University of Edinburgh, School of Mathematics

Incomplete Data Analysis, 2020/2021

Multiple imputation of univariate missing data: the `mice` package

Vanda Inácio

Here we will learn how to use the `mice` package in practice. For now, we will only deal with univariate missingness, we will later expand the scope to the case of several variables with missing values. Before proceeding, I leave the reference to the manual of the package

https://cran.r-project.org/web/packages/mice/index.html

I will start by simulating some data and then imposing MAR missingness.

```r
set.seed(1)
n <- 100
x1 <- runif(n, 0, 5)
x2 <- runif(n, 0, 10)
beta0 <- 5
beta1 <- 3
beta2 <- 1
y <- rnorm(n, beta0 + beta1*x1 + beta2*x2, 1)

x2 <- ifelse(x1 > 4.2, NA, x2)
#checking the percentage of missing values
sum(is.na(x2))/n
```

```
## [1] 0.13
```

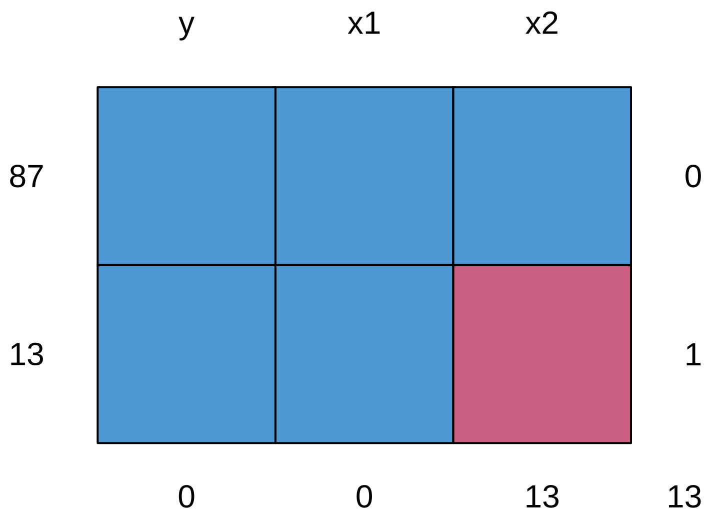
```r
#constructing a dataframe with the 3 variables
simdata <- data.frame("y" = y, "x1" = x1, "x2" = x2)
```

The package `mice` has the function `cc` that returns the complete cases. This function is useful when working with real data as it easily allows some exploratory analyses based on the complete cases.

```r
require(mice)
cc(simdata)
nrow(cc(simdata))
```

As we have seen back in week 2, `mice` also has a function that allows visualising the missing data patterns.

```
md.pattern(simdata)
```



```
##    y x1 x2
## 87 1  1  1  0
## 13 1  1  0  1
##    0  0 13 13
```

Another function available in `mice` is `md.pairs`, which calculates the number of observations per patterns for all possible pairs of variables. For a pair of variables, there are four possible missing data patterns: both variables are observed (pattern `rr`), the first variable is observed and the second variable is missing (pattern `rm`), the first variable is missing and the second variable is observed (pattern `mr`), and finally the pattern where both variables are missing (pattern `mm`).

```
md.pairs(simdata)
```

```
## $rr
##      y  x1 x2
## y  100 100 87
## x1 100 100 87
## x2  87  87 87
##
## $rm
##    y x1 x2
## y  0  0 13
## x1 0  0 13
## x2 0  0  0
##
## $mr
##     y x1 x2
## y   0  0  0
## x1  0  0  0
## x2 13 13  0
##
## $mm
##    y x1 x2
## y  0  0  0
## x1 0  0  0
## x2 0  0 13
```

