Visualization of Digital Communication System

Nguyen Minh Tuong, Vo Phi Son, Truong Tuan Vu

2024-05-16

Introduction

Although Claude Shannon's landmark paper "A Mathematical Theory of Communication [1]" has been out there for more than 50 years, the insight into how digital communication systems work is not widespread due to their complicated and counterintuitive working principles [2]. This is because of their heavy reliance on complex mathematical concepts and the continuous emergence of new theories. Consequently, communication theory can be a challenging discipline to master. Furthermore, developing an intuitive grasp of these concepts often requires significant time and effort. Moreover, the role of digital modulation techniques in the wireless communication evolution is very hard to understand if people only read the mathematical formulations.

Therefore, we will design visualizations that maximize accessibility while acknowledging and preserving the inherent mathematical complexities of the subject matter. More precisely, the working mechanisms and insights of various digital communication schemes are chosen in this project. We are interested in visualizing the working mechanism and performance of widely adopted but lesser-known communication schemes such as:

1. Binary phase shift keying
2. Quadrature phase shift keying

It is our hope that this visualization project would enhance the interest and appreciation of our viewer for these concepts. To the best of our knowledge our approach is completely novel since interactive modulation visualization using Shiny apps does not exist.

In this report, we try to answer two questions:

Question 1: How does the choice of modulation schemes affect the performance of a wireless communication system ?

Question 2: How do non-orthogonal multiple access scheme perform in terms of superposition coding, successive interference cancellation, and overall spectral efficiency ?

The Shiny apps is at this web link: https://sonvo.shinyapps.io/project2/

Justification of approach

We will collect the modulated data of these aforementioned schemes through the use of extensive Monte-Carlo simulations under different transmit power levels of transceivers, which is a practical consideration because most transceivers are battery-limited [3]. After data has been collected, we will visualize the corresponding insight regarding their effectiveness and start to make 3D printed sculptures. In case the sculptures is too difficult to manufacture we will try to initiate equivalent 3D hologram.

Importance of Electromagnetic Waves

Electromagnetic waves, including radio waves, are crucial because they propagate where sound and light cannot. For example: - **AM Radio**: With wavelengths in the hundreds of meters, AM radio waves can penetrate structures like walls without much interference. - **Visible Light**: In contrast, visible light has much shorter wavelengths and is easily reflected by objects like metal nails.

Frequency Choices in Wireless Systems

Wireless systems utilize frequencies from a few kilohertz to several hundred gigahertz. The choice of frequency depends on factors like material penetration and signal attenuation. For instance: - **Low Frequencies (AM Radio)**: Penetrate larger structures. - **High Frequencies (Wi-Fi, WiMAX)**: Used for shorter ranges and higher data capacities.

Modulation: Conveying Information Through Waves

Modulation is the process of converting information, such as voice in this case, into a form that can be transmitted over a medium like phone lines or radio waves. This is necessary because the range of sound transmission through air is limited by the power our lungs can generate. By modulating the voice signal onto another medium like wire or radio waves, we can extend the distance over which the voice information can travel effectively.

In simpler terms, modulation involves encoding information like voice into a format suitable for transmission over communication channels like wires or radio frequencies. This allows the voice data to travel much farther than it could through air alone, thanks to the modulation process that adapts the signal for the chosen transmission medium.

A single tone, like $\cos(\omega t)$, carries power but not information. Information is conveyed through changes in the signal, similar to Morse code. Modulation techniques vary a carrier's amplitude, frequency, or phase to encode data, allowing the transmission of information through electromagnetic waves.

For simplicity just think of modulation as a way to map between the information bit 10101001 to a sine wave that can be propagted quickly in the air.

Types of Modulation

- **Analog Modulation**: Varies signal properties continuously.
- **Digital Modulation**: Varies signal properties in discrete steps, though real-world transmission often makes these "digital" signals appear analog due to the inability to transmit perfect sharp edges.

We only discuss two digital modulation scheme Binary Phase Shift Keying (BPSK) and Quadrature Phase Shift Keying (QPSK).

Modulation process

The modulation process, which involves mapping an information signal (like voice or data) onto a higher-frequency sinusoidal carrier wave for transmission over a medium (like wire, air, or space).

A sinusoid, or sine wave, has three main parameters that can be varied: amplitude (height), phase (position), and frequency. Modulation techniques work by varying one or more of these parameters to represent the information being sent.

Essentially, the voice or data signal gets encoded into the sinusoidal carrier wave by altering its amplitude, phase, or frequency patterns according to the information. This modulated carrier wave is then transmitted over the desired medium.

At the receiving end, the process is reversed - the modulated carrier is demodulated to extract the original information signal from the variations in the sine wave parameters.

The medium (wire, air, etc.) simply acts as the channel through which the modulated carrier wave travels. However, noise can corrupt the signal during transmission, which is an unfortunate reality.

So in summary:

1. Information signal (voice, data) is mapped onto a sinusoidal carrier wave by modulating its parameters.
2. The modulated carrier is transmitted over a medium like wire or air.
3. At the receiver, demodulation extracts the original information from the modulated carrier.
4. Noise can disrupt or corrupt the signal during transmission over the medium.

Binary Phase Shift Keying Signal

PSK is a modulation technique employed to transmit digital data by manipulating the phase of a carrier signal. A carrier signal is a basic sinusoidal waveform at a specific frequency. In PSK, the information is encoded by altering the starting point, or phase, of the carrier wave cycles.

In PSK, distinct data bits are represented by various phase shifts of the carrier signal. For instance, a 180-degree phase shift might signify a binary 0, while a 0-degree phase shift might signify a binary 1. Other PSK variations use more than two phase shift combinations to encode multiple bits per symbol.

To summarize, PSK conveys digital information by modulating the phase of a carrier signal rather than its amplitude or frequency. By shifting the phase of the carrier wave, PSK represents digital data as distinct signal states. This technique is extensively used in various communication systems, including Wi-Fi and Bluetooth.

$$PSK\left(t\right)=\begin{cases}\sin\left(2\pi ft\right) & \text{for bit 1}\\ \sin\left(2\pi ft+\pi\right) & \text{for bit 0}\end{cases}$$

This can be illustrated by the following diagram

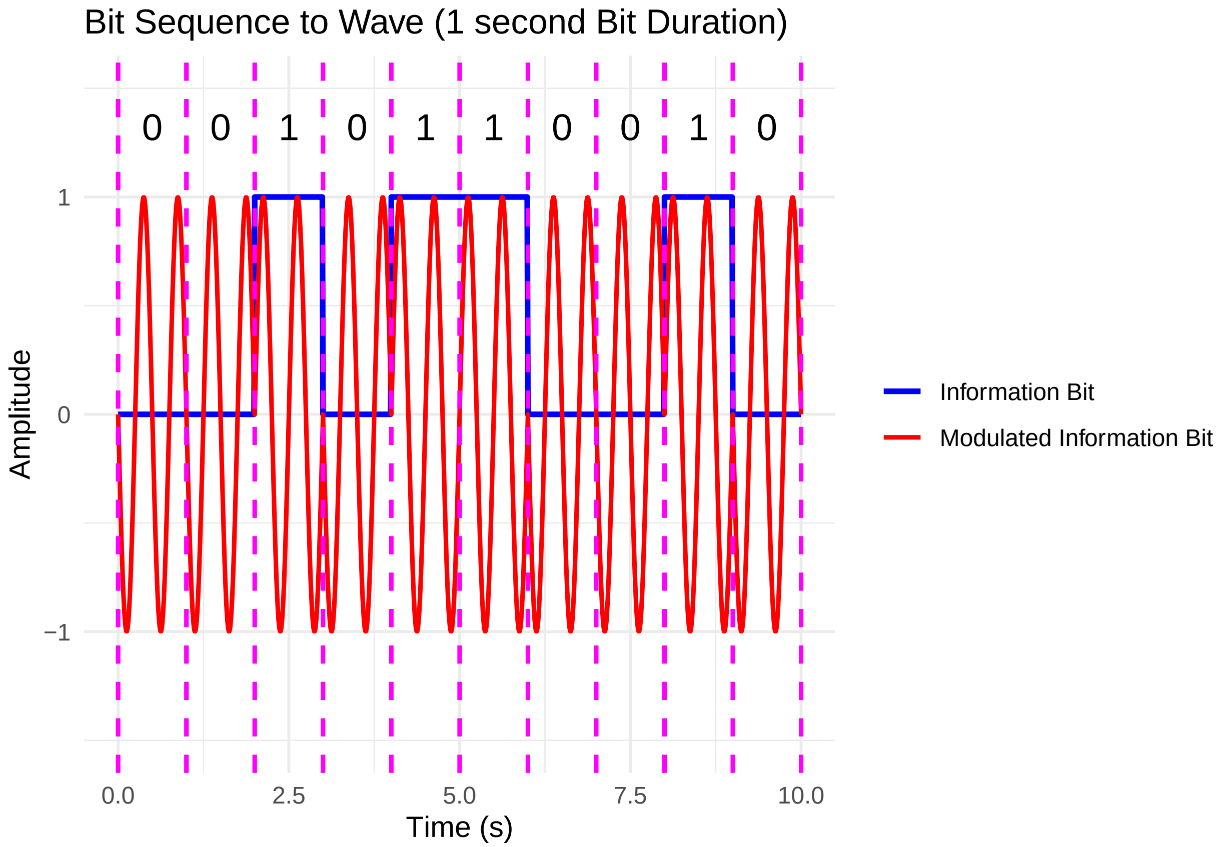
```r
# Load required library
library(ggplot2)
```

```
## Warning: package 'ggplot2' was built under R version 4.3.3
```

```r
# Function to generate square wave
generate_square_wave <- function(bit_sequence, bit_duration) {
  total_time <- nchar(bit_sequence) * bit_duration

  # Time vector with 0.01 second resolution
  t <- seq(0, total_time, by = 0.01)

  # Initialize square wave vector
  square_wave <- numeric(length(t))

  # Generate square wave
  for (i in 1:nchar(bit_sequence)) {
    bit_value <- as.numeric(substr(bit_sequence, i, i))
    start_index <- ((i - 1) * bit_duration / 0.01) + 1
    end_index <- i * bit_duration / 0.01
    square_wave[start_index:end_index] <- bit_value
  }

  return(list(t = t, square_wave = square_wave))
}

# Define parameters
f <- 2  # Frequency in Hz
# User must enter exactly 10 bit
bit_sequence <- '0010110010'
bit_duration <- 1
```

```r
# Generate square wave
wave_data <- generate_square_wave(bit_sequence, bit_duration)
t <- wave_data$t
square_wave <- wave_data$square_wave

# Generate sine waves for bit 0 and bit 1
y1 <- sin(2 * pi * f * t)
y2 <- sin(2 * pi * f * t + pi)

# Modulate square wave
modulated_stream <- ifelse(square_wave == 1, y1, y2)

# Prepare data for plotting
plot_data <- data.frame(Time = t, SquareWave = square_wave, ModulatedStream = modulated_stream)

# Calculate positions for annotations
annotations <- data.frame(
  Time = seq(bit_duration / 2, by = bit_duration, length.out = nchar(bit_sequence)),
  Label = unlist(strsplit(bit_sequence, ""))
)

# Calculate positions for vertical lines
vertical_lines <- data.frame(
  VLineTime = seq(0, max(t), by = bit_duration)
)

# Plot the square wave, modulated signal, annotations, and vertical lines
ggplot(data = plot_data, aes(x = Time)) +
  geom_line(aes(y = SquareWave, colour = "Information Bit"), size = 1.0) +
  geom_line(aes(y = ModulatedStream, colour = "Modulated Information Bit"), size = 0.8) +
  geom_text(data = annotations, aes(x = Time, y = 1.2, label = Label), vjust = -0.5, size = 5, colour =
  geom_vline(data = vertical_lines, aes(xintercept = VLineTime), linetype = "dashed", colour = "magenta
  scale_color_manual(values = c("Information Bit" = "blue", "Modulated Information Bit" = "red")) +
  labs(x = "Time (s)", y = "Amplitude") +
  ggtitle("Bit Sequence to Wave (1 second Bit Duration)") +
  theme_minimal() +
  scale_y_continuous(limits = c(-1.5, 1.5)) +
  theme(legend.title = element_blank())
```

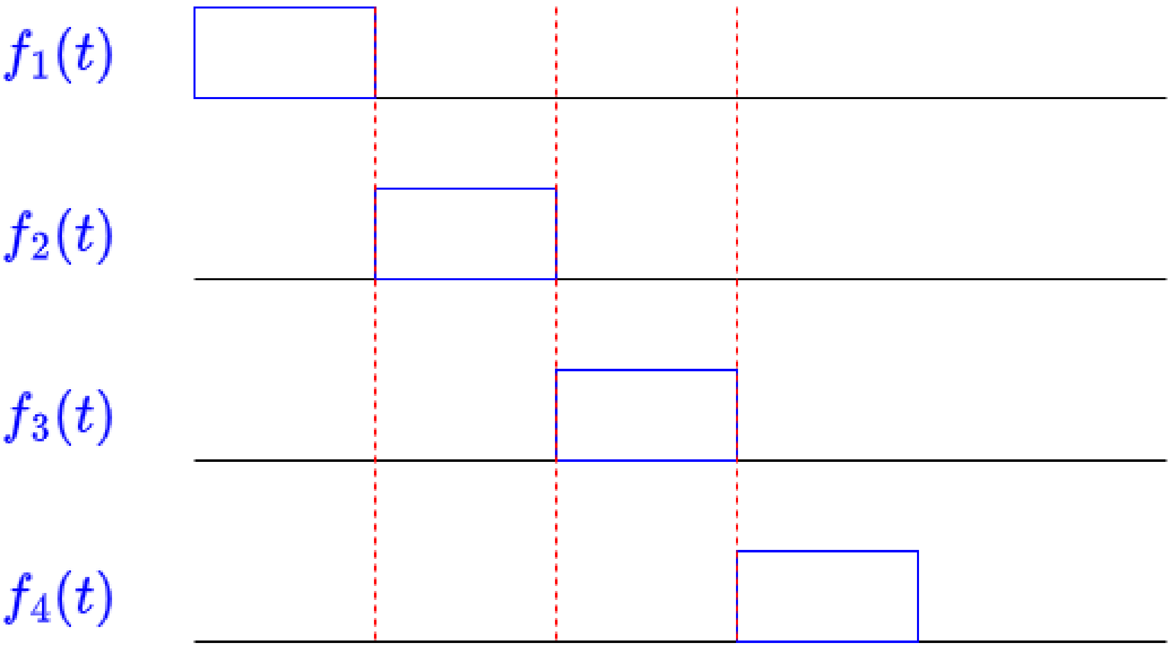
```
## Warning: Using `size` aesthetic for lines was deprecated in ggplot2 3.4.0.
## i Please use `linewidth` instead.
## This warning is displayed once every 8 hours.
## Call `lifecycle::last_lifecycle_warnings()` to see where this warning was
## generated.
```



Basis functions in signal spaces

The exploration of signal spaces offers a geometric perspective on how modulation operates in communication systems. Imagine describing a vector using coordinates (x, y), where this vector results from combining the standard basis vectors (1, 0) and (0, 1). These two functions, which are orthogonal, allow any vector in this space to be expressed as a linear combination of them—these are what we refer to as basis functions.

