Citation: CPT Pharmacometrics Syst. Pharmacol. (2015) 4, e21; doi:10.1002/psp4.21
© 2015 ASCPT All rights reserved

TUTORIAL

Interactive Pharmacometric Applications Using R and the Shiny Package

J Wojciechowski, AM Hopkins and RN Upton*

Interactive applications, developed using Shiny for the R programming language, have the potential to revolutionize the sharing and communication of pharmacometric model simulations. Shiny allows customization of the application's user-interface to provide an elegant environment for displaying user-input controls and simulation output–where the latter simultaneously updates with changing input. The flexible nature of the R language makes simulations of population variability possible thus promoting the combination of Shiny with R in model visualization.
CPT Pharmacometrics Syst. Pharmacol. (2015) **4**, e21; doi:10.1002/psp4.21; published online 18 March 2015

A population pharmacokinetic and/or pharmacodynamic model can be thought of as an elegant and concise description of the underlying mechanisms and variability in a data set. A robust simulation model is an important tool for predicting new data—the consequences of changes in the dosing regimen or covariate influences. Population models have found important roles in the development, regulation, and optimal use of pharmaceuticals.[1,2] However, the process of making predictions from population models has been largely time-consuming, and remained the province of dedicated pharmacometricians using specialized software limiting its wider applicability.[3]

Recent developments such as the ggplot2[4,5] package for the R data analysis and statistical language[6] have allowed sophisticated and flexible plotting of data and population model output. However, the plots are static and to investigate different values for model parameters the models need to be re-simulated and plots updated manually. The Berkeley Madonna software provides an alternative approach where models can be specified as differential equations and the simulated results shown in real-time for different parameter values by use of sliders.[7] While Berkeley Madonna has found useful roles in presenting model predictions to nonpharmacometricians, it is proprietary software (albeit inexpensive) and is not readily adapted to simulations of variability.

Recently, developments in the R language, and in particular the Shiny package for R,[8] have allowed R programmers to interactively show the output for R programs to Web-browsers. Given the general nature of the R language, it is possible to program interactive pharmacometric models with a Shiny Web-browser interface that can be viewed on the localhost (user's own computer) or on another computer accessed by means of the Internet. Applications such as a dosing-education tool aimed at high school students (https://acp-unisa.shinyapps.io/OpenDay/) and a population model simulation tool complete with simulated variability (https://acp-unisa.shinyapps.io/Ibuprofen/) have been developed using the Shiny package and R and can be viewed without an R installation or files containing R code. Unlike other Web-page design methods, only previous experience with the R programming language is required. R and Shiny maintain Berkeley Madonna's key feature of reactively updating output in response to changing input by means of widgets (such as sliders and radio buttons) but owing to the flexible R language and combination with extension packages, the pharmacometrician has control in coding all elements of a population model, the appearance of the application's user-interface, and generated output. We believe such Shiny Web applications have the potential to provide a convenient method of providing model simulations to a broad audience, which may be useful to pharmacometricians and nonpharmacometricians alike.

The aims of this tutorial are to:

- Introduce the application of the Shiny package to pharmacometric model simulations
- Describe the fundamental R code needed to create a Shiny Web application
- Show example code for the three Web applications of increasing complexity
- Briefly describe the various options for deploying Shiny Web applications

GETTING SHINY

This tutorial is primarily targeted toward intermediate/experienced R programmers such as those who use R for processing raw data and NONMEM® output, statistical analyses, and have some experience in coding pharmacometric models in R (nonetheless, an annotated example script for coding a population model in R is available as **Supplementary Pharmacometric Model Example**). It is not recommended to learn R and Shiny concurrently. R is an open source data analysis language and can be downloaded from http://www.r-project.org. There are several online resources for learning R. Shiny is a package for R

Australian Centre for Pharmacometrics, School of Pharmacy and Medical Sciences, University of South Australia, Adelaide, Australia. *Correspondence: R Upton (richard.upton@unisa.edu.au)
Received 17 October 2014; accepted 24 January 2015; published online on 18 March 2015. doi:10.1002/psp4.21

Table 1 Software versions used in development of example material

Software/R Package Library	Version
R	3.1.2
RStudio	0.98.977
Google Chrome	39.0.2171.99
Shiny	0.11
ggplot2	1.0.0
deSolve	1.11
plyr	1.8.1
compiler	3.1.2
doParallel	1.0.8
reshape2	1.4.1
grid	3.1.2
MASS	7.3–37
MBESS	3.3.3

developed by RStudio, and needs to be installed in a given R installation. There are several ways to install packages in R depending on the operating system and R interface. One convenient method is to install the RStudio integrated development environment for R (http://www.rstudio.com/) and use the Tools/Install Packages menu. RStudio will automatically install any additional package dependencies. A list of software and R package versions used in the development of this tutorial's examples is available in **Table 1**. For those less familiar with R, it is important to note that R and its package libraries develop and update at a fast pace. Therefore, in the future the latest versions of the packages used in this tutorial may not be backward compatible with the code presented here.

USING SHINY

Shiny applications are built using two R scripts that communicate with each other: a user-interface script (ui.R), which controls layout and appearance; and a server script (server.R), incorporating instructions for user-input, processing data, and output by utilizing the R language and functions from user-installed packages. Tutorials and exercises for building Shiny applications are provided by RStudio on the Shiny website[9] and are accompanied by articles that describe capabilities of Shiny beyond the scope of the tutorials, a reference page for Shiny functions, as well as a gallery of examples supported by code. The Shiny package also includes 11 built-in examples referred to by the Shiny by RStudio tutorials.

This tutorial provides three working pharmacometrics oriented examples of Shiny Web applications (available online as Supplementary Instructions and Supplementary Examples S1-S3). To enhance a reader's understanding of how to write a Shiny application, it is recommended that this tutorial be read in conjunction with the supplementary code available online and running the applications locally using R or by means of the Internet. They can be accessed at the following Web addresses:

- Flip-Flop Kinetics Application (Example 1): https://acp-unisa.shinyapps.io/FlipFlop/
- Preliminary Patient Education Tool Application (Example 2): https://acp-unisa.shinyapps.io/OpenDay/
- Ibuprofen in Pre-Term Neonates Application (Example 3): https://acp-unisa.shinyapps.io/Ibuprofen/

To run any of the applications locally in R or RStudio, an installation of the Shiny package (and any dependencies) is required, and ui.R and server.R scripts for the application saved in the same directory (i.e., a folder titled with the application's name). To launch the application from RStudio open each ui.R and server.R script in RStudio and click "RunApp," which will appear in the top right hand corner of the source pane. Launching an application from R requires setting the working directory to where the application folder is located, and using the runApp function. When initiating, Shiny will open a Web-browser window for the application.

STRUCTURE OF A SHINY APPLICATION

In this section, we will discuss the core elements of the ui.R and server.R scripts required to create an application. Our first example for creating a user-interface in ui.R and application instructions in server.R is the Flip-Flop Kinetics application—Example 1 (supplementary code available online). This application implements a model for a hypothetical drug described by one-compartment first-order absorption kinetics. It illustrates the concept that for this type of model, there are two parameter sets that give identical results: one with absorption rate faster than elimination rate, and the other with elimination rate faster than absorption rate (flip-flop kinetics).[10]

Figure 1 features a screenshot of the application's browser window with a plot of the concentration–time profile for the drug over a 24-h period, with sliders that control the values for the absorption and elimination rate constants, k_a and k_e, respectively, and volume of distribution, V. Initially (**Figure 1**), the application's plot shows an example where elimination is rate limiting as the value of k_a is greater than k_e ($k_a = 2$, $k_e = 0.2$). When the sliders change, thus allowing the values of the rate constants to approach each other, the plot automatically updates in response. When k_a is less than k_e, the roles of the constants in describing the concentration–time profile swap and the absorption rate constant limits the decline of drug. For example, pairs of values such as $k_a = 0.2$ and $k_e = 2$ when flipped to correspond with the other rate constant (and V is also changed accordingly), do not change the outcome of the plot (**Figure 1**). Rather than a series of simulations and static plots to explore this system, Shiny has allowed the user to control and observe the roles of the rate constants simultaneously.