Let us now use the package `mice` to impute the values in `x2`. We start with the function `mice()` to perform step 1, i.e., to impute the missing values. We already know that the default in `mice` for continuous variables, as it is the case of x_2, is predictive mean matching with $d = 5$ donors and *Type 1* matching (between the cases with missing values and those with observed values). Also, by default in `mice` we have $M = 5$. To know more, type `help(mice)`.

```
imps <- mice(simdata, printFlag = FALSE, seed = 1)
imps
```

```
## Class: mids
## Number of multiple imputations:  5
## Imputation methods:
##     y    x1    x2
##    ""    "" "pmm"
## PredictorMatrix:
##    y x1 x2
## y  0  1  1
## x1 1  0  1
## x2 1  1  0
```

A few comments apply. We set `printFlag = FALSE` which results in silent computation of the missing values and we also use `seed=1` so that our results are reproducible (any other value would obviously work, but fixing the seed outside the function `mice()` will not work). A summary of the imputation results can be obtained by calling the `imps` object. For instance, we see that our saved object `imps` is of class `mids` which stands for *multiply imputed datasets*, which is a special type of object that the `mice` package has set up for storing multiple imputed datasets. We also obtain information about the imputation method used to impute the variables with missing values. In this case only `x2` has missing values and because we have not changed the defaults, unsurprisingly, we have that predictive mean matching was used. Lastly, we have the `predictorMatrix` which, for instance, tell us that `y` and `x1` were used to impute `x2`. It also tells us that in case `y` had missing values, `x1` and `x2` would be used to impute it and similarly for `x1` we would use `y` and `x2`. We can also extract this information from `imps$predictorMatrix`. The default approach in `mice` is to impute one variable based on all other variables.

Now let us look at the imputed values. We can extract them from our `imps` object.

```
imps$imp$x2
```

```
##            1         2        3        4         5
## 4  9.8509522 8.7705754 9.286152 9.240745 7.4107865
## 6  5.0333949 5.0333949 5.476466 5.476466 3.8890510
## 7  2.8323250 1.4821156 3.354875 3.354875 2.8323250
## 18 0.7706438 0.7527575 2.388687 1.734423 0.6380848
## 21 8.5613166 7.4107865 9.286152 7.410786 7.5810305
## 29 2.9373016 1.7344233 2.581659 2.832325 2.7775593
## 52 3.5672691 4.5389549 5.748722 7.293096 6.3349326
## 61 5.0044097 1.9126011 3.913593 3.179637 1.9126011
## 70 5.0333949 1.9126011 3.889051 5.476466 1.7512677
## 76 6.3041412 8.9509410 7.581031 9.850952 8.4061455
## 77 7.2372595 7.1174387 8.405070 8.405070 6.0530345
## 80 8.7705754 9.8509522 7.410786 9.240745 9.8509522
## 94 7.4107865 9.2861520 9.286152 9.850952 7.4107865
```

The row numbers indicate the record number in the original dataset. We can extract, for instance, the imputed values for the first imputed dataset by simply doing the following:

```
imps$imp$x2[,1]
```

