

A *loop* is a sequence of instructions, which are required to be executed more than once on purpose. They are initiated by loop statements (`for` or `while`) and terminated by termination statements (simply `}` or sometimes `break`). Different kinds of loops can be found in almost all practical programs. In this chapter, we consider writing loops and using them for solving various problems. In addition to correct versions, we focus on possible mistakes when writing and implementing loops. Nested loops are also considered for practical purposes, such as matrix–vector multiplications. Finally, we study the iteration concept, which is based on using loops for achieving a convergence.

---

## 2.1 Loop Concept

We first consider simple examples involving basic problems and their solutions using loops.

### 2.1.1 Example: 1-Norm with For Statement

Consider the calculation of the 1-norm of a given vector  $v \in \mathbb{R}^n$ , i.e.,

$$\|v\|_1 = \sum_{i=1}^n |v[i]|.$$

The vector has  $n$  elements. The most trivial algorithm to compute the 1-norm can be described as follows:

- Initialize a sum value as zero.
- Add the absolute value of the first element to the sum value.
- Add the absolute value of the second element (if  $n > 2$ ) to the sum value.
- ...
- Add the absolute value of the last element to the sum value.
- Return the sum value.

Obviously, there is a repetition (adding the absolute value of an element), which can be expressed as a loop. The following R program can be written along this direction:

R Program: Calculation of 1-Norm Using For (Original)

```
01 onenorm_for = function(v){
02   sumvalue = 0
03   for (i in 1:length(v)){
04     sumvalue = sumvalue + abs(v[i])
05   }
06   return(sumvalue)
07 }
```

In this program, we are simply performing addition operations, which could be written as

```
sumvalue = 0 + abs(v[1]) + abs(v[2]) + abs(v[3]) + ...
```

where `abs` is a built-in function (command) in R. But, instead of writing all addition operations, we use a `for` loop. This is because of two major reasons:

- We would like to write a general program, where the input vector `v` may have different numbers of elements.
- Even if the input size is fixed, we are probably unable to write all summation operations one by one if the number of elements in `v` is large.

When the `for` loop is used above, the operations inside the loop, i.e.,

```
sumvalue = sumvalue + abs(v[i])
```

are repeated for  $n$  times. This is due to the expression

```
i in 1:length(v)
```

in the `for` statement, which indicates that the variable `i` will change from 1 to `length(v)`. Here, `length(v)` is an R command that gives the number of elements in `v`. The value of the 1-norm is stored in a scalar variable `sumvalue`, which is returned whenever the loop finishes. The line

```
sumvalue = 0
```

is required to make sure that this scalar is well defined before starting the loop.

At this stage, let's consider some modifications with possible mistakes. In the following program, the loop is constructed correctly, but `sumvalue` is not updated in accordance with the 1-norm.

R Program: Calculation of 1-Norm Using For (Incorrect)

```
01 onenorm_for = function(v){
02   sumvalue = 0
03   for (i in 1:length(v)){
04     sumvalue = sumvalue + abs(v[1])
05   }
06   return(sumvalue)
07 }
```

Specifically, instead of adding the absolute values of the elements in  $v$ , just the absolute value of the first element is added for  $n$  times. Hence, the result (output) is

$$\sum_{i=1}^n |v[1]| = n|v[1]|,$$

which is simply  $n$  times the absolute value of the first element, rather than the 1-norm of the vector.

An example to a correct but poor programming is as follows:

R Program: Calculation of 1-Norm Using For (Restricted)

```
01 onenorm_for = function(v){
02   sumvalue = 0
03   for (i in 1:10){
04     sumvalue = sumvalue + abs(v[i])
05   }
06   return(sumvalue)
07 }
```

In this case, the loop and update operations are written correctly, but the number of elements is fixed to 10. The programmer may be sure that the number of elements in input vectors to be considered and handled via this program is always 10. But, why not to make it more general without too much effort?

The following correct program is quite similar to the original one, but the number of elements is defined as a variable  $n$ :

R Program: Calculation of 1-Norm Using For (Correct)

```
01 onenorm_for = function(v){
02   sumvalue = 0
03   n = length(v)
04   for (i in 1:n){
05     sumvalue = sumvalue + abs(v[i])
06   }
07   return(sumvalue)
08 }
```

In some cases, adding some variables may lead to neater expressions. In the example above, the programmer may find

```
for (i in 1:n){
```

neater than the original expression

```
for (i in 1:length(v)){
```

In addition, in computer programs, it is common to use a variable more than once, and using an extra line `n = length(v)` may prevent repetitive call of the same function, i.e., `length` in this case.

The following is another correct version, where the variable `sumvalue` is initialized as the absolute value of the first element:

R Program: Calculation of 1-Norm Using For (Correct)

```
01 onenorm_for = function(v){
02   sumvalue = abs(v[1])
03   n = length(v)
04   for (i in 2:n){
05     sumvalue = sumvalue + abs(v[i])
06   }
07   return(sumvalue)
08 }
```

Note that the loop is constructed as

`i in 2:n`

instead of

`i in 1:n`

to avoid adding the first element twice. As opposed to the previous examples, this program assumes that the vector has at least two elements, i.e.,  $n > 1$ .