Similarly, consider a set of basis functions that consist of unit width pulses spaced evenly apart in time. Each pulse operates independently, and using these pulses, you can construct any sequence of data, such as a series of square pulses. While each pulse serves as a basis function, this arrangement isn't the most efficient for generating complex signals because it requires many such pulses to create diverse signal forms.



For an optimal set of basis functions, we aim to use the minimum number possible to represent a broad range of independent signals, both digital and analog. These basis functions must satisfy two main criteria:

1. **Unit Energy**: Each basis function, such as the vectors (1, 0) and (0, 1) or specific unit pulses, should have unit energy, making them efficient for signal representation. This is quite easy since we can change the amplitude of a signal to meet this requirement.
2. **Orthogonality**: Each basis function should be orthogonal to every other function within the set. Mathematically, this is expressed as:

$$\int_{-\infty}^{\infty} \phi_i(t)\phi_j(t)\,dt = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases}$$

   This property ensures that the basis functions do not interfere (or cancel) with each other, allowing for clear and distinct signal representation.

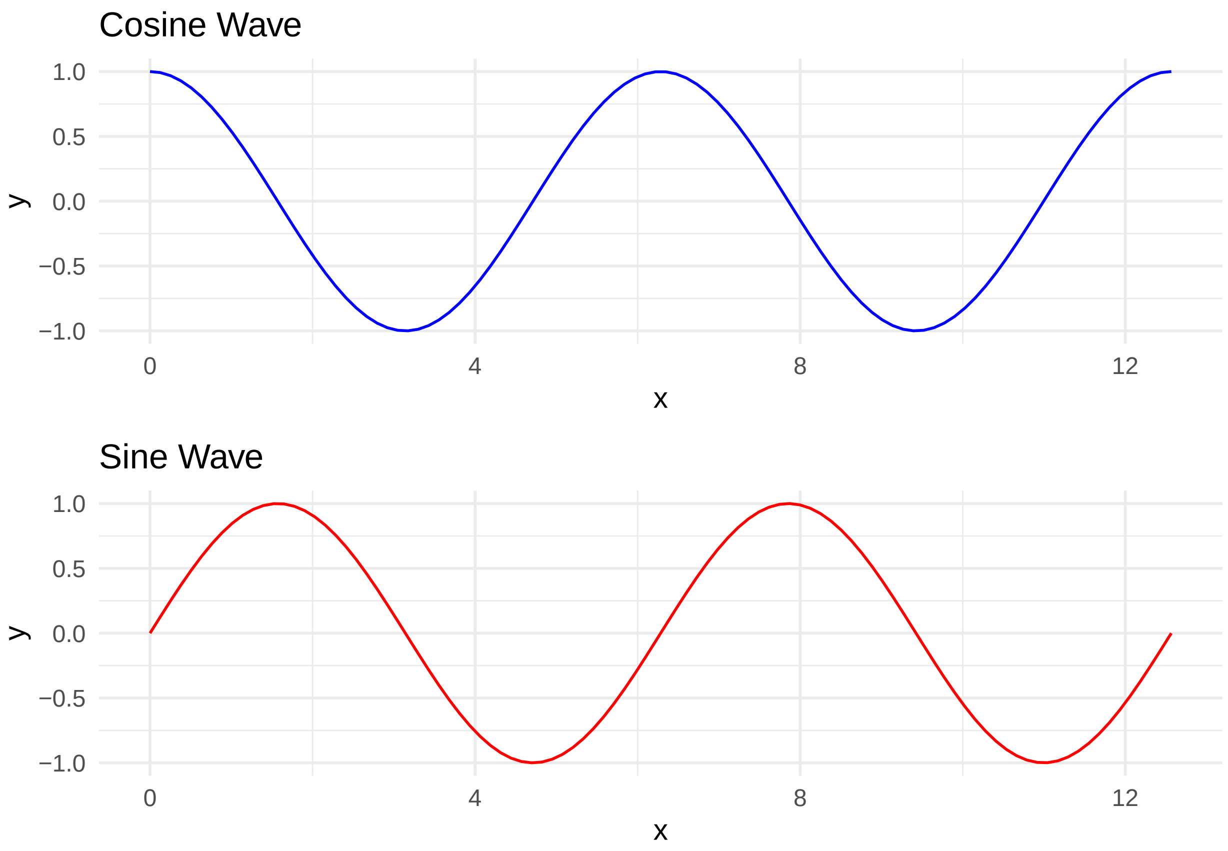
A prime example of effective basis functions widely used in real communications systems is the sine and cosine waves, both of which have unit amplitude. These functions not only meet the criteria of orthogonality and unit energy but also form the fundamental basis for modulating signals in most communication systems.

```r
# Load the necessary library
library(ggplot2)

# Define the data for cosine and sine waves over two periods
x_values <- seq(0, 4 * pi, length.out = 100)
cosine_data <- data.frame(x = x_values, y = cos(x_values))
sine_data <- data.frame(x = x_values, y = sin(x_values))

# Plot the cosine wave
p1 <- ggplot(cosine_data, aes(x, y)) +
  geom_line(color = "blue") +
  ggtitle("Cosine Wave") +
  theme_minimal()
```

```r
# Plot the sine wave
p2 <- ggplot(sine_data, aes(x, y)) +
  geom_line(color = "red") +
  ggtitle("Sine Wave") +
  theme_minimal()

# Combine the plots
library(gridExtra)
```

```
## Warning: package 'gridExtra' was built under R version 4.3.3
```

```r
grid.arrange(p1, p2, ncol = 1)
```



Cosine Wave

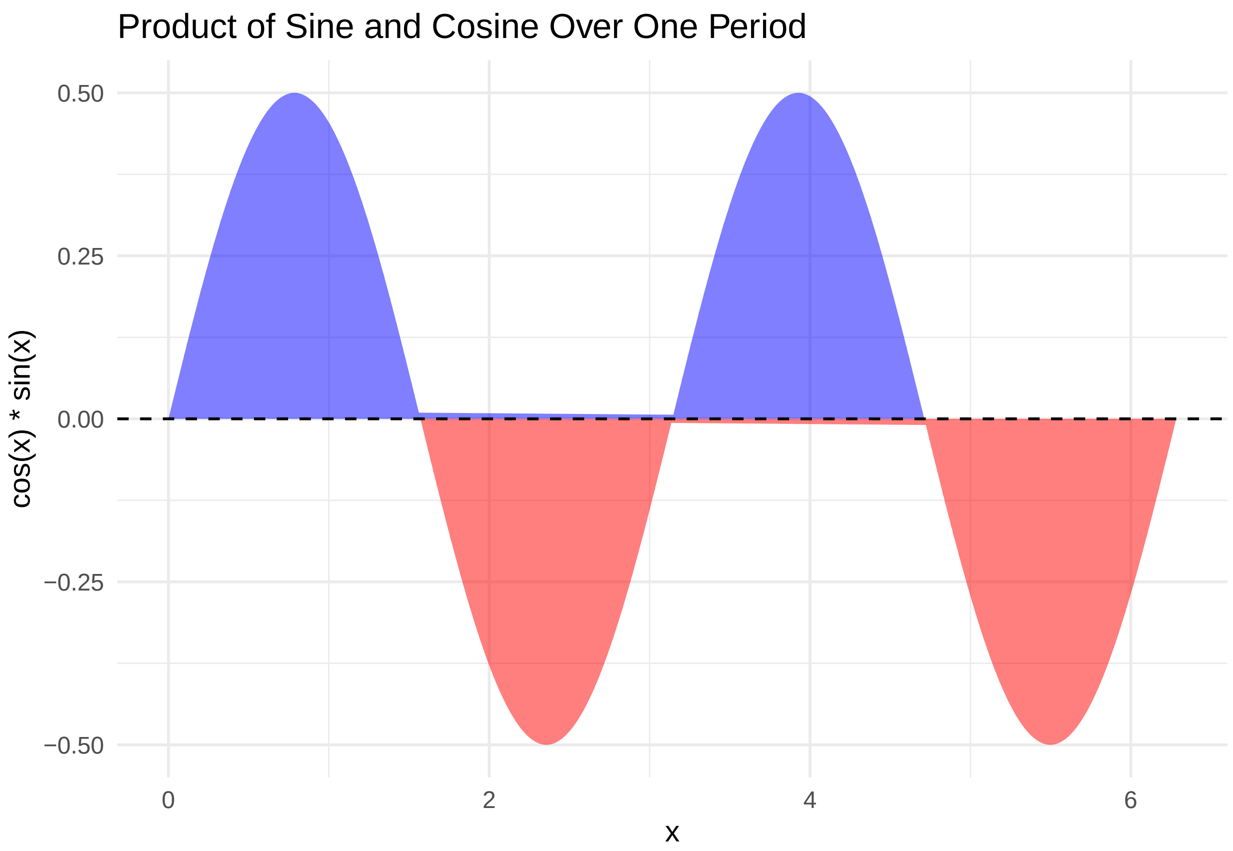
Sine Wave

Orthogonality criterion

```r
library(ggplot2)
library(dplyr)
```

```
##
## Attaching package: 'dplyr'

## The following object is masked from 'package:gridExtra':
##
##     combine

## The following objects are masked from 'package:stats':
##
##     filter, lag
## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```r
# Define the data for sine and cosine and their product over one period (0 to 2*pi)
x_values <- seq(0, 2 * pi, length.out = 500)
data <- data.frame(
  x = x_values,
  cosine = cos(x_values),
  sine = sin(x_values),
  product = cos(x_values) * sin(x_values)
)

# Add a column for fill based on the sign of the product
data$fill_color <- ifelse(data$product >= 0, 'blue', 'red')

# Plotting
p <- ggplot(data, aes(x = x)) +
  geom_area(aes(y = product, fill = fill_color), alpha = 0.5) +
  geom_hline(yintercept = 0, linetype = "dashed", color = "black") +
  labs(title = "Product of Sine and Cosine Over One Period",
       x = "x",
       y = "cos(x) * sin(x)") +
  theme_minimal() +
  scale_fill_identity()  # Use the colors specified in the data frame

print(p)
```



In-phase channel and quadrature channel

Consider a signal as a vector. There are two primary methods to represent this vector: rectangular and polar forms.

1. **Rectangular Form**: In this representation, we use two perpendicular components termed the In-phase (I) and Quadrature (Q) components. Imagine these as x and y coordinates on a Cartesian plane. The component along the x-axis is called the In-phase component, denoted as s_{11}, and the y-axis component is the Quadrature component, denoted as s_{12}.
2. **Polar Form**: The same signal can also be described by its magnitude and angle. The magnitude of the signal vector is calculated as $S = \sqrt{I^2 + Q^2}$, where I and Q are the In-phase and Quadrature components respectively. The angle θ of the vector, representing the phase of the signal, is determined by $\theta = \tan^{-1}\left(\frac{Q}{I}\right)$.

These methods provide a complete description of the signal. While rectangular form focuses on the coordinate-based projection, polar form emphasizes the signal vector's direction and length.

Bits versus symbols

In digital communication, the terms 'bit' and 'symbol' represent different concepts, although both can be depicted using wave functions like sinusoids. A 'bit' is the basic unit of digital information. In contrast, a 'symbol' represents a specific arrangement or pattern of bits and is the unit of data that the transmission medium actually conveys.

Think of bits as small items or 'widgets,' and symbols as the 'boxes' that carry these widgets during transport. Just as you can pack one widget or multiple widgets into a box, a single symbol can represent one bit or multiple bits, depending on the modulation scheme used. This packing of multiple bits into a single symbol is essentially what modulation involves.

In the context of analog communications, a symbol corresponds to a particular waveform shape, determined by a set convention, that encodes a set number of bits.

To clarify further, the term ‘baud rate’ refers to the number of symbols transmitted per second. For example, if a communication system transmits at 200 bauds, it means it sends 200 symbols per second, regardless of how many bits each symbol represents.

The frequency of these transmissions, such as a 1 Hz signal, doesn’t convey the actual rate of data transmission but serves as an illustrative measure of how often symbols are sent.

BPSK analogy

Imagine a ship at sea that needs to send a distress signal to an airplane overhead using a simple light-based communication system. The captain uses a bright light and two designated spots on either side of the ship’s mast. When the light is placed on the right, it represents the binary ‘1’, and when it’s on the left, it represents ‘0’. This setup is based on the premise that observers (airplanes) understand the binary meaning of the light positions.

This method of signaling uses a one-dimensional approach, where the captain only moves the light horizontally (left to right) to change the symbol or bit being communicated.

The act of moving the light between these two positions can be compared to a type of digital modulation known as Binary Phase Shift Keying (BPSK). In BPSK, information (bits) is encoded in the phase of a carrier wave, not in the amplitude or frequency. Specifically, two phases of the carrier wave are used: one with zero degrees (representing one symbol, say ‘s1’) and another with 180 degrees (representing the other symbol, say ‘s2’). This shift in phase directly corresponds to the light moving from one side to the other.

In BPSK, typically, there’s no vertical component (Y-axis) in the signal—it’s all based on the X-axis changes. The vector representing the signal flips along the X-axis based on the bit value, with no change in the Y-axis because the sine of both 0° and 180° is zero, resulting in no Y-component.

BPSK transmission revisited

Let us examine how to send the bit sequence `0010110010` using BPSK, where each bit corresponds to a specific phase of a carrier wave. In BPSK, we use two phases: one for bit ‘0’ and another for bit ‘1’. Specifically, let’s denote a 0 degree phase shift as symbol s1 for bit ‘0’ and a 180 degree phase shift as symbol s2 for bit ‘1’.

