Sick-Sicker case study - as part of the framework paper

DARTH workgroup

2019-02-27

How to read this document

This document is meant to guide you through the `R` code of a full functional decision model, we use to showcase the DARTH framework described in the *A need for change! A coding framework for improving transparency in decision modeling* paper. This document is supportive material. Therefore, you are recommended to first read the paper itself before continue reading this file.

In this case-study, we perform a cost-effectiveness analysis (CEA) using a previously published 4-state model called the Sick-Sicker model. (Enns et al. 2015) The code of this model is structured using the DARTH framework. This framework can be downloaded from the DARTH-git on GitHub (https://github.com/DARTH-git/Decision-Modeling-Framework). We recommend downloading this complete framework as a .zip file containing all folders. Unzip the folder and save it on your computer. In this framework, you find multiple folders as described in Table 1 of the paper. We refer to the folders of this framework using *italic* style for their names. You probably found this markdown manuscript in the *manuscript* folder. Our figures can be found in the *figs* folder, data is the *data* folder and when we talk about functions, you can find them in the *functions* folder. All other `R` files can be found in the *R* folder. In this document we refer a lot to the `R` code but we do not always show all of this. Therefore, it is very useful to follow along while reading this document. To make sure you are able to run all code when folloing along, please first install all package we use by running the **00_general-setup.R** file in the `R` folder.

```
source("R/app0_packages-setup.R")
```

The Sick-Sicker model

In the Sick-Sicker model, a hypothetical disease affects individuals with an average age of 25 years and results in increased mortality, increased treatment costs and reduced quality of life (QoL). We simulate this hypothetical cohort of 25-year-old individuals over a lifetime (or reaching age 100 years old) using 75 annual cycles, represented with `n.t`. The cohort start in the "Healthy" health state (denoted "H"). Healthy individuals are at risk of developing the illness, at which point they would transition to the first stage of the disease (the "Sick" health state, denoted "S1"). Sick individuals are at risk of further progressing to a more severe stage (the "Sicker" health state, denoted "S2"), which is a constant probability in this case example. There is a chance that individuals in the Sick state eventually recover and return back to the Healthy healthy state. However, once an individual reaches the Sicker health state, they cannot recover; that is, the probability of transitioning to the Sick or Healthy health states from the Sicker health state is zero. Individuals in the Healthy state face background mortality that is age-specific (i.e., time-dependent). Sick and Sicker individuals face an increased mortality in the form of a hazard rate ratio (HR) of 3 and 10 times, respectively, on the background mortality rate. Sick and Sicker individuals also experience increased health care costs and reduced QoL compared to healthy individuals. Once simulated individuals die, they transition to the "Dead" health state (denoted "D"), where they remain. Figure 1 shows the state-transition diagram of the Sick-Sicker model. The evolution of the cohort is simulated in one-year discrete-time cycles. Both costs and quality adjusted life years (QALYs) are discounted at an annual rate of 0.03 %.

Two alternative strategies exist for this hypothetical disease: a no-treatment and a treatment strategy. Under the treatment strategy, Sick and Sicker individuals receive treatment and continue doing so until they recover or die. The cost of the treatment is additional to the cost of being Sick or Sicker for one year. The treatment improves QoL for those individuals who are Sick but has no effect on the QoL of those who are sicker. To evaluate these two alternative strategies, we perform a cost-effectivenss analysis (CEA).

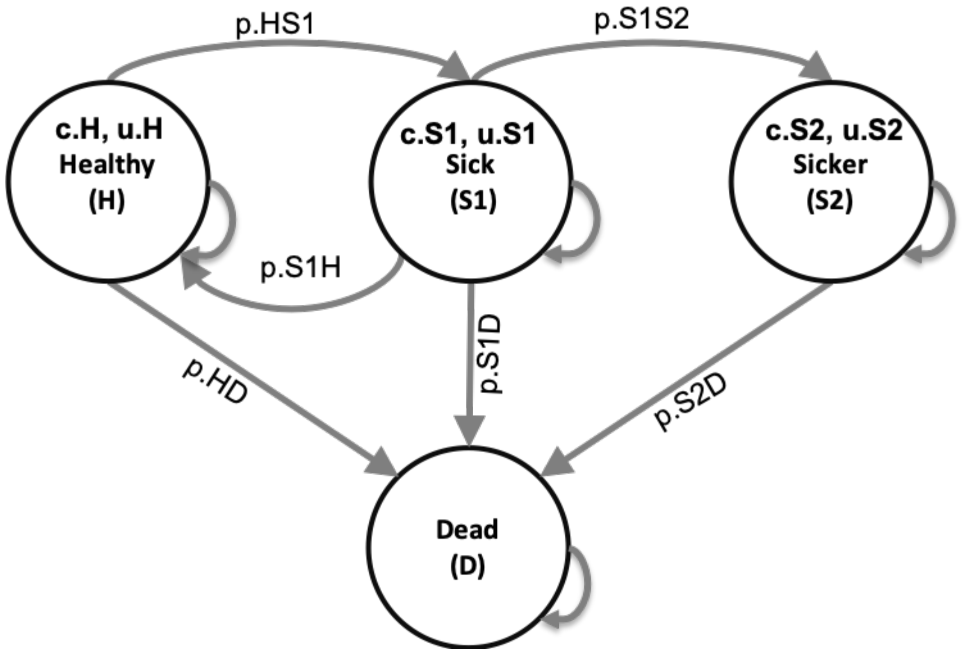


Figure 1: State-transition diagram of the Sick-Sicker model. Healthy individuals can transition towards Sick, they can die or they can stay healthy. Sick individuals can recover, transitioning back to healthy, they can die, or they stay sick. Once individuals are Sicker, they stay Sicker until they die.

