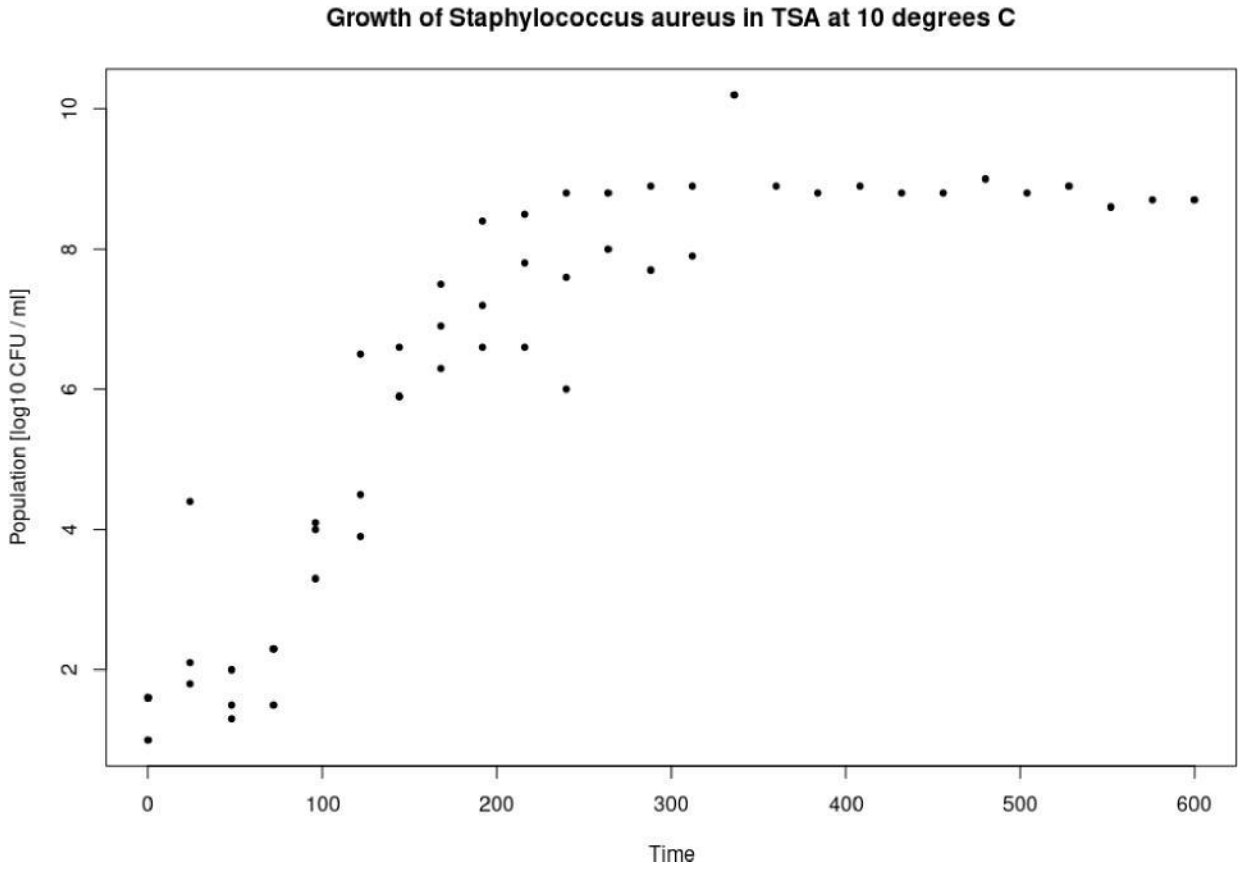
Dr Tomasz Kurowski
t.j.kurowski@cranfield.ac.uk
`

