fitSR.R - A tutorial

This is a brief guide to using `fitSR.R` to analyse data from Experiments 1-3 as described by Dunn, Anderson and Stephens, *A shift representable model of recognition memory.* This file and associated scripts and data are available at `https://osf.io/n62y4/files`. The executable scripts mentioned in this tutorial are available in the file `fitSR tutorial scripts.R`. It may be helpful to run the relevant scripts from this file as you work through the tutorial. We make use of three R functions. These must be loaded as per:

```
 > wd <- getwd()
> source(file.path(wd,"readexp.R")
> source(file.path(wd,"fitSR.R")
> source(file.path(wd,"standardize.R")
```

There are three data files available: `"Experiment1.csv"`, `"Experiment2.csv"`, and `"Experiment3.csv"` although in this tutorial, we only use the data from Experiment 1. It can be conveniently read using the function `readexp.R`:

```
> y1 <- readexp(1)
```

This returns a data structure with three elements: `y1$participants` is a list of matrices of counts for each participant, `y1$total` is the matrix of counts summed over participants, and `y1$mean` is the matrix of mean counts. Thus:

```
 > y1$participants[[1]]
```

	Sure.old	Probable.old	Possible.old	Possible.new	Probable.new	Sure.new
1	4	7	15	23	37	34
2	5	4	2	5	3	1
3	11	2	3	1	0	3
4	15	2	0	0	2	1
5	4	1	2	3	4	6
6	8	4	5	1	2	0
7	15	1	0	3	1	0

```
> y1$total
```

	Sure.old	Probable.old	Possible.old	Possible.new	Probable.new	Sure.new
1	185	354	601	1091	1430	1739
2	342	145	108	123	89	93
3	470	147	113	60	50	60
4	623	127	72	29	30	19
5	305	145	122	103	119	106
6	435	151	123	69	67	55
7	579	126	73	49	36	37

```
> y1$mean
```

	Sure.old	Probable.old	Possible.old	Possible.new	Probable.new	Sure.new
1	4.111111	7.866667	13.355556	24.2444444	31.7777778	38.6444444
2	7.600000	3.222222	2.400000	2.7333333	1.9777778	2.0666667
3	10.444444	3.266667	2.511111	1.3333333	1.1111111	1.3333333
4	13.844444	2.822222	1.600000	0.6444444	0.6666667	0.4222222
5	6.777778	3.222222	2.711111	2.2888889	2.6444444	2.3555556
6	9.666667	3.355556	2.733333	1.5333333	1.4888889	1.2222222
7	12.866667	2.800000	1.622222	1.0888889	0.8000000	0.8222222

We now analyse `y1$total` using `fitSR.R`:

```
> out <- fitSR(y1$total)
```

The default number of steps (`nstep`) is 20 but a larger number is required for these data. I haven't automated this so some judgment is required. I find it best to approach an appropriate number of steps incrementally. On my laptop this analysis over 20 steps takes about 8sec. This is very slow but I am not a good enough programmer in R to make it go faster (help, please). To look at the output, it is best to use:

```
> str(out,1)
```

because there are several components that are quite large that you don't need to look at (`out$parallel` for one). This returns:

```
List of 20
$ fit :  num 0.22
$ nstep :  num 20
$ mstep :  int 20
$ data :  num [1:35] 0.966 0.9 0.789 0.587 0.322 ...
$ predicted :  num [1:35(1d)] 0.966 0.9 0.789 0.587 0.322 ...
$ z :  num [1:35] 55 41 30 19 0 26 12 1 -10 -29 ...
$ x :  num [1:12] 0 29 42 57 26 39 49 55 41 30 ...
$ fits :  num [1:20] 8.31 6.45 5.61 5.07 3.92 ...
$ minfits :  num [1:20] 8.31 6.45 5.61 5.07 3.92 ...
$ permutation:  num [1:35] 35 34 33 30 21 31 26 22 15 7 ...
$ design :  num [1:35, 1:12] -1 -1 -1 -1 -1 0 0 0 0 0 ...
$ weights :  num [1:35, 1:35] 5400 0 0 0 0 0 0 0 0 0 ...
$ parallel :List of 451
$ type :  chr "Search"
$ init :  num [1:595] 1 1 1 1 1 1 1 1 1 1 ...
$ tol :  num 1e-05
$ duration :  chr "8.08 sec elapsed"
$ y :  int [1:7, 1:6] 185 342 470 623 305 435 579 354 145 147 ...
..- attr(*, "dimnames")=List of 2
$ py :  num [1:7, 1:6] 185 342 470 623 305 ...
$ g2 :  num 4.52
```

