Creating Dynamic Documents with R and Python as a Computational and Visualization Tool for Teaching Differential Equations

Supplemental Material

B. Kostadinov[*]*, J. Thiel, and S. Singh, New York City College of Technology, CUNY*

Abstract

This is a supplemental material for the PRIMUS article with the same title, published in the Special Issue on Modeling in Differential Equations Courses. This material is posted on the PRIMUS website.

Contents

Introduction

This is a supplemental material for the PRIMUS article with the same title, published in the Special Issue on Modeling in Differential Equations Courses. This material is available for download from the PRIMUS website. The motivation for the main PRIMUS article was two-fold:

[*]Corresponding Author. **Contact** Boyan Kostadinov at bkostadinov@citytech.cuny.edu

1. We wanted to discuss and illustrate various free computational resources that can be used for numerically solving systems of linear or nonlinear differential equations that make investigation of realistic modeling scenarios possible, along with creating reproducible dynamic documents that combine plain text, TEX expressions and R or Python code with the purpose of creating publication quality project modules and reports, lecture notes, exams, tutorials, papers etc.
2. We wanted to promote *SIMIODE*, an online community for faculty and students that provides rich repositories with interesting applications, student projects and modeling scenarios, lecture notes and tutorials, video, data, instructional technologies, discussion areas etc.

SIMIODE provides a comprehensive community approach to motivating the study of differential equations from a modeling perspective. See [1] for more details. Another online community dedicated to providing resources that support the teaching and learning of ordinary differential equations is *CODEE: Community of Ordinary Differential Equations Educators*, [2].

Installing TEX R, Python and RStudio

In order to be able to create dynamic documents with R and Python, using R Markdown with the `Knitr` R package, we need to install TEX, R, Python, and RStudio. The following software installers are available for Windows, Mac, and Linux.

- MiKTeX installer for Windows: http://miktex.org/download
- MacTeX installer for Mac: http://tug.org/mactex/
- R installers: http://cran.stat.ucla.edu/
- RStudio installers: https://www.rstudio.com/
- Anaconda distribution: https://www.anaconda.com/download/

The Anaconda distribution is an open source, easy-to-install Python and R distribution. It includes the most popular scientific packages for Python and more than 80 of the most used R packages for data science. The Anaconda Navigator, which is installed automatically with the Anaconda distribution, is a graphical interface that offers a one stop solution to scientific computing with Python and R. It contains the new Jupyter Lab, Jupyter Notebook, RStudio, and Spyder IDE.

Computing with R and Python on the Cloud

We list some free resources for cloud-based computing with R and Python:

- CoCalc: https://cocalc.com/
- SageMath: http://www.sagemath.org/
- MS Azure Notebooks: https://notebooks.azure.com/

In particular, *CoCalc*: Collaborative Calculation in the Cloud, offers free and paid accounts for online computing using SageMath, R, Python, Julia, Octave, etc., but it also offers document authoring capabilities using Jupyter Notebooks based on Markdown and LATEX . Additionally, *CoCalc* allows for the creation of TEX files with embedded SageMath code. This feature can be used to create randomized exams and homework assignments, similar to what can be done in RStudio using R Markdown. The Microsoft Azure Notebooks provide free cloud-based computing environments for R and Python. One can also set up a server and run the free RStudio server in the Cloud to have a web-based computing environment for R and Python.

It is worth mentioning that for expensive commercial software, including Matlab, Mathematica, Maple, SAS and SPSS, the CUNY Virtual Desktop provides free remote access anytime, anywhere to computing resources for all CUNY students and faculty.

Creating Dynamic Documents in RStudio

We can create dynamic documents in RStudio, [3], using the R Markdown format, [4], which allows us to write lecture notes, randomizable exams, project reports, presentations and even publication quality papers. The default computational engine is the R programming language [5], but it can be changed to Python (plus a few other options), and the same R Markdown document can contain both R and Python code chunks. However, running Python code inside R Markdown requires the R package `reticulate`, which provides R interface to Python.

R Markdown is a custom version of Markdown, which is a simple format that combines plain text, LaTeX expressions (without the heavy mark-up structure of a typical LaTeX document), and R or Python code chunks. The R Markdown document is rendered into the final publication format after clicking the `Knit` button in RStudio, and the same source file can produce different output formats. The LaTeX expressions, along with all numerical and graphical output, obtained from the code chunks during the knitting process, are automatically included into the output document with publication quality formatting and color-coding of the code chunks. In our experience, we find the R Markdown documents extremely useful for developing educational materials, as well as organizing, implementing, and maintaining research projects. R Markdown and the `Knitr` package offer the following features and more:

- Many output formats, including HTML, PDF, and MS Word.
- Interactive R Notebooks using the R Markdown format.
- Presentations with Beamer, ioslides, reveal.js and Slidy.
- Including raw LaTeX expressions within Markdown.
- Including chunks of R, Python, SQL, Bash and other code.
- Expanded support for tables and bibliographies.
- Compiling HTML, PDF or MS Word notebooks from R scripts.
- Creating interactive R Markdown documents with Shiny.

Note that PDF output (including Beamer slides) requires TeX. In fact, this supplement was created from a simply formatted R Markdown document using the authoring capabilities in the `Knitr` R package in RStudio. The most interactive formats are based on the R Notebooks and the more advanced Shiny documents that allow for developing interactive web-based applications. The same R Markdown format is used to create the interactive R Notebooks since the code chunks can be executed individually and the output they produce is immediately rendered below the code chunk. Thus, the R Notebooks allow for interactive step-by-step development inside RStudio, or a coding HTML or PDF presentation outside of RStudio.

For a short introduction to R, see [6]. The papers [7, 8] give a taste of the computational power of R in the context of implementing probabilistic simulations using R. The R programming language is still primarily used for any kind of statistical computing and stochastic simulations, and it is one of the most popular programming environments for doing modern data science, along with Python. A more recent paper [9], by one of the authors, implements in R a stochastic model for simulating the Electoral College distribution and predicting the outcome of the US Presidential Election. However, R is suitable for any kind of numerical computing and sophisticated visualizations thanks to the thousands of core and contributed packages.

In this paper, we focus our attention mostly on how R can be used to numerically solve systems of ordinary differential equations, using wrappers, implemented in the `deSolve` package. We also show how Python can be used for numerically solving systems of ODEs in the context of a basic SIR model.

R Markdown Basics

Once, TEX, R, Python and RStudio have been installed, we can make the best of the RStudio IDE (Integrated Development Environment). A new R Markdown document can be created inside RStudio by clicking the green ⊕ button in the upper-left corner of the RStudio interface. In the drop-down menu, the third option is to create an R Markdown document. Choosing the R Markdown option opens a window asking the user to specify a Title, an Author and a default output format (HTML, PDF, Word), which can later be changed in the header at the top of the R Markdown file (Rmd). At this point, one can also choose to create a presentation or a Shiny document, all based on R Markdown. Clicking the `OK` button generates an R Markdown template populated with some sample text and code to get you started. The final output format is rendered by clicking the `Knit` button in the upper-left corner. Compiling the output document with the `Knit` button also creates the TEX source file for the final document (but this output option must be checked first). Chunks of R or Python code can be inserted into the source Rmd document by clicking the `Insert` button in the upper-right corner, and plain text and TEX expressions can freely be added. The compiled final output includes text, code (optional), rendered TEX expressions and the numerical and graphical output from any code chunks embedded within the source Rmd document, all formatted to publication quality. Any code chunk can be independently evaluated inside the Rmd file, and the result immediately displayed beneath the code. In addition to the code chunks, one can also evaluate inline R expressions by enclosing them with a single back-tick qualified with `r` on the left side. Specifically, the inline R expression:

```
`r exp(sqrt(2))`
```

computes $e^{\sqrt{2}}$ and returns the number 4.1133 (rounded to 4 digits).

