**Title: Building Efficient Microsimulation Models in R: A How-To Guide**

Aaron N Winn1, Abdullah I Abdelaziz1, Wael Mohamed2,3, Jyotirmoy Sarker1, Robert Smith2,3

1. Retzky College of Pharmacy, University of Illinois Chicago, Chicago, Illinois, USA

2. Sheffield University

3. Dark Peak Analytics

**Abstract**

Background: Microsimulation models are needed in pharmacoeconomics for simulating individual patient trajectories when there is substantial individual-level variability, history dependence, or process complexity, which a Markov framework cannot adequately capture. Given the complexity of the disease process these models are trying to capture, building these models efficiently is crucial. However, guidance on creating computationally efficient and reproducible microsimulation models in R is limited. This manuscript provides a comprehensive framework for developing efficient microsimulation models.

Methods: We demonstrate the development of an R-based efficient microsimulation model for type 2 diabetes using the United Kingdom Prospective Diabetes Outcomes Model (UKPDS) equations. The model simulates disease progression, associated healthcare costs, and QALYs for hypothetical cohorts. Key strategies for efficiency include leveraging vectorized operations, modular programming, parallelization, and RCPP (how to write C++ code in R) for computationally intensive functions. The approach balances flexibility with scalability, ensuring the model is adaptable to various diseases and outcomes. We measure the improvements in computational time for each improvement in the model. Moreover, we compare the computational times to an Excel-based version of the model.

Results: The R-based microsimulation model demonstrated significant improvements in computational efficiency compared to its Excel-based counterpart, with runtime reduced by over 50% and gains increasing as the cohort size increased. Modular functions written in RCPP improved execution speed while maintaining readability. Additional features, such as probabilistic analysis and visual display of the outputs, were integrated seamlessly within the R framework.

Conclusions: This study provides a step-by-step guide for building efficient microsimulation models in R for health economics. By adopting best practices in programming and computational optimization, health economists can create robust, scalable, reproducible, and transparent models. The framework can serve as a foundation for future work, advancing the use of R for complex health economic analyses.

**Introduction**

Decision-analytic models offer a structured approach to integrating clinical and economic evidence in a systematic manner and is increasingly used to inform pricing and policy decisions [1]. These models support a range of applications, including projecting long-term outcomes from clinical trials, linking intermediate endpoints to final health outcomes, conducting indirect treatment comparisons, and guiding early-stage decisions in health technology development [1]. Health decision modeling encompasses a variety of methodological approaches, including decision trees, Markov models, discrete event simulations, and hybrid models [2]. Among these, state-transition cohort models, particularly Markov models, are widely used in health economics and decision science [3, 4]. Markov models conceptualize disease progression as a series of transitions between discrete health states, each governed by predefined transition probabilities [3]. These models enable long-term projections of disease burden and support cost-effectiveness analyses under different policies or intervention scenarios [4].

Despite their widespread use, deterministic cohort models have important limitations. These models assume population homogeneity by applying the same transition probabilities to all individuals, thereby overlooking patient-level differences [5]. More critically, they rely on the Markov assumption, which stipulates that future transitions depend only on the individual's current health state, not on their prior history [6, 7]. This assumption restricts the model’s ability to capture path-dependent trajectories, declining treatment effects, and interactions between comorbidities, all of which are crucial for accurately modeling real-world disease progression, treatment effectiveness, and associated costs. Adding health states helps address this issue but increases the complexity and computational demands of the Markov model and can quickly result in “state explosion”.

To overcome these limitations, individual-based state-transition models, also known as microsimulation models, have been developed [6]. These simulate individual patient pathways over time, allowing for heterogeneity in baseline characteristics, history-dependent transitions, and stochastic variation in outcomes [8]. By operating at the individual level, microsimulation models offer a more granular and realistic representation of disease dynamics. They are particularly well-suited for modeling chronic diseases, oncology, and personalized medicine contexts, where prior health states can meaningfully influence future outcomes [8].

The advantages of microsimulation come at a computational cost. Since these models simulate large numbers of individual trajectories over long time horizons, they impose heavy demands on memory and processing time. Efficient implementation strategies are therefore critical. Techniques such as vectorization, parallel computing, and memory-efficient data structures are commonly used to manage these challenges and improve performance.

R has emerged as a widely adopted programming environment for health decision modeling due to its open-source nature, rich ecosystem of statistical libraries, and powerful simulation capabilities [4, 9]. It facilitates transparent and reproducible modeling workflows and supports integration of statistical analysis with simulation. Additionally, vectorization in R can enhance performance by reducing reliance on slow iterative loops and enabling faster execution of large-scale simulations.

However, despite its increasing use, educational resources on implementing microsimulation models in R remain limited. The aim of this tutorial is to develop an efficient and scalable R-based microsimulation model for type 2 diabetes, using techniques such as vectorization, modular programming, parallelization, and RCPP integration to improve computational performance. It also aims to evaluate runtime improvements and demonstrate additional features like probabilistic analysis and output visualization.

**Methods**

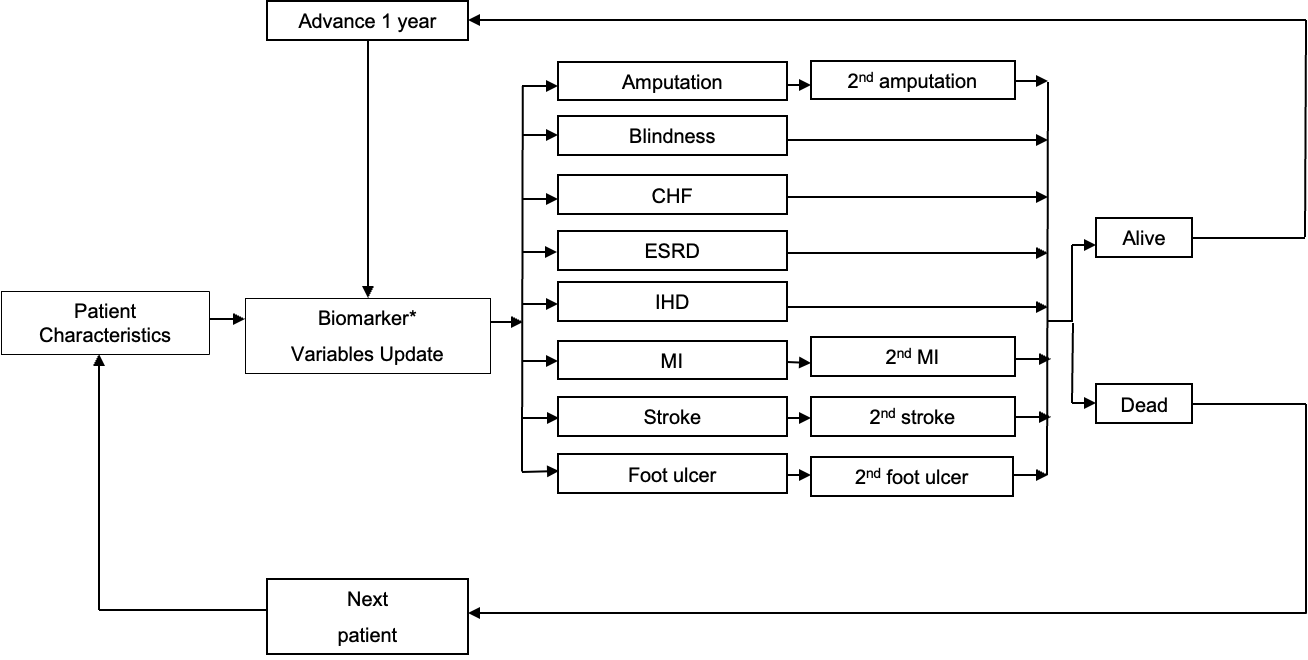
UKPDS Model Example

The United Kingdom Prospective Diabetes Outcomes Model (UKPDS) is a widely used diabetes simulation model that is based on a trial of over 5,000 patients. Patients have been followed for over 30 years. Using this data, researchers have developed a system of equations that project patients' biomarkers and events over time. These equations form the foundation of our microsimulation model.