At this point, it is useful to compare `out$mstep` which gives the step at which the current minimum was found and `out$nstep` which gives the total number of steps (i.e., permutations) searched. In this case, `out$mstep=20` which is the same as the number of steps, `out$nstep=20`. As a rule of thumb, I generally want `mstep` to be less than half `nstep`. So, we need to up the number of steps. Rather than starting again and wasting another 8sec, we can start from where we left off using the `input` argument this time with an addition 40 steps (bringing the total number of steps to 60):

```
> out <- fitSR(y1$total,nstep=40,input=out)
```

which (after about 12sec on my laptop) gives:

```
> str(out,1)
```

```
List of 20 $ fit :  num 0.0228
$ nstep :  num 60
$ mstep :  int 28
$ data :  num [1:35] 0.966 0.9 0.789 0.587 0.322 ...
$ predicted :  num [1:35(1d)] 0.966 0.9 0.789 0.587 0.322 ...
$ z :  num [1:35] 78 58 43 25 0 36 16 1 -17 -42 ...
$ x :  num [1:12] 0 42 61 87 37 56 73 78 58 43 ...
$ fits :  num [1:60] 8.31 6.45 5.61 5.07 3.92 ...
$ minfits :  num [1:60] 8.31 6.45 5.61 5.07 3.92 ...
$ permutation:  num [1:35] 35 34 33 30 21 31 26 22 15 8 ...
$ design :  num [1:35, 1:12] -1 -1 -1 -1 -1 0 0 0 0 0 ...
$ weights :  num [1:35, 1:35] 5400 0 0 0 0 0 0 0 0 0 ...
$ parallel :List of 451
$ type :  chr "Search"
$ init :  num [1:595] 1 1 1 1 1 1 1 1 1 1 ...
$ tol :  num 1e-05
$ duration :  chr "11.79 sec elapsed"
$ y :  int [1:7, 1:6] 185 342 470 623 305 435 579 354 145 147 ...
..- attr(*, "dimnames")=List of 2
$ py :  num [1:7, 1:6] 185 342 470 623 305 ...
$ g2 :  num 0.611
```

We see that `out$mstep` is now less than half `out$nstep` so we can be satisfied with the solution. Another informative check is to plot the fit value and minimum fit value for each step. Try this:

```
> plot(1:outnstep, outfits, main="Experiment 1 total", cex.main=1.5,col="red",type="l",
```

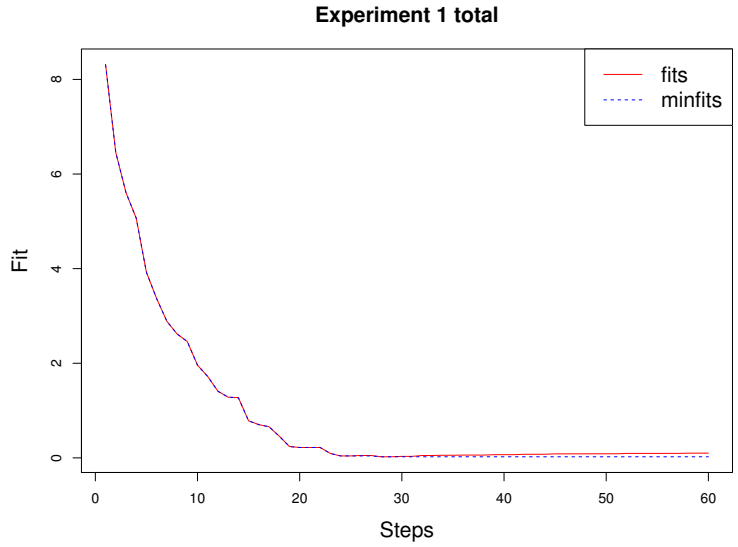

Figure 1: Plot of fits and minfits at each step in the analysis of `y1$total`.

```
xlab="Steps",ylab="Fit",cex.lab=1.5)
> points(1:out$nstep,out$minfits,col="blue",type="l",lty=2)
> legend("topright", legend = c("fits", "minfits"),col = c("red", "blue"), cex = 1.5,
lty=c(1,2))
```