The ui.R script (supplementary code available online) encodes instructions for the application's layout, appearance, user-input widgets (interactive Web elements such as sliders, buttons, selection boxes, check boxes, etc.), and the output to be displayed. The main elements describing the user-interface of this application are shown below:

```
fluidPage(fluidRow(
h2("Flip-Flop Kinetics"),
```

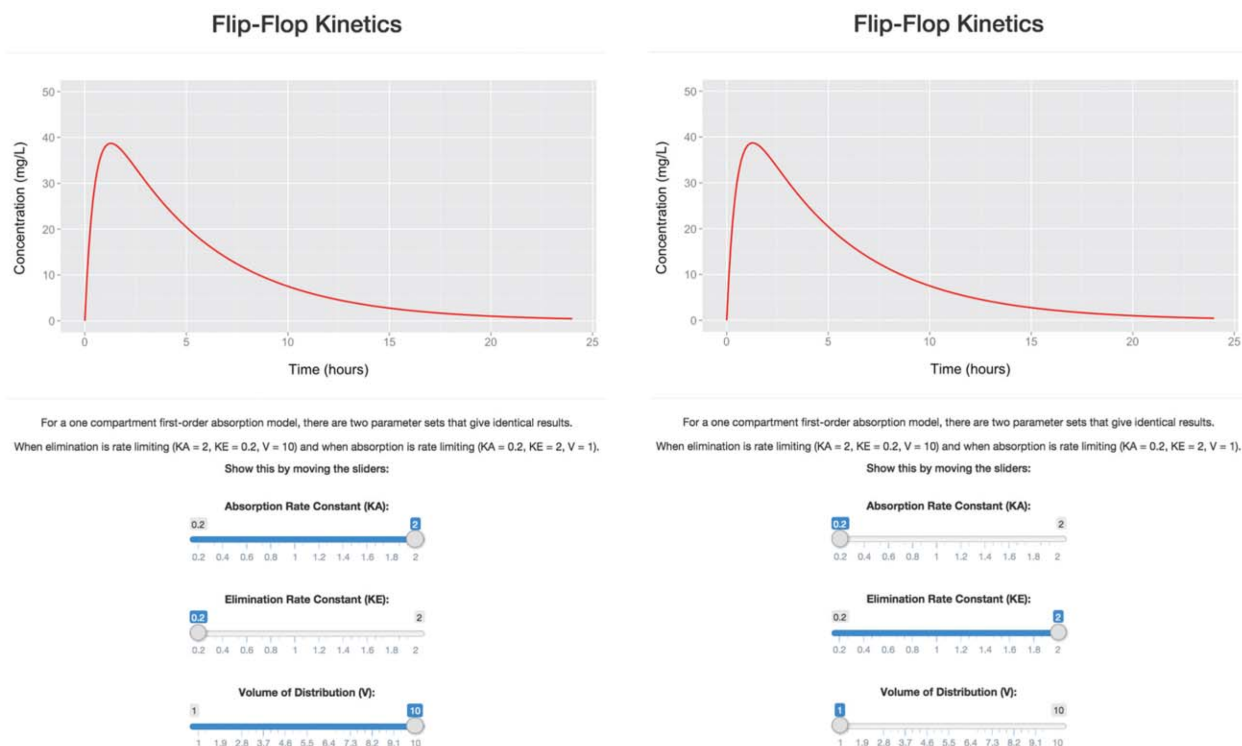


Figure 1 Flip-Flop Kinetics application. Shown are screenshots of the application featuring a one-compartment first-order absorption model running in a browser-window. The calculated concentration–time profile over 24 h is plotted as a red solid line. *Left:* Upon application initiation, elimination is rate limiting, with slider values of absorption rate constant (KA) = 2, elimination rate constant (KE) = 0.2, volume of distribution (V) = 10. *Right:* The slider values for KA and KE have been reversed where KA = 0.2 and KE = 2, i.e., absorption is rate limiting. For pairs of values to give identical results, the slider for V also needs to be changed (V is a scalar that allows the same numerical solution).

```
  plotOutput("plotCONC"),
  sliderInput("KA", "Absorption Rate Constant:", min = 0.1, max = 2.1, value = 2, step = 0.1),
  sliderInput("KE", "Elimination Rate Constant:", min = 0.1, max = 2.1, value = 0.2, step = 0.1),
  sliderInput("V", "Volume of Distribution (V):", min = 1, max = 10, value = 10, step = 0.05),
  align = "center"))
```

All code for the contents of the user-interface is required to be within the brackets of a layout function. Layout functions such as fluidPage (there a number available) create a canvas for the interface and use fluidRow to position user-input widgets (sliderInput—a function for creating a slider) and output (plotOutput—function for plot object). Each layout function has its own framework for positioning elements where others, including fixedPage and navbarPage, can create different styled pages based on their own functions. However, they all follow the same hierarchical structure where element functions (such as widgets—sliderInput) are embraced within a positioning function (such as fluidRow), within a layout function (fluidPage). Functions of the same level are written as a string within their superior function, separated by ",".

Error messages related to Shiny (and other loaded R packages) appear in the loaded Web browser upon application initiation, and once the application is terminated they also appear in the R Console. Examining if pairs of function brackets are closed before running the application can help limit a large number of error messages. In particular, functions placed earlier in the script are more difficult to track when ui.R becomes more detailed. Using free source code editor software, or directly writing scripts from RStudio, is encouraged when writing Shiny applications as they can aid in highlighting any unclosed brackets (among other navigational and editing benefits). If there is a major error in which the application is not functional a Web-browser page will still open, however, will be grayed-out. Other error messages generally provide a line number referring to the source code or the function name in question. When testing the coding of a pharmacometric model, it is recommended to write a generic R script ensuring that it runs successfully before incorporating the model into a Shiny application.

The structure of server.R code plays an important role in defining instructions for the application, while minimizing redundant computation and maximizing application speed (**Figure 2**). The function, shinyServer, requires an input object (values called *from* ui.R) and an output object (objects to be called *by* ui.R). Objects that change depending on input from widgets in the ui.R, such as input$KA in

```
#Define server logic required to for FlipFlopPKApp
    shinyServer(function(input, output) {

#Generate a plot of the data
#Also uses the inputs to build the plot (ggplot2 package)
    output$plotCONC <- renderPlot({

        #Collect input from user-widgets
        KA <- input$KA
        KE <- input$KE
        V  <- input$V

        #Calculate function of Dose, V, KA and KE (where F = 1)
        if (KA==KE) A <- 0 else A <- (D*KA)/(V*(KA-KE))

        #Calculate concentration
        CONC <- A*exp(-KE*TIME) - A*exp(-KA*TIME)

        #Create a dataframe of time and concentration data
        conc.data <- data.frame(TIME = TIME, CONC = CONC)

        #ggplot2 object
        plotobj <- ggplot(data = conc.data)
        plotobj <- plotobj + geom_line(aes(x = TIME, y = CONC), colour = "red", size = 1)
        plotobj <- plotobj + scale_y_continuous("Concentration (mg/L) \n", lim = c(0,50))
        plotobj <- plotobj + scale_x_continuous("\nTime (hours)", lim = c(0,24))
        print(plotobj)

    })

})
```

Figure 2 server.R excerpt from flip-flop kinetics application. The output object, plotCONC, uses the renderPlot function to call widget values for absorption rate constant (KA), elimination rate constant (KE), and volume of distribution (V) from the user-interface, creates a data frame consisting of drug concentrations (CONC) at each specified time-point (TIME; time sequence not shown), and plots the concentration–time profile using the ggplot2[5] package. As output and the renderPlot function are reactive, they are placed within shinyServer.

this example, are termed "reactive." Every time input from a widget changes, the value for the reactive object updates to reflect this change.

Expressions for defining and processing reactive objects are required to be enclosed within a render* function (for returning reactive output to the user-interface, where * denotes an object description such as "plot," "table," or "text") or a reactive expression (often for controlling a, or series of, reactive data frames that can be sent to the user-interface by render* functions). The renderPlot function (**Figure 2**) containing reactive input objects (KA, KE, and V from sliders), expressions for calculating concentration (CONC), and a ggplot2[5] object plotting resultant concentrations over time (plotobj), will re-execute with every reaction to a widget change and will store the updated plot object as plotCONC for the output object. Detailed code can significantly slow down the application; therefore, it is best to limit code within render* and shinyServer functions to just that is reactive or computationally simple. Code at the beginning of the script before shinyServer is only run once on application initiation. This area is the most appropriate place for loading libraries, source code, datasets, and constant expressions (such as defining time) that do not need to be called each occasion when input from a widget changes, thus limiting redundant computation time. These functions and libraries can also be stored in a third script called global.R.

