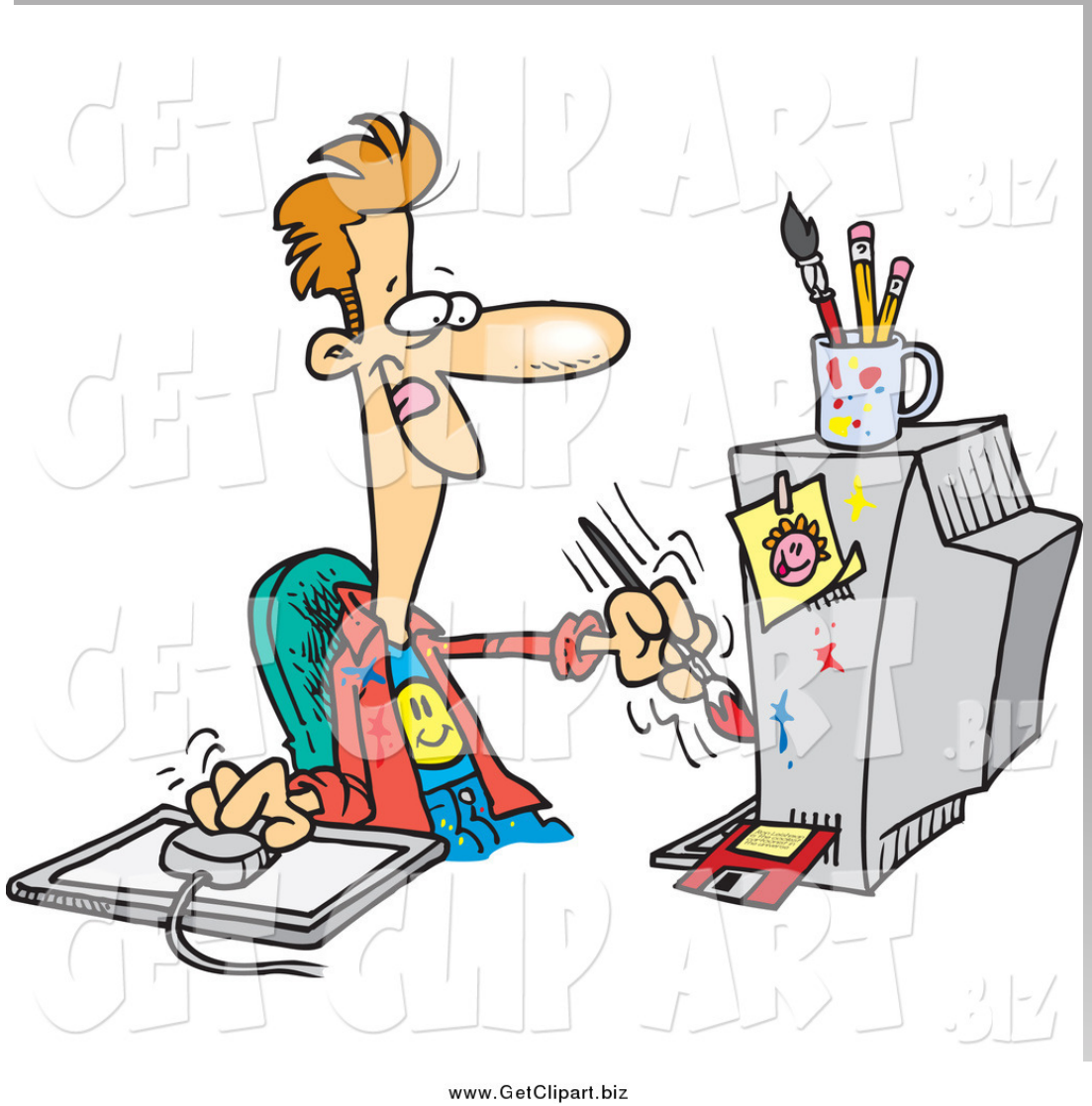
Best Practices for Computing Simulations using R and SAS®:

Wednesday, May 13th, 2015 12^{30} - 14^{30}

Federico Campigotto
Department of Biostatistics and Computational Biology
Dana-Farber Cancer Institute

Outline:

- Motivation for this training session(s)
- List of 24 best practices organized by 8 groups
- A few examples and applications using SAS and R codes
- Open discussion



Motivation

Coding and programming is an art.

- Can one teach it?

