**Appendix 1** Simulation code

1. Firstly remove the saved data in the memory of R software and save the current time for measuring the simulation time.

rm(list = ls()) # Delete everything that is in R's memory  
time <- Sys.time() # Save system time

2. Set up the working directory to let the software know where is the folder we are using. The last line of code is for automatically setting the working directory to the file location.

# Set the working file directory path. Paste your working directory within the quotes  
# Ensure that the backslashes \ are changed to forward slashes /  
# You can put as many file paths in as you like here, R will take the last one that worked  
# The try function simply masks error reporting: it will work if one of the directories attempted is correct  
  
try(setwd("Your working directory 1"), silent = TRUE)  
try(setwd("Your working directory 2"), silent = TRUE)  
try(setwd(dirname(sys.frame(1)$ofile)), silent = TRUE) # Drive file path  
try(setwd(dirname(rstudioapi::getActiveDocumentContext()$path)), silent = TRUE) # Drive file path

3. Import saved parameter inputs by reading the files in *InputFiles* folder.

# Import the required data from the data sheets created by the Excel file  
InputTable <- data.frame(read.table(file = "InputFiles/InputTable.txt", header = T, row.names = 1)) # Apply the first column as row names  
DefineStages <- data.frame(read.table(file = "InputFiles/Stages.txt", header = T)) # Stage; Name;Type;Scale;Shape;Utility  
Incidence <- data.frame(read.table(file = "InputFiles/Incidence.txt", header = T)) # Age\_Interval; Annual\_Incidence; Cumulative\_Failure  
Survival <- data.frame(read.table(file = "InputFiles/LifeTable.txt", header = T)) # Interval\_LifeTable; Survival  
ScreenSchedule <- data.frame(read.table(file = "InputFiles/ScreenSchedule.txt", header = T)) # ScheduleNumber; StartAge; StopAge; TestApplied1; Interval1  
TestPerformance <- data.frame(read.table(file = "InputFiles/TestPerformance.txt", header = T)) # Test; Sn; Sp; Disutility  
TreatmentSuccess <- data.frame(read.table(file = "InputFiles/TreatmentSuccess.txt", header = T)) # PreClinicalProbability; ClinicalProbability  
Disutility <- data.frame(read.table(file = "InputFiles/OtherDisutility.txt", header = T)) # PreClinicalProbability; ClinicalProbability  
DiscountRates <- data.frame(read.table(file = "InputFiles/DiscountRates.txt", header = T)) # Costs; Effects; DiscountYear  
Costs <- data.frame(read.table(file = "InputFiles/Costs.txt", header = T)) # PrimaryScreen; FollowUp; TreatmentScreenDetected; TreatmentClinical  
source("InputFiles/Misc.txt") # SampleSize; Threshold  
source("InputFiles/Options.txt")

4. Prepare an Input table that has the parameter values for every scenario. First, make a list of parameters that we want to simulate for one-way sensitivity analysis. Remove the symbol # in front of the parameter to include this parameter for sensitivity analysis.

Then, create a data frame Input from InputTable which the first column records the parameter values for the base-case scenario. InputTable is a table recorded all the parameter input values in Excel.

Create a string to save the names of scenarios which will be used as column names of Input table. We save Input, the name of the first column of this table, to the variable ScenarioNames.

# To make a list of parameters that are included in scenario analysis  
ParameterNames <- c("StageScale2",  
 "StageScale3",  
 #"StageUtility1", # These parameters are not included in our scenario analysis  
 #"StageUtility2",  
 #"StageUtility3",  
 "TestSensitivity1",  
 "TestSpecificity1",  
 #"TestDisutility1",  
 "PreClinicalProbability",  
 "ClinicalProbability",  
 #"DisutilityTriage",  
 #"DisutilityTrt",  
 #"DiscountRateCost",  
 #"DiscountRateEffect",  
 #"DiscountYear",  
 "CostPrimaryScreen",  
 "CostFollowUp",  
 "CostTrtScreen",  
 "CostTrtClinical",  
 "Incidence", # In this example, we change the value of incidence and survival proportionally across age groups  
 "Survival"  
 )  
  
Input <- data.frame(InputTable[, "Input"], row.names = rownames(InputTable)) # Create a table to record parameter inputs  
ScenarioNames <- "Input" # Create a variable to save the names of scenarios

5. Prepare the parameters’ input for each scenario if the user intends to simulate the one-way sensitivity analysis. Each parameter has two values separately for the scenario of low value and the scenario of high value. Each scenario only changes one parameter at a time and the rest parameters remain the fixed default values as the base-case scenario. Two loops are created for both parameters and scenarios. ParameterNames are defined in step 4 and ScenarioValue includes “Low” and “High”. Because incidence and survival are defined by a series of parameters, symbolling different age groups, we have the parameters associated with these two parameters change simultaneously for one scenario. In the end of the loop, assign the variable ScenarioNames to the table Input.

if (ScenarioAnaylsis == TRUE){  
 ScenarioValue <- c("Low", "High") # Our default has two scenarios for each parameter  
 for (Parameter in ParameterNames){ # The list of parameters for scenario analysis  
 for (Scenarios in ScenarioValue){  
 InputValue <- InputTable[, "Input"]  
 if (!(Parameter %in% c("Incidence", "Survival"))){ # Parameters, incidence and survival, have different risks for age groups  
 InputValue[rownames(InputTable) %in% Parameter] <- InputTable[rownames(InputTable) %in% Parameter, Scenarios]  
 } else if (Parameter == "Incidence") {  
 InputValue[rownames(InputTable) %in% paste("Incidence", Incidence$AgeInterval, sep = "\_")] <-   
 InputTable[rownames(InputTable) %in% paste("Incidence", Incidence$AgeInterval, sep = "\_"), Scenarios]  
 } else {  
 InputValue[rownames(InputTable) %in% paste("Survival", Survival$AgeInterval, sep = "\_")] <-   
 InputTable[rownames(InputTable) %in% paste("Survival", Survival$AgeInterval, sep = "\_"), Scenarios]  
 }  
 Input <- cbind(Input, InputValue) # Save parameter value to the input table  
 ScenarioNames <- c(ScenarioNames, paste(Parameter, Scenarios, sep = "\_"))  
 }  
 }   
}  
  
colnames(Input) <- ScenarioNames

6. Define the screen schedules. If the user intends to simulate the screening schedules defined in Excel, we saved the relevant parameters into strings for later simulation. If the user wants to systematically define the schedules, run the code below to produce all the possible screening strategies. The default is screening starting at age 25 and stop at age 100 with a fixed interval of 1-10 years, which includes the one-time screening within this age range. The first modality defined in Excel remains the same, as the only screening modality applied in the screening programmes. Create a list to record all the strategy names.

# We are able to redefine the screen schedule  
if (ExcelDefinedScreening == TRUE){  
 StartAges <- ScreenSchedule[, "StartAge"]  
 StopAges <- ScreenSchedule[, "StopAge"]  
 IntervalSwitchAge1s <- ScreenSchedule[, "IntervalSwitchAge1"]  
 IntervalSwitchAge2s <- ScreenSchedule[, "IntervalSwitchAge2"]  
 IntervalSwitchAge3s <- ScreenSchedule[, "IntervalSwitchAge3"]  
 Interval1s <- ScreenSchedule[, "Interval1"]  
 Interval2s <- ScreenSchedule[, "Interval2"]  
 Interval3s <- ScreenSchedule[, "Interval3"]  
 Interval4s <- ScreenSchedule[, "Interval4"]  
 TestSwitchAges <- ScreenSchedule[, "ScreenSwitchAge"]  
 ScreenTest1s <- ScreenSchedule[, "TestApplied1"]  
 ScreenTest2s <- ScreenSchedule[, "TestApplied2"]  
} else {  
 # Simulate all the possible screening strategies, assuming the screen interval remains fixed throughout the programme  
 Start <- 25 # The starting age of screening  
 Stop <- 100 # The stop age of screening  
 Frequency <- c(1:10) # The screen interval is an integer in the range of 1-10 years  
   
 # One-time screen in the lifetime  
 StartAges <- StopAges <- c(seq(Start, Stop, by = 1))  
 Interval1s <- rep(0, (Stop - Start + 1)) # No screening interval  
 ScreenTest1s <- rep(1, (Stop - Start + 1)) # We assume all the strategies use the first screening modality  
 IntervalSwitchAge1s <- IntervalSwitchAge2s <- IntervalSwitchAge3s <-   
 Interval2s <- Interval3s <- Interval4s <-   
 TestSwitchAges <- ScreenTest2s <- rep(NA, (100 - 25 + 1)) # No screening interval and other screening modality  
   
 # The example here only changes starting age, stop age, and screening intervals  
 for (i in c(seq(Start ,Stop, by = 1))){   
 for (j in c(seq(Start, Stop, by = 1))){  
 for (k in Frequency){  
 if ((i < j) & ((j - i) %% k == 0)){ # Remove non-integer screens   
 StartAges <- c(StartAges, i)  
 StopAges <- c(StopAges, j)  
 IntervalSwitchAge1s <- c(IntervalSwitchAge1s, NA)  
 IntervalSwitchAge2s <- c(IntervalSwitchAge2s, NA)  
 IntervalSwitchAge3s <- c(IntervalSwitchAge3s, NA)  
 Interval1s <- c(Interval1s, k)  
 Interval2s <- c(Interval2s, NA)  
 Interval3s <- c(Interval3s, NA)  
 Interval4s <- c(Interval4s, NA)  
 TestSwitchAges <- c(TestSwitchAges, NA)  
 ScreenTest1s <- c(ScreenTest1s, 1) # We assume all the strategies use the same screening modality  
 ScreenTest2s <- c(ScreenTest2s, NA)   
 }  
 }  
 }  
 }  
}  
  
# We give strategies shorter names  
strategies <- paste(StartAges, StopAges, Interval1s, sep = "\_")  
#strategies <- paste(StartAges, StopAges, Interval1s, IntervalSwitchAge1s, IntervalSwitchAge2s, IntervalSwitchAge3s,  
# Interval2s, Interval3s, Interval4s, TestSwitchAges, ScreenTest1s, ScreenTest2s, sep = "\_")

7. Run all the scenarios with a loop of all the input values defined in Input. Each column has a set of parameter values for one simulation.

for (CurrentRun in c(1: ncol(Input))){

8. Have a fixed random seed for the simulation, so we can have identical output when we want to compare results. The number is changeable but needs to be the same if intending to produce constant outputs.

set.seed(2021) # Set a seed to be able to reproduce the same results

9. Create an outcome table to record the disease history for all the individuals. Our model in default has five health states. With an additional column to record individuals’ ID numbers, create an array Outcomes of six columns and the number of rows equals the length of the sample size.