We can use pretty much any TEX expression inside the R Markdown document, either as an inline or display style expressions. In general, an inline TEX expression can be included using the following syntax: `$inlineExpr$` (no white space is allowed adjacent to the $ delimiters). For display equations, we write `$$displayEqn$$`, or use `\begin{equation} ... \end{equation}` format for numbered equations.

The following line with plain text and inline TEX and R expressions:

```
The expression $\frac{12-2^3}{-2}-3\times (2-5)^3$ gives `r (12-2^3)/(-2)-3*(2-5)^3`.
```

results in: The expression $\frac{12-2^3}{-2} - 3 \times (2-5)^3$ gives 79.

Note that the result is not hard-coded, and if we change the R expression in the Rmd file, the answer in the output document will also change after we knit it again. For more details on R Markdown, see [4].

Running Python in R Markdown

The `reticulate` R package provides R interface to Python. The package offers tools for easy interoperability between R and Python. If your RStudio is using `knitr` version 1.18 or higher, then the `reticulate` Python engine is enabled by default, as long as `reticulate` is installed, and no further setup is required.

The `reticulate` package can be installed from CRAN as follows:

```
install.packages("reticulate")
```

The call `Sys.which("python")` shows the Python version found on your PATH. By default, `reticulate` uses this version of Python, but the `reticulate` function `use_python()` allows us to specify an alternative version of Python. For example, we can specify Anaconda Python 3.6 as follows:

```
library(reticulate)
use_python("~/anaconda3/bin/python3.6")
```

Some of the main features of the `reticulate` package are the following:

- Binding to different versions of Python.

- Providing a Python engine for R Markdown.
- Translating between R and Python objects.

The Python engine for R Markdown, provided by the `reticulate` package, offers some great features:

- Running Python code chunks with shared variables, exactly like R code chunks.
- Printing numerical and graphical (`matplotlib`) output from Python code chunks.
- Having access in R chunks to objects created inside Python chunks, using the `py` object. For example, `py$x` in an R chunk would access the `x` variable created in a Python chunk.
- Having access in Python chunks to objects created inside R chunks using the `r` object. For example, `r.x` in a Python chunk would access the `x` variable created in an R chunk.

For more details, see the `reticulate` package documentation.

Problem Randomization for Quizzes and Exams

Using a Markdown format, problems can be randomized by combining LaTeX with R or Python expressions that would allow various problem parameters to be randomized, and more generally, the problems to be algorithmically structured. This can be done either in RStudio, using R Markdown (Rmd), or in CoCalc, using Jupyter notebooks. We show an example written in an Rmd file in RStudio. The idea is to use one or more R variables in order to randomize various parameter inside a LaTeX math expression, using inline code chunks. More specifically, the R code chunk below creates a variable `a`, which contains a single number, sampled uniformly at random from the sequence $-7, -6, \ldots, -1$.

```
a<-sample(-7:(-1),1)
```

We then use the variable `a` to randomize a parameter inside a LaTeX math expression, using inline code chunks such as `` `r a^2` ``, and `` `r abs(a)` ``. More specifically, we can randomize the coefficients in the limit below with the following mixture of LaTeX and inline R code:

```
$\lim_{x\to `r a`} \frac{x^2 - `r a^2`}{x + `r abs(a)`}$
```

which produces a randomized version of the limit: $\lim_{x\to -2} \dfrac{x^2-4}{x+2}$.

Basic Function Plotting in R

We can plot the graph of the function $f(x) = 2\sin(6\pi x) + 4\cos(8\pi x)$ with the following R code, inserted in a proper code chunk, based on using the plotting function `curve()`:

```
f<-function(x) 2*sin(6*pi*x) + 4*cos(8*pi*x) # f(x) as an R function
curve(f,from=-2,to=2,col="blue",n=750,type="p",pch=20,cex=0.3)
abline(h=0,v=0) # add x and y axis
```

The given R code produces the plot in Figure 1. In the Rmd source file, we used some other optional graphical parameters to fine-tune the graphical output.

Kepler's Third Law From Least Squares

In this section, we illustrate how Kepler's Third Law can be discovered from fitting a power model to planetary data. With this example we want to illustrate the general approach to fitting a model to data using linear and nonlinear least squares in R. For more details, see the section of the same title in the main PRIMUS article.

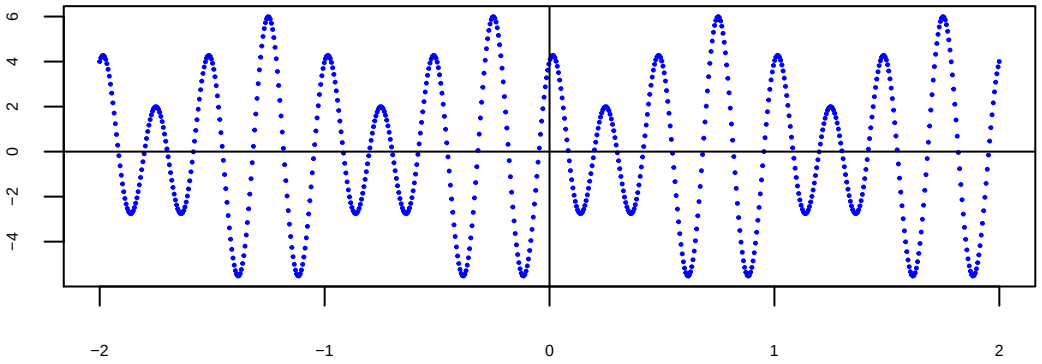

Figure 1: A scatterplot of $f(x)$.

We use planetary data collected from NASA's Lunar and Planetary Science Division, [10]. The planetary data in Table 1 were derived from the original NASA data by normalizing the distances and the orbital periods in Earth's units. The goal is to use the normalized planetary data to find a relationship between the average distance from the Sun, given in AU, and the orbital period, given in Earth's years. For this purpose, we use a power model with two parameters α and β, of the form:

$$T = \alpha D^{\beta}, \tag{1}$$

where T is the orbital period in Earth's years, and D is the average distance from the Sun in AU. This particular model can be linearized, so we can fit the model in (1) using either linear or nonlinear least squares. In the next two subsections, we illustrate how to do this in R.

Complete R Code for Fitting the Model with Linear Least Squares

We can linearize the model in (1), and then apply linear least squares to fit the model parameters. The linearization of (1) consists of taking the logarithm on both sides (all values are positive):

$$\log(T) = \log(\alpha) + \beta \log(D). \tag{2}$$

Now, we have a linear model with respect to the unknown parameters $\gamma = \log(\alpha)$ and β. However, the response variable is now $y = \log(T)$, and the explanatory (predictor) variable is $x = \log(D)$. Thus, the linearized model can be written in the form:

$$y = \gamma + \beta x. \tag{3}$$

The main R function used to carry out linear regression (least squares), and fit linear models to data, is `lm()`. Linear models for `lm()` are specified symbolically. A typical model has the form `response~predictor`, where `response` is the numeric response vector, and `predictor` is a numeric vector that specifies a linear predictor. In our case, we can use the formula `y~x` in order to fit the model parameters γ and β in our linear model (3) to the given data.

