Solution 1: Euclid's Algorithm

The following is the pseudocode for Euclid's GCD algorithm:

Algorithm 1 *Euclid*($m : \mathbb{N}$, $n : \mathbb{N}$)

if $m = 0$ **then**
  return n
else if $n = 0$ **then**
  return m
else if $m \leq n$ **then**
  return EUCLID(m, $n \bmod m$)
else
  return EUCLID($m \bmod n$, n)
end if

(a) Write out a table of the values of m and n, as well as the ultimate action (return statement) taken by the algorithm, for the inputs $m = 270$ and $n = 192$:

m	n	Action
270	192	**return** EUCLID(78, 192)
...	...	...

(b) Write this algorithm in R.

(a) Trace of Euclid's algorithm for $m = 270$ and $n = 192$:

m	n	Action
270	192	**return** EUCLID(78, 192)
78	192	**return** EUCLID(78, 36)
78	36	**return** EUCLID(6, 36)
6	36	**return** EUCLID(6, 0)
6	0	**return** 6

(b) Euclid's algorithm implemented in R:

```r
euclid <- function(m, n) {
  checkmate::assertIntegerish(m, lower = 0, tol = 0)
  checkmate::assertIntegerish(n, lower = 0, tol = 0)

  # Base cases
  if (m == 0) {
    return(n)
  } else if (n == 0) {
    return(m)
  }

  # Recursive cases
  if (m <= n) {
```

```
    return(euclid(m, n %% m))
  } else {
    return(euclid(m %% n, n))
  }
}

# Example usage
euclid(48, 18)

## [1] 6

euclid(101, 103)

## [1] 1

euclid(0, 10)

## [1] 10

euclid(10, 0)

## [1] 10
```

Solution 2: Palindrome

A "palindrome" is a word or sentence that reads the same forwards and backwards, for example "radar".

Write the pseudocode for an algorithm that checks if an array of letters $A[1 \dots n]$ (e.g. $A = [\text{'r'}, \text{'a'}, \text{'d'}, \text{'a'}, \text{'r'}]$) is a palindrome…

(a) …using any of the pseudocode constructs from the lecture (but without recursion).

(b) …using none of the loop constructs (no **for** or **while**), by calling your algorithm recursively. You will likely need to use the additional construct **del** $A[i]$, which deletes the i-th element from the array A and turns it from a length n array into a length $n-1$ array.

You can check for (in)equality of letters, e.g. $A[3] = A[7]$ or $A[3] \neq A[7]$, and can use standard arithmetic operations. You may find floor division: "$\lfloor x/y \rfloor$", yielding the greatest integer less than or equal to x/y, to be useful.

Solution for the palindrome algorithm:

(a) Algorithm using loop constructs:

Algorithm 2 *IsPalindrome*($A[1 \dots n]$: array of letters)

```
for i = 1 to ⌊n/2⌋ do
    if A[i] ≠ A[n − i + 1] then
        return false
    end if
end for
return true
```

(b) Recursive algorithm without loop constructs:

Algorithm 3 *IsPalindromeRecursive*($A[1...n]$: array of letters)

if $n \leq 1$ **then**
    return true
end if
if $A[1] \neq A[n]$ **then**
    return false
end if
del $A[n]$ ▷ Ordering of deletion is important: If we delete $A[1]$ first, then $A[n-1]$ needs to be deleted next.
del $A[1]$
return IsPalindromeRecursive(A)