Figure 1 shows the results. What we see is that the fit at each step falls steeply up to about step 20 and, because it is falling, the current minimum (corresponding to a candidate solution) also falls identically since if each fit value is less than the previous minimum fit value then it becomes the new minimum fit value. Eventually, however, this decrease stops and the fit value starts to climb, albeit very slowly while the final minimum fit stays constant. While this suggests that the optimal solution has been found, there is no guarantee that the fit value will continue to rise. It may well fall again, possibly below the obtained minimum fit value, thereby finding a better solution. This is the reason for the heuristic of stopping when the total number of steps is twice the number of steps to find the current minimum. The idea is to give the algorithm the same number of steps to see if it can find a better solution.

We can also look at the quality of the final fit to the data. The predicted response counts are contained in `out$py`:

```
> out$py
```

	[,1]	[,2]	[,3]	[,4]	[,5]	[,6]
[1,]	185	354	601	1091	1430.0	1739.0
[2,]	342	145	108	123	89.0	93.0
[3,]	470	147	113	60	52.5	57.5
[4,]	623	127	72	29	30.0	19.0
[5,]	305	145	122	103	119.0	106.0
[6,]	435	151	121	71	64.5	57.5
[7,]	579	128	71	49	36.0	37.0

which can be compared to `out$total` above. In addition, we can look at the G^2 value as a measure of goodness-of-fit:

```
> out$g2

  [1] 0.6110967
```

Given that we are happy (enough) that we have a solution, the next thing to do is to plot `out$data` against `out$z`. The former is a vector with components equal to one minus the cumulated proportions of `y1$total`. To see these proportions in their usual matrix form, type:

```
> cp <- t(matrix(t(1-out$data),5,7)); cp
```

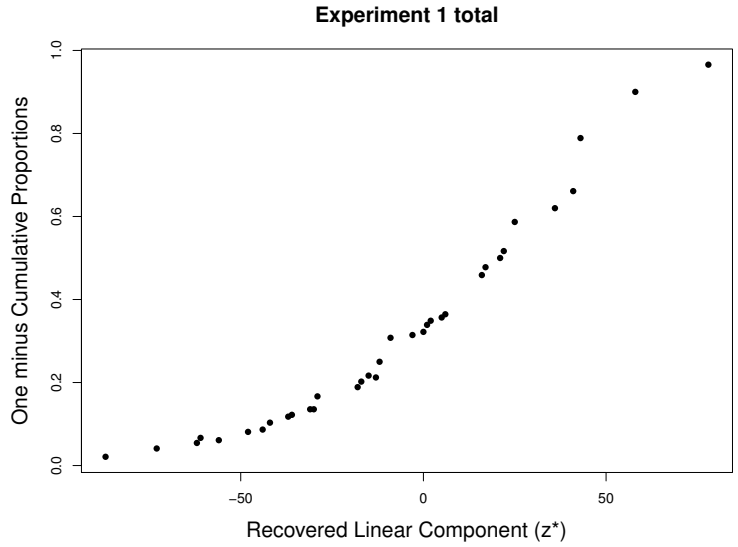

Figure 2: Plot of one minus cumulated proportions against the recovered linear component ($\mathbf{z}^*$).

