# ESS212-HW2

Yiting Yu

2024-02-09

Yiting's Github repository url

## Problem 1

$$\begin{aligned}L[n+2] =& L[n+1]+L[n]\\ L[0] =& 2\\ L[1] =& 1\end{aligned}$$

### a)

$$\begin{aligned}L[2] &= L[0]+L[1] = 3\\ L[3] &= L[2]+L[1] = 4\\ L[4] &= L[3]+L[2] = 7\end{aligned}$$

### b)

$$\begin{aligned}L[n] =& A(\frac{1+\sqrt{5}}{2})^n - B(\frac{1-\sqrt{5}}{2})^n\\ L[0] =& A(\frac{1+\sqrt{5}}{2})^0 - B(\frac{1-\sqrt{5}}{2})^0\\ =& A-B\\ \Rightarrow& A-B=2\\ L[1] =& A\frac{1+\sqrt{5}}{2} - B\frac{1-\sqrt{5}}{2} = 1\\ \Rightarrow& \begin{cases}A-B=2\\ A\frac{1+\sqrt{5}}{2}-B\frac{1-\sqrt{5}}{2}=1\end{cases} \qquad \begin{cases}A=1\\ B=-1\end{cases}\\ verify: L[n] =& (\frac{1+\sqrt{5}}{2})^n + (\frac{1-\sqrt{5}}{2})^n\\ L[n+1] =& (\frac{1+\sqrt{5}}{2})^{n+1} + (\frac{1-\sqrt{5}}{2})^{n+1}\\ L[n]+L[n+1] =& (\frac{1+\sqrt{5}}{2})^n(1+\frac{1+\sqrt{5}}{2}) + (\frac{1-\sqrt{5}}{2})^n(1+\frac{1-\sqrt{5}}{2})\\ =& (\frac{1+\sqrt{5}}{2})^n\frac{3+\sqrt{5}}{2} + (\frac{1-\sqrt{5}}{2})^n\frac{3-\sqrt{5}}{2}\\ =& (\frac{1+\sqrt{5}}{2})^{n+2} + (\frac{1-\sqrt{5}}{2})^{n+2} = L(n+2)\end{aligned}$$

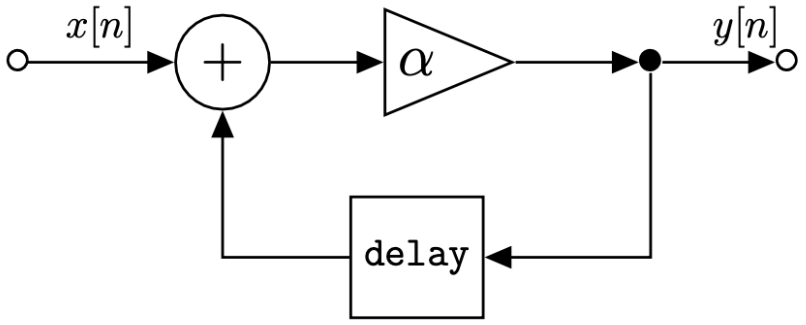


Figure 1: Figure 1: Block diagram for an input output system Problem 2(a)

