Solution 1: Big-O Proofs

For the following statements, either prove them or give a counterexample.

(a) $\max[|f(n)|, |g(n)|] \in \Theta(|f(n)| + |g(n)|)$.

(b) For eventually nonzero $f(n)$ and $g(n)$, $f(n) \in O(g(n))$ implies $g(n) \in O(f(n))$.

(c) For eventually nonzero $f(n)$ and $g(n)$, $f(n) \in O(g(n))$ if and only if $g(n) \in \Omega(f(n))$.

(d) $f(n) \in O(f(2n))$.

$f(n)$ being "eventually nonzero" here means that there is some n_0 so that for all $n > n_0$, $f(n) \neq 0$.

(a) $\max(|f(n)|, |g(n)|) \in \Theta(|f(n)| + |g(n)|)$**.**

This statement is **true**.

To prove $\max(|f(n)|, |g(n)|) \in \Theta(|f(n)|+|g(n)|)$, we need to show that there exist positive constants c_1, c_2 and a natural number n_0 such that for all $n \geq n_0$: $c_1(|f(n)| + |g(n)|) \leq \max(|f(n)|, |g(n)|) \leq c_2(|f(n)| + |g(n)|)$.

- **Upper bound (O):** We want to show $\max(|f(n)|, |g(n)|) \leq c_2(|f(n)| + |g(n)|)$. By definition of maximum, $\max(|f(n)|, |g(n)|) \geq |f(n)|$ and $\max(|f(n)|, |g(n)|) \geq |g(n)|$. If $|f(n)| \geq |g(n)|$, then $\max(|f(n)|, |g(n)|) = |f(n)|$. And $|f(n)| \leq |f(n)| + |g(n)|$ (since $|g(n)| \geq 0$). If $|g(n)| > |f(n)|$, then $\max(|f(n)|, |g(n)|) = |g(n)|$. And $|g(n)| \leq |f(n)| + |g(n)|$ (since $|f(n)| \geq 0$). In both cases, $\max(|f(n)|, |g(n)|) \leq |f(n)| + |g(n)|$. So, we can choose $c_2 = 1$ and we can pick $n_0 = 1$. For $n \geq n_0$, $\max(|f(n)|, |g(n)|) \leq 1 \cdot (|f(n)| + |g(n)|)$.
- **Lower bound (Ω):** We want to show $c_1(|f(n)| + |g(n)|) \leq \max(|f(n)|, |g(n)|)$. This is equivalent to showing $|f(n)| + |g(n)| \leq \frac{1}{c_1} \max(|f(n)|, |g(n)|)$. Consider $|f(n)| + |g(n)|$. We know $|f(n)| \leq \max(|f(n)|, |g(n)|)$ and $|g(n)| \leq \max(|f(n)|, |g(n)|)$. Therefore, $|f(n)| + |g(n)| \leq \max(|f(n)|, |g(n)|) + \max(|f(n)|, |g(n)|) = 2\max(|f(n)|, |g(n)|)$. So, $|f(n)| + |g(n)| \leq 2\max(|f(n)|, |g(n)|)$. This means we can choose $c_1 = 1/2$ and $n_0 = 1$. For $n \geq n_0$, $\frac{1}{2}(|f(n)| + |g(n)|) \leq \max(|f(n)|, |g(n)|)$.

Combining both parts, for $c_1 = 1/2$, $c_2 = 1$, and any n_0, we have: $\frac{1}{2}(|f(n)| + |g(n)|) \leq \max(|f(n)|, |g(n)|) \leq 1 \cdot (|f(n)| + |g(n)|)$ for all $n \geq n_0$. Thus, $\max(|f(n)|, |g(n)|) \in \Theta(|f(n)| + |g(n)|)$.

(b) **Statement: For eventually nonzero** $f(n)$ **and** $g(n)$**,** $f(n) \in O(g(n))$ **implies** $g(n) \in O(f(n))$**.**

This statement is **false**.

Counterexample: Let $f(n) = n$ and $g(n) = n^2$. $f(n) \in O(g(n))$ because $n \leq 1 \cdot n^2$ for all $n \geq 1$. Here, $c = 1, n_0 = 1$. Now, let's check if $g(n) \in O(f(n))$. This would mean $n^2 \leq c' \cdot n$ for some constant $c' > 0$ and for all $n \geq n'_0$. Dividing by n (for $n > 0$), this implies $n \leq c'$. However, this inequality cannot hold for all $n \geq n'_0$ because n can grow arbitrarily large, while c' is a constant. Therefore, $g(n) \notin O(f(n))$.

