# The SEITL model

## Objectives

The aim of this session is to familiarise yourselves with the SEITL model. This model has been proposed as a mechanistic explanation for a two-wave influenza A/H3N2 epidemic that occurred on the remote island of Tristan da Cunha in 1971. Given the small population size of this island (284 habs), demographic stochasticity is expected to play a significant role in the dynamics of the epidemic so that the use of a stochastic model is recommended. However, as you will see in the next lecture, fitting a stochastic model is computationally much more intensive than fitting a deterministic model. We will therefore close this session by fitting the deterministic SEITL model. Actually, this will be useful to prepare the next session, when you will fit the stochastic model.

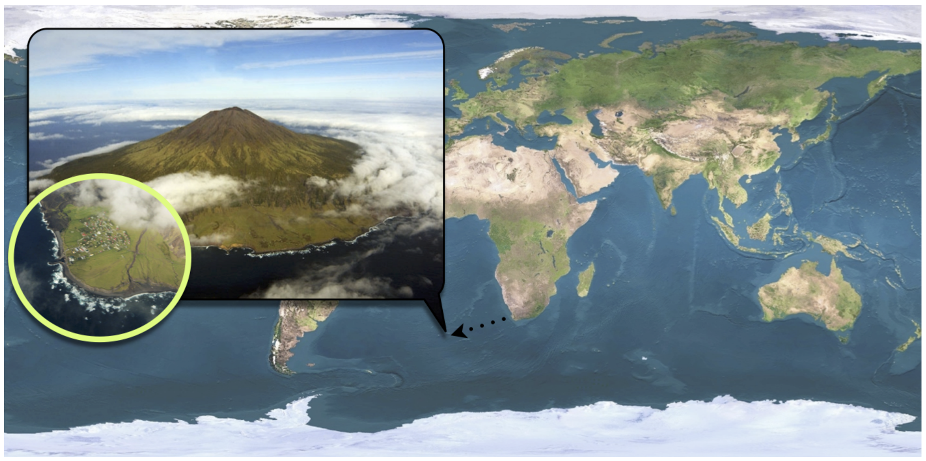
In this session you will

1. Familiarise yourself with the structure and parameters of the SEITL model
2. Explore the role of demographic stochasticity in the dynamics of the model
3. Fit the deterministic SEITL model using MCMC

But first of all, let's have a look at the data.

## Tristan da Cunha outbreak

Tristan da Cunha is a volcanic island in the South Atlantic Ocean. It has been inhabited since the $19^{th}$ century and in 1971, the 284 islanders were living in the single village of the island: Edinburgh of the Seven Seas. Whereas the internal contacts were typical of close-knit village communities, contacts with the outside world were infrequent and mostly due to fishing vessels that occasionally took passengers to or from the island. These ships were often the cause of introduction of new diseases into the population (Samuels1963). Focusing on influenza, no epidemic has been reported since an epidemic of A/H1N1 in 1954. In this context of a small population with limited immunity against influenza, an unusual epidemic occurred in 1971, 3 years after the global emergence of the new subtype A/H3N2.



On August 13, a ship returning from Cape Town landed five islanders on Tristan da Cunha. Three of them developed acute respiratory disease during the 8-day voyage and the other two presented similar symptoms immediately after landing. Various family gatherings welcomed their disembarkation and in the ensuing days, an epidemic started to spread rapidly throughout the whole island population. After three weeks of propagation, while the epidemic was declining, some islanders developed second attacks and a second peak of cases was recorded. The epidemic faded out after this second wave and lasted a total of 59 days.

Among the 284 islanders, 273 (96%) experienced at least one attack and 92 (32%) experienced two attacks. Unfortunately, only 312 of the 365 attacks (85%) are known to within a single day of accuracy and constitute the dataset reported by Mantle & Tyrrell in 1973 (ref:Mantle1973}.

