A.1 | Numerical Optimization in Exploratory Data Analysis

Having developed and evaluated different methods of exploring central tendency, dispersion, and covariability in one, two, and more than two dimensions, we can now turn our attention to *numerical optimization*. This computational method involves finding a set of model parameters that maximizes or minimizes an objective function in order to arrive at an optimal solution to some problem. Although this method is more commonly used in the computation of model parameter estimates (as we will see in Chapter 3), it can also be used in certain aspects of exploratory data analysis. First, however, we need to develop a better understanding of the syntax and structure of functions in R, both built-in and those defined by the user themselves.

A.1.1 | The Structure of Functions in R

At its core, a *function* is simply a collection of statements organized together to perform a specific task. In addition to a large number of built-in functions, some of which we have explored before, R allows its users to create their own functions to accomplish various tasks. These *user-defined functions* are stored as objects in R and, in turn, return values to the user that may then be stored as objects themselves.

In R, we can use the `function()` keyword to define a function.

```
function_name <- function(argument) {
   # function body
}
```

The basic syntax of a function includes four components that we will explore in more detail – the *name*, *argument*, *body*, and *return value*.

As an example, let's create a function that converts temperatures from Fahrenheit to Celsius using the following equation:

$$°\mathrm{C} = (°\mathrm{F} - 32) \times \frac{5}{9} \tag{A.1}$$

In R, we can implement this equation using the `function()` keyword and basic operations.

```
fahrenheit_to_Celsius <- function(temp_F = 0) {
   temp_C <- (temp_F - 32) * 5/9
   return(temp_C)
}
```

The *name* of this new function is `fahrenheit_to_Celsius`, which is stored in R as an object. This allows us to call this and other user-defined functions much as we would a built-in function.

The object `temp_F` serves as the *argument*, which is a value passed to the function, with a default value of `0`. Although most functions have at least one argument, the inclusion of arguments is actually optional and there are functions, such as `getwd()`, with no arguments.

The function *body* is enclosed inside curly brackets and contains the collection of statements that will be executed when the function is called. In our example, these statements take the argument `temp_F`, subtract `32` from it, multiply the result by `5/9`, and then store the resulting value in a new object `temp_C`.

The final step is to *return* this value back to the user using the `return()` function.

Now that this function has been defined in R, we can use it to convert any temperature from Fahrenheit to Celsius.

```
fahrenheit_to_Celsius(32)
[1] 0

fahrenheit_to_Celsius(212)
[1] 100
```

Using our example function, we can see that a temperature of 32°F corresponds to a temperature of 0°C, whereas a temperature of 212°F corresponds to 100°C. These of course represent the freezing and boiling points of water.

Finally, it is important to note that functions can be defined to accept more than a single argument. For example, we can define a function that takes, as arguments, the number of pounds and ounces that a person weighs and return the equivalent weight in kilograms.

```
poundOunces_to_Kilograms <- function(lb = 100, oz = 0) {
   pounds <- lb + oz / 16
   kilograms <- pounds * 0.453592
   return(kilograms)
}

poundOunces_to_Kilograms(15, 15.2)
[1] 7.234792
```

This weight of 15 pounds and 15.2 ounces or, equivalently, 7.23 kilograms, corresponds to Jacob Schmitz, Jr., the biggest baby on record in Minnesota, who was born on July 16, 1936.

Alternatively, multiple arguments can be concatenated together into a single argument using the `c()` function.

```
poundOunces_to_Kilograms_2 <- function(weight = c(100,0)) {
   pounds <- weight[1] + weight[2] / 16
   kilograms <- pounds * 0.453592
   return(kilograms)
}

poundOunces_to_Kilograms_2(c(15, 15.2))
[1] 7.234792
```

When passing multiple arguments as a single object, each component can be refenced using standard vector notation with brackets. This second approach, as we will see below, is particularly useful when performing numerical optimization.

A.1.2 | Numerical Optimization in R

Numerical optimization is the selection of a best element or set of elements, with regard to some criterion, from some set of available alternatives. Optimization problems of all sorts arise in numerous quantitative disciplines ranging from computer science and engineering to operations research and economics.

