The runcode package – troubleshooting

Haim Bar and HaiYing Wang

April 16, 2021

When using the runcode package, you have to be aware of some usage rules, so this document attempts to anticipate all the possible user or system errors, and show how to interpret the output and fix the problems.

Before you start using runcode

Before using the package make sure that you have Python (version 3.x) installed and in your path. Also, you must install the required LaTeX packages: morewrites, tcolorbox, xcolor, inputenc, textgreek, filecontents, xifthen, xparse, xstring, and fvextra. The minted package is optional but recommended.

If you try to compile this file from its tex source, you will get errors (because the purpose of this document is to demonstrate errors and how to fix them.) It is assumed that you are using a cloned runcode repository, so files used in this document are ones in the directory structure as it is stored on github.

The files generated by runcode live in the project's directory, so we assume the user has the necessary permissions to create, modify, delete files and subfolder.

For the server mode the folder must contain a configuration file for each statistical language used (R, julia, matlab, Python). If such files don't exist, runcode will create them with default values. Remember to check if the defaults work for you. For example, you may need to change the port number, or increase the timeout parameter.

The preamble

Several options are available via the `\usepackage[options]{runcode}` statement in the preamble. See the package's documentation for details. Here, we focus on troubleshooting only.

First and foremost, if it appears that the runcode functions are not executed, be sure to turn on the `shell-escape` option. The main log file will contain a relevant warning if this option is not enabled.

Equally important, the command-line tools you intend to call from runcode must be installed, and in your path (e.g., R, Julia, Matlab, Python).

The runcode package can call any command-line function when it is used in 'batch-mode'. That is, when the command-line tool is called separately each time a computation is performed from within the tex document (upon compilation). When using R, Julia, or Matlab, the user can maintain a continuous session to the corresponding command-line tool. This saves initialization time, and allows to keep a session's history, for performing steps sequentially. This is the recommended way to use runcode with R, Julia, and Matlab. At the present time, no other languages are supported for 'server-mode' operation.

When code is included in the manuscript, it is done via the `\showCode` command (see below). By default, the code-highlighting it done via the minted package. If code highlighting doesn't work, check the the minted package is installed properly. Python (3.x) also has to be installed. If you have trouble with the installation of minted, use the `nominted` option. This will cause runcode to use the fvextra for code display, instead.

If you are using the server-mode, be aware that some editors terminate all child processes at the end of LaTeXcompilation. For example, Emacs with Auctex behaves this way. For this case, use the nohup option, and the server will not be terminated by the parent process.

If you want the server to be stopped after each compilation, use the stopserver option. While you are compiling the tex document often, you may want to keep the server side running in order to save time during initialization, and to maintain the variables and results already in memory. However, it is a good idea to use the stopserver option when you are done, just to prevent any confusion when

using runcode for multiple projects. Such confusion may only occur if the configuration file for two files in two different projects (say, A and B) use the same port number. Suppose your new project is B, and the runcode server of project A is still running and using the same port. In this case you may, for example,

- Accidentally overwrite variables in project A, or use the wrong ones in project B if they were also defined in A;
- Try to run code which is stored in project B, but you get an error message in the pdf file such as '`cannot open file 'mycode.R': No such file or directory`'. The reason for this error is that the other instance of the runcode server uses project A's base directory, and the file mycode.R is not in the right path.

It's best to check that port numbers are unique, and when a project is not expected to be compiled for a while to enable the stopserver option.

The showCode command

`\showCode` prints the source code, using minted for a pretty layout. It takes 4 arguments. Arg #1 is the programming language, Arg #2 is the source file name, Args #3 and #4 are the first and last line to show (optional).

If the source file name does not exist, you will get a red and bold error message (remember that the compiler is case-sensitive, so test.R is not the same as Test.R).

```
\showCode{R}{Sim23.R}
```

showCode: File Sim23.R does not exist!

In contrast, when the file exists, as in this example

```
\showCode{R}{paper/supplement/Code/code1.R}
```

the file will be shown correctly:

```r
set.seed(0) ## fix the random number
x = rnorm(100)
y = 1+x+rnorm(100)
fit = lm(y~x)
print(summary(fit))
```

If the programming language is misspecified or not recognized by minted or fvextra, the code highlighting may not be shown correcly.

```
\showCode{matlab}{aper/supplement/Code/code1.R}
```

```
set.seed(0) ## fix the random number
x = rnorm(100)
y = 1+x+rnorm(100)
fit = lm(y~x)
print(summary(fit))
```

If the line number in Arg #3 exceeds the actual number of lines in the code, the code box will be empty, and the LaTeX compiler will show an error message in its log file ('Empty verbatim environment').

```
\showCode{R}{paper/supplement/Code/code1.R}[6][8]
```

```

```

If the number in Arg. #4 is greater than the number of lines in the file, minted will show the code up to the last line (so this misspecification is harmless).

```
\showCode{R}{paper/supplement/Code/code1.R}[4][8]
```

```r
fit = lm(y~x)
print(summary(fit))
```

The runExtCode command

`\runExtCode` runs an external code. It takes 4 arguments. Arg #1 is the executable program, Arg #2 is the source file name, Arg #3 is the output file name (optional - if not given, the counter codeOutput is used). Arg #4 controls when to run the code (optional - if not given, it listens to `\runcode`; run = force the code to run; cache or anything else = use cache).