## Problem 2

**a)**

$$y[n] = \alpha(y[n-1] + x[n])$$

**b)**

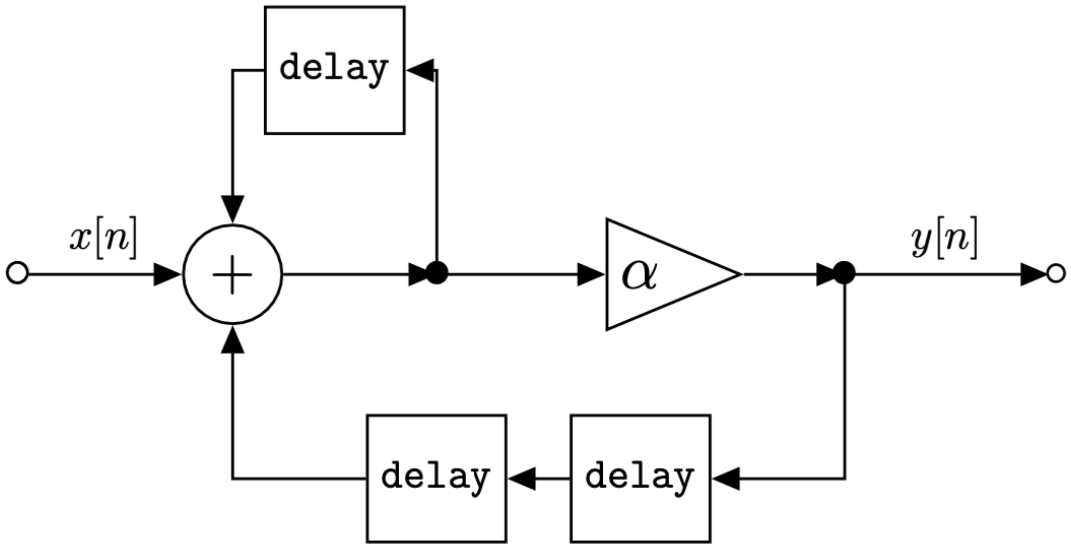


Figure 2: Figure 2: Block diagram for an input output system Problem 2(b)

$$y[n] = \alpha(x[n] + x[n-1] + y[n-2])$$

## Problem 3

**a)**

$$y[n] = 0.2y[n-1] + \beta x[n]$$

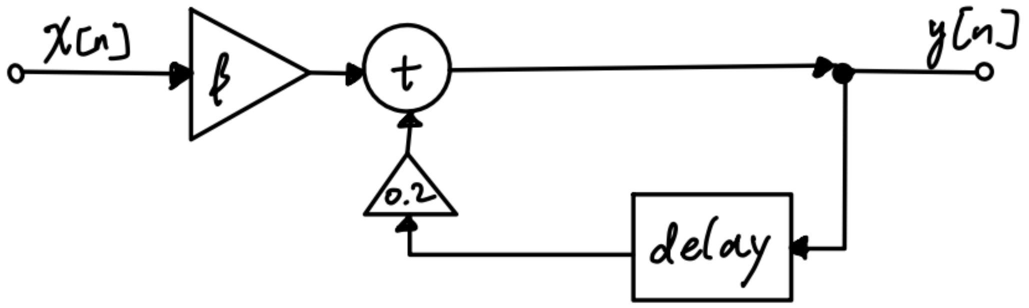


Figure 3: Figure 3: Block diagram for an input output system Problem 3(a)

**b)**

$$y[n] = 2y[n-1] - y[n-2] + x[n] + x[n-1]$$

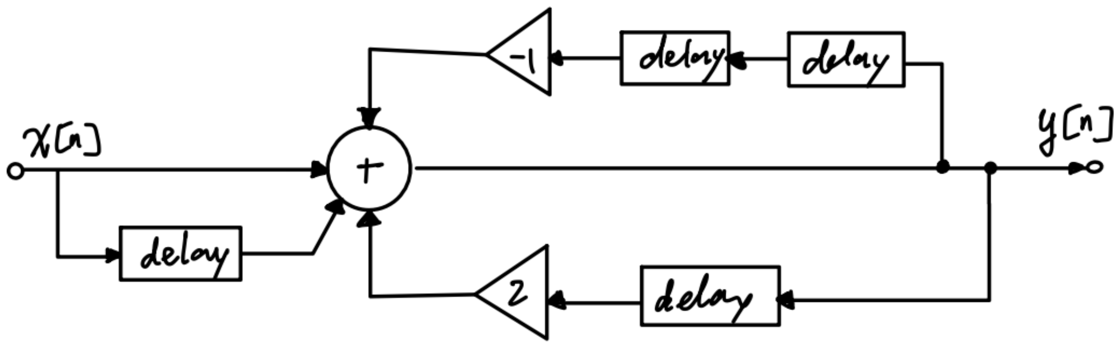


Figure 4: Figure 4: Block diagram for an input output system Problem 3(b)

## Problem 4

**a)**

$$\begin{aligned} y[n] &= 0.2y[n-1] + \beta x[n] \\ y[1] &= 0.2y[0] + \beta x[1] \\ y[0] &= 0.2y[-1] + \beta x[0] \end{aligned}$$

No, we need to know the value of $y[-1]$ to compute the value of $y[n]$ for $n = 0, 1, 2, 3, ..., N$.