(c) **Statement: For eventually nonzero** $f(n)$ **and** $g(n)$**,** $f(n) \in O(g(n))$ **if and only if** $g(n) \in \Omega(f(n))$**.**

This statement is **true**.

By definition:

- $f(n) \in O(g(n))$ means there exist constants $c_1 > 0$ and n_1 such that for all $n \geq n_1$, $|f(n)| \leq c_1|g(n)|$.
- $g(n) \in \Omega(f(n))$ means there exist constants $c_2 > 0$ and n_2 such that for all $n \geq n_2$, $c_2|f(n)| \leq |g(n)|$.

Consider the core inequalities for $n \geq \max(n_1, n_2)$:

$$|f(n)| \leq c_1|g(n)| \quad \text{and} \quad c_2|f(n)| \leq |g(n)|$$

Since $|f(n)|, |g(n)|, c_1, c_2$ are all positive for sufficiently large n, we can rearrange the inequalities:

$$|f(n)| \leq c_1|g(n)| \iff \frac{1}{c_1}|f(n)| \leq |g(n)|$$

$$c_2|f(n)| \leq |g(n)| \iff |f(n)| \leq \frac{1}{c_2}|g(n)|$$

Let $c_2 = 1/c_1$. Since $c_1 > 0$, it follows that $c_2 > 0$. The condition $|f(n)| \leq c_1|g(n)|$ for all $n \geq n_1$ holds if and only if the condition $|g(n)| \geq c_2|f(n)|$ holds for $c_2 = 1/c_1$ and for all $n \geq n_1$.

Thus, the existence of $c_1 > 0, n_1$ for the O definition is equivalent to the existence of $c_2 = 1/c_1 > 0, n_2 = n_1$ for the Ω definition. The two statements are therefore equivalent.

(d) **Statement:** $f(n) \in O(f(2n))$**.**

This statement is **false**.

Counterexample: Let $f(n) = 2^{-n}$. Then $f(2n) = 2^{-2n}$. For $f(n) \in O(f(2n))$, we would need to find constants $c > 0$ and n_0 such that for all $n \geq n_0$: $f(n) \leq c \cdot f(2n)$ $2^{-n} \leq c \cdot 2^{-2n}$ Dividing by 2^{-2n} (which is positive): $2^{-n}/2^{-2n} \leq c$ $2^{-n+2n} \leq c$ $2^n \leq c$ As $n \to \infty$, $2^n \to \infty$. Therefore, for any constant $c > 0$, the inequality $2^n \leq c$ cannot hold for all $n \geq n_0$. Thus, $f(n) = 2^{-n}$ is a counterexample.

Note: The statement is true for functions whose absolute value is eventually non-decreasing. If $|f(n)|$ is non-decreasing, then $|f(n)| \leq |f(2n)|$ for sufficiently large n. In this case, $f(n) \in O(f(2n))$ holds with $c = 1$.

Solution 2: Big-O Order

(a) Partition the following functions into Θ-equivalence classes, where $f \sim g$ means $f \in \Theta(g)$. List the classes in increasing order of asymptotic growth rate, denoted by $\prec$, where $C_i \prec C_j$ means $f(n) \in o(g(n))$ for all $f \in C_i$ and $g \in C_j$.

In all of these, log is the natural logarithm.

- $f(n) = n^2$
- $g(n) = n^2 + \log n$
- $h(n) = n^n$
- $i(n) = \log n$
- $j(n) = (\log n)^2$
- $k(n) = \log(n^2)$
- $l(n) = 2^{2^n}$
- $m(n) = 2^n$
- $n(n) = n!$
- $p(n) = 2^{\log n}$

You may need to use the following approximation of $n!$, called Stirling's approximation:

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + O\left(\frac{1}{n}\right)\right)$$

(b) In which of the cases in (a) does the base of the logarithm matter? I.e., in which case is a given function not in the Θ-equivalence class of the same function using a logarithm of a different base?

(a) **Part 1: Ordering Function Classes**

First, let's simplify some functions and determine their Θ-classes:

- $g(n) = n^2 + \log n \in \Theta(n^2)$ since n^2 dominates $\log n$.
- $k(n) = \log(n^2) = 2\log n \in \Theta(\log n)$.
- $p(n) = 2^{\log n} = (e^{\log 2})^{\log n} = (e^{\log n})^{\log 2} = n^{\log 2}$.