The model tracks changes in patient biomarkers and the occurrence of diabetes-related complications over time. Event equations use parametric survival models, while biomarker dynamics are modeled using linear regressions[[1]](#footnote-1). The simulation updates each individual's state annually based on these models.

This tutorial demonstrates how to implement the UKPDS-based microsimulation in R, beginning with data structure setup and progressing through event simulation, cost and utility calculation, and efficiency enhancements.

Figure 1. The Structure of the UKPDS model



**One at Person at a Time Model, A data frame approach (2-dimensional array)**

We will now build the microsimulation model in R that simulates a patient's trajectory of biomarkers and events over a 20-year time horizon. The core of this model is matrix multiplication. To facilitate efficient, easy-to-read and write code, we create a matrix of coefficients from the UKPDS publications and a matrix of patients with consistent dimensions to facilitate matrix multiplication. To generate patients’ sojourn overtime, we can then simply multiply these two matrices together repeatedly for each biomarker and event[[2]](#footnote-2). The model is itself a series of functions that both call each other (nest) and are run sequentially. Each of these functions is a self-contained unit of code that performs a specific task. Generally, creating functions and calling them is a good practice in programming as it makes the code easier to read, understand and debug. The steps of this process are as follows:

1) Create a function to simulate biomarkers called biomarker,

2) Create a function to that uses the biomarker function to update all biomarkers called update\_all\_biomarkers which runs through all the biomarkers and updates any biomarker related variables,

3) Create event functions which use a weibull model called weibull\_event and when using a logistic model called logistic\_event,

4) Create a function to that uses the weibull\_event and logistic\_event functions to update all events called update\_health\_events which runs through all the events and updates any event related variables,

5) Create mortality functions which use a gompertz model called gompertz\_event,

6) Create a mortality function which uses the gompertz\_event and logistic\_event functions to predict if patients die within a cycle,

7) function to feed patients through the model one at a time,

8) Define the discount rate, costs and quality of life weights and create a matrix that can track these across patients,

9) Create a matrix of patient equations,

10) Create a patient dataset,

11) Now simulate patients using the functions we have created, assign costs and QALYs, and save the patients’ overall results.

Using the UKPDS Equations

We construct a coefficient array to support efficient matrix multiplication for predicting biomarkers and health events, using parameters from the UKPDS publications. To begin, we organize the coefficients in a spreadsheet where each column represents an equation, and each row represents a parameter. Missing parameter values are replaced with zeros to ensure compatibility with matrix operations. This spreadsheet is then imported into R. We extract the parameter names (rows) and equation names (columns), and optionally allow for multiple bootstrap replicates, though this tutorial uses only the average estimates. The resulting 3D array has parameters along one axis, equations along another, and bootstrap replicates as slices. Finally, we split the array into two components: one for individual-level traits and another for non-individual traits, including regression intercepts (lambda), shape parameters for time-to-event models (rho), and death-related parameters.

# # Step 1: Import the matrix of coefficients ####

# Read the coefficient matrix from a CSV or RData file

df\_UKPDS\_coef <- readr::read\_csv("data/ukpds\_coef.csv") # Load coefficient matrix from CSV

# Replace NAs with 0s to avoid missing values in calculations

df\_UKPDS\_coef[is.na(df\_UKPDS\_coef)] <- 0

# Extract parameter names (used as row names)

v\_coef\_names <- df\_UKPDS\_coef$Parameter # Get row names from the 'Parameter' column

# Determine the number of parameters (rows)

n\_coef <- length(v\_coef\_names) # Count the number of parameters

# Extract factor names (used as column names), excluding the first column

v\_factors\_names <- colnames(df\_UKPDS\_coef[-1]) # Get column names excluding 'Parameter'

# Determine the number of factors (columns)

n\_equa <- length(v\_factors\_names) # Count the number of factors

# Allow for bootstrapped coefficients

n\_boot <- 1

# Create an array that holds onto everything!

a\_coef\_ukpds <- array(

data = NA,

dim = c(n\_coef, n\_equa, n\_boot),

dimnames = list(v\_coef\_names, v\_factors\_names, paste0("boot\_rep\_", 1:n\_boot))

)

dimnames(a\_coef\_ukpds)

# Fill in the array with coefficients from the dataset

a\_coef\_ukpds[, v\_factors\_names, 1] <- as.matrix(

df\_UKPDS\_coef[, v\_factors\_names, drop = FALSE]

)

# Extract individual characteristics at initial bootstrap slice

a\_coef\_ukpds\_ind\_traits<- a\_coef\_ukpds[1:62, , "boot\_rep\_1", drop = FALSE]

a\_coef\_ukpds\_other\_ind\_traits<- a\_coef\_ukpds[63:65, , ,drop = FALSE]

Setting Up the Patient Population and Trace

We import a synthetic dataset of 250,000 newly diagnosed patients with type 2 diabetes. Each patient will be simulated individually over a 20-cycle (year) time horizon. We also set the simulation parameters, including the random seed, cohort size, and number of cycles, to ensure reproducibility.

To support matrix multiplication with the coefficient array, we construct a patient trace matrix whose column names correspond to the parameter rows in the coefficient array. This matrix is split into individual-level traits and person-invariant characteristics to align with model requirements. Although the simulation processes patients individually, we create a full matrix structure in advance. A function, initialize\_patients, retrieves a specific patient’s baseline data and populates the trace matrix with their characteristics. The matrix dimensions are defined such that rows represent time cycles and columns represent model variables. Additional variables—such as transformed or derived inputs needed for modeling—are generated as part of the initialization process.

# Step 2: Load the patient dataset and save as matrix

m\_ukpds\_pop <- read\_csv("data/population.csv") |>

as.matrix()

seed <- 1234 # random number generator state

num\_i <- 250000 # number of simulated individuals

# Define the number of time points

num\_cycles <- 20 # maximum length of a simulation

set.seed(seed) # set the seed to ensure reproducible samples below

v\_ids <- paste("id", 1:num\_i, sep ="\_")

v\_cycle\_nms <- paste("cycle", 0:num\_cycles, sep ="\_")

# Create matrix with columns for each variable and a row for each cycle

m\_all\_ind\_traits <- matrix(

data = NA,

nrow = length(cycles),

ncol = n\_coef\_names,

dimnames = list(cycles,v\_coef\_names)

)

# need to select an individual patient from the dataset to simulate

#' Initialize baseline values for multiple patients

#'

#' @param num\_patients The total number of patients to process.

#' @param ukpds\_pop A data frame containing patient characteristics.

#' @param m\_ind\_traits A matrix to store patient data.

#' @return The updated matrix with initialized patient data.

#' @export

initialize\_patients <- function(num\_patients, ukpds\_pop, m\_ind\_traits) {

patient<- num\_patients

# 1. Create a vector of column names containing the individual characteristics you want to copy:

v\_ind\_traits <- c(

"age", . . .