**b)**

If $x[n] = 0$ for $n > N$, then $y[n] = 0.2y[n-1]$ for $n > N$.

## Problem 5

$$
\begin{aligned}
S[n+2] =& 2S[n+1] + 2S[n] \\
S[0] =& 0 \\
S[1] =& 1
\end{aligned}
$$

### a)

```r
# a)
fn_Sn <- function(N) {
  S <- numeric(N + 1)  # initialize S with enough space

  # initial conditions
  S[1] <- 0  # S[0] = 0
  S[2] <- 1  # S[1] = 1

  if (N == 0) {
    return(S[1])  # return S[0] for N = 0
  }

  for (n in 1:N) {
    S[n + 2] <- 2 * S[n + 1] + 2 * S[n]
  }

  return(S[N + 1])  # return the N-th term
}
```

### b)

$$S[n] = Aa^n + Bb^n$$

$$
\begin{aligned}
S[n+2] =& 2S[n+1] + 2S[n] \\
\Rightarrow & r^2 = 2r + 2 \\
\Rightarrow & r^2 - 2r - 2 = 0 \\
& r = \frac{2 \pm \sqrt{(-2)^2 - 4 \times (-2)}}{2} \\
& a = 1 + \sqrt{3} \\
& b = 1 - \sqrt{3}
\end{aligned}
$$

$$
\begin{aligned}
initial\, condition: & \begin{cases} S[0] = 0 \\ S[1] = 1 \end{cases} \\
& \Rightarrow \begin{cases} A + B = 0 \\ (1+\sqrt{3})A + (1-\sqrt{3})B = 1 \end{cases} \\
& \Rightarrow \begin{cases} A = \frac{\sqrt{3}}{6} \\ B = -\frac{\sqrt{3}}{6} \end{cases} \\
& \Rightarrow S[n] = \frac{\sqrt{3}}{6}(1+\sqrt{3})^n - \frac{\sqrt{3}}{6}(1-\sqrt{3})^n
\end{aligned}
$$

**c)**

```r
compute_Sn <- function(n) {
  A <- sqrt(3) / 6
  B <- - sqrt(3) / 6
  a <- 1 + sqrt(3)
  b <- 1 - sqrt(3)
  Sn <- A * a^n + B * b^n
  return(Sn)
}

test_fn_Sn <- function() {
  passed <- TRUE
  tol <- 1e-6

  for (n in 0:20) {
    expected <- compute_Sn(n)
    actual <- fn_Sn(n)

    if (abs(expected - actual) > tol) {
      cat(sprintf("Test failed for n = %d: expected %f, got %f\n", n, expected, actual))
      passed <- FALSE
    } else {
      cat(sprintf("Test passed for n = %d: expected %f, got %f\n", n, expected, actual))
    }
  }

  if (passed) {
    cat("All tests passed.\n")
  } else {
    cat("Some tests failed.\n")
  }
}

test_fn_Sn()
```

```
## Test passed for n = 0: expected 0.000000, got 0.000000
## Test passed for n = 1: expected 1.000000, got 1.000000
## Test passed for n = 2: expected 2.000000, got 2.000000
## Test passed for n = 3: expected 6.000000, got 6.000000
## Test passed for n = 4: expected 16.000000, got 16.000000
## Test passed for n = 5: expected 44.000000, got 44.000000
## Test passed for n = 6: expected 120.000000, got 120.000000
## Test passed for n = 7: expected 328.000000, got 328.000000
## Test passed for n = 8: expected 896.000000, got 896.000000
## Test passed for n = 9: expected 2448.000000, got 2448.000000
## Test passed for n = 10: expected 6688.000000, got 6688.000000
## Test passed for n = 11: expected 18272.000000, got 18272.000000
## Test passed for n = 12: expected 49920.000000, got 49920.000000
## Test passed for n = 13: expected 136384.000000, got 136384.000000
## Test passed for n = 14: expected 372608.000000, got 372608.000000
## Test passed for n = 15: expected 1017984.000000, got 1017984.000000
## Test passed for n = 16: expected 2781184.000000, got 2781184.000000
## Test passed for n = 17: expected 7598336.000000, got 7598336.000000
## Test passed for n = 18: expected 20759040.000000, got 20759040.000000
## Test passed for n = 19: expected 56714752.000000, got 56714752.000000
## Test passed for n = 20: expected 154947584.000000, got 154947584.000000
## All tests passed.
```