The Θ-equivalence classes are:

- $C_1 = \{i(n), k(n)\} = [\Theta(\log n)]$
- $C_2 = \{j(n)\} = [\Theta((\log n)^2)]$
- $C_3 = \{p(n)\} = [\Theta(n^{\log 2})]$ (Note: $\log 2 \approx 0.693$)
- $C_4 = \{f(n), g(n)\} = [\Theta(n^2)]$
- $C_5 = \{m(n)\} = [\Theta(2^n)]$
- $C_6 = \{n(n)\} = [\Theta(n!)]$
- $C_7 = \{h(n)\} = [\Theta(n^n)]$
- $C_8 = \{l(n)\} = [\Theta(2^{2^n})]$

The classes are ordered by asymptotic growth rate as follows: $C_1 \prec C_2 \prec C_3 \prec C_4 \prec C_5 \prec C_6 \prec C_7 \prec C_8$

Justification for non-obvious relations:

- $C_5 \prec C_6$: $2^n \in o(n!)$. Consider the ratio $\frac{n!}{2^n} = \frac{1 \cdot 2 \cdot 3 \cdots n}{2 \cdot 2 \cdot 2 \cdots 2}$. For $n \geq 4$, the terms $\frac{k}{2}$ are ≥ 2, so the ratio grows without bound.
- $C_6 \prec C_7$: $n! \in o(n^n)$. Using Stirling's approximation $n! \approx \sqrt{2\pi n}(n/e)^n$. Compare $\frac{n^n}{n!} \approx \frac{n^n}{\sqrt{2\pi n}(n/e)^n} = \frac{e^n}{\sqrt{2\pi n}}$, which tends to ∞.
- $C_7 \prec C_8$: $n^n \in o(2^{2^n})$. Compare $\log(n^n) = n\log n$ and $\log(2^{2^n}) = 2^n \log 2$. Since $n \log n \in o(2^n)$, the logarithm of 2^{2^n} grows much faster than the logarithm of n^n, implying $n^n \in o(2^{2^n})$.

(b) **Part 2: Base of Logarithm**

The base of the logarithm matters if changing the base changes the Θ-class. The change of base formula is $\log_b n = \frac{\log_a n}{\log_a b}$. Since $\log_a b$ is a constant factor, changing the base does *not* affect the Θ-class for functions where the logarithm appears as a factor or is the primary term, such as $\log n$, $(\log n)^2$, or $n^c \log n$.

However, if the logarithm appears in an exponent, the base matters. Consider $p(n) = 2^{\log_b n}$. Using the change of base to base e (natural log, as specified): $p(n) = 2^{\frac{\ln n}{\ln b}}$. $p(n) = (2^{1/\ln b})^{\ln n} = (e^{\ln 2/\ln b})^{\ln n} = (e^{\ln n})^{\ln 2/\ln b} = n^{(\ln 2/\ln b)}$. The exponent $(\ln 2/\ln b)$ directly depends on the base b. Changing b changes the exponent and therefore changes the function's polynomial degree, thus altering its Θ-class.

Therefore, the base of the logarithm matters only for the function $p(n) = 2^{\log n}$.

Solution 3: Empirical Epsilon

(a) Write a function in R that computes, for a given positive floating point number x, the smallest number $u(x)$ such that $x + u(x) \neq x$ in machine arithmetic. Use a bisection method: start with candidates u_{lower}, for which you know that $x + u_{\text{lower}} = x$, and u_{upper}, for which you know that $x + u_{\text{upper}} \neq x$. Then, repeatedly check the midpoint of the interval $[u_{\text{lower}}, u_{\text{upper}}]$ on whether adding it to x yields x or not, and replace either the lower or upper bound by the midpoint. Make sure that your function works with very small numbers: The midpoint between u_{lower} and u_{upper} might not be representable as a value distinct from either u_{lower} or u_{upper}, your function should not get stuck in an infinite loop in this case.

(b) Use your function to calculate the value of $u(x)$ for a reasonably dense grid of values of x between 2^{-1024} and 2^{-1010} on a logarithmic scale. ("reasonably dense" here means you should not only consider integer powers of 2. Notice that these are all very small numbers.) Plot the result on a log-log plot. What do you observe?

