Parameter Estimation

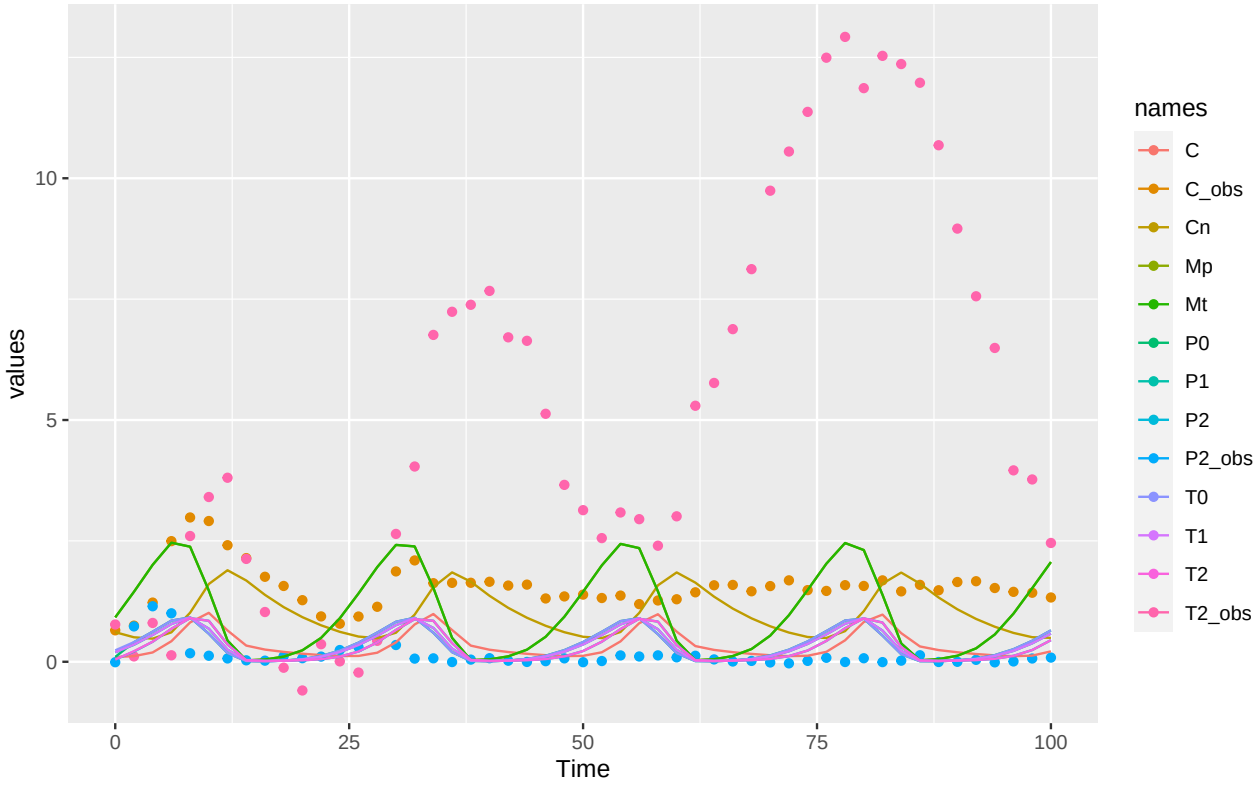
Jürgen Pahle (juergen@pahle.de)

Introduction

In this workflow we will go through a simple example of doing parameter estimation with CoRC[1].

You can see a *time course* of the species of the model below. The points are measurements of the concentrations of two species (P2, T2) in the model and the sum of two other species concentrations (C + Cn).

```
#> # A COPASI model reference:
#> Model name: "Circadian Clock"
#> Number of compartments: 1
#> Number of species: 10
#> Number of reactions: 29
```



As you can see, the model does not describe the data perfectly. We will try and get a better result by changing the parameters of our model.

[1]Förster J., Bergmann F.T., Pahle J. (2021) CoRC: the COPASI R Connector. Bioinformatics 37(17):2778-2779, https://doi.org/10.1093/bioinformatics/btab033

Parameter estimation

We want to find the parameters of our model that describe the data best. We call this *Parameter Estimation*. We have 46 parameters in our model, so finding the right combination of parameters by chance and just trying out is nearly impossible. Even only using a limited subset of these parameters (as we will do) will not work that way. We will use *algorithms* for that.

