Solution 1: IEEE 754 Limits

Consider a system of IEEE 754-like floating-point machine numbers with 10 bits, where the first bit is the sign bit, the next $E = 4$ bits are the exponent, and the last $M = 5$ bits are the mantissa. As is typical for floating-point representations, the bias of the exponent is $2^{E-1} - 1$, and special values are represented equivalently to the IEEE 754 standard.

(a) Determine the value of the smallest positive number (both normalized and denormalized), the biggest positive denormalized number, and the biggest finite number represented in this system. What are each of their binary representations?

(b) What are the binary representations of the numbers 0, ∞, and `NaN`? How many `NaN`s are there?

(c) Write down the binary representations of the number 1, the smallest representable number larger than 1, and the largest representable number smaller than 1. What is the value of each of these numbers? What do you observe about the binary representations?

(d) Determine the number of normalized numbers in this system.

(e) Determine the number of denormalized numbers in this system.

We are given a floating-point system with 10 bits: 1 sign bit (S), $E = 4$ exponent bits, and $M = 5$ mantissa bits. The exponent bias is $2^{E-1} - 1 = 2^{4-1} - 1 = 7$. The structure is $SEEEEMMMMM$. The range of exponent values for normalized numbers is $e_{\text{min}} = 1 - \text{bias} = 1 - 7 = -6$ to $e_{\text{max}} = (2^E - 2) - \text{bias} = (16 - 2) - 7 = 14 - 7 = 7$. The exponent bit patterns range from `0001` (for $e = -6$) to `1110` (for $e = 7$). Denormalized numbers use the exponent bit pattern `0000`, which corresponds to the exponent value $e_{\text{min}} = -6$. Special values use the exponent bit pattern `1111`.

(a) Smallest and largest numbers:

- Smallest positive denormalized number: This occurs with sign 0, exponent bits `0000`, and the smallest non-zero mantissa `00001`.

  $\boxed{\text{Binary representation: } \texttt{0 0000 00001}}$

  Value: $(-1)^0 \times 2^{e_{\text{min}}} \times (0.00001)_2 = 1 \times 2^{-6} \times (1 \times 2^{-5}) = 2^{-11} = 1/2048 \approx 0.000488$.

- Smallest positive normalized number: This occurs with sign 0, the smallest normalized exponent pattern `0001`, and mantissa `00000`.

  $\boxed{\text{Binary representation: } \texttt{0 0001 00000}}$

  Value: $(-1)^0 \times 2^{e_{\text{min}}} \times (1.00000)_2 = 1 \times 2^{-6} \times 1 = 2^{-6} = 1/64 = 0.015625$.

- Largest positive denormalized number: This occurs with sign 0, exponent bits `0000`, and the largest mantissa `11111`.

  $\boxed{\text{Binary representation: } \texttt{0 0000 11111}}$

  Value: $(-1)^0 \times 2^{e_{\text{min}}} \times (0.11111)_2 = 2^{-6} \times (1 - 2^{-5}) = 1/64 - 1/2048 \approx 0.015137$. We observe that the largest denormalized number is smaller than the smallest normalized number, where the difference is the "resolution" of the denormalized numbers.

- Largest finite positive number: This occurs with sign 0, the largest normalized exponent pattern `1110`, and the largest mantissa `11111`.

  $\boxed{\text{Binary representation: } \texttt{0 1110 11111}}$

  Value: $(-1)^0 \times 2^{e_{\text{max}}} \times (1.11111)_2 = 1 \times 2^7 \times (1 + 2^{-1} + 2^{-2} + 2^{-3} + 2^{-4} + 2^{-5}) = 2^7 \times (1 + 31/32) = 2^7 \times (63/32) = 2^7 \times 63 \times 2^{-5} = 63 \times 2^2 = 63 \times 4 = 252$. Alternatively: $2^7 \times (2 - 2^{-5}) = 2^8 - 2^{7-5} = 256 - 4 = 252$.

(b) Special values:

- Zero (+0): Sign 0, exponent `0000`, mantissa `00000`.

  $\boxed{\text{Binary: } \texttt{0 0000 00000}}$

  Note that -0 has representation `1 0000 00000`.
- Infinity ($\pm\infty$): Exponent `1111`, mantissa `00000`.

  $\boxed{\text{Binary for } +\infty\text{: } \texttt{0 1111 00000}}$ $\boxed{\text{Binary for } -\infty\text{: } \texttt{1 1111 00000}}$
- Not a Number (NaN): Exponent `1111`, mantissa non-zero.

  $\boxed{\text{Binary: } \texttt{S 1111 MMMMM} \text{ where } \texttt{MMMMM} \neq \texttt{00000}}$

  Number of NaNs: There are 2 choices for the sign bit and $2^5 - 1 = 31$ choices for the non-zero mantissa. Total NaNs $= 2 \times 31 = 62$. Most likely error by students here would be missing the sign bit.