# Define number of health stages with the stage arrival  
 # This model only features five states: (1)Disease Free; (2)Preclinical Disease; (3)Clinical Disease; (4)Cause-Specific Death; (5)Other-Cause Death  
 Outcomes <- array(NA, dim = c(SampleSize, 6)) # Create an array of the length of the sample size  
 colnames(Outcomes) <- c("PersonNumber", paste(DefineStages[2:nrow(DefineStages), "Name"]), "AllCauseDeath") # Set column names  
 Outcomes[, "PersonNumber"] <- c(1:SampleSize) # Set the first column to be the unique person-number for each individual

10. Create a function, OnsetFunction, to produce the sampled age of entering a specific health state for each individual from an age-specific probability. This is based on linear interpolation that is useful for filling the gaps with the limited known data points on the curve. It will be applied to simulate the age of entering the preclinical state and other-cause death based on the probabilities for different age groups.

# The intervening columns correspond to the arrival of the intermediate disease states  
 # This model studies the cohort with the same age, so here does not need to simulate the "Stage1Arrival"  
 # Define the generic onset function which applies an age-specific probability of entering a specific stage  
 OnsetFunction <- function(x){  
 unlist(approx(probability, age, x, ties = max)[2], use.names = F) # Ties = max is required because of the possibility of multiple zero probabilities of disease at younger ages  
 }

11. Calculate the cumulative probability of being disease-free. To achieve this, we set the initial CumulativeFailure to one, meaning nobody gets the disease. Then, run a loop to calculate the cumulative probabilities by the previous probability of having the disease-free minus the probability of getting disease within that age range. The probability of getting the disease within that age range is the annual probability of getting the disease times by the length of years for a specific age group.

CumulativeFailure <- 1  
 IncidenceTable <- data.frame(Incidence$AgeInterval,   
 as.numeric(Input[paste("Incidence", Incidence$AgeInterval, sep = "\_"), CurrentRun]),   
 CumulativeFailure  
 )  
 colnames(IncidenceTable) <- c("AgeInterval", "AnnualIncidence", "CumulativeFailure")  
 for (c in c(2:nrow(IncidenceTable))){  
 IncidenceTable[c, "CumulativeFailure"] <- (IncidenceTable[c - 1, "CumulativeFailure"] - (IncidenceTable[c, "AnnualIncidence"] \* (IncidenceTable[c, "AgeInterval"] - IncidenceTable[c - 1, "AgeInterval"])))  
 }

12. To run the OnsetFunction, remove the constant probabilities for the age groups in their early age or late age. In our example, there is no incidence before age 30, so simulating the probability for age 30 is meaningless and is removed from the simulation.

# Define the specific values of probability and age for disease onset  
 probability <- IncidenceTable$CumulativeFailure  
 age <- IncidenceTable$AgeInterval  
 # The probability distribution need to be curtailed at the top and bottom for non-unique probability for the approx function to work as intended  
 age <- age[max(which(probability == max(probability))):length(probability)] # Remove the lower bound values  
 probability <- probability[max(which(probability == max(probability))):length(probability)] # Remove the lower bound values  
 age <- age[1:min(which(probability == min(probability)))] # Remove the upper bound values  
 probability <- probability[1:min(which(probability == min(probability)))] # Remove the upper bound values

13. Simulate the age of entering the preclinical health state based on the incidence of different age groups. The variable X saves the random values between 0 and 1 for each individual. The age of entering preclinical state is calculated based on the random value and its corresponding age with the self-defined function OnsetFunction.

x <- runif(SampleSize) # Create a vector of random values  
 Outcomes[, "Preclinical"] <- OnsetFunction(x) # Find the age of disease onset by applying the general function

14. Simulate the age of entering the other-cause death based on survival (life-table) by accommodating step 12 and step 13. Survival is already defined as the cumulative probabilities, so there is no need to recalculate it, but we need to remove the constant probabilities in the early age and the late age (same as step 12). Run step 13 but save the results to the column named OtherCauseDeath in the table Outcomes.

# Define the specific values of probability and age for other cause death  
 # The intervals in the age-specific incidence are defined by those used in the life table  
 probability <- as.numeric(Input[paste("Survival", Survival$AgeInterval, sep = "\_"), CurrentRun])  
 age <- Survival$AgeInterval  
 # The probability distribution need to be curtailed at the top and bottom for non-unique probability for the approx function to work as intended  
 age <- age[max(which(probability == max(probability))):length(probability)] # Remove the lower bound values  
 probability <- probability[max(which(probability == max(probability))):length(probability)] # Remove the lower bound values  
 age <- age[1:min(which(probability == min(probability)))] # Remove the upper bound values  
 probability <- probability[1:min(which(probability == min(probability)))] # Remove the upper bound values  
 x <- runif(SampleSize) # Create a vector of random values  
 Outcomes[, "OtherCauseDeath"] <- OnsetFunction(x) # Find the age at other-cause death by applying the generic onset function

15. Simulate the ages of entering other health states except for preclinical and other-cause death. The age of entering the health state is the age of entering the previous state plus the sojourn time of this previous state. Apply a loop for the number of stages minus one to exclude the final stage of other-cause death. Inside the loop, we create random numbers again for drawing the corresponding duration from the defined distributions. Distributions, including constant, exponential, and Weibull, can be chosen to sample the sojourn time of each health state. Notably, constant duration does not need a random sample because it assumes every individual has a fixed length of duration. After simulating the ages of entering health states, check if the ages are over 100 which is out of our simulation scope. This study assumes that all the people die before 100 years old and 100 is the maximum age of entering any health state in this simulation.

# The model needs a loop here to go through the disease stages  
 for (Stage in 1:(nrow(DefineStages) - 1)){  
 # Apply the sojourn time to the stages  
 # Retrieve the distribution type, scale and shape  
 DurationType <- Input[paste("StageType", Stage, sep = ""), CurrentRun]  
 DurationScale <- Input[paste("StageScale", Stage, sep = ""), CurrentRun]  
 DurationShape <- Input[paste("StageShape", Stage, sep = ""), CurrentRun]  
   
 if (!(is.na(DurationType))){  
 if (DurationType == 1){Duration <- rep(DurationScale, SampleSize)} # Set the preclinical distribution to be constant  
 if (DurationType == 2){Duration <- -(log(1 - runif(SampleSize))) \* DurationScale} # Set the preclinical distribution to be exponentially distributed  
 if (DurationType == 3){Duration <- ((-log(1 - runif(SampleSize))) ^ (1 / DurationShape)) \* DurationScale} # Set the preclinical distribution to be Weibull distribution, and the alternative code: rweibull(SampleSize, shape = DurationShape, scale = DurationScale)  
 Outcomes[, Stage + 1] <- Outcomes[, Stage] + Duration # Find the end of the preclinical period by adding the onset to the duration  
 }  
   
 # This study assumes all the people died before aged 100  
 Outcomes[which(Outcomes[, Stage + 1] > 100), Stage + 1] <- 100  
 }

16. Define AllCauseDeath by comparing an individual’s cause-specific death and other-cause death. The lower age is the age of all-cause death. Both two health states, cause-specific death and other-cause death, are absorbing states, meaning individuals cannot enter other states after they enter either of them.

# Find the all-cause death  
 Outcomes[, "AllCauseDeath"] <- pmin(Outcomes[, "CauseSpecificDeath"], Outcomes[, "OtherCauseDeath"], na.rm = TRUE)

17. Find individuals who are disease-free, sick, and clinically diagnosed in their disease history. Healthy individuals are those who never develop disease or those who develop disease after their other-cause deaths. Sick individuals are the complete cohort excluding the healthy individuals. Clinical cases are the sick individuals clinically presented before they die from other-cause.

# Identify individuals presenting with clinical disease  
 DiseaseFree <- which((Outcomes[, "Preclinical"] >= Outcomes[, "OtherCauseDeath"]) | (is.na(Outcomes[, "Preclinical"]))) # Censor onset ages greater than death; first identify those who never develop disease  
 Sick <- Outcomes[-DiseaseFree, "PersonNumber"]  
 ClinicalCases <- Sick[which(Outcomes[Sick, "Clinical"] <= Outcomes[Sick, "OtherCauseDeath"])]

18. Find cured cases and redefine their age of all-cause death. Create sample numbers for each individual, CureSeed, and then compare CureSeed to the treatment success probability. CuredCases are clinical cases whose sample number is lower than the treatment success probability. For those cured cases, they die from other-cause, instead of cause-specific death. Adjust the all-cause death in our Outcomes table for the cured cases.