Parameter estimation is an important topic when handling a model. In **CoRC** you have to

1. Build or load a model
2. Define an experiment (with data to be fitted)
3. Define parameters that will be fitted
4. Run Parameter Estimation

We will go through these steps individually and explain what needs to be done as well as show visually how parameter estimation improves your fit.

Setup:

First, we have to load the required packages. Please make sure, you have **CoRC** as well as **ggplot2** installed before calling the library function.

```
library(CoRC)
library(ggplot2)
```

1. Load a model

As stated above, to make a parameter estimation, you have to have a model to work on. If you want to know how to build you own model, you can click here.

In this workflow, we will instead load an SBML-model. This model is the Circadian Clock model that we used previously.

```
loadSBML("CircadianClock_2.xml")
#> # A COPASI model reference:
#> Model name: "Circadian Clock"
#> Number of compartments: 1
#> Number of species: 10
#> Number of reactions: 29
```

We can inspect the species of the model like this:

```
getSpecies()
#> # A tibble: 10 x 13
#>    key      name  compartment type    unit  initial_concentration initial_number
#>    <chr>    <chr> <chr>       <chr>   <chr>                 <dbl>          <dbl>
#>  1 P0{cell} P0    cell        reacti~ nmol~                 0.100         6.02e13
#>  2 Mt{cell} Mt    cell        reacti~ nmol~                 0.919         5.53e14
#>  3 Mp{cell} Mp    cell        reacti~ nmol~                 0.919         5.53e14
#>  4 T0{cell} T0    cell        reacti~ nmol~                 0.236         1.42e14
#>  5 P1{cell} P1    cell        reacti~ nmol~                 0.196         1.18e14
#>  6 P2{cell} P2    cell        reacti~ nmol~                 0.0456        2.75e13
#>  7 T1{cell} T1    cell        reacti~ nmol~                 0.196         1.18e14
#>  8 T2{cell} T2    cell        reacti~ nmol~                 0.0456        2.75e13
#>  9 C{cell}  C     cell        reacti~ nmol~                 0.100         6.02e13
#> 10 Cn{cell} Cn    cell        reacti~ nmol~                 0.612         3.68e14
```

```
#> # i 6 more variables: concentration <dbl>, number <dbl>, rate <dbl>,
#> #   number_rate <dbl>, initial_expression <chr>, expression <chr>
```

This works in a similar way for reactions (`getReactions()`) and parameters (`getParameters()`)

If you have **COPASI** installed, you can also have a look at the model there by starting the GUI with this model loaded already:

```
openCopasi()
```

2. Define an experiment

Defining an experiment in **CoRC** means telling the program which data to fit and what the data actually describes.

So we first need data. It is always a good idea to take a look at your data before working with it. This way you can make sure nothing unexpected is happening.

```
data <- read_tsv("data2_CoRC.txt", col_types = cols())

data
#> # A tibble: 51 x 4
#>     Time C_obs   P2_obs T2_obs
#>    <dbl> <dbl>    <dbl>  <dbl>
#>  1     0 0.650 -0.00926  0.771
#>  2     2 0.747  0.731    0.111
#>  3     4 1.22   1.15     0.803
#>  4     6 2.50   1.00     0.134
#>  5     8 2.98   0.176    2.60
#>  6    10 2.91   0.125    3.41
#>  7    12 2.41   0.0688   3.81
#>  8    14 2.14   0.0298   2.13
#>  9    16 1.76   0.0228   1.03
#> 10    18 1.57   0.117   -0.123
#> # i 41 more rows
```

Then you have to define the experiment for COPASI. You need the data, as well as the type and mappings for the species. You can choose a weight method for your data (which prevents parameters getting fitted more closely just because they have higher values).

Your data columns in your data file can be of type “time”, “dependent” and “independent”, and if you want to exclude a column you can choose “ignore”.

The mapping argument in the function maps the data columns with the species in your model. In our case, the provided data is time course data, and our values are “transient concentrations”. They are denoted like this: {[Species]}. You can find these notation by using the function `getSpeciesReferences()`.

Time needs to be mapped with `NA`.

Allowed weight methods are `"mean"`, `"mean_square"`, `"sd"`, and `"value_scaling"`.

```
fit_experiments <- defineExperiments(
  data = data,
  type = c("time", "ignore", "dependent", "dependent"),
  mapping = c(NA, "{Values[C_obs]}", "{[P2]}", "{[T2]}"),
  weight_method = "mean_square"
)
```

3. Define parameters

We now have to define parameters that will be fitted.

