# Proposed changes to speed function interface

Sam Rogers

## Summary

Supporting complex designs such as crossed factorial designs and more complex hierarchical designs will require some changes to the interface of the `speed` function.

## Background

The `speed` package currently supports a variety of relatively simple experimental designs, including completely randomised designs, randomised complete block designs, Latin square designs. It can also handle some more complex hierarchical designs such as split-plot and strip-plot designs. A key development goal is extending support to even more complex designs such as factorial designs and hierarchical cases like one-step multi-environment trial designs.

We are mindful of the fact that many users of the package are not R programmers, and so we want to keep the interface as simple as possible for people to use, especially for the more basic designs. However, we also want to provide a lot of flexibility for more advanced users. To this end, we are considering some changes to the interface for specifying the speed function, especially in the context of more complex designs.

Objectives for any changes to the speed interface, roughly in order of priority:

- Ensure that the interface is intuitive and easy to use
- Maintain simplicity for basic designs
- Ensure maximum flexibility for more complex designs
- Provide clear documentation and examples
- Provide good defaults for common use cases
- Allow for custom objective functions to be easily integrated
- Support for both formula and non-formula interfaces
- Minimise the learning curve for new users
- Consistency with other R packages and conventions
- Maintain backward compatibility where possible

## Current Function Signature

The current function interface for `speed` is as follows:

```
speed <- function(data,
                  swap,
                  swap_within = "1",
                  spatial_factors = ~ row + col,
                  grid_factors = list(dim1 = "row", dim2 = "col"),
                  iterations = 10000,
                  early_stop_iterations = 2000,
                  obj_function = objective_function,
                  quiet = FALSE,
                  seed = NULL,
                  ...)
```

This interface is relatively straightforward and works well for simple designs. Arguments are given as single values or simple formulas. For more complex designs, such as split-plot designs, the user currently specifies the design using named lists to match the different components. This allows for a lot of flexibility, because it can enable the user to specify arbitrary nesting structures, and to create designs where different objective functions or different numbers of iterations are applied to different levels of the design for example. However, this gets complicated and verbose as designs increase in complexity, and it is not very intuitive for new users. It may be that a user ends up with several named lists to specify the design.

## The Challenge

The challenge is to find an interface that is both flexible enough to handle complex designs, while also being simple and intuitive for basic designs. We want to avoid a situation where the user has to learn a lot of new syntax or concepts to use the package effectively. We also want to ensure that the interface is consistent with other R packages and conventions, so that users can easily transfer their knowledge from one package to another.

### Design Cases

At a conceptual level, there are three basic cases that we need to consider:

1. Single factor designs (e.g. completely randomised, randomised complete block, Latin square, balanced incomplete block, 2D blocking). These are relatively straightforward, and can be handled with the current interface.

2. Multi factor designs *with* hierarchicy (e.g. split-plot, strip-plot, MET). These are more complex, and require a more flexible interface to specify the different levels of nesting and the relationships between them. These designs are partially supported in the current interface.
3. Multi factor designs *without* hierarchicy (e.g. factorial designs with multiple factors that are crossed). These are also complex, and require a different approach to specifying the design. These designs are not currently supported in the package.

Case 3 and some situations arising from case 2 are the main focus of the changes under consideration.

### Specific Challenges

- Some designs (e.g. split-plot, strip-plot, split-split plot) require multiple levels of nesting, where the higher level units all move together during swaps, while lower level units can move independently (within the constraints of the higher level units). We need a way to specify this hierarchy and how treatments are assigned and bounded at each level clearly and intuitively.
- Factorial designs have multiple factors that are crossed without any hierarchy. This means all experimental units can move independently within the constraints of the design. A user may want to optimise the design with respect to the multiple factor main effects, or combinations of factors or both. We need a way to specify these factors and how a user wants to optimise the design clearly and intuitively.
- Some designs may require different objective functions, iterations and/or spatial or grid factors to be applied at different levels of the design. We need a way to specify these options clearly and intuitively.

## Proposed Interface Options

We are considering two main options for the interface to better support complex designs while maintaining simplicity for basic designs. These options are not mutually exclusive, and we could potentially implement both. We are also open to other suggestions. The first option is basically an extension of the current interface, while the second option is a more radical change. A third option is a formula interface, which we are planning to implement regardless of which of the other two options we choose as an alternative interface for those who prefer it.

## 1. Extended Current Interface (Parallel lists style)

The current interface supports hierarchical designs using named lists to specify the different levels of the design across the different arguments. However, some designs, such as factorial designs, will not fit within this framework as they do not have a hierarchical structure. Similarly, MET designs may have a hierarchical structure, but the treatments typically do not follow a strict nesting structure, as locations may share some treatments but not others. This means that the current approach of using named lists to specify the different levels of the design may not be sufficient.