### 2.1.2 Example: 1-Norm with While Statement

Another program to calculate the 1-norm of a given vector is shown below. Compared to the previous programs, the `for` loop is replaced with a `while` loop. Even though a different program is implemented now, the underlying algorithm remains the same, i.e., the 1-norm of a vector is calculated by adding the absolute values of its elements one by one.

R Program: Calculation of 1-Norm Using While (Original)

```
01 onenorm_while = function(v){
02   sumvalue = 0
03   i = 1
04   while (i <= length(v)){
05     sumvalue = sumvalue + abs(v[i])
06     i = i + 1
07   }
08   return(sumvalue)
09 }
```

Note the following specific commands due to the structure of the `while` statement:

- The variable `i` is initialized as 1 before the loop.
- In addition to the update of the variable `sumvalue`, the variable `i` is incremented inside the loop as `i = i + 1`.

These are because the `while` statement indicates only a condition for stopping the loop whereas no information is provided for the initialization or incrementation, as opposed to the `for` statement, where all possible values of the variable `i` are clearly defined.

Again, let us consider some modifications with possible mistakes. In the following program, the incrementation `i = i + 1` is performed at an incorrect place:

R Program: Calculation of 1-Norm Using While (Incorrect)

```
01 onenorm_while = function(v) {  
02   sumvalue = 0  
03   i = 1  
04   while (i <= length(v)) {  
05     i = i + 1  
06     sumvalue = sumvalue + abs(v[i])  
07   }  
08   return(sumvalue)  
09 }
```

This means that the result (output) is

$$\|v\|_1 = \sum_{i=2}^{n+1} |v[i]|$$

instead of the 1-norm of the vector. This expression is mathematically invalid, whereas the program is not expected to give the correct answer (1-norm of the vector). On the other hand, the behavior of the program is actually unpredictable since the program tries to access to the  $(n + 1)$ th element of a vector of  $n$  elements. In our case (using R), this probably leads to a not-a-number (NaN) result, but in practice, it is possible that a junk number in memory is extracted by coincidence leading to an incorrect result at the end.

Another incorrect program, where the incrementation of `i` is forgotten, is as follows:

R Program: Calculation of 1-Norm Using While (Incorrect)

```
01 onenorm_while = function(v) {  
02   sumvalue = 0  
03   i = 1  
04   while (i <= length(v)) {  
05     sumvalue = sumvalue + abs(v[i])  
06   }  
07   return(sumvalue)  
08 }
```

This simple mistake leads to the famous *infinite loop*. Since `i` is not incremented, the condition in the `while` statement is always satisfied. Hence, the program continues infinitely (at least in theory!), adding the absolute value of the first element repetitively. This is a very serious problem.

Consider now the following example, where the initialization of `i` is forgotten:

R Program: Calculation of 1-Norm Using While (Incorrect)

```
01 onenorm_while = function(v) {  
02   sumvalue = 0  
03   while (i <= length(v)) {  
04     sumvalue = sumvalue + abs(v[i])  
05     i = i + 1  
06   }  
07   return(sumvalue)  
08 }
```

This is again a case where the behavior of the program is unpredictable. The variable `i` is simply undefined before the `while` statement; hence, we probably get an error indicating that this variable is not found. But, more dangerously, it is possible that `i` is actually defined (probably incorrectly) in the R workspace before this program is used. In such a case, one may expect that the program gives an incorrect result or a not-a-number (NaN).

A common mistake in loops is mixing `for` and `while` statements, such as the loop in the following incorrect program.

R Program: Calculation of 1-Norm Using For (Incorrect)

```
01 onenorm_for = function(v) {  
02   sumvalue = 0  
03   i = 1  
04   for (i in 1:length(v)) {  
05     sumvalue = sumvalue + abs(v[i])  
06     i = i + 1  
07   }  
08   return(sumvalue)  
09 }
```

There are two mistakes in this program. The harmless one is the initialization `i = 1`, which is actually not required since a `for` loop is used and this statement already defines the initial value of `i`. However, the second mistake, i.e.,

$$i = i + 1$$

inside the loop, is very dangerous. This is because the loop variable `i` that should be controlled by the `for` statement is modified inside the loop. Luckily, R can handle this by omitting the update inside the loop. But, using some other languages, such a mistake may lead to an erratic behavior that is difficult to control. In general, loop variables should not be modified or used for other purposes, except proper increase or decrease commands in `while` loops.