## Problem 6

### a)

$$x[n+1] = \frac{1}{2}(x[n] + \frac{c}{x[n]})$$

$$\begin{aligned}
&Newton's\, method: \\
x[n+1] =& x[n] - \frac{f(x[n])}{f'(x[n])} \\
set\, f(x) =& x^2 - c \\
f(x[n]) =& x[n]^2 - c,\ f'([n]) = 2x[n] \\
x[n+1] =& x[n] - \frac{x[n]^2 - c}{2x[n]} \\
=& x[n] - \frac{1}{2}x[n] + \frac{c}{2x[n]} \\
=& \frac{1}{2}(x[n] + \frac{c}{x[n]})
\end{aligned}$$

Therefore, the above nonlinear recurrence relation can be used to compute the square root of c.

### b)

```r
sqrt_est <- function(c, tol=1e-6) {
  if (c < 0) {
    stop("c must be non-negative")
  }

  # initial guess for the square root of c
  x <- ifelse(c > 0, c / 2, 0)

  while (abs(x^2 - c) >= tol) {
    x <- 0.5 * (x + c / x)
  }

  return(x)
}

# e.g.
(sqrt_est(1225))
```

```
## [1] 35
```

```r
(sqrt_est(pi))
```

```
## [1] 1.772454
```

```r
(sqrt_est(0.1004))
```

```
## [1] 0.3168596
```

### c)

```r
library(testthat)

test_that("sqrt_est correctly estimates the square root", {
  expect_equal(sqrt_est(1225), sqrt(1225), tol = 1e-6)
  expect_equal(sqrt_est(pi), sqrt(pi), tol = 1e-6)
  expect_equal(sqrt_est(0.1004), sqrt(0.1004), tol = 1e-6)
})
```