First, let us take a look at all the parameters in the model:

```
getParameters()
#> # A tibble: 46 x 5
#>    key                   name  reaction         value mapping
#>    <chr>                 <chr> <chr>            <dbl> <chr>
#>  1 (sP).k                k     sP                 0.9 <NA>
#>  2 (Mp transcription).V  V     Mp transcription   1   <NA>
#>  3 (Mp transcription).K  K     Mp transcription   0.9 <NA>
#>  4 (Mp transcription).n  n     Mp transcription   4   <NA>
#>  5 (Mp degradation).Km   Km    Mp degradation     0.2 <NA>
#>  6 (Mp degradation).V    V     Mp degradation     0.7 <NA>
#>  7 (Mt transcription).V  V     Mt transcription   1   <NA>
#>  8 (Mt transcription).K  K     Mt transcription   0.9 <NA>
#>  9 (Mt transcription).n  n     Mt transcription   4   <NA>
#> 10 (Mt degradation).Km   Km    Mt degradation     0.2 <NA>
#> # i 36 more rows
```

Now, we only want to fit the reaction rates (parameters with V). To make our fit parameters we need to make a list of lists with the attributes of the different parameter-estimation settings. To *define* a parameter for parameter estimation, we use the `defineParameterEstimationParameter()` function.

```
fit_parameters <- list(
  defineParameterEstimationParameter(
    ref = "{(Mp degradation).V}",
    start_value = getParameters("(Mp degradation).V")$value,
    lower_bound = 0.1,
    upper_bound = 2
  ),
  defineParameterEstimationParameter(
    ref = "{(Mp transcription).V}",
    start_value = getParameters("(Mp degradation).V")$value,
    lower_bound = 0.1,
    upper_bound = 2
  ),
  defineParameterEstimationParameter(
    ref = "{(P2 degradation).V}",
    start_value = getParameters("(P2 degradation).V")$value,
    lower_bound = 0.1,
    upper_bound = 3
  ),
  defineParameterEstimationParameter(
    ref = "{(T2 degradation).V}",
    start_value = getParameters("(T2 degradation).V")$value,
    lower_bound = 0.1,
    upper_bound = 3
  ),
  defineParameterEstimationParameter(
    ref = "{(Mt degradation).V}",
    start_value = getParameters("(Mt degradation).V")$value,
    lower_bound = 0.1,
    upper_bound = 2
```

```
  ),
  defineParameterEstimationParameter(
    ref = "{(Mt transcription).V}",
    start_value = getParameters("(Mt transcription).V")$value,
    lower_bound = 0.1,
    upper_bound = 2
  ),
  defineParameterEstimationParameter(
    ref = "{(C transport).k1}",
    start_value = getParameters("(C transport).k1")$value,
    lower_bound = 0.1,
    upper_bound = 10
  ),
  defineParameterEstimationParameter(
    ref = "{(C transport).k2}",
    start_value = getParameters("(C transport).k2")$value,
    lower_bound = 0.1,
    upper_bound = 10
  )
)
```

4. Run parameter estimation

To show how well our parameter estimation works, we want to print the model before and after parameter estimation. To do this, we have to *run* two time course evaluations, one with the parameters now, and one with the parameters after the parameter estimation.

```
before <- runTimeCourse(duration = 100, dt = 2)$result
```

After doing this, we will now actually run the parameter estimation. We need the parameters, experiments with our data that will be fitted, and specify the method. We want to use the Levenberg Marquardt method but other methods are available as well. You can find them using the function `getValidReactionFunctions()` with your function as an argument.