)

# 2. Assign all these columns in a single step.

m\_ind\_traits[1, v\_ind\_traits] <- m\_ukpds\_pop[patient, v\_ind\_traits]

# 3. Handle any variables that aren't in m\_ukpds\_pop.

m\_ind\_traits[1, "heamo"] <- 15

m\_ind\_traits[1, "heamo\_first"] <- 15

# Event history tracking

event\_vars <- c("amp\_event", . . . "ulcer\_hist")

# rescale variables

for (var in event\_vars) {

m\_ind\_traits[1, var] <- m\_ukpds\_pop[patient, var]

}

m\_ind\_traits[1,"sbp\_real"]<- m\_ind\_traits[1,"sbp"]\*10

m\_ind\_traits[1,"egfr\_real"]<- m\_ind\_traits[1,"egfr"]\*10

m\_ind\_traits[1,"hdl\_real"]<- m\_ind\_traits[1,"hdl"]/10

m\_ind\_traits[1,"heart\_rate\_real"]<- m\_ind\_traits[1,"heart\_rate"]\*10

m\_ind\_traits[1,"ldl\_real"]<- m\_ind\_traits[1,"ldl"]/10

# Set default values for lambda, rho, and death

# can i return 2 matrix in the final statement?

m\_other\_ind\_traits[1, "lambda"] <- 0

m\_other\_ind\_traits[1, "rho"] <- 1

m\_other\_ind\_traits[1, "death"] <- 0

# Atrial Fib and PVD do not update

m\_ind\_traits[, "atria\_fib"] <- m\_ind\_traits[1, "atria\_fib"]

m\_ind\_traits[, "pvd\_event"] <- m\_ind\_traits[1, "pvd\_event"]

return(m\_ind\_traits)

}

Now that we have create and understand the format of the coefficients and patients data, we can use this information to create functions to predict patients disease progression. First, we define a function to predict the value of a biomarker for the next cycle based on current-period patient data. This function uses matrix multiplication: it multiplies the current values from the trace matrix by the corresponding coefficients for the biomarker equation, adds the intercept, and returns a single predicted value. This modular structure enables efficient and consistent application across multiple biomarkers and patients.

# biomarker function

#' @param m\_ind\_traits A matrix containing patient characteristics over time.

#' @param a\_coef\_ukpds\_ind\_traits A 3D array of coefficients used for calculating risk.

#' @param biomarker\_eq A character string specifying the health outcome equation (e.g., "ihd").

#' @param time\_step An integer indicating the row in `m\_ind\_traits` to use for calculations.

#'

#' @return The updated biomarker is stored.

biomarker <- function(m\_ind\_traits,

a\_coef\_ukpds\_ind\_traits,

biomarker\_eq,

time\_step) {

# Calculate patient-specific factors using model coefficients and patient data

updated\_biomarker <- (m\_ind\_traits[max(1,time\_step-1),] %\*% a\_coef\_ukpds\_ind\_traits[, biomarker\_eq, 1] +

a\_coef\_ukpds\_other\_ind\_traits["lambda", biomarker\_eq, 1] )

return(updated\_biomarker)

}

We also create a function to update all biomarkers. This function calls the individual biomarker function described earlier and additionally updates any derived or transformed variables needed for downstream models, such as lagged values, threshold indicators, or scaled versions (see the appendix for detailed code). This function returns a matrix of updated values.

In the UKPDS model, health events (excluding mortality) are modeled using a combination of Weibull, exponential, and logistic regression approaches. Because exponential models are a special case of the Weibull model where the shape parameter equals one, we use a single Weibull-based function to handle both cases. Transition probabilities are computed as follows: we first calculate the linear predictor—referred to as patient\_factors—using matrix multiplication with patient characteristics and model coefficients, adding the intercept. This is then multiplied by disease duration (typically the time step) and raised to the shape parameter (rho) from the coefficient array to obtain the cumulative hazard. We compute the cumulative hazard at time 𝑡 and 𝑡 + 1, subtract their exponentiated difference, and then calculate the transition probability. As in a standard Monte Carlo simulation, we compare this probability to a random uniform draw to determine whether the event occurs for a given individual in that cycle.

# biomarker function

#' @param m\_ind\_traits A matrix containing patient characteristics over time.

#' @param a\_coef\_ukpds\_ind\_traits A 3D array of coefficients used for calculating risk.

#' @param health\_outcome A character string specifying the health outcome equation (e.g., "ihd").

#' @param health\_event A character string specifying the health outcome event in the patient trace.

#' @param time\_step An integer indicating the row in `m\_ind\_traits` to use for calculations.

#'

#' @return Whether the event occurred.

#' @export

weibull\_event <- function(m\_ind\_traits, a\_coef\_ukpds\_ind\_traits, health\_outcome, health\_event, time\_step) {

# Calculate patient-specific factors using model coefficients and patient data

patient\_factors <- (m\_ind\_traits[time\_step,] %\*% a\_coef\_ukpds\_ind\_traits[, health\_outcome, 1] +

as.vector(a\_coef\_ukpds\_other\_ind\_traits["lambda", health\_outcome, 1]) )

# Compute cumulative hazard at the current time step

cum\_hazard\_t <- exp(patient\_factors) \* (m\_ind\_traits[time\_step, "diab\_dur"]^(a\_coef\_ukpds\_other\_ind\_traits["rho", health\_outcome, 1]) )

# Compute cumulative hazard at the next time step (by adding 1 year to diabetes duration)

cum\_hazard\_t1 <- exp(patient\_factors) \* ((m\_ind\_traits[time\_step, "diab\_dur"] + 1)^(a\_coef\_ukpds\_other\_ind\_traits["rho", health\_outcome, 1]) )

# Calculate transition probability

trans\_prob <- 1 - exp(cum\_hazard\_t - cum\_hazard\_t1)

# Simulate whether the event occurs by comparing with a random uniform value

event <- trans\_prob > runif(1)

# Return the updated matrix

return(event)

}

For events modeled via logistic regression, we follow a similar approach. The linear predictor is calculated using matrix multiplication and added intercept. This value is then transformed using the logistic function to estimate a transition probability, which is again compared against a random uniform draw to determine event occurrence (see Appendix or GitHub for full code).

Using these two base functions—weibull\_event and logistic\_event—we define a more comprehensive function to simulate all relevant events within a cycle. The UKPDS model distinguishes between incident events (occurring in the current cycle) and historical events (those that occurred previously). As originally designed the UKDPS model randomizes the order in which events are processed in each cycle to avoid order effects. We define a list of events of interest and create associated flags for both current occurrence (e.g., \_event) and history (e.g., \_hist). At the beginning of each cycle, the event history columns are updated based on prior outcomes. Each event is then simulated using the appropriate modeling approach. Certain events require special handling—for example, myocardial infarction (MI) has sex-specific equations, ulcer is modeled using logistic regression, and amputation depends on ulcer history and allows for recurrence. The event update function returns a matrix of event indicators for each individual per cycle. Below we show snippets of the full code to provide some additional detail for the reader, see the appendix or GitHub for full code.

# updated health events function

#' @param m\_ind\_traits A matrix containing patient-level data, including health event history.

#' @param a\_coef\_ukpds\_ind\_traits A coefficient matrix used in Weibull and logistic event calculations.

#' @param time\_step An integer indicating the current time step to update events.

#'

#' @return Updated `m\_ind\_traits` matrix with event and history values updated for the given time step.

update\_health\_events <- function(m\_ind\_traits, a\_coef\_ukpds\_ind\_traits, time\_step) {

# Ensure m\_ind\_traits remains a matrix

if (!is.matrix(m\_ind\_traits)) {

stop("m\_ind\_traits must be a matrix.")