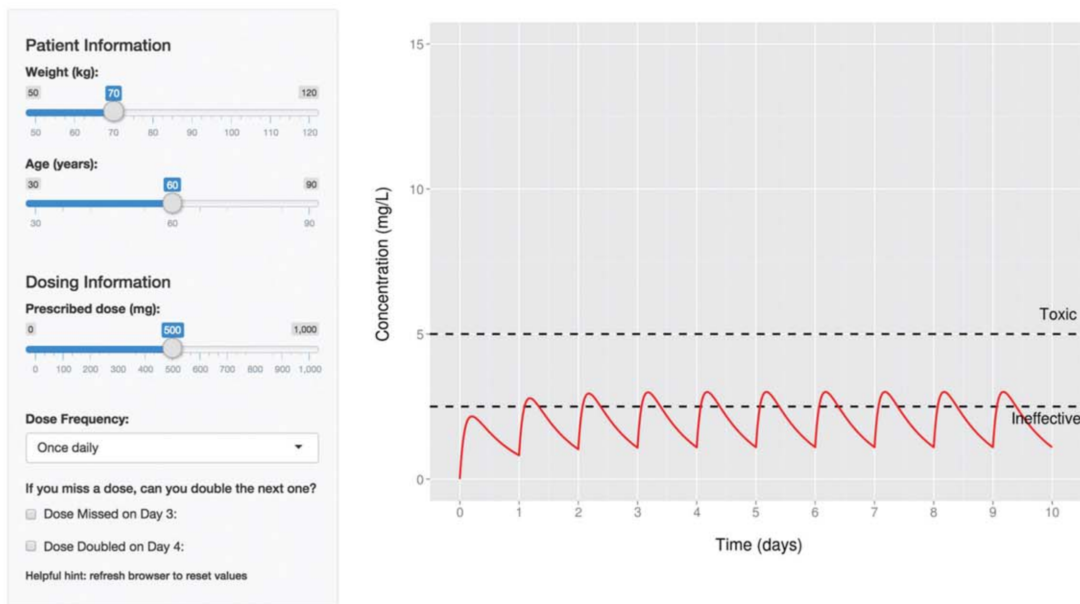
BUILDING A USER-INTERFACE (ui.R)

Example 2 is a Shiny application that incorporates several layout functionalities and widgets in its ui.R script (supplementary code available online). Shiny's variety of customizable layouts and prebuilt widgets allows easy development of user-interfaces for applications. From the options available, developers can explore layouts that adapt to the different browser sizes of devices (i.e., phone, tablet, or computer), add a sidebar or tabs that organize and differentiate between input (widgets) and output (tables, plots, etc.), or change what is visible when specific input conditions are met. The example application implements a simple patient education tool developed for an Open Day at a university.

Figure 3 features a screenshot of the application's browser window with a plot of the 10-d concentration–time profile for a hypothetical orally administered drug with one-compartmental first-order absorption kinetics affected by patient age (on clearance) and weight (on volume of distribution). When the slider values for age, weight, and dose change, or when a dose frequency is selected, the plot of the concentration–time profile is updated. The application was intended to provide final year high school students and their parents with no background in pharmacy or pharmacokinetics an understanding of why some drugs require altered dosing regimens for different ages and weights to prevent subsets of patients experiencing toxicity or lack of efficacy. The students and their parents were able to also explore profiles with missed or doubled doses by means of integrated check boxes.

Layout

Shiny calls elements in the ui.R script sequentially, i.e., from top to bottom, and left to right. Within a layout function, elements inside are ordered in the sequence they are to appear in the user-interface. **Figure 3** features a fixedPage layout—a page layout that aligns elements, such as input control widgets or output plots and text, in rows and columns in a fixed width. Rows (fixedRow) allow embraced



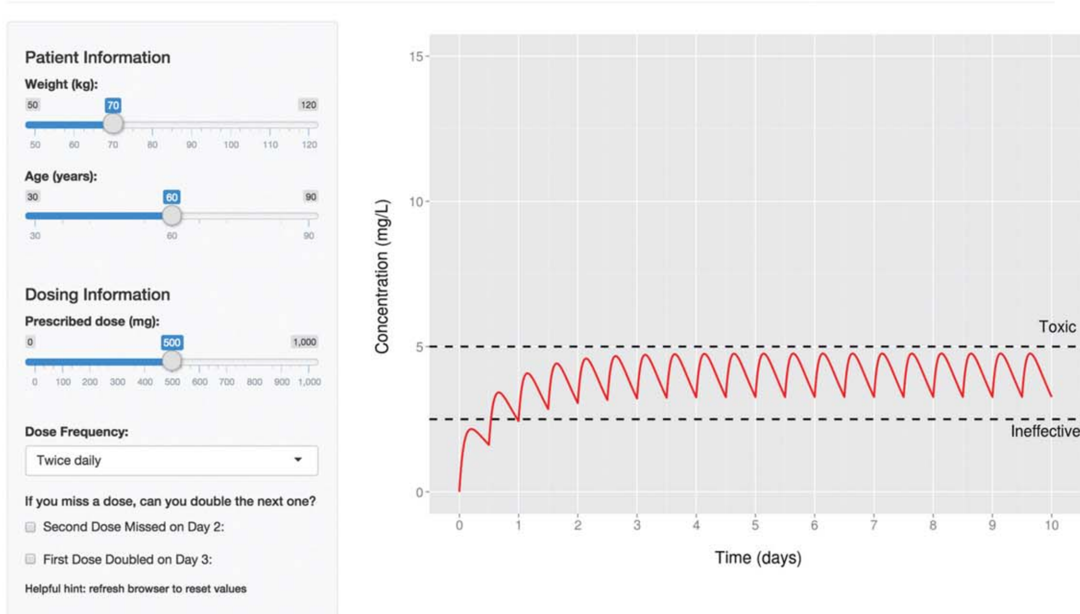


Figure 3 Preliminary patient education tool application. Shown are screenshots of the application featuring a one-compartment first-order absorption model running in a browser-window. The concentration–time profile is depicted as the solid red line, and black horizontal dashed lines represent the hypothetical therapeutic window (drug is ineffective at less than 2.5 mg/L, and toxic when greater than 5 mg/L). *Top:* Upon application initiation, the "Dose Frequency" selection box is pre-set to once daily and the output plot displays the concentration–time profile for the hypothetical drug over 10 d (10 doses) accordingly. *Bottom:* A scenario where the twice-daily dosing regimen is selected. The plot automatically updates to show the concentration–time profile when 20 doses are administered within the same 10-d period. The "missed dose" and "double dose" checkboxes also automatically update to describe specific doses in the twice-daily regimen.

elements to appear on the same line, whereby columns (column) delegate the horizontal space and the position for an element within a 12-unit wide grid. Unlike fixedPage, fluidPage (and fluidRow) allows the page layout to adjust to the dimensions of the browser that the application is running in. Below represents an excerpt of the code describing the heading layout in **Figure 3**.

```
fixedPage(fixedRow(
column(10, h2("How many times a day and at what dose do you need to take this medication so that it is effective but not toxic?", align = "center"), offset = 1)),
```

Elements are each assigned a column and a width. Adding elements below the heading requires another row and repeating the process underneath the existing code using fixedRow and column, with the same dimensions above or different ones depending on the content to be added. However, fixedPage and fluidPage layouts are not limited to fixedRow and fluidRow functions, respectively. The application exploits a sidebarLayout (equivalent level to fixedRow) to create a sidebar shown as a bordered section with a background in the user-interface. Rather than arranging elements into columns, sidebarLayout can assign them to the sidebar with sidebarPanel or to an unformatted area with mainPanel. Other layout functions include tabsetPanel and navlistPanel for dividing the user-interface into discrete sections (i.e., separate tabs or items on a navigation list for different plots or tables that would be cluttered if in one section).[9]