Finally, the following is a nice and correct variation, where the vector elements are accessed in a reversed order:

R Program: Calculation of l-Norm Using While (Correct)

```
01 onenorm_while = function(v){
02   sumvalue = 0
03   i = length(v)
04   while (i >= 1){
05     sumvalue = sumvalue + abs(v[i])
06     i = i - 1
07   }
08   return(sumvalue)
09 }
```

Note how `i` is initialized and updated inside the loop, whereas the condition of the `while` statement is constructed accordingly.

### 2.1.3 Example: Finding the First Zero

Lets assume that we would like to find the location of the first zero element of a vector  $v \in \mathbb{R}^n$ . First, consider the following program using a `for` statement:

R Program: Finding the First Zero Using For (Original)

```
01 findzero_for = function(v){
02   for (i in 1:length(v)){
03     if (v[i] == 0){
04       return(i)
05     }
06   }
07 }
```

Similar to the previous examples, the elements of the vector are accessed from 1 to  $n$ . But, interestingly, the `return` statement is placed inside the loop. This is because whenever we find a zero element, we would like to stop (there is no need to go on) and return the index of this element. Note that this condition is checked by the `if` statement as

```
if (v[i] == 0){
```

while the variable `i` is changed from 1 to  $n$ .

The program above does not return anything if there is no any zero in the vector being considered. Even though printing noting would be a good indication for the absence of a zero, one may desire a kind of warning message to be printed in this special case. In fact, it is quite easy to do this as follows:

R Program: Finding the First Zero Using For (Correct)

```
01 findzero_for = function(v){
02   for (i in 1:length(v)){
03     if (abs(v[i]) == 0){
04       return(i)
05     }
06   }
07   return("Vector does not contain zero element!")
08 }
```

We only added a single line

```
return("Vector does not contain zero element!")
```

just after the end of the loop without any extra condition. This is sufficient because we know that, if there is a zero, the program returns its index and stops immediately at line 04. Hence, line 07 is never executed if there is a zero in the vector. Otherwise (if there is no zero in the vector), the loop ends without any return operation, and line 07 is executed next to print out the desired warning.

The algorithm for finding the first zero can also be implemented using a while statement. Consider the following:

R Program: Finding the First Zero Using While (Incorrect)

```
01 findzero_while = function(v){
02   i = 1
03   while (v[i] != 0){
04     i = i + 1
05   }
06   return(i)
07 }
08 }
```

In this program, we start by setting the variable `i` to 1. Then, it is incremented as

```
i = i + 1
```

while the element being considered, i.e., `v[i]`, is not zero. This also means that the loop stops (hence, `i` is not incremented any further) whenever the element is zero and the condition

```
v[i] != 0
```

is not satisfied. The final value of `i` is returned as the index of the first zero element.

The program above looks good, but unfortunately it suffers from a serious problem. When there is no zero element in the input vector `v`, the loop tries to continue even after the last element is checked. Then, the loop attempts to access to the  $(n + 1)$ th element of the vector, which leads to an error. This is quite different from printing nothing, and the program above can be considered as incorrect.



As a remedy, one can insert an additional condition to stop the `while` loop when all elements are considered but no zero is found. In other words, the loop variable `i` should not be allowed to become larger than `length(v)` whether the vector contains zero or not. Consider the following updated program:

R Program: Finding the First Zero Using While (Incorrect)

```
01 findzero_while = function(v){  
02   i = 1  
03   while (v[i] != 0 && i <= length(v)){  
04     i = i + 1  
06   }  
07   return(i)  
08 }
```

Note that the combined expression

$$v[i] \neq 0 \ \&\& \ i \leq \text{length}(v)$$

means that both two conditions, i.e., `v[i] != 0` and `i <= length(v)`, need to be satisfied in order to `while` loop continues. This program is much better than the previous one since the additional condition in the `while` statement, i.e.,

$$i \leq \text{length}(v)$$

stops the loop whenever the value of `i` exceeds the length of `v`. Unfortunately, even though it does not give any run-time error, this program is also incorrect. A problem occurs again in the special case, i.e., where there is no zero. Specifically, if there is no zero in the vector `v`, the value of  $(n + 1)$  is returned incorrectly as the index of the first zero element. In fact, the program should return nothing or print a warning message to indicate that no zero is found. Hence, we need to add a conditional statement as follows:

R Program: Finding the First Zero Using While (Correct)

```
01 findzero_while = function(v){  
02   i = 1  
03   while (v[i] != 0 && i <= length(v)){  
04     i = i + 1  
05   }  
06   if (i <= length(v)){  
07     return(i)  
08   }  
09   else{  
10     return("Vector does not contain zero element!")  
11   }  
12 }
```

The final program above is correct, but it looks more complicated than the corresponding program (including the warning message) using a `for` loop. In many cases, depending on the problem and algorithm, using `for` or `while` might be

easier than the other, even though the resulting programs have almost the same efficiency.