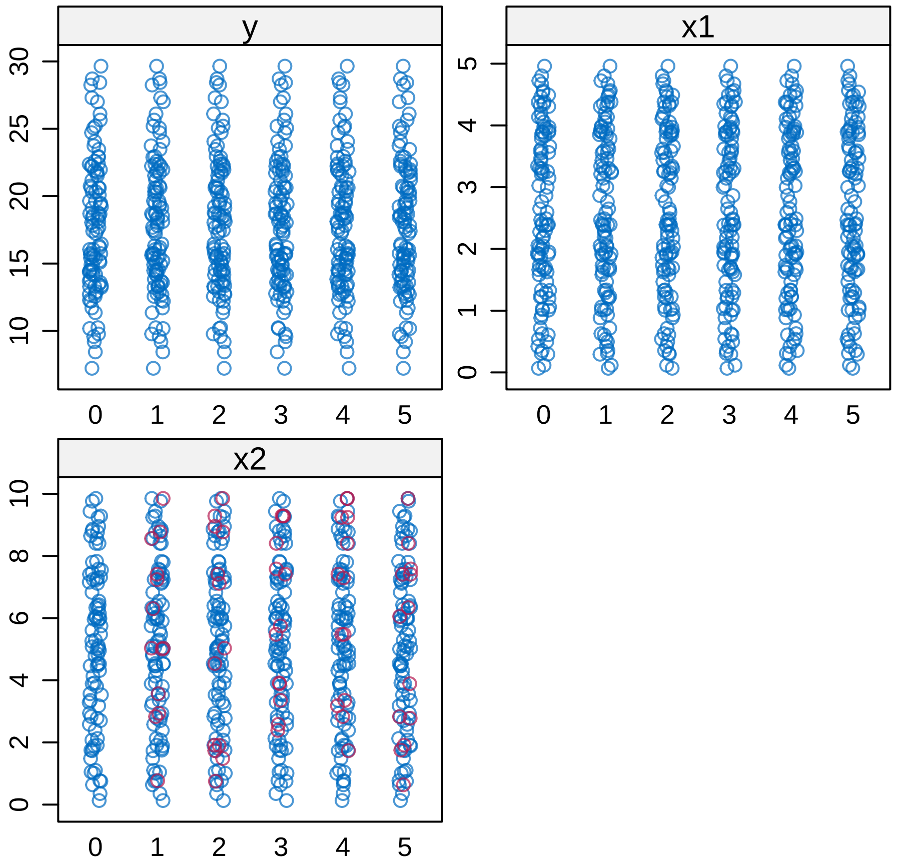
```
##  [1] 9.8509522 5.0333949 2.8323250 0.7706438 8.5613166 2.9373016 3.5672691
##  [8] 5.0044097 5.0333949 6.3041412 7.2372595 8.7705754 7.4107865
```

The (completed) imputed datasets can be extracted by using the `complete` function. As a way of illustrating the usage of this function, I am extracting the first and second completed datasets.

```
com1 <- complete(imps, 1)
com2 <- complete(imps, 2)
com1
com2
```

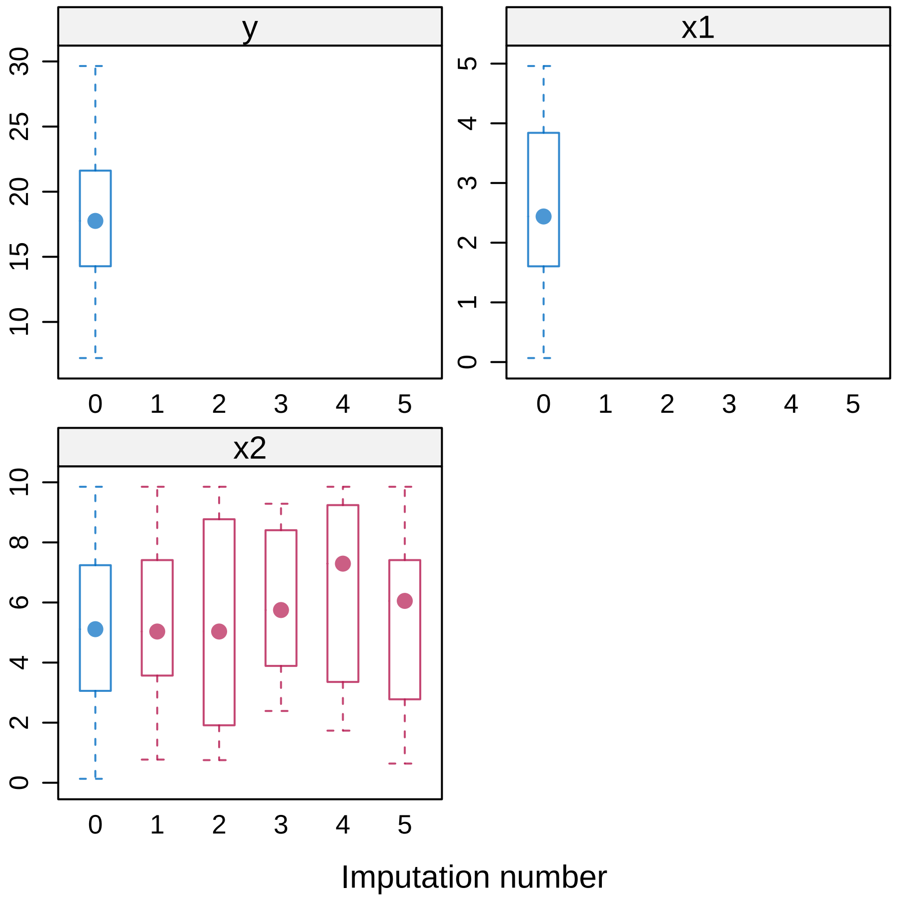
It is also important to visualise the imputation results and the package `mice` provides several plotting tools. This allows us to check whether imputations are plausible. As van Buuren and Groothuis-Oudshoorn say in their paper describing the `mice` package (p. 11): "*Imputations should be values that could have been obtained had they not been missing. Imputations should be close to the data*". One way to do this is through the `stripplot` function.

```
stripplot(imps)
```