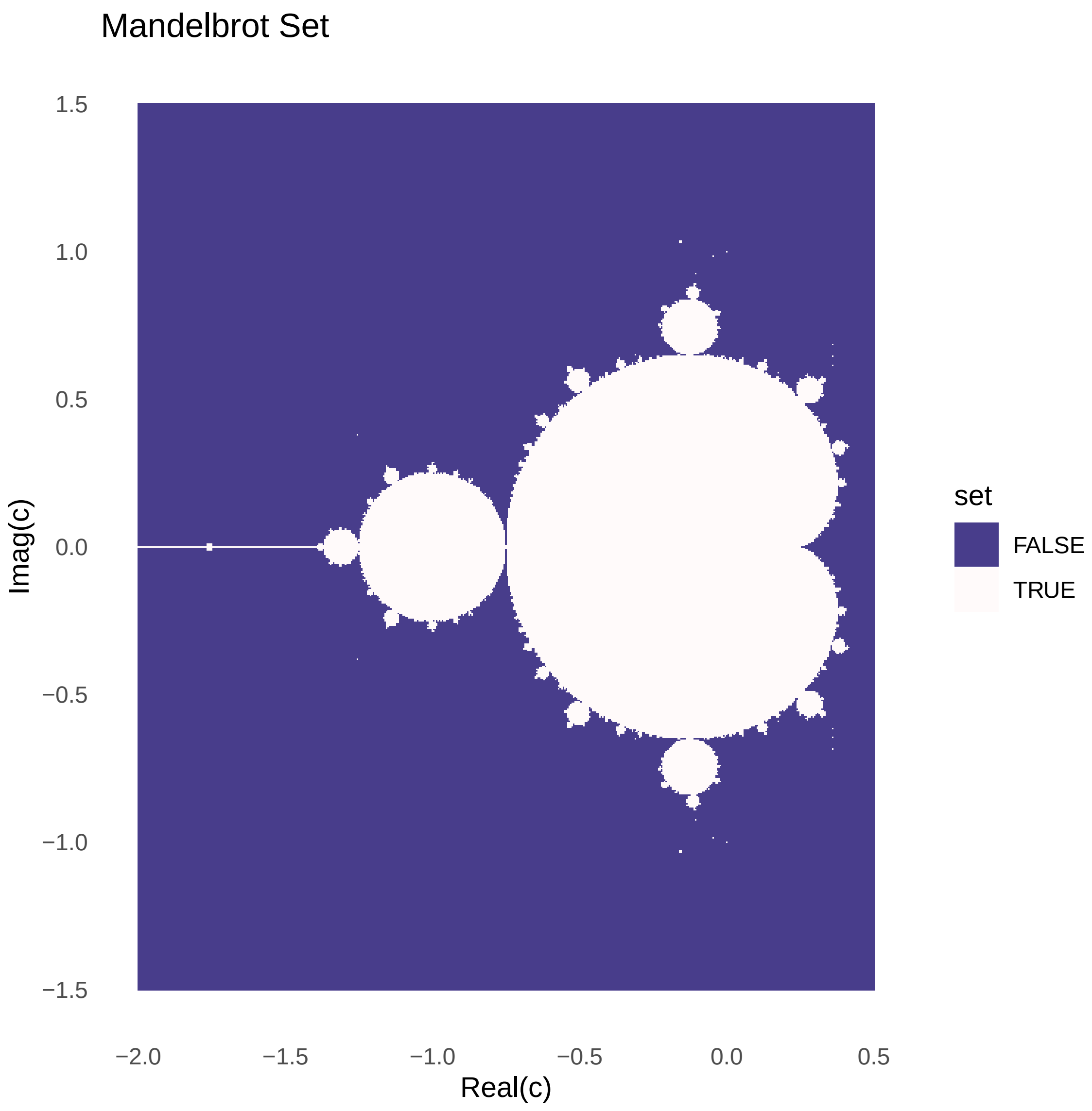
```
## Test passed
```

### d)

The values of $x_0 = x[0]$ should be positive to converge the recurrence relation to the square root of c.

## Problem 7

```r
library(ggplot2)

mandelbrot <- function(c, max_iter) {
  z <- 0
  for(i in 1:max_iter) {
    z <- z^2 + c
    if(Mod(z) > 2) {
      return(FALSE)
    }
  }
  return(TRUE)
}

x_min <- -2
x_max <- 0.5
y_min <- -1.5
y_max <- 1.5
x <- seq(x_min, x_max, by = 5e-3)
y <- seq(y_min, y_max, by = 5e-3)
grid <- expand.grid(x = x, y = y)

# check each point in the grid
max_iter <- 500
points <- vector("list", length=nrow(grid))
for(i in 1:nrow(grid)) {
  c <- complex(real = grid$x[i], imaginary = grid$y[i])
  points[[i]] <- mandelbrot(c, max_iter)
}

# convert the results to a data frame
grid$set <- unlist(points)

ggplot(grid, aes(x, y)) +
  geom_tile(aes(fill = set)) +
  scale_fill_manual(values = c("TRUE" = "snow", "FALSE" = "darkslateblue")) +
  scale_y_continuous(breaks = seq(-1.5, 1.5, by = 0.5)) +  theme_minimal() +
  labs(title = "Mandelbrot Set", x = "Real(c)", y = "Imag(c)") +
  theme(panel.grid.major = element_blank(), panel.grid.minor = element_blank())
```



## Problem 8

$$
\begin{aligned}
E_0 &= \frac{1}{2}mv^2 + mgz \\
t_0 &= 0,\ v(0) = 0 \\
E_o &= mgz_0\, at\, time\, t_0 = 0 \\
\Rightarrow \frac{dz}{dt} &= v = -\sqrt{2g(z - z_0)} \\
z(0) &= z_0 \\
z(t + \Delta t) &= z(t) + \Delta t v \\
&= z(t) - \Delta t(\sqrt{2g(z - z_0)})
\end{aligned}
$$