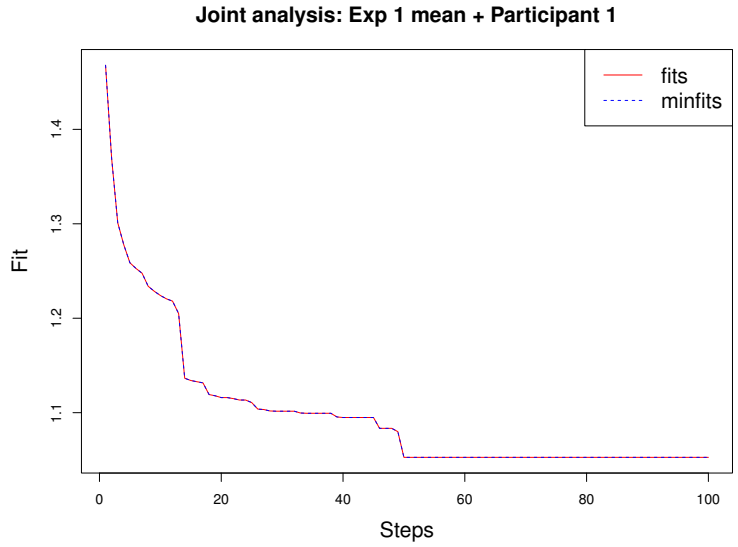

Figure 3: Plot of fits and minfits at each step in the joint analysis of `y1$mean` and `y1$participants[[1]]`.

```
      [,1]       [,2]       [,3]      [,4]      [,5]
[1,] 0.03425926 0.09981481 0.2111111 0.4131481 0.6779630
[2,] 0.38000000 0.54111111 0.6611111 0.7977778 0.8966667
[3,] 0.52222222 0.68555556 0.8111111 0.8777778 0.9333333
[4,] 0.69222222 0.83333333 0.9133333 0.9455556 0.9788889
[5,] 0.33888889 0.50000000 0.6355556 0.7500000 0.8822222
[6,] 0.48333333 0.65111111 0.7877778 0.8644444 0.9388889
[7,] 0.64333333 0.78333333 0.8644444 0.9188889 0.9588889
```

The reason for taking multiple transposes in the previous line of code is because I adopt the convention of indexing by column and then by row rather than by row and then by column (favoured by R). The vector `out$z` contains the corresponding recovered linear component. The plot of `out$data` against `out$z` should then describe the form of the shift representable function F. The following command should generate something like Figure 2:

```
> plot(out$z,out$data,pch=16,main="Experiment 1 total", cex.main=1.5,
xlab="Recovered Linear Component (z*)", ylab="One minus Cumulative Proportions",
cex.lab=1.5)
```

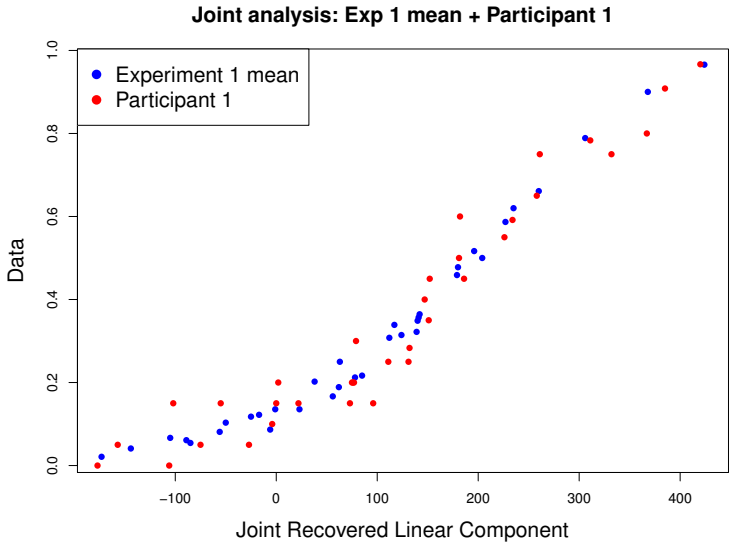

Figure 4: Plot of one minus cumulated proportions against the joint recovered linear component $\mathbf{z}^*(P)$ with $P = \{$`y1$mean`, `y1$participants[[1]]`$\}$.