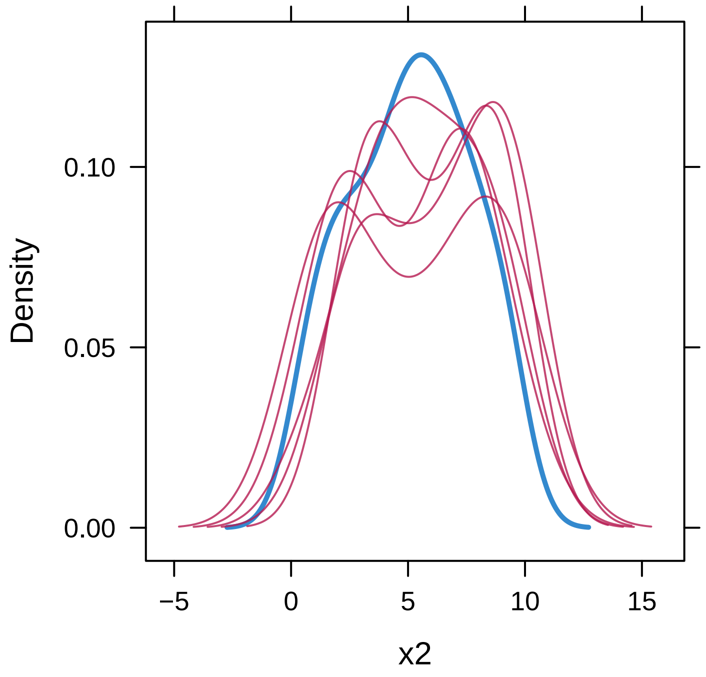
Blue circles denote observed data and red circles imputed data. The panels for `y` and `x1` contain only blue dots because these two variables are fully observed. If there are no large differences between the imputed and observed values then we can conclude that the imputed values are plausible. Here we can see that the red circles follow the blue circles well. If there are discrepancies, interpretation is more difficult, as this may be due to a bad imputation model, due to the missing mechanism not being MAR or due to a combination of both. This plot is most useful when there are not many data points. Alternatively we can use the function `bwplot`, which produces a boxplot of the observed and imputed data.

```
bwplot(imps)
```



There is also the possibility of visualising the kernel density estimates of the observed and imputed data.

```r
densityplot(imps)
```



Aside comment: note that here the densities assign positive mass to negative `x2` values and we know that this should not be the case (`x2` was simulated from a uniform (0,10) distribution). By looking at the boxplots in the previous figure, we see that all imputed values are above zero, and so the imputed values are plausible. However, due to the fact that we are using kernel estimates for the densities, which is a nonparametric density estimator, with a reduced sample size, in combination with the fact that the default kernel is the normal one and there values close to zero, leads to mass assigned to negative values.