Motivational paper:

OPEN ACCESS Freely available online

PLOS | B

Community Page

Best Practices for Scientific Computing

Greg Wilson[1]*, D. A. Aruliah[2], C. Titus Brown[3], Neil P. Chue Hong[4], Matt Davis[5], Richard T. Gu
Steven H. D. Haddock[7], Kathryn D. Huff[8], Ian M. Mitchell[9], Mark D. Plumbley[10], Ben Waugh[11]
Ethan P. White[12], Paul Wilson[13]

1 Mozilla Foundation, Toronto, Ontario, Canada, 2 University of Ontario Institute of Technology, Oshawa, Ontario, Canada, 3 Michigan State Universi Michigan, United States of America, 4 Software Sustainability Institute, Edinburgh, United Kingdom, 5 Space Telescope Science Institute, Baltimore, States of America, 6 University of Toronto, Toronto, Ontario, Canada, 7 Monterey Bay Aquarium Research Institute, Moss Landing, California, United S 8 University of California Berkeley, Berkeley, California, United States of America, 9 University of British Columbia, Vancouver, British Columbia, Canad University of London, London, United Kingdom, 11 University College London, London, United Kingdom, 12 Utah State University, Logan, Utah, United 13 University of Wisconsin, Madison, Wisconsin, United States of America

Introduction

Scientists spend an increasing amount of time building and using software. However, most scientists are never taught how to do this efficiently. As a result, many are unaware of tools and practices that would allow them to write more reliable and maintainable code with less effort. We describe a set of best

error from another group's code was not disc
publication [6]. As with bench experiments, not
done to the most exacting standards; however, s
aware of best practices both to improve their o
for reviewing computational work by others.

This paper describes a set of practices that a
have proven effective in many research setting

ox 1. Summary of Best Practices

. Write programs for people, not computers.

) A program should not require its readers to hold mor than a handful of facts in memory at once.
) Make names consistent, distinctive, and meaningful.
) Make code style and formatting consistent.

. Let the computer do the work.

) Make the computer repeat tasks.
) Save recent commands in a file for re-use.
) Use a build tool to automate workflows.

. Make incremental changes.

) Work in small steps with frequent feedback and cours correction.
) Use a version control system.
) Put everything that has been created manually in versio control.

. Don't repeat yourself (or others).

) Every piece of data must have a single authoritativ representation in the system.
) Modularize code rather than copying and pasting.
) Re-use code instead of rewriting it.

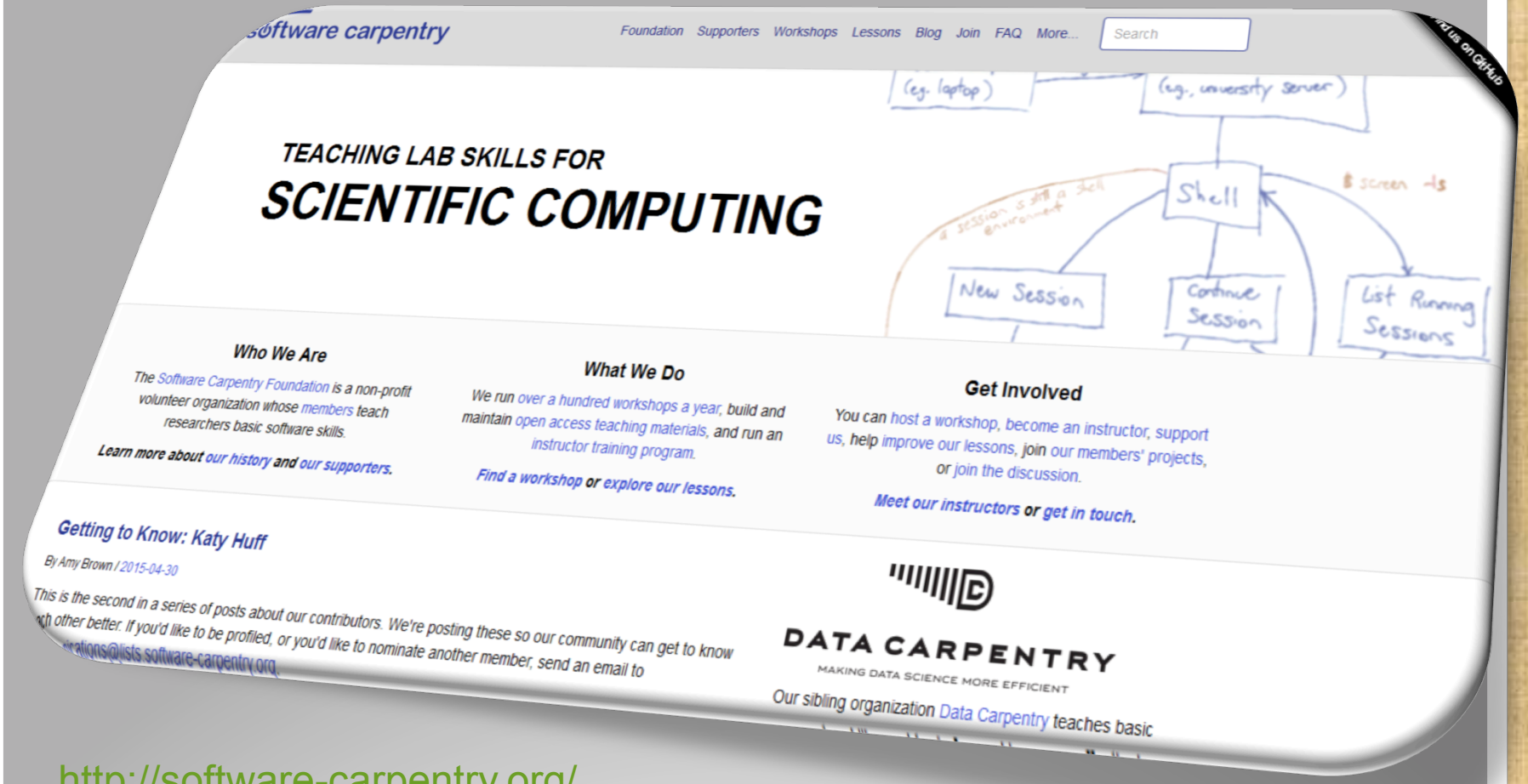
. Plan for mistakes.