Following this scheme, the bit sequence `0010110010` translates into the symbol sequence:

- `0` becomes s1,
- `0` becomes s1,
- `1` becomes s2,
- `0` becomes s1,
- `1` becomes s2,
- `1` becomes s2,
- `0` becomes s1,
- `0` becomes s1,
- `1` becomes s2,
- `0` becomes s1.

This results in the symbol sequence: s1 s1 s2 s1 s2 s2 s1 s1 s2 s1.

When this sequence is modulated onto a carrier wave, each transition from s1 to s2 or from s2 to s1 represents a 180-degree phase shift in the wave. If visualized, each s1 symbol would be seen as a phase of 0 degrees, and each s2 as a phase of 180 degrees, with the shifts occurring at each bit change.

```r
# Load required library
library(ggplot2)
```

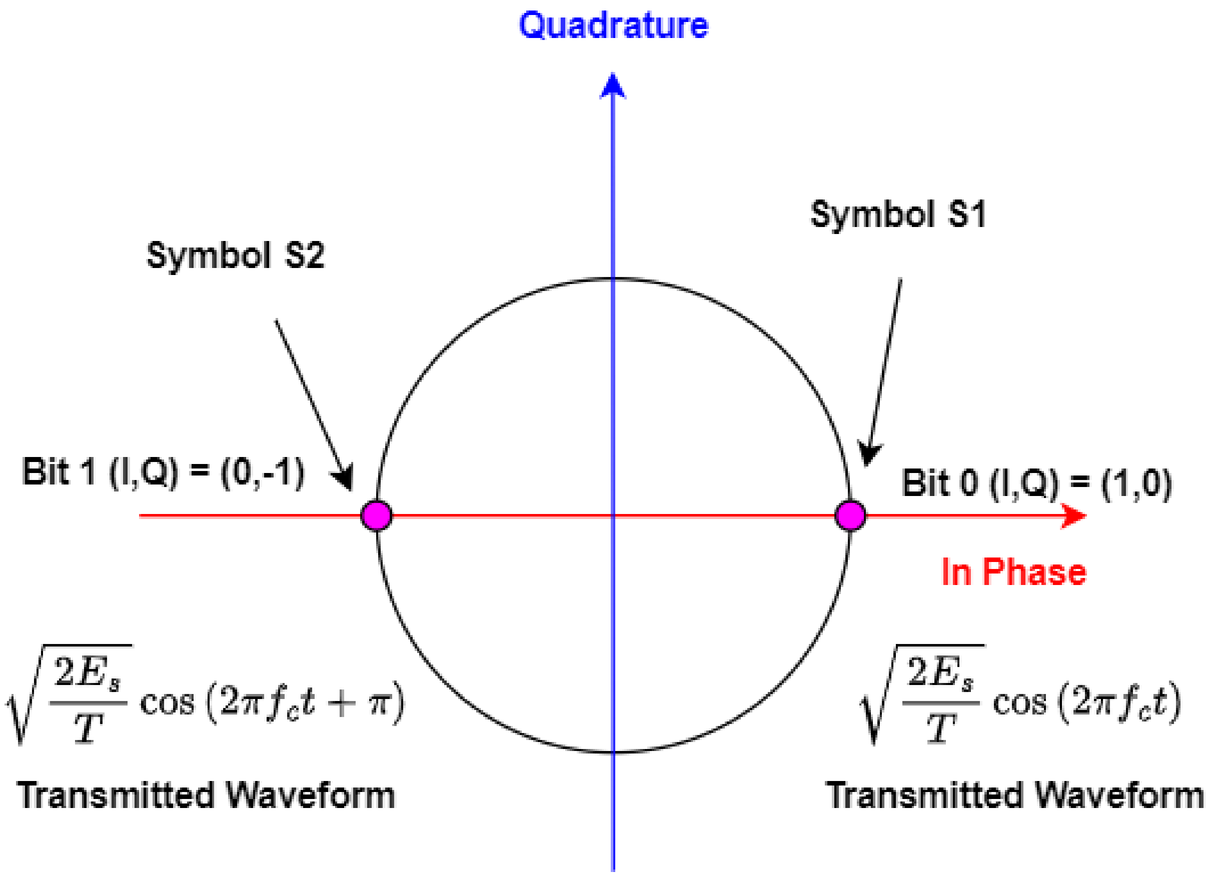
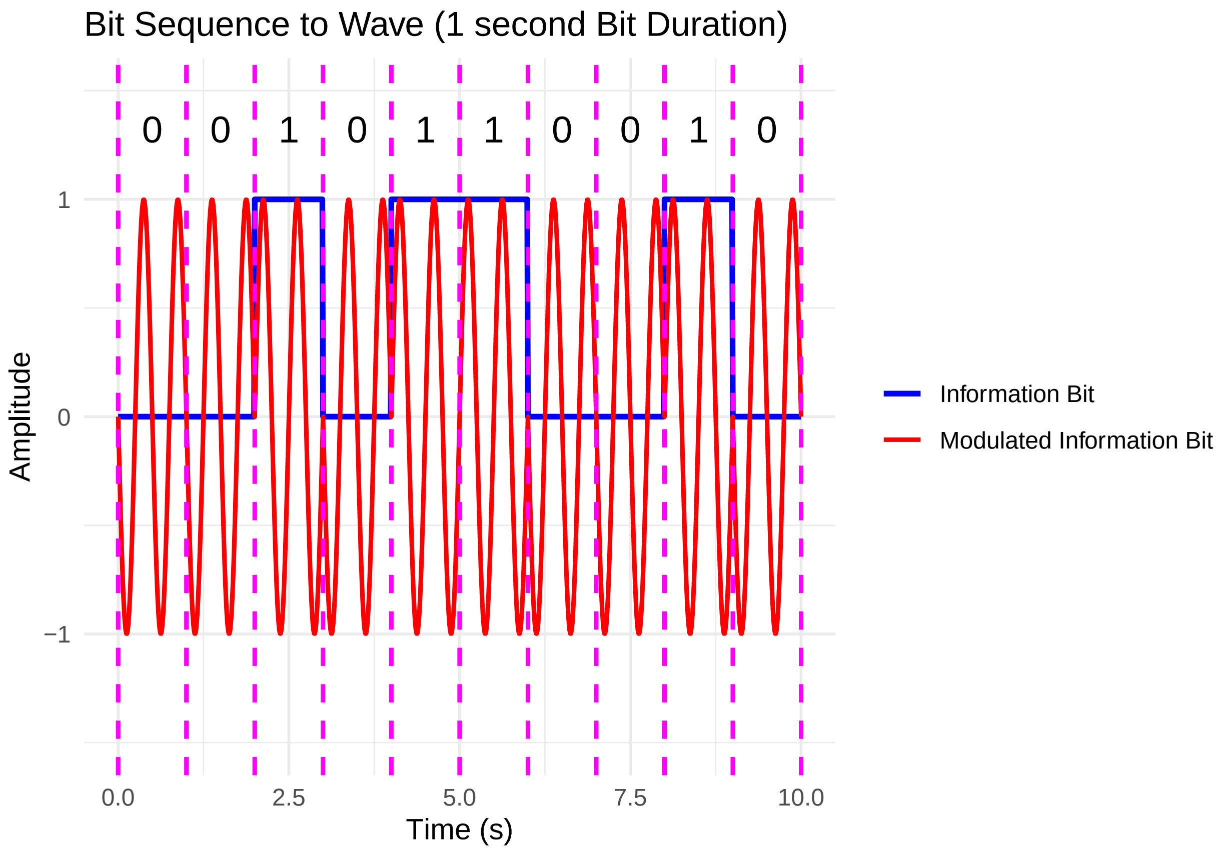


Figure 1: “Constellation diagram”

```
# Function to generate square wave
generate_square_wave <- function(bit_sequence, bit_duration) {
  total_time <- nchar(bit_sequence) * bit_duration

  # Time vector with 0.01 second resolution
  t <- seq(0, total_time, by = 0.01)

  # Initialize square wave vector
  square_wave <- numeric(length(t))

  # Generate square wave
  for (i in 1:nchar(bit_sequence)) {
    bit_value <- as.numeric(substr(bit_sequence, i, i))
    start_index <- ((i - 1) * bit_duration / 0.01) + 1
    end_index <- i * bit_duration / 0.01
    square_wave[start_index:end_index] <- bit_value
  }

  return(list(t = t, square_wave = square_wave))
}

# Define parameters
f <- 2  # Frequency in Hz
# User must enter exactly 10 bit
bit_sequence <- '0010110010'
bit_duration <- 1

# Generate square wave
wave_data <- generate_square_wave(bit_sequence, bit_duration)
t <- wave_data$t
square_wave <- wave_data$square_wave

# Generate sine waves for bit 0 and bit 1
y1 <- sin(2 * pi * f * t)
y2 <- sin(2 * pi * f * t + pi)

# Modulate square wave
modulated_stream <- ifelse(square_wave == 1, y1, y2)

# Prepare data for plotting
plot_data <- data.frame(Time = t, SquareWave = square_wave, ModulatedStream = modulated_stream)

# Calculate positions for annotations
annotations <- data.frame(
  Time = seq(bit_duration / 2, by = bit_duration, length.out = nchar(bit_sequence)),
  Label = unlist(strsplit(bit_sequence, ""))
)

# Calculate positions for vertical lines
vertical_lines <- data.frame(
  VLineTime = seq(0, max(t), by = bit_duration)
)
```

```
# Plot the square wave, modulated signal, annotations, and vertical lines
ggplot(data = plot_data, aes(x = Time)) +
  geom_line(aes(y = SquareWave, colour = "Information Bit"), size = 1.0) +
  geom_line(aes(y = ModulatedStream, colour = "Modulated Information Bit"), size = 0.8) +
  geom_text(data = annotations, aes(x = Time, y = 1.2, label = Label), vjust = -0.5, size = 5, colour =
  geom_vline(data = vertical_lines, aes(xintercept = VLineTime), linetype = "dashed", colour = "magenta
  scale_color_manual(values = c("Information Bit" = "blue", "Modulated Information Bit" = "red")) +
  labs(x = "Time (s)", y = "Amplitude") +
  ggtitle("Bit Sequence to Wave (1 second Bit Duration)") +
  theme_minimal() +
  scale_y_continuous(limits = c(-1.5, 1.5)) +
  theme(legend.title = element_blank())
```



In practical terms, if you could view this signal on a network analyzer with a carrier frequency significantly higher than 1 Hz, you would notice the signal switching back and forth between these two phases. However, because the carrier frequency is high, the wave contains many cycles between each phase shift, making the transitions appear very rapid and frequent.

The term ‘transition’ in this context refers to these moments when the signal flips its phase. It’s crucial in BPSK since the integrity of data transmission relies on accurately detecting these phase changes despite potential distortions caused by amplifier non-linearities. Amplifiers can struggle with these rapid changes, which may introduce errors if not properly managed.

Quadrature Phase Shift Keying

“Imagine a ship with a captain who has devised a sophisticated signaling method using four designated spots on the deck: East, West, North, and South. Each direction corresponds to a unique combination of two bits, thereby allowing the transmission of two bits with each signal flash. This setup, if operated in the same timeframe as a simpler single-bit system, would double the communication speed.

This technique introduces an additional dimension to the signaling because it utilizes two basis movements (East-West and North-South) to define four unique symbols (00, 01, 10, 11). This method is known as Quadrature Phase Shift Keying (QPSK), which is a two-dimensional modulation technique. In contrast to Binary Phase Shift Keying (BPSK) that transmits one bit per symbol along one dimension, QPSK sends two bits per symbol by incorporating two dimensions (sine and cosine waves).

Mathematical Representation: QPSK extends BPSK by employing M-ary signaling, where 'M' indicates the number of possible symbol states. For QPSK, M equals 4, implying four possible phase shifts for the carrier wave. The expression for a QPSK signal can be described inline as:

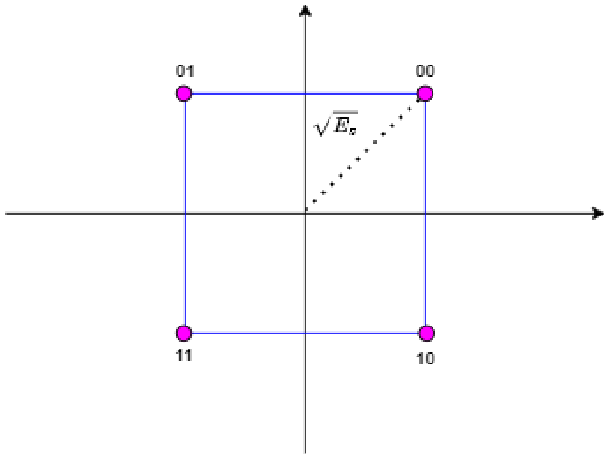
$s_i(t) = Ap(t)\cos(2\pi f_c t + \frac{2\pi i}{M})$

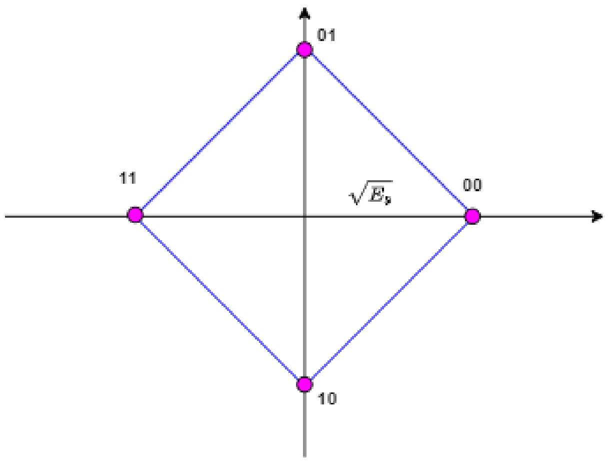
Here, $p(t)$ is a pulse shaping function, f_c is the carrier frequency, and i varies from 1 to M, denoting different phase shifts.

In digital phase modulation like QPSK, changes in the phase of the carrier wave encode different data bits. For QPSK, with M=4, each modulation angle θ_i is calculated as:

$\theta_i = \frac{2\pi i}{M}$

These phase shifts enable the encoding of two bits per symbol, enhancing the data transmission efficiency compared to BPSK. This modulation is part of what's known as rotation invariant, where the system's overall power and performance remain constant, irrespective of the constellation diagram's rotation."