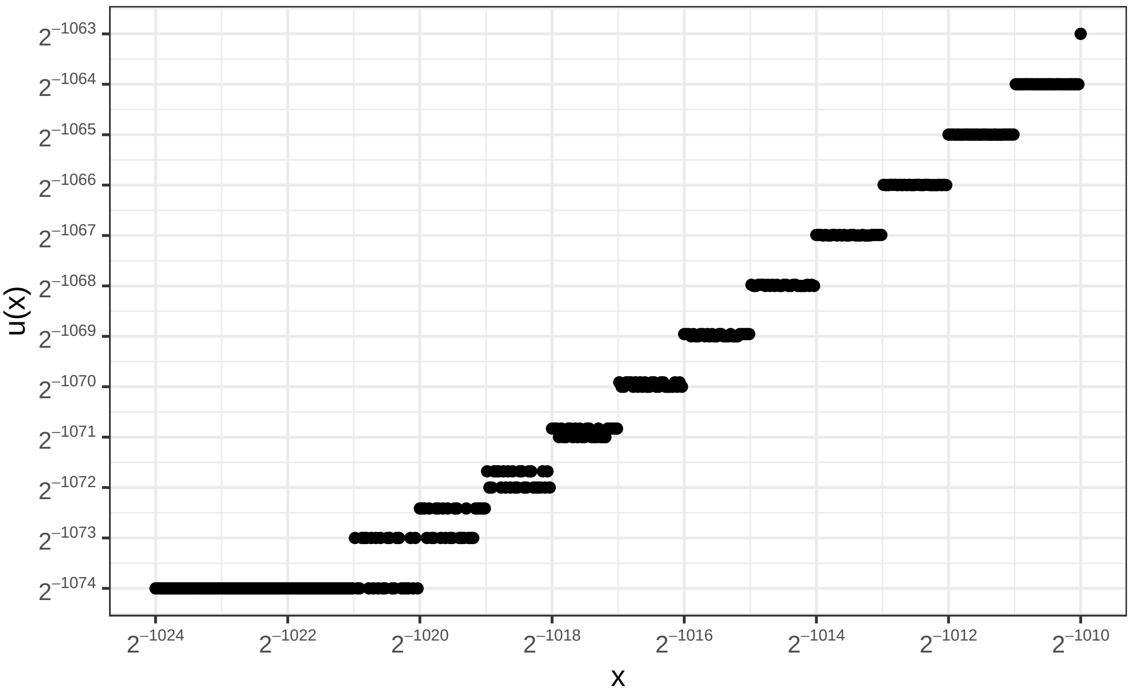
Bonus: (think about this, but don't worry if you can't solve it) Why is $u(x)$ not monotonically non-decreasing? I.e., why are there apparently random fluctuations, with values of $x_1 < x_2$ for which $u(x_1) > u(x_2)$?

(a) The following R-code computes the value of $u(x)$ for a positive scalar x, using the bisection method:

```r
calcEpsilon <- function(x) {
  checkmate::assertNumber(x, lower = 0, finite = TRUE)
  if (x == 0) stop("x must be positive")
  # Upper bound: we know that x + x is always != x
  # We could use x * .Machine£double.eps, but that could make problems for
  # denormalized numbers. Handling denormalized numbers separately would be more
  # efficient, but here we prioritize clarity and simplicity.
  upper <- x
  # Lower bound: we know that x + 0 is always == x
  lower <- 0
  repeat {
    candidate <- (upper + lower) / 2
    # We terminate when we have full precision: the difference between upper and
    # lower bound is less than 1 ULP
    if (candidate == lower || candidate == upper) return(upper)
    if (x + candidate == x) {
      lower <- candidate
    } else {
      upper <- candidate
    }
  }
}
```

(b) The following R-code plots the result of the bisection method for a grid of values of x between 2^{-1024} and 2^{-1010} on a logarithmic scale:

```r
library(ggplot2)
x.grid <- 2^seq(-1024, -1010, length.out = 400)
u <- vapply(x.grid, calcEpsilon, numeric(1))

ggplot(data.frame(x = log2(x.grid), u = log2(u)), aes(x = x, y = u)) +
  geom_point() +
  scale_x_continuous(
    breaks = seq(-1024, -1010, by = 2),
    labels = function(x) parse(text = paste0("2^", x))
  ) +
  scale_y_continuous(
    breaks = function(x) seq(ceiling(min(x)), floor(max(x)), by = 1),
    labels = function(x) parse(text = paste0("2^", x))
  ) +
  xlab("x") +
  ylab("u(x)") +
  theme_bw() +
  theme(aspect.ratio = 0.6)
```



Note we had to `log2()` the values to plot them, since `ggplot2`'s log scales fail for denormalized numbers.