# Identify individuals cured successfully  
 CureSeed <- runif(SampleSize) # Save random numbers for treatment  
 CuredCases <- ClinicalCases[which(CureSeed[ClinicalCases] <= Input["ClinicalProbability", CurrentRun])] # Find treatment to be successful if probabilities less than probability of successful treatment  
 Outcomes[CuredCases, "AllCauseDeath"] <- Outcomes[CuredCases, "OtherCauseDeath"]

19. Create two series of discounting factors for costs and effectiveness respectively. The discount year is also considered. For effectiveness, we create a variable Acc\_DF\_Effects for recording the cumulative discounted life-years. We discount the results on a discrete basis. The beginning of the simulation has the discounting factor equal to one (without discounting). A fixed discounting factor is applied to everything that happens throughout the year and the discounting factor changes at the end of every year.

# Create a reference series of discount factors for 100 years  
 DF\_Effects <- round((1 + Input["DiscountRateEffect", CurrentRun]) ^ -(c(1:100) - Input["DiscountYear", CurrentRun]), digits = 4)  
 DF\_Costs <- round((1 + Input["DiscountRateCost", CurrentRun]) ^ -(c(1:100) - Input["DiscountYear", CurrentRun]), digits = 4)  
 Acc\_DF\_Effects <- NA # Create a variable to save accumulated discounted life-years  
 for (year in c(1:100)){  
 Acc\_DF\_Effects <- c(Acc\_DF\_Effects, sum(DF\_Effects[1:year]))  
 }  
 Acc\_DF\_Effects <- Acc\_DF\_Effects[-1] # Remove NA

20. Create a self-defined function PresentValue to calculate the present value of effectiveness (life-years). The discounted life-years are calculated by referencing the integer part of the life-years to the accumulated discounted factors and the rest decimal numbers times to the discounting factor of the next year.

The life-years shorter than one year will be discounted by the first discount factor. If the input number is NA, then output NA.

# Define the present value function for effectiveness  
 PresentValue <- function(x){  
 if (x >= 1 & !(is.na(x))){  
 Acc\_DF\_Effects[trunc(x)] + (x - trunc(x)) \* DF\_Effects[trunc(x) + 1]  
 } else if (!(is.na(x))){  
 x \* DF\_Effects[1]  
 } else {  
 x  
 }  
 }

21. Prepare for calculating quality-of-life adjusted outcomes. First, we prepare the discount and undiscounted life-years. The undiscounted life-years are the individuals’ age of all-cause deaths. The discounted life-years are the applying the PresentValue function to the undiscounted life-years. We create SickOutcome which only sick people included, because sick people experience more complicated quality-of-life adjustment by entering other health states. Similarly, Dis\_SickOutcome is created for discounting results.

# Calculate the effects  
 UD\_QALYS <- Outcomes[, "AllCauseDeath"]  
 D\_QALYS <- D\_LYS <- sapply(UD\_QALYS, PresentValue)  
 StagesInOrder <- c(paste(DefineStages[2:(nrow(DefineStages) - 2), "Name"]), "AllCauseDeath")  
 SickOutcome <- as.data.frame(Outcomes[Sick, StagesInOrder]) # Create a table that only sick people included  
 Dis\_SickOutcome <- apply(SickOutcome, MARGIN = c(1, 2), PresentValue) # Discount life-years for each health state among sick people  
 Utility <- Input[paste("StageUtility", c(1:(nrow(DefineStages) - 2)), sep = ""), CurrentRun] # Quality-of-life utility weights for each state

22. Create a self-defined function Accumulated\_QALYS to calculate the sum of QALYs of all the health states. QALYs are calculated by the sojourn time of a specific health state times to its corresponding utility weight. Then sum up all the QALYs from every health state. The QALYs from the first health state are the duration of being disease-free times by the utility weight of the disease-free. The duration of being disease-free is the age of entering the preclinical state. Then, create a loop for the rest health states.

# Define the function for quality-of-life adjustment  
 Accumulated\_QALYS <- function(x, output){  
 QALYS <- x[1] \* Utility[1] # For the first health state, simply multiply the life-years by the corresponding utility weight  
 for (state in (2:length(StagesInOrder))){ # For the remaining health states  
 QALYS <- QALYS + Utility[state] \* (x[state] - x[state - 1]) # Add up the QALYs in each health state  
 if (state == length(StagesInOrder)){ # The last health state  
 break  
 }  
 }  
 return(QALYS)  
 }

23. Apply the Accumulated\_QALYS function to the sick people, including the undiscounted results, UD\_QALYS, and discounted results, D\_QALYS. For those who are not sick, adjust their QALYs with the utility weight of the healthy state.

UD\_QALYS[DiseaseFree] <- UD\_QALYS[DiseaseFree] \* Utility[1] # Quality-of-life adjusted but not discounted  
 UD\_QALYS[Sick] <- apply(SickOutcome, 1, Accumulated\_QALYS)  
 D\_QALYS[DiseaseFree] <- D\_QALYS[DiseaseFree] \* Utility[1] # Quality-of-life adjusted and discounted  
 D\_QALYS[Sick] <- apply(Dis\_SickOutcome, 1, Accumulated\_QALYS)

24. Calculate the dis-utility due to the treatment. Only clinical cases receive treatments and we assumed they received the treatment at the end of the year. In the case of calculating undiscounted QALYs loss due to treatment, simply multiply the number of clinical cases by the dis-utility of each treatment. For the calculation of discounted QALYs loss, every dis-utility is discounted by the corresponding discounting factor.

# Dis-utility due to the treatment  
 ClinicalOnsetAges <- ceiling(Outcomes[ClinicalCases, "Clinical"]) # Identify the clinical ages and round up to integers for discounting  
 UD\_TreatmentHarm <- length(ClinicalCases) \* Input["DisutilityTrt", CurrentRun] # Undiscounted QALYs loss due to treatment  
 D\_TreatmentHarm <- sum(Input["DisutilityTrt", CurrentRun] \* DF\_Effects[ClinicalOnsetAges]) # Discounted QALYs loss due to treatment

25. Sum up the life-years and QALYs of each individual, including discounted and undiscounted. The calculation of QALYs for the no-screening scenario needs to deduct the dis-utility due to treatment.

# Calculate the effectiveness outcomes without screening  
 UD\_NoScreening\_LYG <- sum(Outcomes[, "AllCauseDeath"])  
 D\_NoScreening\_LYG <- sum(D\_LYS)  
 UD\_NoScreening\_QALY <- sum(UD\_QALYS) - UD\_TreatmentHarm  
 D\_NoScreening\_QALY <- sum(D\_QALYS) - D\_TreatmentHarm

26. Estimate the cost of the no-screening scenario. In this study, we only consider the treatment cost, LateTreatmentCost, for the no-screening strategy which cost is applied to the patients presented clinically. Undiscounted cost is the number of clinical cases multiplied by the unit cost of each treatment. Discounted cost is the sum of the product of the unit cost of each treatment and the corresponding discounting factor.

# Retrieve the costs parameters  
 ScreenCost <- Input["CostPrimaryScreen", CurrentRun]  
 TriageCost <- Input["CostFollowUp", CurrentRun]  
 EarlyTreatmentCost <- Input["CostTrtScreen", CurrentRun]  
 LateTreatmentCost <- Input["CostTrtClinical", CurrentRun]  
   
 # Calculate the costs of treating clinical disease under no screening  
 UD\_NoScreening\_Costs <- length(ClinicalCases) \* LateTreatmentCost  
 D\_NoScreening\_Costs <- sum(DF\_Costs[ClinicalOnsetAges] \* LateTreatmentCost) # Find the discounted treatment costs

27. Calculate intermediate outcomes, including the average preclinical sojourn time, the number of cancer deaths, over-diagnosis, and clinical cases. The preclinical duration is the difference in the age of entering the clinical state and preclinical state. The number of cancer deaths is calculated by identifying whose all-cause death is equal to cause-specific death. Due to no-screening, there is no case of being over-diagnosed. The number of clinical cases is the length of the string ClinicalCases which records all the IDs of clinical cases.

# Calculate intermediate outcomes: effects  
 PreClinicalDuration <- mean(Outcomes[ClinicalCases, "Clinical"] - Outcomes[ClinicalCases, "Preclinical"]) # Average preclinical sojourn time  
 N\_CancerDeaths <- length(which(Outcomes[, "AllCauseDeath"] == Outcomes[, "CauseSpecificDeath"])) # The number of cases that die at the age of disease death  
 N\_Overdiagnosed <- 0  
 N\_ClinicalCases <- length(ClinicalCases) # The number of cases that present clinically

28. Create a data frame CostEffectivenessResults to record the main cost-effectiveness outcomes.

# Record cost-effectiveness results  
 CostEffectivenessResults <- data.frame(StrategyName = c("NoScreening", strategies),  
 UD\_EffectsL = as.numeric(UD\_NoScreening\_LYG),  
 UD\_EffectsQ = as.numeric(UD\_NoScreening\_QALY),  
 UD\_Costs = as.numeric(UD\_NoScreening\_Costs),  
 D\_EffectsL = as.numeric(D\_NoScreening\_LYG),  
 D\_EffectsQ = as.numeric(D\_NoScreening\_QALY),  
 D\_Costs = as.numeric(D\_NoScreening\_Costs),  
 UD\_ICER\_L = as.numeric(NA),  
 UD\_ICER\_Q = as.numeric(NA),  
 D\_ICER\_L = as.numeric(NA),  
 D\_ICER\_Q = as.numeric(NA)  
 )

29. Create a self-defined function, AgeDistribution, to track how individuals move among health states every year. Apply this function to the Outcomes table which records individuals’ disease history.