### 2.1.4 Example: Infinity Norm

Consider the calculation of the  $\infty$ -norm of a given vector  $v \in \mathbb{R}^n$ , i.e.,

$$\|v\|_{\infty} = \max_{1 \leq i \leq n} |v[i]|.$$

As the formula states, the  $\infty$ -form of a vector is the maximum of the absolute values of its elements. The following program, which checks the absolute values of all elements one by one using a `for` loop, is suitable for finding the  $\infty$ -norm:

R Program: Calculation of Infinity-Norm (Original)

```
01 infinitynorm = function(v) {  
02   maxvalue = 0  
03   for (i in 1:length(v)) {  
04     if (abs(v[i]) > maxvalue) {  
05       maxvalue = abs(v[i])  
06     }  
07   }  
08   return(maxvalue)  
09 }
```

In this program, the elements of the vector `v` is considered from 1 to  $n$ . Inside the loop, there is an `if` statement to compare the absolute value of the element being considered with the variable `maxvalue`. At any instance, this variable, i.e., `maxvalue`, stores the largest absolute value of the elements that have been considered so far. Then, if the absolute value of the element being considered is larger than `maxvalue`, this variable should be updated as

`maxvalue = abs(v[i])`

accordingly. A program without a conditional statement, such as the following one, would be incorrect:

R Program: Calculation of Infinity-Norm (Incorrect)

```
01 infinitynorm = function(v) {  
02   maxvalue = 0  
03   for (i in 1:length(v)) {  
04     maxvalue = abs(v[i])  
05   }  
06   return(maxvalue)  
07 }
```

The program above returns nothing but the absolute value of the last element of the input vector `v`.

We have seen different programs to calculate two different norms of a given vector. At this stage, the following question may arise: Is there any better way to calculate these norms instead of writing these programs? In fact, the answer is yes. For example, consider the following command for the  $\infty$ -norm:

```
max(abs(v))
```

It is just a single line, and there is even no need to put this command in a function format. Alternative, if  $v$  is correctly defined as a column vector, using

```
norm(v, "I")
```

also works. These examples show that, before attempting to write any program, it is usually better to check whether the programming language (which is R in our case) already provides the desired function or not. For example, using R, there is a `norm` function, which can be used as above, not only for the  $\infty$ -norm, but also for some other norms in mathematics. In addition to saving time for programming, these built-in functions (programmed by the language developers) are generally more efficient (e.g., faster) than those written by users. Of course, no language can provide all functions required. Hence, in real life, computer programs often involve multiple contributions, where user functions and built-in functions are used together appropriately.

---

## 2.2 Nested Loops

There is no limitation in putting a loop inside another loop. In such a nested structure, however, the loop variables should be used very carefully, and they should not be mixed. In addition, one should keep in mind that nested loops can be computationally expensive, while they may be implemented if no alternative exists.

### 2.2.1 Example: Matrix–Vector Multiplication

Consider the multiplication of a matrix  $A \in \mathbb{R}^{m \times n}$  with a vector  $x \in \mathbb{R}^n$ . If  $y = Ax$ , we have

$$y[i] = \sum_{j=1}^n A[i, j]x[j]$$

for  $i = 1, 2, \dots, m$ . Hence, a code segment to obtain an element of  $y$  can be

```
sumvalue = 0
n = ncol(A)
for (j in 1:n){
  sumvalue = sumvalue + A[i,j]*x[j]
}
y[i] = sumvalue
```

In this code, the command `n = ncol(A)` gives the number of columns in the input matrix  $A$ . We set the value of the variable `sumvalue` to 0 and update it inside the loop by adding the multiplication of a matrix element with the corresponding

element of the input vector  $x$ . When the loop finishes, the final value of `sumvalue` is stored in  $y$ . Note that the variable  $i$ , which corresponds to the index of the output vector, is assumed to be constant at this stage.

The code segment above should be repeated for all elements of the output vector  $y$ , i.e., for different values of  $i$ . Therefore, we need this loop to be placed inside another loop as shown in the following program:

**R Program: Matrix-Vector Multiplication (Original)**

```
01 matvecmult = function(A,x){
02   m = nrow(A)
03   n = ncol(A)
04   y = matrix(0,nrow=m)
05   for (i in 1:m){
06     sumvalue = 0
07     for (j in 1:n){
08       sumvalue = sumvalue + A[i,j]*x[j]
09     }
10     y[i] = sumvalue
11   }
12   return(y)
13 }
```

In this program, the command `nrow(A)` gives the number of rows in the input matrix  $A$ . This value is stored in the variable  $m$ , similar to the number of columns that is stored in  $n$ . Note that different variables, i.e.,  $i$  and  $j$ , are used for the outer and inner loops, respectively.