To explain this plot, it helps to remember key constants of IEEE 754 double precision floating point numbers:

- The exponent has 11 bits, with range -1022 to 1023.
- The significand has 52 bits plus 1 hidden bit.
- The smallest normalized number is 2^{-1022}.
- The smallest denormalized number is $2^{-1022} \times 2^{-52} = 2^{-1074} \approx 4.94 \times 10^{-324}$.
- The "machine epsilon" is $\epsilon_m = 2^{-52} \approx 2.22 \times 10^{-16}$. With default IEEE 754 rounding (round to nearest, ties to even), the largest number we can add to a positive normalized number x that does not yield a number different from x is between $\frac{1}{4}\epsilon_m \times x = 2^{-54} \times x$ and $\frac{1}{2}\epsilon_m \times x = 2^{-53} \times x$.

Observations:

- For $x < 2^{-1022}$, $u(x)$ is constant, since these are represented as denormalized numbers. For these, the smallest number that can be added to x to get a number different from x is the smallest denormalized number, 2^{-1074}.
- For $2^{-1022} \leq x < 2^{1021}$, x is a normalized number, so the smallest number we can add *should* be at most $\frac{1}{2}\epsilon_m \times x \approx \frac{1}{2} \times 2^{-52} \times 2^{-1022} = 2^{-1075}$. However, this is smaller than the smallest denormalized number! So the smallest nonzero value that can be added is 2^{-1074} here, as well.
- For $x \geq 2^{1021}$, we observe strange behavior: The value of $u(x)$ fluctuates, apparently randomly, between $2^{\lfloor \log_2(x) \rfloor - 53}$ and $2^{\lfloor \log_2(x) \rfloor - 53} + 2^{-1074}$.

The difference between a value x and the next larger representable number is called *unit of least precision* (ULP), which is $\mathrm{ULP}(x) = 2^{\lfloor \log_2(x) \rfloor - 52} = 2^{e-52}$ for normalized numbers, where e is the binary exponent of $x = 1.b_1 b_2 \ldots b_{52} \times 2^e$. Adding a value larger than $\frac{1}{2}\mathrm{ULP}(x)$ to x will yield a number different from x. Adding a value of *exactly* $\frac{1}{2}\mathrm{ULP}(x)$ will yield a number different from x only if the least significant bit of x is 1, because the default rounding mode breaks ties by rounding to the nearest even number!

This means that $\frac{1}{2}\mathrm{ULP}(x) = 2^{\lfloor \log_2(x) \rfloor - 53}$ is sometimes large enough to get a number different from x. However, if the least significant bit of x is 0, then a value greater than $\frac{1}{2}\mathrm{ULP}(x)$ is needed. This is the next larger representable number, $\frac{1}{2}\mathrm{ULP}(x) + 2^{-1074}$. (For larger values of x not in the plot, the next representable number would actually be $\frac{1}{2}\mathrm{ULP}(x) + \mathrm{ULP}(\frac{1}{2}\mathrm{ULP}(x))$).

Consider the example $x = 2^{-1018}$, i.e.
$2^{-1018} \times \texttt{1.0000000000000000000000000000000000000000000000000000}|_2$,
where the vertical delimiter indicates the position of the least significant bit that is stored in the significand.

The next larger representable number is $2^{-1018} + 2^{-1018} \times 2^{-52} = 2^{-1018} + 2^{-1070}$, i.e.
$2^{-1018} \times \texttt{1.0000000000000000000000000000000000000000000000000001}|_2$.

The number we need to add to x needs to be larger than $2^{-1070}/2 = 2^{-1071}$; if it is exactly 2^{-1071}, the sum is rounded to the nearest even number, which is still x. This next larger number needs to be of the form $2^{-1018} \times \texttt{0.0000000000000000000000000000000000000000000000000000}|\texttt{10}\ldots\texttt{1}_2$.

However, we are in the realm of denormalized numbers here, so the last "1" should be at the position of the smallest denormalized number, i.e. at position 2^{-1074}.