# Record age distribution of each health state  
 AgeDistribution<- function(x, output){  
 Age.cut <- cut(x, seq(0, 100, by = 1), right = FALSE)  
 return(table(Age.cut))  
 }  
 IntermediateOutcomes\_AgeDist <- apply(Outcomes[, !(colnames(Outcomes) %in% "PersonNumber")], 2, AgeDistribution)

30. Save the table IntermediateOutcomes\_AgeDist to the following folder.

write.table(IntermediateOutcomes\_AgeDist,   
 paste("OutputFiles/LatestResults/IntermediateOutcomtes/AgeDistribution/", colnames(Input)[CurrentRun], "\_NoScreening", ".txt", sep = ""),  
 row.names = F, col.names = T, sep = '\t')

31. Create a data frame IntermediateOutcomes to record the intermediate outcomes of each strategy. We have the results of the no-screening saved in the first row. The rest rows will be updated with the simulation outputs below.

# Record intermediate outcomes  
 IntermediateOutcomes <- data.frame(StrategyName = c("NoScreening", strategies),  
 UD\_ScreenCosts = as.numeric(0),  
 UD\_TriageCosts = as.numeric(0),  
 UD\_EarlyTreatmentCosts = as.numeric(0),  
 UD\_LateTreatmentCosts = as.numeric(UD\_NoScreening\_Costs),  
 D\_ScreenCosts = as.numeric(0),  
 D\_TriageCosts = as.numeric(0),  
 D\_EarlyTreatmentCosts = as.numeric(0),  
 D\_LateTreatmentCosts = as.numeric(D\_NoScreening\_Costs),  
 N\_CancerDeaths = as.numeric(N\_CancerDeaths),  
 N\_Overdiagnosed = as.numeric(N\_Overdiagnosed),  
 N\_ClinicalCases = as.numeric(N\_ClinicalCases)  
 )

32. Simulate screening strategies with a loop for the number of strategies defined previously.

for (Strategy in c(1:length(strategies))){

33. Print the progress of the simulation, showing the current run and the current strategy with the total number of simulations and strategies behind the slash separately.

# To track the progress  
 print(paste("Simulation ", CurrentRun, " / ", ncol(Input), " : Strategy ", Strategy, " / ", length(strategies), sep = ""))

34. Retrieve the parameters for defining the simulated screening programme.

# Retrieve the strategy-specific values  
 StartAge <- StartAges[Strategy]  
 StopAge <- StopAges[Strategy]  
 IntervalSwitchAge1 <- IntervalSwitchAge1s[Strategy]  
 IntervalSwitchAge2 <- IntervalSwitchAge2s[Strategy]  
 IntervalSwitchAge3 <- IntervalSwitchAge3s[Strategy]  
 Interval1 <- Interval1s[Strategy]  
 Interval2 <- Interval2s[Strategy]  
 Interval3 <- Interval3s[Strategy]  
 Interval4 <- Interval4s[Strategy]  
 TestSwitchAge <- TestSwitchAges[Strategy]  
 ScreenTest1 <- ScreenTest1s[Strategy]  
 ScreenTest2 <- ScreenTest2s[Strategy]

35. Adjust the screening schedules by rounding the number of screening. If Interval1 equals zero, only one screening is implemented throughout the lifetime and it happens at the screening starting age. If Interval1 is not zero, the screening ages are generated by the following code. First, create two strings to save the intervals, IntervalVector, and the ages changing the screening intervals, ChangeAges. Second, have a loop for the number of intervals. Inside the loop, we update the screening stop age from ChangeAges and use this stop age to find the closest number of screenings. If it can be round up to two numbers, this model chooses the larger number of screenings. Use this number of screenings to calculate the correct stop age and produce the screening ages before this stop age. The maximum of the screening ages is the starting age of the next loop to calculate the number of screenings. Screens records the adjusted screening ages and NumberOfScreens is the total number of screenings.

# Amend the screening schedule  
 if (Interval1 == 0){ # One-time screen in the lifetime  
 Screens <- StartAge  
 NumberOfScreens <- 1  
 } else {  
 Intervals <- 1 # Determine the number of intervals, and the default is one interval  
 if (!is.na(IntervalSwitchAge1)){Intervals <- 2}  
 if (!is.na(IntervalSwitchAge2)){Intervals <- 3}  
 if (!is.na(IntervalSwitchAge3)){Intervals <- 4}  
   
 IntervalVector <- c(Interval1, Interval2, Interval3, Interval4) # Create a vector of the screening intervals  
 ChangeAges <- sort(na.omit(c(StopAge, IntervalSwitchAge1, IntervalSwitchAge2, IntervalSwitchAge3))) # Create a vector of ages at which screening changes  
 Screens <- NULL # Create an empty vector to bind the sequence of screens  
 for (Interval in 1:Intervals ){ # Loops through the Intervals  
 StopAge <- ChangeAges[Interval] # Redefine the StopAge as the ChangeAge in this interval  
 NumberOfScreens <- (StopAge - StartAge) / IntervalVector[Interval] + 1 # Find the number of screens  
 NumberOfScreens <- ifelse(NumberOfScreens %% 1 == 0.5, ceiling(NumberOfScreens), round(NumberOfScreens)) # Rounding the number of screens  
 StopAge <- StartAge + (NumberOfScreens - 1) \* IntervalVector[Interval] # Redefine the actual StopAge following rounding  
 Screens <- c(Screens, seq(StartAge, StopAge, IntervalVector[Interval])) # Create the sequence of screens from the start and stop ages  
 StartAge <- max(Screens) # Update the final screen age as the Start Age for cases in which the interval changes  
 }  
 Screens <- unique(Screens) # Remove duplicates from the screen schedule  
 NumberOfScreens <- length(Screens) # Update the number of screens  
 }

36. Generate an array ScreenCounts to record the outcomes of screening. The first three columns record the background information of screening, including the screening number for tracking which round of screening is implemented, the applied screening modality, and the screening age. Before the test switch age, TestSwitchAge, the ScreenTest1 is the applied screening modality. After the test switch age, the screening modality, ScreenTest2, is used. The rest four columns are the number of individuals receiving the screening, the number of individuals in the preclinical state that can be diagnosed by the screening (although this is unknown in reality), the number of true positives, and the number of false positives.

# Create an array of the screen counts: This is an array recording the number of screens and consequent results  
 ScreenCounts <- array(NA, dim = c(NumberOfScreens, 7)) # Create an array from the screen schedule  
 colnames(ScreenCounts) <- c("ScreenNumber", "TestApplied", "ScreenAge", "N\_Screens", "RealPostives", "TruePositives", "FalsePositives") # Name the columns  
 ScreenCounts[, "ScreenNumber"] <- c(1:NumberOfScreens) # Write in a list equivalent to the number of screens and the per round screening age  
 ScreenCounts[, "ScreenAge"] <- Screens  
 ScreenCounts[, "TestApplied"] <- ScreenTest1 # Insert the appropriate test number  
 ScreenCounts[which(ScreenCounts[, "ScreenAge"] >= TestSwitchAge), "TestApplied"] <- ScreenTest2

37. Prepare for screening strategy simulation. Create an outcome table, ScreenedOutcomes, to record individuals’ disease history. Create two variables to record the clinical and cured cases’ IDs.

# Now run screening  
 ScreenedOutcomes <- Outcomes # Create the screen-adjusted outcomes from the natural history outcomes  
 ScreenedClinicalCases <- NULL # Create an empty vector to record clinical cases  
 ScreenedCuredCases <- NULL # Create an empty vector to record cured cases

38. Simulate the screening intervention. Create a loop for the number of screens. Within each loop, reset the sample seed to have a constant chance of being diagnosed for the same age across the screening programmes. In this example, we have set.seed refer to the table of random numbers of the screening age plus 2020. The user can modify 2020 to any number or delete it, as long as it is fixed across the screening strategies. The reason for choosing the screening age for set.seed is that we expect the screening implemented at a specific age will have the same performance across the screening programmes. Undiagnosed individuals have the same chance detected by the screening if the screening implemented at the same age. This is to minimise the statistical noise from the sampling. This issue can also be solved by increasing the simulation sample size.

After referring to the intended table of random numbers, produce two sets of sample numbers for all the individuals, separately for test sensitivity and test specificity. Retrieve relevant parameter values, including screening age, test sensitivity, and test specificity.

Identify individuals who are eligible for screening. First, these individuals must be alive at the moment of implementing screening. These individuals can be found by comparing their all-cause deaths and the screening age. Second, these individuals cannot be already diagnosed at the moment of implementing screening, so compare their clinical ages and the screening age. Notably, individuals who never develop the disease can be screened as well. For those who do not have clinical ages, their results of comparison to the screening age are NA. We modify these NA to TRUE, so these people can receive the screening. Only when individuals meet the first and the second conditions, they are qualified for receiving screening.

Recognise all the individuals can be screened for positives and negatives. Individuals who can be screened for positives, AllPositives, are the individuals qualified for screening and in the preclinical state. These individuals have entered the preclinical state at the moment of screening. Compare their ages of entering preclinical state and the screening age and make sure individuals who never enter the preclinical state do not meet these criteria. Individuals who can be screened for negatives, AllNegatives, are the individuals who are qualified for screening but not in the preclinical state.

Distinguish the true positives and false positives by comparing the sample numbers and the test sensitivity and specificity. Test sensitivity is the percentage of correctly identifying patients with the disease. Conversely, test specificity is the percentage of correctly identifying patients without the disease. Accordingly, we apply two different cohorts, AllPositives and AllNegatives, to separately find the true positives and false positives.

Record the results back to table ScreenCounts, including the number of individuals receiving the screening, the number of people who can be screened positive, the number of true positives, and the number of false positives.