### Supporting Factorial Designs

We propose to extend this approach to also support factorial designs by allowing users to specify multiple factors in the `swap` argument either via named lists or vectors. They may also want to optimise the interaction term, so that the interaction structure of the design should not be broken. For example, a user could specify a factorial design with two factors `A` and `B` as `swap = c("A", "B", "A:B")`. This would indicate that both factors can be swapped independently, while also swapping and maintaining the interaction structure during the optimisation process.

### Supporting Multi-Environment Trial Designs

For hierarchical designs where treatments are linked such as a split-plot design, the user could continue to use named lists to specify the different levels of the design as they do currently, but they would need to use a different argument to indicate that the treatments are linked across the different levels. For example, a user could specify a split-plot design with two levels of treatments `wp_treatment` and `sp_treatment` as `swap = list(wp = "wp_treatment", wp = "sp_treatment")` and a new argument `linked = TRUE`. This would indicate that the treatments are linked across the different levels of the design, and that the optimisation process should maintain this linkage during swaps. To generalise this further, we could allow the user to specify which factors are linked via a named list, e.g., `linked = list(wp = TRUE, sp = FALSE)`. This would allow for more complex designs where only some (or no) treatments are linked across levels.

Alternatively, we could introduce a new function `all()` which would wrap around the factors that are linked, e.g., `swap = list(wp = all("wp_treatment"), sp = "sp_treatment")`. This would indicate that all the units with the same `wp_treatment` should move together during swaps, while the `sp_treatment` can move independently within the constraints of the `wp_treatment`. This approach would be more consistent with how other R packages handle similar situations (e.g., `dplyr::across()`).

### Pros

- Maintains signposting for people less familiar with R via auto-completion of argument names and popups in RStudio etc. This allows users to see all the available arguments and their descriptions easily.

### Cons

- May become verbose and complicated for very complex designs with many named lists.

## 2. Nested List Interface

### Pros

- More structured and organised for complex designs
- Easier to read and understand the design structure at a glance

### Cons

- More complex syntax may be intimidating for new users
- Less familiar for existing users

## 3. Formula Interface

We also plan to introduce a formula interface for specifying the experimental design. This would allow users to specify the design using a formula syntax similar to that used in other R packages (e.g., `lm`, `asreml`). For simple designs this wouldn't be very different to either of the current setups, but could be more intuitive for users with statistical modelling experience in more complex designs. For example, a user could specify a randomised complete block design as `swap = ~treatment`, `swap_within = ~block`. For a split-plot design, they could specify `swap = ~wp_treatment + sp_treatment`, `swap_within = ~block/wholeplot`. For a factorial design, they could specify `swap = ~A * B` to indicate that both factors and their interaction should be optimised.

### Pros

- Intuitive and familiar for users of other R packages, especially those with statistical modelling experience
- Concise and expressive syntax

#### Cons

- Not as intuitive for beginners and users without statistical modelling experience
- Potential for confusion around the model-free nature of the optimisation process

## Comparison of Proposed Interface Options

For simplicity in this discussion, we will assume a data frame `data` has been provided with the appropriate columns for the design factors, and the other arguments not related to the design specification (e.g., `iterations`, `quiet`, `seed`, etc.) remain unchanged.

### RCB Design Example

#### 1. Parallel lists style

```
rcb_design <- speed(df_rcb,
                    swap = "treatment",
                    swap_within = "block")
```

#### 2. Nested List style

```
rcb_design <- speed(df_rcb,
                    optimise = list(swap = "treatment",
                                    swap_within = "block"))
```

#### 3. Formula Interface

```
rcb_design <- speed(df_rcb,
                    swap = ~treatment,
                    swap_within = ~block)

# or equivalently

rcb_design <- speed(df_rcb,
                    optimise = list(swap = ~treatment,
                                    swap_within = ~block))
```

### Split-Plot Design Example

In this case, we have two levels of treatments: `wp_treatment` and `sp_treatment`. The whole-plot treatments are assigned to whole plots, which are nested within blocks. The subplot treatments are assigned to subplots, which are nested within whole plots. The whole plots move together during swaps, while the subplots can move independently within the constraints of the whole plots. This is signified by the `linked` argument or the `all()` function.