In the program above, the variable `sumvalue` is reinitialized as zero before each inner loop. Having said this, the following program is incorrect:

**R Program: Matrix-Vector Multiplication (Incorrect)**

```
01 matvecmult = function(A,x){
02   m = nrow(A)
03   n = ncol(A)
04   y = matrix(0,nrow=m)
05   sumvalue = 0
06   for (i in 1:m){
07     for (j in 1:n){
08       sumvalue = sumvalue + A[i,j]*x[j]
09     }
10     y[i] = sumvalue
11   }
12   return(y)
13 }
```

Using this program, where `sumvalue` is initialized only once outside the loops, only the first element of  $y$  can be calculated correctly. Then, the variable `sumvalue` contains accumulated contributions from earlier calculations leading to incorrect values in the other elements, i.e., from  $y[2]$  to  $y[m]$ .

When there are nested loops, their order is always an issue to be considered by the programmer. In the original matrix–vector multiplication program, the outer and inner loops are constructed for the rows and columns of the matrix  $A$ , respectively. Specifically, the variable of the outer loop  $i$  represents the rows of  $A$ , whereas the variable of the inner loop  $j$  represents its columns. This is called a *rowwise* processing of the matrix, because the matrix is accessed row by row, e.g., first all elements in the first row are considered, second all elements in the second row are considered, etc. A *columnwise* processing is also possible, corresponding to a switch of the outer and inner loops in the program.

The nested loops are best switched when there is no operation between them (i.e., no operation inside the outer loop and outside the inner loop). In the original program, line 05, i.e.,

```
sumvalue = 0
```

is between two loops. Therefore, before attempting to write a matrix–vector multiplication with the matrix accessed columnwise, we can modify the original algorithm slightly by removing the variable `sumvalue`:

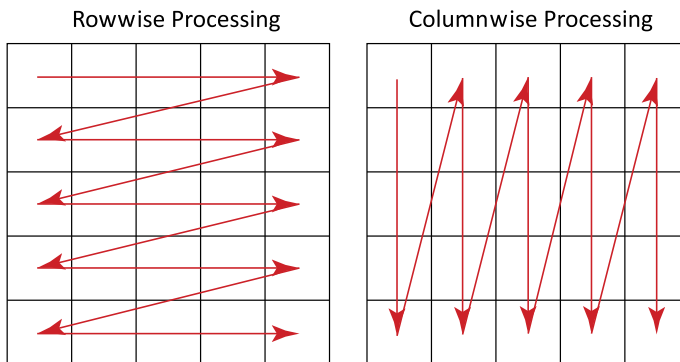
R Program: Matrix–Vector Multiplication (Correct)

```
01 matvecmult = function(A,x){  
02   m = nrow(A)  
03   n = ncol(A)  
04   y = matrix(0,nrow=m)  
05   for (i in 1:m){  
06     for (j in 1:n){  
07       y[i] = y[i] + A[i,j]*x[j]  
08     }  
09   }  
10   return(y)  
11 }
```

This program works correctly since the output vector  $y$  is initialized as zero in line 04. Moreover, it is now convenient to switch the loops to obtain a columnwise processing of the input matrix as follows:

R Program: Matrix–Vector Multiplication (Correct)

```
01 matvecmult = function(A,x){  
02   m = nrow(A)  
03   n = ncol(A)  
04   y = matrix(0,nrow=m)  
05   for (j in 1:n){  
06     for (i in 1:m){  
07       y[i] = y[i] + A[i,j]*x[j]  
08     }  
09   }  
10   return(y)  
11 }
```



**Fig. 2.1** Rowwise and columnwise processing of a matrix

Note how the elements of the matrix  $A$  are now used columnwise. As an example, Fig. 2.1 illustrates rowwise and columnwise processing of a  $5 \times 5$  matrix.

For the matrix–vector multiplication programs demonstrated above, one should also note how the elements of the input and output vectors are used. In the rowwise processing, the input vector  $x$  is traced repetitively, whereas the output vector  $y$  is traced only once. This is reversed in the columnwise partitioning, where the input vector  $x$  is traced once, while the output vector  $y$  is traced repetitively.

### 2.2.2 Example: Closest-Pair Problem

Consider the following problem. Given  $n$  points in the two-dimensional space, i.e.,  $(x_k, y_k)$  for  $k = 1, 2, \dots, n$ , find the two closest points. As a *brute-force* approach, where all possible solutions are considered, we can compute the distance between each pair. Then, the minimum of these distances can be selected. We can follow this approach, but instead of storing the distance values between all pairs, we may compute them on-the-fly and compare with a variable `minimumdistance`, which is simply the minimum distance encountered so far. After considering all possible pairs, this variable and the corresponding index information can be returned as the outputs. Along this direction, the following program can be written:

## R Program: Finding the Closest Pair (Original)

```

01 findclosest = function(x,y){
02   n = length(x)
03   minimumdistance = sqrt((x[1]-x[2])^2+(y[1]-y[2])^2)
04   ibackup = 1
05   jbackup = 2
06   for (i in 1:(n-1)){
07     for (j in (i+1):n){
08       distance = sqrt((x[i]-x[j])^2+(y[i]-y[j])^2)
09       if (distance < minimumdistance){
10         minimumdistance = distance
11         ibackup = i
12         jbackup = j
13       }
14     }
15   }
16   list(minimumdistance,ibackup,jbackup)
17 }