Observe who can be cured by the treatment. We refer CureSeed as the random numbers which are produced when we simulate the no-screening scenario, so individuals have compatible results. This can avoid the situation that an individual fails the treatment in the screening programme and is alive in the no-screening scenario despite a higher treatment success chance with the screening. The impact of this issue can also be minimized by simulating a larger cohort.

Finally, update the individuals’ natural history with the screening intervention. For the true-positives, update their age of being diagnosed (entering the clinical state). For those cured by treatment, update their all-cause deaths with their other-cause deaths.

Record the clinical cases’ IDs to ScreenedClinicalCases and cured cases’ IDs to ScreenedCuredCases.

for (ScreenNumber in 1:nrow(ScreenCounts)){  
 set.seed(ScreenCounts[ScreenNumber, "ScreenAge"] + 2020) # Fixed sample seed for the same screening age across screening programmes  
 ScreenSnSeed <- runif(SampleSize) # Save random numbers for the test sensitivity  
 ScreenSpSeed <- runif(SampleSize) # Save random numbers for the test specificity  
 # Define the screen age  
 ScreenAge <- ScreenCounts[ScreenNumber, "ScreenAge"]  
 # Retrieve the test sensitivity and specificity  
 ScreenSn <- Input[paste("TestSensitivity", ScreenCounts[ScreenNumber, "TestApplied"], sep = ""), CurrentRun]  
 ScreenSp <- Input[paste("TestSpecificity", ScreenCounts[ScreenNumber, "TestApplied"], sep = ""), CurrentRun]  
   
 Alive <- ScreenedOutcomes[, "AllCauseDeath"] >= ScreenAge  
 NotDiagnosed <- ScreenedOutcomes[, "Clinical"] >= ScreenAge  
 NotDiagnosed[is.na(NotDiagnosed)] <- TRUE # People who never develop the disease are also not diagnosed  
 ScreenEligible <- Alive \* NotDiagnosed # Only when both conditions (alive and not diagnosed) meet  
   
 # Identify those in the preclinical stage at the screen age  
 Preclinical <- ScreenedOutcomes[, "Preclinical"] <= ScreenAge  
 Preclinical[is.na(Preclinical)] <- FALSE # People who never develop the disease are not in the preclinical stage  
 AllPositives <- which((Preclinical \* ScreenEligible) == 1) # Save patient numbers that can be screened for positive  
 # Identify the negatives as the complement of the positives from within the ScreenEligible set  
 AllNegatives <- which(ScreenEligible == 1)[!(which(ScreenEligible == 1) %in% AllPositives)]  
   
 # Find the true positives by sampling without replacement over all positives in proportion to the test sensitivity  
 TruePositives <- AllPositives[ScreenSnSeed[AllPositives] <= ScreenSn]  
 # Find the false positives by sampling without replacement over the negatives in proportion to the test specificity  
 FalsePositives <- AllNegatives[ScreenSpSeed[AllNegatives] >= ScreenSp]  
 # It's useful to have a per screening round count of the number of positives, all positives, true positives and false positives  
 ScreenCounts[ScreenNumber, c("N\_Screens", "RealPostives", "TruePositives", "FalsePositives")] <- cbind(length(which(ScreenEligible == 1)),  
 length(AllPositives),  
 length(TruePositives),  
 length(FalsePositives)  
 )  
 # Censor these successfully treated individuals  
 ScreenedCured <- TruePositives[which(CureSeed[TruePositives] <= Input["PreClinicalProbability", CurrentRun])]  
   
 # Now update the screen-adjusted outcomes  
 ScreenedOutcomes[TruePositives, "Clinical"] <- ScreenAge # Update the age of entering clinical state  
 ScreenedOutcomes[ScreenedCured, "AllCauseDeath"] <- ScreenedOutcomes[ScreenedCured, "OtherCauseDeath"] # Update all-cause death  
 # Record clinical cases  
 ScreenedClinicalCases <- c(ScreenedClinicalCases, TruePositives) # Save patient numbers that are diagnosed by screening  
 ScreenedCuredCases <- c(ScreenedCuredCases, ScreenedCured) # Save patient numbers that are cured by screening  
 }

39. Save the intermediate outcomes of screening performance, ScreenCounts, to the following folder.

# Record intermediate outcomes  
 write.table(ScreenCounts, paste("OutputFiles/LatestResults/IntermediateOutcomtes/ScreenCounts/", colnames(Input)[CurrentRun], "\_", strategies[Strategy], ".txt", sep = ""), row.names = F, col.names = T, sep = '\t')

40. Calculate the number of over-diagnosis, clinical cases, and cancer deaths for the intermediate outcomes. The over-diagnoses are the individuals dying from other-cause before they are diagnosed in their natural disease history, but unnecessarily diagnosed by the screening intervention. The clinical cases are the patients that are diagnosed in their disease history or by the screening programmes. NotScreenedClinicalCases are the patients diagnosed in their disease history but not the screening, despite the implementation of screening programmes. N\_CancerDeaths is the number of patients whose cause-specific death is equal to its all-cause death.

N\_Overdiagnosed <- sum(!ScreenedClinicalCases %in% ClinicalCases) # Patients are not diagnosed in their natural history but diagnosed with screening  
 N\_ClinicalCases <- length(ClinicalCases) + sum(!ScreenedClinicalCases %in% ClinicalCases) # The number of patients diagnosed in natural history and screening  
 NotScreenedClinicalCases <- ClinicalCases[!ClinicalCases %in% ScreenedClinicalCases] # Patient numbers who are diagnosed without screening  
 NotScreenedCuredCases <- NotScreenedClinicalCases[which(CureSeed[NotScreenedClinicalCases] <= Input["ClinicalProbability", CurrentRun])] # Receive treatment with clinical probability  
 N\_CancerDeaths <- length(which(ScreenedOutcomes[, "AllCauseDeath"] == ScreenedOutcomes[, "CauseSpecificDeath"])) # The number of cases that die at the age of disease death

41. Apply the self-defined function AgeDistribution to the Outcomes table which records individuals’ disease history. This function is defined in step 29. Then, save the results to the following folder.

# Record age distribution of each health state  
 IntermediateOutcomes\_AgeDist <- apply(ScreenedOutcomes[, !(colnames(ScreenedOutcomes) %in% "PersonNumber")], 2, AgeDistribution)  
 write.table(IntermediateOutcomes\_AgeDist, paste("OutputFiles/LatestResults/IntermediateOutcomtes/AgeDistribution/", colnames(Input)[CurrentRun], "\_", strategies[Strategy], ".txt", sep = ""), row.names = F, col.names = T, sep = '\t')

42. Prepare for calculating quality-of-life adjusted outcomes, similar to step 21 for the no-screening scenario. First, we save the discount and undiscounted life-years from the ScreenedOutcomes. Apply the PresentValue function defined in step 20 to discount the life-years.

# Assess effects  
 UD\_QALYS <- ScreenedOutcomes[, "AllCauseDeath"]  
 D\_QALYS <- D\_LYS <- sapply(UD\_QALYS, PresentValue) # Apply the discounting function that we defined above

43. Update the disease history of sick individuals, SickOutcome. We assume the quality-of-life (utility weight) only changes when individuals enter the clinical states in natural history. Early diagnosis does not influence the quality-of-life, while clinically having symptoms reduces the quality-of-life. The new SickOutcome has the natural disease history for all health states, except for all-cause deaths which will be replaced by the screening outcomes.

# Quality-of-life only changes when entering another health state in their natural history  
 SickOutcome <- cbind(SickOutcome[ , - length(StagesInOrder)], ScreenedOutcomes[Sick, "AllCauseDeath"]) # Update the age of death in the natural history of all the sick people

44. Apply the PresentValue function defined in step 20 to discount the ages of health states and apply the Accumulated\_QALYS function defined in step 22 to calculate the QALYs for the sick. For individuals who never develop the disease, adjust their QALYs with the utility weight of the healthy state.

Dis\_SickOutcome <- apply(SickOutcome, MARGIN = c(1, 2), PresentValue) # Apply the discounting function  
   
 UD\_QALYS[DiseaseFree] <- UD\_QALYS[DiseaseFree] \* Utility[1] # Quality-of-life adjusted but not discounted  
 UD\_QALYS[Sick] <- apply(SickOutcome, 1, Accumulated\_QALYS)  
 D\_QALYS[DiseaseFree] <- D\_QALYS[DiseaseFree] \* Utility[1] # Quality-of-life adjusted and discounted  
 D\_QALYS[Sick] <- apply(Dis\_SickOutcome, 1, Accumulated\_QALYS)

45. Calculate discounted and undiscounted QALYs loss due to screening and triage. Create a variable DiscountFactor\_Screen to record the discounting factors corresponding to the screening ages. These are applied to discount the QALYs loss from screening and triage. The total QALYs loss due to screening is the number of screenings multiply the dis-utility of one screening test. Notably, the test dis-utility might be different if the screening strategy changes the screening modality in the middle of the programme, so we refer to the modality specified in the ScreenCounts. The total triage service is the number of true-positives and false-positives multiply the dis-utility of one triage.