Widgets

Shiny widgets are interactive elements that allow users to explore different values or categories of parameters or variables. They store values chosen by the user, which are called by server.R, processed by render* functions or reactive expressions for output, and sent to the user-interface for display. Therefore, changing a widget will change the value called by server.R and the resulting output object. The Shiny package comes with a family of prebuilt widgets—paired with an R function and a logical string of arguments. They have help available by typing "?" and the widget's input function name into R (i.e., ?sliderinput). Every widget function requires a name (for input values to be called from ui.R by server.R that will not be seen by the user) and a label argument (that labels the widget in the user-interface seen by the user). Other arguments that complete the function are dependent on the widget type such as value, min, max and step for sliders (widget type for age and weight in **Figure 3**), and choices for selection boxes. A benefit that some widgets have over others is that they can ensure users are constrained to selecting only plausible values or scenarios—either restricting for biological plausibility or by the limits of what has been coded in server.R. The application has widgets for patient age, weight, dose, and dose frequency with the option to miss a dose or double it, in the sidebar. Below is the ui.R widget code for the selection box representing "Dose Frequency," within the positioning function—sidebarPanel (supplementary code available online for other widgets).

```
sidebarPanel(selectInput("FREQ", "Dose Frequency:", choices = list("Once daily" = 1, "Twice daily" = 2, "Three times daily" = 3), selected = 1)),
```

A selection box is created by the selectInput function where "FREQ" is the widget's name called into server.R as input$FREQ and "Dose Frequency:" is its user-interface label. It also requires an argument for choices—a list of text labels allocated to a numerical value. It can be customized to include selected (assign the numerical value of one of the choices to appear on application start-up), allow for multiple choices, and allocate a width of the box in pixels. As dose frequency is a categorical variable, the selection box was used to limit options a user could choose, opposed to entering any value. This was not just for clinical practicality (avoiding eight times daily regimens and restricting to integers), but for each dosing regimen an R expression was required, hence all possible values could not be coded (supplementary code available online—refer to server.R). However, these restrictions could have been achieved with a slider widget that restricted a user to only slide the bar between 1, 2, or 3 where "1" was once daily, "2" was twice daily, and "3" was three times daily, for example.

```
sliderInput("FREQ", "Dose Frequency (times per day):", min = 1, max = 3, value = 1, step = 1),
```

Given the target audience, a selection box was able to provide a concise description with its use of text than a slider (that would have required interpretation of the scale's values in its label or other text). Deciding on a widget requires consideration of the user-audience, ability to communicate its intent and the type of variables that it will be attached to, i.e., continuous versus categorical.

It is easy to get inundated with widgets that crowd the user-interface, and it is likely that not all widgets are required to be freely available to the user. Shiny offers options that can hide and show elements (including widgets) when specific conditions are met through the use of conditionalPanel in ui.R or renderUI in server.R. In the previous example, the selection box for "Dose Frequency" was pre-set to "once daily" (selected = 1) and **Figure 3** highlights that in this state, the checkboxes for missed/doubled doses read, "Dose Missed on Day 3" and "Dose Doubled on Day 4." When "twice daily" is selected, not only does the plot change to reflect twice daily dosing but the descriptions of missed/doubled doses change too (**Figure 3**). Here is an example of how this can be achieved using the conditionalPanel function within sidebarPanel in the ui.R script:

```
conditionalPanel(condition = "input.FREQ == 2",
checkboxInput("DOSE26", "Second Dose Missed on Day 2:", value = FALSE),
checkboxInput("DOSE27", "First Dose Doubled on Day 3:", value = FALSE)),
```

The condition argument is an expression evaluated repeatedly to determine if the subsequent elements should be

displayed. Hence, only when the input for FREQ (selection box widget name for "Dose Frequency") equals "2," the two checkbox widgets enclosed within conditionalPanel will appear. As seen by the label arguments for each checkboxInput function, the descriptions were designed to be reflective of the selected dose frequency. Rather than labeling the checkboxes as "Missed dose" or "Double next dose" for all regimens, the phrases describe the affected dose by day and which dose within that day of the 10-d period. This guided users to where on the time-scale a change in concentration would occur when boxes were ticked. As checkbox widgets do not take numerical values, their value argument for initial input only handles TRUE or FALSE—where FALSE is not ticked and does not affect concentration.

There are many other widget types. Those most applicable to pharmacometrics include radio buttons (radioButtons—discussed later), slider ranges (sliderInput—where two end-points on a slider can be selected and the value argument takes two values), and download buttons (downloadButton). PDF reports of output can be generated from applications using R Markdown (http://rmarkdown.rstudio.com/) and the knitr[11] package. This requires a high level of R programming experience and is beyond the scope of this tutorial.

Headings, lines, and breaks

Some Shiny functions have taken on names and functions equivalent to the HTML5 language such as those for creating headings. Heading (h) functions are coded into the layout the same way as widgets—in a positioning function such as fixedRow. Headings can be created in different sizes, where h1 produces a first level header (largest), h6—a sixth level header (smallest), and p creates a paragraph of text. Heading alignment is adjusted using the align argument:

```
h2("How many times a day and at what dose do you need to take this medication so that it is effective but not toxic?", align = "center")
```

Lines, hr(), and breaks, br(), create a division between the heading and the functional elements of the application. Both can assume any level of the layout hierarchy, i.e., can be the same level as positioning functions such as fixedRow or the same as widgets.

Displaying reactive output

Shiny *Output functions call reactive R objects defined in server.R to the user-interface, where * denotes an object description such as "plot," "table," or "text." They are built into the user-interface in the same manner as widgets by placing an *Output function within a positioning function in the ui.R script. The excerpt from ui.R below uses plotOutput to add a reactive plot to mainPanel that is updated when the user changes a widget:

```
mainPanel(plotOutput("plotCONC", height = 600, width = 800))
```

Every *Output function requires a name for the reactive object that will be called from server.R. Here, plotOutput takes the argument, "plotCONC," where plotCONC is a ggplot2[5] object in server.R (supplementary code available online). The names of output objects are not seen by users and *Output functions do not take label arguments; therefore, headings or titles need to be built into the expression for the reactive object in server.R or a heading element in ui.R. Each *Output function is unique and apart from the common name argument, they can take their own specific arguments (for example, plotOutput can also take height and width arguments to define the dimensions of the plot). The names of other *Output functions are quite self-explanatory to their task such as textOutput, imageOutput, tableOutput, htmlOutput, and uiOutput.

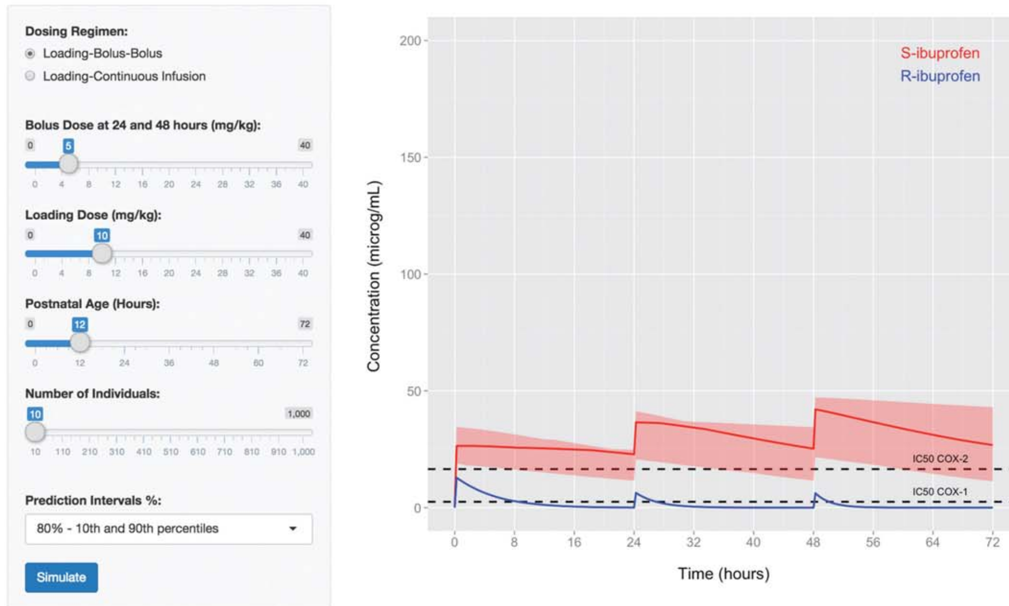
DEFINING APPLICATION INSTRUCTIONS (server.R)

The server controls the processing of user-input to display output to the user-interface. For those experienced with the R language, writing server.R consists of using the same packages, functions, and arguments to process information as previously in R with the only difference being considering whether specific tasks need to be placed in the reactive part of the script.