We therefore have $u(2^{-1018}) = 2^{-1071} + 2^{-1074}$:

$$\begin{array}{rccl}
 & 2^{-1018} & \times & \texttt{1.0000000000000000000000000000000000000000000000000000}|_2 \\
+ & 2^{-1018} & \times & \texttt{0.0000000000000000000000000000000000000000000000000000}|\texttt{1001}_2 \\
= & 2^{-1018} & \times & \texttt{1.0000000000000000000000000000000000000000000000000000}|\texttt{1001}_2 \\
\text{rounding to} & 2^{-1018} & \times & \texttt{1.0000000000000000000000000000000000000000000000000001}|_2
\end{array}$$

The number that we can add to this number to get to the next larger number is now only 2^{-1071}, since the next number is "even" (its last bit is 0):

$$\begin{array}{rccl}
 & 2^{-1018} & \times & \texttt{1.0000000000000000000000000000000000000000000000000001}|_2 \\
+ & 2^{-1018} & \times & \texttt{0.0000000000000000000000000000000000000000000000000000}|\texttt{1000}_2 \\
= & 2^{-1018} & \times & \texttt{1.0000000000000000000000000000000000000000000000000001}|\texttt{1000}_2 \\
\text{rounding to} & 2^{-1018} & \times & \texttt{1.0000000000000000000000000000000000000000000000000010}|_2
\end{array}$$

so $u(2^{-1018} + 2^{-1070}) = 2^{-1071}$, which is less than $u(2^{-1018})$.

We can verify this in R:

```r
x <- 2^-1018
ulp.x <- 2^-1070
(x + ulp.x) - x

#> [1] 7.90505e-323

(x + ulp.x / 2) - x

#> [1] 0

(x + (ulp.x / 2 + 2^-1074)) - x  # The parentheses before ulp.x are necessary!

#> [1] 7.90505e-323
```

R handily has a function that prints the bit patterns of numbers:

```r
#### Output is as follows:
#>      SS EE EE EE EE EE EE EE EE EE EE EE MM MM MM MM MM ...
rev(numToBits(x))

#>  [1] 00 00 00 00 00 00 00 00 00 01 00 01 00 00 00 00 00 00 00 00 00 00 00 00 00
#> [26] 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
#> [51] 00 00 00 00 00 00 00 00 00 00 00 00 00 00

rev(numToBits(ulp.x / 2 + 2^-1074))

#>  [1] 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
#> [26] 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
#> [51] 00 00 00 00 00 00 00 00 00 00 01 00 00 01

rev(numToBits(x + (ulp.x / 2 + 2^-1074)))

#>  [1] 00 00 00 00 00 00 00 00 00 01 00 01 00 00 00 00 00 00 00 00 00 00 00 00 00
#> [26] 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
#> [51] 00 00 00 00 00 00 00 00 00 00 00 00 00 01
```

