MATH 227B: Mathematical Biology
Homework 1

Karthik Desingu

January 15, 2026

Problem 1

Computer uses binary to represent numbers. Using this information, write Matlab code to obtain the machine precision as accurately as possible.

The Matlab code to determine the *machine precision*, defined as the smallest positive floating–point number ε such that, in floating–point arithmetic,

$$1 + \varepsilon > 1$$

is shown in Listing 1. This quantity characterizes the spacing between the real number 1 and the next larger representable floating–point number.

```
% double-precision floating point numbers.
eps_machine_double = 1.0;
rnd_num_double = double(1.0);
while rnd_num_double + eps_machine_double > rnd_num_double
    eps_machine_double = eps_machine_double / 2;
end
eps_machine_double = eps_machine_double * 2;

disp( for double-precision floating point number: );
disp(eps_machine_double);


% single-precision floating point numbers.
rnd_num_single = single(1.0);
eps_machine_single = 1.0;
while rnd_num_single + eps_machine_single > rnd_num_single
    eps_machine_single = eps_machine_single / 2;
end
eps_machine_single = eps_machine_single * 2;

disp( for single-precision floating point number: );
disp(eps_machine_single);

% OUTPUT.
>> for double-precision floating point number:
   2.2204e-16
```

```
>> for single-precision floating point number:
   1.1921e-07
```

Listing 1: Matlab script to numerically estimate the machine precision.

```
>> machine_precision
for double-precision floating point number:
   2.2204e-16

for single-precision floating point number:
   1.1921e-07
```

Figure 1: Output from the code to compute machine precision for double and single precision floating point numbers.

We begin by setting $\varepsilon = 1$. This initial value is much larger than the desired machine precision, but it guarantees that $1 + \varepsilon$ is representably different from 1. The algorithm then repeatedly replaces ε by $\varepsilon/2$. The choice to divide by 2 at each step is essential: floating–point numbers are stored in base 2, so halving corresponds exactly to moving one step down the representation of binary fractions. In binary arithmetic, division by 2 is equivalent to shifting the significand (or, the mantissa, in other words) one bit to the right, allowing the algorithm to explore progressively smaller representable numbers without skipping any steps.

After each halving, the algorithm checks whether $1+\varepsilon$ is still strictly greater than 1 when computed on the machine. As long as this condition holds, ε is large enough to alter the stored floating–point value of 1. Eventually, however, ε becomes smaller than the spacing between 1 and the next representable floating–point number. When this happens, the exact real number $1 + \varepsilon$ exists mathematically, but after rounding to the nearest representable binary floating–point number, it is stored as exactly 1. At this point, the condition

$$1 + \varepsilon > 1$$

fails, and the iteration terminates.

Because the loop stops only after ε has become too small to affect the result, the final value of ε tested is *half* of the desired machine precision. Multiplying by 2 recovers the last value of ε for which $1 + \varepsilon$ was distinguishable from 1, which is precisely the machine epsilon.

This procedure directly reflects the structure of binary floating–point arithmetic. Near 1, double–precision floating–point numbers, based on looking up online, have the form

$$1 + k\,2^{-52}, \qquad k \in Z,$$

so the spacing between adjacent representable numbers is $2^{-52} \approx 2.22 \times 10^{-16}$.

The algorithm effectively searches for this spacing by going through successive powers of two until rounding eliminates the increment. The resulting value is therefore the smallest representable power of two that changes the stored value of 1, which is exactly the machine precision.

The second piece of code repeats the above for *single-precision* floating point numbers. Like before, the representable single–precision floating–point numbers have the form, based on an online search,

$$1 + k\,2^{-23}, \qquad k \in Z.$$

Consequently, the spacing between consecutive representable numbers in a neighborhood of 1 is $2^{-23} \approx 1.19 \times 10^{-7}$.

Both the estimates from the code are close to the analytical estimates.

Problem 2

Expand $\dfrac{sin(x)}{x} - 1$ in a power series about 0. Calculate the number of terms that are necessary to ensure a relative error between 10^{-7} and 10^{-16} for any x in $[0, 1]$.