Joint Analysis

We now conduct a joint analysis of `out$mean` and `out$participants[[1]]`. The reason for using `out$mean` rather than `out$total` is to weight both data sets more or less equally. Here we go:

```
> outm1 <-fitSR(list(y1$mean,y1$participants[[1]]),nstep=100)
```

The following script plots changes in joint fits and minfits:

```
 > plot(1:outm1$nstep,outm1$fits, main="Joint analysis:  Exp 1 mean + Participant 1",
cex.main=1.5, col="red",type="l", xlab="Steps", ylab="Fit", cex.lab=1.5)
> points(1:outm1$nstep,outm1$minfits,col="blue",type="l",lty=2)
> legend("topright", legend = c("fits", "minfits"),col = c("red", "blue"),
cex = 1.5, lty=c(1,2))
```

Figure 3 shows the resulting plot of fits and minimum fits against the number of steps. In this case, the solution was found on step 50 (conveniently half way to 100) although the fit values then stayed constant rather than showing any sign of increasing. On my laptop, this solution was found after 492 sec (a little over 8 min) showing once again the inefficiency of the R code.

The following script plots the observed data against the joint recovered linear component for each element of $P = \{$`y1$mean`, `y1$participants[[1]]`$\}$. Data from `y1$mean` occupy the first 35 (7×5) components of the relevant output vectors and data from `y1$participants[[1]]` occupy the next 35 components of the output vectors. Thus:

```
 > plot(outm1$z[1:35],outm1$data[1:35],pch=16,col="blue",
main="Joint analysis:  Exp 1 mean + Participant 1", cex.main=1.5,
xlab="Joint Recovered Linear Component", ylab="Data",cex.lab=1.5)
> points(outm1$z[36:70],outm1$data[36:70],pch=16,col="red")
> legend("topleft",legend=c("Experiment 1 mean","Participant 1"),col=c("blue","red"),
pch=16, cex=1.5)
```

Figure 4 shows the resulting plot.

Standard Joint Analysis

Although not necessary, it is instructive to analyze the two data sets `y1$mean` and `y1$participants[[1]]` separately. Thus:

```
> outm <- fitSR(y1$mean, nstep=60); out1 <- fitSR(y1$participants[[1]],nstep=25)
```

And then compare the G^2 values with those obtained by the joint analysis. Thus:

```
c(outm$g2, out1$g2); outm1$g2
```