The dataset of daily incidence can be loaded and plotted as follows:

```r
data(FluTdC1971)
head(FluTdC1971)
```

```
##         date time obs
## 1 1971-08-13    1   0
## 2 1971-08-14    2   1
## 3 1971-08-15    3   0
## 4 1971-08-16    4  10
## 5 1971-08-17    5   6
## 6 1971-08-18    6  32
```

```r
ggplot(data = FluTdC1971, aes(x = date, y = obs)) + geom_bar(stat = "identity") +
    theme_bw()
```

## SEITL model

One possible explanation for the rapid reinfections reported during this two-wave outbreak is that following recovery from a first infection, some islanders did not develop a long-term protective immunity and remained fully susceptible to reinfection by the same influenza strain that was still circulating. This can be modelled as follows:

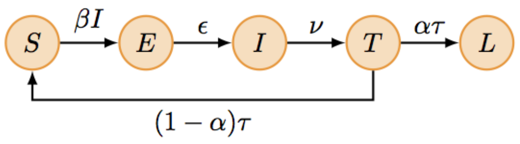


Figure 1: The SEITL model extends the classical SEIR model to account for the dynamics and host heterogeneity of the immune response among the islanders. Following recovery, hosts remain temporarily protected against reinfection thanks to the cellular response. Accordingly, they enter the T stage (temporary protection). Then, following down-regulation of the cellular response, the humoral response has a probability $\alpha$ to reach a level sufficient to protect against reinfection. In this case, recovered hosts enter the L stage (long-term protection) but otherwise they remain unprotected and re-enter the susceptible pool (S).

The SEITL model can be described with five states (S, E, I, T and L) and five parameters:

1. basic reproductive number ($R_0$)
2. latent period ($D_{\text{lat}}$)
3. infectious period ($D_{\text{inf}}$)
4. temporary-immune period ($D_{\text{imm}}$)
5. probability of developing a long-term protection ($\alpha$).

and the following deterministic equations:

$$\begin{cases} \frac{dS}{dt} = -\beta S \frac{I}{N} + (1-\alpha)\tau T \\ \frac{dE}{dt} = \beta S \frac{I}{N} - \epsilon E \\ \frac{dI}{dt} = \epsilon E - \nu I \\ \frac{dT}{dt} = \nu I - \tau T \\ \frac{dL}{dt} = \alpha \tau T \end{cases}$$

where $\beta = R_0/D_{\text{inf}}$, $\epsilon = 1/D_{\text{lat}}$, $\nu = 1/D_{\text{inf}}$, $\tau = 1/D_{\text{imm}}$ and $N = S + I + R$ is constant.

However, in order to account for the type of data reported during the outbreak we need to add one state variable and one parameter to the model:

- The dataset represents daily incidence counts: we need to create a 6th state variable - called Inc - to track the number of daily new cases. Assuming that new cases are reported when they become symptomatic and infectious, we have the following equation for the new state variable:

$$\frac{d\text{Inc}}{dt} = \epsilon E$$

- The dataset is incomplete: only 85% of the cases were reported. In addition, we need to account for potential under-reporting of asymptomatic cases. We assume that the data were reported according to a Poisson process with reporting rate $\rho$. Since this reporting rate is unknown (we can only presume that it should be below 85%) we will include it as an additional parameter.

The deterministic SEITL model is already implemented as a `fitmodel` object, which can be loaded into your **R** session by typing:

```
example(SEITL_deter)
```

**Take 5 min** to look at the different elements of the model. In particular, you might be interested in how the daily incidence is computed in the function `SEITL_deter$simulate` and how the likelihood function `SEITL_deter$pointLogLike` accounts for under-reporting.

## Deterministic vs Stochastic simulations

### Deterministic simulations

Now, let's assess whether the deterministic SEITL model can reproduce the two-wave outbreak of Tristan da Cunha. You can start simulating the model using guess values for the initial state and the parameters. To give you some hints, here is a summary of information on influenza found in the literature:

1. The $R_0$ of influenza is commonly estimated around 2. However, it can be significantly larger ($R_0 > 10$) in close knit communities due to their exceptional contact rates.