In the R code chunks listed below, we provide the complete R code that prints Table 1 of the normalized planetary data, implements the linear least squares, prints Table 2 with the fitted model parameters, re-fits the linear model after removing the intercept, which turns out to be statistically insignificant, prints Table 3 with the re-fitted model parameters, and finally plots in Figure 2 the data (as red dots), and the fitted model (as a blue curve), on the same plot.

```
library(xtable) # must install xtable first
options(xtable.comment = FALSE) # suppress comments
# for xtable, one must set results='asis' in the chunk header
planet<-c("Mercury","Venus","Earth","Mars","Jupiter","Saturn","Uranus",
          "Neptune","Pluto")
# Planetary data: distance is given in units of 10^6 km, and period in days
dist<-c(57.9,108.2,149.6,227.9,778.6,1433.5,2872.5,4495.1,5906.4)
period<-c(88,224.7,365.2,687,4331,10747,30589,59800,90560)
# Normalized planetary data
D<-dist/dist[3] # normalize distance relative to Earth's
T<-period/period[3] # normalize period relative to Earth's Year
data<-cbind(T,D) # column-bind the data vectors into a matrix
row.names(data)<-planet # use planet names as the row names
colnames(data)<-c("Period T [Earth's Years]","Distance D [AU]")
# print a table of normalized planetary data
xtable(data, digits = 6, caption = "Normalized planetary data.",label = 'table1')
```

	Period T [Earth's Years]	Distance D [AU]
Mercury	0.240964	0.387032
Venus	0.615279	0.723262
Earth	1.000000	1.000000
Mars	1.881161	1.523396
Jupiter	11.859255	5.204545
Saturn	29.427711	9.582219
Uranus	83.759584	19.201203
Neptune	163.745893	30.047460
Pluto	247.973713	39.481283

Table 1: Normalized planetary data.

```
# Linear least squares to fit a linear model: y = a + bx
y<-log(T) # response data vector
x<-log(D) # predictor data vector
model<-lm(y~x) # linear regression that fits a and b
# print a table of fitted model parameters
xtable(summary(model),caption = "Linear least squares fit for the model $y=a+bx$.",
       label = 'table2')
```

	Estimate	Std. Error	t value	Pr(>|t|)
(Intercept)	-0.0003	0.0012	-0.25	0.8095
x	1.4988	0.0005	2787.34	0.0000

Table 2: Linear least squares fit for the model $y = a + bx$.

```
# Re-fit the linear model without intercept: y = bx
new.model<-lm(y~0+x)
# print a table of re-fitted model parameters
xtable(summary(new.model),caption = "Linear least squares fit for the model $y=bx$.",
       label = 'table3')
```

	Estimate	Std. Error	t value	Pr(>|t|)
x	1.4987	0.0004	3980.08	0.0000

Table 3: Linear least squares fit for the model $y = bx$.

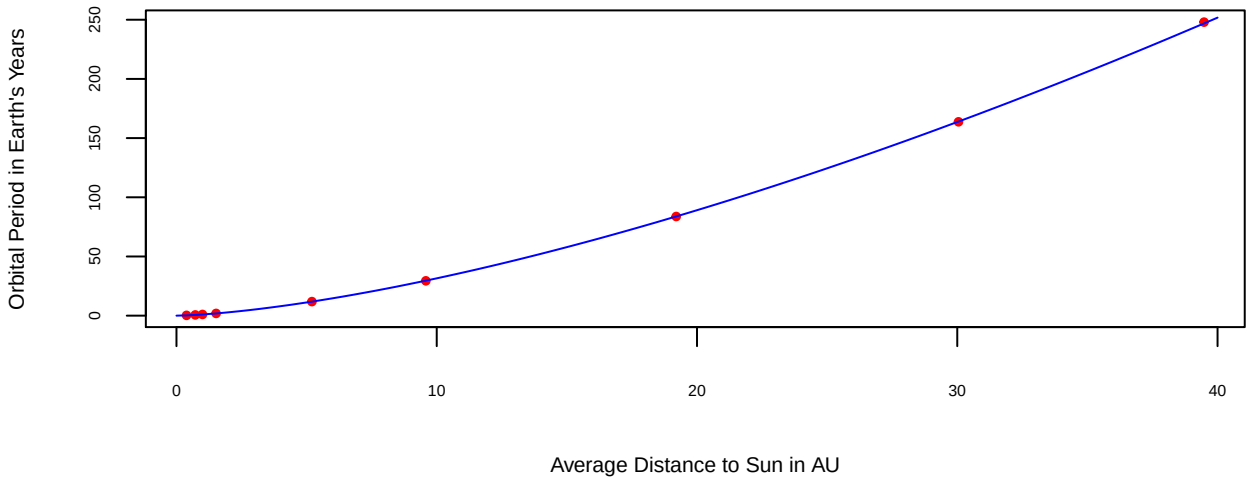

Figure 2: The planetary data and the model fitted with linear least squares.

```
fitT<-function(D) D^{new.model$coef[1]} # fitted model as an R function
# Plot data points and fitted model on the same plot
plot(D,T,pch=20,cex=0.8,col="red",xlab="Average Distance to Sun in AU",
     ylab="Orbital Period in Earth's Years",cex.lab=0.7,cex.axis=0.5)
curve(fitT,from=0,to=40,add=TRUE,col="blue",lwd=1) # add graph of fitT
```

Complete R Code for Fitting the Model with Nonlinear Least Squares

The ability to implement nonlinear least squares is often required for real-world applications, given that models are often complex and nonlinear, and they cannot be usually linearized.

We can use the nonlinear regression function `nls()` in R, and work directly with Kepler's model $T = \alpha D^{\beta}$, without having to linearize it first. However, there is one difficulty in using nonlinear regression. The `nls()` function uses a numerical method, which requires a *starting guess* for each of the parameters being fitted. If a starting guess is not sufficiently close to the actual value, the `nls()` routine may fail to converge. Starting guesses for each parameter being fitted are given as a list to the argument `start`. For Kepler's model, the signature of `nls()` is the following:

```
fit<-nls(T~alpha*D^beta,data = data.frame(D,T),start=list(alpha=1,beta = 1.5))
```

In the code below, we implement the nonlinear least squares fit with `nls()`, and plot in Figure 3, the data points (as red dots), and the fitted model (as a blue curve).

```
df<-data.frame(D,T) # bind D and T into a dataframe
fit<-nls(T ~ alpha*D^beta, data = df, start=list(alpha=1,beta = 1.5))
summary(fit)# summary of nonlinear least squars fit
```

```
## Formula: T ~ alpha * D^beta
## Parameters:
##       Estimate Std. Error t value Pr(>|t|)
## alpha 0.972270   0.009207   105.6  1.8e-12 ***
## beta  1.507312   0.002675   563.5  < 2e-16 ***
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
## Residual standard error: 0.2067 on 7 degrees of freedom
## Number of iterations to convergence: 3
## Achieved convergence tolerance: 1.713e-08
```

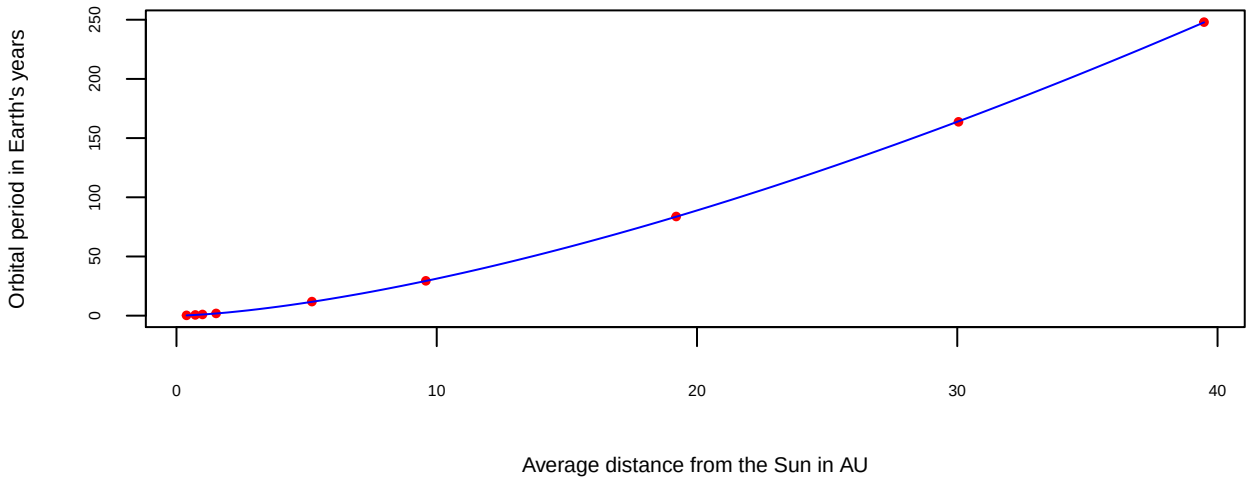

Figure 3: The planetary data and the model fitted with nonlinear least squares.

```
# plot the data points
plot(D,T,pch=20,cex=0.8,xlab="Average distance from the Sun in AU",
     ylab = "Orbital period in Earth's years",col = "red",cex.lab=0.7,cex.axis=0.5)
# add a plot of the fitted model over more refined distance vector d
d <- seq(min(D), max(D), length.out = 100)
lines(d,predict(fit, list(D = d)),col = "blue",lwd = 1)
```

For more details, see the section of the same title in the main PRIMUS article.