We study the function

$$f(x) = \frac{\sin x}{x} - 1 \quad \text{for } x \in [0, 1],$$

where the value at $x = 0$ is defined as

$$f(0) = \lim_{x \to 0} \left(\frac{\sin x}{x} - 1 \right) = 0.$$

This limit exists because the sine function satisfies $\sin x \approx x$ near $x = 0$; more precisely, the ratio $\sin x / x$ approaches 1 as $x \to 0$. To justify this definition, we compute the limit explicitly. Using the identity

$$\sin x = x \cos\theta \quad \text{for some } \theta \text{ between 0 and } x$$

(which follows from the mean value theorem applied to $\sin x$), we obtain

$$\frac{\sin x}{x} = \cos\theta.$$

As $x \to 0$, the intermediate point θ also tends to 0, and since $\cos\theta \to 1$ as $\theta \to 0$, it follows that

$$\lim_{x \to 0} \frac{\sin x}{x} = 1.$$

Therefore,

$$\lim_{x \to 0} \left(\frac{\sin x}{x} - 1 \right) = 0,$$

which confirms that defining $f(0) = 0$ makes f continuous at $x = 0$. Thus, f is a well-defined function on the entire interval $[0, 1]$.

We begin with the Taylor expansion of the sine function about $x = 0$,

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots ,$$

which converges for all real x. Dividing this series term by term by x gives

$$\frac{\sin x}{x} = 1 - \frac{x^2}{3!} + \frac{x^4}{5!} - \frac{x^6}{7!} + \cdots .$$

Subtracting 1 yields

$$f(x) = -\frac{x^2}{3!} + \frac{x^4}{5!} - \frac{x^6}{7!} + \cdots = \sum_{k=1}^{\infty} (-1)^k \frac{x^{2k}}{(2k+1)!}.$$

Let

$$S_N(x) = \sum_{k=1}^{N} (-1)^k \frac{x^{2k}}{(2k+1)!}$$

denote the truncation after the first N nonzero terms. The degree of the polynomial $S_N(x)$ is $2N$. This choice is natural because the power series for $f(x)$ contains only even powers of x. Each additional term in the series increases the degree by two, and truncating after N terms therefore produces a polynomial of degree $2N$. Consequently, $S_N(x)$ is exactly the Taylor polynomial of f about $x = 0$ that includes all terms up to and including x^{2N}.

We wish to estimate the error

$$R_N(x) = f(x) - S_N(x)$$

uniformly for $x \in [0, 1]$. In order to estimate the error $R_N(x)$, we must control how far the function $f(x)$ can deviate from its Taylor polynomial. Taylor's theorem provides such a control by expressing the error in terms of a higher-order derivative of f. Specifically, the theorem states that if a function has a continuous $(m + 1)$-st derivative on an interval containing 0 and x, then the difference between the function and its Taylor polynomial of degree m at 0 can be written as a multiple of the $(m+1)$-st derivative evaluated at some intermediate point. Therefore, to obtain a uniform bound on $R_N(x)$, it is necessary to bound the size of this derivative on the interval $[0, 1]$.

To do this, we use Taylor's theorem with remainder. Since the power series above converges for all x, the function f has derivatives of all orders on $[0, 1]$, and its Taylor polynomial of degree $2N$ about $x = 0$ is exactly $S_N(x)$. Taylor's theorem then states that for each $x \in [0, 1]$ there exists a number ξ between 0 and x such that

$$R_N(x) = \frac{f^{(2N+2)}(\xi)}{(2N+2)!}\, x^{2N+2}.$$

To bound this remainder, we must estimate the size of the derivatives of f. This can be done directly from the power series. Differentiating the series for

$f(x)$ term by term $2N+2$ times gives

$$f^{(2N+2)}(x) = \sum_{k=N+1}^{\infty} (-1)^k \frac{(2k)!}{(2k+1)!}\, x^{2k-(2N+2)} = \sum_{k=N+1}^{\infty} (-1)^k \frac{x^{2k-(2N+2)}}{2k+1}.$$

For $x \in [0,1]$, each power $x^{2k-(2N+2)}$ lies between 0 and 1, and therefore

$$\left| f^{(2N+2)}(x) \right| \leq \sum_{k=N+1}^{\infty} \frac{1}{2k+1} \leq \frac{1}{2N+3}.$$

This shows that the $(2N+2)$-nd derivative of f is bounded uniformly on $[0,1]$.

Substituting this bound into the remainder formula yields

$$|R_N(x)| \leq \frac{1}{(2N+2)!} \cdot \frac{1}{2N+3}\, x^{2N+2} = \frac{x^{2N+2}}{(2N+3)!}.$$