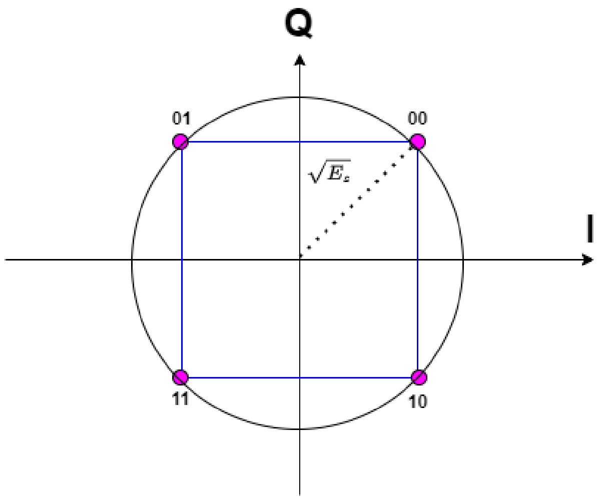


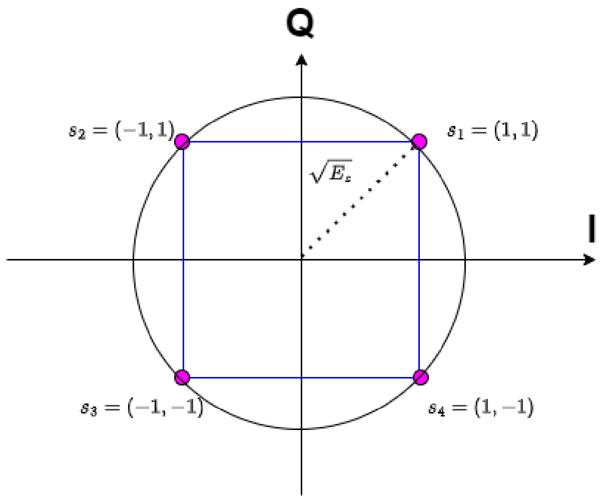


As such, the general equation in the case of arbitrary M-PSK Modulation is $s_i\,(t) = \sqrt{\frac{2E_s}{T}}\cos\left(2\pi f_c t + \frac{2\pi i}{M}\right)$. For example with QPSK then $i = 0, 1, 2, 3$ and $M = 4$

Symbol	**Bit Group**	$s(t)$
s_1	00	$\sqrt{\frac{2E_2}{T}}\cos\left(2\pi f_c t + \frac{\pi}{4}\right)$
s_2	01	$\sqrt{\frac{2E_2}{T}}\cos\left(2\pi f_c t + \frac{3\pi}{4}\right)$
s_3	10	$\sqrt{\frac{2E_2}{T}}\cos\left(2\pi f_c t + \frac{5\pi}{4}\right)$
s_4	11	$\sqrt{\frac{2E_2}{T}}\cos\left(2\pi f_c t + \frac{7\pi}{4}\right)$

From the table above, the corresponding visualization is

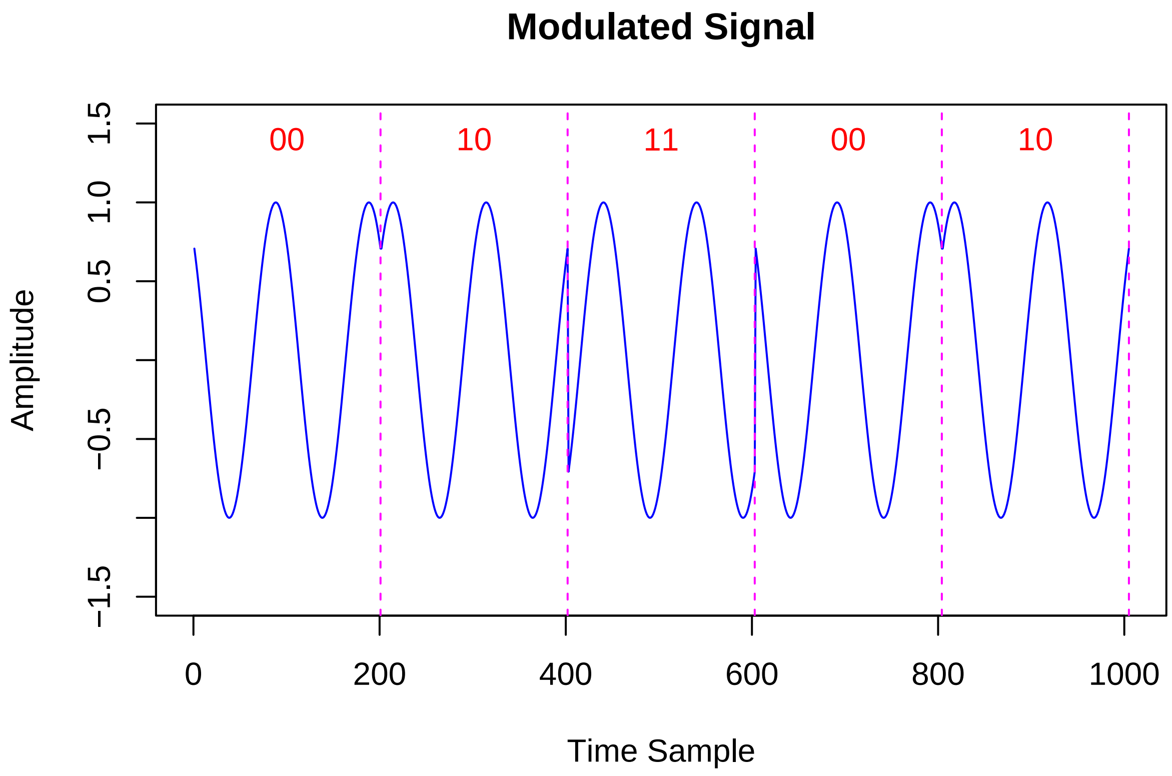




```
# Clear workspace and console
#rm(list = ls())
#graphics.off()
split_bit_sequenceQPSK <- function(bit_sequence) {
  substring_length <- 2
  num_substrings <- floor(nchar(bit_sequence) / substring_length)
  bit_substrings <- vector("list", num_substrings)
  for (i in seq_len(num_substrings)) {
    start_index <- (i - 1) * substring_length + 1
    bit_substrings[[i]] <- substring(bit_sequence, start_index, start_index + substring_length - 1)
  }
  return(bit_substrings)
}

generate_square_waveQPSK <- function(bit_sequence, bit_duration) {
  if (typeof(bit_sequence) != "character") {
    stop("Input bit_sequence must be a string.")
  }
  if (bit_duration <= 0) {
    stop("Bit duration must be a positive value.")
  }

  total_time <- nchar(bit_sequence) * bit_duration
  t <- seq(0, total_time, by = 0.01)

  square_wave <- numeric(length(t))
  for (i in seq_along(bit_sequence)) {
    bit_value <- as.numeric(substr(bit_sequence, i, i))
    start_index <- (i - 1) * (bit_duration / 0.01) + 1
    end_index <- i * (bit_duration / 0.01)
    square_wave[start_index:end_index] <- bit_value
  }
  return(list(wave = square_wave, time = t))
}

# Constants and input
fc <- 1
```

```r
# User must enter exactly 10 bit
bit_sequence <- "0010110010"
bit_duration <- 1
#Length of substring is 2 bit
substrings <- split_bit_sequenceQPSK(bit_sequence)

# Generate square wave
square_wave_data <- generate_square_waveQPSK(bit_sequence, bit_duration)
square_wave <- square_wave_data$wave
t <- square_wave_data$time

temp <- numeric(0)
symbol_duration <- 2 * bit_duration
t_symbol <- seq(0, symbol_duration, by = 0.01)

# Generate modulated signals
for (substring in substrings) {
  if (substring == "00") {
    s1 <- cos(2 * pi * fc * t_symbol + pi / 4)
    temp <- c(temp, s1)
  } else if (substring == "01") {
    s2 <- cos(2 * pi * fc * t_symbol + 3 * pi / 4)
    temp <- c(temp, s2)
  } else if (substring == "11") {
    s3 <- cos(2 * pi * fc * t_symbol + 5 * pi / 4)
    temp <- c(temp, s3)
  } else if (substring == "10") {
    s4 <- cos(2 * pi * fc * t_symbol + 7 * pi / 4)
    temp <- c(temp, s4)
  }
}

# Display substrings
#print(substrings)







# Plotting
plot(seq_along(temp), temp, type = "l", col = "blue", xlab = "Time Sample", ylab = "Amplitude", xlim = c

# Annotating each substring and adding dashed vertical lines
time_increment <- length(t_symbol)  # Each symbol is repeated for the length of t_symbol
for (i in seq_along(substrings)) {
  # Calculate the midpoint of the time interval for this substring
  midpoint <- (i - 0.5) * time_increment
  # Annotate the substring on the plot
  text(midpoint, 1.4, substrings[[i]], col = "red")

  # Draw dashed vertical lines at the end of each time increment
  if (i < length(substrings)) {  # No line at the end of the last segment to avoid duplication
```

```
    abline(v = i * time_increment, lty = "dashed", col = "magenta")
  }
}

# Optionally, add a final line if needed at the end of the last segment
abline(v = length(substrings) * time_increment, lty = "dashed", col = "magenta")
```



Bit error rate

In radio communication, data is transmitted over airwaves, which are prone to random noise, primarily from the hardware and the environment. This noise usually follows a Gaussian distribution and causes random errors in the received data. Occasionally, many errors might occur all at once, especially due to problems like signal fading or movement (Doppler effects).

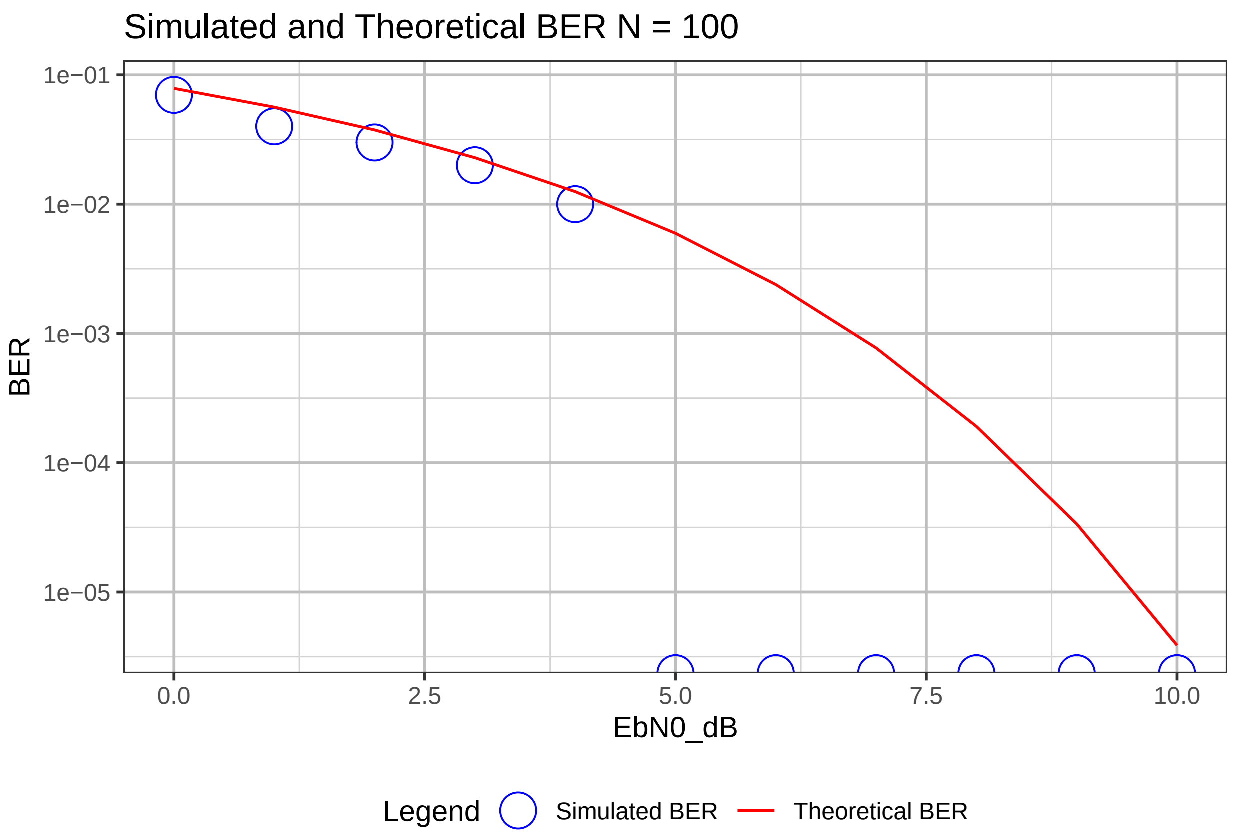
However, errors are technically not a part of the signal until it's interpreted by a receiver. In digital communication, the original signal is analog, and 'bit errors' only exist once the digital receiver tries to interpret (decode) the signal. The type of modulation (how data is encoded into the signal) greatly influences how likely errors are because it determines how closely packed data symbols are in the signal. The closer they are, the harder it is for the receiver to distinguish them without errors. This sensitivity of the receiver in making correct decisions is crucial in determining the quality of the communication link. If we sent `0000` and receive `1000` this implies that our bit error rate is 25.