We assume that most of the parameters of the Sick-Sicker model and their uncertainty have been previously estimated and are known to the analyst. However, while we can identify those who are afflicted with the illness through obvious symptoms, we can not easily distinguish those in the Sick state from the those in the Sicker state. Thus, we can not directly estimate state-specific mortality hazard rate ratios, nor do we know the transition probability of progressing from Sick to Sicker. Therefore, we calibrate the model to different epidemiological data. We internally validated the calibrated model by comparing the predicted outputs from the model evaluated at the calibrated parameters against the calibration targets.

As part of the CEA, we conducted different deterministic sensitivity analysis (SA), including one-way and two-way SA, and tornado plots. To quantify the effect of parameter uncertainty on decision uncertainty, we conducted a probabilistic sensitivity analysis (PSA) and reported our uncertainty analysis results with a cost-effectiveness acceptability curve (CEAC), cost-effectiveness acceptability frontier (CEAF) and expected loss curves (ELC). We also conducted a value of information (VOI) analysis to determine whether potential future research is needed to reduce parameter uncertainty. All steps of the CEA will be described using the different components of the framework.

01 Define model inputs

As described in the paper, in this component we declare all model input variables and set their values. The `R` script running all components of this section is the **01_model-inputs.R** file in the `R` folder.

The input to inform the values is categorized in three categories: external, estimated, and calibrated. The majority of the Sick-Sicker model parameters are informed by external data. Only three parameter values need to be estimated using model calibration.

In this component, we start with the general setup of the model, specifying among others the time horizon, number of health states, prevalence of the different health states at the start of the model and discount rates. The next step is to declare and inform the external parameter. The initial model parameter values and `R` variable names are presented in Table 1.

Table 1: Description of the initial parameters with their R name and value of the Sick-Sicker model.

Parameter	**R name**	**Value**
Time horizon (n_t)	`n.t`	75 years
Names of health states (n)	`v.n`	H, S1, S2, D
Annual discount rate (costs/QALYs)	`d.c/d.e`	3%
Annual transition probabilities		
- Disease onset (H to S1)	`p.HS1`	0.15
- Recovery (S1 to H)	`p.S1H`	0.5
- Disease progression (S1 to S2) in the time-homogenous model	`p.S1S2`	0.105
Annual mortality		
- All-cause mortality (H to D)	`p.HD`	age-specific
- Hazard rate ratio of death in S1 vs H	`hr.S1`	3
- Hazard rate ratio of death in S2 vs H	`hr.S2`	3
Annual costs		
- Healthy individuals	`c.H`	$2,000
- Sick individuals in S1	`c.S1`	$4,000
- Sick individuals in S2	`c.S2`	$15,000
- Dead individuals	`c.D`	$0
- Additional costs of sick individuals treated in S1 or S2	`c.Trt`	$12,000
Utility weights		
- Healthy individuals	`u.H`	1.00
- Sick individuals in S1	`u.S1`	0.75
- Sick individuals in S2	`u.S2`	0.50
- Dead individuals	`u.D`	0.00
Intervention effect		
- Utility for treated individuals in S1	`u.Trt`	0.95

The all-cause mortality of healthy individuals is age specific and derived from the Human Mortality database. We use the age-, sex- and race- (ASR) specific mortality rates for the US population from 2015. This information is stored in the **01_all-cause-mortality-USA-2015.csv** file in the *data* folder, which we load into our R environment.

All other external parameter values are stored in the **01_init-params.csv** file in the *data* folder. Based on this file a R data frame with the base-case parameters is generated by the `f.generate_init_params` function. Using a function to create a basecase parameters dataframe might see complex, but it is important for the sensitivity analysis we will do in component 5 of the framework. Below we guide you through the components of the function.

```
print.function(f.define_init_params) # print the code of the function
```

```
## function ()
## {
##     df.params.init <- read.csv(file = "data/01_init-params.csv")
##     df.params.init <- data.frame(c.H = df.params.init$c.H, c.S1 = df.params.init$c.S1,
##         c.S2 = df.params.init$c.S2, c.D = df.params.init$c.D,
##         c.Trt = df.params.init$c.Trt, u.H = df.params.init$u.H,
##         u.S1 = df.params.init$u.S1, u.S2 = df.params.init$u.S2,
##         u.D = df.params.init$u.D, u.Trt = df.params.init$u.Trt,
##         p.HS1 = df.params.init$p.HS1, p.S1H = df.params.init$p.S1H,
##         p.S1S2 = df.params.init$p.S1S2, hr.S1 = df.params.init$hr.S1,
##         hr.S2 = df.params.init$hr.S2)
##     return(df.params.init)
## }
## <bytecode: 0x7fe47c49a478>
```

In this code you can see that the functions start calling the .csv file and stores the values of the parameters. All of this is combined in a dataframe. The Sick-Sicker model does not have estimated parameters, but there are three parameters that need to be estimated via model calibration. In this stage of the framework, we simply set these parameters to valid "dummy" values that are compatible with the next phase of the analysis, model implementation, but are ultimately just placeholder values until we conduct the calibration phase. This means that these values will be replaced by the best-fitted calibrated values after we performed the calibration.

02 Model implementation

In this component, we build the backbone of the decision analysis: the implementation of the model. This component is performed by the **02_simulation-model.R** script. This file itself is not very large. It is simpply loading some packages, sources the input from component 01, sources the function `f.decision_model` that is used the capture the dynamic process of the Sick-Sicker example, runs this function and stores the output. The output of the model is the traditional cohort trace, describing how the cohort is distributed among the different health states over time. This trace will be used in many of the other component sections.