Bonus content: Implementation in R:

```r
# Iterative approach
isPalindrome <- function(letters) {
  checkmate::assertCharacter(letters, n.chars = 1)

  n <- length(letters)
  for (i in seq_len(floor(n/2))) {
    if (letters[[i]] != letters[[n-i+1]]) {
      return(FALSE)
    }
  }
  return(TRUE)
}

# Recursive approach without loops
isPalindromeRecursive <- function(letters) {
  checkmate::assertCharacter(letters, n.chars = 1)

  n <- length(letters)
  if (n <= 1) {
    return(TRUE)
  }
  if (letters[[1]] != letters[[n]]) {
    return(FALSE)
  }
  letters <- letters[-n]
  letters <- letters[-1]
  return(isPalindromeRecursive(letters))
}

# Examples
example.texts <- c("radar", "hello", "level", "a", "", "madam", "racecar", "hello world")
example.texts <- strsplit(example.texts, "")

example.texts[[1]]

## [1] "r" "a" "d" "a" "r"

## Iterative approach results
for (text in example.texts) {
  cat(sprintf("'%s' is %sa palindrome\n", paste(text, collapse = ""),
    ifelse(isPalindrome(text), "", "not ")))
}
```

```
## 'radar' is a palindrome
## 'hello' is not a palindrome
## 'level' is a palindrome
## 'a' is a palindrome
## '' is a palindrome
## 'madam' is a palindrome
## 'racecar' is a palindrome
## 'hello world' is not a palindrome

## Recursive approach results
for (text in example.texts) {
  cat(sprintf("'%s' is %sa palindrome\n", paste(text, collapse = ""),
    ifelse(isPalindromeRecursive(text), "", "not ")))
}

## 'radar' is a palindrome
## 'hello' is not a palindrome
## 'level' is a palindrome
## 'a' is a palindrome
## '' is a palindrome
## 'madam' is a palindrome
## 'racecar' is a palindrome
## 'hello world' is not a palindrome
```

Solution 3: Numeric Differentiation

The derivative of a function f at a point x is defined as:

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

Write an R function `derive(f, x0, h)` that computes the derivative of a given R-function `f` at a point `x0` for a given step size `h` by using the definition above.

Check your implementation by computing the derivative of the following functions and compare against the analytical solutions. What value of `h` is needed to get a good approximation of the derivative? What do you observe about the relative / abolute difference between the numeric and the analytical derivative?

$f(x)$	x_0	Analytical $f'(x)$
x^3	1	$3x^2$
$\cos(x^2)$	10^{-3}	$-2x\sin(x^2)$
$\sqrt{x}$	10^{-6}	$\frac{1}{2\sqrt{x}}$
$\sqrt{x-1000}$	$1000 + 10^{-6}$	$\frac{1}{2\sqrt{x-1000}}$

Numeric differentiation:

```
derive <- function(f, x0, h) {
  (f(x0 + h) - f(x0)) / h
}
```

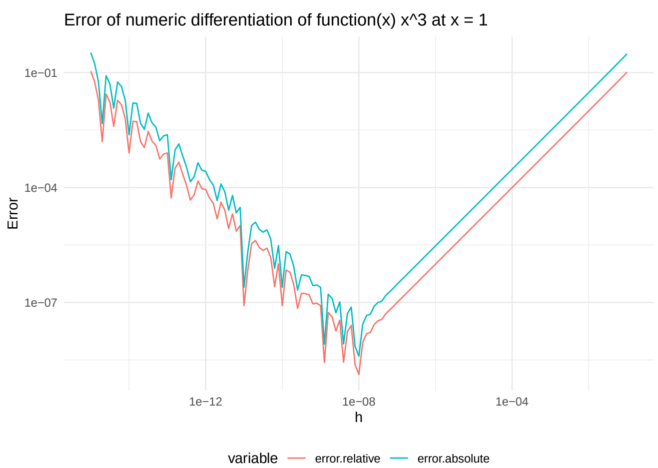
Note that the $h > 0$ is not strictly necessary, but we restrict ourselves to positive h here ("forward differentiation").

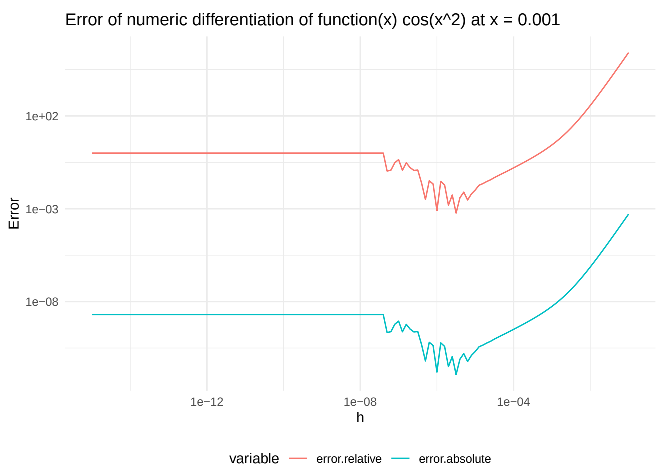
We use the following function to plot both the relative and the absolute error of the numeric derivative vs. the analytical derivative:

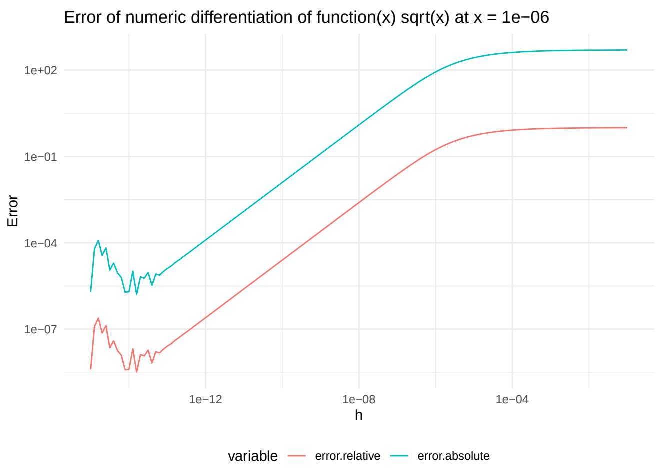
```r
suppressPackageStartupMessages(library("ggplot2"))
library("data.table")
plotError <- function(f, x0, f.prime) {
  h <- 10^seq(-15, -1, by = 0.1)
  error.relative <- abs(derive(f, x0, h) - f.prime(x0)) / abs(f.prime(x0))
  error.absolute <- abs(derive(f, x0, h) - f.prime(x0))
  data <- data.table(h = h, error.relative = error.relative, error.absolute = error.absolute)

  ggplot(melt(data, id.vars = "h"), aes(x = h, y = value, color = variable)) +
    geom_line() +
    scale_x_log10() +
    scale_y_log10() +
    xlab("h") +
    ylab("Error") +
    theme_minimal() +
    ggtitle(sprintf("Error of numeric differentiation of %s at x = %.2g", deparse(substitute(f)), x0)) +
    theme(legend.position = "bottom", aspect.ratio = 0.6)
}
```

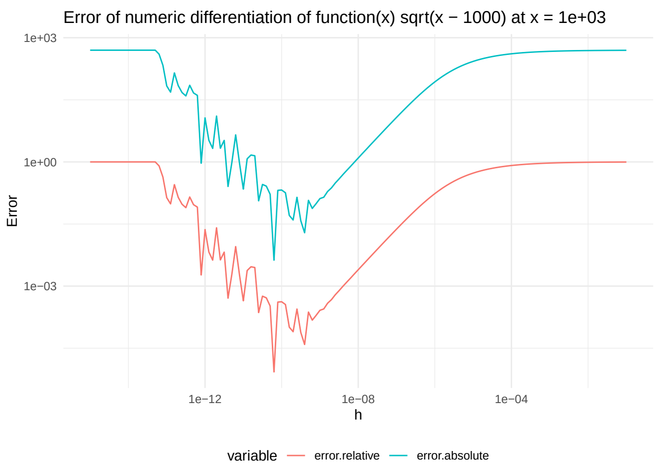
We now plot the error for the given functions:

```r
plotError(function(x) x^3, 1, function(x) 3 * x^2)
plotError(function(x) cos(x^2), 10^-3, function(x) -2 * x * sin(x^2))
plotError(function(x) sqrt(x), 10^-6, function(x) 1 / (2 * sqrt(x)))
plotError(function(x) sqrt(x - 1000), 1000 + 10^-6, function(x) 1 / (2 * sqrt(x - 1000)))
```









Observations:

- There is a trade-off between h too large (approximation error) and h too small (round-off error).
- $\cos(x^2)$ is almost 1 for $x = 10^{-3}$, so there is a large cancellation error when computing $f(x_0 + h) - f(x_0)$.
- Numbers for $sqrt x$ at $x = 10^{-6}$ are very small, so there is little round-off error. (The division by h makes numbers larger again; this only affects the absolute error). The best result is obtained for small h which minimizes approximation error.
- Shifting input by 10^{-6} in case of $sqrt x - 1000$ makes rounding error more prominent and the best h is larger.

Solution 4: Matrix Product

(a) Write a function that computes the product of two numeric matrices, `a` and `b`. Do not use R's built-in matrix multiplication functions and compute the product element-wise, instead.