Solving ODE Systems with R Package `deSolve`

In this section, we focus on the computational aspects of numerically solving systems of linear and nonlinear first-order ODEs. The ODE system can be solved numerically using the R library `deSolve`, which has a number of general solvers for Initial Value Problems (IVP) of Ordinary Differential Equations (ODE), Partial Differential Equations (PDE), Differential Algebraic Equations (DAE), and Delay Differential Equations (DDE). See [11, 12, 13] for more details on `deSolve`. To download and install the library one can run the following code:

```
# only the first time the library is installed
install.packages("deSolve")
```

Once the library is installed, it must be loaded to be used. This can be done with `library(deSolve)`.

```
# once installed, load the library every time you want to use it
library(deSolve)
```

From the `deSolve` library, we use the solver `ode()`, which is really a wrapper around a number of ODE solvers implemented in FORTRAN and C. The signature of the ODE solver is the following:

```
ode(y,times,func,parms,method)
```

where the default method is `lsoda()`, used for solving initial value problems for systems of first-order ODEs. The R function `lsoda()` provides an interface to the FORTRAN ODE solver of the same name. The documentation of `deSolve` describes the arguments of `ode()`. For more details, see the main PRIMUS article, and the documentation of the `deSolve` package.

In the next sections, we illustrate how the `ode()` solver can be used to solve numerically linear and nonlinear systems of ODEs, as well as how to visualize the resulting numerical solutions.

The Lorenz Butterfly Simulation in R

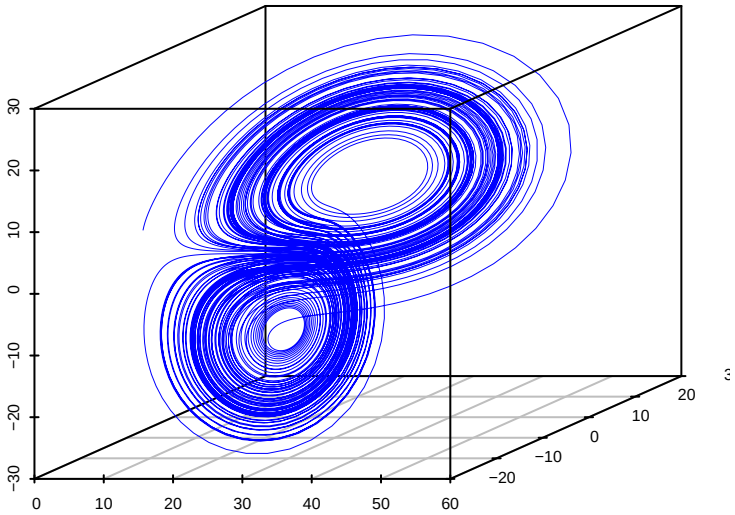

Figure 4: The Lorenz butterfly $(x(t), y(t), z(t))$.

The Lorenz model was created to represent idealized behavior of the Earth's atmosphere. The nonlinear system of ODEs in (4)-(6) implements the Lorenz model. This is a famous example of a system, investigated in many papers and presented in many texts, which leads to chaotic behavior, shown in Figure 4.

$$\frac{dx}{dt} = ax + yz \tag{4}$$

$$\frac{dy}{dt} = b(y - z) \tag{5}$$

$$\frac{dz}{dt} = cy - z - xy, \tag{6}$$

where x and y represent the horizontal and vertical temperature distribution, and z the convective flow. This first-order nonlinear system can be solved numerically using the `ode()` solver of the R library `deSolve`, described in the article Section *Solving ODE Systems with R package* ***deSolve***. The signature of the ODE solver is `ode(y,times,func,parms,method)`, and we can use it for solving IVP for systems of first-order ODEs. We use the following values for the parameters $a = -3$, $b = -10$ and $c = 30$, and initial conditions at time zero $x(0) = 1$, $y(0) = 2$ and $z(0) = 3$.

The function argument `func` is implemented as the R function `Der`, whose R code is given below, which computes the derivatives x', y' and z', specified by the system.

```
Der<- function(t, initials, parms) { # Derivative func argument for ode()
  with(as.list(c(initials,parms)), {
    dx <- a*x + y*z
    dy <- b*(y - z)
    dz <- c*y - z - x*y
    return(list(c(dx, dy, dz)))
  })}
```

Inside the body of our custom function `Der()`, we use the built-in R function `with(data,expr)`, which evaluates the R expression(s) `expr` in a local environment constructed from `data` that may be a list, a data frame etc. In the case of `Der()`, the data is specified by the list that comes from the vector `initials` with initial data, and the vector `parms` with parameter values. The call `c(initials,parms)` combines the two vectors. There are three R expressions for `dx,dy,dz` that are evaluated, using the numerical values of the named variables in the vector `initials` (`x,y,z`), and the named parameters in the vector `parms` (`a,b,c`). The list obtained from the vector `c(dx,dy,dz)` is returned. For example, the three expressions below return a list whose first element is the vector `c(dx,dy)` $= (7, 0)$. See the code below.

```
initials<-c(x = 3, y = 1)
parms<-c(a = 2, b = 3)
with(as.list(c(initials,parms)),{
  dx <- a*x + y
  dy <- x - b*y
list(c(dx,dy))
})
```

```
## [[1]]
## [1] 7 0
```

We solve the IVP for 100 time units, generating output every 0.01 time units, by passing the vector `times`, defined below, to the `times` argument of the `ode()` solver:

```
times<-seq(from=0,to=100,by=0.01) # a vector of time points
```

The vector `times` is created using the built-in R function `seq()`, which generates numbers from 0 to 100 with step 0.01. Note that the time step in the vector `times` does not affect the time step for the ODE integrator, which is determined by the solver. Let `initials` be the vector of initial values of x, y and z with name attributes, created using the built-in R function `c()`, which combines numbers into vectors:

```
initials<-c(x=1,y=2,z=3) # a vector with named state variables
```

In a similar way, we define the vector `parms`, which contains the numerical values of the named parameters.

```
parms<-c(a=-3,b=-10,c=30) # a vector with named parameters
```

Now, the ODE solver `ode()` is ready to be called with all required arguments supplied.

```
output<-ode(y=initials,times=times,func=Der,parms=parms)
```

The `ode()` solver returns the object `output` of class `deSolve`, which is a matrix whose first column always contains the values of the vector `times` and it is named `time`, and the other columns are the solution vectors. In this case, we have three other columns named `x,y,z`. We can see inside the matrix `output` by displaying the first 3 rows using the function `head()` (6 rows is the default value).

```
head(output,2)
```

```
##      time        x        y        z
## [1,] 0.00 1.000000 2.000000 3.000000
## [2,] 0.01 1.037009 2.122193 3.563496
```

The `deSolve` object `output` can be passed to the `plot()` method to get the plots of the solution vectors x, y, z over time. The call is very simple:

```
plot(output,col="blue") # plotting the solution vectors in output
```

We can also generate the 3D scatter plot of the solution vectors by using the `scatterplot3d()` function in the R library `scatterplot3d`, which must be installed first and then loaded to be used, similar to the `deSolve` library. We just need to extract the x, y and z solution vectors from the corresponding columns

of the matrix in `output`. For example, the x-vector can be extracted with `output[,"x"]`. Thus, we get the 3D scatterplot with the following R code:

```
# only the first time the library is installed
install.packages("scatterplot3d")
# load the library every time you want to use it
library(scatterplot3d)
scatterplot3d(output[,"x"],output[,"y"],output[,"z"],type = "l")
```

The results of the plotting calls are shown in Figure 4 and Figure 5, which visualize the solution of the ODE system. Figure 4 is the famous Lorenz butterfly.