The function `f.decision_model` is generated in the **02_simulation-model_functions.R file**. As described in the paper, constructing a model as a function at this stage facilitates subsequent stages of the model development and analysis, as these processes will all call the same model function, but pass different parameter values and/or calculate different final outcomes based on the model outputs. In the next part of this section, we will describe the code of the function.

```r
print.function(f.decision_model) # print the code of the function
```

```
## function (v.params)
## {
##     with(as.list(v.params), {
##         p.HDage <- 1 - exp(-v.r.asr[(n.age.init + 1) + 0:(n.t -
##             1)])
##         p.S1Dage <- 1 - exp(-v.r.asr[(n.age.init + 1) + 0:(n.t -
##             1)] * hr.S1)
##         p.S2Dage <- 1 - exp(-v.r.asr[(n.age.init + 1) + 0:(n.t -
##             1)] * hr.S2)
##         a.P <- array(0, dim = c(n.states, n.states, n.t), dimnames = list(v.n,
##             v.n, 0:(n.t - 1)))
##         a.P["H", "H", ] <- (1 - p.HDage) * (1 - p.HS1)
##         a.P["H", "S1", ] <- (1 - p.HDage) * p.HS1
##         a.P["H", "D", ] <- p.HDage
##         a.P["S1", "H", ] <- (1 - p.S1Dage) * p.S1H
##         a.P["S1", "S1", ] <- (1 - p.S1Dage) * (1 - (p.S1S2 +
##             p.S1H))
##         a.P["S1", "S2", ] <- (1 - p.S1Dage) * p.S1S2
##         a.P["S1", "D", ] <- p.S1Dage
##         a.P["S2", "S2", ] <- 1 - p.S2Dage
##         a.P["S2", "D", ] <- p.S2Dage
##         a.P["D", "D", ] <- 1
##         m.indices.notvalid <- arrayInd(which(a.P < 0 | a.P >
##             1), dim(a.P))
##         try(if (dim(m.indices.notvalid)[1] != 0) {
```

```
##             v.rows.notval <- rownames(a.P)[m.indices.notvalid[,
##                 1]]
##             v.cols.notval <- colnames(a.P)[m.indices.notvalid[,
##                 2]]
##             v.cycles.notval <- dimnames(a.P)[[3]][m.indices.notvalid[,
##                 3]]
##             df.notvalid <- data.frame(`Transition probabilities not valid:` = matrix(paste0(paste(v.r
##                 v.cols.notval, sep = "->"), "; at cycle ", v.cycles.notval),
##                 ncol = 1), check.names = FALSE)
##             message("Not valid transition probabilities")
##             stop(print(df.notvalid), call. = FALSE)
##         })
##         valid <- apply(a.P, 3, function(x) all.equal(sum(rowSums(x)),
##             n.states))
##         if (!isTRUE(all.equal(as.numeric(sum(valid)), as.numeric(n.t)))) {
##             stop("This is not a valid transition Matrix")
##         }
##         m.M <- matrix(0, nrow = (n.t + 1), ncol = n.states, dimnames = list(0:n.t,
##             v.n))
##         a.A <- matrix(0, nrow = (n.t + 1), ncol = n.states, dimnames = list(0:n.t,
##             v.n))
##         m.M[1, ] <- v.s.init
##         for (t in 1:n.t) {
##             m.M[t + 1, ] <- m.M[t, ] %*% a.P[, , t]
##         }
##         return(list(a.P = a.P, m.M = m.M))
##     })
## }
## <bytecode: 0x7fe47ecd8ee8>
```

The `f.decision_model` function is informed by the argument `v.params`. Via this argument we specify the parameter values. For our Sick-Sicker model, these parameters are stored in the dataframe `df.params`, which we passed into the function as shown below.

```r
l.out.stm <- f.decision_model(v.params = df.params.init) # run the function
```

You probably noticed that the prefix of the argument is telling us it has to be a vector, while we pass a dataframe into the function. This is totally fine. The function is able to deal with both vectors, usefull for small models, as well as dataframes, when you have more parameter info.

The functions starts with calculating the age-specific transition probabilities based on the rates in the data frame. This means that the parameters will become vectors of length `n.t`, describing the probability do die for all ages. The next part of the function, generates the age-specific transition probability matrices and stores them in the initiated array, `a.P`. The transition probability matrices a core component of a state-transition cohort model. This matrix contains the probabilities of transitioning from the current health state, indicated by the rows, towards the new health states, specified in the columns. More information about creating these matrices is described in a paper about state-transition models using R (F Alarid-Escudero et al. 2018). Because we have age-specific transition probabilities, the transition probability matrix is different each cycle. Since these probabilities are only depending on the age of the cohort, and not on other events, we can generate all matrices at the start of the model. This results in `n.t` matrices stored in the `a.P` array, of dimensions `n.s` x `n.s` x `n.t`. When running the model, we can index the correct transition probability matrix corresponding with the current age of the cohort. To make sure we are using correct values, the functions include some code to check if both the transition probabilities them self, as well as the transition probability matrices are valid. Below we print some time-points of the probability array to give you an impression about the resulting array. The values in the transition probability matrices show you the increased probabilities of transitioning to death form all health states towards the end.

```
## , , 0
##
##            H        S1        S2           D
## H  0.8491385 0.1498480 0.0000000 0.001013486
## S1 0.4984813 0.3938002 0.1046811 0.003037378
## S2 0.0000000 0.0000000 0.9899112 0.010088764
## D  0.0000000 0.0000000 0.0000000 1.000000000
##
## , , 1
##
##            H        S1        S2            D
## H  0.8491513 0.1498502 0.0000000 0.0009985012
## S1 0.4985037 0.3938180 0.1046858 0.0029925135
## S2 0.0000000 0.0000000 0.9900597 0.0099402657
## D  0.0000000 0.0000000 0.0000000 1.0000000000
##
## , , 2
##
##            H        S1        S2           D
## H  0.8490910 0.1498396 0.0000000 0.001069428
## S1 0.4983976 0.3937341 0.1046635 0.003204853
## S2 0.0000000 0.0000000 0.9893570 0.010642959
## D  0.0000000 0.0000000 0.0000000 1.000000000
```

```
##            H        S1         S2         D
## H  0.6055199 0.1068564 0.00000000 0.2876237
## S1 0.1807584 0.1427991 0.03795926 0.6384833
## S2 0.0000000 0.0000000 0.03365849 0.9663415
## D  0.0000000 0.0000000 0.00000000 1.0000000
```