The server.R script (Example 2) is divided into two sections: an area for calling and processing reactive input and creating reactive output enclosed within shinyServer, and an area for nonreactive expressions and functions not dependent on widget-input (outside of shinyServer). The premise for the arrangement is based on separating expressions and functions that need to be re-evaluated upon input changes and those that never need to. All code before shinyServer (loading package libraries, defining a time sequence, ggplot2[5] themes and the function for a differential equation system) will be executed once on application initiation and will not be re-executed unless the Web-browser for the application is refreshed or the application is re-run from R. The results of these commands are not dependent on changes from input, thus their position outside of shinyServer limits the re-evaluation of nonreactive code. On the other hand, the reactive expression and the render* function within shinyServer of this example are re-executed on every change of a widget as they are dependent on widget input.

Reactive expressions

Some applications require more complex processing of multiple reactive objects and data, that when placed inside render* braces significantly slow the application's re-evaluation of the output. Furthermore, there may be instances where a reactive data frame may need to be called by multiple render* functions, which cannot be achieved when enveloped in a single one. In this application (Example 2), a data frame of time and concentrations at each time-point is constructed from reactive objects controlled by user-input widgets surrounded by reactive braces (just like a render* function). The content of the reactive data frame is used to produce a reactive plot of the concentration–time profile (renderPlot). A skeleton of the reactive expression of the server.R script for this application is provided below:



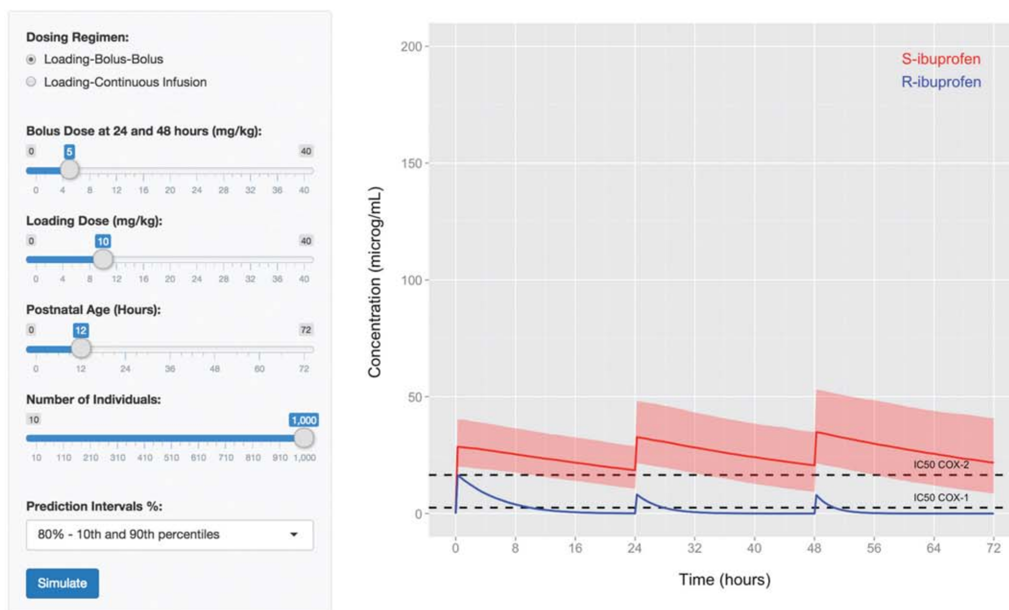


Figure 4 Ibuprofen in preterm neonates application. Shown are screenshots of the application featuring the population model by Gregoire *et al.* (2008)[14] running in a browser-window. The blue solid line represents the simulated median concentrations for R-ibuprofen over the 72-h period, and the red solid line and the boundaries of the red shaded ribbons are the median, and the 10th and 90th percentiles for simulated S-ibuprofen concentrations, respectively. Black horizontal dashed lines represent the IC50 for COX-1 (2.5 μg/ml) and COX-2 (16.5 μ/ml) enzymes. *Top:* Upon application initiation, the number of individuals simulated is small (10) for rapid loading of all Web-page elements, and the IV loading-bolus-bolus regimen doses most commonly used in practice (10-5-5 mg/kg) is selected. *Bottom:* A scenario where 1,000 concentration–time profiles of the same regimen as above are simulated.

```
shinyServer(function(input, output) {
sim.data <- reactive({
#Series of expression and functions (R package functions and user-defined) that collect widget values called by input$X to create a data frame called sim.data.df, comprised of time and the drug concentration. })
```

The reactive expression uses a series of R expressions that use widget input to return an object, or in this case a data frame—sim.data.df, that updates whenever widgets change. As discussed in Example 1, reactive objects are required to be enclosed within a render* function or a reactive expression. Therefore, in Example 2, all user-input widget values that are involved in the creation of sim.data.df are enclosed within the reactive function defining sim.data. In addition to widget values such as input$X, all expressions dependent on reactive input for defining sim.data are required to be within the reactive braces—this includes "if" statements, differential equations (as seen in Example 3), installed package functions, and standard R functions. However, reactive expressions cannot send their object to the user-interface; therefore, doing so requires a render* function, such as renderPlot.

Sending output to the user-interface

Objects appearing in the user-interface reactive to widget input or dependent on a reactive data frame (such as sim.data.df) are built into the output object by shinyServer. An element, such as plotCONC, defining a ggplot2[5] object is defined within the shinyServer braces (outside the reactive expression) as:

```
output$plotCONC <- renderPlot({ plotobj <- ggplot(sim.data()) + ... + ...
print(plotobj) })
```

Each output object requires a render* function that corresponds to the type of reactive object needed to be made. In pharmacometrics, data are often in the form of a table, which is illustrated by a plot or further analyzed to obtain a statistical summary. Therefore, renderPlot, renderText, and renderTable are commonly required to display a reactive plot, text or table, respectively. Example 2 uses renderPlot to create plotCONC using the data frame—sim.data.df, where sim.data is called using its name followed by parentheses. As sim.data is reactive, it will check if the widget input has changed, update the data, and renderPlot will redraw the plot accordingly.

Here, the ggplot2[5] package is used to create the plot object. The ggplot argument initializes a ggplot object and states the input data frame, from which defined variables can be used to create a plot. Although not shown as part of Example 2, creating a table object may require further processing of sim.data that can be achieved by an R function with sim.data as the input:

```
output$tableObject <- renderTable ({
function.name <- function(sim.data) {
#Expressions modifying sim.data into a new summary data frame }
function.name(sim.data())
}) #Brackets closing "output$tableObject"
```

This demonstrates a scenario where sim.data can be simultaneously used to create multiple, or combinations of, reactive plots and tables using a single reactive expression, which could not be achieved if all code was embraced in a single render* function. For creating a reactive text object, the structure of the previous code can be followed using renderText in place of renderTable in server.R.

SIMULATING POPULATION MODELS WITH R AND SHINY

Shiny applications can use R packages such as deSolve[12] (differential equation solver) to represent population models as differential equations. A population model in Shiny allows stochastic simulation of model predictions that embody, for example, between subject variability. Traditionally, this has been achieved by simulating, for example, concentration–time profiles for several individuals administered a dosing regimen (i.e., using NONMEM®),[13] plotting the outcome (often summarized by a median and confidence intervals) and saving the image, and repeating the process for a new dose. The plots created are static and the process of simulation can be time-consuming thus rationalization of the number and types of regimens tested is often required. Shiny applications allow for graphical output to be produced simultaneously with changing input (such as dosing regimen or covariate values); therefore, the user can rapidly view the results of candidate scenarios. Although other model visualization software such as Berkeley Madonna can also simultaneously update output, they are not readily adapted to simulate variability.

The third example Web application (**Figure 4**) demonstrates how stochastic simulation in Shiny can be used to explore a series of ibuprofen dosing regimens for patent ductus arteriosus (PDA) closure in preterm neonates less than 32 weeks gestation (Example 3; supplementary code available online). The application embodies a population pharmacokinetic model of ibuprofen enantiomers, R- and S-ibuprofen, in preterm infants[14] described as a series of differential equations. The model includes one-compartment models for each of R- and S-ibuprofen with unidirectional bioconversion of R-ibuprofen into S-ibuprofen and the effect of increasing postnatal age increasing elimination of the R-enantiomer.[14] Using the R programming language, the application simulates a patient population from random sampling of the parameter distributions of the population model to provide each individual with their own parameter set. Parameter sets and the differential equation system are provided as input to a differential equation solver to calculate the concentration–time profiles for R- and S-ibuprofen from 0 to 72 h at 15-min increments for each individual of the population. From the resultant data frame, the median and lower and upper percentiles for S-ibuprofen concentrations

(red solid line and shaded ribbon, respectively) and median R-ibuprofen concentrations (blue solid line) are calculated, plotted and displayed in the user-interface (**Figure 4**).