Database interface practical - continued

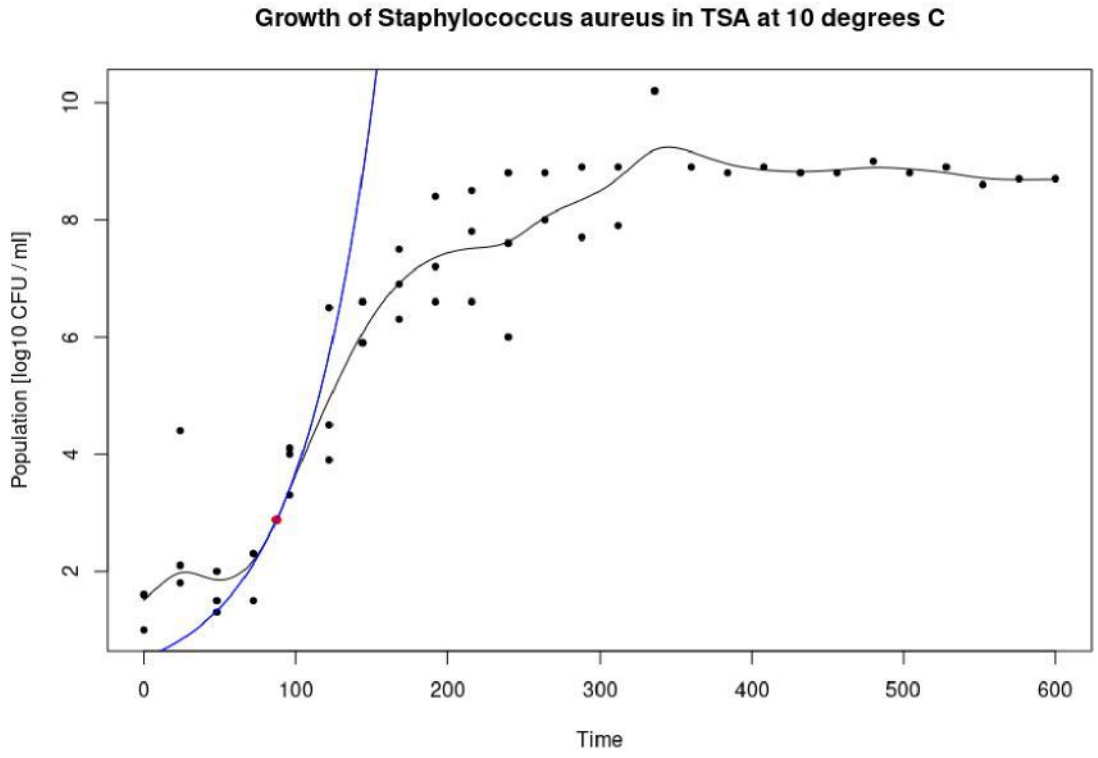
Today's practical is a direct continuation of the one from Wednesday, so feel free to continue your previous work if you have not finished yet before proceeding to the tasks below. At the very least your REST API should have an endpoint which returns data points for a given experiment.



As you have likely noticed by viewing scatter plots (one is shown above – note that it has more data points than the data that you are using) of our experimental data sets in R, the population (as represented by the concentration of bacteria expressed in $\log_{10}$ of CFU / ml) in most bacterial growth scenarios follows a sigmoid (S-shaped) curve, starting with a **lag phase** where the population is stable, followed by an **exponential growth phase** (the CFU measurements are logarithmic, so exponential growth appears linear in our

plots), then finally reaching a **stationary phase**, where growth plateaus and stops due to a limiting factor (e.g. running out of nutrients). This is generally followed by **decline** or **death phase**, though that is not present in our data. This growth can be modelled using various parametric models (e.g. the Gompertz or Baranyi curves), but we will use a non-parametric function to approximate the curve and estimate the maximum growth rate (usually designated as μ_{max}). Install the R package `growthrates`, and use the `fit_spline` function on data points retrieved from your API as shown below. You can also try `fit_easylinear`, which will attempt to fit a linear model to the exponential phase, and `fit_growthmodel`, which will fit a parametric model of your choice (but will require estimates for initial parameters).

```
fit <- fit_spline(datapoints$time, datapoints$cfu)
plot(fit)
coef(fit) # Report on the growth coefficient
```



Calling R from Node

In the previous steps we have been accessing our API from R, but the reverse is also possible. We will use a small npm package called `r-script`. Install it and import it in your router:

```
const rscript = require('r-script');
```

Internally, the `r-script` package uses the `child process` module, which can launch other command-line applications from Node and retrieve their results. In this case, it will be launching the Rscript interpreter (e.g. Rscript.exe on Windows) to run a script written in R, but you could also use `child_process` to run scripts written in any language, like Python scripts or Bash scripts. Have a look at https://nodejs.org/api/child_process.html if you want to try doing that later, but for now we will use the simpler `r-script` package instead of using `child_process` directly.