}

# Initialize event variables and update history

events <- c("amp", "blindness", "chf", "esrd", "ihd", "mi", "stroke", "ulcer")

# create event and history column names once and save each group in a vector

v\_event\_cols <- paste0(events, "\_event")

v\_history\_cols <- paste0(events, "\_hist")

# Update history columns in one vectorized call

m\_ind\_traits[time\_step, v\_event\_cols] <- 0

m\_ind\_traits[time\_step, v\_history\_cols] <- pmax(

m\_ind\_traits[max(1, time\_step - 1), v\_history\_cols],

m\_ind\_traits[max(1, time\_step - 1), v\_event\_cols]

)

m\_ind\_traits[time\_step, "amp\_event2"] <- 0

# Randomize event order

randomized\_events <- sample(events)

for (events in randomized\_events) {

if (events == "amp") {

amp1\_no\_ulcer <- weibull\_event(m\_ind\_traits, a\_coef\_ukpds\_ind\_traits, health\_outcome = "amp1\_no\_ulcer", health\_event = "amp\_event", time\_step = time\_step)

amp1\_yes\_ulcer <- weibull\_event(m\_ind\_traits, a\_coef\_ukpds\_ind\_traits, health\_outcome = "amp1\_yes\_ulcer", health\_event = "amp\_event", time\_step = time\_step)

m\_ind\_traits[time\_step, "amp\_event"] <- (amp1\_no\_ulcer \* (m\_ind\_traits[time\_step, "ulcer\_hist"] == 0)) +

(amp1\_yes\_ulcer \* (m\_ind\_traits[time\_step, "ulcer\_hist"] == 1))

#ensure that this is a new event

m\_ind\_traits[time\_step, "amp\_event"] <- m\_ind\_traits[time\_step, "amp\_event"] \* (m\_ind\_traits[time\_step, "amp\_hist"] == 0)

amp2 <- weibull\_event(m\_ind\_traits, a\_coef\_ukpds\_ind\_traits, health\_outcome = "amp2", health\_event = "amp\_event2", time\_step = time\_step)

m\_ind\_traits[time\_step, "amp\_event2"] <- 0

m\_ind\_traits[time\_step, "amp\_event2"] <- amp2 \* (m\_ind\_traits[time\_step, "amp\_hist"] == 1)

}

. . .

} else if (events == "ulcer") {

m\_ind\_traits[time\_step, "ulcer\_event"] <- logistic\_event(m\_ind\_traits, a\_coef\_ukpds\_ind\_traits, health\_outcome = "ulcer", health\_event = "ulcer\_event", time\_step = time\_step)

m\_ind\_traits[time\_step, "ulcer\_event"] <- m\_ind\_traits[time\_step, "ulcer\_event"] \* (m\_ind\_traits[time\_step, "ulcer\_hist"] == 0)

} else {

m\_ind\_traits[time\_step, paste0(events, "\_event")] <- weibull\_event(m\_ind\_traits, a\_coef\_ukpds\_ind\_traits, health\_outcome = events, health\_event = paste0(events, "\_event"), time\_step = time\_step)

m\_ind\_traits[time\_step, paste0(events, "\_event")] <- m\_ind\_traits[time\_step, paste0(events, "\_event")] \* (m\_ind\_traits[time\_step, paste0(events, "\_hist")] == 0)

}

}

return(m\_ind\_traits)

}

Mortality in the UKPDS model is modeled using either logistic regression or Gompertz survival models. These follow the same general framework as other events (see Appendix or GitHub for code). To simulate mortality, we define a composite function that integrates both the gompertz\_event and logistic\_event approaches. The UKPDS mortality model includes four equations, depending on whether the patient experienced an event in the current period and whether they have a history of events. Specifically:

* If an event occurred during the cycle, mortality is predicted using a logistic model, stratified by whether there was event history.
* If no event occurred, mortality is modeled using Gompertz functions, also stratified by event history.

We create indicator variables to capture event occurrence and history across eight complications. These indicators are then used to route the patient through the appropriate mortality model. Each model returns a mortality probability, which is combined with the indicators to yield a single probability of death. If the patient died in a prior cycle, this status is carried forward. The function returns whether the patient died in each period.

# mortality function

#' @param m\_ind\_traits A matrix containing patient-level data, including health event history.

#' @param m\_other\_ind\_traits A matrix containing lmabda, rhos and death.

#' @param a\_coef\_ukpds\_ind\_traits A coefficient matrix used in Gompertz and logistic event calculations.

#' @param time\_step An integer specifying the time step at which mortality should be calculated.

#'

#' @return The updated `m\_ind\_traits` matrix with the mortality status recorded for the specified time step.

mortality <- function(m\_ind\_traits, m\_other\_ind\_traits, a\_coef\_ukpds\_ind\_traits, time\_step) {

# Calculate new health event occurrence and prior history

# Define events of interest

events <- c("amp", "blindness", "chf", "esrd", "ihd", "mi", "stroke", "ulcer")

# Create vectors containing events and event-history names:

v\_event\_cols <- paste0(events, "\_event")

v\_hist\_cols <- paste0(events, "\_hist")

# Get the maximum across those columns, for the given time\_step

# Calculate any new health event

new\_event <- max(m\_ind\_traits[time\_step, v\_event\_cols])

# Calculate any prior history of health events

any\_history <- max(m\_ind\_traits[time\_step, v\_hist\_cols])

# Determine event-history combinations

nhne <- new\_event == 0 & any\_history == 0 # No history, no event

yhne <- new\_event == 0 & any\_history == 1 # Yes history, no event

nhye <- new\_event == 1 & any\_history == 0 # No history, new event

yhye <- new\_event == 1 & any\_history == 1 # Yes history, new event

# Mortality calculations using Gompertz and logistic models

death\_nhne <- gompertz\_event(m\_ind\_traits, a\_coef\_ukpds\_ind\_traits, health\_outcome = "death\_nhne",

health\_event = "death\_nhne", time\_step = time\_step)

death\_yhne <- gompertz\_event(m\_ind\_traits, a\_coef\_ukpds\_ind\_traits, health\_outcome = "death\_yhne",

health\_event = "death\_yhne", time\_step = time\_step)

death\_nhye <- logistic\_event(m\_ind\_traits, a\_coef\_ukpds\_ind\_traits, health\_outcome = "death\_1st\_event",

health\_event = "death\_nhye", time\_step = time\_step)

death\_yhye <- logistic\_event(m\_ind\_traits, a\_coef\_ukpds\_ind\_traits, health\_outcome = "death\_yhye",

health\_event = "death\_yhye", time\_step = time\_step)

# Calculate new mortality status

new\_death <- nhne \* death\_nhne + yhne \* death\_yhne + nhye \* death\_nhye + yhye \* death\_yhye

# Update the mortality status in the matrix for the given time step

m\_other\_ind\_traits[time\_step, "death"] <- new\_death + m\_other\_ind\_traits[max(time\_step - 1, 1), "death"]

return(m\_other\_ind\_traits)

}

With all key functions now defined, we are ready to run the full simulation for one patient at a time. For each patient, we initialize their baseline traits and carry forward any time-invariant characteristics across all cycles (ie rows). Within each cycle, we update predictable time-varying characteristics (e.g., age, diabetes duration), simulate clinical events and mortality, and update biomarkers. Once the patient’s trajectory is complete, we calculate their costs and QALYs and summarize their outcomes into a single row, which is added to the final dataset of simulated patients. Code of the patient loop

for (patient in 1:250000) {

#create a patient population

m\_ind\_traits <- initialize\_patients(patient, ukpds\_pop, m\_ind\_traits)

#part of the initialization process

m\_other\_ind\_traits[1, "death"]<-0

# carry forward time invariant characteristics

m\_ind\_traits[ ,"age\_diag"]<-m\_ind\_traits[1 ,"age\_diag"]

m\_ind\_traits[ ,"black"]<-m\_ind\_traits[1 ,"black"]

m\_ind\_traits[ ,"indian" ]<-m\_ind\_traits[1 ,"indian"]

m\_ind\_traits[ ,"female" ]<-m\_ind\_traits[1,"female" ]

m\_ind\_traits[ ,"smoke"]<- m\_ind\_traits[1,"smoke"]

for (time\_step in 1:num\_cycles) {

m\_other\_ind\_traits[time\_step,"death"]<-m\_other\_ind\_traits[max(time\_step-1,1),"death"]

m\_other\_ind\_traits[time\_step,"lambda"]<-1

m\_other\_ind\_traits[time\_step,"rho"]<- 1

m\_ind\_traits[time\_step,"age"]<-m\_ind\_traits[max(1,time\_step-1),"age"] +1

m\_ind\_traits[time\_step,"diab\_dur"]<-m\_ind\_traits[max(1,time\_step-1),"diab\_dur"]+1

m\_ind\_traits[time\_step,"diab\_dur\_log"]<- (log(m\_ind\_traits[time\_step,"diab\_dur"]))