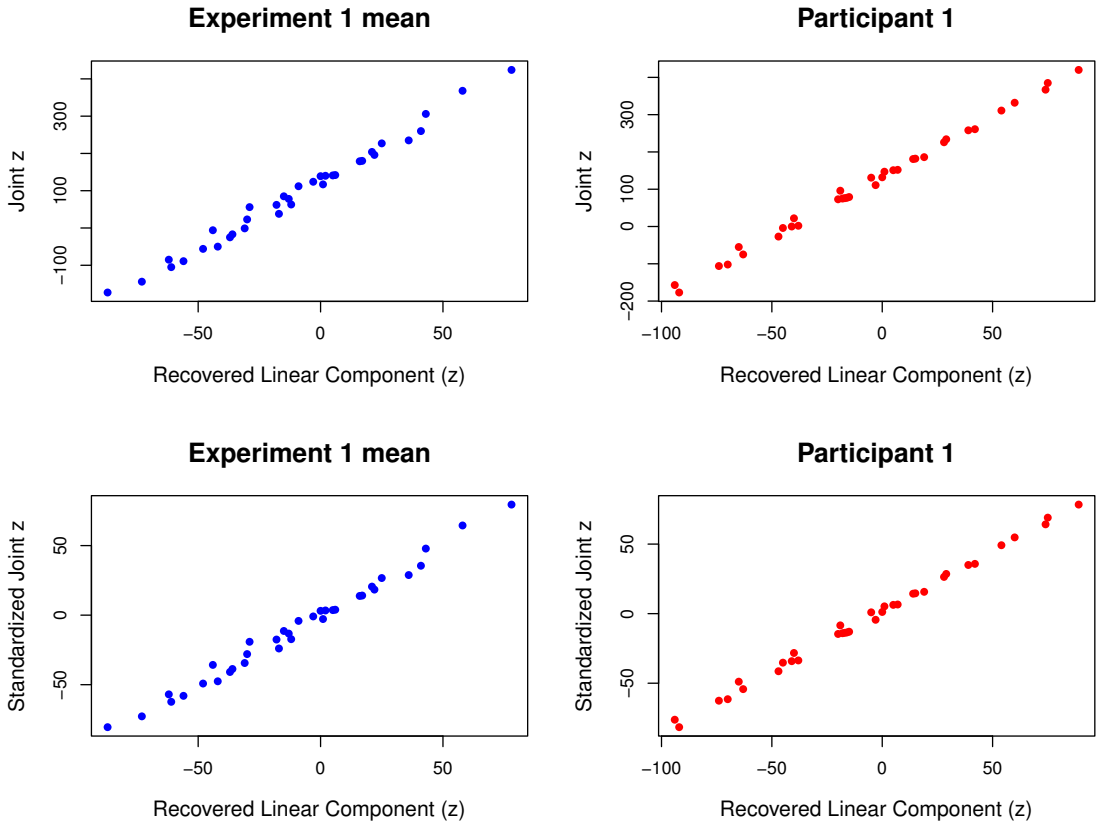

Figure 5: Top left: Plot of joint recovered linear component of `y1$mean` against corresponding recovered linear component. Top right: Plot of joint recovered linear component of `y1$participants[[1]]` against corresponding recovered linear component. Bottom left: Plot of standard joint recovered linear component of `y1$mean` against corresponding recovered linear component. Bottom right: Plot of standard joint recovered linear component of `y1$participants[[1]]` against corresponding recovered linear component.

```
[1] 0.01357993 14.13543381
[1] 2.721796 18.878094
```

This tells us two things. First, the G^2 value is greater for `y1$participants[[1]]` than for `y1$mean` which is attributable to the greater variability/error in the former. Second, both G^2 values increase from the individual to the joint analysis, attributable to the requirement to fit both sets of data to the same function. Nevertheless, the correlations between the individual recovered linear components and the corresponding joint recovered linear components are very high. Thus:

```
 > c(cor(outm$z,outm1$z[1:35]),cor(out1$z,outm1$z[36:70]))
[1] 0.9934934 0.9981873
```

We now standardize the joint analysis with `y1$mean` serving as the standard and `y1$participants[[1]]` serving as the target. Thus:

```
 > sj <- standardize(outm, out1, outm1)
> par(mfrow = c(2, 2))
> plot(outm$z,outm1$z[1:35],pch=16,col="blue", main="Experiment 1 mean", cex.main=1.5,
xlab="Recovered Linear Component (z)", ylab="Joint z", cex.lab=1.25)
> plot(out1$z,outm1$z[36:70],pch=16,col="red", main="Participant 1", cex.main=1.5,
xlab="Recovered Linear Component (z)", ylab="Joint z", cex.lab=1.25)
> plot(outm$z,sj$zs,pch=16,col="blue", main="Experiment 1 mean", cex.main=1.5,
xlab="Recovered Linear Component (z)", ylab="Standardized Joint z", cex.lab=1.25)
> plot(out1$z,sj$zt,pch=16,col="red", main="Participant 1", cex.main=1.5,
xlab="Recovered Linear Component (z)", ylab="Standardized Joint z", cex.lab=1.25)
> par(mfrow = c(1,1))
```