) Add assertions to programs to check their operation.
) Use an off-the-shelf unit testing library.
) Turn bugs into test cases.
) Use a symbolic debugger.

. Optimize software only after it works correctly.

) Use a profiler to identify bottlenecks.
) Write code in the highest-level language possible.

Webpage linked to the paper



- http://software-carpentry.org/

Training series:

- **Best Practices for Computing Simulations using R and SAS®**
  by Federico Campigotto

- Introduction to **SAS** Macros
  by Hui Huang Monday, May 18th, 10^{00}-12^{00}

- Better Practices for Statistical Computing in **R**
  by Robert Redd Wednesday, May 20th, 1^{30}-3^{00}

- Efficient computation in **Python**: Big Data processing and Parallel Computing
  by Luca Pinello *date to be announced*

Motivation

Computing skills are the prerequisites for reproducible computational research

Unreliable software or codes can lead to serious errors impacting the central conclusions of published research (*see Zeeya Merali, "Why Scientific Computing Does Not Compute". Nature, 467, 775; October 14, 2010*)

Recent high-profile retractions, technical comments, corrections for errors in computational methods includes papers in Science, PNAS and other high-profile journals

In addition, because software are often used for more than a single project and it is often reused by and shared with other scientists, computing errors can have disproportionate cascading impacts on the scientific process.

Effectiveness or efficiency or both?

Effectiveness: it relates the *objectives* to the *outcome*.

Efficiency: it relates the *input* (in terms of resources used, i.e. time, space, costs, efforts,…) to the *output*.

So, a code or a program written in whatever language could be:

a) Neither efficient nor effective
b) Efficient, but not effective
c) Effective, but not efficient
d) both efficient and effective

Best practices…

- …for an efficient and effective code

A. Write Programs for People, not Computers

- Hard to tell if code that is difficult to understand is doing what it is supposed to do (i.e. is it effective?)
- Hard for other scientists to re-use it...
- ...including your future self

1.a: So break programs into **short, readable functions**, each taking only a few parameters

2.a: Make names **consistent, distinctive, and meaningful**.

3.a: Make code style and formatting **consistent**.

B. Let the Computer Do the Work

- 4.b: Make the computer repeat computational tasks
- 5.b: Save recent commands in a file for re-use
- 6.b: Use a management tool to automate workflows

Write **short** programs, scripts, macros, functions for everything

Turn history into scripts: An accurate record of how a result was produced saves time and reduces errors and supports reproducibility in the large

To avoid errors and inefficiencies from repeating commands manually, a **single command** can regenerate everything that needs to be regenerated

C. Make Incremental Changes

- 7.c: Work in small steps with frequent feedback and course correction
- 8.c: Use a version control system (VCS)
- 9.c: Put everything that has been created manually in version control

How to keep track of **changes**? How to **collaborate** with peers on a program or a data set?

Solution: store snapshots of projects' files in a **repository** (VCS), to track changes; to allow changes to be undone; to support independent parallel development. This is essential for **collaboration**

Save in VCS not just software, but also papers, raw images, ...
Leave out things automatically generated by the computer
Use build tools to reproduce those instead

C. Make Incremental Changes

Save in VCS not just software, but also papers, raw images, ...