For BPSK, the theoretical BER is $Q\left(\sqrt{\frac{2E_b}{N_0}}\right)$ where $Q\left(x\right) = \frac{1}{\sqrt{2\pi}} \int\limits_{u=x}^{u=\infty} \exp\left(-\frac{u^2}{2}\right) du$ and $\frac{E_b}{N_0}$ is the energy per bit to noise power spectral density ratio. Interestingly, this formula can be validated through the use of Monte-Carlo simulation

```r
# Load the required package
library(ggplot2)

# Read the CSV file
data <- read.csv("BER_data_N100.csv")

# Create the plot
ggplot(data, aes(x = EbN0_dB)) +
  geom_point(aes(y = Simulated_BER, color = "Simulated BER"), shape = 1,size = 6) +  # Set color inside
  geom_line(aes(y = Theoretical_BER, color = "Theoretical BER")) +  # Set color inside aes() for legend
  scale_y_log10() +
  xlab("EbN0_dB") +
  ylab("BER") +
  ggtitle("Simulated and Theoretical BER N = 100") +
  theme_bw() +  # Base theme with grid
  theme(panel.grid.major = element_line(color = "grey", size = 0.5),  # Customize major grid lines
        panel.grid.minor = element_line(color = "lightgrey", size = 0.25),  # Customize minor grid line
        legend.position = "bottom") +  # Adjust legend position
  scale_color_manual(name = "Legend", values = c("Simulated BER" = "blue", "Theoretical BER" = "red")) +
  guides(color = guide_legend(override.aes = list(shape = c(1, NA))))  # Ensure correct shapes in legen
```

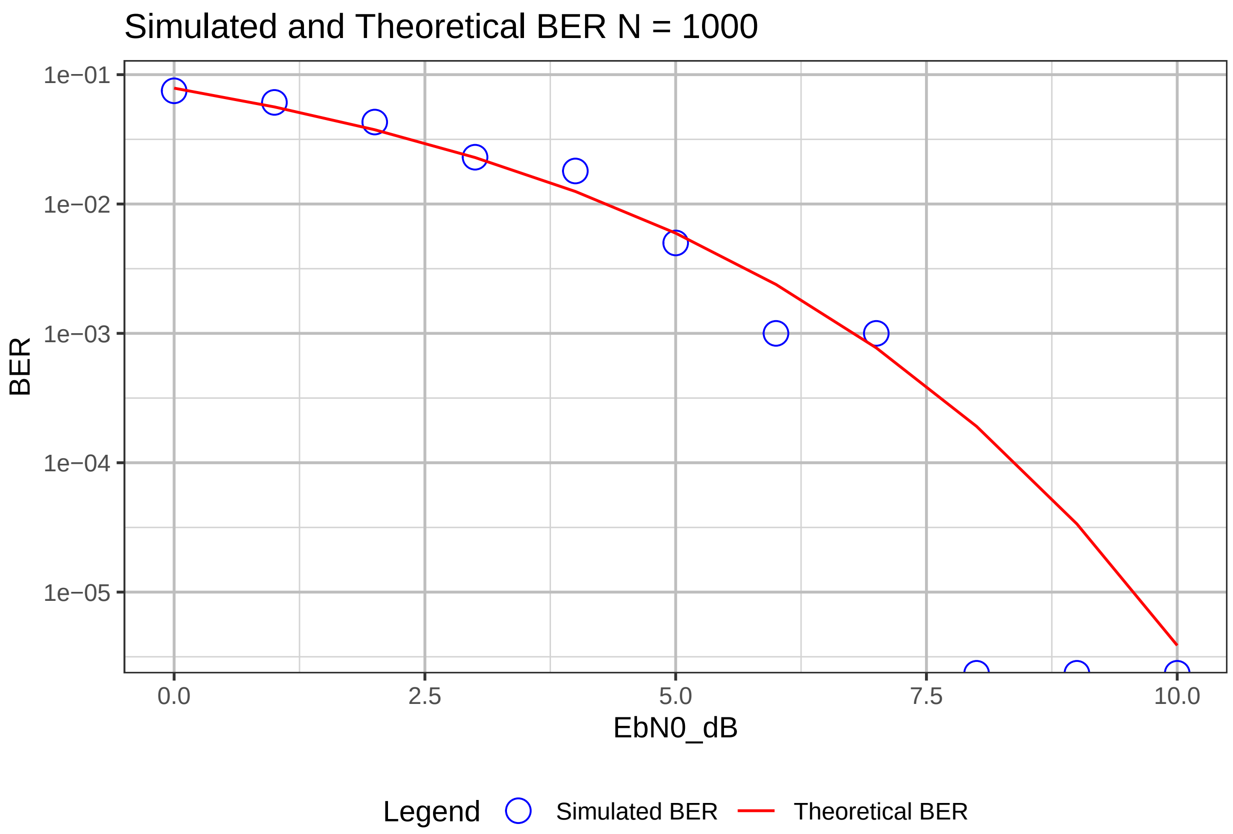
```
## Warning: The `size` argument of `element_line()` is deprecated as of ggplot2 3.4.0.
## i Please use the `linewidth` argument instead.
## This warning is displayed once every 8 hours.
## Call `lifecycle::last_lifecycle_warnings()` to see where this warning was
## generated.
```

```
## Warning in scale_y_log10(): log-10 transformation introduced infinite values.
```



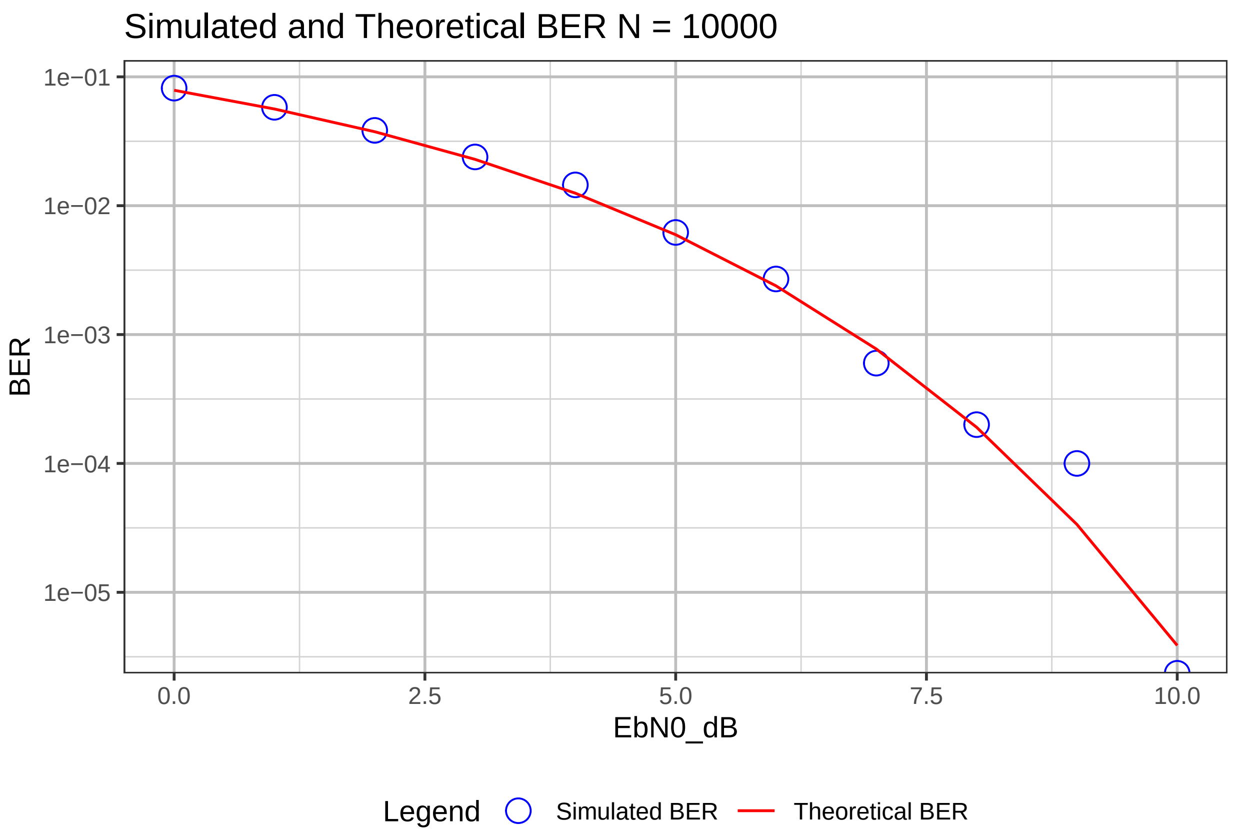
```
# Load the required package
library(ggplot2)

# Read the CSV file
data <- read.csv("BER_data_N1000.csv")

# Create the plot
ggplot(data, aes(x = EbN0_dB)) +
  geom_point(aes(y = Simulated_BER, color = "Simulated BER"), shape = 1,size = 4) +  # Set color inside
  geom_line(aes(y = Theoretical_BER, color = "Theoretical BER")) +  # Set color inside aes() for legend
  scale_y_log10() +
  xlab("EbN0_dB") +
  ylab("BER") +
  ggtitle("Simulated and Theoretical BER N = 1000") +
  theme_bw() +  # Base theme with grid
  theme(panel.grid.major = element_line(color = "grey", size = 0.5),  # Customize major grid lines
        panel.grid.minor = element_line(color = "lightgrey", size = 0.25),  # Customize minor grid line
        legend.position = "bottom") +  # Adjust legend position
  scale_color_manual(name = "Legend", values = c("Simulated BER" = "blue", "Theoretical BER" = "red")) +
  guides(color = guide_legend(override.aes = list(shape = c(1, NA))))  # Ensure correct shapes in legen
```

```
## Warning in scale_y_log10(): log-10 transformation introduced infinite values.
```



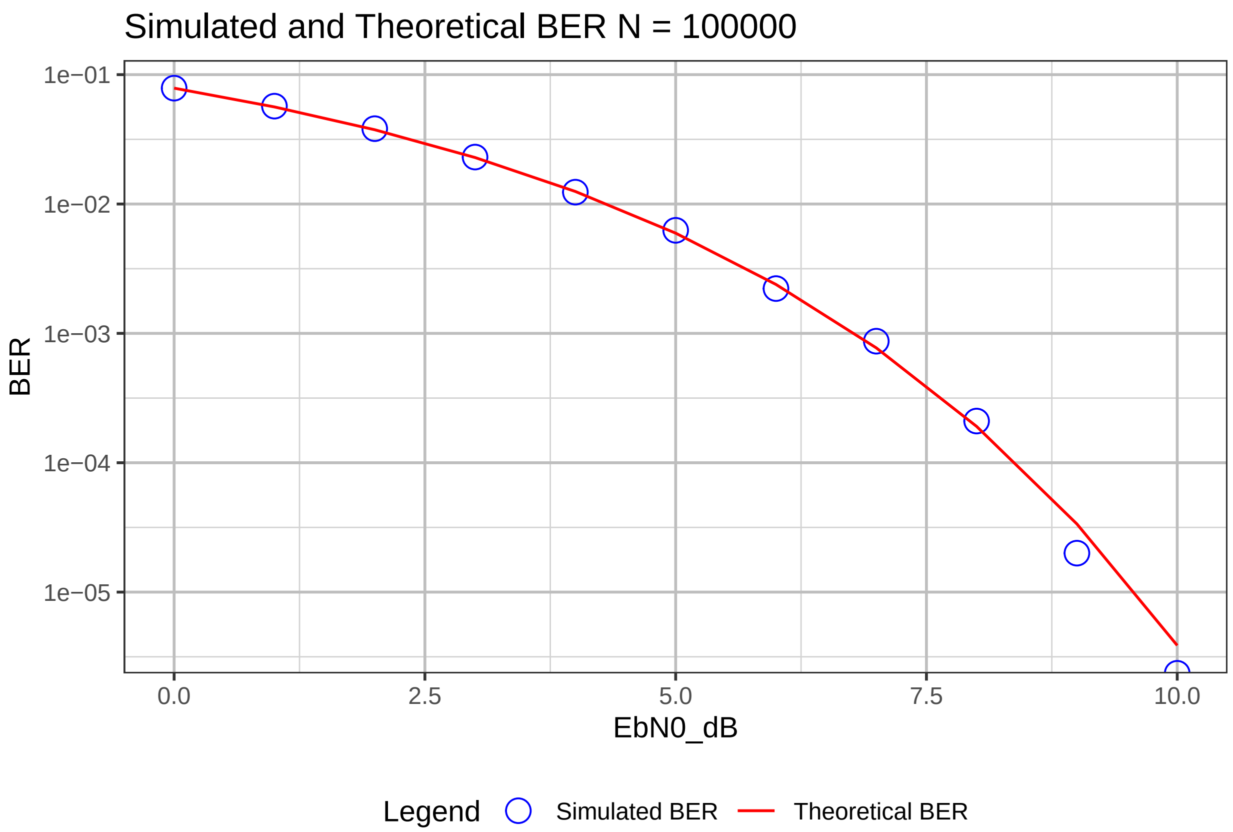
```r
# Load the required package
library(ggplot2)

# Read the CSV file
data <- read.csv("BER_data_N10000.csv")

# Create the plot
ggplot(data, aes(x = EbN0_dB)) +
  geom_point(aes(y = Simulated_BER, color = "Simulated BER"), shape = 1,size = 4) +  # Set color inside
  geom_line(aes(y = Theoretical_BER, color = "Theoretical BER")) +  # Set color inside aes() for legend
  scale_y_log10() +
  xlab("EbN0_dB") +
  ylab("BER") +
  ggtitle("Simulated and Theoretical BER N = 10000") +
  theme_bw() +  # Base theme with grid
  theme(panel.grid.major = element_line(color = "grey", size = 0.5),  # Customize major grid lines
        panel.grid.minor = element_line(color = "lightgrey", size = 0.25),  # Customize minor grid line
        legend.position = "bottom") +  # Adjust legend position
  scale_color_manual(name = "Legend", values = c("Simulated BER" = "blue", "Theoretical BER" = "red")) +
  guides(color = guide_legend(override.aes = list(shape = c(1, NA))))  # Ensure correct shapes in legen
```

```
## Warning in scale_y_log10(): log-10 transformation introduced infinite values.
```