```

The inputs of this program are vectors  $x$  and  $y$  that store the  $x$  and  $y$  coordinates of the given points, respectively. Both vectors have  $n$  elements, where  $n$  is stored in a variable  $n$ . Initially, the variable `minimumdistance` is set to the distance between the first and second points as

$$\text{minimumdistance} = \sqrt{(x[1]-x[2])^2 + (y[1]-y[2])^2}$$

To keep the track of the pair with the minimum distance, we also use the variables `ibackup` and `jbackup`, which are initially set to 1 and 2, respectively. After these initializations, we have two `for` loops to select different points and to compute the distances between them. In the outer loop, the variable  $i$  changes from 1 to  $n - 1$ . In the inner loop, the variable  $j$  changes from the value of  $i+1$  to  $n$ . This way, all possible pairs are considered without any duplication as the value of  $i$  is always smaller than the value of  $j$ .

Inside the loops, the distance between the  $i$ th and  $j$ th points is calculated as

$$\text{distance} = \sqrt{(x[i]-x[j])^2 + (y[i]-y[j])^2}$$

This value is then compared with the variable `minimumdistance`, which stores the minimum distance up to that point. If `distance` is smaller than `minimumdistance`, then `minimumdistance` should be updated accordingly, as well as the indices, i.e.,

```

minimumdistance = distance
ibackup = i
jbackup = j

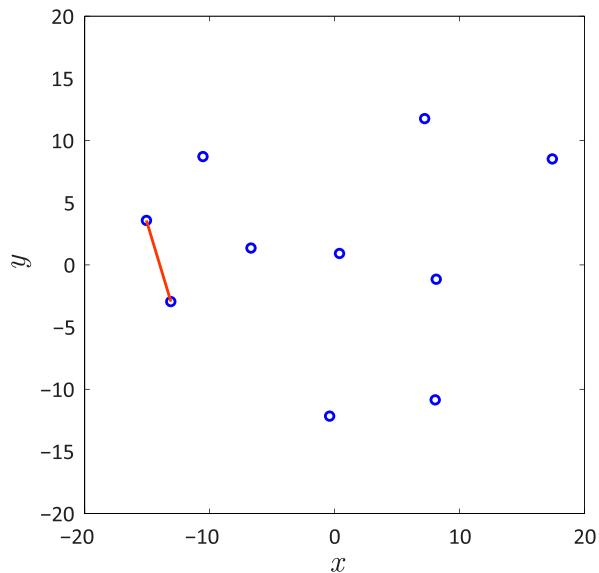
```

Finally, note that, instead of a `return` statement, we use

```
list(ibackup,jbackup,minimumdistance)
```

at the end of the program to print out the minimum distance and the indices of the corresponding points.

**Fig. 2.2** The closest pair among 10 points



As an example, Fig. 2.2 depicts 10 points on the  $x$ - $y$  plane, and the closest pair found by using the program above.

## 2.3 Iteration Concept

In a broad sense, an *iterative procedure* is a process of repeating a set of instructions to approach a target. Each repetition is called an *iteration*, and the output of an iteration is the input of the next iteration. Hence, each iteration depends on all previous iterations. The aim in performing iterations is to *converge* to a steady state, but divergence is not uncommon in many iterative solutions.

### 2.3.1 Example: Number of Terms for $e$

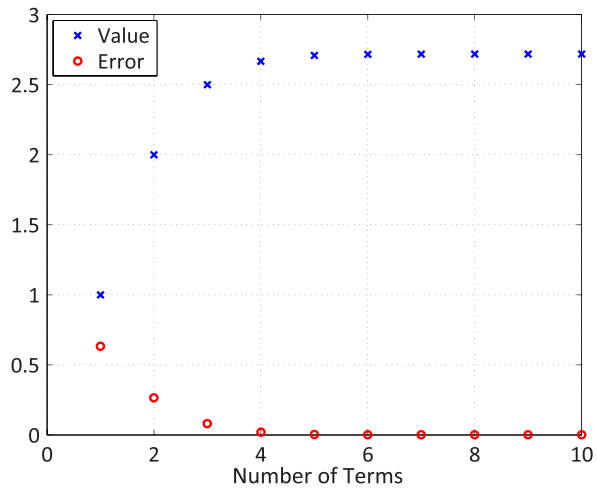
Assume that we would like to find the number of terms in the expression

$$e = \sum_{i=0}^{\infty} \frac{1}{i!} \approx \sum_{i=0}^n \frac{1}{i!}$$

to approximate the value of  $e$  with a given error threshold. The following iterative program can be used for this purpose:



**Fig. 2.3** Convergence of the series to the value of  $e$



R Program: Finding the Number of Terms for  $e$  (Original)

```
01 numeroftermsfore = function(desirederror){
02   refvalue = exp(1)
03   term = 1
04   sumvalue = 1 / factorial(term-1)
05   while (abs((refvalue-sumvalue)/refvalue) > desirederror){
06     term = term + 1
07     sumvalue = sumvalue + 1/factorial(term-1)
08   }
09   return(term)
10 }
```