The Rscript binary has to be available in the environment that you launch Node from. You can check if that is the case is by typing "Rscript" into your command line. If the command is unavailable, you will need to add the path to the Rscript binary to the PATH environment variable. You can do this in your **server.js** file like so:

On Windows:

```
const RSCRIPT_ PATH = "C:\\path\\to\\R\\bin";
process.env.PATH += ";" + RSCRIPT_PATH;
```

On macOS / Linux:

```
const RSCRIPT_ PATH = "/path/to/R";
process.env.PATH += ":" + RSCRIPT_PATH;
```

If you use RStudio, the location of your R installation will be listed under Tools > Global Options... > General > R version. You can use it in `RSCRIPT_PATH` in the above examples.

`

Now, let's prepare an R script which will estimate and report the growth rate based on times and CFU measurements. Create a file called fit_model.R, with the following code:

```
# "needs" instead of "library" or "require" for r-script
needs(growthrates)
attach(input[[1]])

fit <- fit_spline(times, cfus)
coef(fit)['mumax']
```

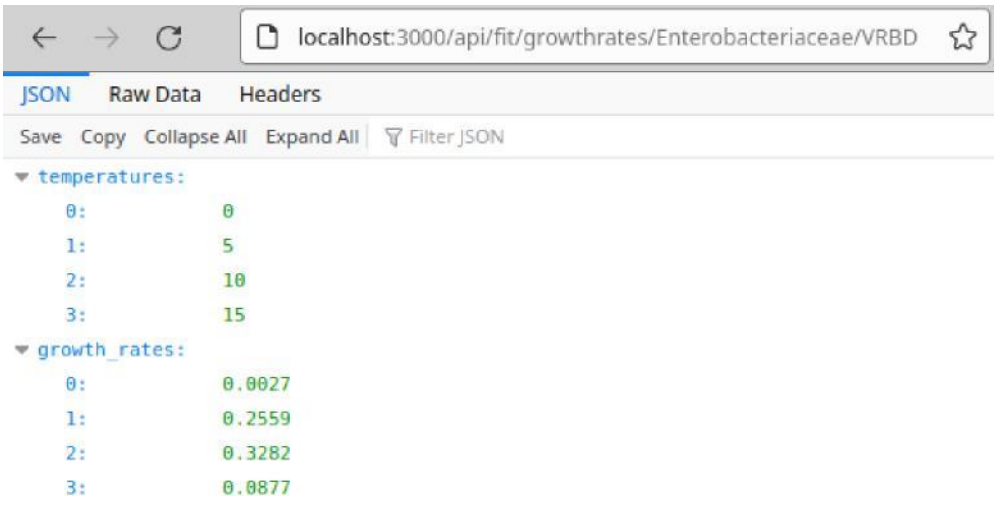
It expects to attach a vector called "times" and another vector called "cfus" from the standard input, then it tries to fit a spline to the time series and returns the maximum growth rate (mumax) on the standard output. You can input data by using the data method of the rscript object, and call an R script by using the call method. Implement the following endpoint in your router, which will report the growth rate for a specific experiment:

```
microbe_router.get('/fit/growth/:experiment_id',function(req, res)
{
    const query = 'SELECT time, cfu FROM datapoints ' +
                  'WHERE experiment_id = ? ORDER BY time;';
    const parameters = [
        req.params.experiment_id
    ];

    db.all(query, parameters, function (err, rows) {
        if (err) {
            throw err;
        }

        const r_input = {
            times: [],
            cfus: []
        };

        for (let i = 0; i < rows.length; i++) {
            r_input.times.push(rows[i].time);
            r_input.cfus.push(rows[i].cfu);
        }
```