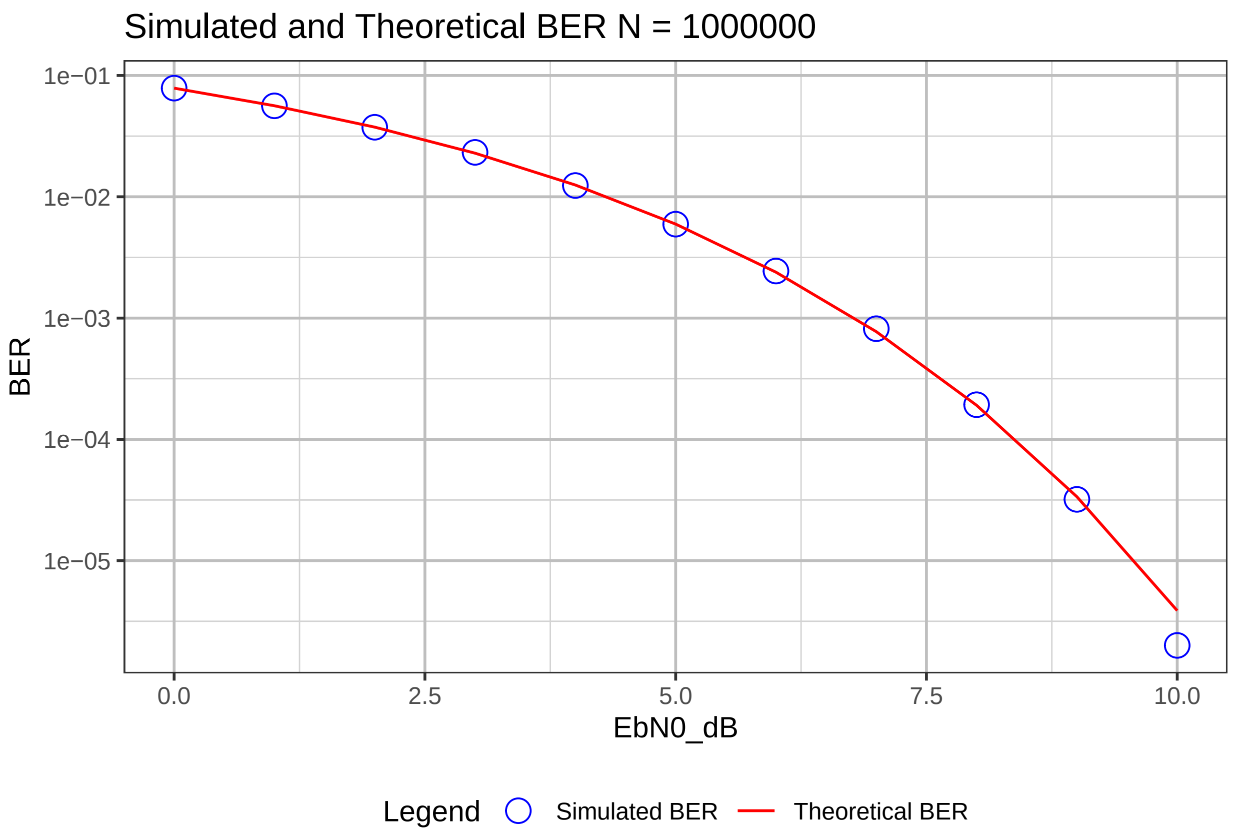


```r
# Load the required package
library(ggplot2)

# Read the CSV file
data <- read.csv("BER_data_N100000.csv")

# Create the plot
ggplot(data, aes(x = EbN0_dB)) +
  geom_point(aes(y = Simulated_BER, color = "Simulated BER"), shape = 1,size = 4) +  # Set color inside
  geom_line(aes(y = Theoretical_BER, color = "Theoretical BER")) +  # Set color inside aes() for legend
  scale_y_log10() +
  xlab("EbN0_dB") +
  ylab("BER") +
  ggtitle("Simulated and Theoretical BER N = 100000") +
  theme_bw() +  # Base theme with grid
  theme(panel.grid.major = element_line(color = "grey", size = 0.5),  # Customize major grid lines
        panel.grid.minor = element_line(color = "lightgrey", size = 0.25),  # Customize minor grid line
        legend.position = "bottom") +  # Adjust legend position
  scale_color_manual(name = "Legend", values = c("Simulated BER" = "blue", "Theoretical BER" = "red")) +
  guides(color = guide_legend(override.aes = list(shape = c(1, NA))))  # Ensure correct shapes in legen
```

```
## Warning in scale_y_log10(): log-10 transformation introduced infinite values.
```



```
# Load the required package
library(ggplot2)

# Read the CSV file
data <- read.csv("BER_data_N1000000.csv")


# Create the plot
ggplot(data, aes(x = EbN0_dB)) +
  geom_point(aes(y = Simulated_BER, color = "Simulated BER"), shape = 1,size = 4) +  # Set color inside
  geom_line(aes(y = Theoretical_BER, color = "Theoretical BER")) +  # Set color inside aes() for legend
  scale_y_log10() +
  xlab("EbN0_dB") +
  ylab("BER") +
  ggtitle("Simulated and Theoretical BER N = 1000000") +
  theme_bw() +  # Base theme with grid
  theme(panel.grid.major = element_line(color = "grey", size = 0.5),  # Customize major grid lines
        panel.grid.minor = element_line(color = "lightgrey", size = 0.25),  # Customize minor grid line
        legend.position = "bottom") +  # Adjust legend position
  scale_color_manual(name = "Legend", values = c("Simulated BER" = "blue", "Theoretical BER" = "red")) +
  guides(color = guide_legend(override.aes = list(shape = c(1, NA))))  # Ensure correct shapes in legen
```



The same goes for other modulation schemes, note that BPSK and QPSK have the same BER but the spectral efficiency of QPSK is double than that of BPSK. This implies that we could send more data reliably without sacrificing more energy.

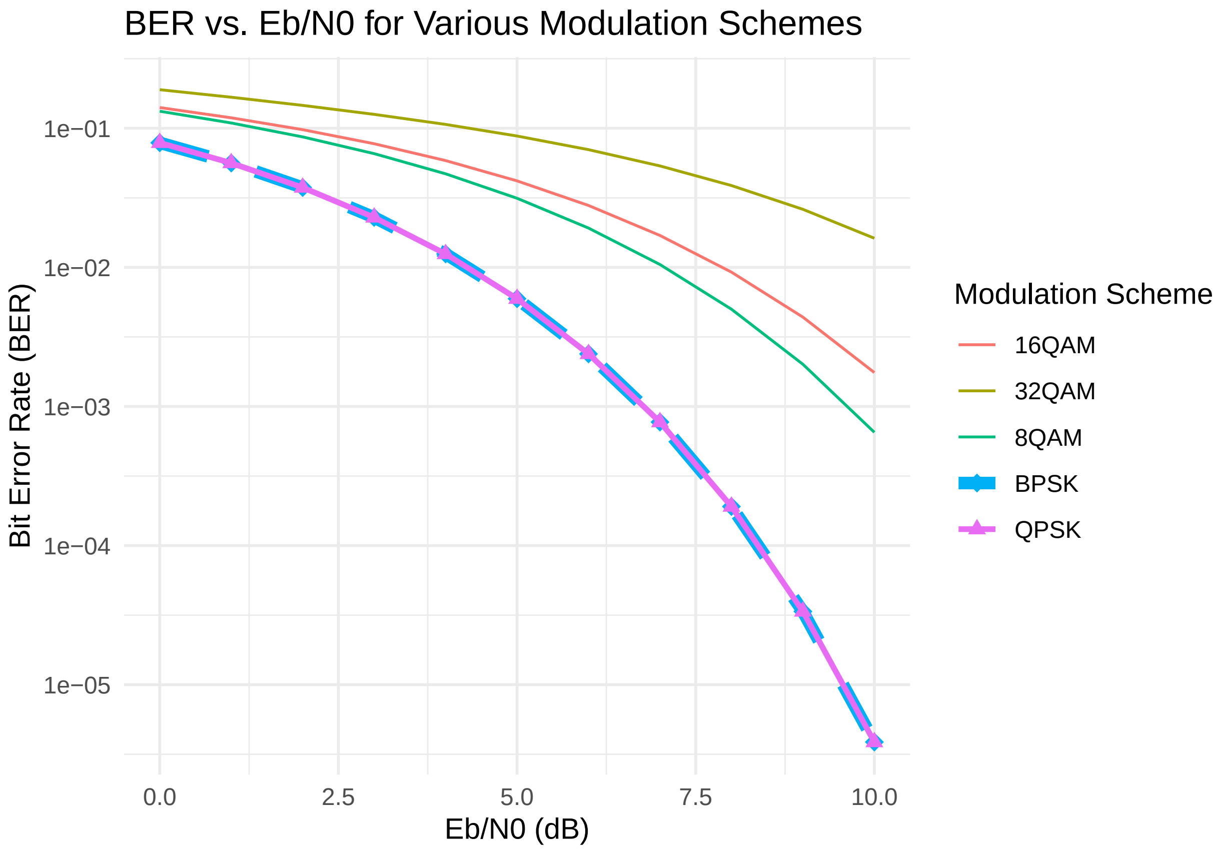
```r
# Load the necessary libraries
library(ggplot2)
library(readr)
```

```
## Warning: package 'readr' was built under R version 4.3.3
```

```r
# Read the data
data <- read_csv("BER_theoretical_data.csv")
```

```
## Rows: 11 Columns: 6
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## dbl (6): EbN0_dB, BPSK, QPSK, 8QAM, 16QAM, 32QAM
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```r
# Create the plot with a logarithmic y-axis
p <- ggplot(data, aes(x = EbN0_dB)) +
  geom_line(aes(y = BPSK, color = "BPSK"), linetype = "dashed", size = 2.2) +
  geom_point(aes(y = BPSK, color = "BPSK"), shape = 18, size = 3) + # Add points for BPSK
  geom_line(aes(y = QPSK, color = "QPSK"), linetype = "solid", size = 1.0) +
  geom_point(aes(y = QPSK, color = "QPSK"), shape = 17, size = 2) + # Add points for QPSK
  geom_line(aes(y = `8QAM`, color = "8QAM")) +
  geom_line(aes(y = `16QAM`, color = "16QAM")) +
  geom_line(aes(y = `32QAM`, color = "32QAM")) +
  labs(title = "BER vs. Eb/N0 for Various Modulation Schemes",
       x = "Eb/N0 (dB)",
       y = "Bit Error Rate (BER)",
       color = "Modulation Scheme") +
  scale_y_log10() +  # Apply logarithmic scale to y-axis
  theme_minimal()

# Display the plot
print(p)
```



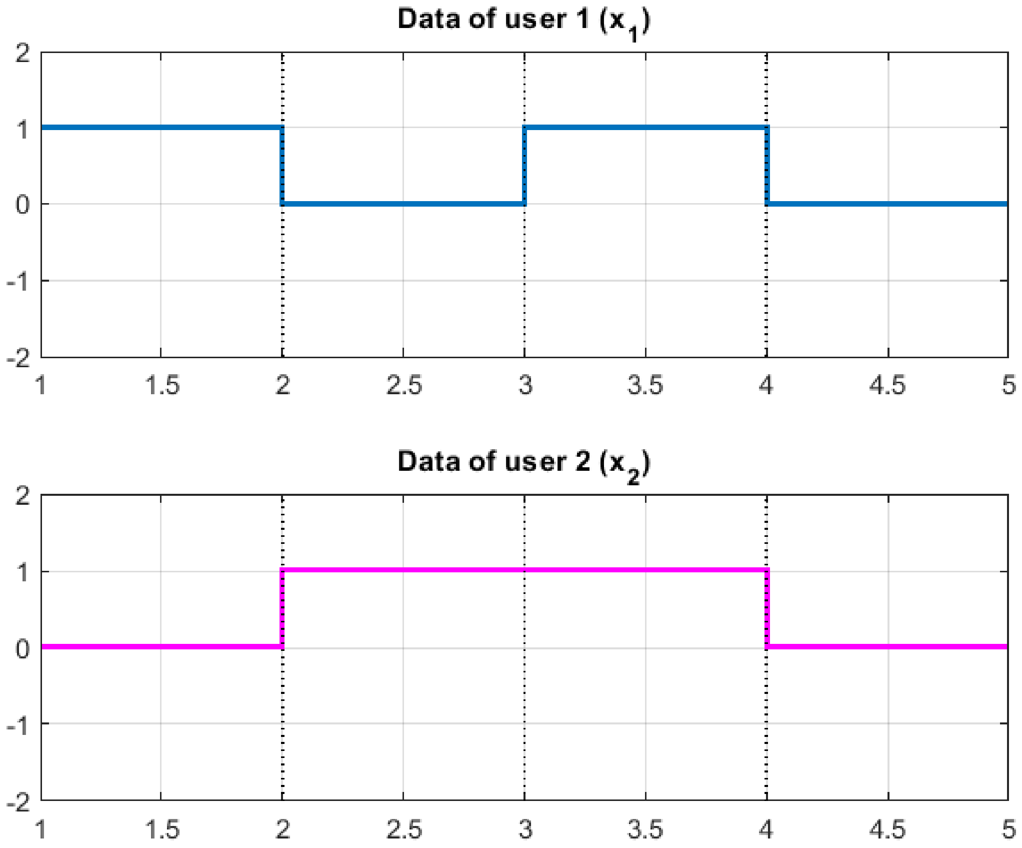
Introduction to Non-Orthogonal Multiple Access (NOMA)

Non-Orthogonal Multiple Access (NOMA) is a potential multiple access scheme for 5G networks. One intriguing aspect of NOMA is its ability to allow multiple users to transmit and receive simultaneously on the same frequency. This is made possible by two key operations: superposition coding at the transmitter side and successive interference cancellation (SIC) at the receiver side. In this section, we will focus on superposition coding.