After the array is filled, the cohort trace matrix, `m.M`, of dimensions `n.t` x `n.s` is initiated. This matrix will store the state occupation at each point in time. The first row of the matrix is informed by the initial state vector `v.s.init`. For the remaining points in time, we iteratively multiply the cohort trace with the age-specific transition probability matrix indexed from array `a.P`. The model results are stored in a list, called `l.out.stm`. This list contains the array of the transition probability matrix for all cycles `t` and the cohort trace `m.M`.

```
head(l.out.stm$m.M)     # show the top part of the cohort trace
```

```
##          H        S1         S2           D
## 0 1.0000000 0.0000000 0.00000000 0.000000000
## 1 0.8491385 0.1498480 0.00000000 0.001013486
## 2 0.7957468 0.1862564 0.01568695 0.002309774
## 3 0.7684912 0.1925699 0.03501425 0.003924648
## 4 0.7484793 0.1909659 0.05478971 0.005765035
## 5 0.7306193 0.1873106 0.07413838 0.007931783
```

```
tail(l.out.stm$m.M)     # show the bottom part of the cohort trace
```

```
##             H           S1           S2         D
## 70 0.009928317 0.0022433565 2.035951e-04 0.9876247
## 71 0.007153415 0.0015935925 1.311619e-04 0.9911218
## 72 0.005058845 0.0011174473 8.674540e-05 0.9937370
## 73 0.003460336 0.0007552206 5.436484e-05 0.9957301
## 74 0.002289594 0.0004937081 3.298632e-05 0.9971837
## 75 0.001475636 0.0003151589 1.985106e-05 0.9981894
```

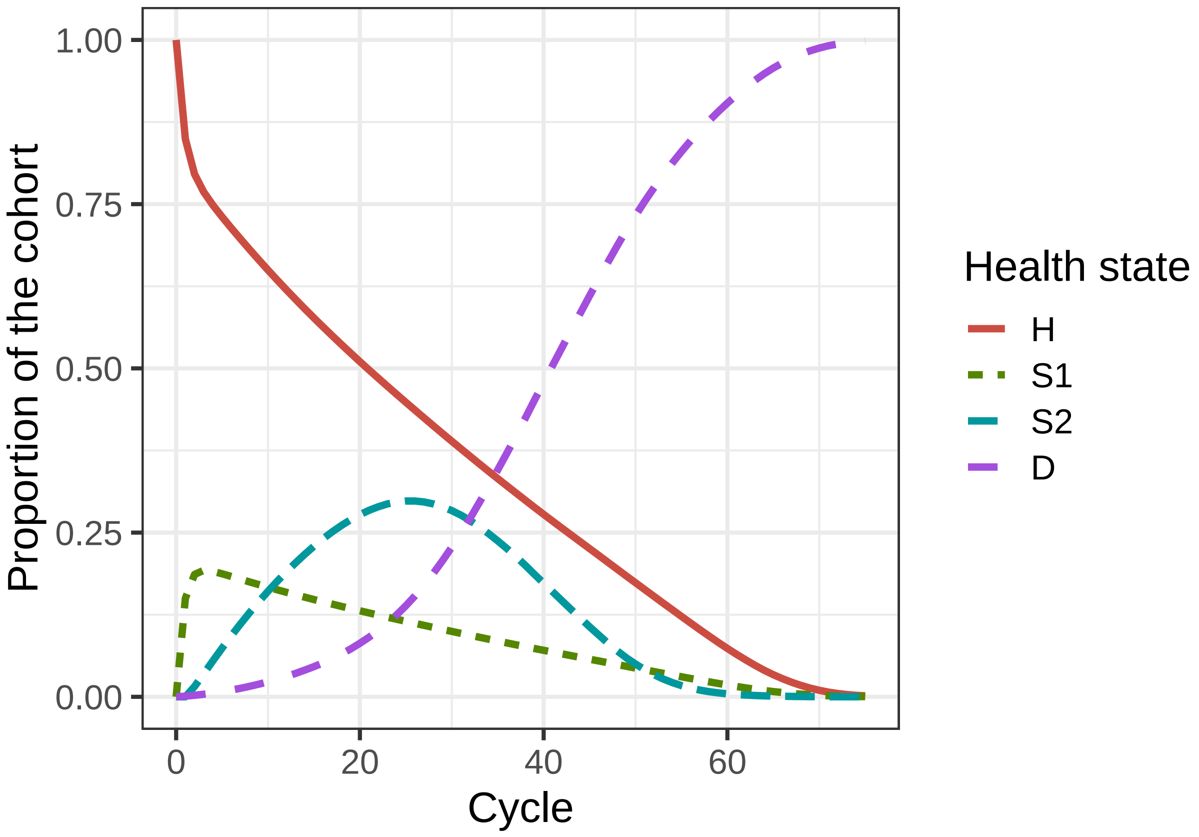


Figure 2: Cohort trace of the Sick-Sicker cohort model

Via the code below, we can graphically show the model dynamics by plotting the cohort trace. This figure shows the distribution of the cohort among the different health states at each time point.

03 Model calibration

In this component, unknown or highly uncertain model parameters are estimated by calibrating model outputs to match specified calibration targets. This proces is executed by the **03_calibration.R** file. The target data is stored in the **03_calibration-targets.RData** file. The **03_calibration.R** file include much more code compared to the `R` scripts of the previous components. Like in compontent 02, we start with loading inputs and functions. In addition, we load the calibration target data. Next, in section 03.2 Visualize targets, we create a graph for each of our targets, survival, prevalence and the proportion who are Sicker, among all those afflicted (Sick+Sicker).