2. Both the average latent and infectious periods for influenza have been estimated around 2 days each.
3. The down-regulation of the cellular response is completed 15 days after symptom onset on average.
4. Serological surveys have shown that the seroconversion rate to influenza (probability to develop specific antibodies) is around 80%. However, it is likely that **not** all seroconvereted individuals acquire a long-term protective humoral immunity.
5. Between 20 and 30% of the infection with influenza are asymptomatic.
6. There is very limited cross-immunity between influenza viruses A/H1N1 and A/H3N2.

Based on these informations and the description of the outbreak, propose one or more set(s) of values for the parameters (`theta`) and initial state (`state.init`) of the model? You can also use our 3 sets of values that can be loaded by typing `example(SEITL_guess_values)`.

In the previous practical you have simulated trajectories using `genObsTraj` and then plot them with `plotTraj`. Actually, the function `plotFit` combine these two steps by simulating and displaying model trajectories against the data:

```
theta.bad.guess <- c(R0 = 2, D.lat = 2, D.inf = 2, alpha = 0.9, D.imm = 13,
    rho = 0.85)
state.init.bad.guess <- c(S = 250, E = 0, I = 4, T = 0, L = 30, Inc = 0)
plotFit(SEITL_deter, theta.bad.guess, state.init.bad.guess, data = FluTdC1971)
```

```
## Error: inherits(fitmodel, "fitmodel") is not TRUE
```

You can also display all states variables (not only the observation) by passing `all.vars=TRUE`

```
plotFit(SEITL_deter, theta.bad.guess, state.init.bad.guess, data = FluTdC1971,
    all.vars = TRUE)
```

Although the simulation of the trajectory is deterministic, the observation process is stochastic, hence the noisy `obs` time-series. You can appreciate the variability of the observation process by plotting several replicates (use the argument `n.replicates`):

```
plotFit(SEITL_deter, theta.bad.guess, state.init.bad.guess, data = FluTdC1971,
    n.replicates = 100)
```

By default, this function plots the mean, median as well as the 95th and 50th percentiles of the replicated simulations. Alternatively, you can visualize all the simulated trajectories by passing `summary=FALSE`:

```
plotFit(SEITL_deter, theta.bad.guess, state.init.bad.guess, data = FluTdC1971,
    n.replicates = 100, summary = FALSE)
```

Now, **take 10 min** to get yourself familiar with the function `plotFit` and explore the dynamics of your model for different parameter and initial state values. In particular, try different values for $R_0 \in [2-15]$ and $\alpha \in [0.3-1]$. For which values of $R_0$ and $\alpha$ do you get a descent fit?

If you didn't get a descent fit, try `theta.guess3` and `state.init.guess3` that you can load by typing `example(SEITL_guess_values)`.

Once you have a descent fit, you should note the low prevalence between the two waves. This suggests that the continuous approximation made by the deterministic model might not be appropriate and that we should use a stochastic model. In contrast to deterministic models, stochastic models explicitly take into account the discrete nature of individuals.

## Stochastic simulations

The stochastic SEITL model is already implemented as a `fitmodel` object, which can be loaded into your **R** session by typing:

```
example(SEITL_sto)
```

As you can read from the loading messages, `SEITL_sto` mainly differs from `SEITL_deter` through the `simulate` function, which replaces the deterministic equations solver by a stochastic simulation algorithm. More precisely, this algorithm takes a list of transitions (`SEITL_transitions`) and a function to compute the transition rate (`SEITL_rateFunc`). **Take 5 min** to have a look at the function `SEITL_sto$simulate` and make sure you understand all the transitions and rates.

If you are curious about how the stochastic model is simulated, you can have a look at the code of the function `simulateModelStochastic`. You will note that it calls the function `ssa.adaptivetau` of the **R** package `adaptivetau`, and then process the returned data frame in order to extract the state of the model at the desired observation times given by `times`.

Then **take 10 min** to explore the dynamics of the stochastic SEITL model with the function `plotFit`. Note that `SEITL_sto` has the same `theta.names` and `state.names` as `SEITL_deter`, which means that you can use the same parameters and initial state vectors as in the previous section. In addition, you can plot the time-series of the proportion of faded out simulations by passing `p.extinction=TRUE` to `plotFit`.