**a)**

$$\begin{aligned}z(\Delta t) =&z(0+\Delta t)\\ =&z(0)+\Delta t v\\ =&z_0-\Delta t(\sqrt{2g(z-z_0)})\\ z(2\Delta t) =&z(0+2\Delta t)\\ =&z(0)+2\Delta t v\\ =&z_0-2\Delta t(\sqrt{2g(z-z_0)})\end{aligned}$$

```r
g <- 9.81
z0 <- 100
delta_t <- 1e-6
v <- 0

# Will-E scheme, Euler method
z <- z0 - 1e-6
t <- 0

while (z > 0) {
  v <- sqrt(2 * g * (z0 - z))
  delta_z <- v * delta_t
  if (z - delta_z < 0) {
    z <- 0
  } else {
    z <- z - delta_z # Loop until z <= 0
  }
  t <- t + delta_t
}

real_t <- sqrt(2 * z0 / g)

# Results
cat("Time when z reaches 0 calculated by the Will-E scheme:", t, "seconds\n")
```

```
## Time when z reaches 0 calculated by the Will-E scheme: 4.51479 seconds
```

```r
cat("Time when z reaches 0 by the physical fact:", real_t, "seconds\n")
```

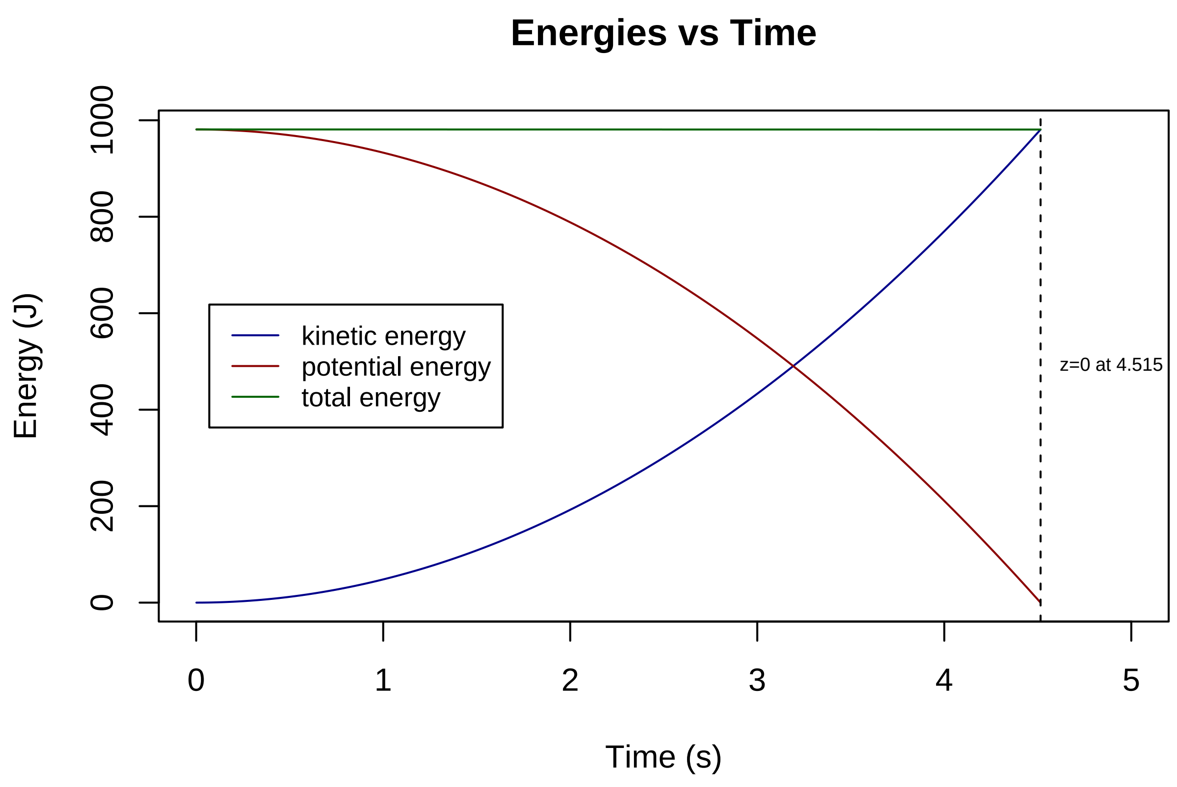
```
## Time when z reaches 0 by the physical fact: 4.515236 seconds
```

The expected time for Will-E to reach the bottom of the canyon is 4.51479 seconds. Will-E's scheme requires a very small time step ($\Delta t$), or the time computed using the Euler method is inaccurate. However, choosing a small enough time step may result in computationally inefficient results. Will-E's scheme doesn't satisfy the law of conservation of energy.