Adjustments to the defaults used by the predictive mean matching function `mice.impute.pmm` can be made by simply entering the arguments to be altered into the main `mice()` call. They will be automatically passed down to `mice.impute.pmm`. For instance, the number of donors to be sampled from can be set via the `donors` argument. let us now change this argument to three and we will also create $M = 10$ copies of the dataset (instead of the default $M = 5$).

```
imps_alt <- mice(simdata, m = 10, donors = 3, printFlag = FALSE, seed = 1)
imps_alt
```

```
## Class: mids
## Number of multiple imputations:  10
## Imputation methods:
##     y    x1    x2
##    ""    "" "pmm"
## PredictorMatrix:
##    y x1 x2
## y  0  1  1
## x1 1  0  1
## x2 1  1  0
```

Suppose now that we want to change our method for imputing the missing values. Specifically, suppose that we want to use the method `norm.boot`. There are two possible ways of doing it. The simplest way and feasible only when the number of variables to be imputed is small is to change the method argument directly in the `mice()` call.

```
imps_normb <- mice(simdata, method = "norm.boot", printFlag = FALSE, seed = 1)
imps_normb$imp$x2[,1]
```

```
##  [1] 10.194490  4.595104  2.550149  2.953703  9.353378  2.049476  6.779982
##  [8]  2.345993  4.452382  9.132930  8.257026  8.847016 10.444725
```

An alternative way is to do a setup run of `mice()` without iterations (`maxit=0`) and to extract and modify the method from there.

```
imps0 <- mice(simdata, maxit = 0)
meth <- imps0$method
meth
```

```
##     y    x1    x2
##    ""    "" "pmm"
```

```
meth["x2"] <- "norm.boot"
imps_norm2 <- mice(simdata, method = meth, printFlag = FALSE,
                   seed = 1)
imps_norm2
```

```
## Class: mids
## Number of multiple imputations:  5
## Imputation methods:
##           y          x1          x2
##          ""          "" "norm.boot"
## PredictorMatrix:
```

```
##    y x1 x2
## y  0  1  1
## x1 1  0  1
## x2 1  1  0
```

```
imps_norm2$imp$x2[,1]
```

```
##  [1] 10.194490  4.595104  2.550149  2.953703  9.353378  2.049476  6.779982
##  [8]  2.345993  4.452382  9.132930  8.257026  8.847016 10.444725
```

The setup run is also useful to customize our imputation model. Variables in the columns of the `predictorMatrix` can be switched on or off by using a 1 or a 0 to include or exclude them from the imputation model, respectively. In this way the imputation models for each variable with missing data can be customized (remember that the default is to use all variables in the dataset to impute the variable(s) with missing data). In the hypothetical case that we only want to impute `x2` using `y`, and not both `y` and `x1` (note that this is only to exemplify how to customize the imputation model, I am not saying this is necessarily the way to go in this case).

```
pred <- imps0$predictorMatrix
pred[3,2] <- 0
imps_norm_pred <- mice(simdata, method = meth, predictorMatrix = pred, printFlag = FALSE,
                       seed = 1)
imps_norm_pred
```

```
## Class: mids
## Number of multiple imputations:  5
## Imputation methods:
##           y          x1          x2
##          ""          "" "norm.boot"
## PredictorMatrix:
##    y x1 x2
## y  0  1  1
## x1 1  0  1
## x2 1  0  0
```

We will now proceed to step 2, and we will use the function `with()`. Suppose that our substantive model, i.e., our model of interest, is the model we have used to generate the data, that is

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \varepsilon, \quad \varepsilon \sim \mathrm{N}(0, 1).$$

Just for the sake of illustration, I will be using the completed datasets stored in the object `imps`, using `mice`'s defaults.

```
fits <- with(imps, lm(y ~ x1 + x2))
class(fits)
```

```
## [1] "mira"   "matrix"
```