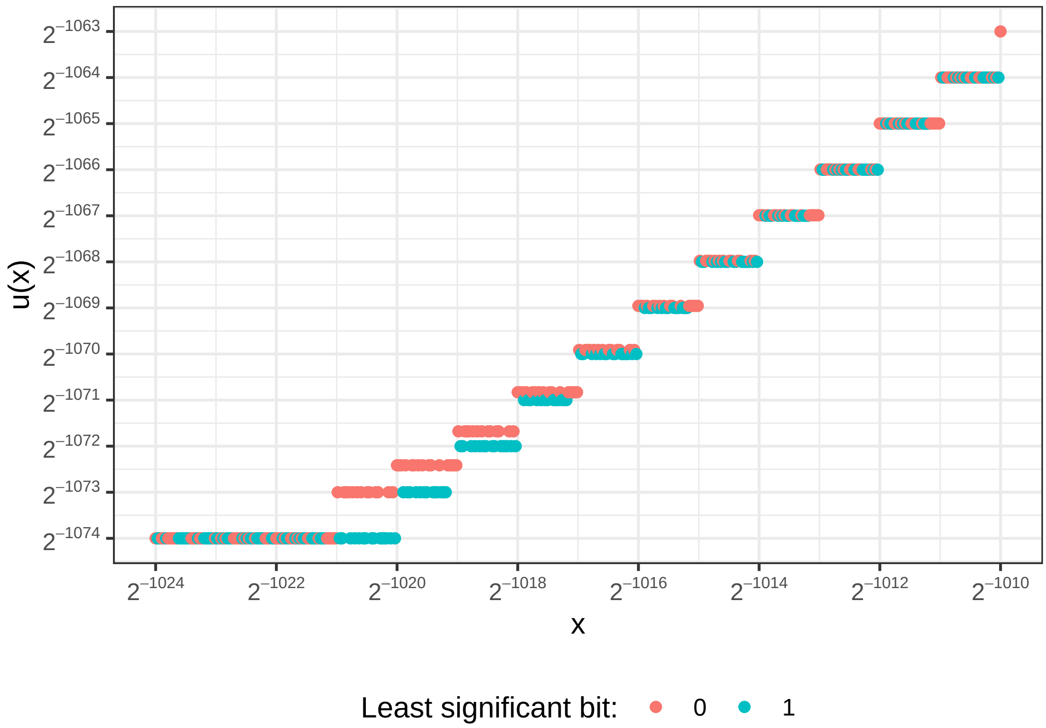
For the next larger number, we have

```r
x.next <- x + ulp.x

(x.next + ulp.x / 2) - x.next

#> [1] 7.90505e-323

#>      SS EE EE EE EE EE EE EE EE EE EE EE MM MM MM MM MM ...
rev(numToBits(x.next))

#>  [1] 00 00 00 00 00 00 00 00 00 01 00 01 00 00 00 00 00 00 00 00 00 00 00 00 00
#> [26] 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
#> [51] 00 00 00 00 00 00 00 00 00 00 00 00 00 01

rev(numToBits(x.next + ulp.x / 2))

#>  [1] 00 00 00 00 00 00 00 00 00 01 00 01 00 00 00 00 00 00 00 00 00 00 00 00 00
#> [26] 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
#> [51] 00 00 00 00 00 00 00 00 00 00 00 00 01 00
```

Let's plot the values of $u(x)$ again, this time coloring the points according to the least significant bit of x:

```r
lsbits <- vapply(x.grid, function(x) as.numeric(numToBits(x))[[1]], numeric(1))
```

```r
ggplot(data.frame(x = log2(x.grid), u = log2(u), lsbit = factor(lsbits)),
    aes(x = x, y = u, color = lsbit)) +
  geom_point() +
  scale_x_continuous(
    breaks = seq(-1024, -1010, by = 2),
    labels = function(x) parse(text = paste0("2^", x))
  ) +
  scale_y_continuous(
    breaks = function(x) seq(ceiling(min(x)), floor(max(x)), by = 1),
    labels = function(x) parse(text = paste0("2^", x))
  ) +
  scale_color_discrete(name = "Least significant bit:") +
  xlab("x") +
  ylab("u(x)") +
  theme_bw() +
  theme(aspect.ratio = 0.6, legend.position = "bottom")
```



Final note: All of this relies on the default rounding mode. There is, in fact, an R package `ieeeround` that lets you change the rounding mode (not recommended!). For example, setting the rounding mode to `FE.UPWARD` means that adding 2^{-1074} to (finite) x will always yield a number different from x. This even changes the default printer for `numerics`: $1 + 2^{-52}$ is now printed as `1.00...01` instead of `1`.

```r
1 + 2^-1074

#> [1] 1

(1 + 2^-1074) - 1

#> [1] 0

1 + .Machine$double.eps

#> [1] 1

(1 + .Machine$double.eps) - 1

#> [1] 2.220446e-16

# Remember that machine epsilon is 2^-52:
2^-52 == .Machine$double.eps

#> [1] TRUE
```

Now let's see what happens if we change the rounding mode:

```r
library(ieeeround)
fesetround(FE.UPWARD)

#> [1] 0

1 + 2^-1074

#> [1] 1.000001

(1 + 2^-1074) - 1  # we see that what we got was 1 + 2^-52

#> [1] 2.220447e-16

1 + .Machine$double.eps

#> [1] 1.000001

(1 + .Machine$double.eps) - 1  # same as above

#> [1] 2.220447e-16
```

Depending on the R setup, evaluating "`2^-52`" now returns a value that differs from `.Machine$double.eps`, because the `^`-function (exponentiation) is not evaluated precisely in this rounding mode![1] This is why changing the rounding mode is a bad idea if you don't know what you're doing!

```r
2^-52 == .Machine$double.eps

#> [1] FALSE

2^-52 - .Machine$double.eps

#> [1] 4.930381e-32
```

[1] See `https://inria.hal.science/hal-04159652v2/document` if you want to know more.