At its core, numerical optimization involves maximizing or minimizing an objective function by systematically choosing input values from within an allowed set and computing the value of the function at that input.

In exploratory data analysis, transformations are commonly used to convert a dimension or variable from one shape to another. This is most often done to decrease the skewness of a dimension in order to make it more closely adhere to a normal bell-shaped pattern (see Section 1.2.4). Aside from logarithms, power transformations are the most frequently used type of data transformations and are achieved by raising each data point in a dimension to a specific power. Common power transformations include square roots (power of one half), squares (power of two), and reciprocals (power of negative one); however, any real number can be used instead.

If the goal is to decrease skewness, a power transformation should be selected in such a way as to make the skewness as close to zero as possible. In this way, identifying the best power transformation is a numerical optimization problem. The input value is the real number to which all the data points are to be raised and the objective function is the skewness of the transformed dimension. Different input values are then systematically evaluated until the absolute smallest skewness (in terms of distance from zero) is achieved. The input value that corresponds to this optimized skewness is then used to define the power transformation.

In order to demonstrate this computational approach, we first need to define an appropriate function in R to return the absolute skewness of a distribution following the application of some power transformation to, for instance, the second dimension in our example data.

```
poweredSkewness <- function(power = 1) {
   transformedValues <- (MG[,2]) ^ power
   resultingSkewness <- abs(skewness(transformedValues))
   return(resultingSkewness)
}
```

Note that we have used the `abs()` function to return the absolute value of the skewness because we want the skewness to be as close as possible to zero (not as numerically small as possible, which would include negative skewness).

We can use this function to compare, for example, whether a square root or a reciprocal transformation provides a less skewed result when compared to the original skewness of the dimension, which is approximately 1.344 (see Section 1.2.3).

```
poweredSkewness(0.5)
[1] 0.6087153

poweredSkewness(-1)
[1] 0.6947116
```

We can see that the square root transformation (taking the data points to the power of one half) provides a less skewed distribution than does a reciprocal transformation (taking the data to the power of negative one). The question then becomes how we can find the power that provides the absolute smallest skewness possible.

One approach to such an optimization problem is *brute force*, which attempts to calculate all possible solutions within a range of input values and afterward determine which provided the best result. Such a method is fairly straightforward but requires considerable computational power, especially if you consider how many significant digits to include in the input values.

Given that optimization problems have existed for centuries, a wide range of *algorithms* have been developed to more efficiently find the best set of input values to minimize a given function. Although a full exploration of these algorithms is beyond the scope of this text (and contains enough content to fill multiple semesters), we will make use of one specific algorithm for the duration of this text.

Particle Swarm Optimization or PSO is an optimization algorithm that is inspired by the social swarming behavior of a flock of birds or school of fish. This computational method is initiated with a group of random candidate solutions known as *particles* that are then allowed to move around the *search space* according to simple mathematical equations.

Much like a school of fish, each particle's movement is influenced by its immediate surroundings (nearby values of the objective function) while also taking into account better locations found by other particles. This is expected to iteratively move the swarm toward the optimal solution.

In R, we can use the psoptim() function in the pso package to implement this algorithm.

```
library(pso)
psoptim(par = c(1), fn = poweredSkewness, lower = -9e9, upper = 9e9)
$par
[1] 0.1725209

$value
[1] 4.584801e-17

$counts
 function iteration  restarts
    12000      1000         0

$convergence
[1] 2

$message
[1] "Maximal number of iterations reached"
```

Just like every other optimization algorithm, PSO begins with a user-defined set of initial values, identified by the par argument, that will serve as the starting point for further optimization of the function identified by the fn argument. The lower and upper arguments provide bounded constrains to the optimization and can simply be set to any ridiculously low and high number.

The most important aspect of the output (which we can explore in more detail using the help documentation available via ?psoptim) is contained in $par, which identifies the set of values that the algorithm determined would produce the absolute smallest function result, which in turn is contained in $value. Using our example output, we can see that a power transformation of approximately 0.173 produces a distribution with a skewness of almost exactly zero. Assuming that the algorithm was successful, then there is no value that this distribution could be raised to that would produce a smaller absolute skewness than that seen above.