The model embodies important information about the pharmacokinetics of each enantiomer in neonates, from which changes in the concentration–time profile can be examined and re-examined when input doses or regimens change. This application incorporates radio buttons to represent two dosing regimens (SELECT). The “loading-bolus-bolus” regimen reflects the most common pharmacotherapeutic management for PDA closure in preterm neonates consisting of three intravenous (IV) ibuprofen doses administered every 24 h; a 10 mg/kg IV loading dose on day one (LDOSE, often within first 12 h of life), and two subsequent 5 mg/kg IV doses (BDOSE) on days two and three (i.e., 10-5-5 mg/kg). A second course of a higher dosed regimen (20-10-10 mg/kg) is also commonly administered in individuals with PDA resistant to the first course,[15] which can be simulated by adjusting the “Loading Dose” and “Bolus Dose” sliders. “Loading-continuous infusion” is a regimen which has not been commonly implemented in this practice. This Shiny application can explore the loading dose (LDOSE) and 72-h continuous infusion rate (CDOSE) required to achieve 90% of S-ibuprofen concentrations above the IC50 for cyclooxygenase-2 (COX-2; 16.5 μg/ml),[16] and how it compares with current practice. Radio buttons take similar arguments to selection boxes by commonly sharing choices and selected:

```
radioButtons(“SELECT”, “Dosing Regimen:”, choices = list(“Loading-Bolus-Bolus” = 1, “Loading-Continuous Infusion” = 2), selected = 1),
```

As there are only two options, the radio buttons display all possible options to the user without adding considerable length to the sidebar. This application also includes several other widgets. Using sliders, users can explore the effect of “delaying” therapy on the profiles for R- and S-ibuprofen by increasing the postnatal age (AGE) of the population, and the changes in the median and percentile predictions depending on the number of individuals (n) simulated. The selection box containing percentile values for the prediction intervals (CI) controls the proportion of concentrations displayed in the plot. Finally, the submit button (SUBMIT) labeled “Simulate” has two functions:

- Prevents the application from automatically updating on every widget change
- Clicking it controls when to re-simulate once widget selections have been finalized

Incorporating a large number of widgets allows a variety of scenarios to be simulated without altering the input dataset, model code or R processing code for output.

Simulating concentration–time profiles for a population
The cornerstone of Shiny applications is their ability to simultaneously update output when input changes. When simulating a large population and solving differential equations it is imperative that speed is not compromised. This includes writing efficient R code, placing nonreactive code outside of shinyServer, compiling functions that are recalled when widget input changes and running processes in parallel where applicable.

In Example 3, random number generators simulate *n* number of random effect parameters for each parameter from a normal distribution (mean of zero and standard deviation defined by the model’s variability for each population parameter). The value of one random effect parameter is matched with others generated in the same position of the *n*-value long sequence, where the population’s parameter values are log-normally distributed and calculated using the population value and the individual’s value for the random effect for the corresponding parameter. Although not demonstrated in this example, simulation with random unexplained variability or a variance-covariance matrix requires adaptation of the code provided in the example script for coding a population model in R (supplementary material available online). As the population size is dependent on widget input, n, code defining parameter values is reactive and required to be placed in a reactive expression in shinyServer. Despite being subject to re-evaluation upon each widget change, this code is computationally simple and does not impede on the application’s speed.

Defining and solving a system of differential equations
Many pharmacokinetic systems are too complicated to be expressed as an analytical solution; therefore, we yield to the computationally slower option of using differential equations. The model[14] in Example 3 can be coded in R as (supplementary code available online):

```
dA[1] = RateL + RateCB + K21*A[2] - KE1*A[1]
dA[2] = RateL + RateCB - K21*A[2] - (KE2+0.155*((T+AGE)/24))*A[2]
```

Where dA[1] and dA[2] are the differential equations describing the rate of change in the amount of S- and R-ibuprofen, respectively. RateL is a function describing the input of R- and S-ibuprofen into their respective compartments by means of a 15-min loading dose infusion, RateCB is a function describing two subsequent IV bolus doses or a 72-h continuous infusion depending on which regimen is selected, K21 is the unidirectional bioconversion of R- to S-ibuprofen, and KE1 and KE2 are the elimination rate constants for S- and R-ibuprofen, respectively.

In Example 3, lsoda of the deSolve[12] package is used to calculate the amount of R- and S-ibuprofen in each of their respective compartments every 15 min from 0 to 72 h.

The lsoda function takes several arguments:

```
sim.data <- lsoda(y = A_0, times = TIME, func = DES, parms = THETAlist)
```

Where A_0 (y) are the initial/state values of the differential system, TIME (times) specifies times for calculating A in each compartment, DES (func) is an R function of the

differential equations for the model, and THETAlist (parms) is a list of parameter values used in the function DES.

Example 2 also incorporated a differential equation system (one-compartment first-order oral absorption kinetics) where lsoda solved for the amount of drug at each of the pre-specified times. Only one parameter set was used to simulate a concentration–time profile, ultimately representing one individual as a solid red line (**Figure 3**). lsoda can only take one set of parameter values, i.e., one value for each rate constant—KA and K10 as for Example 2 or one value for each of K21, KE1, and KE2 as in Example 3. However, Example 3 is able to explore the effects of different covariate values or dosing regimens on a population, and can determine median concentrations and prediction intervals. Each individual's parameter set of this example is arranged as a single row in the parameter data frame, which is provided as input to the R function, simulate.conc (containing lsoda, and expressions for initial conditions and input parameters). Using ddply from the plyr[17] package, simulate.conc is passed to each row of the parameter data frame to calculate the amount of R- and S-ibuprofen at each time given each individual's parameter set.

The application in Example 3 instantly updates after clicking the "Simulate" button when the slider value for n is 10 individuals. However, predictions for the median, and upper and lower percentiles are poor as shown by jagged lines (**Figure 4**). Increasing the population size to 1,000 increases update time (time dependent on computer processing power) as the application is processing more information, but achieves an improved prediction of the time-course.

In this example, a slider determines population size. Every time the widget changes, the reactive expression within shinyServer recognizes that its stored values are out-dated and they are re-evaluated to accommodate the new values. When a user is highly active, i.e., rapidly changing widget values, the application can be updating many times a second. In this example, moving the "Number of Individuals" slider rapidly from 10 to 1,000 causes the application to update and show predictions for 1,000 individuals. However, if a user accidentally moves the slider to 900, pauses, and moves the slider to 1,000, the application will update first to reflect 900 individuals and will update again to reflect the intended 1,000—approximately doubling the time. This is also an issue when 1,000 individuals are required but the user is fine-tuning the desired dose. Here, the "Simulate" button dictates when to update the application allowing users to modify their selections before running a simulation.

Increasing application speed using compiled functions

The main advantage of using Shiny is that it allows for automatic updating when input changes—but when the R code and function is computationally intense such as for stochastic simulations it may need a minute or more to update the plot, thus negating the reactive benefits of Shiny. Speed increases may be possible by using the compiler[6] package for R. R is an interpreted language which means code is saved as text files which is reduced to machine instructions at runtime. In contrast, compiled languages save the code directly to an executable file of machine instructions often with speed benefits. The compiler package and the cmpfun function provide some of the benefits of compiled code in R by implementing a byte code compiler. There is a time overhead in initially compiling a function, but thereafter the compiled function is usually faster to run. In Example 3, the functions defining the model's differential equations (DES), and for solving the system (simulate.conc) are called *n* times (number of simulated individuals) on every widget change as they are both enclosed in the reactive expression for sim.data. Each of the functions can be compiled using the cmpfun function of the compiler package:

```
DES.cmpf <- cmpfun(DES)
simulate.conc.cmpf <- cmpfun(simulate.conc)
```

Increasing application speed using parallel processing

On computers with multiple processors, multiple cores, or both, it is possible to increase application speed using parallel processing. The doParallel[18] package and its dependencies, foreach[19] (for creating a loop structure) and parallel,[6] can assign multiple cores to a function on both Windows and Unix-like platforms (such as Mac OS X). Computationally intense jobs, such as calculating concentration–time profiles from differential equations for 1,000 individuals in Example 3, can be executed faster when split and assigned to multiple cores of the computer as each fragment is run parallel in time. If four cores are used, a population of 1,000 is divided into four groups of 250 individuals and executed simultaneously. Like compiling a function, time is required initially to set up the cluster of cores, and the more cores that are assigned, the longer the initial set-up time.