Also, we specify that we want to update our model. This means, that all estimated parameters will be updated with the parameters of the best estimation. To compare the fit to the previous parameters we need to make sure we keep the previous fit. We already did that with our time course in the last chunk.

```
result <-
  runParameterEstimation(
    parameters  = fit_parameters,
    experiments = fit_experiments,
    method = list(
      method = "LevenbergMarquardt",
      log_verbosity = 2
    ),
    update_model = TRUE
  )
```

You can have a nicely readable version of the result by using the function `str(result)`. For space-reasons we will only take a look at the fitted parameters, but feel free to take a look at anything you find interesting.

```
result$parameters
#> # A tibble: 8 x 8
#>   parameter            lower_bound start_value value upper_bound std_deviation
#>   <chr>                      <dbl>       <dbl> <dbl>       <dbl>         <dbl>
```

```
#> 1 (Mp degradation).V          0.1        0.7 0.804          2       0.0491
#> 2 (Mp transcription).V        0.1        0.7 0.904          2       0.0481
#> 3 (P2 degradation).V          0.1        2   0.573          3       0.0181
#> 4 (T2 degradation).V          0.1        2   1.90           3       0.0630
#> 5 (Mt degradation).V          0.1        0.7 0.315          2       0.00886
#> 6 (Mt transcription).V        0.1        1   0.938          2       0.0263
#> 7 (C transport).k1            0.1        0.6 0.659         10       0.0505
#> 8 (C transport).k2            0.1        0.2 0.234         10       0.0119
#> # i 2 more variables: coeff_of_variation <dbl>, gradient <dbl>
```

5. Visualize results

Now we have estimated and updated the parameters of our current model. To compare our old model parameters to our new, we run another time course.

```
after <- runTimeCourse(duration = 100, dt = 2)$result
```

We will use ggplot for visualizing our results. If you have never worked with ggplot, this way to define a plot will look unusual to you.

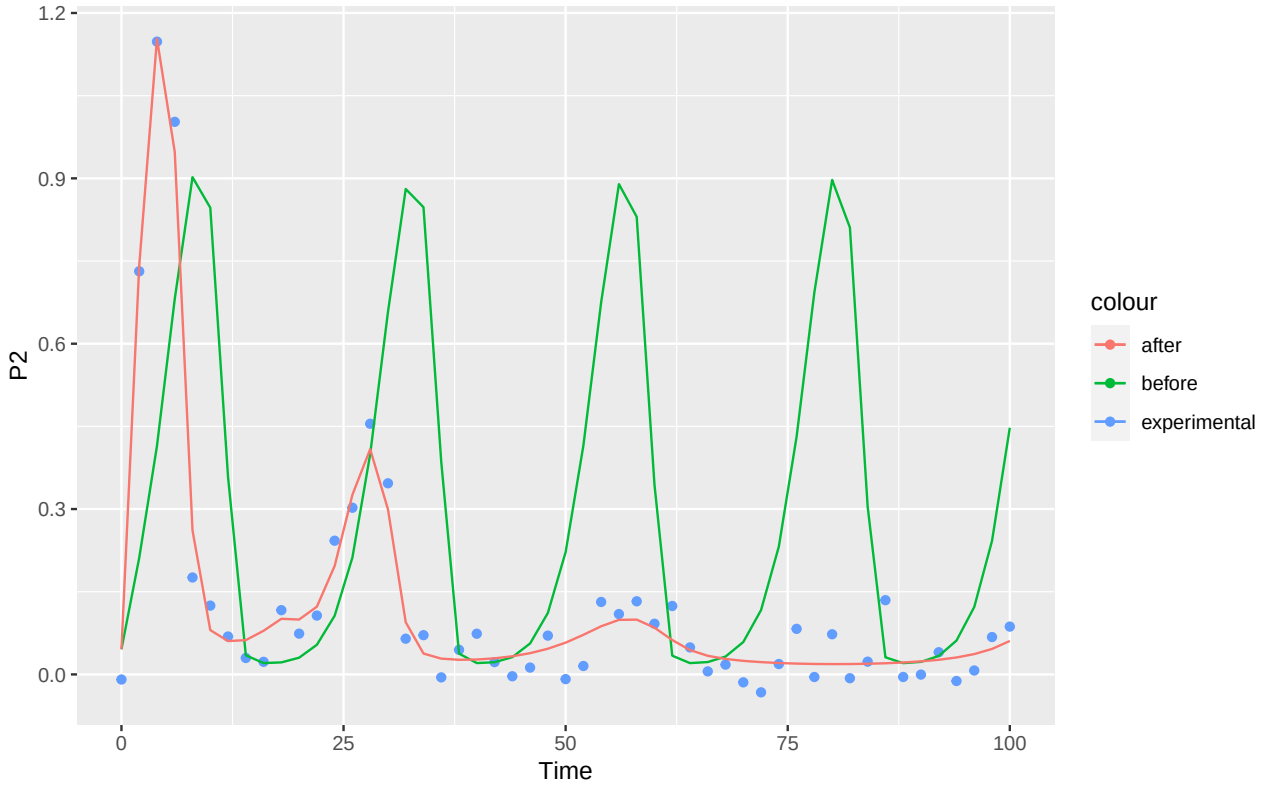
We first want to plot our experimental data, as well as two time courses (before and after) for *P2* and *T2*.

```
P2 <-
  ggplot(mapping = aes(x = Time, y = `P2`)) +
  geom_point(data = data, aes(color = "experimental", y = `P2_obs`)) +
  geom_line (data = before, aes(color = "before")) +
  geom_line (data = after, aes(color = "after"))

T2 <-
  ggplot(mapping = aes(x = Time, y = `T2`)) +
  geom_point(data = data, aes(color = "experimental", y = `T2_obs`)) +
  geom_line (data = before, aes(color = "before")) +
  geom_line (data = after, aes(color = "after"))
```