In SAS for instance, it is a good practice to save:

a) Program files
b) Logs
c) Listings of outputs

D. Do not Repeat Yourself (or Others)

- 10.d: Every piece of data must have a single authoritative representation in the system
- Avoid "code clones": they are difficult to maintain
- 11.d: Modularize code rather than copying and pasting
- 12.d: Re-use code instead of rewriting it

Define constants and variables exactly **once**, raw data files should have a **single** version

Reducing code cloning **reduces error rates** and increases comprehension

better to find an **established library or package** than to attempt to write our own routines for well established problems

E. Plan for mistakes

- Mistakes are inevitable, so it is necessary to verify and maintain the validity of the code over time
- Tests check to see whether the code matches the person's expectations of its behavior, whether the code works properly and the results are valid

13.e: Add **assertions** to programs to ensure inputs are valid, outputs are consistent, …

- they ensure that if something goes wrong, the program stops right away (i.e. it simplifies debugging)
- they explain the program as well as check its implementations

14.e: Add **automated tests**, to make sure single pieces of code work correctly, pieces of code work well when combined, and that the behaviour of a program does not change when the details are modified

15.e: Turn bugs into **test cases** by writing tests that trigger a bug that has been found in the code and (once fixed) will prevent the bug from reappearing unnoticed.

16.e: Use an **interactive program debugger** to explore the program as it runs and use breakpoints to stop program at particular points or when particular things are true

F. Optimize Software Only After It Works Correctly

- So get it right first (code effectiveness), then make it fast (code efficiency)
- Write code in the highest-level language possible

"The first rule of program optimization: **Do not do it**. The second rule of program optimization (for experts only!): **Do not do it yet**." M. A. Jackson.

17.f: Using higher-level languages helps program **comprehensibility**

18.f: Report how much time is spent on each line of code to identify **bottlenecks**

then rewrite core pieces (possibly in a lower-level language) to get the "**fast**" version

G. Document Design and Purpose, not Mechanics

- Good documentation helps people understand code
- This makes code more reusable and lowers maintenance
- Reference documentation and description of design decisions are keys for improving the understandability of code

19.g: Description of **what a function**, package, macro **does** and its **inputs** and **outputs**
20.g: When possible, rather than write a paragraph to explain a complex piece of code, **re-organize** the code itself, so that it does not need such an explanation
21.g: **Embed the documentation** for the function, package, macro in it, so it is more likely that it is kept up to date and it is more accessible to interactive help

G. Document Design and Purpose, not Mechanics: Two examples in R

Z:\training\Designs of experiments\Design Calculations\Sample size calculations\Phase II\bintest.R - R Editor

```
#############################################################################################
#
# Single-stage designs
#
# Description
#               Function for sample sizes for single-stage designs based on exact binomial
#               test
#
# Usage
#               bintest(p0low, p0high, p1low, p1high, n.max, r=n.max, alpha=0.1, beta=0.1)
#
# Arguments
#               p0low          start value of p0
#               p0high         highest value of p0
#               p1low          start value of p1
#               p1high         highest value of p1
#               n.max          maximum sample size
#               r              cut-off values
#               alpha          type I error rate
#               beta           type II error rate
#
# Details
#               Returns overall power and type I error rate (constrained by alpha, beta,
#               and n.max arguments) for specified ranges of p0 and p1.
# Examples
#               bintest(.1,.15,.2,.2,n.max=80,alpha=.08,beta=.24)
#
# References
#               Khan, Sarker, Hackshaw. Smaller sample sizes for phase II trials based on
#               exact tests with actual error rates by trading-off their nominal levels of
#               significance and power. British J of Cancer (2012) 107, 1801-9.
#
# See also
#               ph2single (clinfun package); bin1samp (desmon package); bintest.sas
#
#############################################################################################
```

