UNIVERSITY OF ILLINOIS CHICAGO

Building Efficient Microsimulation Models in R

Aaron N Winn
Abdullah I Abdelaziz
Wael Mohamed
Jyotirmoy Sarker
Robert Smith

UIC RETZKY COLLEGE OF PHARMACY
The Department of Pharmacy Systems, Outcomes and Policy

Motivation

- Microsimulation is needed when individual-level variability matters.
  - Traditional Markov models assume memoryless transitions and population averages.
  - History dependence and individual heterogeneity require more flexible modeling.
  - Cannot capture history dependence, treatment waning, and comorbidities.
  - Workarounds (e.g., tunnel states) add complexity and explode state space.
- Off-the-shelf packages often aren't "quite right" and either require customization or more

For the 21-year-old version of myself

edical Decision Making
olume 38, Issue 3, April 2018, Pages 400-422

tps://doi-org.proxy.cc.uic.edu/10.1177/0272989X18754513

utorial

Microsimulation Modeling for Health Decision Sciences Using R: A Tutorial

line M. Krijkamp[1], Fernando Alarid-Escudero[2], Eva A. Enn
lunink[5,6,7], and Petros Pechlivanoglou[8,9]

bstract

Microsimulation models are becoming increasingly common i
ealth. Because microsimulation models are computationally mo
ohort models, the use of computer programming languages in
ommon. R is a programming language that has gained reco
odeling. It has the capacity to perform microsimulation m
ommonly used for decision modeling, incorporate statistical a
roduce more transparent models and reproducible results. H
nplementation of microsimulation models in R exists. In this tut

RETZKY COLLEGE OF PHARMACY
The Department of Pharmacy Systems, Outcomes and Policy

Medical Decision Making
Volume 40, Issue 2, February 2020, Pages 242-248

https://doi.org/10.1177/0272989X19893973

Brief Reports

A Multidimensional Array Representation of State-Transition Model Dynamics

Eline M. Krijkamp[1,*], Fernando Alarid-Escudero[2,*], Eva A. Enns[3], Petros Pechlivanoglou[4,5], M.G. Myriam Hunink[6,7], Alan Yang[8], and Hawre J. Jalal[9]

Abstract

Cost-effectiveness analyses often rely on cohort state-transition models (cSTMs). The cohort trace is the primary outcome of cSTMs, which captures the proportion of the cohort in each health state over time (state occupancy). However, the cohort trace is an aggregated measure that does not capture information about the specific transitions among health states (transition dynamics). In practice, these transition dynamics are crucial in many applications, such as incorporating transition rewards or computing various epidemiological outcomes that could be used for model calibration and validation (e.g., disease

Objectives

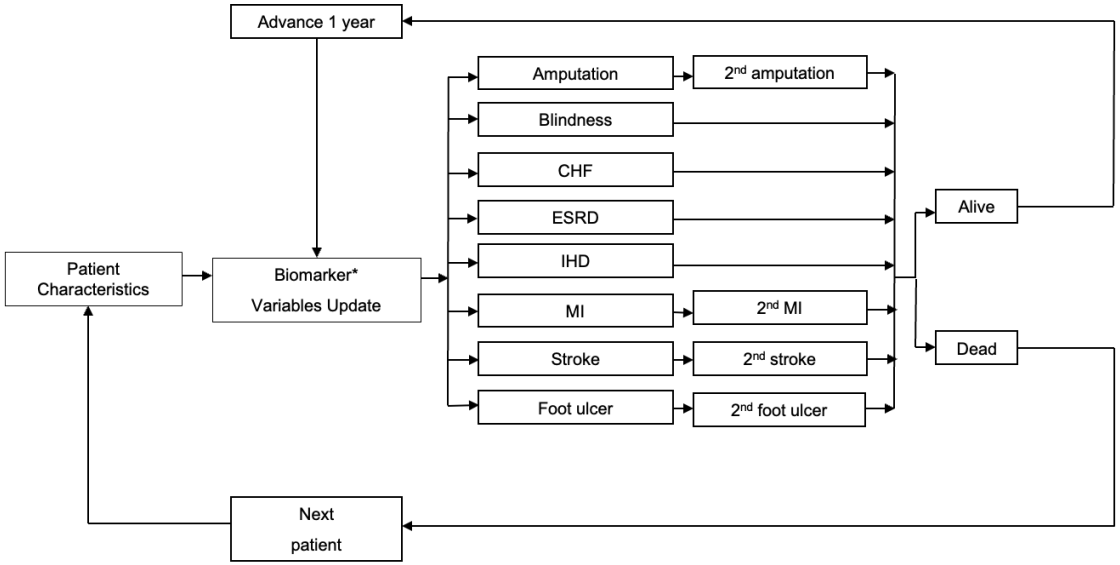
- Prove that…
  - Microsims aren't hard in R
  - Vectorization a fancy word for something very simple and easy.
  - Including C++ using Rcpp isn't that hard.