In section 03.3 Run calibration algorithms, we set the parameters we like to calibrate to fixed values and test if our function performing the calibrations, called `f.calibration_out`, works.

```r
print.function(f.calibration_out) # print the funtions
```

```
## function (v.params.calib)
## {
##     v.params <- df.params.init
##     v.params["p.S1S2"] <- v.params.calib["p.S1S2"]
##     v.params["hr.S1"] <- v.params.calib["hr.S1"]
##     v.params["hr.S2"] <- v.params.calib["hr.S2"]
##     l.out.stm <- f.decision_model(v.params = v.params)
##     v.os <- 1 - l.out.stm$m.M[, "D"]
##     v.prev <- rowSums(l.out.stm$m.M[, c("S1", "S2")])/v.os
##     v.prop.S2 <- l.out.stm$m.M[, "S2"]/rowSums(l.out.stm$m.M[,
##         c("S1", "S2")])
##     l.out <- list(Surv = v.os[c(11, 21, 31)], Prev = v.prev[c(11,
##         21, 31)], PropSicker = v.prop.S2[c(11, 21, 31)])
##     return(l.out)
## }
```

This function is informed by the argument `v.params.calib`. This vector contains the values of the three parameters of interest. The placeholder values are replaced by these values and with these values the model is evaluated. This is done by running the `f.decision_model` function, described in component 02. This results in a new list with output of the model. With this new output, the epidemiological output, overall survial, disease prevalence and the proportion of Sicker in the Sick and Sicker state are calculated. The estimated values for these epidemiological outcomes at the timepoints t=11, t=21 and t=31, are combined in a list, called `l.out`. This is the output from the `f.calibration_out` function.

Now we know our function works, we specify our calibration parameters in section 03.3.1. This includes setting the seed, specifying the number of random samples, the name of the input paramters and most important, the range on input search spaces and the name of the calibration targets, `Surv`, `Prev`, `PropSick`.

In the next section of this component to calibrate the Sick-Sicker model, section 03.3.2, we use a Bayesian approach using the incremental mixture importance sampling (IMIS) algorithm [Teele2006], which has been used to calibrate health policy models (Raftery, A., Bao 2010, Menzies, Pandya, and Kim (2017), Rutter2018). Bayesian methods allow us to quantify the uncertainty in the calibrated parameters even in the presence of non-identifiability (Fernando Alarid-Escudero et al. 2018). We assumed a normal likelihood and uniform priors. For a more detailed description of IMIS for Bayesian calibration, different likelihood functions and prior distributions, we refer the reader to the tutorial for Bayesian calibration by Menzies et al. (Menzies, Pandya, and Kim 2017). We use the `IMIS` function from the likenamed package that is calling the functions `f.log_lik` to draw samples from the the posterior distribution of multivariate variable. (Raftery and Le Bao 2012) This IMIS function is using the functions `likelihood`, `sample.prior`, `f.log_prior` and `prior`, that are specified in the **03__calibration__functions.R** file. We respectively specify the incremental sample size at each iteration of IMIS, the desired posterior sample size at the resample stage, the maximum number of iterations in IMIS and the number of optimizers which could be 0. The function returns a list, which we called `l.fit.imis`, with the posterior resamples, the diagnostic statistics at each IMIS iteration and the centers of Gaussian components. We store the posterior resamples in the matrix `m.calib.post`.

In the next section, 03.4, we explore the posterior distributions. From these distributions, we compute the posterior mean, median and 95% credible interval, mode and maximum-a-posteriori (MAP). All for these summary statistics are combined in a data frame called `df.posterior.summ`.

Interpreting these statistics can be quite hard and is easier via a graph. In section 03.4.2 we visualize the posterior distributions. These figures are save in the *figures* folder and shown below.