G. Document Design and Purpose, not Mechanics:

```
################################################################################################
#
# Power calculations for one- and two-sample t-tests using ratio of means and CV
#
# Description
#                Compute power of test using coefficient of variation or determine other
#                parameters to obtain target power.
#
# Usage
#                power.cv(n = NULL, f = NULL, cv = NULL, sig.level = NULL, power = NULL,
#                        type = c('two.sample', 'one.sample', 'paired'),
#                        alternative = c('two.sided', 'less', 'greater'),
#                        distribution = c('t', 'normal', 'log.normal')))
#
# Arguments
#                n               number of observations (per group)
#                f               ratio of means (>1)
#                cv              coefficient of variation
#                sig.level       significance level (type I error probability)
#                power           power of test (1 - type II error probability)
#                type            type of t-test
#                alternative     one- or two-sided test
#                distribution    underlying distribution assumption
#
# Details
#                Exactly one of n, f, cv, sig.level, and power must be NULL.
#                and n.max arguments) for specified ranges of p0 and p1.
# Value
#                Object of class "power.htest," a list of the arguments (including the comput
#                one) augmented with method and note elements.
# Note
#                uniroot is used to solve power equation for unknowns, so you may see errors
#                from it, notably about inability to bracket the root when invalid arguments
#                are given.
# Examples
#                power.cv(n = NULL, 1.25, .2, .05, .8, distribution = 'normal')
#                power.cv(13, 1.25, .2, .05, power = NULL, distribution = 't')
#                power.cv(13, 1.25, .2, .05, power = NULL, distribution = 'log.normal')
#
# References
#                Van Belle, G., Martin, D. Sample size as a function of coefficient of
#                variation and ratio of means. Am Statistician, Vol. 47, No. 3 (Aug., 1993),
#                pp. 165-7.
#                Martin, D.C., and van Belle, G. Approximations for Power and Sample Size for
#                Student's t-Test. Technical Report 125 (1991), University of Washington, Dep
#                of Biostatistics.
#
#
################################################################################################
```

H. Collaboration and code review

- Reviews of code can eliminate errors and improves readability
- Code review is a good way to spread knowledge and a good practice in a team
- it ensures that critical skills are not lost when a person leave the group

22.h: Code review is better when it is done **before** the code has been committed to a shared version control repository

23.h: An option is the so-called **pair programming** (especially to bring someone new up to speed or when tackling tricky problems)

24.h: It is recommended to maintain a **shared to-do list** of tasks to be performed and errors to be fixed to avoid duplicated work and to transfer tasks to different people

Examples and applications using SAS and R codes

Random number generator

```
DATA normal;
 CALL STREAMINIT(12345);
  DO i=1 to 10000;
   Z=RAND('NORMAL',0,1);
   OUTPUT;
  END;
RUN;

PROC SGPLOT DATA=normal;
 HISTOGRAM Z;
 DENSITY Z / TYPE=Normal(mu=0 sigma=1);
 XAXIS LABEL='Simulated Samples';
run;
```

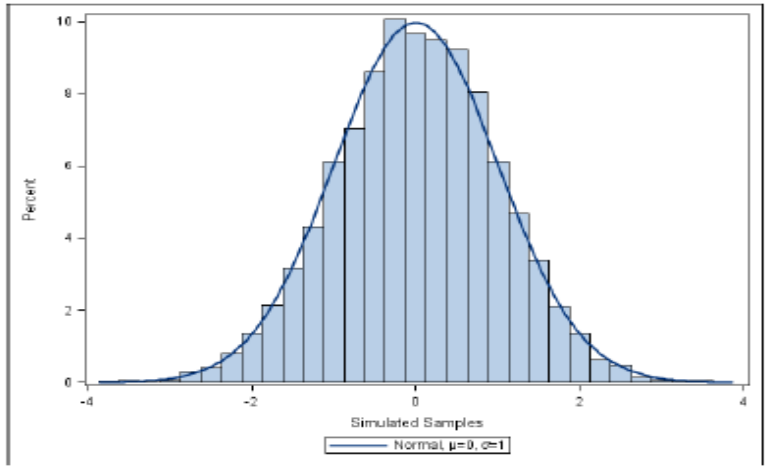


Figure 7. A SAS histogram with a normal overlay

When SAS overlays a normal curve with a histogram the default y-axis is percent, while in R the default y-axis is density. A script in R to generate similar results is shown below in Figure 8.

```
set.seed(12345)
samples<-rnorm(10000,0,1)
hist(samples,
     freq=FALSE,
     breaks=50,
     xlab='Simulated Samples',
     main='Samples from Normal(0,1)')
curve(dnorm(x,0,1),add=TRUE)
```

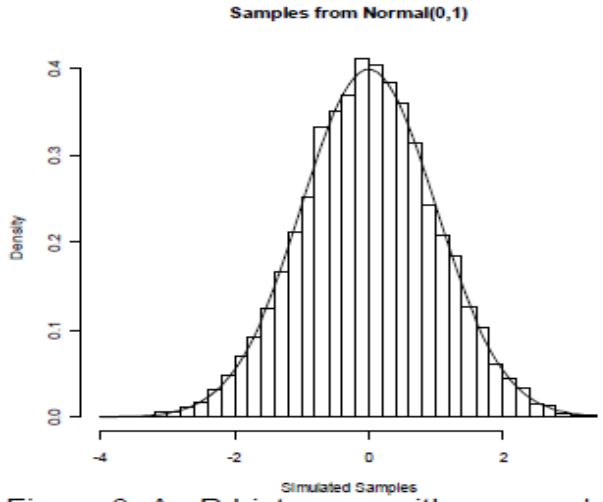


Figure 8. An R histogram with a normal overlay

```
PROC IML;
 CALL RANDSEED(12345);
 z = J(10,1);
 CALL RANDGEN(z,'Normal',0,1);
 PRINT z;
QUIT;
```