Case Study: United Kingdom Prospective Diabetes Model (UKPDS)

Well-known diabetes simulation model based on >5,000 patients.

Rich dataset with 30+ years of follow-up.

Includes equations for biomarkers and diabetes complications.



RETZKY COLLEGE OF PHARMACY
The Department of Pharmacy Systems, Outcomes and Policy

Building a Micro-simulation

- Create or take existing patients
- Create a coefficient matrix
- Create equations for events, biomarkers, and mortality over time
- Simulate patients
- Store outputs and summarize costs and QALYs

UIC RETZKY COLLEGE OF PHARMACY
The Department of Pharmacy Systems, Outcomes and Policy

What this looks like in Excel


But the run times…
103 minutes!

R: Coefficients

- Coefficient array: 2D (or 3D) (parameters × equations (x bootstrap))

```
# 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
```

R: Coefficients continued

```
# 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]
```

R: Patients

- Import patient data and clean
- Load the individual patient
- Patient trace: 2D matrix (time × traits)

```
# 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)
)
```

R: Loading a Patient Function

```
# 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, 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)
}
```

R: Equation Functions

```
# 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)
}
```

```
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)
}
```

Then, simulate individuals one at a time

Vectorization

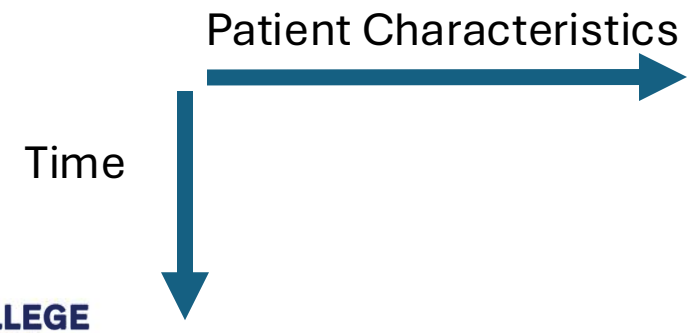
- Simulate the entire cohort… using 3D arrays.

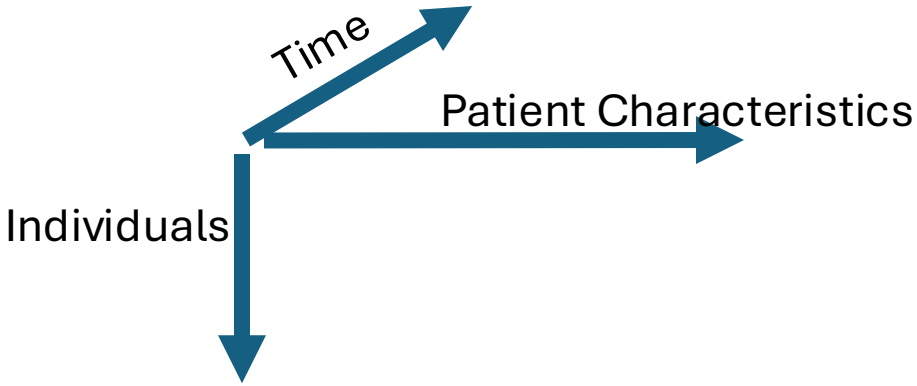
Instead of going “down” for time, imagine going “through”

Creating the Patient Trace

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 eac 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 )`