# Calculate utility loss due to screening test  
 DiscountFactor\_Screen <- DF\_Effects[ScreenCounts[, "ScreenAge"]]  
 UD\_ScreenHarm <- sum(ScreenCounts[,"N\_Screens"] \* Input[paste("TestDisutility", ScreenCounts[,"TestApplied"], sep=""), CurrentRun])  
 D\_ScreenHarm <- sum(ScreenCounts[,"N\_Screens"] \* Input[paste("TestDisutility", ScreenCounts[,"TestApplied"], sep=""), CurrentRun] \* DiscountFactor\_Screen) # Discounted  
 UD\_TriageHarm <- sum((ScreenCounts[, "TruePositives"] + ScreenCounts[, "FalsePositives"]) \* Input["DisutilityTriage", CurrentRun]) # All the test positives are triaged  
 D\_TriageHarm <- sum((ScreenCounts[, "TruePositives"] + ScreenCounts[, "FalsePositives"]) \* Input["DisutilityTriage", CurrentRun] \* DiscountFactor\_Screen)

46. Calculate discounted and undiscounted QALYs loss due to treatment. A perfect triage is assumed, so only the true-positives receive the treatments. The treatment is assumed to be carried out at the end of the year of diagnosis. For patients who are diagnosed in their natural history, round up their ages of being diagnosed in the ScreenedOutcomes and find the corresponding discounting factors. For patients who are diagnosed in the screening programmes, apply the corresponding discounting factor of screening ages, DiscountFactor\_Screen defined in step 45.

# Calculate utility loss due to treatment  
 ClinicalOnsetAges <- ceiling(ScreenedOutcomes[NotScreenedClinicalCases, "Clinical"]) # Identify the clinical ages and round up to integers for discounting  
 UD\_TreatmentHarm <- sum(ScreenCounts[,"TruePositives"] \* Input["DisutilityTrt", CurrentRun]) + # Diagnosed with screening  
 length(NotScreenedClinicalCases) \* Input["DisutilityTrt", CurrentRun] # Diagnosed without screening  
 D\_TreatmentHarm <- sum(ScreenCounts[,"TruePositives"] \* Input["DisutilityTrt", CurrentRun] \* DiscountFactor\_Screen) + # Receive treatment at the age of screening  
 sum(Input["DisutilityTrt", CurrentRun] \* DF\_Effects[ClinicalOnsetAges]) # Receive treatment when entering clinical state in natural history

47. Sum up the life-years and QALYs of each individual, including discounted and undiscounted. The calculation of QALYs for the screening intervention needs to deduct the dis-utility from screening, triage, and treatment.

# Calculate the discounted life-years without screening  
 UD\_Strategy\_LYG <- sum(ScreenedOutcomes[, "AllCauseDeath"])  
 D\_Strategy\_LYG <- sum(D\_LYS)  
 UD\_Strategy\_QALY <- sum(UD\_QALYS) - UD\_ScreenHarm - UD\_TriageHarm - UD\_TreatmentHarm  
 D\_Strategy\_QALY <- sum(D\_QALYS) - D\_ScreenHarm - D\_TriageHarm - D\_TreatmentHarm

48. Estimate the discounted and undiscounted treatment costs of patients diagnosed clinically. In the screening programmes, some patients are diagnosed with screening and some patients are diagnosed with symptoms when they enter the clinical state in their natural disease history. In this step, we calculate the treatment costs for patients clinically presented by the number of NotScreenedClinicalCases multiply the unit cost of the treatment for the systematic disease. For discounted late treatment costs, apply the discounting factors that are based on the discounting rate of costs and correspond to the ages entering clinical state. Also, update the variable DiscountFactor\_Screen with the discounting rate of costs.

# Assess costs  
 # Count the late treatment costs  
 DiscountFactor\_Screen <- DF\_Costs[ScreenCounts[, "ScreenAge"]]  
 UD\_LateTreatmentCosts <- length(NotScreenedClinicalCases) \* LateTreatmentCost  
 D\_LateTreatmentCosts <- sum(DF\_Costs[ClinicalOnsetAges] \* LateTreatmentCost) # Find the discounted treatment costs

49. Estimate the undiscounted costs of screening, triage, and treatment for the screening-detected. For undiscounted screening costs, add up the number of individuals receiving the screening and times to the unit screening cost. The unit triage cost applies to all the true-positives and the false-positives. The treatement costs for the screening-detected is only for those are true-postives.

# First calculate the undiscounted amounts: the costs of primary screening, triage and treating preclinical disease under screening  
 UD\_ScreenCosts <- sum(ScreenCounts[, "N\_Screens"] \* ScreenCost) # The primary test cost applies to all screens  
 UD\_TriageCosts <- sum((ScreenCounts[, "TruePositives"] + ScreenCounts[, "FalsePositives"]) \* TriageCost) # The cost of triage applies to all screen true and false positives  
 UD\_EarlyTreatmentCosts <- sum(ScreenCounts[, "TruePositives"] \* EarlyTreatmentCost) # The early treatment costs applies to all true positives (assuming a 100% specific triage test)

50. Estimate the discounted costs of screening, triage, and treatment for the screening-detected. Create an array ScreenCostArray to save the screening ages, discounted screening costs, discounted triage costs, and discounted treatment costs. Use the DiscountFactor\_Screen defined in step 48 to discount the cost estimates for the screen-detected.

# Now calculate the discounted values  
 ScreenCostArray <- array(NA, dim = c(NumberOfScreens, 4)) # Create a blank array for the discounted costs which can also be saved as intermediate outcomes  
 colnames(ScreenCostArray) <- c("Age", "D\_Screen", "D\_Triage", "D\_EarlyTreatment") # Name the columns in this array  
 ScreenCostArray[, "Age"] <- ScreenCounts[, "ScreenAge"] # Apply the appropriate discount factors according to the screen ages  
 ScreenCostArray[, "D\_Screen"] <- DiscountFactor\_Screen \* ScreenCounts[, "N\_Screens"] \* ScreenCost # Multiply the screen, triage and early treatment numbers by the discount factor  
 ScreenCostArray[, "D\_Triage"] <- DiscountFactor\_Screen \* (ScreenCounts[, "TruePositives"] + ScreenCounts[, "FalsePositives"]) \* TriageCost  
 ScreenCostArray[, "D\_EarlyTreatment"] <- DiscountFactor\_Screen \* ScreenCounts[, "TruePositives"] \* EarlyTreatmentCost

51. Calculate the undiscounted and discounted total costs. Sum up the costs of screening, triage, and treatment for both symptomatic disease and screening-detected.

# Sum up the undiscounted and discounted costs  
 UD\_TotalCosts <- UD\_ScreenCosts + UD\_TriageCosts + UD\_EarlyTreatmentCosts + UD\_LateTreatmentCosts  
 D\_TotalCosts <- sum(ScreenCostArray[, c("D\_Screen", "D\_Triage", "D\_EarlyTreatment")]) + D\_LateTreatmentCosts

52. Update the IntermediateOutcomes with the results for this specific screening strategy.

# Record intermediate outcomes and the first column is the name of the strategy  
 IntermediateOutcomes[Strategy + 1, - 1] <- c(UD\_ScreenCosts = UD\_ScreenCosts,   
 UD\_TriageCosts = UD\_TriageCosts,   
 UD\_EarlyTreatmentCosts = UD\_EarlyTreatmentCosts,   
 UD\_LateTreatmentCosts = UD\_LateTreatmentCosts,   
 D\_ScreenCosts = sum(ScreenCostArray[, "D\_Screen"]),   
 D\_TriageCosts = sum(ScreenCostArray[, "D\_Triage"]),   
 D\_EarlyTreatmentCosts = sum(ScreenCostArray[, "D\_EarlyTreatment"]),   
 D\_LateTreatmentCosts = D\_LateTreatmentCosts,   
 N\_CancerDeaths,   
 N\_Overdiagnosed,   
 N\_ClinicalCases   
 )

53. Update the CostEffectivenessResults with the simulation outcomes for the specific screening strategy. End the loop of the simulation if it is the last screening strategy.

# Sum up the undiscounted and discounted costs  
 CostEffectivenessResults[Strategy + 1, c("UD\_EffectsL", "UD\_EffectsQ", "UD\_Costs", "D\_EffectsL", "D\_EffectsQ", "D\_Costs")] <-c(UD\_Strategy\_LYG,   
 UD\_Strategy\_QALY,   
 UD\_TotalCosts,   
 D\_Strategy\_LYG,   
 D\_Strategy\_QALY,   
 D\_TotalCosts   
 )  
 } # Close the loop over all the strategies

54. Create a self-defined function SDfinder to search for the simple dominance among the screening strategies. Strategies are simple dominated (SD) if they, compared to any other strategy, have lower effectiveness but higher costs or higher costs but lower effectiveness. Strategies lablled with NA mean they are not SD.

# Mark those subject to simple dominance as SD  
 SDfinder <- function(x, output){ # This simply compares each strategy to all others to find cases of simple dominance  
 if (sum((as.numeric(x[1]) <= AllEffects) \* (as.numeric(x[2]) > AllCosts)) >= 1){  
 return("SD")  
 }  
 if (sum((as.numeric(x[1]) < AllEffects) \* (as.numeric(x[2]) >= AllCosts)) >= 1){  
 return("SD")  
 }  
 return("NA")  
 }

55. Employ the function SDfinder defined in step 54 to identify the strategies that are SD with the discounted and undiscounted results, including LYs and QALYs.

AllCosts <- as.numeric(CostEffectivenessResults[, "D\_Costs"])  
 AllEffects <- as.numeric(CostEffectivenessResults[, "D\_EffectsL"])  
 CostEffectivenessResults[, "D\_ICER\_L"] <- apply(CostEffectivenessResults[, c('D\_EffectsL', 'D\_Costs')], 1, SDfinder) # Discounted LYs gained  
 AllEffects <- as.numeric(CostEffectivenessResults[, "D\_EffectsQ"])  
 CostEffectivenessResults[, "D\_ICER\_Q"] <- apply(CostEffectivenessResults[, c('D\_EffectsQ', 'D\_Costs')], 1, SDfinder) # Discounted QALYs gained  
 AllCosts <- as.numeric(CostEffectivenessResults[, "UD\_Costs"])  
 AllEffects <- as.numeric(CostEffectivenessResults[, "UD\_EffectsL"])  
 CostEffectivenessResults[, "UD\_ICER\_L"] <- apply(CostEffectivenessResults[, c('UD\_EffectsL', 'UD\_Costs')], 1, SDfinder) # Undiscounted LYs gained  
 AllEffects <- as.numeric(CostEffectivenessResults[, "UD\_EffectsQ"])  
 CostEffectivenessResults[, "UD\_ICER\_Q"] <- apply(CostEffectivenessResults[, c('UD\_EffectsQ', 'UD\_Costs')], 1, SDfinder) # Undiscounted QALYs gained

56. Create a self-defined function EDfinder to identify the strategies which are extended dominated (ED). First, create a BoundarySet that excludes the SD strategies. In the case of BoundarySet only have a single strategy or all the strategies with the same cost and effecitveness, only one cost-effectiveness estimate on the efficient frontier. This cost-effectiveness estimate should be the reference case without numeric ICER.