## b)

$$\begin{aligned}m\frac{d^2z}{dt^2} =&-mg\\ \frac{dz}{dt} =&v\\ \frac{dv}{dt} =&-g\end{aligned}$$

```r
g <- 9.81
z0 <- 100
m <- 1
delta_t <- 1e-3

# initial conditions
z <- z0
v <- 0
t <- 0

ts <- c()
ki_es <- c()
po_es <- c()
tt_es <- c()
posi <- c()

# Euler forward method
while (z > 0) {
  v <- v + g * delta_t
  z <- max(0, z - v * delta_t) # ensure z >= 0
  t <- t + delta_t

  ki_e = 0.5 * m * v^2
  po_e = m * g * z
  tt_e = ki_e + po_e

  ts <- append(ts, t)
  ki_es <- append(ki_es, ki_e)
  po_es <- append(po_es, po_e)
  tt_es <- append(tt_es, tt_e)
  posi <- append(posi, z)
}

plot(ts, ki_es, type = 'l', col = 'darkblue', xlim = c(0, 5),
     ylim = c(0, max(tt_es)), xlab = "Time (s)", ylab = "Energy (J)",
     main = "Energies vs Time")
lines(ts, po_es, col = 'darkred')
lines(ts, tt_es, col = 'darkgreen')
legend("left", inset=.05, legend = c("kinetic energy", "potential energy", "total energy"),
       col = c("darkblue", "darkred", "darkgreen"), lty = 1, cex = 0.8)

# z = 0
bottom_t <- ts[which.min(posi > 0)]
abline(v = bottom_t, col = "black", lty = 2)
text(bottom_t, max(tt_es) * 0.5, labels = paste("z=0 at", round(bottom_t, 6), "s"),
     pos = 4, cex = 0.6)
```



```
print(paste("Time when z=0 calculated by the Euler method:", round(bottom_t, 6), "seconds"))
```

```
## [1] "Time when z=0 calculated by the Euler method: 4.515 seconds"
```

```
cat("Time when z reaches 0 by the physical fact:", real_t, "seconds\n")
```

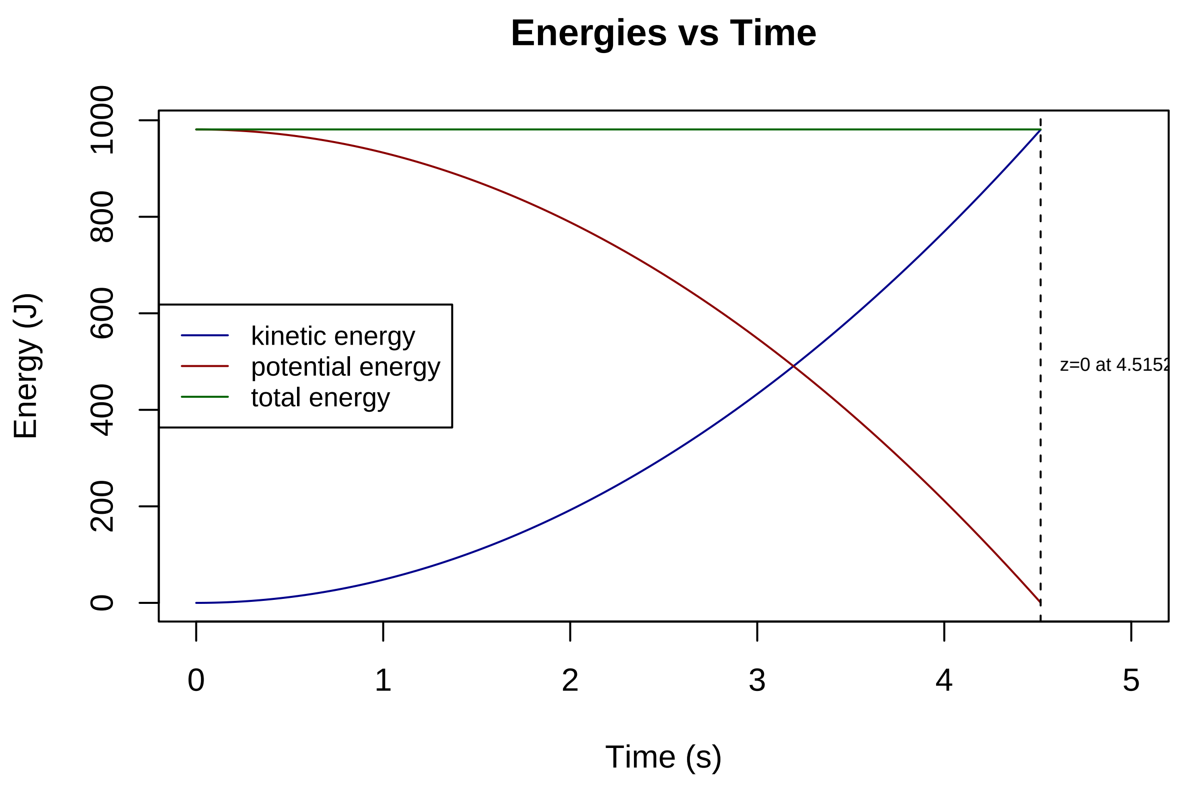
```
## Time when z reaches 0 by the physical fact: 4.515236 seconds
```