(b) Compare the runtime performance of your function with R's built-in matrix multiplication function `a %*% b`: Perform a systematic benchmark for squared matrices of sizes $n \in \{2^1, 2^2, \ldots, 2^{12}\}$ (only up to 2^8 for your own function) and plot the results in a log-log plot. What do you observe? The `microbenchmark` package may help you here.

(a)
```r
matrixMultiply <- function(a, b) {
  ## We skip checkmate since we want to test performance later
#  checkmate::assertMatrix(a, mode = "numeric")
#  checkmate::assertMatrix(b, mode = "numeric", nrows = ncol(a))

  # Initialize result matrix with zeros
  result <- matrix(0, nrow = nrow(a), ncol = ncol(b))

  # Perform matrix multiplication
  for (i in seq_len(nrow(a))) {
    for (j in seq_len(ncol(b))) {
```

```r
      for (k in seq_len(ncol(a))) {
        result[i, j] <- result[i, j] + a[i, k] * b[k, j]
      }
    }
  }

  return(result)
}

# Test with small matrices to verify correctness
a <- matrix(1:6, nrow = 2)
b <- matrix(7:12, nrow = 3)
matrixMultiply(a, b)

##      [,1] [,2]
## [1,]   76  103
## [2,]  100  136

a %*% b

##      [,1] [,2]
## [1,]   76  103
## [2,]  100  136
```

(b) Now let's benchmark our implementation against R's built-in matrix multiplication. The solution sheet computes the benchmark for larger matrices than required in the exercise, for better illustration of the asymptotic behavior.

```r
library("microbenchmark")
library("ggplot2")
library("data.table")

ooms.custom <- 9
ooms.builtin <- 13

matrices.a <- lapply(1:ooms.builtin, function(ex) matrix(rnorm(2^ex * 2^ex), nrow = 2^ex))
matrices.b <- lapply(1:ooms.builtin, function(ex) matrix(rnorm(2^ex * 2^ex), nrow = 2^ex))
```

In the following, we construct the microbenchmark() call. `substitute()` is the elegant approach here, but you could also use `eval(str2lang(...))` or even hardcode the expressions.

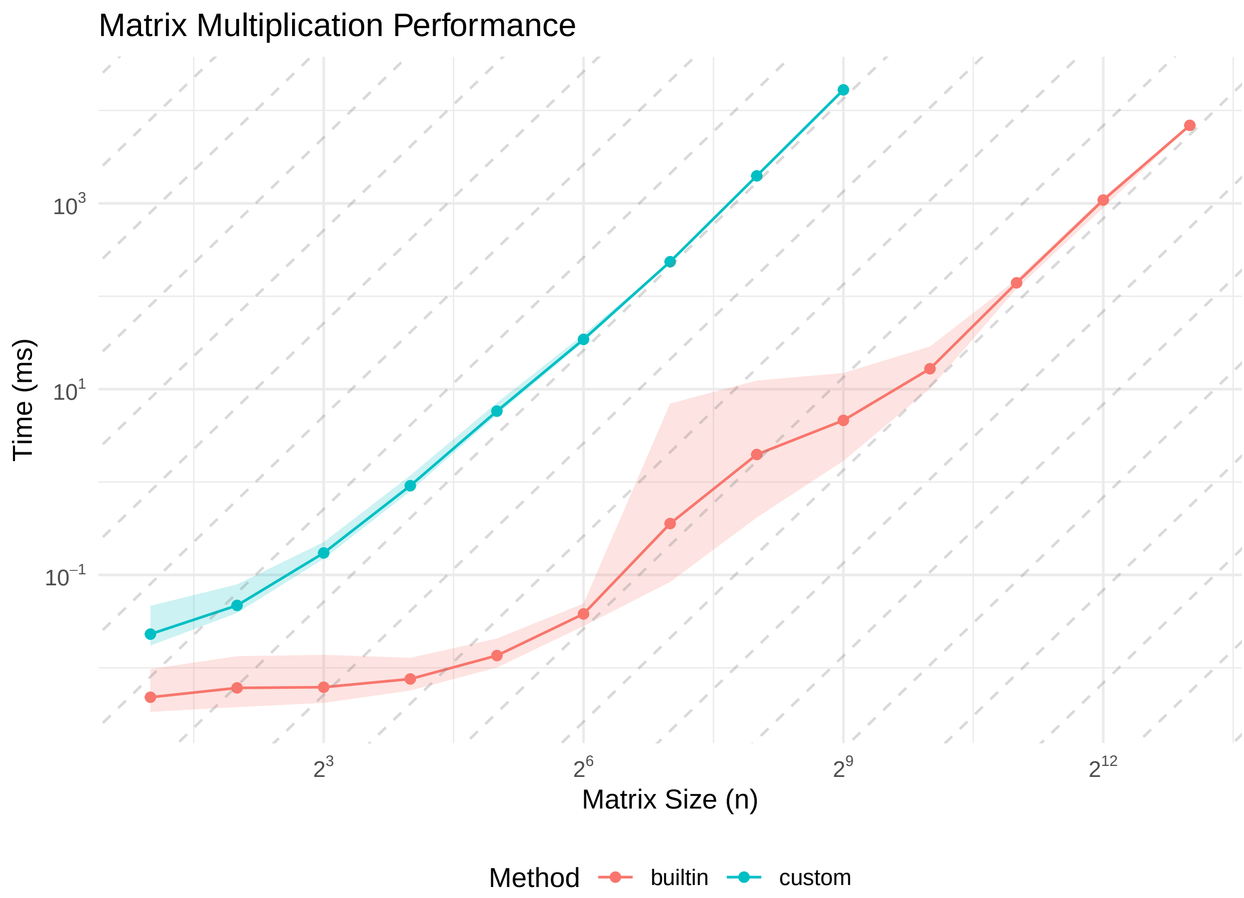
```r
expressions.custom <- lapply(1:ooms.custom, function(ex) {
  substitute(matrixMultiply(matrices.a[[ex]], matrices.b[[ex]]), list(ex = ex))
})
names(expressions.custom) <- paste0("custom_", seq_along(expressions.custom))

expressions.builtin <- lapply(seq_along(matrices.a), function(ex) {
  substitute(matrices.a[[ex]] %*% matrices.b[[ex]], list(ex = ex))
})
names(expressions.builtin) <- paste0("builtin_", seq_along(expressions.builtin))

expressions <- c(expressions.custom, expressions.builtin)

## Use fewer repetitions for the last few expressions, since they are very slow
times <- c(rep(100, ooms.custom - 3), 3, 3, 3, rep(100, ooms.builtin - 3), 3, 3, 3)
```