In this program, the variable `term` represents the number of terms used in the series. After this variable is incremented inside the `while` loop, a new term is added into the series as

$$\text{sumvalue} = \text{sumvalue} + 1/\text{factorial}(\text{term}-1)$$

where `factorial` is the built-in R function for the factorial. Hence, the variable `sumvalue` is updated in each repetition, and the loop continues while the relative error is larger than the desired value represented by the scalar input `desirederror`. This comparison can be seen in the `while` statement as

```
while (abs((refvalue-sumvalue)/refvalue) > desirederror){
```

where `abs((refvalue-sumvalue)/refvalue)` is the relative error. Note that the reference value is obtained by using the built-in function of R, i.e., `exp(1)`.

Figure 2.3 depicts how the variable `sumvalue` approaches  $e$ , and the error is reduced to zero as the number of terms increases. In other words, `sumvalue` converges to  $e$ , whereas the error converges to zero.

### 2.3.2 Example: Geometric Series

Lets consider another iterative procedure, where the number of terms in the infinite geometric series

$$\frac{1}{1-x} = \sum_{i=0}^{\infty} x^i \approx \sum_{i=0}^n x^i, \quad |x| < 1,$$

is to be found again for a given error criteria. The following program can be used:

R Program: Finding the Number of Terms in the Geometric Series (Original)

```
01 numberoftermsingeo = function(x,desirederror){
02   refvalue = 1/(1-x)
03   term = 1
04   sumvalue = x^(term-1)
05   while (abs((refvalue-sumvalue)/refvalue) > desirederror){
06     term = term + 1
07     sumvalue = sumvalue + x^(term-1)
08   }
09   return(term)
10 }
```

In this case, the program has two inputs, i.e.,  $x$  and `desirederror`. This program works fine when the variable  $x$ , corresponding to the value of  $x$  in the formula above, fits into the definition of the geometric series. In other words, a convergence is achieved if the absolute value of the input  $x$  is smaller than 1. Otherwise, no convergence occurs, since the geometric series becomes mathematically invalid.

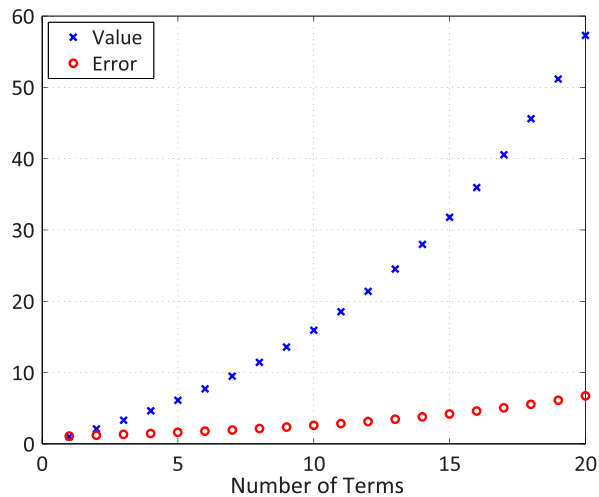
As an example, Fig. 2.4 depicts the variable `sumvalue` and the corresponding error with respect to the number of terms when the program is used for  $x$  equal to 1.01. The value of `sumvalue` does not converge to any value, whereas the error increases unboundedly as the iterations go on. Hence, in this example, convergence is not achieved, and iterations *diverge*. Note that, for those faulty values of  $x$ , the algorithm above never stops (infinite loop occurs), which may be considered as a poor programming.

### 2.3.3 Example: Babylonian Method

Let us write an iterative program using the Babylonian method, i.e.,

$$x_{n+1} = 0.5(x_n + 5/x_n),$$

to approximate the square root of 5 with 0.001 error. We can start with  $x_0 = 2$  and assume that the “exact” value of  $\sqrt{5}$  is not available. Hence, we stop iterations when the values in two consecutive iterations are sufficiently close to each other, i.e.,  $|x_{n+1} - x_n| < 0.001$ . The proposed algorithm can be implemented as follows.

**Fig. 2.4** Divergence of the geometric series for  $x = 1.01$ 

R Program: Babylonian Method for Square-Root of 5 (Original)

```

01 babylonianforsqrtfive = function(){
02   xold = 2
03   xnew = 0.5*(xold + 5/xold)
04   while (abs(xnew-xold) > 0.001){
05     print(xold)
06     xold = xnew
07     xnew = 0.5*(xold + 5/xold)
08   }
09   return(xnew)
10 }

```

This is a quite special program for a specific purpose; there is no input, but the output is the approximate value of  $\sqrt{5}$ . In addition, the history of iterations is printed out by using `print(xold)` in line 05. There are two variables to keep the values of  $x$ . These are the old value `xold` and the new value `xnew`. The variable `xold` is initially set to 2, whereas the variable `xnew` is calculated by using the formula above. The iterative process is constructed by using a `while` statement, which compares the absolute difference of `xold` and `xnew` with the target error 0.001. In the loop, `xold` is updated by simply copying `xnew`, whereas `xnew` is recalculated using the formula again. Note that the order of these updates (first `xold` using `xnew`, then `xnew` using the new value) is important.