# ready to simulate

# event prediction at t

m\_ind\_traits <- update\_health\_events(m\_ind\_traits, a\_coef\_ukpds\_ind\_traits, time\_step = time\_step)

# mortality prediction at t

m\_other\_ind\_traits <- mortality(m\_ind\_traits, m\_other\_ind\_traits, a\_coef\_ukpds\_ind\_traits, time\_step = time\_step)

#predict the risk factors for the next cycle (t+1)

m\_ind\_traits<- update\_all\_biomarkers(m\_ind\_traits, a\_coef\_ukpds\_ind\_traits, time\_step = time\_step, next\_row = time\_step+1)

}

m\_ind\_traits\_new <- m\_ind\_traits[-nrow(m\_ind\_traits), ]

m\_summary <- matrix(

data = NA,

nrow = length(cycles),

ncol = 4,

dimnames = list(cycles,column\_names)

)

m\_summary <- m\_summary[-nrow(m\_summary), ]

... assign costs and qalys

patient\_summary\_file[patient,"cost"]<-sum(m\_summary[,"cost"])

patient\_summary\_file[patient,"disc\_costs"]<-sum(m\_summary[,"disc\_costs"])

patient\_summary\_file[patient,"qalys"]<-sum(m\_summary[,"qalys"])

patient\_summary\_file[patient,"disc\_qalys"]<-sum(m\_summary[,"disc\_qalys"])

}

**Running a Population Simultaneously Through a Model, A Cube Approach (3-dimensional array)**

The previous approach mirrors how many health economists initially learn to build microsimulation models—running one patient at a time—similar to traditional Markov cohort modeling in Excel. Consistent with prior research, we now demonstrate how to improve simulation speed by leveraging R’s vectorization capabilities. Rather than simulating one individual at a time, we process the entire cohort simultaneously—conceptually, like many joyful individuals skipping across a field together, hand in hand singing “We are the world”.

To accomplish this, we restructure the population matrix to an array. Now, columns still represent variables, but rows represent individuals, and time progresses across slices of a three-dimensional array.[[3]](#footnote-3) This shift dramatically improves computational efficiency and scalability, with only minimal modifications to the underlying code.

As in the prior approach, we begin by loading the coefficient array, patient dataset, and defining simulation parameters. We then create a 3D array: columns correspond to model variables, rows to individuals, and slices to time cycles. This array is separated into time-varying and time-invariant components. The primary change is creating an array instead of a matrix, using the same variables we have already created:

|  |  |
| --- | --- |
| One at a time approach | Vectorized approach |
| # Create matrix with columns for each variable and a row for each cycle  m\_all\_ind\_traits <- matrix(  data = NA,  nrow = length(cycles),  ncol = n\_coef\_names,  dimnames = list(v\_cycle\_nms,v\_coef\_names)  ) | # Create AN ARRAY with columns for each variable, row for each person, and a  # slice for each period  a\_all\_ind\_traits <- array(  data = NA,  dim = c(num\_i, n\_coef, num\_cycles + 1),  dimnames = list(v\_ids, v\_coef\_names , v\_cycle\_nms ) |

Additionally, instead of using the initialize\_patients function, we fill the time 0 of the patient trace.

# Fill patient trace

## First confirm all trait names in m\_ukpds\_pop and a\_all\_ind\_traits match

stopifnot(

all(dimnames(a\_all\_ind\_traits)[[2]] %in% colnames(m\_ukpds\_pop))

)

## Identify m\_ukpds\_pop columns with time-invariant data

v\_t\_invar\_col\_nms <- grep(

pattern = "^age\_diag$|^black$|^indian$|^female$|^smoke$|\_first$",

x = colnames(m\_ukpds\_pop),

ignore.case = FALSE,

value = TRUE

)

## Identify m\_ukpds\_pop columns with time-varying data

v\_t\_var\_col\_nms <- setdiff(

dimnames(a\_all\_ind\_traits)[[2]], v\_t\_invar\_col\_nms

)

a\_all\_ind\_traits[, v\_t\_var\_col\_nms, 1] <- m\_ukpds\_pop[

, v\_t\_var\_col\_nms, drop = FALSE

]

a\_all\_ind\_traits[, v\_t\_invar\_col\_nms, ] <- m\_ukpds\_pop[

, v\_t\_invar\_col\_nms, drop = FALSE

]

# break array up into stuff individual traits

a\_ind\_traits <- a\_all\_ind\_traits[,1:62,]

# remaining array that captures lambda, rho and death

a\_other\_ind\_traits <- a\_all\_ind\_traits[,63:65,]

Next, we redefine the biomarker, event, and mortality functions. These functions perform the same calculations as before, but now operate across the entire cohort in parallel, taking advantage of R’s strengths in vectorized computation[[4]](#footnote-4).

Improving the Speed of the Model Judiciously using RCPP

After building our R-based simulation, we next explore whether computational performance can be further improved using C++. In R, this is commonly achieved using the Rcpp package, which allows developers to write and call C++ code from within R. For matrix operations, we use the RcppArmadillo extension, which provides a fast and flexible interface to the Armadillo linear algebra library.

Rcpp is the most widely used framework for integrating C++ with R, offering a balance between performance and usability. Rather than evaluating C++ expressions directly, we focus on converting entire R functions into C++ equivalents, allowing us to benchmark and compare performance. If there is a meaningful speed improvement, we recommend adopting the C++ version; otherwise, maintaining the function in R is typically preferable for ease of review and debugging.

**Getting Started with Rcpp**

Rcpp functions can be written inline using cppFunction(), which compiles and links C++ code directly in the R environment. For more complex use cases, C++ code can be stored in .cpp files and sourced using Rcpp::sourceCpp("path\_to\_file.cpp"). This approach allows better organization and scalability.

To bridge R and C++, Rcpp provides convenient wrappers for basic R data types. Below are examples illustrating how to define and assign scalars and vectors using three approaches: STL (Standard Template Library), Rcpp, and Armadillo.  
Scalars and Vectors: The following C++ functions demonstrate how scalars and vectors are defined (initialized) and assigned values within a function environment.

|  |  |  |  |
| --- | --- | --- | --- |
| Data Type | STL (Standard C++) | Rcpp | Armadillo |
| Integer Scalar | int i = 42; | Rcpp::IntegerVector i = 42; | N/A |
| Double Scalar | double d = 3.14; | Rcpp::NumericVector d = 3.14; | N/A |
| Char Scalar | char c = 'A'; | Rcpp::CharacterVector c = "A"; | N/A |
| Integer Vector | std::vector<int> v = {1, 18, 315}; | Rcpp::IntegerVector v = Rcpp::IntegerVector::create(1, 18, 315); | arma::ivec v = {1, 18, 315}; |
| Double Vector | std::vector<double> v = {1.1, 2.2, 3.3}; | Rcpp::NumericVector v = Rcpp::NumericVector::create(1.1, 2.2, 3.3); | arma::vec v = {1.1, 2.2, 3.3}; |
| Character Vector | std::vector<std::string> v = {"A", "B", "C"}; | Rcpp::CharacterVector v = Rcpp::CharacterVector::create("A", "B", "C"); | arma::field<std::string> v = {"A", "B", "C"}; |

Matrices, Cubes and Lists:

The following C++ functions demonstrate how matrices, cubes and lists are defined (initialized) and assigned values within a function environment.

* **Matrices**: You can use Rcpp::IntegerMatrix, Rcpp::NumericMatrix, or arma::imat / arma::dmat to define and manipulate matrices.
* **Cubes (3D arrays)**: Armadillo’s arma::cube structure is used to create and modify 3D arrays. For example, individual slices can be filled or assigned row-wise and column-wise values.
* **Lists**: Use Rcpp::List to return heterogeneous objects, such as vectors, matrices, and cubes, from a single function. This is especially useful for organizing intermediate results from C++ functions.