The object `fits` contains the results of fitting $M = 5$ complete data linear models based on the imputed datasets. The class of `fits` is `mira`, which stands for *multiply imputed repeated analysis*. We can extract the results and corresponding summary of the, say, first and second imputed datasets by doing

```
fits$analyses[[1]]
```

```
##
## Call:
## lm(formula = y ~ x1 + x2)
##
## Coefficients:
```

```
## (Intercept)           x1           x2
##       4.865        2.970        1.031
```

```r
summary(fits$analyses[[1]])
```

```
##
## Call:
## lm(formula = y ~ x1 + x2)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -1.8701 -0.7319 -0.1479  0.5521  2.7268
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  4.86483    0.31565   15.41   <2e-16 ***
## x1           2.96977    0.08084   36.74   <2e-16 ***
## x2           1.03149    0.04159   24.80   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.076 on 97 degrees of freedom
## Multiple R-squared:  0.954,  Adjusted R-squared:  0.9531
## F-statistic:  1006 on 2 and 97 DF,  p-value: < 2.2e-16
```

```r
fits$analyses[[2]]
```

```
##
## Call:
## lm(formula = y ~ x1 + x2)
##
## Coefficients:
## (Intercept)           x1           x2
##       4.918        3.022        1.004
```

```r
summary(fits$analyses[[2]])
```

```
##
## Call:
## lm(formula = y ~ x1 + x2)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -1.9502 -0.6319 -0.1125  0.5266  2.3282
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  4.91822    0.29110   16.89   <2e-16 ***
## x1           3.02248    0.07521   40.19   <2e-16 ***
## x2           1.00371    0.03727   26.93   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.001 on 97 degrees of freedom
## Multiple R-squared:  0.9602, Adjusted R-squared:  0.9593
## F-statistic:  1169 on 2 and 97 DF,  p-value: < 2.2e-16
```

The final step is to combine (pool) the analyses to the final estimates using the `pool` function.

```
ests <- pool(fits)
ests
```

```
## Class: mipo    m = 5
##          term m estimate        ubar            b           t dfcom       df
## 1 (Intercept) 5 4.920379 0.091143492 0.0114905348 0.104932134    97 60.87176
## 2          x1 5 2.962442 0.006066019 0.0054888169 0.012652599    97 11.14942
## 3          x2 5 1.022686 0.001560622 0.0001699438 0.001764554    97 65.64469
##         riv    lambda       fmi
## 1 0.1512850 0.1314053 0.1586034
## 2 1.0858160 0.5205713 0.5883379
## 3 0.1306739 0.1155717 0.1413400
```

```
summary(ests, conf.int = TRUE)
```

```
##          term estimate std.error statistic       df      p.value     2.5 %
## 1 (Intercept) 4.920379 0.3239323  15.18953 60.87176 0.000000e+00 4.2726084
## 2          x1 2.962442 0.1124838  26.33662 11.14942 2.150902e-11 2.7152710
## 3          x2 1.022686 0.0420066  24.34584 65.64469 0.000000e+00 0.9388085
##     97.5 %
## 1 5.568149
## 2 3.209613
## 3 1.106563
```

The object `ests` is of class `mipo`, meaning *multiply imputed pooled outcomes*. Its printed output resembles the output of an `lm` object, but note that its content is different: `pool` gathers the data in `mipo` in a `mira` way that makes summarising the statistics using `summary` easier. One cannot therefore use `residuals` or `predict` to obtain residuals or predictions from the final estimated model.

The column `estimate` correspond to the pooled regression coefficients and their corresponding standard error is available in `std.error`. By further inspecting the output we have columns corresponding to `ubar`, which is the within-imputation variance $\bar{U}$, `b` corresponds to the between-imputation variance, `rvi`, which stands for *relative increase in variance* due to the missing values and as we have learned last week, its expression is given by $\frac{B+\frac{B}{M}}{\bar{U}}$, the column corresponding to `lambda`, which is the proportion of variance in the parameter of interest due to the missing values and which is given by $\frac{B+\frac{B}{M}}{VT}$. Finally, `fmi` contains the *fraction of missing information* as defined in Rubin (1987), and it depends on `rvi` but we will not study it further.