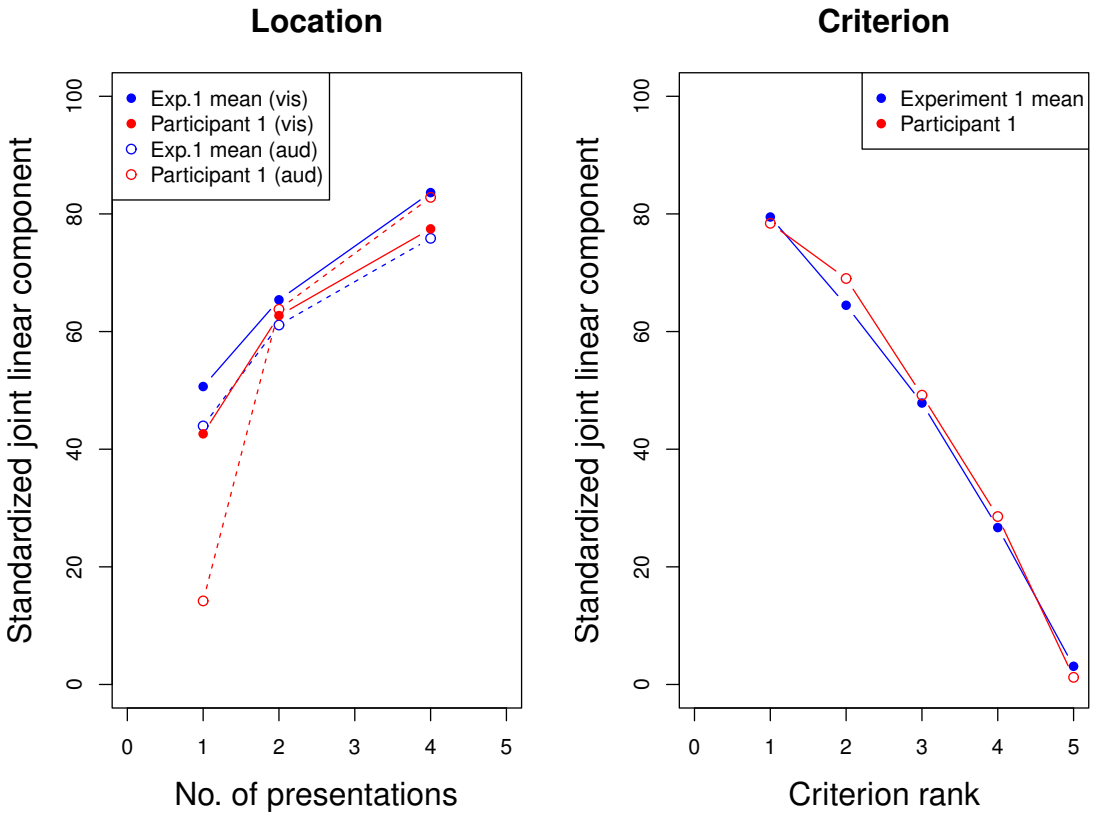

Figure 6: Left: Plot of differences in standardized location parameters with respect to the *New* condition location parameter. Right: Plot of differences in standardized criterion parameters with respect to the *New* condition location parameter.

Figure 5 shows the effect of standardization. The upper two graphs plot each joint recovered linear component against each individual recovered linear component for `y1$mean` on the left and `y1$participants[[1]]` on the right. The lower two graphs are the same except that the joint recovered linear components are replaced by the standardized joint recovered linear components. It is apparent that these components are now on a common scale.

Given that the two sets of parameters are in the same units, they can be plotted on the same scale. The following script generates the graphs shown in Figure 6.

```
dm <- sj$xs[2:7]-sj$xs[1]; d1 <- sj$xt[2:7]-sj$xt[1]
par(mfrow = c(1,2))
x = c(1,2,4)
plot (x, dm[1:3],type="b", pch=16,col="blue",main='Location',cex.main=1.5,
xlab="No.  of presentations",ylab="Standardized joint linear component",
ylim=c(0,100),xlim=c(0,5), cex.lab=1.5)
points (x, dm[4:6],type="b",pch=1,col="blue",lty=2)
points (x, d1[1:3],type="b",pch=16,col="red",lty=1)
points (x, d1[4:6],type="b",pch=1,col="red",lty=2)
legend("topleft",legend=c("Exp.1 mean (vis)","Participant 1 (vis)","Exp.1 mean (aud)",
"Participant 1 (aud)"),col=c("blue","red","blue","red"),pch=c(16,16,1,1), cex=1)
cm <- sj$xs[8:12]-sj$xs[1]; c1 <- sj$xt[8:12]-sj$xt[1]
x = c(1,2,3,4,5)
plot (x, cm[1:5],type="b", pch=16,col="blue",main='Criterion',cex.main=1.5,
xlab="Criterion rank",ylab="Standardized joint linear component",ylim=c(0,100),
xlim=c(0,5),cex.lab=1.5)
points (x, c1[1:5],type="b",pch=1,col="red",lty=1)
legend("topright",legend=c("Experiment 1 mean","Participant 1"),col=c("blue","red"),
pch=16, cex=1)
par(mfrow = c(1,1))
```