Setting up a cluster of cores for parallel processing is written in the server.R script outside shinyServer. Below is an excerpt of code from Example 3 for setting up a cluster of cores, which is operational on both Windows and Mac OS X platforms (mainly written to accommodate Windows).

```
cl <- makePSOCKcluster(detectCores() - 1)
clusterEvalQ(cl, list(library(deSolve), library(foreach)))
registerDoParallel(cl)
```

When the doParallel package is installed, R can detect the number of cores the computer has by using the detectCores() function. Here, makePSOCKcluster creates multiple copies (in this example, one less than the number of computer cores) of R that communicate over sockets. clusterEvalQ sends the required packages to each R copy for the process that is to be run in parallel, and registerDoParallel is a parallel backend compatible for both Windows and Mac OS X for running the function in parallel using the core cluster defined earlier.

Some R package functions have been specifically designed for parallel processing when a cluster of cores has been appropriately set-up. In Example 3, the ddply

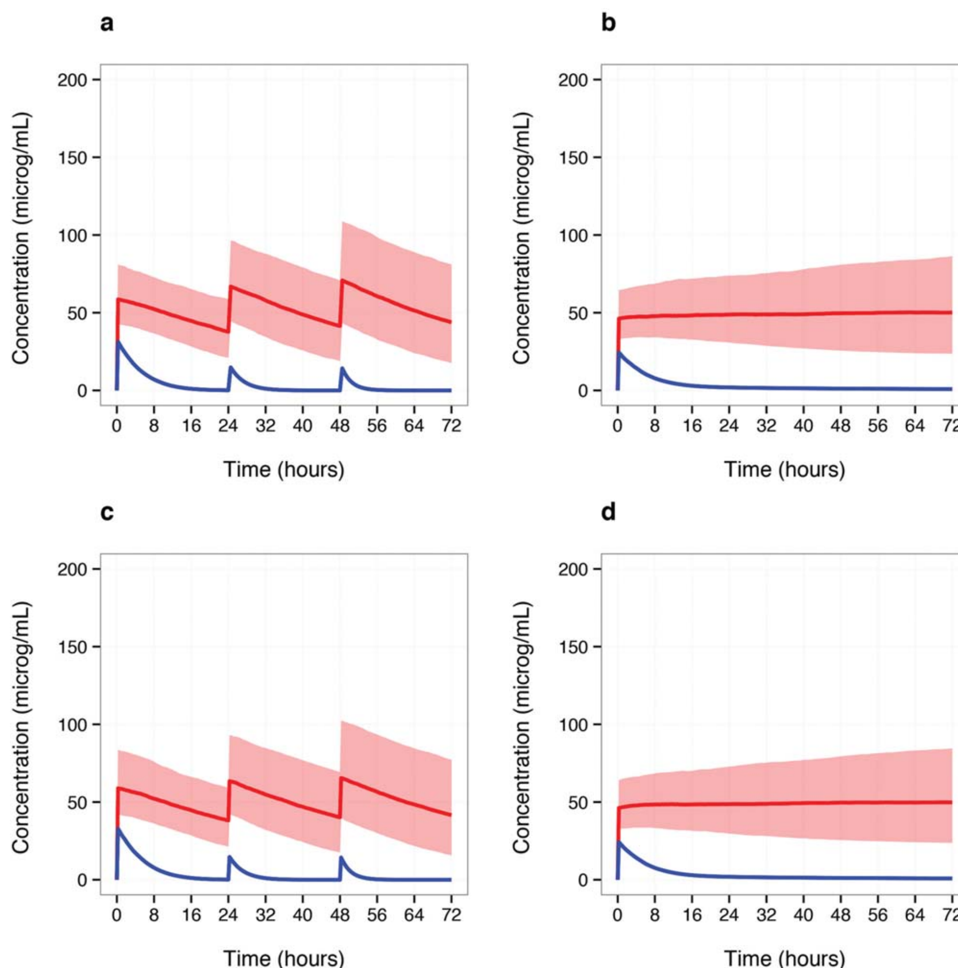


Figure 5 Comparison of Shiny and NONMEM® simulation outputs. Shown are the median concentrations at 15-min intervals for R- (blue solid line) and S-ibuprofen (red solid line), and the red shaded region represents the 10th and 90th percentiles for S-ibuprofen simulated concentrations, over a 72-h period. For each dosing regimen, a 1 kg patient with a post-natal age of 12 h was used to create a simulated population. **(a,c)** Concentration–time profiles based on the simulation of 1,000 individuals administered the IV loading-bolus-bolus dosing regimen (20-10-10 mg/kg). **(b,d)** Concentration–time profiles based on the simulation of 1,000 individuals administered the IV loading-continuous infusion regimen (16 mg/kg loading dose, 24 mg/kg continuously infusion over 72 h). **(a,b)** Shiny Application Example 3 output; code embedded in Shiny's server.R script was converted to a standard R-script to produce plots. **(c,d)** NONMEM® output; simulation output was processed and plotted using R.

function takes .parallel as an argument—where when .parallel equals TRUE the simulate.conc.cmpf function is applied in parallel. Therefore, parameter sets for multiple individuals can be processed simultaneously thus increasing application speed.

Comparing the Shiny application and NONMEM® simulations

Performing an independent check of the differential equation system and simulated population characteristics of a Shiny application is important. For applications such as Example 3, R simulation output can be evaluated with the simulation output provided by NONMEM®[13] (or similar) when given an equivalent population (i.e., input dataset) and dosing regimen.

Figure 5 compares the simulated output for a loading-bolus-bolus regimen (20-10-10 mg/kg) and a loading-continuous infusion regimen (16–24 mg/kg) from Example 3 with the output provided if the same population model was coded in NONMEM® and an equivalent population of 1,000 individuals (i.e., post-natal age of 12 h and 1 kg weight) administered the same dosing regimens and their resultant concentration–time profiles were simulated at 15-min intervals. The series of plots indicate that the application produces median concentrations for R- and S-ibuprofen (blue and red solid lines, respectively), and 80% empirical confidence intervals for S-ibuprofen (red shaded ribbon) comparable to those simulated by NONMEM®.

Simulating the population (1,000 individuals) *and* updating the output plot using a Shiny application was considerably faster than that achieved by the combined NONMEM® simulation and R processing times (NONMEM® + R method). **Table 2** shows the mean computation times to simulate and process output for the two dosing regimens

Table 2 Shiny versus NONMEM® + R processing: comparison[a] of run times

Method[b]	IV loading-bolus-bolus (20–10-10 mg/kg) (seconds)	IV loading-continuous infusion (16–24 mg/kg) (seconds)
NONMEM® + R Processing[c]	≈ 385	≈ 366
Shiny: one core	≈ 103	≈ 60
Shiny: four cores, compiled functions	≈ 35	≈ 27

[a]Based on application Example 3 with n = 1,000 and evaluations at 15-min intervals over 72 h.
[b]Simulations were performed using a Dell® Power Edge R910 server with 4 x 10 core Xeon 2.26 GHz processors and 256 GB of RAM running Windows 8 Server SP1 64-bit.
[c]NONMEM® Version VII Level 3.0[14] using the Wings for NONMEM (Version 734) interface (http://wfn.sourceforge.net/), with Intel® Visual Fortran Composer XE 2013 Update 1, and ADVAN8 subroutine.

shown in **Figure 5** by three different methods (i.e., NONMEM® + R, uni-core Shiny, multi-core Shiny with compiled functions) when run four times each. Note that using NONMEM® also required considerably more user time. Whereas running Example 3's application with one core and no compiled functions largely improved the run time when compared with NONMEM® + R (≈ four- to sixfold reduction), an over 60-s wait for plot results was required for both dosing regimens. Compiling DES and simulate.conc functions and the addition of parallel processing for concentration–time profiles further improved the Shiny run times (≈ two- to threefold reduction).

Whereas only two dosing scenarios have been demonstrated using Example 3, the user-interface of this Shiny application provided an easy-to-use platform to explore other scenarios by changing widget values for both the pharmacometricians who developed the application and the nonpharmacometricians who raised the research question.