`

```
        rscript('fit model.R').data(r input)
                                .call(function (err, growth_coef) {
            if (err) {
                throw err;
                // Encoding and displaying R error message
                console.log(String.fromCharCode.apply(null, err));

            }
            const output = {
                experiment_id: req.params.experiment_id,
                growth_coefficient: growth_coef
            };
            res.json(output);
        });
    });
});
```

Maximum growth rate is a function of temperature. Implement an API route which reports all the temperatures and their corresponding maximum growth rates for a given organism and medium. The result could look like this:



Notice that the maximum growth rate increases with temperature up to a point, before falling again, hinting at ideal conditions for the bacteria.

Generating an R REST API with plumber

Calling R from Node is useful and represents a very generic technique. You could actually use your Node back-end to launch any other tool or scripts and make their results available through your REST API. If you look at the code of `r-script` package on GitHib (https://github.com/joshkatz/r-script), you can see that this is all it does, using a function called `child_process.spawn`.

However, this approach is also somewhat restrictive, as you are forced to use have R set up on your server and it has to be very tightly coupled with your JavaScript code. A more flexible solution might be to deploy a second REST API, which will serve R-specific requests. We will use an R package called **plumber** to do just that. The package allows you to transform ordinary R functions into REST API endpoints by adding specific annotations (comments beginning with `#*`) above each function.

Open RStudio, install the `plumber` package, and create a script called `hello_plumber.R`. Use the code below to create a simple REST endpoint for a GET request:

```
library(plumber)
#* @apiTitle Greeting API in Plumber

#* @get /greeting
function() {
    return("Hello world!")
}
```

You can deploy the REST API on port 3001 by typing the following command into an R console:

```
plumber::plumb("hello plumber.R")$run(port = 3001)
```

Test the API with curl or a browser (use http://localhost:3001/greeting).

You can also use parameters, which can have a default value (NULL in the example below) set in the function declaration:

```
#* @param name
#* @get /greeting
function(name = NULL) {
    if (is.null(name)) {
        return("Hello world!")
    } else {
        return(paste("Hello", name, "!"))
    }
}
```

In the REST API, these are used as keyword parameters, rather than the positional parameters that we were using in our Express API. Keyword parameters follow a question mark (`?`) after the URL, are provided as key-value pairs and can be specified in any order (separated by `&` symbols). Test the endpoint you implemented above at http://localhost:3001/greeting?name=Tom.

You can very easily return images by specifying a serializer. We will generate a simple histogram of random values and return a PNG image:

```
#* @get /histogram
#* @serializer png
function(sample_size=1000, breaks=100) {
    samples <- rnorm(sample_size)
    hist(samples, breaks=as.numeric(breaks))
}
```

Note that inputs will be treated as strings by default, so we may need to convert them to numbers. Test the route at:

http://localhost:3001/histogram?sample_size=500&breaks=50

As you may remember from yesterday (the final "Basic visualisation in R" part of the practical), it very easy to make REST API requests in R. You can adapt the code from yesterday, and make your plumber API (running on port 3001) retrieve data from your Express API (running on port 3000), then generate a bacterial growth curve plot for a particular experiment:

```
#* @get /growth_curve
#* @serializer png
function(experiment_id) {
    request <- GET(paste("http://localhost:3000/api/datapoints/",
                         experiment_id, sep=''))
    response <- content(request, as = "text", encoding = "UTF-8")
    df <- fromJSON(response)
    plot(df$time, df$cfu)
}
```

Test the above endpoint, then try to implement a new endpoint, which should return plots like the one on page two of this practical – with a growth model curve fitted to the data points. This is your final (non-extra) task!

Extra task

Implement a REST endpoint in your Express API which will make a request for a growth curve plot to your plumber API, and return the plot to the user. You could use any of a number packages to send the request (from Node to plumber). I would recommend **needle** (https://www.npmjs.com/package/needle), but the **request**, **node-fetch**, **axios**, **bent**, and **got** packages are other popular options. The `XMLHttpRequest` class which you have used on Monday to execute HTTP queries from JavaScript is not appropriate here, since it is specific to the browser and cannot normally be used in Node.