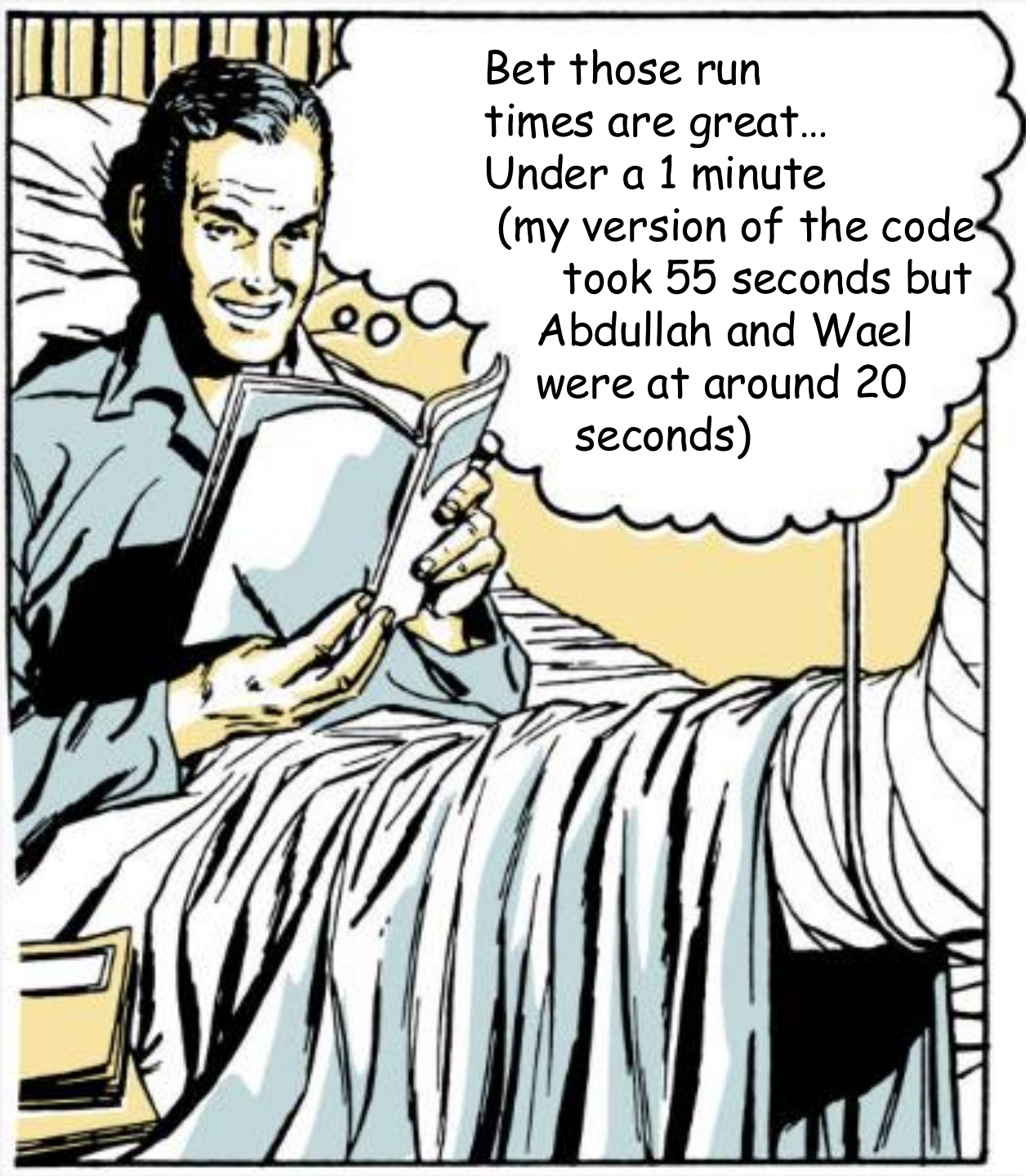




Updating the biomarker function

One at a time approach	Vectorized approach

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)
}
	biomarker <- function(
	  m_ind_traits,
	  a_coef_ukpds_ind_traits,
	  a_coef_ukpds_other_ind_traits,
	  biomarker_eq) {
	
	  # Calculate patient-specific factors using model coefficients and patient data
	  m_updated_biomarker <- m_ind_traits %*% a_coef_ukpds_ind_traits[, biomarker_eq, 1] +
	
	a_coef_ukpds_other_ind_traits["lambda", biomarker_eq, 1]
	
	  return(m_updated_biomarker)
	}


Bet those run times are great… Under a 1 minute (my version of the code took 55 seconds but Abdullah and Wael were at around 20 seconds)

Code Profiling

- Identify which parts of the code are most in need of optimization using the profvis package
- Once you've identified target sections of code, isolate them into standalone functions and begin developing more efficient versions
- Use tools like microbenchmark, bench, or tictoc to compare the runtimes of your original code and the optimized version

Using Rcpp and Armadillo for Speed

RETZKY COLLEGE OF PHARMACY
The Department of Pharmacy Systems, Outcomes and Policy

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.

R Code (gompertz_event2)	C++ Code (gompertz_eventC)	Explanation

`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.

Parallel Computing for PSA

- Use `future`, `furrr`, `parallel` for outer-loop PSA.
- Divide simulations across cores or nodes.
- Effective for scaling probabilistic sensitivity analysis.

Key Findings

- Excel is slow
- Microsimulations can easily be constructed in R by writing a handful of functions
- Converting a one-at-a-time to a vectorized microsimulation model requires minimal code modification
- Can increase speed with minimal use of C++

RETZKY COLLEGE OF PHARMACY
The Department of Pharmacy Systems, Outcomes and Policy