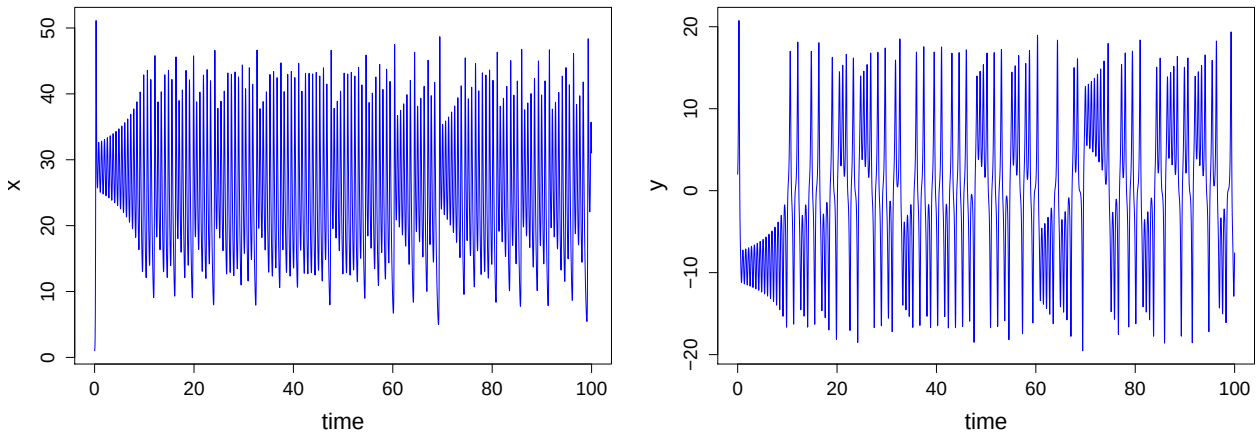

Figure 5: The $x(t)$ and $y(t)$ solution vectors against time.

Complete R Code Chunk for Simulating the Lorenz Butterfly

```
# Load libraries after installing them
library(deSolve)
library(scatterplot3d)
par(mfrow=c(2,2)) # 2 by 2 matrix plot
# parameters for the system of ODEs
parms<-c(a=-3,b=-10,c=30)
# initial data for the system of ODEs
initials <- c(x = 1, y = 2, z = 3)
# derivative function
Der<- function(t, initials, parms) {
  with(as.list(c(initials,parms)), {
    dx <- a*x + y*z
    dy <- b*(y - z)
    dz <- c*y - z - x*y
    return(list(c(dx, dy, dz)))
  })
}
# Solve the IVP for 100 time units, producing output every 0.01 time units
times <- seq(from = 0, to = 100, by = 0.01) # vector of time points
# Create an object of class deSolve
output <- ode(y = initials, times = times, func = Der, parms = parms)
# Plot the ode object
plot(output,col="blue",lwd=0.1,cex.main=2,cex.axis=1.4,cex.lab=1.8)
```

```
# Create the 3D scatterplot
scatterplot3d(output[,"x"],output[,"y"],output[,"z"],type = "l",lwd=0.1,
              angle=30,color="blue",xlab="",ylab="",zlab="",
              main="Lorenz attractor (x,y,z)",
              cex.main=2,cex.axis=1.4,cex.lab=1.8)
```

The Linear Kinetic System of First-Order Reactions

Typically, a chemical reaction consists of a number of steps coupled together. We consider a 3-step chain of first-order reactions, that is, without autocatalysis. We start with a reactant R, which produces reactant X, which produces reactant Y, which produces the final product Z. We can represent this chain of reactions as in (7), where the rates for each step of the chain are given above the arrows.

$$R \overset{ar}{\to} X \overset{bx}{\to} Y \overset{cy}{\to} Z \tag{7}$$

For this first-order kinetic scheme, we can write down the linear system of differential equations that specifies the time evolution of all concentrations r, x, y and z of R, X, Y and Z, respectively.

$$r' = -ar \tag{8}$$
$$x' = ar - bx \tag{9}$$
$$y' = bx - cy \tag{10}$$
$$z' = cy \tag{11}$$

where the constants a, b and c are all positive. Note that since we are considering a closed system, we must have $r(t) + x(t) + y(t) + z(t) = \text{const}$ at any time t. This, of course, implies that $r' + x' + y' + z' = 0$. This linear system can be integrated and analytical solutions obtained. We assume that initially the system contains only reactant R with initial concentration $r(0) = r_0$, $x(0) = 0$ and $y(0) = 0$. In particular, we have the following expressions for the concentrations r, x and y:

$$r(t) = r_0 e^{-at} \tag{12}$$
$$x(t) = \frac{ar_0}{b-a}(e^{-at} - e^{-bt}) \tag{13}$$
$$y(t) = \frac{abr_0}{b-a}\left(\frac{e^{-at} - e^{-ct}}{c-a} - \frac{e^{-bt} - e^{-ct}}{c-b}\right) \tag{14}$$

We leave it as an exercise for the reader to verify these expressions. It is instructive to also solve numerically the linear kinetic system given in (8)-(11), and compare the numerical solution with the exact analytical solution given in (12)-(14). For this purpose, we provide numerical values for all parameters and initial data: $a = 0.002$, $b = 0.08$, $c = 1$, $r_0 = 1/a$, $x_0 = 0$ and $y_0 = 0$.

We define the parameters and the initial data via two vectors with named values.

```
# parameters with named values
parms<-c(a = 0.002, b = 0.08, c = 1, r0 = 500)
# initial data with named values
initials<-c(x = 0, y = 0)
```

We can now use the `ode()` solver from the `deSolve` library to compute the numerical solution for this IVP. We need to specify the derivative function based on (8)-(10), given by the R code below.

```
Der<- function(t, initials, parms) {
  with(as.list(c(initials,parms)), {
    dx <- a*r0*exp(-a*t) - b*x
    dy <- b*x - c*y
    return(list(c(dx, dy)))
  })}
```

On the other hand, the analytical solution for $x(t)$ and $y(t)$ in (13)-(14) can be implemented by first creating a vector of time points `times`, and then obtaining the corresponding vectors `xt` and `yt` for x and y by using the vectorizing capabilities of R. In particular, the three lines of R code below implement this approach.

```
times <- seq(from = 1, to = 700, by = 0.1) # vector of time points
# Analytical solution vectors of the same size as times
xt<-a*r0/(b-a)*(exp(-a*times)-exp(-b*times))
yt<-a*b*r0/(b-a)*((exp(-a*times)-exp(-c*times))/(c-a)-(exp(-b*times)-exp(-c*times))/(c-b))
```

In Figure 6, we plot the numerical solution vectors for x and y against time, and in Figure 7, we plot the time evolution of the numerical solution vectors $(x(t), y(t), t)$, but we also superimpose the analytical solution vectors `(xt,yt,times)` to make sure that the numerical and analytical solutions are the same, within the default solver error. We plot the numerical solution vectors with blue dots and the analytical solution vectors with red dots, but it may be hard to distinguish them because they overlap, as they should.

For a short problem-based introduction to mathematical modeling in chemical engineering, we refer the reader to the UMAP Module [18].