We can also only select the columns we are interested in from the summary, as illustrated below. Further note that the argument `conf.int = TRUE` computes a 95% confidence interval for the (pooled) coefficient estimates.

```
summary(ests, conf.int = TRUE)[, c(2, 3, 7, 8)]
```

```
##   estimate std.error     2.5 %   97.5 %
## 1 4.920379 0.3239323 4.2726084 5.568149
## 2 2.962442 0.1124838 2.7152710 3.209613
## 3 1.022686 0.0420066 0.9388085 1.106563
```

For **linear** regression models, the pooled R^2 can be calculated using the function `pool.r.squared()`.

```
pool.r.squared(fits, adjusted = TRUE)
```

```
##                est     lo 95     hi 95 fmi
## adj R^2 0.9565584 0.934472 0.9713136 NaN
```

The arguments `adjusted` specifies whether the adjusted R^2 or the standard R^2 is returned.

To conclude, let us check the effect of the choice of M on the results which, of course, in practice, depends on the particular analysis we are doing.

```
#using the default M=5 but changing the seed
ests_seed2 <- pool(with(mice(simdata, printFlag = FALSE, seed = 11), lm(y ~ x1 + x2)))
ests_seed3 <- pool(with(mice(simdata, printFlag = FALSE, seed = 111), lm(y ~ x1 + x2)))

summary(ests, conf.int = TRUE)[, c(2, 3, 6, 7, 8)]
```

```
##   estimate std.error      p.value     2.5 %   97.5 %
## 1 4.920379 0.3239323 0.000000e+00 4.2726084 5.568149
## 2 2.962442 0.1124838 2.150902e-11 2.7152710 3.209613
## 3 1.022686 0.0420066 0.000000e+00 0.9388085 1.106563
```

```
summary(ests_seed2, conf.int = TRUE)[, c(2, 3, 6, 7, 8)]
```

```
##   estimate  std.error p.value     2.5 %   97.5 %
## 1 4.916537 0.32029351       0 4.2758627 5.557211
## 2 2.967583 0.08533977       0 2.7957871 3.139380
## 3 1.021980 0.04153976       0 0.9389839 1.104976
```

```
summary(ests_seed3, conf.int = TRUE)[, c(2, 3, 6, 7, 8)]
```

```
##   estimate  std.error p.value    2.5 %   97.5 %
## 1 4.953026 0.29949460       0 4.358058 5.547995
## 2 2.934533 0.08122499       0 2.772357 3.096709
## 3 1.025633 0.04075525       0 0.944386 1.106881
```

```
#using the M=20 and changing the seed
ests_seed1_20 <- pool(with(mice(simdata, printFlag = FALSE, seed = 1, m = 20), lm(y ~ x1 + x2)))
ests_seed2_20 <- pool(with(mice(simdata, printFlag = FALSE, seed = 11, m = 20), lm(y ~ x1 + x2)))
ests_seed3_20 <- pool(with(mice(simdata, printFlag = FALSE, seed = 111, m = 20), lm(y ~ x1 + x2)))

summary(ests_seed1_20, conf.int = TRUE)[, c(2, 3, 6, 7, 8)]
```

```
##   estimate  std.error p.value    2.5 %   97.5 %
## 1 4.989416 0.30828928       0 4.376294 5.602539
## 2 2.927841 0.08412481       0 2.760049 3.095633
## 3 1.020766 0.04059519       0 0.940015 1.101517
```

```
summary(ests_seed2_20, conf.int = TRUE)[, c(2, 3, 6, 7, 8)]
```

```
##   estimate  std.error p.value     2.5 %   97.5 %
## 1 4.959724 0.29870062       0 4.3662580 5.553191
## 2 2.943216 0.08141206       0 2.7810592 3.105373
## 3 1.021504 0.03921262       0 0.9435842 1.099424
```

```
summary(ests_seed3_20, conf.int = TRUE)[, c(2, 3, 6, 7, 8)]