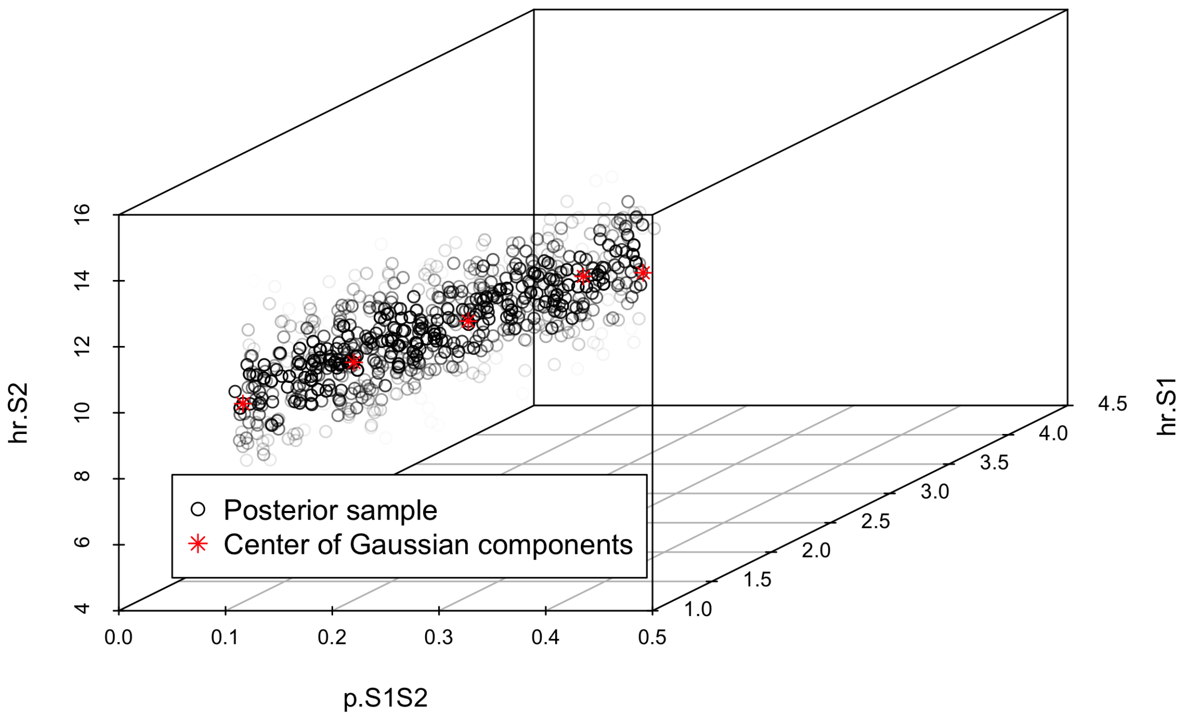
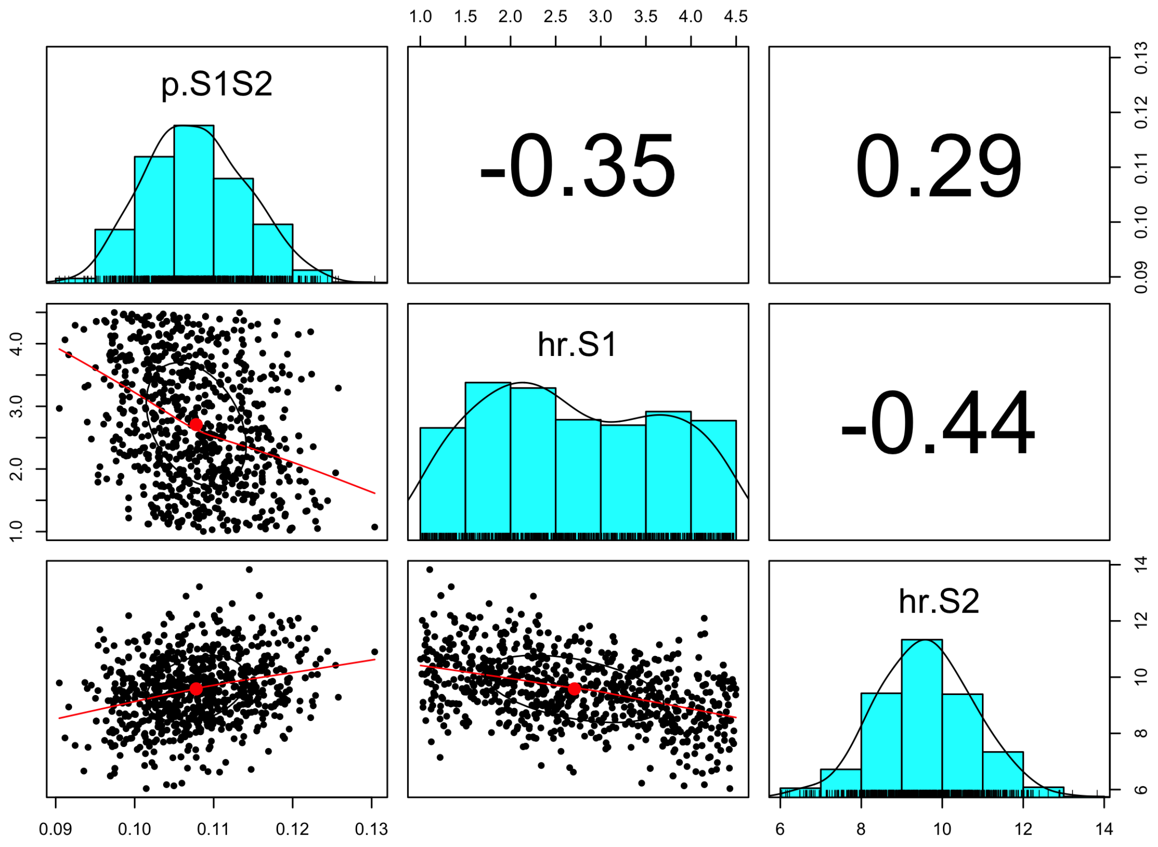


Figure 3: Posterior distribution joint



[HOW DO WE INTERPRET THESE FIGURES?]

Finally, the posterior and MAP from the IMIS calibration are stored in the file *03_imis-output.RData*. Storing this data as a datafile gives us the possibility to inport the data without re-running the calibration code.

For illustration purposes, we also provide code to calibrated the Sick-Sicker model with the Nelder-Mead algorithm (Nelder and Mead 1965), which was initialized at 100 different starting points sampled from the prior distributions of the calibrated parameters. This calibration exercise can be found in the file `app2_calibration-nelder-mead.R`, but we will not go into details for this file.

04 Validation

In this section, we like to check the internal validity of our Sick-Sicker model before we continue with the analysis. Checking the internal validity means that we need to make sure that the output corresponds with our input. For example, we like to check that Sicker individuals can not recover and that most of our cohort died by the end of the time horizon. We also plot our model-predicted outputs against the calibration targets. Therefore, the **04_validation.R** file stats with sourcing all previously described functions and generated calibration data.

We compute the model-predicted outputs for each sample of posterior distribution as well as for the MAP estimate. Section 04.5 is computing these model-predicted outputs for each sample using the function `f.data_summary`.