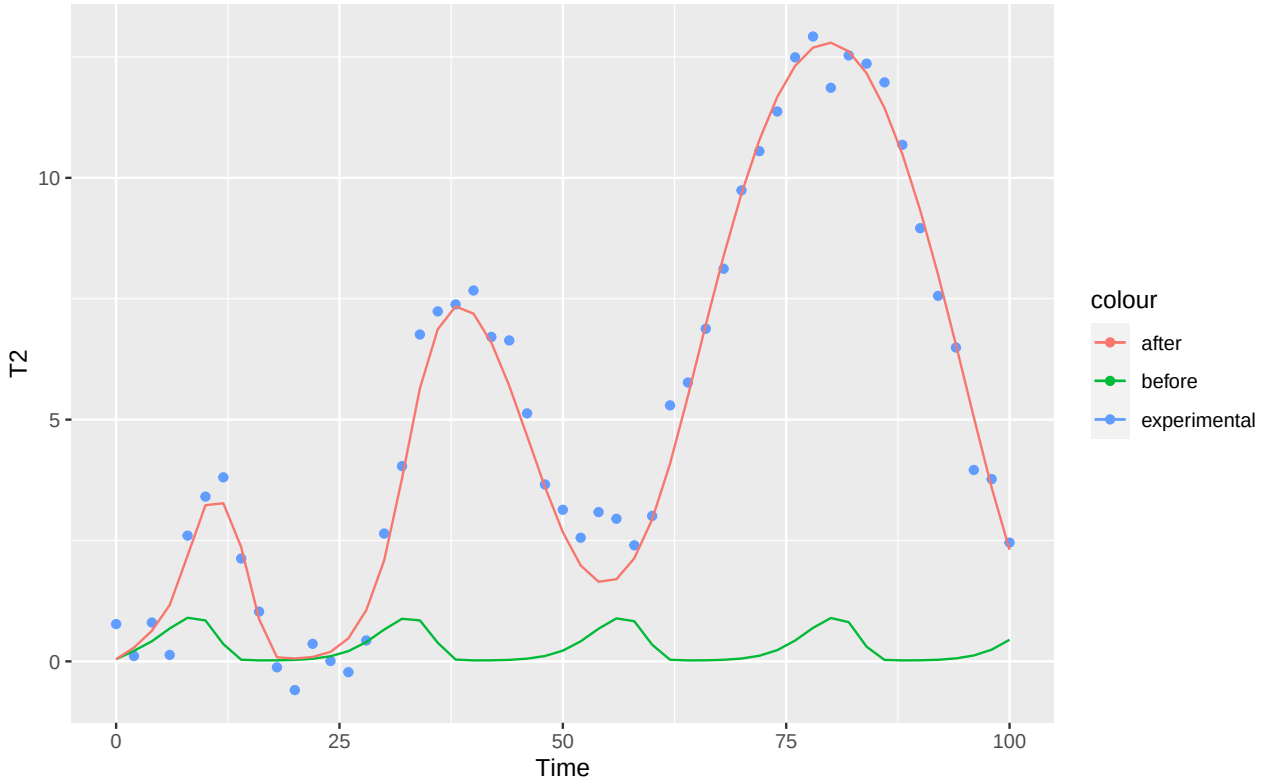
At the end, our result for P2

```
P2
```



as well as our result for T2

```
T2
```



show, how well the parameter estimation was able to fit our model to the data.

At the end, we can *unload* our model, to free some memory.

```
unloadModel()
```

References