The time when z reaches 0 calculated by the Euler forward scheme is better than the result computed by the Will-E scheme since it is closer to the real-time when z=0. However, the Euler forward scheme also doesn't satisfy the energy conservation principle.

### c)

```
g <- 9.81
z0 <- 100
m <- 1
delta_t <- 1e-3
tt_time <- sqrt(2 * z0 / g)
N <- as.integer(tt_time / delta_t)

v <- rep(0, N)
z <- rep(0, N)
times <- seq(0, tt_time, length.out = N)

# initial conditions
v[1] <- 0
z[1] <- z0

ki_es <- rep(0, N)
po_es <- rep(0, N)
```

```r
tt_es <- rep(0, N)

# Leap-frog scheme for i=2
v[2] <- v[1] + g * delta_t
z[2] <- z[1] - 0.5 * g * delta_t^2

# Leap-frog iteration for i = 3 to N
for (i in 3:N) {
  v[i] <- v[i-2] + 2 * g * delta_t
  z[i] <- z[i-2] - 2 * v[i-1] * delta_t

  ki_es[i] <- 0.5 * m * v[i]^2
  po_es[i] <- m * g * (z[i] - 0)
  tt_es[i] <- ki_es[i] + po_es[i]
}

ki_es[1] <- 0.5 * m * v[1]^2
po_es[1] <- m * g * (z[1] - 0)
tt_es[1] <- ki_es[1] + po_es[1]

plot(times[-2], ki_es[-2], type = 'l', col = 'darkblue', xlim = c(0, 5),
     ylim = c(min(po_es[-2]), max(tt_es[-2])), xlab = "Time (s)",
     ylab = "Energy (J)", main = "Energies vs Time")
lines(times[-2], po_es[-2], col = 'darkred')
lines(times[-2], tt_es[-2], col = 'darkgreen')
legend("left", legend = c("kinetic energy", "potential energy", "total energy"),
       col = c("darkblue", "darkred", "darkgreen"), lty = 1, cex = 0.8)

t_bottom <- times[which.min(posi > 0)]
abline(v = t_bottom, col = "black", lty = 2)
text(t_bottom, max(tt_es) * 0.5, labels = paste("z=0 at", round(t_bottom, 6), "s"),
     pos = 4, cex = 0.6)
```



```
print(paste("Time when z=0 calculated by the leap-frog scheme:", round(t_bottom, 6), "seconds"))
```

```
## [1] "Time when z=0 calculated by the leap-frog scheme: 4.515236 seconds"
```

```
cat("Time when z reaches 0 by the physical fact:", tt_time, "seconds\n")
```

```
## Time when z reaches 0 by the physical fact: 4.515236 seconds
```

The expected time for Will-E to reach the bottom computed by the leap-frog scheme is 4.515236 seconds, the same as the time calculated by the physical fact. Therefore, the leap-frog scheme satisfies the principle of conservation of energy, and it is better than the Euler forward and Will-E scheme.