In other circumstances, order the strategies from the least effective to the highest. The least effective strategy is the reference, the first strategy on the efficient frontier. If there is only one strategy or if all the strategies have the same effectiveness and costs, only one reference case lies on the efficient frontier without any ICER estimations. If there is more than one strategy having different outcomes, calculate the cost-effectiveness ratio compared to the reference case for all the strategies having higher costs and effectiveness than the reference case. Find the strategy with the lowest ratio as the next strategy on the efficient frontier. Compared to this strategy, the strategies having lower effectiveness are extended dominated (ED). Then, exclude these ED strategies and recalculate the cost-effectiveness ratio compared to the new strategy identified on the efficient frontier. This loop continues until the strategy with the largest effectiveness is identified. This function returns BoundarySet, the strategies are ED or on the efficient frontier.

# Find cost-effective screening strategies on the efficient frontier  
 EDfinder <- function(CEresults, Effects, Costs, ICER, output){  
 # Define the boundary set as the non strictly-dominated strategies  
 BoundarySet <- CEresults[setdiff(seq(1:nrow(CEresults)), which(CEresults[, ICER] == "SD")),]  
 # Order the boundary set in terms of effectiveness  
 BoundarySet <- BoundarySet[order(BoundarySet[, Effects]),]  
 # The first efficient strategy is the strategy with the least effectiveness  
 EfficientSet <- 1  
   
 if (nrow(BoundarySet) == 1 | (length(unique(BoundarySet[, Effects])) == 1 & length(unique(BoundarySet[, Costs])) == 1)){ # Only one strategy or strategies with the same outcomes  
 BoundarySet[, ICER] <- "reference" # This strategy is the only strategy on the efficient frontier, so without ICER  
 } else {  
 # Set the ICERs as numeric  
 BoundarySet[, ICER] <- as.numeric(BoundarySet[, ICER])  
 # Put a the break condition at the beginning that stops the routine if the end of the set is reached  
 if (max(EfficientSet) < nrow(BoundarySet)){  
 repeat{  
 # Update the ICER calculation relative to the new reference  
 for (i in ((max(EfficientSet) + 1): nrow(BoundarySet))){  
 # Calculate the column of ACERs relative to the least-effective non simply dominated strategy  
 BoundarySet[i, ICER] <- round((BoundarySet[i, Costs] - BoundarySet[max(EfficientSet), Costs])/  
 (BoundarySet[i, Effects] - BoundarySet[max(EfficientSet), Effects]), 3)  
 }  
 # Find the lowest ICER  
 j <- max(EfficientSet) + which(BoundarySet[(max(EfficientSet) + 1):nrow(BoundarySet), ICER] ==   
 min(BoundarySet[(max(EfficientSet) + 1):nrow(BoundarySet), ICER], na.rm=TRUE))  
 # Update what forms the efficient set  
 EfficientSet <- c(EfficientSet, j)  
   
 if (max(EfficientSet) == nrow(BoundarySet)){break}  
 }  
 }  
 BoundarySet[-EfficientSet, ICER] <- "ED"  
 # Set the ICER of the least effective strategy in the efficient set to "reference"  
 BoundarySet[1, ICER] <- "reference"  
 }  
 return(BoundarySet)  
 }

57. Employ the function EDfinder defined in step 56 to identify the strategies that are ED with the discounted and undiscounted results, including LYs and QALYs.

BoundarySet\_L <- EDfinder(CostEffectivenessResults, "D\_EffectsL", "D\_Costs", "D\_ICER\_L")  
 BoundarySet\_Q <- EDfinder(CostEffectivenessResults, "D\_EffectsQ", "D\_Costs", "D\_ICER\_Q")  
 BoundarySet\_UL <- EDfinder(CostEffectivenessResults, "UD\_EffectsL", "UD\_Costs", "UD\_ICER\_L")  
 BoundarySet\_UQ <- EDfinder(CostEffectivenessResults, "UD\_EffectsQ", "UD\_Costs", "UD\_ICER\_Q")

58. Write the calculated ICERs and ED to the main outcome table CostEffectivenessResults.

# Write in the ICERs of the efficient strategies  
 CostEffectivenessResults[match(BoundarySet\_L[, "StrategyName"], CostEffectivenessResults[, "StrategyName"]), "D\_ICER\_L"] <- BoundarySet\_L[, "D\_ICER\_L"]  
 CostEffectivenessResults[match(BoundarySet\_Q[, "StrategyName"], CostEffectivenessResults[, "StrategyName"]), "D\_ICER\_Q"] <- BoundarySet\_Q[, "D\_ICER\_Q"]  
 CostEffectivenessResults[match(BoundarySet\_UL[, "StrategyName"], CostEffectivenessResults[, "StrategyName"]), "UD\_ICER\_L"] <- BoundarySet\_UL[, "UD\_ICER\_L"]  
 CostEffectivenessResults[match(BoundarySet\_UQ[, "StrategyName"], CostEffectivenessResults[, "StrategyName"]), "UD\_ICER\_Q"] <- BoundarySet\_UQ[, "UD\_ICER\_Q"]

59. Update the tables BoundarySet\_L, BoundarySet\_Q, BoundarySet\_UL, and BoundarySet\_UQ to record the boundary sets for discounted LYs, discounted QALYs, undiscounted LYs, and undiscounted QALYs. For each table, exclude ED strategies and only leave strategies that are on the efficient frontier.

BoundarySet\_L <- BoundarySet\_L[which(!BoundarySet\_L[, "D\_ICER\_L"] %in% "ED"), ] # Update the BoundarySet by removing strategies with "ED"  
 BoundarySet\_Q <- BoundarySet\_Q[which(!BoundarySet\_Q[, "D\_ICER\_Q"] %in% "ED"), ]  
 BoundarySet\_UL <- BoundarySet\_UL[which(!BoundarySet\_UL[, "UD\_ICER\_L"] %in% "ED"), ]  
 BoundarySet\_UQ <- BoundarySet\_UQ[which(!BoundarySet\_UQ[, "UD\_ICER\_Q"] %in% "ED"), ]

60. Save the main results, CostEffectivenessResults, and intermediate outcomes, IntermediateOutcomes, for all the screening strategies to the assigned folder. Also, the strategies on the efficient frontier are also saved in independent files for different outcomes, discounted or undiscounted, QALYs or LYs.

# Return the results for all values and the efficient set within the R session  
 # Save the overall results and efficient frontier results  
 write.table(IntermediateOutcomes, paste("OutputFiles/LatestResults/IntermediateOutcomtes/CEbreakdown/", colnames(Input)[CurrentRun], ".txt", sep = ""),   
 row.names = F, col.names = T, sep = '\t')  
 write.table(CostEffectivenessResults, paste("OutputFiles/LatestResults/MainCEresults/OutputTable\_", colnames(Input)[CurrentRun], ".txt", sep=""),   
 row.names = F, col.names = T, sep = '\t')  
 write.table(BoundarySet\_L, paste("OutputFiles/LatestResults/MainCEresults/BoundarySet/BoundarySet\_L\_", colnames(Input)[CurrentRun], ".txt", sep=""),   
 row.names = F, col.names = T, sep = '\t')  
 write.table(BoundarySet\_Q, paste("OutputFiles/LatestResults/MainCEresults/BoundarySet/BoundarySet\_Q\_", colnames(Input)[CurrentRun], ".txt", sep=""),   
 row.names = F, col.names = T, sep = '\t')  
 write.table(BoundarySet\_UL, paste("OutputFiles/LatestResults/MainCEresults/BoundarySet/BoundarySet\_UL\_", colnames(Input)[CurrentRun], ".txt", sep=""),   
 row.names = F, col.names = T, sep = '\t')  
 write.table(BoundarySet\_UQ, paste("OutputFiles/LatestResults/MainCEresults/BoundarySet/BoundarySet\_UQ\_", colnames(Input)[CurrentRun], ".txt", sep=""),   
 row.names = F, col.names = T, sep = '\t')

61. Plot the cost-effectiveness plane only when simulating the base-case scenario and the user intends to have the cost-effectiveness plane.

# Plot cost-effectiveness plane   
 if ((colnames(Input)[CurrentRun] == "Input") & (CEplane == TRUE)){ # Only plot when it is the base-case

62. Plot the cost-effectiveness plane with the user-defined outcomes, QALYs or LYs.

# Define the outcome measure  
 if (MeasureQALYs == TRUE){  
 OutcomeMeasure <- "D\_EffectsQ"  
 ICERMeasure <- "D\_ICER\_Q"  
 BoundarySet <- BoundarySet\_Q  
 } else {  
 OutcomeMeasure <- "D\_EffectsL"  
 ICERMeasure <- "D\_ICER\_L"  
 BoundarySet <- BoundarySet\_L  
 }

63. Have all the results reference back to the no-screening strategy which is the first row of the outcome tables, including CostEffectivenessResults and BoundarySet. The no-screening will be the origin of the cost-effectiveness plane. Also, having the average effectiveness and costs per capita is easier to compare the results when simulating different sample sizes.

