Solutions to Midterm Exam 553.481/681
Spring Semester 2024

Problem 1. IEEE floating-point arithmetic has levels of precision beyond single and double. All of these in standard layout have the first bit for the sign, the next r bits for the stored exponent SE, and the final p bits for the fraction F. The stored exponent is the true exponent E plus the bias, $SE = E + B$, where $B = 2^{r-1} - 1$. In particular, the following precision levels exist:

$$\begin{array}{lll} \text{quadruple:} & r = 15, & p = 112 \\ \text{octuple:} & r = 19, & p = 236 \end{array}$$

In hexadecimal format, quadruple-precision numbers are thus represented by strings of 32 hexadecimal digits, whereas octuple-precision numbers are represented by strings of 64 hexadecimal digits.

Consider the following two IEEE octuple-precision numbers in hexadecimal format:

(i) `40000921fb54442d18469898cc51701b839a252049c1114cf98e804177d4c762`
(ii) `bffff6a09e667f3bcc908b2fb1366ea957d3e3adec17512775099da2f590b066`

(a) Round both of these numbers to quadruple precision (in binary arithmetic or equivalently in hexadecimal) and give for both the hexadecimal representation of the IEEE quadruple-precision number in standard layout.

(b) Round both of the quadruple-precision numbers in (a) to double precision (once again in binary or hexadecimal arithmetic) and give for both the hexadecimal representation of the IEEE double-precision number in standard layout.

(c) Find the decimal representation of the double-precision numbers in (b) to 16 significant figures. You may use `hex2num` in Matlab to check your answer, but explain independently how you arrive at your answers.

Solution: (a) First, for (i), the hexadecimal digits `40000` when converted to binary give `01000000000000000000`. Since the first digit is `0`, the sign is positive. The remaining digits `1000000000000000000` correspond exactly to the bias B plus 1, so that $E = 1$. This same exponent may be represented in quadruple precision by `100000000000000` so that the sign and exponent are represented by `0100000000000000` which converts to `4000` in hexadecimal.

The fraction may be rounded to `921fb54442d18469898cc51701b8` since the next digit `3` is smaller than `8`. Thus, we obtain altogether

`4000921fb54442d18469898cc51701b8`

Second, for (ii), the hexadecimal digits `bffff` when converted to binary give `10111111111111111111`. Since the first digit is `1`, the sign is negative. The remaining digits `0111111111111111111` correspond exactly to the bias B, so that $E = 0$. This same exponent may be represented in quadruple precision by `011111111111111` so that the sign and exponent are represented by `1011111111111111` which converts to `bfff` in hexadecimal.

The fraction may be rounded to `6a09e667f3bcc908b2fb1366ea95` since the next digit `7` is smaller than `8`. Thus, we obtain altogether

`bfff6a09e667f3bcc908b2fb1366ea95`

(b) The rounding to double precision is very similar. First, for (i), the hexadecimal digits representing the sign and stored exponent become `400`, while the fraction is rounded to `921fb54442d18` since the next digit is `4` and smaller than `8`. Thus, we obtain altogether

`400921fb54442d18`

Second, for (ii), the hexadecimal digits representing the sign and stored exponent become `bff`, while the fraction is rounded to `6a09e667f3bcd` since the next digit was `9` and thus the final `c` had to be rounded up to `d`. Thus, we obtain altogether

`bff6a09e667f3bcd`

(c) We can get the decimal values from the following script

```
a=10;b=11;c=12;d=13;e=14;f=15;

SE=polyval([4 0 0],16)
if SE<2^11
E=SE-1023
sign=1;
else
E=SE-(1023+2048)
sign=-1;
end

F=polyval(fliplr([9 2 1 f b 5 4 4 4 2 d 1 8]),1/16)/16

pi2=sign*(1+F)*2^E

SE=polyval([b f f],16)
if SE<2^11
E=SE-1023
```

```
sign=1
else
E=SE-(1023+2048)
sign=-1
end

F=polyval(fliplr([6 a 0 9 e 6 6 7 f 3 b c d]),1/16)/16
nsqrt22=sign*(1+F)*2^E
```

Applying this code, we obtain:

(i) For `400921fb54442d18` the decimal representation 3.141592653589793, which is a double-precision approximation to the number π.

(ii) For `bff6a09e667f3bcd` the decimal representation -1.414213562373095, which is a double-precision approximation to the number $-\sqrt{2}$.

Problem 2. Continuous functions on the closed interval $[a,b]$ can be assigned a norm

$$\|f\| = \max_{x\in[a,b]} |f(x)|$$

and we then say that a sequence $\{f_n\}$ of such functions converges to a function f, or $f = \lim_{n\to\infty} f_n$, if and only if

$$\lim_{n\to\infty} \|f_n - f\| = 0.$$

(a) If $I(f) = \int_a^b f(x)\,dx$ is the Riemann integral, then show that

$$|I(f)| \le (b-a)\cdot\|f\|.$$

(b) Is integration well-posed for continuous functions on the closed interval $[a,b]$? In other words, is there a well-posed output $I(f)$ for input f?