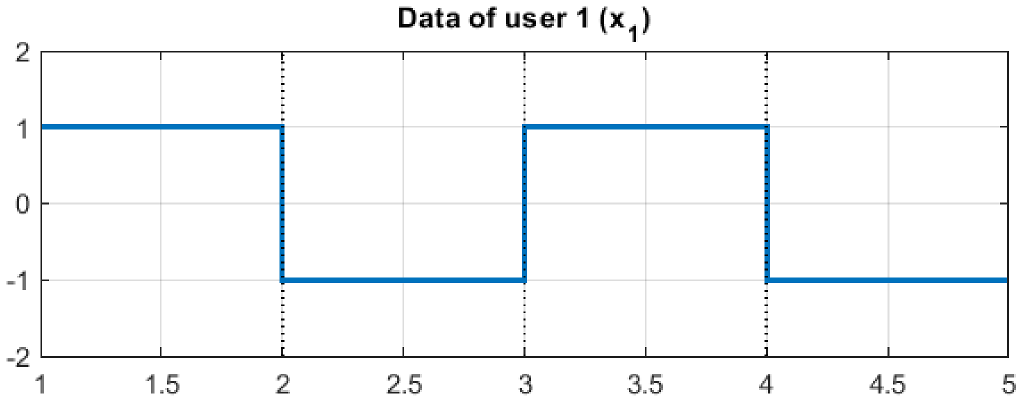
Example of Superposition Coding

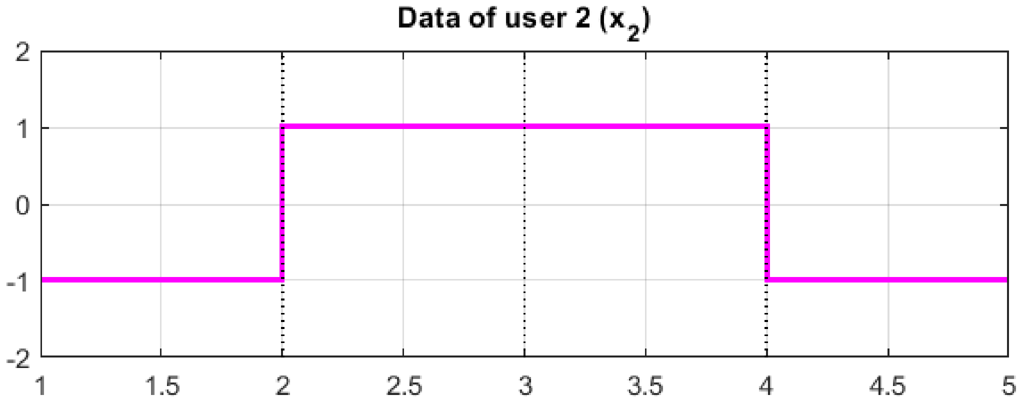
Let's consider two users, User 1 and User 2, communicating simultaneously using the same frequency. Let `x1` represent User 1's data and `x2` represent User 2's data. For simplicity, assume each user has 4 bits of data to send, denoted as follows:

- `x1 = 1010`
- `x2 = 0110`



Digital Modulation Before transmission, `x1` and `x2` must undergo digital modulation. Using Binary Phase Shift Keying (BPSK) for simplicity: - BPSK maps `0` to `-1` and `1` to `+1`. - After BPSK modulation: - `x1` becomes `+1 -1 +1 -1` - `x2` becomes `-1 +1 +1 -1`

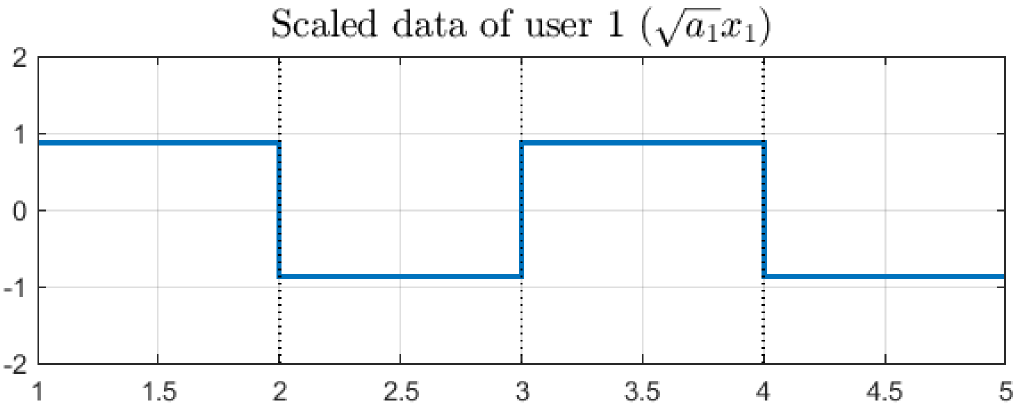




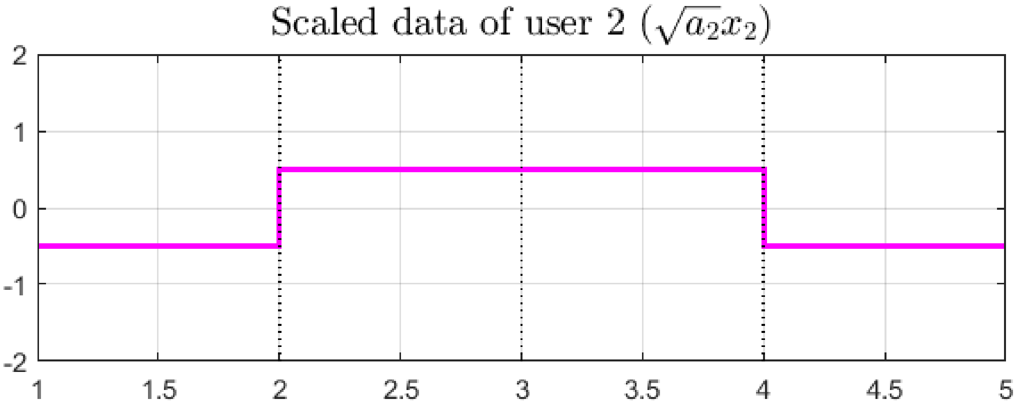
Superposition Coding Superposition coding involves power domain multiplexing, essentially adding `x1` and `x2` after scaling them with different power levels. Assume `a1 = 0.75` for User 1 and `a2 = 0.25` for User 2, with the rule that `a1 + a2 = 1`.

First, scale `x1` and `x2` with the square roots of `a1` and `a2` respectively: - `√a1 = √0.75 = 0.866` - `√a2 = √0.25 = 0.5`

After scaling: - `√a1 * x1 = 0.866 -0.866 0.866 -0.866` - `√a2 * x2 = -0.5 0.5 0.5 -0.5`

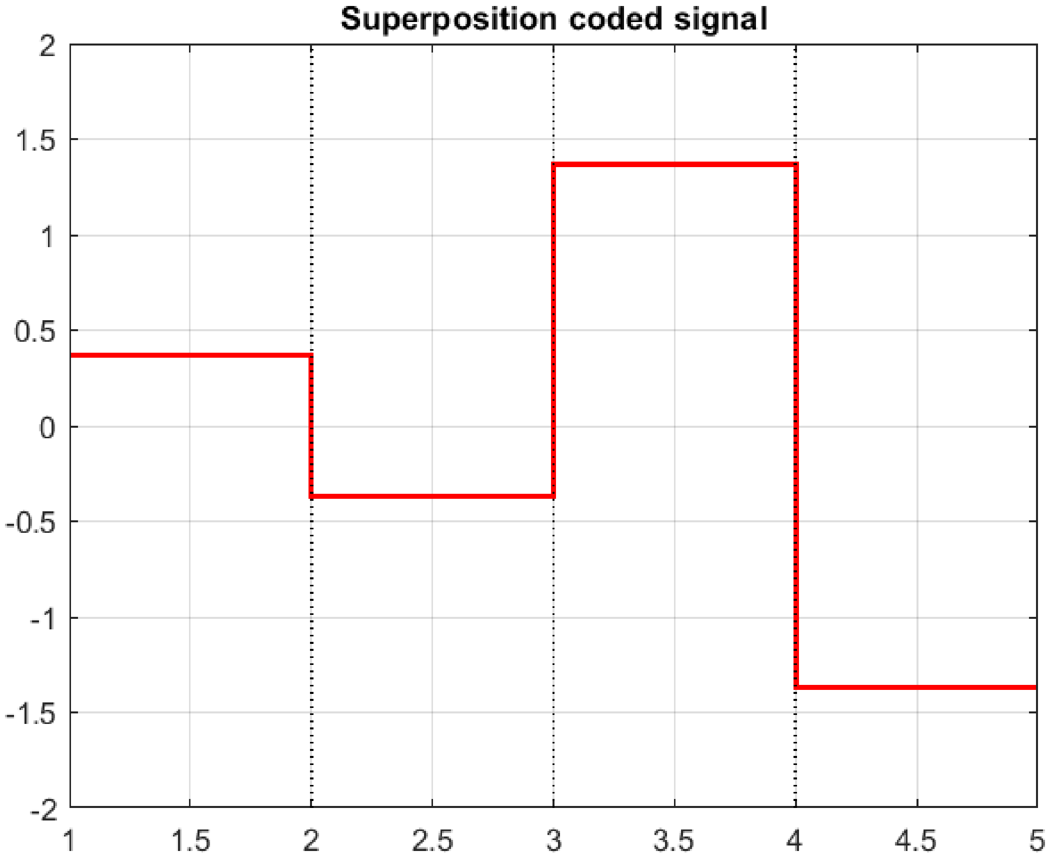


Scaled data of user 1 ($\sqrt{a_1}x_1$)



Scaled data of user 2 ($\sqrt{a_2}x_2$)

Next, add the scaled signals together to form the superposition coded signal: `- x = √a1 * x1 + √a2 * x2` - Resulting in `x = 0.366 -0.366 1.366 -1.366`

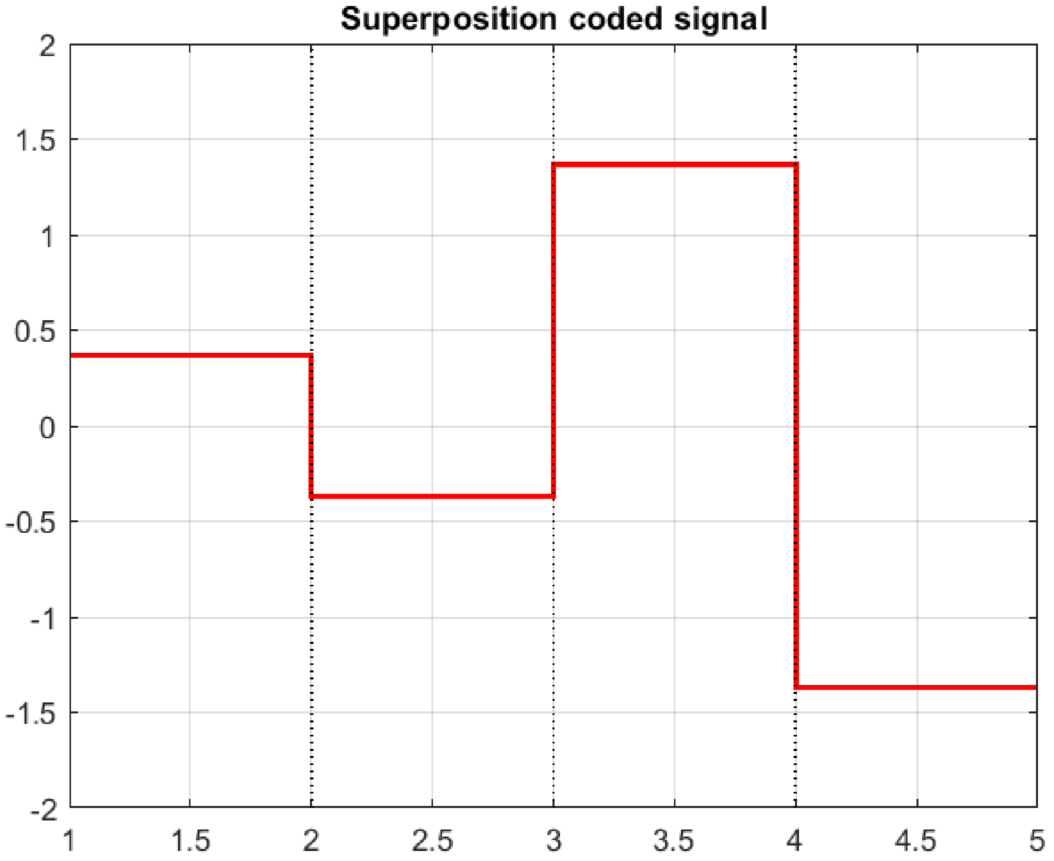


The signal `x` is the superposition coded NOMA signal that is transmitted into the channel. This demonstrates how superposition coding is performed. In the next section, we will explore how to recover `x1` and `x2` from `x` using successive interference cancellation (SIC).

Recap and Successive Interference Cancellation (SIC) in NOMA

Recap

Previously, we discussed two users with data `x1` and `x2`. We performed BPSK modulation on both `x1` and `x2` and then combined them using superposition coding to get the signal `x = √a1x1 + √a2x2`, where `a1` and `a2` are the power weights assigned to `x1` and `x2` respectively, such that `a1 + a2 = 1`. We used `x1 = 1010` and `x2 = 0110` as examples. After BPSK modulation and choosing `a1 = 0.75` and `a2 = 0.25`, we obtained the superposition coded signal `x`.



### Successive Interference Cancellation (SIC)

Before decoding, let's understand the SIC algorithm. SIC is an iterative process where data is decoded in the order of decreasing power levels. The data of the user with the highest power is decoded first, followed by the data of the next highest power user, and so on. For our two-user NOMA system, the SIC steps are as follows:

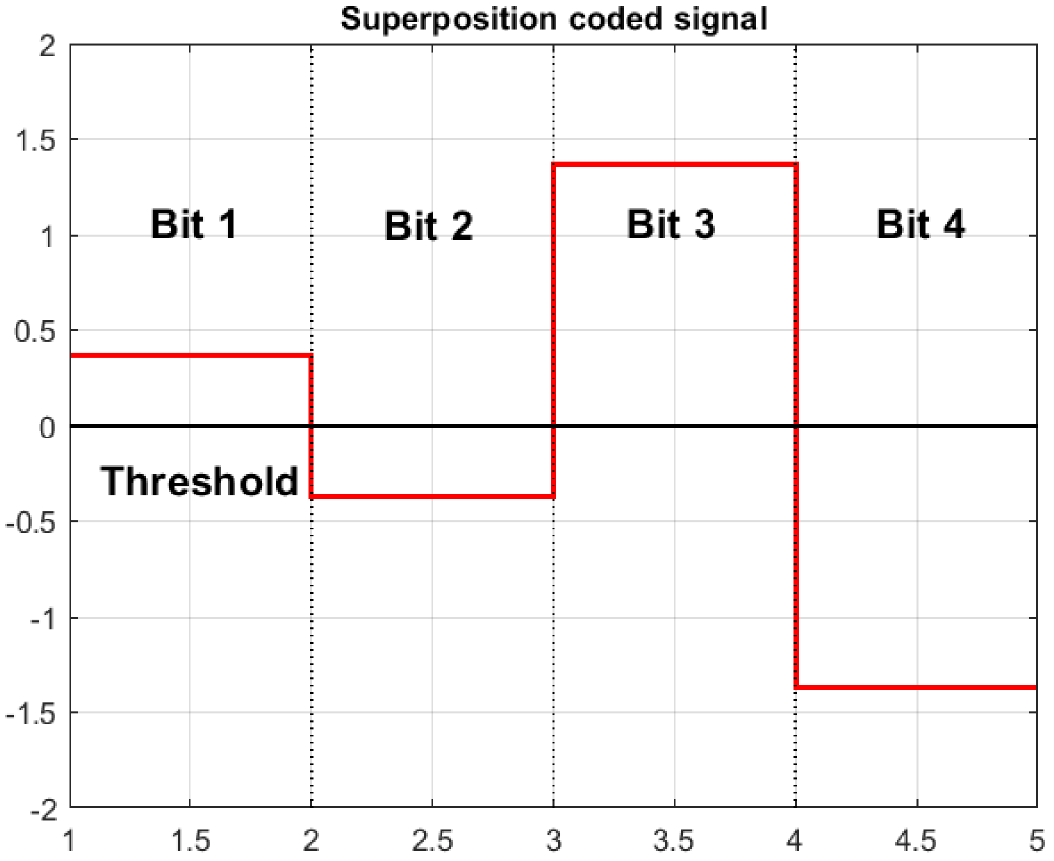
1. **Directly decode `x` to obtain the high-power signal**.
   - If `x1` has more weight (i.e., `a1 > a2`), direct decoding of `x` gives `x1`.
2. **Subtract the decoded high-power signal from `x`**.
   - Multiply the decoded `x1` by its corresponding weight and subtract it from `x`: `x - √a1x1`.
3. **Decode the remaining signal to get the low-power signal**.
   - Decode `x - √a1x1` to obtain `x2`.