# Define the costs and effects, reference to the no-screening  
 Effects <- (CostEffectivenessResults[, OutcomeMeasure] - CostEffectivenessResults[1, OutcomeMeasure]) / SampleSize  
 Costs <- (CostEffectivenessResults[, "D\_Costs"] - CostEffectivenessResults[1, "D\_Costs"]) / SampleSize  
 BoundarySet$EfficientEffects <- (BoundarySet[, OutcomeMeasure] - BoundarySet[1, OutcomeMeasure]) / SampleSize  
 BoundarySet$EfficientCosts <- (BoundarySet[, "D\_Costs"] - BoundarySet[1, "D\_Costs"]) / SampleSize

64. Identify the optimal strategy. If the minimum of the available ICERs is below the decision threshold, the optimal strategy is the one with the maximum ICER just lower than the threshold. If all the strategies’ ICERs are above the decision threshold, no ICER is available and the reference is the optimal strategy.

# Identify the highest of the cost-effective ICERs given the threshold  
 ICERs <- as.numeric(BoundarySet[, ICERMeasure]) # All the ICERs  
 if (min(ICERs, na.rm = TRUE) <= Threshold){  
 OptimalStrategy <- rownames(BoundarySet)[which(ICERs == (max(ICERs[ICERs <= Threshold], na.rm=TRUE)))] # Find the maximum ICER lower than the threshold  
 } else {  
 OptimalStrategy <- rownames(BoundarySet)[BoundarySet[, ICERMeasure] %in% "reference"] # All the ICERs are higher than the threshold, so the reference is the optimal strategy  
 }

65. Look up the cost and effectiveness estimates and corresponding ICER of the optimal strategy.

CostEffectiveICER <- BoundarySet[OptimalStrategy, ICERMeasure] # Find the corresponding ICER  
 CostEffectiveEffects <- as.numeric(BoundarySet[OptimalStrategy, "EfficientEffects"]) # Find the corresponding costs and effect  
 CostEffectiveCosts <- as.numeric(BoundarySet[OptimalStrategy, "EfficientCosts"])

66. Set up the tick marks for axes and save their ranges. The setup of TicksEffects and TicksCosts may differ from case to case.

# Set the tick marks: after plotting several times, these numbers are chosen to best present the axis in our example  
 TicksEffects <- pretty(c(round(min(Effects) - (max(Effects) - min(Effects)) / 20, 5), round(max(Effects) + (max(Effects) - min(Effects)) / 20, 5)))  
 TicksCosts <- pretty(c(0, max(Costs) \* 1.02))  
 # Set the ranges for both values  
 xRange <- range(TicksEffects)  
 yRange <- range(TicksCosts)

67. Generate the ICER label and axis label for the measurement of effectiveness.

# Create ICER and axis labels  
 ICERLabels <- gsub("NA", "", noquote(format(BoundarySet[, ICERMeasure], big.mark = "")))  
 if (MeasureQALYs == TRUE){LabLabel <- "Effects (QALY, per capita)"} else {LabLabel <- "Effects (LYG, per capita)"}

68. Create a plot window of a particular size.

# Set the plot dimension  
 windows.options(width = 12, height = 9)

69. Set the margin sizes of the plot in the following order: bottom, left, top, and right.

# Set the margin parameters  
 par(mar = c(5, 6, 1, 1))

70. Plot the cost-effectiveness plane which effectiveness is on the X axis. The user-defined parameter HealthOnXAxis controls which axis effectiveness lies on. If the effectiveness is not on the X axis, step 71 produces the plot with the effectiveness on the opposite axis.

Create a plot with Effects on the X axis, and Costs on the Y axis. Remove X and Y axes tick labels by setting xaxt and yaxt to "n". Delete the X and Y axes labels by modifying xlab and ylab to blank strings. Change axes limits xlim and ylim to xRange and yRange that defined as step 66. Choose the symbol of the points by setting pch to any number between 1 and 25, and the default is the filled circle. Adjust the symbol size with cex and the symbol colour with col.

Place the axes with the function axis. The first number is the position of the axis. Argument 1 refers to the X axis and 2 refers to the Y axis. Indicate the tickmarks with at and rotate the axis labels horizontally with las.

Add the titles of the X and Y axes with xlab and ylab. Argument mgp controls the placing of the axis title, axis labels, and axis line. Argument cex.lab modifies the size of the axis label.

Add a line and points with boundary data. Use lwd to control the width and pch to choose the symbol of the points. Add the point of the optimal strategy if the user wants to show the optimal strategy on the plot. Apply cex to control the size of symbols. Use col to manage the colour of the symbols.

Place the labels of ICERs, ICERLabels, to the right of the symbols. First, create a LabelYOffset for specifying the space we want to leave in between. Then, add texts with the specified position. Argument pos = 4 means plot the texts to the right. Also, the text size is controlled by cex.

if (HealthOnXAxis == TRUE){  
 # Plot the plane  
 plot(Effects, Costs, yaxt = "n", xaxt = "n", xlab = "", ylab = "", xlim = xRange, ylim = yRange, pch = 16, cex = 0.5, col = "grey")  
 # Add the axes  
 axis(1, at = TicksEffects, las = 1)  
 axis(2, at = TicksCosts, las = 1)  
 # Add in the X and Y axis labels with spacing  
 title(ylab = "Costs ($, per capita)", xlab = LabLabel, mgp = c(3.75,1.75,0), cex.lab = 1.2)  
 # Add the efficient frontier  
 lines(BoundarySet$EfficientEffects, BoundarySet$EfficientCosts, lwd = 2)  
 points(BoundarySet$EfficientEffects, BoundarySet$EfficientCosts, pch = 19, lwd = 2)  
 # Add the optimal strategy  
 if (ShowOptimalStrategy == TRUE){points(CostEffectiveEffects, CostEffectiveCosts, pch = 19, lwd = 3, cex = 1, col = "forestgreen")}  
 # Add ICERs to the frontier  
 LabelYOffset <- max(BoundarySet$EfficientCosts) / 10 # Generate a label offset to place the values at an offset  
 if (ShowICERs == TRUE){text(BoundarySet$EfficientEffects, BoundarySet$EfficientCosts - LabelYOffset, labels = ICERLabels, pos = 4, cex = 0.6)} # Apply the text  
 } else {

71. Plot the cost-effectiveness plane which effectiveness is on the Y axis. Similarly to step 70, the only difference is to change all the commands relevant to the axes.

Create a plot with Costs on the X axis, Effects on the Y axis. Change axes limits xlim and ylim to yRange and xRange. Modify the first number of the axis function to have 1 refer to cost and 2 refer to effectiveness. Adjust the titles of the X and Y axes with xlab and ylab to have correct labels for costs and effectiveness. All the functions regarding lines, points, and texts have the cost data first specified for plotting on the X axis, followed by the effectiveness for the Y axis. In our example, we create two variables, LabelYOffset and LabelXOffset, to specify the space between the symbols and the texts. These two variables are flexible and adjustable for best plotting the texts.

More details of arguments can be found in step 70.

plot(Costs, Effects, yaxt = "n", xaxt = "n", xlab = "", ylab = "", xlim = yRange, ylim = xRange, pch = 16, cex = 0.5, col = "grey")  
 # Add the axes  
 axis(2, at = TicksEffects, las = 1)  
 axis(1, at = TicksCosts, las = 1)  
 # Add in the X and Y axis labels with spacing  
 title(xlab = "Costs ($, per capita)", ylab = LabLabel, mgp = c(3.75,1.75,0), cex.lab = 1.2)  
 # Add the efficient frontier  
 lines(BoundarySet$EfficientCosts, BoundarySet$EfficientEffects, lwd = 2)  
 points(BoundarySet$EfficientCosts, BoundarySet$EfficientEffects, pch = 19, lwd = 2)  
 # Add the optimal strategy  
 if (ShowOptimalStrategy == TRUE){points(CostEffectiveCosts, CostEffectiveEffects, pch = 19, lwd = 3, cex = 1, col = "forestgreen")}  
 # Add ICERs to the frontier  
 LabelYOffset <- max(BoundarySet$EfficientEffects) / 18 # Generate a label offset to place the values at an offset  
 LabelXOffset <- max(BoundarySet$EfficientCosts) / 40  
 if (ShowICERs == TRUE){text(BoundarySet$EfficientCosts + LabelXOffset, BoundarySet$EfficientEffects + LabelYOffset, labels = ICERLabels, pos = 2,cex = 0.6)} # Apply the text  
 }

72. Save the plot into the file format of .jpeg as default. The width and the height of the diagram can be adjusted. Have higher resolution, res, to ensure the clarity of the diagram.

Save the plot into .pdf or .png if needed.

# Save the graph as a .jpeg as a matter of course  
 SaveCEplane <- recordPlot()  
 jpeg("OutputFiles/Figures/Figure.jpeg", width = 4000, height = 2400, res = 300)  
 replayPlot(SaveCEplane)  
 dev.off()  
   
 # Save the graph as a PDF or png if that option is ticked  
 if (SaveGraphPDF == TRUE){  
 pdf("OutputFiles/Figures/Figure.pdf", width = 15, height = 9)  
 replayPlot(SaveCEplane)  
 dev.off()  
 }  
 if (SaveGraphPNG == TRUE){  
 png("OutputFiles/Figures/Figure.png", width = 4000, height = 2400, res = 300)  
 replayPlot(SaveCEplane)  
 dev.off()  
 }  
 }  
}

73. Print the simulation time by using the current time minus the variable time we saved in the first step of this simulation.

Sys.time() - time # The execution time