What differences do you notice between stochastic and deterministic simulations? Conclude on the role of demographic stochasticity on the dynamics of the SEITL model.

## Exponential vs Erlang distributions

So far, we have assumed that the time spent in each compartment was following an exponential distribution. This distribution has the well known property of being memoryless, which means that the probability of leaving a compartment doesn't depend on the time already spent in this compartment. Although mathematically convenient, this property is not realistic for many biological processes such as the contraction of the cellular response. In order to include a memory effect, it is common to replace the exponential distribution by an Erlang distribution. This distribution is parametrised by its mean $m$ and shape $k$ and can be modelled by $k$ consecutive sub-stages, each being exponentially distributed with mean $m/k$. As illustrated below the flexibility of the Erlang distribution ranges from the exponential ($k = 1$) to Gaussian-like ($k >> 1$) distributions.

We can extend the SEITL as follows:

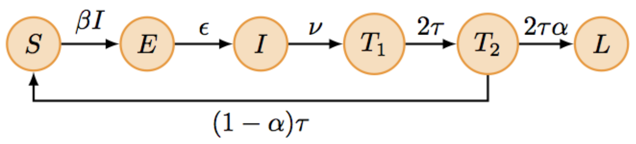


Figure 2: The SEIT2L model extends the SEITL model to account for memory effect in the contraction of the cellular response. The time spend in the $T$ compartment (temporary protection) follow an Erlang distribution with mean $D_{\text{imm}} = 1/\tau$ and shape equal to 2.

The deterministic and stochastic SEIT2L models are already implemented as `fitmodel` objects, which can be loaded into your **R** session by typing:

```
example(SEIT2L_deter)
example(SEIT2L_sto)
```

**Take 5 min** to have a look at the function `simulate` of these SEIT2L models and make sure you understand how the Erlang distribution for the $T$ compartment is coded.

Now, **take 10 min** to compare the dynamics of the SEITL and SEIT2L model using `plotFit`. Use your best guess from the previous exercise as well as stochastic and deterministic simulations. Note that although SEITL and SEIT2L share the same parameters, their state variables differ so you need to modify the `state.init` of SEIT2L accordingly.

Can you notice any differences? If so, which model seems to provide the best fit? Do you understand why the shape of the epidemic changes as you change the distribution of the $T$ compartment?

## Fitting the deterministic models

As you will see in the next session, fitting a stochastic model is computationally much more intensive than fitting a deterministic model. This can lead to a waste of time and computational resources:

1. If you initialise the MCMC with a `theta.init` far from the region of high posterior density, the chain might take a long time to reach this region and you will have to burn a lot of iterations.
2. If the covariance matrix of the Gaussian proposal is very different from the posterior, this will result in poor mixing and sub-optimal acceptance rate and you will have to thin a lot your chain.

In this context, it might be useful to run a MCMC on the deterministic model first, which is fast, and then learn from the output of this chain to initialise the chain for the fit of the stochastic model. The rational for this approach is that the deterministic model is an approximation of the stochastic model and should capture, in most cases, the same dynamics.

So the objective of the last part of this session is to fit the deterministic SEITL and SEIT2L models to the Tristan da Cunha outbreak. This will prepare the next session and enable us to compare both models and assess whether one is significantly better than the other.

To save time, half of the group will fit the deterministic SEITL model and the other half will fit the deterministic SEIT2L model. In the rest of the session, most of the example will refer to the SEITL, the same command work for the SEIT2L model by adjusting the name of the command (i.e. 'example(SEIT2L_deter)' instead of 'example(SEITL_deter)').

### Run a MCMC

Here you could use the function `my_mcmcMH` that you have already coded but it might result in poor acceptance rates since the models have 6 parameters to be estimated. Accordingly, using the adaptive MCMC might be a better choice.