If the source file name does not exist, you will get a red and bold error message:

```
\runExtCode{julia}{test2.jl}{test2}
```

runExtCode: File test2.jl does not exist!

The first argument can be any command-line executable, including user-defined program names (compiled code, aliases, etc.). Because of that, we do not perform validity check before trying to execute it. For example, the following will not produce any error message:

```
\runExtCode{C}{test.R}{}
```

but it will be possible to see an empty file in the tmp folder. This will be obvious to the user once the `\includeOutput` command is executed (see below).

Note that `\runExtCode` requires the specific program and command line arguments. For example, using R as the language will not work:

```
\runExtCode{R}{paper/supplement/Code/code1.R}{testWithR}
```

It will create the file tmp/testWithR.tex, and in it you will see

```
ARGUMENT 'paper/supplement/Code/code1.R' __ignored__
```

The correct way to use `\runExtCode` with R is

```
\runExtCode{Rscript --save --restore}{paper/supplement/Code/code1.R}{test}
```

See the package's documentation for more working examples.

The runR, runJulia, runMatlab commands

When using the *server-mode* of runcode, we can use the shortcuts to R, Julia, or Matlab instead of `\runExtCode`. The usage is similar, but the language name is inferred from the command. For example, we can have:

```
\runR{paper/supplement/Code/code1.R}{testWithrunR}
```

If the source code file doesn't exist, we get an error message as with `\runExtCode`

```
\runJulia{test2.jl}{test2}
```

runExtCode: File test2.jl does not exist!

The includeOutput command

`\includeOutput` is used to embed the output from executed code. It takes 2 arguments: Arg #1 is the output file name (optional - if not given, the counter codeOutput is used). Arg #2 is the optional type output with default "vbox" (vbox = verbatim in a box, tex = pure latex, or inline).

We first have to run the code

```
\runR{paper/supplement/Code/code1.R}{testWithrunR}
```

To see the output, we use

```
\includeOutput{testWithrunR}[vbox]
```

```
Call:
lm(formula = y ~ x)

Residuals:
    Min      1Q  Median      3Q     Max
-1.5900 -0.8153 -0.1531  0.6379  2.8379

Coefficients:
```

```
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  0.95130    0.09629    9.88   <2e-16 ***
x            1.13879    0.10960   10.39   <2e-16 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 0.9626 on 98 degrees of freedom
Multiple R-squared:  0.5242,        Adjusted R-squared:  0.5193
F-statistic:   108 on 1 and 98 DF,  p-value: < 2.2e-16
```

Although the physical file is named tmp/testWithrunR.tex, we drop the tmp/ prefix and the .tex suffix. If we do include them, the file will not be found and we'll get an error message:
`\includeOutput{testWithrunR.tex}[vbox]`

Output file tmp/testWithrunR.tex.tex not found. Check the file name (it may be that the file name was given with the suffix .tex. If so, remove it). If the file name is correct the problem may be because code execution is disabled and no cache is available. If so, force the code to run again (using the [run] option).

If anything other than vbox, tex, or inline is provided in the brackets, the output will be embedded in the document in plain text. While no error or warning is raised, this may cause problems in the compilation of the tex file and therefore should be avoided. (For example, if the output contains underscores then the compiler will report an error because it would appear that math symbols are used in text mode).

The inln, inlnR, inlnJulia, and inlnMatlab commands

In the *server-mode* only, `\inln` is used to execute short source code and embed resulting output within the text. It takes 3 arguments. Arg #1 is the executable program; Arg #2 is the source code Arg #3 is the type output (if skipped or with empty value the default type is inline; vbox = verbatim in a box).

The syntax of the code has to be correct. If it's not, the error from the statistical software will be embedded in the text. For example, if we use Factorial, instead of the real function (factorial) we get an error message.
`The factorial of 6 is \inlnR{```cat(Factorial(6))```}.`
would produce the following:
The factorial of 6 is Error in Factorial(6) : could not find function "Factorial".

Such errors are not due to LaTeX syntax or compilation, so runcode doesn't highlight them. Automatically identifying and highlighting such errors would require case-by-case analysis of the output of specific command-line tools. For now, it's up to the user to check the code before it is embedded in the tex document, and to check the output after the compilation.

In some cases it is perfectly fine if embedded code does not produce output, but when using `\inln` this is not the case, so we check if the command used within `\inln` produced a zero-byte output file. If it did, we show an appropriate message, like in the following example. Note that the reason no output is produced is that file.csv does not exist.
`The number of columns is \inlnR{```dat <- read.csv("file.csv"); cat(ncol(dat))```}.`
This will result in:
The number of columns is ZERO BYTES IN OUTPUT.