Step-by-Step Example Given `a1 = 0.75` and `a2 = 0.25`, `x1` is given more power than `x2`. Following Step 1, direct decoding of `x` yields `x1`.

Direct decoding involves BPSK demodulation, which uses a simple thresholding method. Set the threshold to zero: if the amplitude exceeds zero, decode as `1`; otherwise, decode as `0`.

For `x = 0.366 -0.366 1.366 -1.366`: - First and third bit exceed zero: decode as `1`. - Second and fourth bit are below zero: decode as `0`.

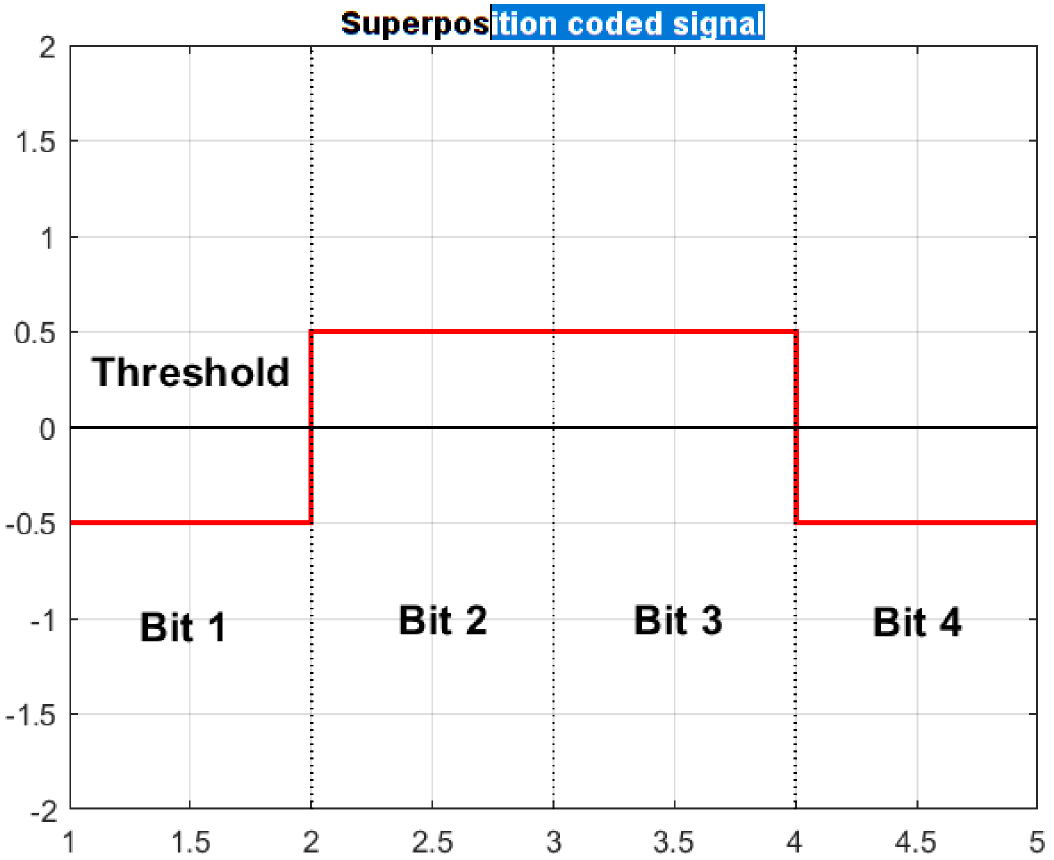
Thus, `x` is decoded to `1010`, which is `x1`.



SIC Steps

1. **Decode `x` to get `x1`**:
   - BPSK demodulation gives `1010` from `x`.
2. **Subtract `√a1x1` from `x`**:
   - Remember `x1 = 1010` was BPSK modulated to `1 -1 1 -1`.
   - Subtract `√a1 * (1 -1 1 -1)` from `x`.
3. **Decode the remaining signal**:
   - The resulting signal is similar to the scaled data of user 2.
   - Apply BPSK demodulation to this signal to get `x2`.

For `√a1 = 0.866` and `x1 = 1 -1 1 -1`: - Subtract `0.866 -0.866 0.866 -0.866` from `x`. - Result: `x - √a1x1 = -0.5 0.5 0.5 -0.5`.



Finally, demodulate this resulting signal: - First and fourth bit decode as `0`. - Second and third bit decode as `1`.

Thus, the decoded sequence is `0110`, which is `x2`.

This demonstrates that SIC can effectively separate `x1` and `x2` from the superposition coded signal `x`.

System Model

In a typical downlink communication scenario, consider a base station serving multiple users. Each user's channel from the base station is denoted by h_i, where i represents the user index. The channels are numbered such that user 1, with h_1, is the farthest and thus has the weakest signal, progressing to user N, who is nearest to the base station and has the strongest signal.

Signal Model for NOMA

Non-Orthogonal Multiple Access (NOMA) allows the transmission of multiple user data on the same frequency band by utilizing superposition coding. The base station sends out a combined signal:

$$x_{\text{NOMA}} = \sqrt{P}\left(\sqrt{\alpha_1}x_1 + \sqrt{\alpha_2}x_2 + \ldots + \sqrt{\alpha_N}x_N\right)$$

where P is the total transmit power and α_i are the power allocation coefficients. At each user's receiver, Successive Interference Cancellation (SIC) is used to decode the signals intended for them, starting from the strongest signal.

Signal Model for OMA

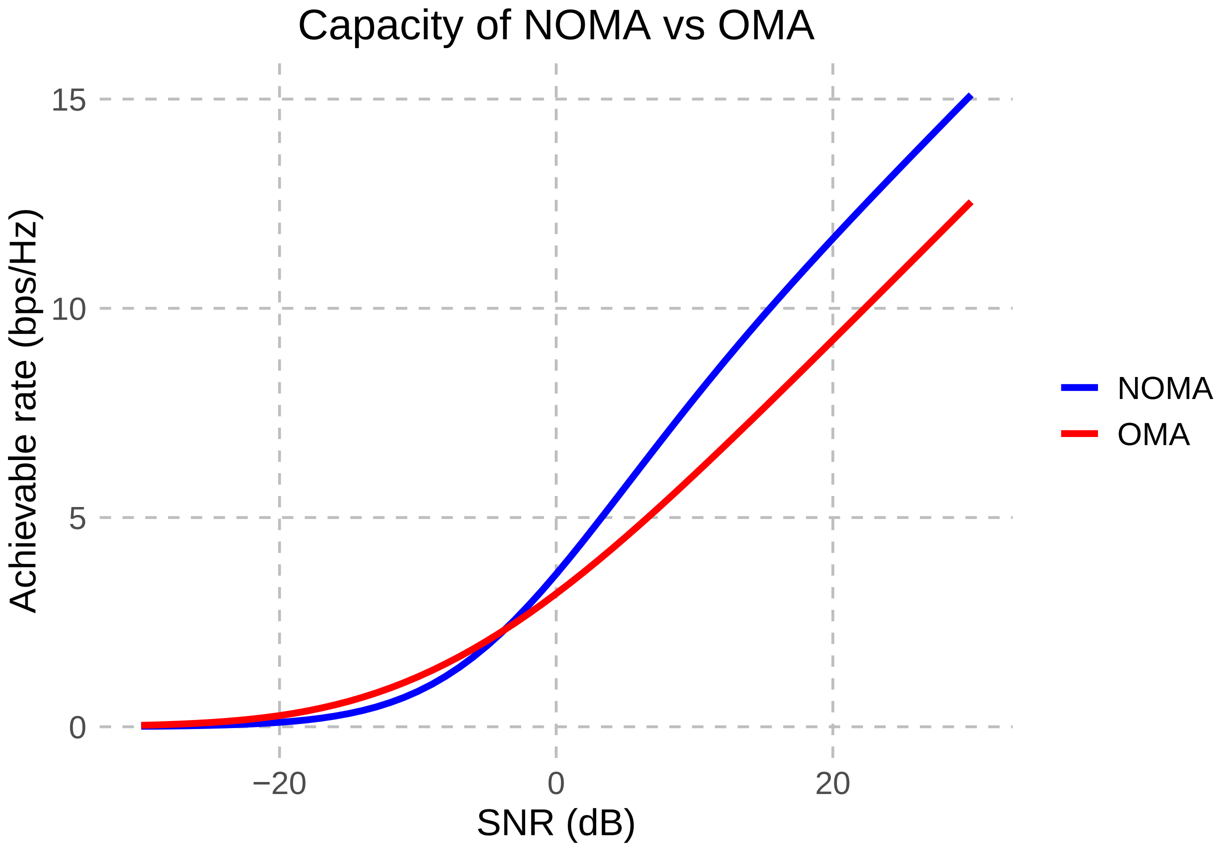
Orthogonal Multiple Access (OMA), such as Time Division Multiple Access (TDMA), assigns different time slots or frequency bands to each user, ensuring no overlap and hence no interference between the signals. For example, in TDMA:

$$x_{\text{OMA}} = \sqrt{P}x_i$$

during the i-th time slot, where i represents the user index.

Monte Carlo Simulation

```r
library(ggplot2)

# Clear workspace
rm(list = ls())

# SNR range
Pt <- seq(-114, -54, by = 1) # in dB
pt <- 10^(Pt/10) # in linear scale

N <- 10000

d1 <- 5; d2 <- 3; d3 <- 2 # Distance of users
eta <- 4 # Path loss exponent

# Rayleigh fading coefficients of both users
h1 <- (sqrt(d1^-eta)) * (complex(real = rnorm(N), imaginary = rnorm(N)) / sqrt(2))
h2 <- (sqrt(d2^-eta)) * (complex(real = rnorm(N), imaginary = rnorm(N)) / sqrt(2))
h3 <- (sqrt(d3^-eta)) * (complex(real = rnorm(N), imaginary = rnorm(N)) / sqrt(2))

# Channel gains
g1 <- Mod(h1)^2
g2 <- Mod(h2)^2
g3 <- Mod(h3)^2

BW <- 10^9
No <- -174 + 10*log10(BW)
no <- (10^-3)*10^(No/10)

# Power allocation coefficients
a1 <- 0.75; a2 <- 0.1825; a3 <- 1 - (a1 + a2)

C_noma_sum <- rep(0, length(pt))
C_oma_sum <- rep(0, length(pt))

for (u in 1:length(pt)) {

  # NOMA capacity calculation
  C_noma_1 <- log2(1 + pt[u]*a1*g1 / (pt[u]*a2*g1 + pt[u]*a3*g1 + no))
  C_noma_2 <- log2(1 + pt[u]*a2*g2 / (pt[u]*a3*g2 + no))
  C_noma_3 <- log2(1 + pt[u]*a3*g3 / no)
  C_noma_sum[u] <- mean(C_noma_1 + C_noma_2 + C_noma_3) # Sum capacity of NOMA
  # OMA capacity calculation
  C_oma_1 <- (1/3)*log2(1 + pt[u]*g1/no) # User 1
  C_oma_2 <- (1/3)*log2(1 + pt[u]*g2/no) # User 2
  C_oma_3 <- (1/3)*log2(1 + pt[u]*g3/no) # User 3
  C_oma_sum[u] <- mean(C_oma_1 + C_oma_2 + C_oma_3) # Sum capacity of OMA

}

SNR <- Pt - No

# Data frame for ggplot
data <- data.frame(SNR, NOMA = C_noma_sum, OMA = C_oma_sum)

# Plot using ggplot2
ggplot(data, aes(x = SNR)) +
  geom_line(aes(y = NOMA, colour = "NOMA"), size = 1.2) +
  geom_line(aes(y = OMA, colour = "OMA"), size = 1.2) +
  labs(x = "SNR (dB)", y = "Achievable rate (bps/Hz)", title = "Capacity of NOMA vs OMA") +
  scale_color_manual(values = c("NOMA" = "blue", "OMA" = "red")) +
  theme_minimal() +
  theme(legend.title = element_blank(),
        plot.title = element_text(hjust = 0.5, size = 16),
        axis.text = element_text(size = 12),
        axis.title = element_text(size = 14),
        legend.text = element_text(size = 12)) +
  theme(panel.grid.major = element_line(colour = "gray", linetype = "dashed"),
        panel.grid.minor = element_blank())
```



### Limitation

The limitation of this project is that Monte Carlo is excessively time consuming. Therefore, we overcome this limitation by using multicore to conduct these simulations.

Future direction

In future work, while deriving a closed form for the bit error rate (BER) can obviate the need for extensive Monte-Carlo simulations, the search for such a closed form is complex and introduces delays in research progress. If the time required to find a closed form significantly exceeds the time needed for simulations, it is not justifiable. Identifying the integration limits for BER events involving two or three random variables is feasible through plotting and inspection. However, as communication systems become increasingly complex, SNR outage, secrecy outage events and their corresponding bit error rate extension may involve four or more random variables, surpassing the limits of inspection-based methods. These events are typically described by systems of nonlinear polynomial inequalities, suggesting that an algebraic geometry approach to characterizing the solution sets of such systems may be a viable direction for future research.

Reference

[1] Shannon, Claude Elwood. "A mathematical theory of communication." The Bell system technical journal 27.3 (1948): 379-423.
[2] Tse, David, and Pramod Viswanath. Fundamentals of wireless communication. Cambridge university press, 2005.
[3] Tranter, William, et al. Principles of communication systems simulation with wireless applications. Prentice Hall Press, 2003.