**Take 5 min** to have a look at the adaptive MCMC implemented in the function `mcmcMH`. As you can see, this function takes similar arguments as your function `my_mcmcMH` (`target`, `theta.init`, `proposal.sd` and `n.iterations`) plus several new ones that control the adaptive part of the algorithm and that you can learn about via the help page of `mcmcMH`. Note also that this function returns a list that contains, in addition to the trace, the acceptance rate and the empirical covariance matrix of the posterior. The latter will be useful to improve the covariance matrix of the Gaussian proposal when we will fit the stochastic model in the next session.

The next step is to write a wrapper function to evaluate the posterior at a given `theta` value. Here again, you could wrap the function `my_posterior` that you have already coded and pass it to `mcmcMH` so that it returns a sample of `theta` from the posterior distribution. However, as you will see at the end of this session, in order to be able to compare different models we also need to track the log-likelihood of the sampled `theta`s. Although this could be done by running `trajLogLike` on each returned `theta`, you might remember that the log-likelihood is actually computed in `my_posterior` so we could just use it. This is exactly what the function `logPosterior` does for you.

**Take 5 min** to look at the code of `logPosterior`. As you can see, this function takes similar arguments as your function `my_posterior` (`fitmodel`, `theta`, `state.init` and `data`) plus one called `margLogLike`, which is a function to compute the log-likelihood of `theta`. By default, it is set to `trajLogLike` so you don't really need to bother with this argument (actually it will be useful in the next session as we will need a different function than `trajLogLike` to compute the log-likelihood). Note also that this function returns a list of two elements:

1. `log.density`: the log of the posterior density
2. `trace`: a vector that contains, among other things, `theta` and its `log.likelihood`. All these information will be added to the `trace` data frame returned by `mcmcMH`.

Now, **take 15 min** to prepare all the inputs to be able to run `mcmcMH` to fit your model. You should proceed as follows:

```r
# the fitmodel
example(SEITL_deter)

# wrapper for posterior
my_posteriorTdC <- function(theta) {

    my_fitmodel <- SEITL_deter
    my_state.init <- c(S = 279, E = 0, I = 2, T = 3, L = 0, Inc = 0)

    return(logPosterior(fitmodel = my_fitmodel, theta = theta, state.init = my_state.init,
        data = FluTdC1971, margLogLike = trajLogLike))

}

# theta to initialise the MCMC
theta.init <- c(R0 = 2, D.lat = 2, D.inf = 2, alpha = 0.8, D.imm = 16, rho = 0.85)

# diagonal elements of the covariance matrix for the Gaussian proposal
```

```
proposal.sd <- c(R0 = 1, D.lat = 0.5, D.inf = 0.5, alpha = 0.1, D.imm = 2, rho = 0.1)

# lower and upper limits of each parameter
lower <- c(R0 = 0, D.lat = 0, D.inf = 0, alpha = 0, D.imm = 0, rho = 0)
upper <- c(R0 = Inf, D.lat = Inf, D.inf = Inf, alpha = 1, D.imm = Inf, rho = 1)

# number of iterations for the MCMC
n.iterations <- 5000

# additional parameters for the adaptive MCMC, see ?mcmcMH for more details
adapt.size.start <- 100
adapt.size.cooling <- 0.999
adapt.shape.start <- 200
```

If you have trouble filling some of the empty bits, have a look at our example.

Then you should be able to run `mcmcMH`:

```
# run the MCMC
my_mcmc.TdC <- mcmcMH(target = my_posteriorTdC, theta.init = theta.init, proposal.sd = propo
    limits = list(lower = lower, upper = upper), n.iterations = n.iterations,
    adapt.size.start = adapt.size.start, adapt.size.cooling = adapt.size.cooling,
    adapt.shape.start = adapt.shape.start)
```

Which should print some informations as the chain runs: acceptance rate, state of the chain, log-likelihood etc. In particular, what can you say about the evolution of the acceptance rate?

Note that we have set-up the number of iterations to 5000. This is a benchmark and if your laptop is quite slow you might want to perform less iterations. Here the objective is to have a short - preliminary - run to calibrate your adaptive parameters before running a longer chain.

**Take 10 min** to change the parameters controlling the adaptive part of the MCMC and look at the effect on the acceptance rate. Try to find a "good" combination of these parameters so that the acceptance rate is near the optimal value of 23% (actually, the algorithm efficiency remains high whenever the acceptance rate is between about 0.1 and 0.6 so any value in between is OK). If you can't find one, look at our example.