#### 1. Parallel lists style

```
sp_design <- speed(df_split,
                   swap = list(wp = all("wp_treatment"),
                               sp = "sp_treatment"),
                   swap_within = list(wp = "block",
                                      sp = "wholeplot"))

# alternatively

sp_design <- speed(df_split,
                   swap = list(wp = "wp_treatment",
                               sp = "sp_treatment"),
                   swap_within = list(wp = "block",
                                      sp = "wholeplot"),
                   linked = list(wp = TRUE, sp = FALSE))
```

#### 2. Nested List Interface

```
# set up the parameters first for clarity
wp <- list(swap = all("wp_treatment"), swap_within = "block")
sp <- list(swap = "sp_treatment", swap_within = "wholeplot")
sp_design <- speed(df_split, optimise = list(wp, sp))

# or equivalently
sp_design <- speed(df_split,
                   optimise = list(wp = list(swap = all("wp_treatment"),
                                             swap_within = "block"),
                                   sp = list(swap = "sp_treatment",
                                             swap_within = "wholeplot")))
```

### 3. Formula Interface

```
# Parallel lists style
rcb_design <- speed(df_split,
                    swap = ~all(wp_treatment) + sp_treatment,
                    swap_within = ~block/wholeplot)
# or
rcb_design <- speed(df_split,
                    swap = ~ wp_treatment + sp_treatment,
                    swap_within = ~block/wholeplot,
                    linked = list(wp_treatment = TRUE,
                                  sp_treatment = FALSE))

# or equivalently
rcb_design <- speed(df_split,
                    optimise = list(swap = ~wp_treatment + sp_treatment,
                                    swap_within = ~block/wholeplot,
                                    linked = list(wp_treatment = TRUE,
                                                  sp_treatment = FALSE)))
```

## Multi-Environment Trial Design Example

In a MET, treatments (typically varieties) will first be allocated to locations (sites), and then to plots within locations. Locations may share some treatments but not others, so the treatments are not strictly nested within locations. However, we may want to ensure that there is a certain level of connectivity between locations, so that some treatments are shared between locations. We may also want to ensure that the design is balanced within locations, so that each treatment appears a similar number of times within each location.

### 1. Parallel lists style

```
met_design <- speed(df_met,
                    swap = list(con = "variety", # connectivity
                                bal = "variety"), # balance
                    swap_within = list(con = "1",
                                       bal = "site"),
                    spatial_factors = list(con = ~ site,
                                           bal = ~ site_row + site_col + site_block))
```

### 2. Nested List Interface

```
con <- list(swap = "variety", swap_within = "1", spatial_factors = ~ site)
bal <- list(swap = "variety", swap_within = "site", spatial_factors = ~ site_row +
              site_col + site_block)
met_design <- speed(df_met, optimise = list(con, bal))

# or equivalently
met_design <- speed(df_met,
                    optimise = list(con = list(swap = "variety",
                                               swap_within = "1",
                                               spatial_factors = ~ site),
                                    bal = list(swap = "variety",
                                               swap_within = "site",
                                               spatial_factors = ~ site_row +
                                                 site_col + site_block)))
```

### 3. Formula Interface

```
# Parallel lists style
met_design <- speed(df_met,
                    swap = ~treatment,
                    swap_within = ~ 1 + (site/block + site/row + site/col))

# or equivalently
met_params <- list(swap = ~wp_treatment + sp_treatment,
                   swap_within = ~ 1 + (site/block + site/row + site/col))
met_design <- speed(df_met,
                    optimise = met_params)
```

## Factorial Design Example

The complication with factorial designs is that there is no hierarchy, so all factors can move independently. However, we may want to ensure that the interaction structure of the design is maintained during swaps, so as to avoid breaking up treatment combinations that are important for estimating interaction effects and result in imbalanced designs. We propose to handle this by allowing users to specify interaction terms in the `swap` argument, e.g., `swap = c("A", "B", "A:B")`. This would indicate that both factors can be swapped independently, while also swapping and maintaining the interaction structure during the optimisation process.

### 1. Parallel lists style

```
fac_design <- speed(df_fac,
                    swap = "A*B",   # Interpreted as c("A", "B", "A:B")
                    swap_within = "block")
```

### 2. Nested List Interface

```
fac_params <- list(swap = "A*B", swap_within = "block")
fac_design <- speed(df_fac, optimise = fac_params)

# or equivalently
sp_design <- speed(df_fac, optimise = list(swap = "A*B", swap_within = "block"))
```

### 3. Formula Interface

```
# Parallel lists style
fac_design <- speed(df_fac,
                    swap = ~A*B,  # Or A + B + A:B
                    swap_within = ~block)

# or alternatively
fac_design <- speed(df_fac, optimise = list(swap = ~A*B, swap_within = ~block))
```

## Questions for Testers

Please consider the proposed alternative interface versions and provide feedback on any of the following questions:

- Which signature feels most natural to use?
- Which would you be most likely to adopt in your own work?
- Are there specific features or arguments that must remain easy to access?
- Do you see any barriers for new users learning the function?
- Are there any specific use cases or designs that you would like to see supported?
- Do you have any other suggestions for improving the clarity and/or flexibility of the interface?