```

```
##   estimate  std.error p.value     2.5 %   97.5 %
## 1 4.943622 0.31144493       0 4.3241406 5.563103
## 2 2.943956 0.08943396       0 2.7647633 3.123148
## 3 1.024435 0.04026630       0 0.9443986 1.104472
```

```
#using the M=50 and changing the seed
ests_seed1_50 <- pool(with(mice(simdata, printFlag = FALSE, seed = 1, m = 50), lm(y ~ x1 + x2)))
ests_seed2_50 <- pool(with(mice(simdata, printFlag = FALSE, seed = 11, m = 50), lm(y ~ x1 + x2)))
ests_seed3_50 <- pool(with(mice(simdata, printFlag = FALSE, seed = 111, m = 50), lm(y ~ x1 + x2)))
```

```r
summary(ests_seed1_50, conf.int = TRUE)[, c(2, 3, 6, 7, 8)]
```

```
##   estimate  std.error p.value     2.5 %   97.5 %
## 1 4.934601 0.30343874       0 4.3318015 5.537400
## 2 2.949604 0.08396988       0 2.7823568 3.116850
## 3 1.024346 0.04010979       0 0.9446435 1.104049
```

```r
summary(ests_seed2_50, conf.int = TRUE)[, c(2, 3, 6, 7, 8)]
```

```
##   estimate  std.error p.value     2.5 %   97.5 %
## 1 4.947824 0.30269210       0 4.3462270 5.549420
## 2 2.946671 0.08658607       0 2.7738311 3.119510
## 3 1.022659 0.03925821       0 0.9446559 1.100661
```

```r
summary(ests_seed3_50, conf.int = TRUE)[, c(2, 3, 6, 7, 8)]
```

```
##   estimate  std.error p.value     2.5 %   97.5 %
## 1 4.945699 0.30578185       0 4.3379153 5.553483
## 2 2.953125 0.08626318       0 2.7810269 3.125224
## 3 1.021044 0.04004144       0 0.9414509 1.100638
```

```r
#using the M=100 and changing the seed
ests_seed1_100 <- pool(with(mice(simdata, printFlag = FALSE, seed = 1, m = 100), lm(y ~ x1 + x2)))
ests_seed2_100 <- pool(with(mice(simdata, printFlag = FALSE, seed = 11, m = 100), lm(y ~ x1 + x2)))
ests_seed3_100 <- pool(with(mice(simdata, printFlag = FALSE, seed = 111, m = 100), lm(y ~ x1 + x2)))

summary(ests_seed1_100, conf.int = TRUE)[, c(2, 3, 6, 7, 8)]
```

```
##   estimate  std.error p.value     2.5 %   97.5 %
## 1 4.960531 0.30267395       0 4.3590596 5.562002
## 2 2.942577 0.08389061       0 2.7754803 3.109673
## 3 1.021493 0.03949196       0 0.9430219 1.099963
```

```r
summary(ests_seed2_100, conf.int = TRUE)[, c(2, 3, 6, 7, 8)]
```

```
##   estimate  std.error p.value     2.5 %  97.5 %
## 1 4.944013 0.30387177       0 4.3401659 5.54786
## 2 2.945124 0.08481618       0 2.7761281 3.11412
## 3 1.023958 0.03973335       0 0.9450059 1.10291
```

```r
summary(ests_seed3_100, conf.int = TRUE)[, c(2, 3, 6, 7, 8)]
```

```
##   estimate  std.error p.value    2.5 %   97.5 %
## 1 4.962865 0.30574579       0 4.355203 5.570527
## 2 2.942194 0.08712432       0 2.768427 3.115962
## 3 1.021138 0.03988191       0 0.941883 1.100394
```

The (pooled) estimates, standard errors, and the bounds of the intervals get more stable as M increases and we can be more confident in any one specific run. Note that whatever value of M we choose, there will always be some variation in results between repeat runs. The point is that with a sufficiently large M, the results will with high probability only differ by a small amount.