These examples are meant to illustrate how detailed and explicit C++ can be relative to R. This added control and efficiency come at the cost of increased verbosity.

Looping and conditionals in C++ follow conventional syntax:

**For loops in C++ to determine if a number is even or odd**:

# For loops and if statements

## looping to print even or odd for 1:10

for\_if <- function() {

# For loop with a conditional statement in R

for (i in 1:10) {

if (i [%%](https://rdrr.io/r/base/Arithmetic.html) 2 == 0) {

[print](https://rdrr.io/r/base/print.html)([paste](https://rdrr.io/r/base/paste.html)(i, "is even"))

} else {

[print](https://rdrr.io/r/base/print.html)([paste](https://rdrr.io/r/base/paste.html)(i, "is odd"))

}

}

}

for\_if()

## The same but in C++

Rcpp::[cppFunction](https://rdrr.io/pkg/Rcpp/man/cppFunction.html)( # arma

code = '

void for\_if\_dev() {

// For loop with a conditional statement in C++

for (int i = 1; i <= 10; ++i) {

if (i % 2 == 0) {

std::cout << i << " is even" << std::endl;

} else {

std::cout << i << " is odd" << std::endl;

}

}

}'

)

for\_if\_dev()

Inside this function, a for loop iterates from 1 to 10. In C++, the for loop syntax is for (int i = 1; i <= 10; ++i). The ++i syntax is a pre-increment operator that increases the value of i before the next iteration check. For each iteration, an if statement checks if the current number i is even (i.e., i % 2 == 0). If the number is even, it prints a message indicating that the number is even std::cout << i << " is even" << std::endl;; otherwise, it prints a message indicating that the number is odd.

It is worth pointing out that the C++ function  requires explicit type declarations, which can lead to more predictable but less flexible behavior. The function, defined and compiled using Rcpp, performs the same addition but expects an integer input. When add\_two\_dev is called with an integer, such as 2, it works as expected. However, when called with a floating-point number like 5.10001, C++’s strict type system converts the input to an integer, truncating the decimal part and returning 7 instead of 7.10001. This demonstrates C++’s emphasis on performance and type safety at the cost of flexibility, making it more suitable for computationally intensive tasks where explicit control over data types is crucial. Integrating C++ with R through Rcpp allows users to leverage the strengths of both languages, combining R’s ease of use with C++’s performance advantages.

Recoding parts of the Microsimulation model in C++ with Rcpp

Our process for recoding specific parts of the model into C++ to achieve speed improvements is included below.

* **Identify Function:** Identify a critical function within the model that executes frequently, such as sampling new health states each cycle.
* **Recoding in C++ with Rcpp:** Utilize the [cppFunction()](https://rdrr.io/pkg/Rcpp/man/cppFunction.html) function from the Rcpp package to convert the identified function into C++ directly within the R script. This process can leverage AI Copilots for efficient conversion.
* **Benchmarking Performance:** Use microbenchmark to compare the execution speed of both the original R function and the newly implemented C++ function. Ensure that both versions produce identical outputs. Prefer the C++ version only if it significantly enhances performance, especially for tasks involving repetitive linear algebra operations or simulations.
* **Move to .cpp File:** Once satisfied with the C++ function’s performance, transfer it to a .cpp file. Organize C++ functions in a dedicated folder (e.g., src/) separate from R functions (typically stored in R/).
* **Source and Call from R:** Use the [sourceCpp()](https://rdrr.io/pkg/Rcpp/man/sourceCpp.html) function from the Rcpp package to source the C++ function from the .cpp file into the R environment. Subsequently, integrate and invoke the C++ function within the model script seamlessly, akin to any R function.

This structured approach should result in efficient utilization of C++ for enhancing computational speed in critical model components, while maintaining code clarity and integration within the R environment.

Now that we have a basic understanding of how to incorporate C++ into our model, let’s update our functions. We will walk the reader through updating the Gompertz function as shown in Table X. First, the Gompertz hazard formula is structurally identical between R and C++, but C++ has strict naming conventions so each component must be explicitly typed and separated in C++. There's no dynamic typing like in R. Similarly, one factor that might be new is that R conveniences (e.g., named indexing) must be replaced with positional logic in C++ and C++/Rcpp gains speed by avoiding dynamic typing and using lower-level matrix operations via Armadillo. This result is the Rcpp code being very verbose and cranky if anything was not declared correctly.

|  |  |  |
| --- | --- | --- |
| R Code (gompertz\_event2) | C++ Code (gompertz\_eventC) | Explanation |
| gompertz\_event2 <- function(  m\_ind\_traits,  m\_coef\_ukpds\_ind\_traits,  m\_coef\_ukpds\_other\_ind\_traits,  health\_outcome) { | // [[Rcpp::export]] auto gompertz\_eventC(  arma::mat& m\_ind\_traits,  const arma::mat& m\_coef\_ukpds\_ind\_traits,  const arma::mat& m\_coef\_ukpds\_other\_ind\_traits,  int health\_outcome\_index) { | The function is exported to R using Rcpp attributes. The input arguments are typed explicitly as Armadillo matrices. `int health\_outcome\_index` replaces R’s use of column names and is adjusted to 0-based indexing inside the function. It is worth mentioning that we are when adding the ‘&’ it passing by reference instead of actually passing the matrix back and forth. |
|  | int n\_rows = m\_ind\_traits.n\_rows; | Retrieves the number of individuals (rows in matrix). This is needed to generate a matrix of random uniform values using `arma::randu` later. |
|  | int idx = health\_outcome\_index - 1; | Adjusts for 0-based indexing used in C++ (R is 1-based). This index is used to access columns in coefficient matrices. |
| patient\_factors <- (m\_ind\_traits %\*% m\_coef\_ukpds\_ind\_traits[, health\_outcome] +  as.vector(m\_coef\_ukpds\_other\_ind\_traits["lambda", health\_outcome])) | arma::vec coef = m\_coef\_ukpds\_ind\_traits.col(idx); double lambda = m\_coef\_ukpds\_other\_ind\_traits(0, idx); arma::vec patient\_factors = m\_ind\_traits \* coef; patient\_factors += lambda; | Performs matrix multiplication and adds the intercept. C++ breaks this into sequential steps with explicit types. `col(idx)` extracts a single column as a vector. Matrix multiplication is performed using `\*`. Scalars must be explicitly extracted. `+=` adds lambda to each element of the result vector. |
| cum\_hazard\_t <- (1 / m\_coef\_ukpds\_other\_ind\_traits["rho", health\_outcome]) \*  exp(patient\_factors) \*  (exp(m\_ind\_traits[, "age"] \* m\_coef\_ukpds\_other\_ind\_traits["rho", health\_outcome]) - 1) | double rho = m\_coef\_ukpds\_other\_ind\_traits(1, idx); double inv\_rho = 1.0 / rho; const arma::vec& age = m\_ind\_traits.col(0); arma::vec patient\_factors\_exp = arma::exp(patient\_factors); arma::mat p\_t0 = arma::exp(age \* rho) - 1.0; arma::mat cum\_hazard\_t = inv\_rho \* (patient\_factors\_exp % p\_t0); | Cumulative hazard at time t using Gompertz function. `col(0)` assumes the first column of m\_ind\_traits is age. Armadillo's `exp()` is element-wise. `%` denotes element-wise multiplication. Intermediate results are stored in named variables for clarity and performance. |
| cum\_hazard\_t1 <- (1 / m\_coef\_ukpds\_other\_ind\_traits["rho", health\_outcome]) \*  exp(patient\_factors) \*  (exp((m\_ind\_traits[, "age"] + 1) \* m\_coef\_ukpds\_other\_ind\_traits["rho", health\_outcome]) - 1) | arma::vec age1 = age + 1; arma::mat p\_t1 = arma::exp(age1 \* rho) - 1.0; arma::mat cum\_hazard\_t1 = inv\_rho \* (patient\_factors\_exp % p\_t1); | Computes cumulative hazard at t+1. The logic is the same but is broken into parts. |
| trans\_prob <- 1 - exp(cum\_hazard\_t - cum\_hazard\_t1) | arma::mat trans\_prob = 1 - arma::exp(cum\_hazard\_t - cum\_hazard\_t1); | Both versions calculate transition probabilities from the difference in cumulative hazard. `arma::exp` is element-wise exponential. |
| event <- (trans\_prob > runif(nrow(m\_ind\_traits))) \* 1 | arma::mat random\_numbers = arma::randu(n\_rows, 1); arma::umat event = trans\_prob > random\_numbers; | Generates uniform random draws and determines whether the event occurred. `arma::randu(n\_rows, 1)` generates a matrix of random uniform values. Logical comparison returns a `umat` (unsigned int matrix) where 1 indicates event occurrence. Armadillo handles this natively. |
| colnames(event) <- health\_outcome |  | Column names are omitted because Armadillo matrices do not store metadata like names. If needed, this must be added in R after returning the object. |
| return(event) | return event; | Returns the logical matrix indicating event occurrence for each individual. |