(c) Numbers around 1:

- Number 1: Value is $1.0 = (-1)^0 \times 2^0 \times 1.0$. The exponent is $e = 0$. The exponent bits are $e + \text{bias} = 0 + 7 = 7$, which is `0111`. The mantissa is `00000`.

  $\boxed{\text{Binary representation: } \texttt{0 0111 00000}}$

  Value: 1.
- Smallest representable number larger than 1: We increment the least significant bit of the mantissa of 1.

  $\boxed{\text{Binary representation: } \texttt{0 0111 00001}}$

  Value: $(-1)^0 \times 2^{7-7} \times (1.00001)_2 = 1 \times (1 + 2^{-5}) = 1 + 1/32 = 33/32$.
- Largest representable number smaller than 1: This is the number immediately preceding 1 in the sequence of representable numbers. It has the next lower exponent ($e = -1$, bits `0110`) and the maximum mantissa (`11111`).

  $\boxed{\text{Binary representation: } \texttt{0 0110 11111}}$

  Value: $(-1)^0 \times 2^{6-7} \times (1.11111)_2 = 2^{-1} \times (1 + 31/32) = 1/2 \times (63/32) = 63/64$. We observe that the distance between 1 and the next larger number is twice the distance between 1 and the next smaller number.
- Observation: Interpreting the binary representations as unsigned integers, the three numbers `0 0110 11111`, `0 0111 00000`, and `0 0111 00001` are consecutive. This holds generally for consecutive floating-point numbers within the same exponent range.

(d) Number of normalized numbers: There are 2 choices for the sign bit. The exponent bits can range from `0001` to `1110`. There are $2^E - 2 = 2^4 - 2 = 16 - 2 = 14$ possible patterns. The mantissa can be any of the $2^M = 2^5 = 32$ patterns. Total normalized numbers = (Sign Choices) $\times$ (Exponent Choices) $\times$ (Mantissa Choices) $= 2 \times 14 \times 32 = 896$.

(e) Number of denormalized numbers: There are 2 choices for the sign bit. The exponent bits must be `0000` (1 choice). The mantissa must be non-zero. There are $2^M - 1 = 2^5 - 1 = 32 - 1 = 31$ possible patterns. Total denormalized numbers = (Sign Choices) $\times$ (Exponent Choices) $\times$ (Mantissa Choices) $= 2 \times 1 \times 31 = 62$.

Solution 2: Decimal to Binary Conversion

Converting a decimal number $xxx.yyyy_{10}$ to a binary number involves two parts:

Integer Part Repeatedly divide the integer part xxx by 2. The remainders, read in reverse order, form the binary integer representation.

Fractional Part Repeatedly multiply the fractional part $yyyy$ by 2. The integer part of the result (0 or 1) is the next binary digit after the point. Remove the integer part and continue with the remaining fractional part. Repeat until the fractional part becomes 0, a pattern repeats, or sufficient precision is achieved.

(a) Represent the number **18.6** in the binary system with at least seven bits after the binary point (so you get a number $\texttt{...xxxxx.yyyyyyy}_2$ with at least seven `y`). What do you notice?

(b) Consider a system of IEEE 754-like floating-point machine numbers with 14 bits, with the first bit for the sign, the next $E = 5$ bits for the exponent, and the last $M = 8$ bits for the mantissa. As is typical for floating-point representations, the exponent is biased by $2^{E-1} - 1$.

   (i) What is the "machine epsilon" of this system?

   (ii) Represent the number **18.6** as a binary machine number with $M = 8$ using your result from (a) by

   - Truncation
   - Rounding to the nearest.

   What are the corresponding values (in decimal)?

   (iii) What are the absolute and relative errors in subtask (b)ii? How do they relate to the machine epsilon?

(a) **Manual Conversion of 18.6:** We convert the integer and fractional parts separately using the division/multiplication method.

Part	**Operation**	**Bit/Rem.**	**Order**
Integer (18)	$18 \div 2 = 9\,\text{R}\,0$	0	↑
	$9 \div 2 = 4\,\text{R}\,1$	1	
	$4 \div 2 = 2\,\text{R}\,0$	0	
	$2 \div 2 = 1\,\text{R}\,0$	0	
	$1 \div 2 = 0\,\text{R}\,1$	1	
Fractional (0.6)	$0.6 \times 2 = 1\,.2$	1	↓
	$0.2 \times 2 = 0\,.4$	0	
	$0.4 \times 2 = 0\,.8$	0	
	$0.8 \times 2 = 1\,.6$	1	
	$0.6 \times 2 = 1\,.2$	1	(repeats)
	$0.2 \times 2 = 0\,.4$	0	
	$0.4 \times 2 = 0\,.8$	0	

Reading the integer remainders bottom-up (10010_2) and fractional bits top-down ($0.1001100..._2$): $18.6_{10} = 10010.100110010011..._2$. With exactly 7 bits after the binary point, we have:

$$18.6_{10} \approx 10010.1001100_2$$

Observation: The binary representation of the fractional part 0.6 is infinitely repeating ($0.1\overline{0011}_2$), which means 18.6 cannot be represented exactly in a finite binary floating-point system.