Complete R Code Chunk for Simulating the Linear Kinetic System

```
library(deSolve) # must install deSolve first
library(scatterplot3d) # must install scatterplot3d first
par(mfrow=c(1,2))
# parameters
parms<-c(a=0.002,b=0.08,c=1,r0=500)
# initial data
initials <- c(x = 0, y = 0)
# derivative function
Der<- function(t, initials, parms) {
  with(as.list(c(initials,parms)), {
    dx <- a*r0*exp(-a*t) - b*x
    dy <- b*x - c*y
    return(list(c(dx, dy)))
  })
}
# We solve the IVP for 700 time units, producing output every 0.1 time unit
times <- seq(from = 1, to = 700, by = 0.1) # vector of time points
# an object of class deSolve
output <- ode(y = initials, times = times, func = Der, parms = parms)
# plotting the ode object sol
plot(output,col="blue",lwd=2,main="",xlab="",cex.main=0.8,cex.axis = 0.6,cex.lab = 0.8)
```

```
# Analytical Solution
# extract the parameter values
a<-parms["a"]; b<-parms["b"]; c<-parms["c"]; r0<-parms["r0"]
xt<-a*r0/(b-a)*(exp(-a*times)-exp(-b*times))
yt<-a*b*r0/(b-a)*((exp(-a*times)-exp(-c*times))/(c-a)-(exp(-b*times)-exp(-c*times))/(c-b))
```

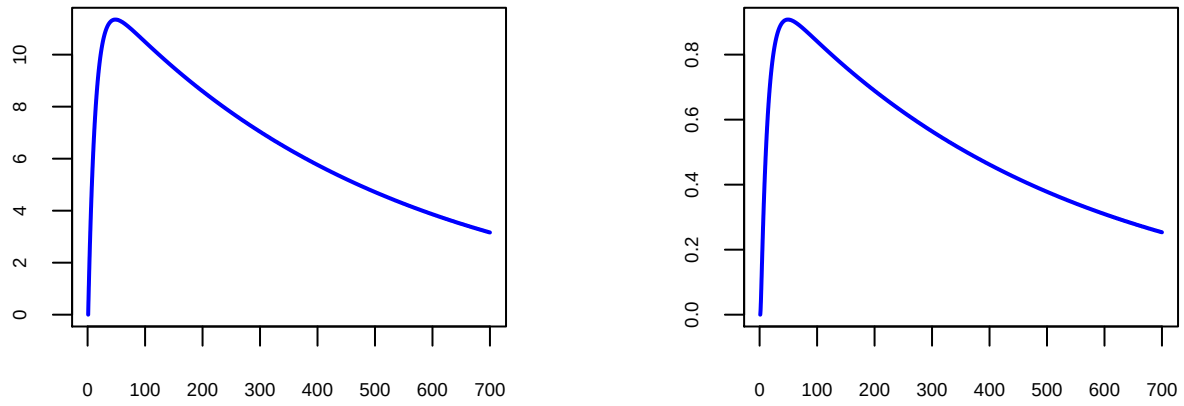

Figure 6: The $x(t)$ (left) and $y(t)$ (right) concentrations of the first-order kinetic system.

```
s3d<-scatterplot3d(output[,"x"],output[,"y"],output[,"time"],angle=60,pch=20,cex=0.9,
                   color="blue",xlab="x(t)",ylab="y(t)",zlab="time t",main="",
                   cex.axis = 0.5,cex.lab = 0.7)
s3d$points3d(xt,yt,times,pch=19,cex=0.2,col="red")
```

The Nonlinear Kinetic System of Higher-Order Reactions

Instead of focusing on a specific chemical reaction, we develop a general model for an autocatalytic process. See [16, 17] for a comprehensive introduction to nonlinear chemical reactions and chemical chaos. Consider an uncatalysed reaction in which species X is being converted into species Y at a rate of ax, where x is the concentration of species X, and a is some positive constant with appropriate units. We can represent this process as $X \overset{ax}{\to} Y$. In the case of an autocatalytic reaction, the species Y also plays the role of a catalyst for its own production. The rate at which species Y is being produced depends now on some power n of the concentration of Y (say y), which must be determined empirically or fitted to given data. We can represent the autocatalysis by the following diagram given in (15), with rate r at which Y is produced:

$$X + nY \to (n+1)Y, \quad r = \alpha x y^n \tag{15}$$

The rate r for species Y is proportional now to the product of the concentration x of species X, and the nth power of the concentration y of species Y, that is, $r = \alpha x y^n$. Here, α is again some positive constant with appropriate units. This empirical observation is often called *the Law of Mass Action*.

We can extend the linear model from the previous section into a simple isothermal autocatalytic model for a closed system. This example is discussed in some detail in [19, 20].

We start with reactant R, which is converted into a final product Z via two reactive intermediates X and Y. We consider a *cubic autocatalysis* with $n = 2$. We can represent this chain of reactions in a manner similar to (7), where the first-order rates for each step of the chain are given above the arrows, while below the arrows we give the higher-order rates.

$$R \overset{ar}{\to} X \underset{\alpha x y^2}{\overset{bx}{\rightrightarrows}} Y \overset{cy}{\to} Z \tag{16}$$

The autocatalytic part of this kinetic scheme is contained in the step $X \underset{\alpha x y^2}{\to} Y$, which represents the cubic autocatalysis $X + 2Y \to 3Y$. For this higher-order kinetic scheme, we can write down the nonlinear system

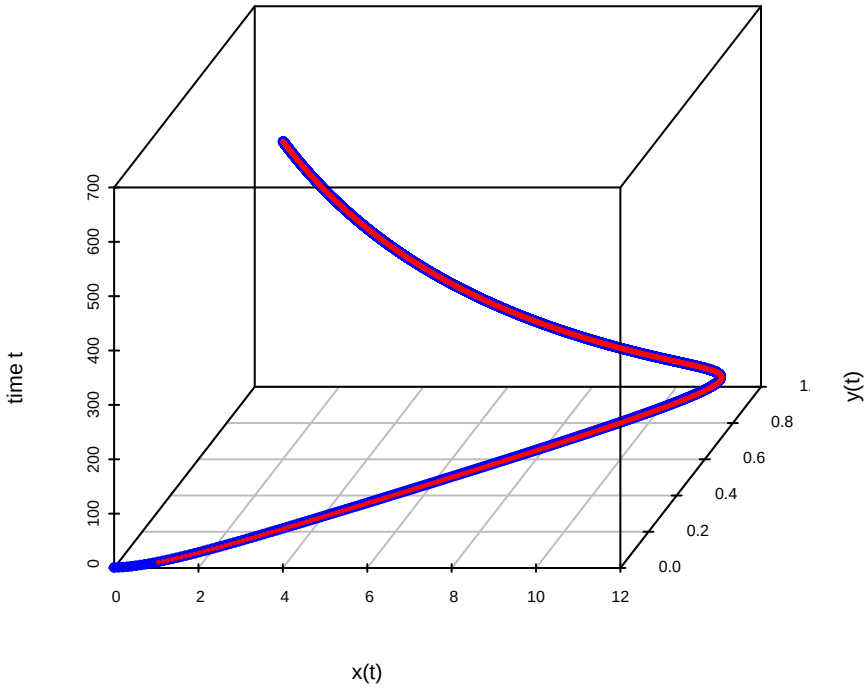

Figure 7: The time evolution $(x(t), y(t), t)$ of the concentrations for two reactive intermediates X and Y, for the linear first-order kinetic system.

of differential equations that specifies the time evolution of all concentrations r, x, y and z of R, X, Y and Z, respectively.

$$r' = -ar \tag{17}$$
$$x' = ar - bx - \alpha xy^2 \tag{18}$$
$$y' = bx - cy + \alpha xy^2 \tag{19}$$
$$z' = cy \tag{20}$$

The nonlinear kinetic system in (17)-(20) has no known analytical solution, but we can solve it numerically. For this purpose, we use the same numerical values for all parameters and initial data, as in the linear case: $a = 0.002$, $b = 0.08$, $c = 1$, $r_0 = 1/a$, $x_0 = 0$ and $y_0 = 0$. We also set the new parameter $\alpha = 1$. We can now use the `ode()` solver from the `deSolve` library to compute the numerical solution for this IVP.

The complete R code for obtaining and plotting the numerical solution of the simulated nonlinear kinetic system is given in the next subsection. In Figure 8, we plot the concentrations $x(t)$ and $y(t)$ of the nonlinear kinetic system as functions of time. In Figure 9, we plot the time evolution $(x(t), y(t), t)$ of the concentrations of the two reactive intermediates X and Y in the nonlinear kinetic system.