Now we run the benchmark. (This can take a bit of time.)

```r
bmr <- microbenchmark(list = expressions, times = times)

## summary() gives us a data.frame with various statistics
bms <- setDT(summary(bmr, include_cld = FALSE, unit = "ms"))

## Extract the matrix size from the expression names
bms[, size := 2 ^ as.numeric(sub(".*_", "", expr))]

## Extract the method from the expression names
bms[, method := sub("_.*", "", expr)]
```

In the following, we plot the median runtime vs. matrix size, and include IQR uncertainty ribbons. We include dashed lines with slope $y \propto x^3$, this should be the theoretical slope for large matrices.

```r
# log-log plot of median time vs. matrix size
ggplot(bms, aes(x = size, y = median, color = method)) +
  geom_ribbon(aes(ymin = lq, ymax = uq, fill = method), alpha = 0.2, color = NA) +
  geom_point() +
  geom_line() +
  scale_x_continuous(
    trans  = "log2",
    breaks = scales::breaks_log(base = 2),
    labels = scales::label_log(base  = 2)  # 2^k notation
  ) +
  scale_y_continuous(
    trans = "log10",
    breaks = scales::breaks_log(base = 10),
    labels = scales::label_log(base  = 10)
  ) +
  geom_abline(slope = 3 * log10(2), intercept = (-20):8, alpha = 0.3, color = "grey50",
    linetype = "dashed") +
  labs(
    title = "Matrix Multiplication Performance",
    x = "Matrix Size (n)",
    y = "Time (ms)",
    color = "Method"
  ) +
  guides(fill = "none") +
  theme_minimal() +
  theme(legend.position = "bottom", aspect.ratio = 0.6)
```



Note: The specific plot depends a lot on the system the code is run on, but the general shape should be the same.

Observations:

- R's builtin implementation is faster for matrices of all sizes.
- For small matrices, matrix size does not have a large influence on runtime, there seems to be a constant overhead.
- There is some anomalous behavior around matrices of size 2^7 that affects some of the measurements. This is likely due to caching effects.
- Asymptotically for large matrices, the runtime should grow as n^3 (dashed lines), for both our implementation and the builtin one.