(b) Floating-point representation:

   (i) **Machine Epsilon:** The machine epsilon, ϵ, is the difference between 1 and the next larger representable number. In a binary system with M mantissa bits, it is given by $\epsilon = 2^{-M}$. For this system, $M = 8$, so the machine epsilon is:

$$\epsilon = 2^{-8} = \frac{1}{256} = 0.00390625$$

   (ii) **Machine Representation ($M = 8$):** The binary number is $10010.100110011..._2$. To represent this in the floating-point format (1 sign, 5 exponent, 8 mantissa), we normalize it: $10010.100110011..._2 = 1.0010100110011..._2 \times 2^4$.

   - Sign bit $S = 0$ (positive number).
   - Exponent $e = 4$. The bias is $B = 2^{E-1} - 1 = 2^{5-1} - 1 = 15$. The biased exponent is $E = e + B = 4 + 15 = 19$. In binary (5 bits): 10011_2.
   - Mantissa: We need the first $M = 8$ bits after the leading 1: 00101001. The bits following are 10011....

   The target format is:

S	E	M

   i. **Truncation:** We simply discard the bits beyond the 8th mantissa bit. Mantissa: 00101001. Representation:

0	10011	00101001

   Value: $1.00101001_2 \times 2^4 = (1 + 2^{-3} + 2^{-5} + 2^{-8}) \times 16 = (1 + 1/8 + 1/32 + 1/256) \times 16 = (256 + 32 + 8 + 1)/256 \times 16 = 297/16 = \mathbf{18.5625}_{10}$.

ii. **Rounding (to nearest, ties to even):** We look at the bits after the 8th mantissa bit: $1.00101001|10011..._2$. The first discarded bit is 1 (the 'guard' bit). Since it's 1, we need to round up. Mantissa: $00101001 + 1 = 00101010$. Representation: $\boxed{0}\boxed{\ 10011\ }\boxed{\ 00101010\ }$
Value: $1.00101010_2 \times 2^4 = (1 + 2^{-3} + 2^{-5} + 2^{-7}) \times 16 = (1 + 1/8 + 1/32 + 1/128) \times 16 = (128 + 16 + 4 + 1)/128 \times 16 = 149/8 = \mathbf{18.625}_{10}$. (Note: If the guard bit was 1 and all subsequent bits were 0, we would round to the nearest *even* number, meaning we'd round up only if the 8th mantissa bit was 1. Here, the subsequent bits are not all zero, so we round up regardless of the 8th bit).

(iii) **Error Analysis:** Let the true value be $T = 18.6$. Machine epsilon $\epsilon = 1/256 \approx 0.003906$.

- **Truncation ($V_T = 18.5625$):** Absolute Error: $|V_T - T| = |18.5625 - 18.6| = |-0.0375| = \mathbf{0.0375}$. Relative Error: $\frac{|V_T-T|}{|T|} = \frac{0.0375}{18.6} \approx \mathbf{0.002016}$.
- **Rounding ($V_R = 18.625$):** Absolute Error: $|V_R - T| = |18.625 - 18.6| = \mathbf{0.025}$. Relative Error: $\frac{|V_R-T|}{|T|} = \frac{0.025}{18.6} \approx \mathbf{0.001344}$.

Relation to Machine Epsilon: The maximum relative error for rounding to nearest is expected to be $\epsilon/2 = (1/256)/2 = 1/512 \approx 0.001953$. The relative error for rounding (0.001344) is indeed less than $\epsilon/2$. The maximum relative error for truncation is expected to be $\epsilon = 1/256 \approx 0.003906$. The relative error for truncation (0.002016) is less than ϵ, but larger than $\epsilon/2$.

Solution 3: Floating Point Numbers in R

The builtin R function `numToBits` converts a numeric vector to a "`raw`" vector containing the binary representation of the numbers, with bits ordered from least to most significant. For example, $1.125 = 1 + 2^{-3}$ is represented as

```r
numToBits(1.125)

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

You can see the sign bit (at the very end) is `0`, the exponent 1 is represented as `01111111111` (in reverse order), and the significand 1.001000... is represented as `001000...`. Note that `raw` vectors can be converted to `logicals`, `integers`, and `numerics` by using the typical R type conversion functions `as.logical()` etc.

Write a function `r()` (short for "representation", single letter for convenience) that takes a numeric scalar and prints it in the format `S 2^EEE * F.FFFFFF...`, where `S` is the sign ("+" or "-"), `EEE` is the exponent (in decimal) of the scaling factor, and `F.FFFFFF...` is the significand (in binary, including the implicit/hidden bit, and including all trailing zeros to 52 digits after the binary point). Your function should handle normalized and denormalized numbers, as well as ±0. The scaling factor of denormalized numbers should be displayed as `2^-1022`.