For more details, see the section of the same title in the main PRIMUS article.

Complete R Code Chunk for Simulating a Nonlinear Kinetic System

```
library(deSolve) # must install deSolve first
library(scatterplot3d) # must install scatterplot3d first
# parameters for the system of ODEs
```

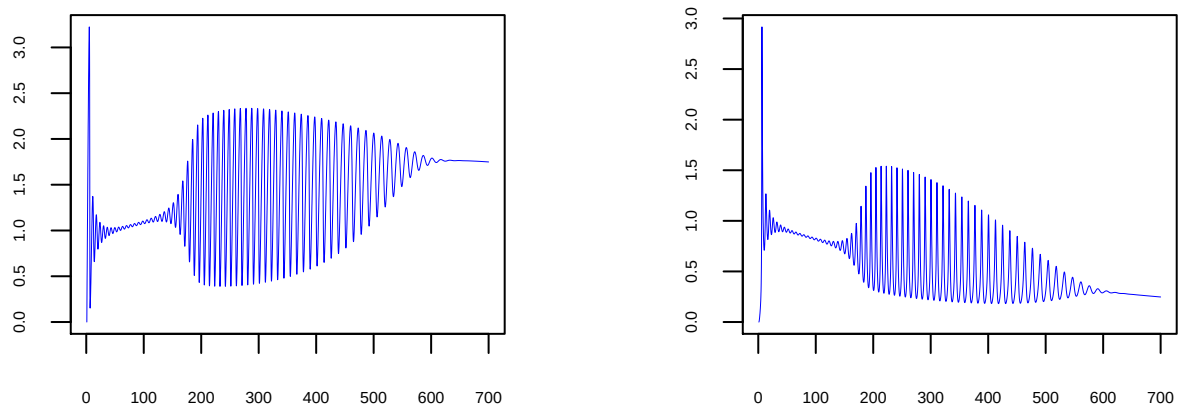

Figure 8: The $x(t)$ (left) and $y(t)$ (right) concentrations of the nonlinear kinetic system.

```
parms<-c(a=0.002,b=0.08,c=1,r0=500,alpha=1)
# initial data for the system of ODEs
initials<-c(x = 0, y = 0)
# derivative function
Der<- function(t, initials, parms) {
  with(as.list(c(initials,parms)), {
    dx <- a*r0*exp(-a*t) - b*x - alpha*x*y^2
    dy <- b*x - c*y + alpha*x*y^2
    return(list(c(dx, dy)))
  })}
# Solve the IVP for 700 time units, producing output every 0.1 time units
times <- seq(from = 1, to = 700, by = 0.1) # vector of time points
# Create an object of class deSolve
output <- ode(y = initials, times = times, func = Der, parms = parms)
par(mfrow=c(1,2)) # 1 by 2 matrix plot
# Plot the ode object
plot(output,col="blue",lwd=0.1,main="",cex.main=0.8,xlab="",cex.axis=0.5,cex.lab=0.6)
```

```
# Create the 3D scatterplot
par(mfrow=c(1,1)) # 1 by 1 matrix plot
scatterplot3d(output[,"x"],output[,"y"],output[,"time"],type="l",lwd=0.1,angle=60,
              color="blue",xlab="x(t)",ylab="y(t)",zlab="time t",cex.axis=0.5,cex.lab=0.7)
```

Python Simulation of the SIR Model

For more details on *Running Python in R Markdown*, inside RStudio, see the subsection of the same title.

An IVP can be solved numerically in Python using the `scipy` module. The `scipy` module is included in the Anaconda distribution of Python and is available for use in CoCalc. For this example, we will be using the solver `odeint`, which is also a wrapper for the `lsoda` solver from the FORTRAN library `odepack`. See [14] for more details. To access and use the function `odeint`, one must load the appropriate modules first:

```
# necessary modules
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import odeint
```

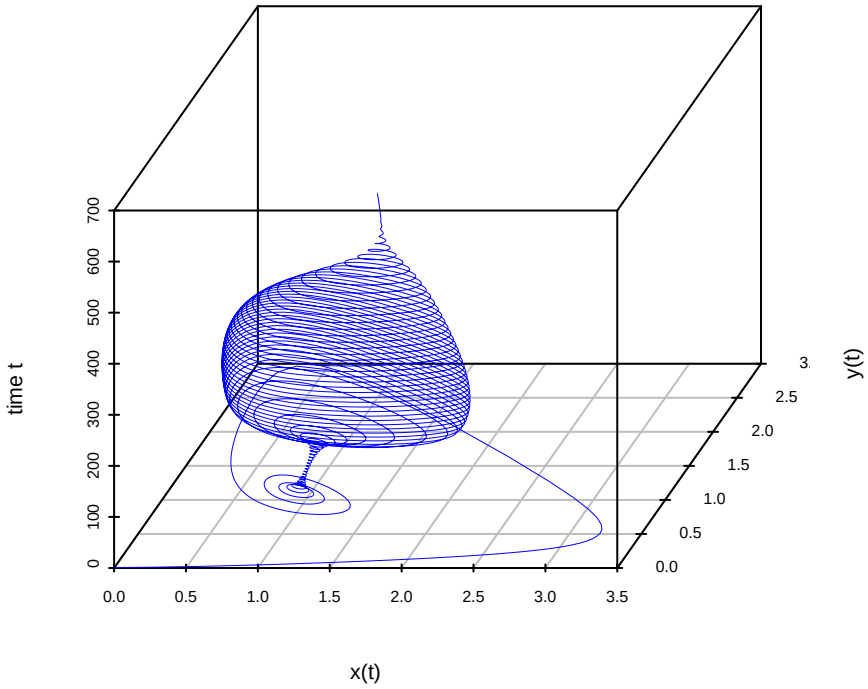

Figure 9: The time evolution $(x(t), y(t), t)$ of the concentrations for two reactive intermediates X and Y for the nonlinear kinetic system.

The modules `numpy` and `matplotlib` are needed to generate appropriate inputs and plots when working with `odeint`. Both are commonly used modules in Python scientific computing and are also included in the Anaconda distribution and available for use in CoCalc.

The function `odeint` has a rather long signature. Here we will concentrate on specifying the minimum number of arguments needed for our application. In particular, we call the solver using only three arguments:

```
odeint(func, y0, t)
```

where the three arguments given above are described in the online documentation [14] as follows:

`func` Callable(`y`, `t0`, ...). Computes the derivative of `y` at `t0`.

`y0` Initial condition on `y` (can be a vector).

`t` A sequence of time points for which to solve for `y`. The initial value point should be the first element of this sequence.

We are now ready to apply the `odeint` solver to a specific example. A relatively simple compartmental model that is often used to model the spread of an infectious disease is the SIR model. This is a model in epidemiology that separates a population affected by disease into three compartments:

1. Susceptibles - the portion of the population which may be infected.
2. Infected - the portion of the population that is currently infected.
3. Recovered - the portion of the population that has recovered from the disease or developed an immunity.

In this example, we will use a simple SIR model of a generic, non-lethal disease as seen in [15]. For this model, s will represent the proportion of the population that is susceptible to infection, i will represent the proportion of the population that is infected, and r will represent the proportion of the population that has recovered from the infection. Note that we are assuming that $s + i + r = 1$ and that time is measured in days. The model is given by the following nonlinear system of ODEs:

$$\frac{ds}{dt} = -bsi \tag{21}$$
$$\frac{di}{dt} = bsi - ki \tag{22}$$
$$\frac{dr}{dt} = ki, \tag{23}$$

where b is the rate constant for the transmission of the infection from infected to susceptibles and k is the rate constant for recovery from the infection. First we define our constants. In this example we will use the constant values $b = 0.9$ and $k = 0.33$.

```
# constants
b, k, finalt, dt = 0.9, 0.33, 50, 0.01
```

The constant `finalt` represents the final time for our simulation, while `dt` represents the time step size in our discrete approximation of this continuous model. Making `dt` small can lead to better simulations at the cost of additional computational time.