(c) Consider the specific sequence of continuous functions $f_n(x) := \frac{1}{n}\sin(n^2x)$ on the interval $[0,2\pi]$. Show that $\lim_{n\to\infty} f_n = 0$. Is it true as well that $\lim_{n\to\infty} f_n' = 0$? Explain your answer.

(d) If we consider functions on the closed interval $[a,b]$ with a continuous derivative, then is differentiation well-posed for the stated norm? In other words, is there a well-posed output $D(f) = f'$ for input f?

Solution: **(a)** We see from the triangle inequality for Riemann integrals $I(f;a,b) = \int_a^b f(x)\,dx$ that

$$|I(f;a,b)| = \left|\int_a^b f(x)\,dx\right| \le \int_a^b |f(x)|\,dx \le \|f\|\int_a^b dx = (b-a)\cdot\|f\|.$$

(b) Well-posedness means that $I(f;a,b)$ must exist, be unique, and be continuous in the data. By the usual theory of Riemann integration, $I(f;a,b)$ exists and is unique for any continuous function f on $[a,b]$. Continuity in the data f follows from

$$|I(f_n;a,b) - I(f;a,b)| = \left|\int_a^b f_n(x)\,dx - \int_a^b f(x)\,dx\right| = \left|\int_a^b (f_n(x)-f(x))\,dx\right| \le (b-a)\|f_n - f\|,$$

by part (a).

Note also by (a) that

$$|I(f;a,b) - I(f;a_n,b)| = |I(f;a,a_n)| \le |a_n - a|\cdot\|f\|$$

so that $I(f;a,b)$ is continuous in the lower endpoint a. A similar argument gives continuity in the upper exponent b.

Thus, integration is well-posed.

(c) For the given function $\|f_n\| = \frac{1}{n} \to 0$ as $n\to\infty$ and thus $f_n \to 0$. On the other hand, $f_n'(x) = n\cos(n^2x)$ so that $\|f_n'\| = n \to \infty$ as $n\to\infty$. Thus, $f_n' \not\to 0$.

(d) Part (c) provides an example for which $f_n \to 0$ but $D(f_n) \not\to 0 = D(0)$. Thus, differentiation is not continuous in the data, in the specified sense, and thus it is not well-posed.

More generally, for any continuously differentiable function f, we can define $f_n(x) := f(x) + \frac{1}{n}\sin(n^2x)$ and then $f_n \to f$ but $D(f_n) \not\to D(f)$.

Problem 3. Consider the following function

$$g(x) := x - \frac{2f(x)f'(x)}{\Delta(x)}, \quad \Delta(x) := 2(f'(x))^2 - f(x)f''(x) \qquad (*)$$

such that the iteration $x_{n+1} = g(x_n)$ locally converges to x_* satisfying $f(x_*) = 0$.

(a) Show that this iteration has at least cubic order of convergence when $f'(x_*) \neq 0$ and when $f \in C^4$ near x_*.

(b) Write a code to implement iteration with the function (*), taking care to minimize the number of floating point operations in each iteration.

(c) Use your code to solve numerically for a root of the function $f(x) = e^x - 3x$ starting with $x_0 = 2$ and $TOL = 10^{-15}$. Compare with the Newton method for the same x_0 and TOL, both in terms of the number of iterations and the wall clock time required. In particular, calculate the ratio of the wall clock times for the two methods and explain this ratio quantitatively.

(d) Repeat part (c) for the function $f(x) = E_1(x) - x$. Note that

$$E_1(x) := \int_x^\infty \frac{e^{-t}}{t} dt$$

and this function can be evaluated with the Matlab function `expint`. In order to explain the ratio of clock times quantitatively, you will need to estimate the amount of time to evaluate the function $f(x)$ versus the time to evaluate $f'(x)$ or $f''(x)$.

Solution: (a) The iteration function g defines *Halley's method* for root-finding. Using $\Delta(x) = 2(f'(x))^2 - f(x)f''(x)$, its first derivative is given by

$$\begin{aligned} g'(x) &= 1 - \frac{2(f'(x))^2}{2(f'(x))^2 - f(x)f''(x)} - f(x)\frac{d}{dx}\left[\frac{2f'(x)}{\Delta(x)}\right] \\ &= -\frac{f(x)f''(x)}{\Delta(x)} - f(x)\frac{d}{dx}\left[\frac{2f'(x)}{\Delta(x)}\right] \end{aligned}$$

Since $\Delta(x_*) = 2(f'(x_*))^2 \neq 0$, we get from $f(x_*) = 0$ that $g'(x_*) = 0$.

