3 -8.4e-16 -1.7e-15 -2.8e-16
 -2 -4.1e-16 -4.1e-16 -2.1e-16
 -1 -3.0e-16 -3.0e-16  1.5e-16
  0  0.0e+00  0.0e+00  0.0e+00
```

```
  1  0.0e+00  0.0e+00  0.0e+00
  2  2.4e-16  2.4e-16  2.4e-16
  3  3.5e-16  3.5e-16  3.5e-16
  4 -5.2e-16 -5.2e-16 -5.2e-16
  5  1.9e-16  3.8e-16  3.8e-16
  6  0.0e+00  0.0e+00  0.0e+00
  7  6.2e-16  8.3e-16  8.3e-16
  8  1.5e-16  1.5e-16  1.5e-16
  9  0.0e+00  0.0e+00  0.0e+00
 10  3.3e-16  3.3e-16  3.3e-16
 11  0.0e+00  0.0e+00  0.0e+00
 12  3.6e-16  3.6e-16  3.6e-16
 13  1.3e-16  0.0e+00  0.0e+00
 14 -1.9e-16 -3.9e-16 -3.9e-16
 15  0.0e+00 -2.8e-16 -2.8e-16
 16  0.0e+00  0.0e+00  0.0e+00
 17 -3.1e-16 -7.7e-16 -7.7e-16
 18  0.0e+00 -2.3e-16 -2.3e-16
 19  1.7e-16  0.0e+00  0.0e+00
 20  2.5e-16  1.2e-16  1.2e-16
 21  3.6e-16  1.8e-16  1.8e-16
 22 -2.7e-16 -2.7e-16 -2.7e-16
```


Notes:

Between `exp0` and `exp1`, we notice that there are cases where the error of `exp0` is larger than the error of `exp1`, and vice-versa. However, among the first cases, we see some in which there is noticeable difference (more than one order) in the errors, while in the second cases, the difference in the errors is small (half an order or less). Avoiding the more complicated operations (exponentiation and factorial) reduces the probability of large errors.

Between `exp1` and `exp2`, we notice that, for quite negative x, `exp1` exhibits huge errors, while `exp2` matches MATLAB's `exp` to machine precision (relative error to the level of machine epsilon). Note that `exp1` uses alternating signs in the summation, which inhibit the danger of catastrophic cancellation, while `exp2` avoids this problem.

Aside 1: Saturation can, sometimes, cause large errors. For example, in the operation $1 + x$, the x will be lost if $x < \epsilon_{mach}$, and the 1 will be lost if $x > 1/\epsilon_{mach}$. We need to exercise caution in such operations. However, in the question of approximating e^x by Taylor series, saturation serves as a convenient technique to let us (and the code) know where to stop the summation.

Aside 2: We would not be able to do the summation from the smallest to the largest term, because we do not know which is the smallest term needed.