Next, we use the `numpy` function `linspace` to create an array `t` containing the sequence of time points for which to solve the model. The `linspace` function accepts three arguments, a starting value, an ending value, and the total number of equally spaced points from the starting value to the ending value. In this case, since we set `finalt = 50` and `dt = 0.01`, our array represents a period of 50 days subdivided into 5000 equally spaced times.

```
t = np.linspace(0, finalt, int(finalt/dt))
```

For this IVP, we will simulate the result of an outbreak of a disease in a population of 3,000 individuals where 20 people are infected.

```
# initial conditions
S0, I0, R0 = 2980, 20, 0
N = S0+I0+R0
y0 = [S0/N,I0/N,R0/N]
```

We can now construct our callable function to represent the system given by (21)-(23). For the callable function needed by `odeint`, we create a function `func(y,t0)` which simply returns the values of `ds`, `di`, and `dr` at time `t0`. Here, `y` is an array containing the values of `s`, `i`, and `r` at time `t0`.

```
# dynamical system
def func(y,t0):
    s, i, r = y
    ds = -b*s*i
    di = b*s*i-k*i
    dr = k*i
    return [ds,di,dr]
```

We can now call our main solver function:

```
sol = odeint(func, y0, t)
```

which returns an array where each entry is an array containing the values of s, i and r at each time point given by `t`. By storing the output of `odeint` in a variable `sol`, we can plot the numerical result of our simulation using the `matplotlib` module.

```
# plot
plt.plot(t,sol)
plt.ylabel('Proportion')
```

```
plt.xlabel('Days')
plt.legend(['Susceptibles','Infected','Recovered'],loc='best')
plt.show()
```

The result of our simulation, shown in Figure 10, suggests that over a period of 30 days, the majority of the population will become infected and recover from the disease.

Complete Python Code Chunk for Simulating the SIR Model

```
# necessary modules
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import odeint
# constants
b, k, finalt, dt = 0.9, 0.33, 50, 0.01
t = np.linspace(0, finalt, int(finalt/dt))
# initial conditions
S0, I0, R0 = 2980, 20, 0
N = S0+I0+R0
y0 = [S0/N,I0/N,R0/N]
# dynamical system and solver functions
def func(y,t0):
    s, i, r = y
    ds = -b*s*i
    di = b*s*i-k*i
    dr = k*i
    return [ds,di,dr]
sol = odeint(func, y0, t)
# plot
plt.plot(t,sol)
plt.ylabel('Proportion')
plt.xlabel('Days')
plt.legend(['Susceptibles','Infected','Recovered'],loc='best')
plt.show()
```

References

[1] *SIMIODE*. https://www.simiode.org/. Accessed January 17, 2018.

[2] *CODEE*. http://ptraci.math.hmc.edu/. Accessed Jan. 17, 2018.

[3] RStudio Version 1.1.383. http://www.rstudio.com. Accessed January 17, 2018.

[4] R Markdown: Creating Dynamic Documents in RStudio. http://rmarkdown.rstudio.com/. Accessed January 17, 2018.

[5] The R Project for Statistical Computing. R version 3.4.1 (2017-06-30). http://www.r-project.org. Accessed January 17, 2018.

[6] A (Very) Short Introduction to R. http://cran.r-project.org/doc/contrib/Torfs+Brauer-Short-R-Intro.pdf. Accessed January 17, 2018.

[7] Kostadinov, B. 2013. Simulation insights using R. *PRIMUS*, 23(3): 208–223. DOI: 10.1080/10511970.2012.718729

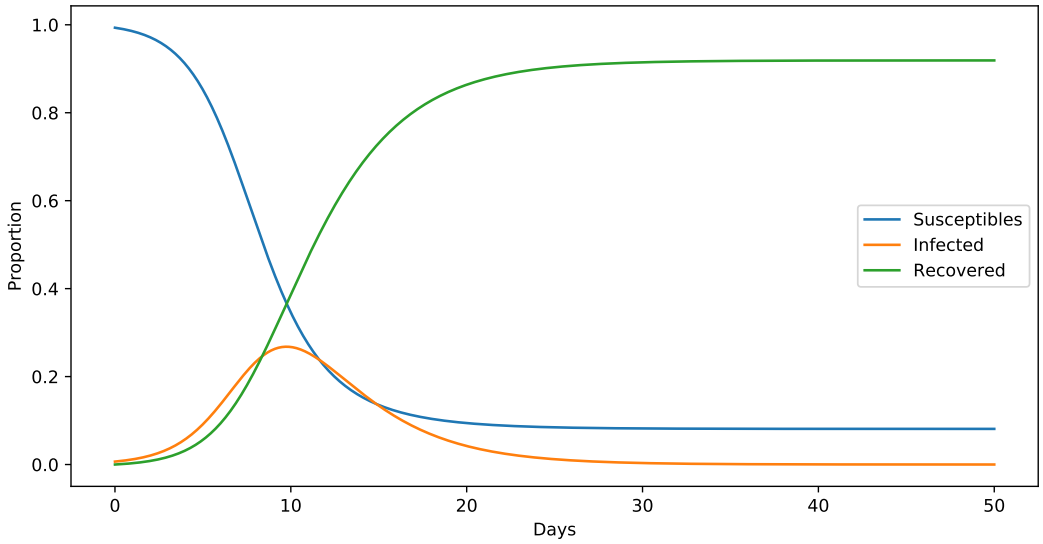

Figure 10: The population proportions $s(t)$, $i(t)$, and $r(t)$ over 50 days.

[8] Benakli, N., Kostadinov, B., Satyanarayana, A. and S. Singh 2016. Introducing computational thinking through hands-on projects using R with applications to calculus, probability and data analysis, *Int. J. Mathematics Education in Science and Technology* 48(3): 393–427. DOI: 10.1080/0020739X.2016.1254296

[9] Kostadinov, B. 2018. Predicting the Next US President by Simulating the Electoral College. *Journal of Humanistic Mathematics*, 8(1): 64–93. DOI: 10.5642/jhummath.201801.05. Available at: http://scholarship.claremont.edu/jhm/vol8/iss1/5

[10] NASA planetary data. https://nssdc.gsfc.nasa.gov/planetary/factsheet/. Accessed January 25, 2018.

[11] Soetaert, K., Petzoldt, T. and R. W. Setzer 2009. **deSolve**: General Solvers for Initial Value Problems. R package version 1.2. http://CRAN.R-project.org/package=deSolve. Accessed January 17, 2018.

[12] Soetaert, K., Petzoldt, T. and R. Setzer 2010. Solving Differential Equations in R: Package deSolve. *J. Statistical Software* 33(9).

[13] Soetaert, K., Cash, J. and F. Mazzia 2012. *Solving Differential Equations in R*. Berlin: Springer.

[14] Documentation for Python ODE Solver `scipy.integrate.odeint`. https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.odeint.html. Accessed February 2, 2018.

[15] Smith, D. and L. Moore 2004. *The SIR Model for Spread of Disease*. MAA Convergence. https://www.maa.org/press/periodicals/loci/joma/the-sir-model-for-spread-of-disease Accessed February 8, 2018.

[16] Gray, P. and S. K. Scott 1990. *Chemical Oscillations and Instabilities*. Oxford: Clarendon Press.

[17] Scott, S. K. 1994. *Chemical Chaos*. Oxford: Clarendon Press.

[18] Nagarkatte, U. and U. R. Hattikudur 1996. Mathematical Modeling in Chemical Engineering. *The UMAP Journal*, 17(2): 97–109.

[19] Borrelli, R. and C. Coleman 2004, 2nd ed. *Differential Equations: A Modeling Perspective*. New York: John Wiley & Sons.

[20] Borrelli, R., Coleman C. and W. Boyce 1992. *Differential Equations Laboratory Workbook: A Collection of Experiments, Explorations and Modeling Projects for the Computer*. New York: John Wiley & Sons.