```r
r(1.125)  # typical case
```

```
#> + 2^0 * 1.0010000000000000000000000000000000000000000000000000
```

```r
r(-1.125)  # negative number
```

```
#> - 2^0 * 1.0010000000000000000000000000000000000000000000000000
```

```r
r(22)
```

```
#> + 2^4 * 1.0110000000000000000000000000000000000000000000000000
```

```r
r(0)  # positive zero
```

```
#> + 0 * 0.0000000000000000000000000000000000000000000000000000
```

```r
r(-0)  # negative zero
```

```
#> - 0 * 0.0000000000000000000000000000000000000000000000000000
```

```r
r(8e-323)  # denormalized number
```

```
#> + 2^-1022 * 0.0000000000000000000000000000000000000000000000010000
```

When trying to understand floating point numbers, this function is very helpful. We encourage you to play around with it, e.g. in the next exercise sheet.

```r
#' Prints the IEEE 754 double precision representation of a number.
#' Formats the output as S 2^E * F.FFFF...
#' Handles normalized, denormalized, and zero values.
#'
#' @param x.val A scalar numeric value.
#'
#' @return Invisible x.val
rRepresentation <- function(x.val) {
  checkmate::assertNumber(x.val)

  # Exercise did not tell us to handle these, but it is more convenient
  if (is.na(x.val)) {
    cat("  NA\n")
    return(invisible(x.val))
  }

  if (is.infinite(x.val)) {
    cat(sprintf("%s Inf\n", ifelse(x.val > 0, "+", "-")))
    return(invisible(x.val))
  }

  if (is.nan(x.val)) {
    cat("  NaN\n")
    return(invisible(x.val))
  }

  # Get 64 bits from the number, reverse to get MSB first
  bits.logical <- as.logical(rev(numToBits(x.val)))

  # Extract components: Sign (1 bit), Exponent (11 bits), Mantissa (52 bits)
  sign.bit <- bits.logical[1]
  exponent.bits <- bits.logical[2:12]
  mantissa.bits <- bits.logical[13:64]

  # Determine sign character
  sign.char <- if (sign.bit) "-" else "+"

  mantissa.str <- paste(as.integer(mantissa.bits), collapse = "")

  # Calculate integer value of exponent bits
  exponent.int <- sum(exponent.bits * 2^(10:0))

  # Check if mantissa bits are all zero
  is.mantissa.zero <- all(!mantissa.bits)

  # Handle different number types based on exponent and mantissa
  if (exponent.int == 0 && is.mantissa.zero) {
    # --- Zero ---
    hidden.bit.char <- "0"
    exponent.str <- "0"
  } else if (exponent.int == 0) {
    # --- Denormalized ---
    # Exponent value is fixed at -1022, significand starts with 0.
    hidden.bit.char <- "0"
    exponent.str <- "2^-1022"
  } else {
    # --- Normalized ---
    # Exponent value is derived from bits minus bias, significand starts with 1.
    # (Exponent is not 0 and not 2047 due to finite check)
    exponent.str <- sprintf("2^%d", exponent.int - 1023) # Bias for double precision
    hidden.bit.char <- "1"  # default to "1", only "0" for denormalized numbers / 0
  }

  output.str <- sprintf("%s %s * %s.%s", sign.char, exponent.str, hidden.bit.char, mantissa.str)

  # Print the formatted string
  cat(output.str, "\n")

  # Return NULL invisibly (common practice for functions primarily used for side effects)
  invisible(x.val)
}

# Create the alias 'r' for convenience
r <- rRepresentation
```

Now let's test the function with the examples provided in the exercise description:

Example 1:

```r
r(1.125)

#> + 2^0 * 1.0010000000000000000000000000000000000000000000000000
```

Example 2:

```r
r(-1.125)

#> - 2^0 * 1.0010000000000000000000000000000000000000000000000000
```

Example 3:

```r
r(22)

#> + 2^4 * 1.0110000000000000000000000000000000000000000000000000
```

Example 4:

```r
r(0)

#> + 0 * 0.0000000000000000000000000000000000000000000000000000
```

Example 5:

```r
r(-0)

#> - 0 * 0.0000000000000000000000000000000000000000000000000000
```

Example 6: *Note:* 8×10^{-323} *is approximately* 2^{-1070}, *which falls into the denormalized range.*

```r
r(8e-323)

#> + 2^-1022 * 0.0000000000000000000000000000000000000000000000010000
```