**Code Profiling**

Now that we’ve demonstrated how to improve performance using Rcpp, we turn to the question of where to apply it. The key is to identify which parts of the code are most in need of optimization. This can be done using the profvis package, which provides a visual representation of code performance. By focusing on the most time-consuming processes and iteratively improving them, modelers can often speed up their simulations by orders of magnitude—helping to overcome the computational challenges inherent in microsimulation studies.

**Start by profiling your code using** the profvis() function. This will provide insights into how long different segments of your code take to execute. It’s important to focus on parts of the code that consume a large proportion of the total runtime, rather than spending time optimizing code that runs quickly. The goal is to identify the “low-hanging fruit”—areas where optimization will have the biggest impact.

**Identify parts of the code that could be improved.** Analyze the profvis output and look for segments that account for a meaningful share of runtime or that seem unnecessarily complex. These are good candidates for optimization. Potential improvements might include: pre-assigning vectors and matrices, vectorizing loops, using faster R packages, and rewriting the function in C++ and calling it from R using Rcpp. The aim is to develop alternatives that are both faster and flexible enough to support further development or changes.

Once you've identified target sections of code, isolate them into standalone functions and begin developing more efficient versions. Vectorization is often a first step, followed by moving to C++ via Rcpp if needed. You might also consider using optimized libraries that offer faster implementations of standard operations.

Use tools like microbenchmark, bench, or tictoc to compare the runtimes of your original code and the optimized version. Make sure to run the tests multiple times for accuracy. Also verify that the optimized code produces results equivalent to the original version—particularly important for simulations where outcomes may depend on subtle numerical behavior.

Based on the benchmarking results, adopt the better-performing version in your main codebase. Replace the original implementation with the faster one to realize the performance gains. Keep in mind that this is an iterative process. Optimization continues until the performance gains no longer justify the time investment.

**Incorporating Parallel Computing into Microsimulation Models**

Parallel computing involves dividing a computational task into independent components that can be executed simultaneously across multiple processors or cores. This approach can significantly reduce processing time, making it particularly beneficial for computationally intensive tasks such as microsimulation and probabilistic sensitivity analysis (PSA).

By default, R is a single-threaded, interpreted language, meaning it processes tasks sequentially. However, R supports parallel execution through several open-source packages, allowing modelers to bypass some of its inherent performance limitations. Open-source R packages like future, furrr, and parallel streamline the implementation of parallel workflows. In addition, cloud computing infrastructure provides access to machines with dozens or even hundreds of cores make using parallel processing even more appealing. As models grow more complex and datasets become larger, leveraging parallelism becomes increasingly necessary to ensure timely execution.

Microsimulation models often meet the key criterion for parallelization: the independence of simulation tasks. For example, individual patient simulations typically operate independently of one another, as do separate PSA iterations. This allows for two natural targets for parallel execution: Inner-loop parallelization, by running multiple individual simulations concurrently and outer-loop parallelization where we run multiple PSA iterations in parallel.

Since PSA tends to be the most computationally demanding component—requiring the model to be re-run hundreds or thousands of times with different parameter values—we focus here on outer-loop parallelization. However, the same principles can be applied to the inner loop.

The following functions illustrate a structured approach to implementing PSA in parallel using future, furrr, and supporting tools.

**1. set\_parallel(): Configure the Execution Environment**

This function sets up the parallel processing strategy appropriate to the user’s operating system and available resources. On Windows systems, multisession is used, while Unix-based systems (Linux/macOS) can support multisession, multicore, or cluster strategies. If no valid configuration is provided, computation defaults to sequential execution.

**2. make\_psa\_chunks(): Divide Simulation Tasks**

This function splits the total number of simulations (or rows in a PSA parameter data frame) into smaller “chunks” that are distributed across cores. It supports two modes:

* **Data indices**: Used when only iteration counts are available.
* **Data frames**: Used when PSA inputs have already been sampled.

Chunks are assigned to workers either equally or based on a user-specified chunk\_size.

**3. run\_psa\_parallel(): Execute the PSA in Parallel**

This function orchestrates the end-to-end PSA process, including:

* Sampling PSA parameter sets
* Configuring the parallel execution environment
* Dividing simulation tasks using make\_psa\_chunks()
* Executing simulations across workers using furrr::future\_map()
* Collecting and combining results from all workers

Each parallel worker evaluates a subset of PSA iterations using purrr::pmap() to apply the model to each sampled configuration. The output is a unified data frame of PSA results.

While parallel computing can substantially reduce execution time, its effectiveness is subject to certain limitations:

* **Core count**: Performance scales with available cores; speed-ups plateau when cores are saturated.
* **Memory**: Sufficient RAM is required to support parallel workers.
* **Task granularity**: Very small or fast tasks may not benefit from parallelization due to overhead.
* **Operating system**: Behavior and available parallel strategies differ between Windows and Unix-like systems.

On cloud-based or high-performance computing environments, these limitations are reduced, enabling large-scale PSA execution in minutes rather than hours.

We will compare model results using microbenchmark and profiv to see the differences between model.

**Results**

When comparing the different versions of the UKPDS model

**Discussion**

This study introduces a comprehensive framework for developing computationally efficient and reproducible microsimulation models within the R programming environment. The primary aim was to address the critical need for robust modeling tools in pharmacoeconomics, particularly for simulating individual patient trajectories that exhibit substantial individual-level variability, history dependence, or process complexity—attributes that traditional Markov frameworks often struggle to capture adequately. A central finding of this research is the significant enhancement in computational efficiency demonstrated by the R-based microsimulation model when compared to its Excel-based counterpart. The R model achieved a remarkable runtime reduction of over 50%, with these performance gains becoming even more pronounced as the simulated cohort size increased. This highlights the inherent scalability of the proposed R-based approach, which is crucial for real-world applications in health economics. The demonstrated efficiency was attained through the strategic integration of several advanced programming and computational optimization techniques. These core strategies include leveraging vectorized operations, specifically by adopting a three-dimensional array (or "cube") approach for data management, implementing modular programming practices, judiciously integrating C++ code via Rcpp for computationally intensive functions, and incorporating parallelization to accelerate large-scale simulations, particularly probabilistic sensitivity analyses. The architectural shift from simulating one patient at a time to processing the entire cohort simultaneously is a fundamental driver of the observed scalability, dramatically improving computational efficiency and scalability with only minimal modifications to the underlying code. This capability is critical for advancing the realism and robustness of health economic evaluations, as it allows for the simulation of much larger and more diverse patient populations and supports more comprehensive uncertainty analyses, such as probabilistic sensitivity analyses (PSAs), which previously were computationally prohibitive.