Simulate censored survival data in SAS

```
/* Simulate censored survival data;
To simulate survival data with censoring, we need to model the hazard functions
for both time to event and time to censoring.
We simulate both event times from a Weibull distribution with a scale parameter
of 1 (this is equivalent to an exponential random variable).
The event time has a Weibull shape parameter of 0.002 times a linear predictor,
while the censoring time has a Weibull shape parameter of 0.004.
A scale of 1 implies a constant (exponential) baseline hazard, but this can be
modified by specifying other scale parameters for the Weibull random variables.
First we'll simulate the data, then we'll fit a Cox proportional hazards
regression model to see the results.
Simulation is relatively straightforward, and is helpful in concretizing
the notation often used in discussion survival data.
After setting some parameters, we generate some covariate values,
then simply draw an event time and a censoring time.
The minimum of these is "observed" and we record whether it was
the event time or the censoring time.
*/

data simcox;
  beta1 = 2;
  beta2 = -1;
  lambdat = 0.002; *baseline hazard;
  lambdac = 0.004; *censoring hazard;
  do i = 1 to 10000;
    x1 = normal(0);
    x2 = normal(0);
    linpred = exp(-beta1*x1 - beta2*x2);
    t = rand("WEIBULL", 1, lambdaT * linpred);
    * time of event;
    c = rand("WEIBULL", 1, lambdaC);
          * time of censoring;
    time = min(t, c);     * which came first?;
    censored = (c lt t);
    output;
  end;
run;

/*
The phreg procedure (section 4.3.1) will show us the effects of the censoring
s well as the results of fitting the regression model.
We use the ODS system to reduce the output.
*/

ods select censoredsummary parameterestimates;
proc phreg data=simcox;
  model time*censored(1) = x1 x2;
run;
```

Summary of the Number of Event and Censored Values

Total	Event	Censored	Percent Censored
10000	5861	4139	41.39

Analysis of Maximum Likelihood Estimates

Parameter	DF	Parameter Estimate	Standard Error	Chi-Square	Pr > ChiSq	Hazard Ratio
x1	1	1.98593	0.02234	7903.8860	<.0001	7.286
x2	1	-0.99174	0.01583	3924.8053	<.0001	0.371

Simulate censored survival data in R

```
library(survival)

n = 10000
beta1 = 2; beta2 = -1
lambdaT = .002 # baseline hazard
lambdaC = .004  # hazard of censoring

x1 = rnorm(n,0)
x2 = rnorm(n,0)
# true event time
T = rweibull(n, shape=1, scale=lambdaT*exp(-beta1*x1-beta2*x2))
C = rweibull(n, shape=1, scale=lambdaC)    #censoring time
time = pmin(T,C)  #observed time is min of censored and true
event = time==T   # set to 1 if event is observed


#Having generated the data, we assess the effects of censoring with the table() function
and load the survival() library to fit the Cox model.

table(event)



coxph(Surv(time, event)~ x1 + x2, method="breslow")

#These parameters result in data where approximately 40% of the observations
are censored.
#The parameter estimates are similar to the true parameter values.
```

```
> table(event)
event
FALSE  TRUE
 4055  5945
>
>
>
> coxph(Surv(time, event)~ x1 + x2, method="breslow")
Call:
coxph(formula = Surv(time, event) ~ x1 + x2, method = "breslow")

       coef exp(coef) se(coef)      z p
x1  1.995      7.35   0.0222  89.9 0
x2 -0.995      0.37   0.0159 -62.7 0

Likelihood ratio test=11404  on 2 df, p=0, n= 10000, number of events= 5945
```