If the program above is implemented and used, we get the steps of the iterative procedure in the R workspace as

```

2
2.25
2.236068

```

where the final value 2.236068 is the required approximation of  $\sqrt{5}$ . In other words, the approximate value of  $\sqrt{5}$  converges to 2.236068.

---

## 2.4 Conclusions

Loops are among the basics of computer programming, in which instructions are often need to be repeated. For example, iterative procedures where iterations are carried out to achieve convergence can easily be implemented using loops. All programming languages, including R, provide special statements to construct loops. Two types of loop statements are common:

- `for`-type statements, where the possible values of the loop variable are clearly defined.
- `while`-type statements, where the continuation criteria are clearly defined, but the loop variable needs to be initialized and incremented manually.

Depending on the problem and the solution algorithm, one of the types may be easier to use than the other.

Loops are very beneficial, but they can easily be written incorrectly. Programmers need to check how loops behave under different circumstances, especially to avoid infinite loops, whereas extra conditions may be required to control special cases.

---

## 2.5 Exercises

1. Write a program using a `for` statement to compute the 1-norm of a given vector  $v \in \mathbb{R}^n$ . Apply the program to an example vector as

```
onenorm_for(matrix(c(4,5,4,3,-1,3,4,5,-4,2),ncol=1))
```

2. Consider the original program for the geometric series. How the program can be changed in order to avoid infinite loop for faulty values of  $x$ ?

3. Write a program to calculate the 2-norm of a given vector  $v \in \mathbb{R}^n$ , i.e.,

$$\|v\|_2 = \sqrt{\sum_{i=1}^n (v[i])^2},$$

using a `for` or `while` loop. Apply it to an example vector as

```
twonorm(matrix(c(5,4,1,6,7,8,-4,15,-2,4),ncol=1))
```

Compare your result with the value given by the built-in function of R, i.e.,

```
norm(matrix(c(5,4,1,6,7,8,-4,15,-2,4),ncol=1),"E")
```

4. Write a program that calculates the sum of cubes of positive integers from 1 to  $n$  for a given value of  $n$ , i.e.,

$$\sum_{i=1}^n i^3.$$

Check your code against the direct formula

$$\sum_{i=1}^n i^3 = \left( \frac{n(n+1)}{2} \right)^2$$

for different values of  $n$ , such as  $n = 3$ ,  $n = 30$ , and  $n = 300$ .

5. Write an R program that counts the number of zeros of a given vector  $v \in \mathbb{R}^n$ . Apply the program to an example vector as

```
countzeros(matrix(c(4,0,3,0,0,3,-4,0,5,0),ncol=1))
```

6. Write a program that finds the smallest element of a given vector  $v \in \mathbb{R}^n$ . Apply the program to three different vectors as

```
findminimum(matrix(c(4,0,3,0,0,3,-4,0,5,0),ncol=1))
findminimum(matrix(c(4,2,3,5,6,3,4,1,5,2),ncol=1))
findminimum(matrix(c(-4,-2,-3,-5,-6,-3,-4,-1,-5,-2),ncol=1))
```

Check that your program works correctly with  $-4$ ,  $1$ , and  $-6$  outputs, respectively.

7. Write a program that finds the two farthest points among  $n$  points in the two-dimensional space, i.e.,  $(x_k, y_k)$  for  $k = 1, 2, \dots, n$ . Apply the program to an example problem as

```
x = matrix(c(1,4,3,-2,-3),ncol=1)
y = matrix(c(2,-2,2,2,-1),ncol=1)
findfarthest(x,y)
```

8. Write a program that calculates the sine function using its Taylor-series expansion, i.e.,

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots = \sum_{i=0}^{\infty} \frac{(-1)^i x^{2i+1}}{(2i+1)!} \approx \sum_{i=0}^n \frac{(-1)^i x^{2i+1}}{(2i+1)!}.$$

The input of the program should be the value of  $x$  in terms of radians and the number of added terms  $n$ . The output should be the approximate value of  $\sin x$ . Test your code for  $x = \pi/3$  and  $n = 1, 2, 3, 4, \dots$ . How many terms are required for six digits of accuracy? Perform similar tests for  $x = 4\pi/3$  and  $x = 7\pi/2$ . Compare your results for different values of  $x$ .



<http://www.springer.com/978-1-4471-5327-6>

Guide to Programming and Algorithms Using R

Ergül, Ö.

2013, XI, 182 p., Hardcover

ISBN: 978-1-4471-5327-6