SHARING SHINY APPLICATIONS

Shiny applications can easily be shared by passing on the ui.R and server.R scripts to a colleague with an installation of R or RStudio and the installed Shiny package. However, sharing is not limited to this mode. Shiny applications can also be hosted on Web servers making them accessible to those unfamiliar with, or do not have an installation of, R. There are three ways that RStudio offers to host a Shiny Web application: ShinyApps.io, Shiny Server, and Shiny Server Pro. Users will find this a slower option than running Shiny on their local machine—predominantly dependent on the speed and latency of their Internet connection. When a Shiny application is updating, the output is grayed out indicating to the user that processing is taking place.

ShinyApps.io

ShinyApps.io (https://www.shinyapps.io/) allows you to upload applications from an R session to a server hosted by RStudio for free but does not guarantee privacy. It links with your existing Google or GitHub account (also free if you do not have one) to have administrative control over the application. Depending on your operating system, other packages and programs may need to be installed before deploying an application. At this date, instructions on how to upload an application are available on the Shiny by RStudio website—http://shiny.rstudio.com/articles/shinyapps.html. The three examples in this tutorial are hosted using ShinyApps.io.

Shiny Server and Shiny Server Pro

Installing the Shiny package in R allows output Web-pages to be served to a port (connection) on the localhost (the user's own computer). These are typically (depending on the configuration) accessed by a URL such as http://127.0.0.1:4748/ or http://localhost:4748/ where 4748 is the particular (configuration dependent) port number that will serve the Web-page.

It is also possible to host your own Shiny server to make Shiny pages available over the Internet. RStudio supplies Shiny Server software—this is not an R package, but separate software which can be installed only on servers running specific Linux distributions. In more sophisticated configurations, the Shiny server might run alongside an Apache Web server and a database such as MySQL with data exchanged between R, the database, Shiny and other Web deployment software. Such configurations do require some Web/programming expertise, and are probably only of interest to those wishing to make a Shiny application available to a wide audience. Shiny Server is open source, and Shiny Server Pro is a commercial version with enhanced security (possibly allowing confidential Web sharing of proprietary material) and additional features.

Shiny resources

Shiny has several resources available, predominantly online, to assist new and advanced users in developing and deploying applications.

- Shiny by RStudio;[9] the developers of Shiny provide a tutorial series for developing and deploying applications built with exercises and discussion boards—http://shiny.rstudio.com/
- Stack Overflow; search for the "shiny" tag and read answers posted by other users or post your own question—http://stackoverflow.com/questions/tagged/shiny
- GitHub—https://github.com/rstudio/shiny
- Shiny Google mailing list; a discussion forum that allows you to receive daily summaries by means of email of questions asked and answered by others of the group—https://groups.google.com/d/forum/shiny-discuss
- Commercially available book, *Web Application Development with R Using Shiny* by Chris Beeley[20]

CONCLUSIONS

The Shiny package of R is part of a larger movement amongst data scientists (e.g., Google charts, https://developers.google.com/chart) exploring interactive data visualization by means of Web-browsers. Like data science, the problem of communicating and sharing the information embodied in pharmacometric models is challenging. Representing pharmacometric models in R requires a degree of competency, but it is a skill that is increasingly common in the pharmacometric work force. We note that models coded in other software such as NONMEM® are relatively easily converted to R, and believe that the

Shiny package of R is an exciting development that allows pharmacometric models coded in R to be made accessible to a wider audience (e.g., drug development teams, clinicians). Elegant and flexible interfaces to models can be produced relatively quickly once the basics of Shiny are mastered. We also believe that pharmacometricians themselves may find Shiny applications a quick and informative way of simulating from their models. We hope that the information and examples provided here is sufficient to allow interested readers to start creating their own pharmacometric Shiny Web applications.

Acknowledgments. The authors acknowledge that the Australian Centre for Pharmacometrics is an initiative of the Australian Government as part of the National Collaborative Research Infrastructure Strategy.

Author Contributions. J.W. wrote the manuscript, designed the research, performed the research, and contributed analytical tools. A.M.H. contributed analytical tools and revised the manuscript. R.N.U. wrote the manuscript, designed the research, performed the research, and contributed analytical tools.

Conflict of Interest. The authors declared no conflict of interest.

1. Zhang, L., Pfister, M. & Meibohm, B. Concepts and challenges in quantitative pharmacology and model-based drug development. *The AAPS J.* **10**, 552–559 (2008).
2. Mould, D.R. & Upton, R.N. Basic concepts in population modeling, simulation, and model-based drug development. *CPT Pharmacometrics Syst. Pharmacol.* **1**, e6 (2012).
3. Bonate, P. What happened to the modeling and simulation revolution? *Clin. Pharmacol. Ther.* **96**, 416–417 (2014).
4. Ito, K. & Murphy, D. Application of ggplot2 to pharmacometric graphics. *CPT Pharmacometrics Syst. Pharmacol.* **2**, e79 (2013).
5. Wickham, H. *ggplot2: Elegant Graphics for Data Analysis*. <http://had.co.nz/ggplot2/book>. (Springer, New York, 2009).
6. R Core Team. *R: A language and Environment for Statistical Computing*. <http://www.R-project.org/>. (R Foundation for Statistical Computing, Vienna, Austria, 2014).
7. Krause, A. & Lowe, P.J. Visualization and communication of pharmacometric models with Berkeley Madonna. *CPT Pharmacometrics Syst. Pharmacol.* **3**, e116 (2014).
8. RStudio Inc. *shiny: Web Application Framework for R*. R package version 0.10.1. <http://CRAN.R-project.org/package=shiny> (2014).
9. <http://shiny.rstudio.com/> Accessed 18 September 2014.
10. Boxenbaum, H. Pharmacokinetics tricks and traps: flip-flop models. *J. Pharm. Pharm. Sci.* **1**, 90–91 (1998).
11. Xie, Y. *knitr: A General-Purpose Package for Dynamic Report Generation in R.* R package version 1.9. (2015).
12. Soetaert, K., Petzoldt, T. & Setzer, R.W. Solving differential equations in R: Package deSolve. *J. Stat. Softw.* **33**, 1–25 (2010).
13. Beal, S., Sheiner, L.B., Boeckmann, A. & Bauer, R.J. *NONMEM User's Guides*. (1989-2009). (ed. Sheiner, L.B.) (Icon Development Solutions, Ellicott City, MD, 2009).
14. Gregoire, N., Desfrere, L., Roze, J.C., Kibleur, Y. & Koehne, P. Population pharmacokinetic analysis of Ibuprofen enantiomers in preterm newborn infants. *J. Clin. Pharacol.* **48**, 1460–1468 (2008).
15. Dani, C. *et al.* High-dose ibuprofen for patent ductus arteriosus in extremely preterm infants: a randomized controlled study. *Clin. Pharmacol. Ther.* **91**, 590–596 (2012).
16. Kato, M., Nishida, S., Kitasato, H., Sakata, N. & Kawai, S. Cyclooxygenase-1 and cyclooxygenase-2 selectivity of non-steroidal anti-inflammatory drugs: investigation using human peripheral monocytes. *J. Pharm. Pharmacol.* **53**, 1679–1685 (2001).
17. Wickham, H. The split-apply-combine strategy for data analysis. *J. Stat. Softw.* **40**, 1–29 (2011).
18. Revolution Analytics and Steve Weston. *doParallel: Foreach parallel adaptor for the parallel package*. R package version 1.0.8. <http://CRAN.R-project.org/package=doParallel> (2014).
19. Revolution Analytics and Steve Weston. *foreach: Foreach looping construct for R.* R package version 1.4.2. <http://CRAN.R-project.org/package=foreach> (2014).
20. Beely, C. *Web Application Development with R Using Shiny* 1st edn. (PACKT Publishing, Birmingham, United Kingdom, 2013).

© 2015 The Authors CPT: Pharmacometrics & Systems Pharmacology published by Wiley Periodicals, Inc. on behalf of American Society for Clinical Pharmacology and Therapeutics. This is an open access article under the terms of the Creative Commons Attribution-NonCommercial License, which permits use, distribution and reproduction in any medium, provided the original work is properly cited and is not used for commercial purposes.

Supplementary information accompanies this paper on the *CPT: Pharmacometrics & Systems Pharmacology* website (http://www.wileyonlinelibrary.com/psp4)