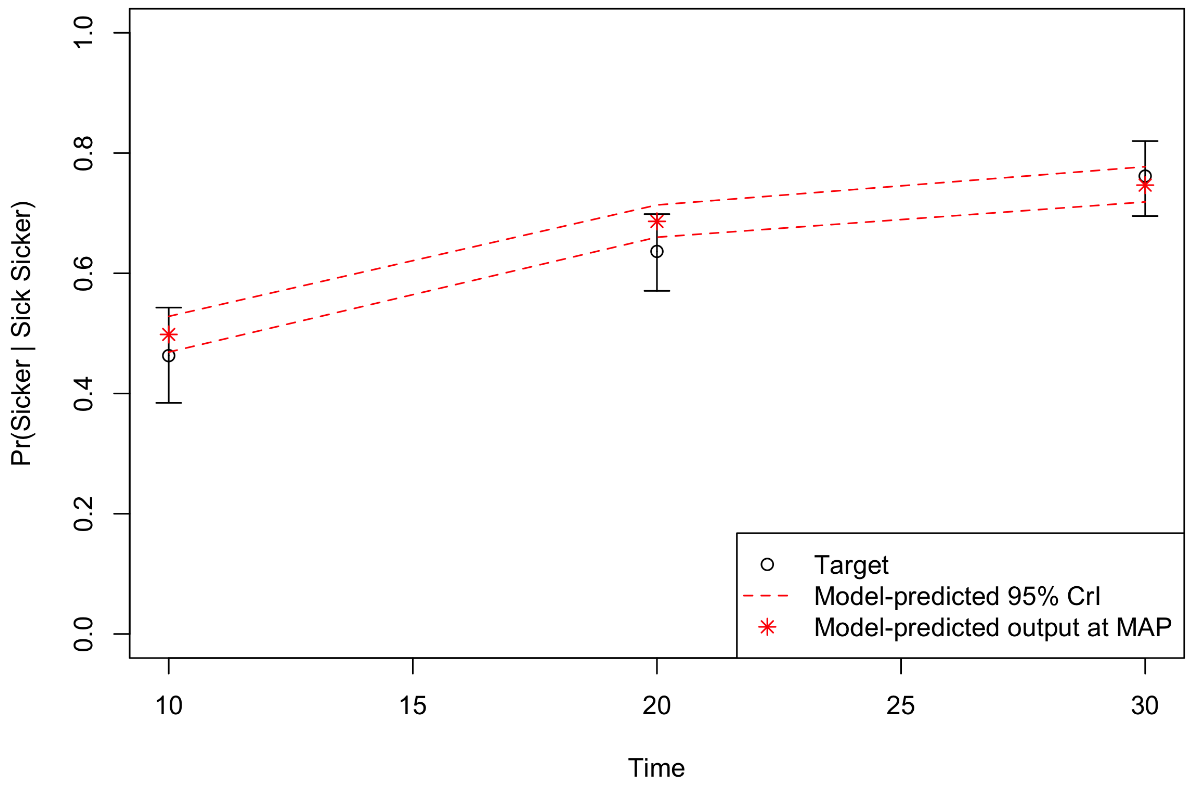
```
print.function(f.data_summary)
```

```
## function (data, varname, groupnames)
## {
##     require(plyr)
##     summary_func <- function(x, col) {
##         c(mean = mean(x[[col]], na.rm = TRUE), median = quantile(x[[col]],
##             probs = 0.5, names = FALSE), sd = sd(x[[col]], na.rm = TRUE),
##             lb = quantile(x[[col]], probs = 0.025, names = FALSE),
##             ub = quantile(x[[col]], probs = 0.975, names = FALSE))
##     }
##     data_sum <- ddply(data, groupnames, .fun = summary_func,
##         varname)
##     data_sum <- plyr::rename(data_sum, c(mean = varname))
##     return(data_sum)
## }
## <bytecode: 0x7fe47ea6ac68>
```

This function is informed by three arguments, `data`, `varname` and `groupnames`.

The computation of the model-predicted outputs using the MAP estimates is done by inserting the `v.calib.post.map` data into the previously described `f.calibration_out` function. This function creates a list including the estimated values for survival, prevalence and the proportion of sicker individuals at the time points 10, 20 and 30.

In sections 04.6 Internal validation: Model-predicted outputs vs. targets we check the internal validation of our model-predicted outputs versus our targets using plots. The generated plots are saved as .png files, which in turn can be used without the need of re-running the code.