Further, the $counts output indicates how many steps the algorithm took before arriving at the optimal solution. The $convergence and $message outputs, on the other hand, indicate if the algorithm successfully converged to an optimal solution.

It is important to note that, unlike many other implementations of numerical optimization algorithms in R, PSO doesn't usually indicate successful convergence even if an optimal solution has been reached. Thankfully, the results derived after one thousand iterations, as was done above, are quite robust compared to these other algorithms. As such, a lack of convergence in PSO is generally not cause for concern.

In conclusion, numerical optimization is a powerful computational method that forms the foundation for many analytical techniques, including aspects of exploratory factor analysis (see Section A.2) and structural equation modeling (see Section A.3). As such, a thorough familiarity with these techniques can be a great asset for professionals in a variety of quantitative fields!

Exercises for Section A.1

A.01 Wolves and Moose on Isle Royale (continued). Exercise 1.1 introduced an excerpt of the data collected on the number of wolves and moose on Isle Royale, a subset of which (omitting any cases with NA values) is recreated below:

	Wolves	Moose	Kill Rate	Predation Rate	Moose Recruitment Rate
1971	20	1183	0.615	0.062	11.4
1972	23	1243	0.819	0.091	10.7
1973	24	1215	0.760	0.090	15.1
1974	31	1203	0.599	0.093	14.7
1975	41	1139	0.645	0.139	12.3
1976	44	1070	0.563	0.139	10.3
1977	34	949	0.298	0.064	6.1
1978	40	845	0.507	0.144	10.2
1979	43	857	0.387	0.117	13
1980	50	788	0.330	0.126	12
1981	30	767	0.217	0.051	23.7
1982	14	780	0.869	0.094	20.7
1983	23	830	0.394	0.065	16.4
1984	24	927	0.440	0.068	14.5
1985	22	976	0.457	0.062	13.1
1986	20	1014	0.670	0.079	16
1987	16	1046	0.549	0.050	16
1988	12	1116	0.864	0.056	15.2
1989	12	1260	0.810	0.046	13.4
1990	15	1315	0.857	0.059	14.8
1991	12	1496	1.029	0.050	13.9
1992	12	1697	1.428	0.061	10.9
1993	13	1784	0.877	0.038	14
1994	17	2017	0.792	0.040	12
1995	16	2117	1.391	0.063	7
1996	22	2398	1.274	0.070	3

The data above include the number of Wolves and Moose, as well as the number of moose killed per wolf (Kill Rate), the proportion of moose being killed by wolves (Predation Rate), and the proportion of moose that are calves (Moose Recruitment Rate) over what is now a 26-year period.

One way that we can explore the covariability between these different dimensions is by using a *linear regression model*. Much like correlation (see Section 1.3.2), linear regression attempts to quantify the covariability between two or more dimensions by calculating a mathematical equation for what is known as the *line of best fit*. in addition to quantifying covariability, this equation also allows us to, for example, predict values of Moose from the information contained in the other available dimensions:

$$\text{Moose} = b_0 + b_1\text{Wolves} + b_2\text{KillRate} + b_3\text{PredationRate} + b_d\text{MooseRecruitmentRate}$$

The most frequently used technique for determining this line of best fit is to find the set of equation coefficients that minimizes the sum of squared *residuals*, the vertical distances between the data points and the fitted line.

a. Construct this data matrix in R with the proper row and column names
b. Use `function()` to create a function that takes, as a single concatenated argument, values for the intercept and four regression coefficients for the linear regression model above and returns corresponding the sum of squared residuals
c. Find the residual sum of squares for a linear model with the following intercept and regression coefficients: 200, -10, -3, -5, and 20
d. Find the residual sum of squares for a linear model with the following intercept and regression coefficients: 200, 10, 3, 5, and 20
e. Explain which of the two equations above provided a better fit to the Isle Royale data
f. Use particle swarm optimization via `psoptim()` to find the best set of intercept and regression coefficients to predict the number of moose
g. Explain if this best-fit model indicates that the proportion of moose being killed by wolves (`Predation Rate`) has a positive or negative covariability with the number of `Moose`