The R-based microsimulation models, as demonstrated in this framework, simulate individual patient pathways over time, explicitly allowing for heterogeneity in baseline characteristics, history-dependent transitions, and stochastic variation in outcomes. This granular, individual-level representation offers a more realistic portrayal of disease dynamics, making the framework particularly well-suited for chronic diseases, oncology, and personalized medicine contexts where individual patient history profoundly influences future outcomes. The enhanced computational efficiency achieved through vectorized operations, Rcpp integration, and parallelization is pivotal for the practical application of these complex models, making comprehensive analyses feasible within reasonable computational timeframes and overcoming the "computational cost" historically associated with microsimulation. This capability enables health economists to move beyond methodological compromises, fostering a paradigm shift towards models that are both clinically rich and computationally tractable, thereby enhancing the credibility and utility of economic evaluations in informing real-world healthcare decisions. The framework's emphasis on modular programming significantly enhances model robustness. By structuring the model as a series of self-contained functions, each performing a specific task, the code becomes inherently easier to read, understand, and debug, allowing for systematic testing and validation of individual components. This leads to more accurate and reliable calculations, ensuring the model accurately reflects disease complexity. Furthermore, the open-source nature of R is a foundational element for reproducibility and transparency. Unlike proprietary software, R code is accessible and auditable, allowing other researchers to replicate the methods and results, and enabling health economists and decision-makers to scrutinize the underlying logic, assumptions, and calculations. The framework explicitly promotes reproducibility by defining simulation parameters such as the random seed, cohort size, and number of cycles, ensuring experiments can be precisely reproduced. This transparency is vital for building trust in model outputs and for facilitating peer review and validation of complex health economic analyses. The framework's ability to handle large-scale simulations is a paramount strength, achieved through several synergistic strategies.

The shift from a "one-at-a-time" patient simulation to processing the "entire cohort simultaneously" using a 3D array dramatically improves computational efficiency by fully leveraging R's inherent strengths in vectorized computation. For computationally intensive functions, the judicious use of Rcpp allows for recoding in C++, providing substantial speed improvements, particularly beneficial for repetitive linear algebra operations. Finally, the incorporation of parallel computing divides computational tasks into independent components, executing them simultaneously across multiple processors, which is especially valuable for microsimulation and probabilistic sensitivity analysis (PSA), allowing the model to be re-run hundreds or thousands of times with different parameter values in a fraction of the time. The true strength of this framework lies in the synergistic integration of these diverse optimization techniques, creating a significantly greater performance improvement than any single method alone, and representing a "best practice" for developing high-performance computational models in health economics.

Despite the significant advancements, it is important to acknowledge several inherent limitations and practical trade-offs. Microsimulation models inherently impose heavy demands on memory and processing time. While optimizations mitigate these demands, they do not eliminate them entirely, meaning large cohorts or long time horizons may still require substantial computational resources. The integration of C++ via Rcpp, while powerful, introduces trade-offs: C++ code is more verbose and strict, requiring explicit type declarations and replacing R's conveniences (like named indexing) with positional logic. C++'s strict type system can lead to less flexible behavior, and Rcpp integration is only recommended if it significantly enhances performance, as the increased complexity is only justified by substantial computational benefits. Furthermore, Armadillo matrices used with Rcpp do not store metadata like column names, requiring manual re-addition in R. Similarly, parallel computing, while offering speed reductions, is limited by core count saturation, memory requirements, task granularity (small tasks may not benefit due to overhead), and operating system differences, which require modelers to adapt their configuration. These factors imply that realizing the full benefits of this framework extends beyond just writing good code; it requires investment in appropriate computational infrastructure and careful environmental setup.

Building upon this robust and efficient framework, several promising avenues for future research emerge. A key direction involves extending the framework's application to a broader spectrum of chronic diseases or complex clinical areas. Further exploration of advanced high-performance computing methods could yield even greater efficiency gains, including more sophisticated Rcpp techniques, GPU computing, or leveraging cloud-based high-performance computing solutions for larger-scale simulations. To broaden accessibility, future work should focus on developing user-friendly R packages or graphical user interfaces (GUIs) based on these principles, abstracting complexities into more intuitive functions. Finally, integrating the framework with emerging data sources and analytical techniques, such as real-world data from electronic health records or machine learning algorithms for dynamic prediction, presents a rich area for research, potentially enhancing advanced uncertainty quantification methods. The computational efficiency achieved in the core microsimulation engine provides a critical foundation for integrating these models with cutting-edge data science advancements, enabling a new generation of dynamic, data-driven health economic analyses that can better inform precision medicine and adaptive policy. This study has successfully presented a comprehensive framework for building efficient, robust, scalable, reproducible, and transparent microsimulation models in R for health economics, serving as a vital foundation for future work and contributing to more informed healthcare decision-making.

**References**

1. Sun X, Faunce T. Decision-analytical modelling in health-care economic evaluations. Eur J Health Econ. 2008 Nov;9(4):313-23.

2. Graves J, Garbett S, Zhou Z, Schildcrout JS, Peterson J. Comparison of Decision Modeling Approaches for Health Technology and Policy Evaluation. Med Decis Making. 2021 May;41(4):453-64.

3. Siebert U, Alagoz O, Bayoumi AM, Jahn B, Owens DK, Cohen DJ, et al. State-transition modeling: a report of the ISPOR-SMDM Modeling Good Research Practices Task Force-3. Med Decis Making. 2012 Sep-Oct;32(5):690-700.

4. Alarid-Escudero F, Krijkamp E, Enns EA, Yang A, Hunink MGM, Pechlivanoglou P, et al. An Introductory Tutorial on Cohort State-Transition Models in R Using a Cost-Effectiveness Analysis Example. Med Decis Making. 2023 Jan;43(1):3-20.

5. Vemer P, Goossens LM, Rutten-van Mölken MP. Not simply more of the same: distinguishing between patient heterogeneity and parameter uncertainty. Med Decis Making. 2014 Nov;34(8):1048-58.

6. Krijkamp EM, Alarid-Escudero F, Enns EA, Jalal HJ, Hunink MGM, Pechlivanoglou P. Microsimulation Modeling for Health Decision Sciences Using R: A Tutorial. Med Decis Making. 2018 Apr;38(3):400-22.

7. Petrou S, Gray A. Economic evaluation using decision analytical modelling: design, conduct, analysis, and reporting. BMJ. 2011 Apr 11;342:d1766.

8. Marshall DA, Grazziotin LR, Regier DA, Wordsworth S, Buchanan J, Phillips K, et al. Addressing Challenges of Economic Evaluation in Precision Medicine Using Dynamic Simulation Modeling. Value Health. 2020 May;23(5):566-73.

9. Jalal H, Pechlivanoglou P, Krijkamp E, Alarid-Escudero F, Enns E, Hunink MGM. An Overview of R in Health Decision Sciences. Med Decis Making. 2017 Oct;37(7):735-46.

1. It is worth noting that this model could have been created with the biomarkers being defined at the start of the simulation, and not updated each cycle as they are not dynamically updated based on other events. However, we wanted to demonstrate a fully dynamic approach so that readers can learn about the most complex version which they can always simplify if needed for their own purposes. [↑](#footnote-ref-1)
2. As detailed later, we can expand the coefficient matrix to an array to facilitate PSA and will expand the patient matrix to an array to vectorize the model. [↑](#footnote-ref-2)
3. If the reader is used to thinking about Excel spreadsheets, the reader could imagine a spreadsheet that has an identical set of columns and rows for 20 different sheets, and time progresses as patients move from one sheet to the next. [↑](#footnote-ref-3)
4. When examining the code careful readers will notice that we pull out a single slice when doing our matrix multiplication, this ensures that we are not passing the entire array back and forth for each prediction which will slow down the simulation process. [↑](#footnote-ref-4)