Since $0 \leq x \leq 1$, we obtain the uniform bound

$$|R_N(x)| \leq \frac{1}{(2N+3)!} \qquad x \in [0,1].$$

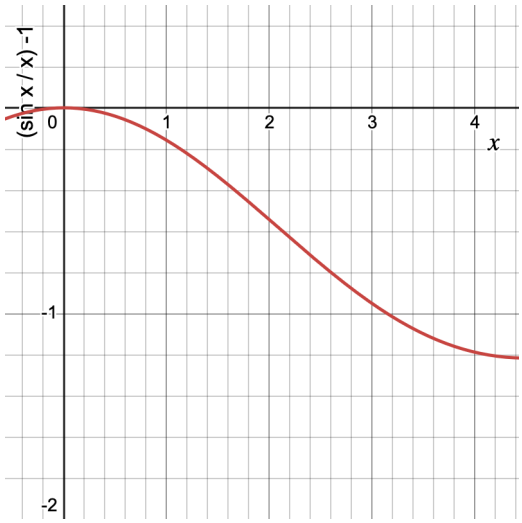

Figure 2: Plot of $\sin x/x - 1$ against x shows that it decreases as x increases in $[0,1]$.

Now, we compute the relative error. For $x > 0$, the quantity $\sin x/x - 1$ is negative, and a simple plot (shown in Figure **??**) shows that it decreases as x increases on $(0,1]$. Consequently, the absolute value is bounded as,

$$0 > \left| \frac{\sin x}{x} - 1 \right| \geq |\sin 1 - 1| \approx 0.1585, \qquad x \in (0,1].$$

Combining this inequality with the bound on $R_N(x)$ gives

$$\frac{|R_N(x)|}{\left|\frac{\sin x}{x} - 1\right|} \leq \frac{1}{(2N+3)!\,|\sin 1 - 1|} \qquad x \in (0, 1].$$

To guarantee a uniform relative error no larger than a prescribed tolerance ε, it is therefore sufficient to choose N such that

$$\frac{1}{(2N+3)!} \leq \varepsilon\,|\sin 1 - 1|.$$

For $\varepsilon = 10^{-7}$, this inequality is satisfied when $2N + 3 = 12$, giving $N = 5$. For $\varepsilon = 10^{-16}$, it is satisfied when $2N + 3 = 19$, giving $N = 8$.

Thus, by using Taylor's theorem and explicit bounds on derivatives, we obtain a clear and general justification for the number of terms required to approximate $\sin x/x - 1$ with a prescribed relative accuracy on the interval $[0, 1]$.

Problem 3

Given a real number on a computer with k-digit rounding arithmetic, analytically estimate the relative error bound for computer representation.

Assume a binary floating–point system ($\beta = 2$) in which numbers are stored using n bits total. Of these bits, one is used for the sign, some are used to store the exponent, and the remaining k bits are used for the mantissa (significand). We assume chopping (truncation toward zero) and that the number $x \neq 0$ is normalized, i.e., is in the standard notation.

A normalized real number can be written as

$$x = \pm m\, 2^e, \qquad 1 \leq m < 2,$$

and the stored value is

$$\mathrm{fl}(x) = \pm \hat{m}\, 2^e,$$

where $\hat{m}$ is obtained by chopping the binary expansion of m after k bits.

The binary expansion of m has the form

$$m = 1.b_1 b_2 \cdots b_k b_{k+1} b_{k+2} \cdots, \qquad b_i \in \{0, 1\},$$

while the chopped mantissa is

$$\hat{m} = 1.b_1 b_2 \cdots b_k.$$

The discarded tail satisfies

$$0 \leq m - \hat{m} = 2^{-(k+1)} b_{k+1} + 2^{-(k+2)} b_{k+2} + \cdots < 2^{-k}.$$

This is because, the discarded part is non-negative, which yields $m - \hat{m} \geq 0$. The maximum value of $m - \hat{m}$ occurs when each digit in the truncated portion

is at its maximum, which is 1 for base $\beta = 2$.

Multiplying the discarded tail by 2^e, the absolute representation error is

$$0 \leq |x - \mathrm{fl}(x)| = |m - \hat{m}|\, 2^e < 2^{1-k} \cdot 2^e = 2^{e-(k-1)}.$$

This quantity is precisely the spacing between adjacent floating–point numbers with exponent e, i.e., one unit in the last place.

The relative representation error is therefore,

$$0 \leq \frac{|x - \mathrm{fl}(x)|}{|x|} = \frac{m - \hat{m}}{m} < \frac{2^{-k}}{m}.$$

Since normalization guarantees $1 \leq m < 2$, we have $1/m \leq 1$, and hence

$$\boxed{0 \leq \frac{|x - \mathrm{fl}(x)|}{|x|} < 2^{-k}.}$$

Furthermore, the mantissa length k is bounded by the total number of bits n used to store the floating–point number. In particular,

$$k \leq n - 1,$$

since at least one bit is needed for the sign (and some bits for the exponent). Thus,

$$2^{-k} \leq 2^{-(n-1)}.$$