Simulations for power calculations with survival data in SAS

```
proc iml;
*-------------------------------------------------------------
* define a module to calculate log-rank statistics;
* takes a matrix as input
* in first column are event times (death or censoring)
* in second column are censoring indicators (1 if censored, 0 otherwise)
* in third column are treatment codes(1/0)
-------------------------------------------------------------;
start logrank(times);
* sort matrix;
b=times;
times[rank(times[,1]),]=b;
times=times||j(nrow(times),1,1);
trt1id=times[,4]#(times[,3]=0);
trt2id=times[,4]#(times[,3]=1);
ntrt1 = sum(trt1id);
ntrt2 = sum(trt2id);
* indicator if death occurred;
death1 = (trt1id)#(1-times[,2]);
death2 = (trt2id)#(1-times[,2]);
death=death1+death2;
cumgone1 = cusum(trt1id);
cumgone2 = cusum(trt2id);
cumgone1_={0}//cumgone1[1:nrow(cumgone1)-1];
cumgone2_={0}//cumgone2[1:nrow(cumgone2)-1];
* number at risk before each event time point;
atrisk1 = ntrt1-cumgone1_;
atrisk2 = ntrt2-cumgone2_;
atrisk=atrisk1+atrisk2;
disp = atrisk||atrisk1||atrisk2||death||death1||death2||times[,1];
create tmpdata from disp;
append from disp;
summary class {col7} var {col1 col2 col3} stat{max} opt{noprint save};
summary class {col7} var {col4 col5 col6} stat{sum} opt{noprint save};
close ;
call delete(tmpdata);
atriskf = col1||col2||col3||col4||col5||col6||col7;
small = loc(col1<2);
if nrow(small)>0 then do;
min0 = min(small)-1;
atriskf = atriskf[1:min0,1:5];
end;

w = atriskf[,5]-(atriskf[,2]#atriskf[,4])/atriskf[,1];
v = (atriskf[,2]#atriskf[,3]#atriskf[,4]#(atriskf[,1]- atriskf[,4]))/(atriskf[,1]#atriskf[,1]#(atriskf[,1]-1));
sum_w=sum(w);
sum_v=sum(v);
chisq = sum_w*sum_w/sum_v;
return(chisq);
finish logrank;
```

Simulations for power calculations with survival data in SAS

```
*------------------------------------------------------------------------
start simulation
------------------------------------------------------------------------;
n1 = 500; * group 1 sample size;
n2 = 500; * group 2 sample size;
n=1000;

*hazard/month;
hctrl = 0.0231;
htrt  = 0.0177;

*study duration(month);
studyduration =j(n,1,72);


* initialize a matrix (n row, 1 column, value=0) used for random number generation;
* value=0:system clock is used for the seed;
myran1=j(n1, 1, 0);
myran2=j(n2, 1, 0);

* create treatment groups;
ctrl=j(n1, 1, 0);
trt=j(n2, 1, 1);

* initialize matrix to which we will append values of chi-square statistics from logrank test;
result = j(1,2,0);


do i=1 to 5000 by 1;
* generate enrollment times, event times;
enroll = (36*(uniform(myran1))) // (36*(uniform(myran2)));
event_ = ((-1)*log(1-uniform(myran1))/hctrl) //
         ((-1)*log(1-uniform(myran2))/htrt);
event = enroll+event_;
censor = (studyduration<event);

* final time of event is the smaller of study ending time and event time;
eventT = studyduration#censor+event#(1-censor);
status = eventT||censor||(ctrl//trt);

* run logrank test and append test statistics to result matrix;
chisq = logrank(status);
result = result//(chisq||i);
end;


* output data with logrank test statistics values;
create result from result [colname={'chisq' 'runid'}];
append from result;
quit;
```

Simulations for power calculations with survival data in SAS

```
707
708
709  * output data with logrank test statistics values;
710  create result from result [colname={'chisq' 'runid'}];
711  append from result;
712  quit;
NOTE: Exiting IML.
NOTE: The data set WORK.RESULT has 5001 observations and 2
      variables.
NOTE: PROCEDURE IML used (Total process time):
      real time           1:24.72
      cpu time            47.11 seconds


713
714
715
716
717  * calculate percent of rejections of null hypothesis by
717! logrank test;
718  data result;
719  set result (where=( runid>0));
720    prob = 1-probchi(chisq,1);
721    if prob<0.05 then reject=1; else reject=0; run;

NOTE: There were 5000 observations read from the data set
      WORK.RESULT.
      WHERE runid>0;
NOTE: The data set WORK.RESULT has 5000 observations and 4
      variables.
NOTE: DATA statement used (Total process time):
      real time           0.17 seconds
      cpu time            0.01 seconds


722
723  proc freq data=result;
724  tables reject;
725  title "Percent of rejections of null hypothesis";
726  run;

NOTE: There were 5000 observations read from the data set
      WORK.RESULT.
NOTE: PROCEDURE FREQ used (Total process time):
      real time           1.18 seconds
      cpu time            0.03 seconds
```

```
* calculate percent of rejections of null hypothesis by logrank test;
data result;
set result (where=( runid>0));
  prob = 1-probchi(chisq,1);
  if prob<0.05 then reject=1; else reject=0; run;

proc freq data=result;
tables reject;
title "Percent of rejections of null hypothesis";
run;
```

Percent of rejections of null hypothesis

reject	Frequency	Percent	Cumulative Frequency	Cumulative Percent
0	407	8.14	407	8.14
1	4593	91.86	5000	100.00