## Short run analysis

Now it's time to use what you've learned in the previous session to analyse your MCMC outputs using the `coda` package. If you didn't manage to run `mcmcMH`, you can use the results from our example.

Note that because there are more than 6 parameters to look at, the `coda` functions used previously to plot the traces, densities and autocorrelations might not be optimal for laptop screens (the axis labels takes too much space). Fortunately, `coda` has another set of functions to make the plots more compacts:

- `xyplot` for the trace.
- `densityplot` for the density.
- `acfplot` for the autocorrelation.

You can find a grouped documentation for these three functions by typing `?acfplot`.

**Take 10 min** to analyse the trace returned by `mcmcMH`. Remember that, with the notation above, the trace can be accessed by `my_mcmc.TdC$trace` and then converted to a `mcmc` object so that the `coda` functions recognize it:

```r
# convert to mcmc object
my_trace <- mcmc(my_mcmc.TdC$trace)
# plot the trace
xyplot(my_trace)
```

Try to determine what burning and thining would be appropriate for your trace. If you are not sure about how to analyse your trace, have a look at our example.

## Long run analysis

You should have noticed that the effective sample size (ESS) is quite small ($< 100$) for your preliminary run. By contrast, the ESS was much higher for the SIR model of the previous session with a similar number of iterations. However, this model has only 2 parameters against 6 for the SEITL and SEIT2L models. Intuitively, the more parameter you have the bigger is the parameter space and the longer it takes to the MCMC algorithm to explore it (**some theoretical results?**). This is why we need to run a much longer chain to achieve a descent ESS ($~1000).

To save time, we have run 2 chains of 100 000 iterations starting from different initial `theta` values:

```r
theta1 = c(R0 = 2, D.lat = 2, D.inf = 2, alpha = 0.8, D.imm = 16, rho = 0.85)
theta2 = c(R0 = 20, D.lat = 2, D.inf = 2, alpha = 0.1, D.imm = 8, rho = 0.3)
```

Each chain took 5 hours on a scientific computing cluster and can be loaded as follows:

```
data(mcmc_TdC_deter_longRun)
# this should load 2 objects in your environment: mcmc_SEITL_theta1 and
# mcmc_SEITL_theta2. Each one is a list of 3 elements returned by mcmcMH
names(mcmc_SEITL_theta1)
## [1] "trace"             "acceptance.rate"   "covmat.empirical"
# the trace contain 9 variables for 100000 iterations
dim(mcmc_SEITL_theta1$trace)
## [1] 100001      9
# let's have a look at it
head(mcmc_SEITL_theta1$trace)
##      R0 D.lat D.inf  alpha D.imm    rho log.prior log.likelihood
## 1 2.000 2.000 2.000 0.8000 16.00 0.8500    -12.81         -445.8
## 2 2.000 2.000 2.000 0.8000 16.00 0.8500    -12.81         -445.8
## 3 3.291 1.928 2.140 0.8244 19.15 0.8684    -12.81         -442.7
## 4 3.644 1.868 1.899 0.8099 18.47 0.8769    -12.81         -438.2
## 5 5.094 1.938 1.497 0.7237 16.55 0.7971    -12.81         -329.7
## 6 5.405 1.981 1.826 0.5987 16.95 0.7286    -12.81         -220.3
##   log.posterior
## 1        -458.6
## 2        -458.6
## 3        -455.5
## 4        -451.0
## 5        -342.5
## 6        -233.1
```

Make sure you choose a chain with a different initial `theta` than your neighbour and **take 10 min** to analyse it and compare your posterior distributions. If you found similar posterior distributions, that means that you can join the two chains to make your posterior even more accurate. **Take 5 min** to do so. Hint: 'coda' knows how to deal with list of `mcmc` objects and most functions will work for these `mcmc.list` objects.

### Correlation

### Model selection

## To go further

- The Poisson process: see bottom of page 3 of this reference for more details.