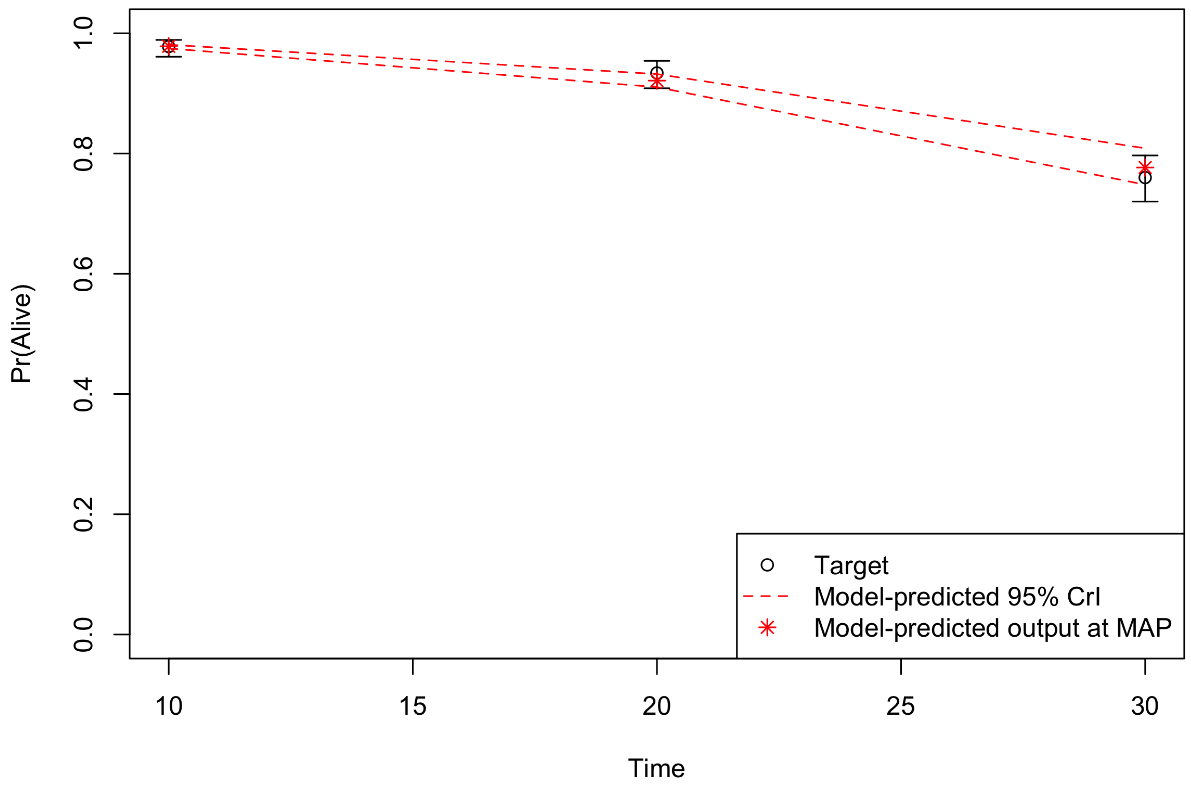


Figure 4: Survival data: Posterior vs targets

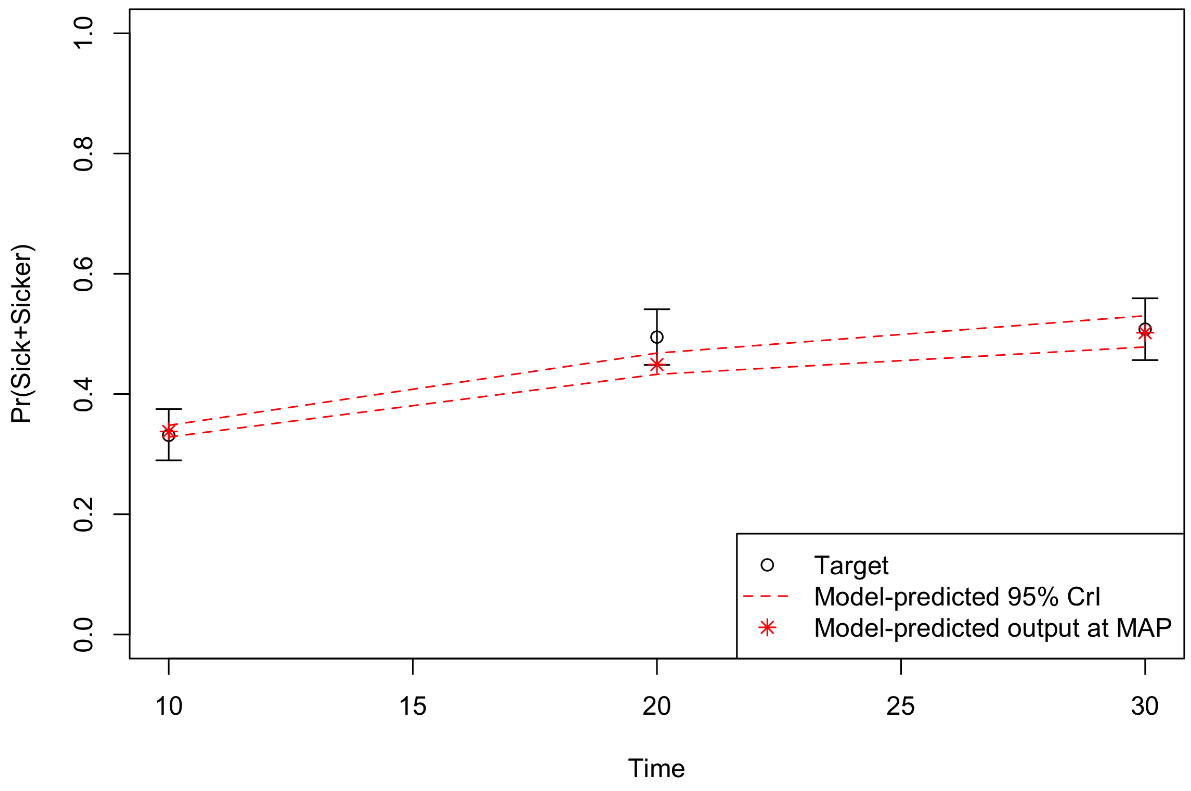


Figure 5: Prevalence data: Posterior vs targets

[Do we need an interpretation of the graphs?]

Alarid-Escudero, F, EM Krijkamp, P Pechlivanoglou, EA Enns, MGM Hunink, and H Jalal. 2018. "State-transition cohort models in R, from conceptualization to implementation: A tutorial." *Medical Decision Making : An International Journal of the Society for Medical Decision Making* XX (X). SAGE PublicationsSage CA: Los Angeles, CA: XX. http://journals.sagepub.com/doi/10.1177/0272989X16686559 http://www.ncbi.nlm.nih.gov/pubmed/28061043.

Alarid-Escudero, Fernando, Richard F. MacLehose, Yadira Peralta, Karen M. Kuntz, and Eva A. Enns. 2018. "Nonidentifiability in Model Calibration and Implications for Medical Decision Making." *Medical Decision Making* 38 (7): 810–21. doi:10.1177/0272989X18792283.

Enns, EA, LE Cipriano, CT Simons, and CY Kong. 2015. "Identifying Best-Fitting Inputs in Health-Economic Model Calibration: A Pareto Frontier Approach." *Medical Decision Making* 35 (2): 170–82. doi:10.1177/0272989X14528382.

Menzies, Nicolas A, Ankur Pandya, and Jane J Kim. 2017. "HHS Public Access." *Pharmacoeconomics* 35 (6): 613–24. doi:10.1007/s40273-017-0494-4.Bayesian.

Nelder, J. A., and R. Mead. 1965. "A Simplex Method for Function Minimization." *The Computer Journal* 7 (4): 308–13. doi:10.1093/comjnl/7.4.308.

Raftery, Adrian, and Le Bao. 2012. *IMIS: Increamental Mixture Importance Sampling.* https://CRAN.R-project.org/package=IMIS.

Raftery, A., Bao, L. 2010. "Estimating and Projecting Trends in HIV/AIDS Generalized Epidemics Using Incremental Mixture Importance Sampling." *Biometrics* 66 (4): 1162–73.