Simulations for power calculations with survival data in R

Z:\training\Computing in R and SAS\Wanling codes\R_sim_originaldesign - R Editor

```
# Result: Power=0.9258 from this simulation
#########################################################################
library(survival)

# sample size
n1 = 500
n2 = 500
n=n1+n2

#hazard/month
hctrl = 0.0231
htrt  = 0.0177

# create treatment groups
ctrl=rep(0,n1)
 trt=rep(1,n2)
arm=c(ctrl,trt)
dim(arm)=c(n,1)

# initialize matrix to output logrank p-value
pvalue=1:1
set.seed(2012)
sim=5000

for (i in 1:sim){

#event time
tevent_c=(-1)*log(1-runif(n1))/hctrl
tevent_t=(-1)*log(1-runif(n2))/htrt
tevent = c(tevent_c, tevent_t)

#administrative censoring time
tcensor = 36+36*runif(n)

#event indicator: 1=event 0=censor
eventYN=(tcensor>tevent)+0

#final time
t=tcensor*(1-eventYN)+tevent*eventYN

#logrank test
survtest=survdiff(Surv(t,eventYN)~arm)
df=1
p=1-pchisq(survtest$chisq,df)
pvalue=rbind(pvalue,p)
}

p_logrank=pvalue[2:nrow(pvalue),]
reject=(p_logrank<0.05)+0
mean(reject)
```

Power calculations for two-stage phase II trials in R

```
# stop if observe <= r1 successes among n1 pts
# conclude ineffecitve if observe <=r2 successes among (n1+n2) pts
#non-promising rate is p0, promising is p1
r1=10
r2=25
n1=20
n2=23
p0=.5
p1=.7
pstoph0=pbinom(r1,n1,p0)
pstoph1=pbinom(r1,n1,p1)
temph0=0
temph1=0
for (m in (r1+1): min(r2,n2)) {
   temph0=temph0+dbinom(m,n1,p0) * pbinom(r2-m,n2,p0)
   temph1=temph1+dbinom(m,n1,p1) * pbinom(r2-m,n2,p1)
}


pacch0=pstoph0+temph0
pacch1=pstoph1+temph1
result=cbind(c(pstoph0,pacch0),c(pstoph1,pacch1))
dimnames(result)=list(c("Stop at first stage","Declared inactive"),c("p0","p1"))
```

```
> result
                              p0         p1
Stop at first stage 0.5880985 0.04796190
Declared inactive   0.9007010 0.09136064
```

Power calculations for two-stage phase II trials in R

```
#check calculations using Robert Gray's twostg R program

library('desmon')
ls("package:desmon")

promh0=twostg(20,23,.5,10,25)
promh1=twostgtwostg(20,23,.7, 10,25)
```

```
> twostg
function (n1, n2, p1, r1, r2)
{
    if (n1 < 1 | n2 < 1 | r1 < 0 | r2 < 0 | p1 <= 0 | r1 > n1 |
        r2 > n2 + n1 | p1 >= 1)
        stop("invalid arguments")
    x1 <- 0:n1
    x2 <- 0:n2
    w1 <- dbinom(x1, n1, p1)
    w3 <- dbinom(x2, n2, p1)
    u1 <- c(outer(x1[-(1:(r1 + 1))], x2, "+"))
    u2 <- c(outer(w1[-(1:(r1 + 1))], w3))
    b1 <- c(w1[1:(r1 + 1)], tapply(u2, u1, sum))
    u <- list(inputs = c(n1 = n1, n2 = n2, p1 = p1, r1 = r1,
        r2 = r2), prob.inactive = c(total = sum(b1[1:(r2 + 1)]),
        sum(b1[1:(r1 + 1)])))
    class(u) <- "twostg"
    u
}
<environment: namespace:desmon>
```

```
> promh0
RX1 is declared inactive if <= 10 responses are observed in
the first 20 cases or <= 25 responses are observed in the first 43 cases
P(RX1 declared inactive) = 0.901

P(stop at first stage) = 0.588
> promh1
RX1 is declared inactive if <= 10 responses are observed in
the first 20 cases or <= 25 responses are observed in the first 43 cases
P(RX1 declared inactive) = 0.0914

P(stop at first stage) = 0.048
>
```

References about this topic:

- Best practices for scientific computing. Wilson G, Aruliah DA, Brown CT, Chue Hong NP, Davis M, Guy RT, Haddock SH, Huff KD, Mitchell IM, Plumbley MD, Waugh B, White EP, Wilson P.PLoS Biol. 2014 Jan; 12(1):e1001745. doi: 10.1371/journal.pbio.1001745. Epub 2014 Jan 7

- http://software-carpentry.org/