Next we calculate

$$g''(x) = -\frac{f'(x)f''(x)}{\Delta(x)} - f(x)\frac{d}{dx}\left[\frac{f''(x)}{\Delta(x)}\right] - f'(x)\left[\frac{2f''(x)}{\Delta(x)} - \frac{2f'(x)\Delta'(x)}{\Delta^2(x)}\right] - f(x)\frac{d^2}{dx^2}\left[\frac{2f'(x)}{\Delta(x)}\right]$$

with

$$\Delta'(x) = 4f'(x)f''(x) - f'(x)f''(x) - f(x)f'''(x) = 3f'(x)f''(x) - f(x)f'''(x).$$

Thus,

$$\begin{aligned} g''(x_*) &= -\frac{3f'(x_*)f''(x_*)}{\Delta(x_*)} + \frac{2(f'(x_*))^2\Delta'(x_*)}{\Delta^2(x_*)} \\ &= -\frac{3f''(x_*)}{2f'(x_*)} + \frac{3f''(x_*)}{2f'(x_*)} \;=\; 0. \end{aligned}$$

since $\Delta'(x_*) = 3f'(x_*)f''(x_*)$. It follows that Halley's method has at least cubic rate of convergence.

(b) A Matlab script to carry out Halley's method as $x_{n+1} = x_n - \frac{1}{\frac{f'(x_n)}{f(x_n)} - \frac{f''(x_n)}{2f'(x_n)}}$, with three multiplications per iteration, is the following:

```

function [x,k,err]=halley(f,Df,DDf,x,tol,itmax)

k=0;
if x ≠ 0
  xold=0;
else
  xold=1;
end

err=abs(x-xold);

while err>tol*max(abs(x),1.0)
     if k+1>itmax
       break
     end
     if f(x)==0
       break
     end
     xold=x;
     y=f(x);
     yp=Df(x);
     ypp=DDf(x);
     Ly=yp/y;
     Ly2=ypp/yp/2;
     x=x-1/(Ly-Ly2);
     k=k+1;
     err=abs(x-xold);
end

end
```

(c) A Matlab script to find the root by each of the methods and to time them is the following:

```
f=@(x) exp(x)-3*x;
Df=@(x) exp(x)-3;
DDf=@(x) exp(x);

tol=1e-15;
itmax=100;
x0=2;

[xH,kH,errH]=halley(f,Df,DDf,x0,tol,itmax)
[xN,kN,errN]=newton(f,Df,x0,tol,itmax)

NTRY=1e4;
timeN=0;
timeH=0;
for ii=1:NTRY

tic
[xH,kH,errH]=halley(f,Df,DDf,x0,tol,itmax);
timeH=timeH+toc;
tic
[xN,kN,errN]=newton(f,Df,x0,tol,itmax);
timeN=timeN+toc;

end

ratioHN=timeH/timeN
```

The root is $x_* = 1.512134551657843$ and 4 iterations are required by the Halley method, while 6 iterations are required by Newton. We find also that the ratio of wall clock times is nearly equal to 1. This is easy to understand because each of the three functions $f(x)$, $f'(x)$, $f''(x)$ are almost the same and require about the same time to compute. Since the Halley method uses 3 function evaluations per iteration, it required a total of 12 function evaluations. Likewise, the Newton method uses 2 function evaluations per iteration, so that it also required a total of 12 function evaluations.

(d) A Matlab script to find the root by each of the methods and to time them is the following:

```
f=@(x) expint(x)-x;
Df=@(x) -exp(-x)./x-1;
DDf=@(x) exp(-x).*(1+x)./x./x;


tol=1e-15;
itmax=100;
x0=2;

[xH,kH,errH]=halley(f,Df,DDf,x0,tol,itmax)
[xN,kN,errN]=newton(f,Df,x0,tol,itmax)

NTRY=1e4;
timeN=0;
timeH=0;
for ii=1:NTRY

tic
[xH,kH,errH]=halley(f,Df,DDf,x0,tol,itmax);
timeH=timeH+toc;
tic
[xN,kN,errN]=newton(f,Df,x0,tol,itmax);
timeN=timeN+toc;

end

ratioHN=timeH/timeN
```

The root is $x_* = 0.527612347201742$ and 4 iterations are required by the Halley method, while 7 iterations are required by Newton. We find also that the ratio of wall clock times is approximately 0.62. To understand this ratio, we time the function evaluations with the script

```
NTRY=1e4;
timeF=0;
timeDF=0;
for ii=1:NTRY

tic
y=f(x0);
timeF=timeF+toc;
tic
yp=Df(x0);
timeDF=timeDF+toc;

end

ratioFDF=timeF/timeDF
```

which shows that evaluating $f(x)$ requires a time about 20 times longer than the time to evaluate $f'(x)$ or $f''(x)$. These results will depend on which Matlab version or other programming language is used. We can therefore estimate in `Matlab2018b` that the time ratio is given by

$$\frac{4(20+2)}{7(20+1)} \doteq 0.6.$$