Chapter 17
Bonus Chapter: Modern Fortran

Men may the olde atrenne, and noght atrede.

Men may the old outrun, but not outwit.

Geoffrey Chaucer

The Fortran programming language (for *for*mula *tran*slation) is often seen as a creature from an ancient world, when time was still young. In fact, it is one of the oldest programming languages still in use. It was originally developed by a team led by John Backus at IBM for scientific and engineering applications. It first appeared in 1957 and soon went on to dominate scientific computing. In this sense, Fortran was the beginning of modern data processing as we practice it today.

Over the years, Fortran evolved into a widely used language for high-performance computing on the world's fastest supercomputers, and even today modern Fortran is the leading language in computationally intensive fields such as fluid dynamics, geophysics, numerical weather prediction, and particle physics simulation. This means that it is a niche language of great importance to a relatively small number of people, i.e. scientists and engineers in selected disciplines.

In this chapter, we provide an introduction to modern Fortran and relate it to the languages discussed previously. For background information and further reading, the book [2] can be recommended. A comprehensive guide is presented in [3]. Very useful in preparing this chapter was also [1]. Recommended internet sites include `fortran-lang.org` and `tutorialspoint.com/fortran`.

Like C/C++, Fortran programs are compiled. Fortran compilers are now available on all machine sizes, from small laptop computers to huge multiprocessors. In the following we rely on the open source compiler GNU Fortran.

Installation

Detailed installation instructions for Windows, Linux and MacOS can be found in [2]. Specifically for macOS, which we will use in the following, the homebrew package manager on `brew.sh` is recommended. Once homebrew is set up, the Fortran compiler can be installed as part of the GNU Compiler Collection, using the Terminal command

```
$ brew install gcc
```

All programs in this chapter were tested using the GNU Fortran compiler 12.2.0. The presentation is based on the current release Fortran 2018.

17.1 Basics

Example 17.1 As a first example, here is the notorious "Hello World" greeting:

```
program hello
print *, 'Hello World!' ! My first Fortran program
end program
```

All Fortran programs begin with the keyword `program` followed by the name of the program. Comments start with an exclamation mark. All characters after it until the end of the line are ignored by the compiler. The `print` command outputs the string after the comma on the screen. The asterisk means that we use the default formatting. Strings in Fortran are enclosed in single or double quotation marks. The program is terminated with the keywords `end program`.

For simplicity, we always assume in this chapter that the programs are stored in a file `test.f90`. The extension '`f90`' refers to an earlier version Fortran 90, but is still the standard convention. Similar to using the gcc compiler in C, the program can be compiled to an executable using the Terminal command

```
$ gfortran test.f90 -o test
```

and can then be run with

```
$ ./test
```

Note that, for historical reasons, Fortran is *case-insensitive*, except for string literals. The `print` comand could for example just as well be written as `PrInT`.

Arithmetical Expressions

The most important numeric data types in Fortran are integers and reals. They can be of different *kinds*, i.e. sizes. For example, to calculate 2^{31}, we must use the 8-byte kind of integers: `2_8**31` gives the correct answer, while `2**31` (short for `2_4**31`) exceeds the range. Similarly, for the calculation of 2^{63}, we need the 16-bit kind `2_16**63` to get the correct answer.

The mathematical operators `+`, `-`, `*`, `/`, `**` behave as usual. Just note that for integers, '`/`' means integer division. The remainder in integer division is calculated with the built-in (in Fortran called *intrinsic*) function `mod`. Example:

```
print *, 5/2, mod(5,2)  ! Out: 2  1
```

By the way: Note that multiple comma-separated items my be passed to the `print` command on the same line.

As usual, mixed expressions between real and integer numbers convert the integers to real numbers. In operations with mixed kinds, the larger kind dominates. For example, the result of multiplying two reals of kind 4 and 8 is of kind 8.

Boolean Operators, Comparisons

Comparison operators are written as in other languages, with one exception: "not equal" is denoted by '/='. The `logical` data type for representing truth values consists of the two constants `.true.` and `.false.`. Note that the dots belong to the designators. In the output, the logical constants are represented by the letters `T` and `F`, for example

```
print *, 1 /= 2  ! Out: T
```

The logical operators are denoted by `.and.`, `.or.`, `.not.`.

Variables and Assignments

Variables are declared in the form

```
integer :: idx, n
real :: pi = 3.14
logical :: answer = .false.
```

In line 1, the variables `idx` and `n` are declared to be of type `integer`. Note that it is possible to collect several declarations in a comma separated sequence in the same line. In line 2, the variable `pi` is declared to be of type `real` and initialized with the value 3.14. The meaning of line 3 should then be obvious.

Note that in lines 1 and 2 the variables are assumed to be of kind 4 by default. If others, 8 or 16, are desired, they must be explictly declared, e.g. `integer(kind=8)`.

Recall that Fortran is case-insensitive. This means for example, that after a declaration of `idx` as above, we can write `Idx` in the program code. In particular, it is not possible to distinguish between, say, a matrix '`A`' and a vector '`a`'.

An even more serious source of subtle errors is that a variable can be used without being declared. Any *undeclared* variable starting with a lettter i through n is automatically assumed to be of type `integer`, all others of type `real`. This was a resource-saving measure in times when each command had to written to a separate punch card.

Today this behavior can be switched off with the instruction '`implicit none`' or equivalently with the compiler option '`-fimplicit-none`'.

17.2 Control Structures: Branching, Loops

Example 17.2 (Rice and Chessboard Problem) We illustrate the main ideas along the program for calculating the reward for the chess inventor, already discussed in Example 2.4 on page 11 and Example 6.4 on page 123:

```
 1 program chessboard
 2   integer :: fieldno
 3   integer(kind=16) :: fieldval, sm = 0
 4   do fieldno = 1, 64
 5     if (fieldno == 1) then
 6       fieldval = 1
 7     else
 8       fieldval = 2 * fieldval
 9     end if
10     sm = sm + fieldval
11   end do
12   print *, sm
13 end program
```

In line 3 we declare the variables to have enough memory to store the required summands and the total sum of rice grains.

Line 4 starts a *counter loop*, comparable to a for loop in e.g. Python. Note that the indentation in lines 5 through 10 is only for better reading. In Fortran, program structure is determined by writing statements on separate lines and using key words. The loop ends in line 11.

Inside the loop, we find the standard form of an if statement in lines 5 to 9. Again, note that the separation of the statements into individual lines is important. Also note that the condition in line 5 must be enclosed in parentheses.

Remark The general form of a do loop can also include an optional third argument to specify an increment (or decrement) step, such that, for example

```
do i = 1, 10, 2
  print *, i
end do
```

generates the sequence 1, 3, 5, 7, 9.

In Fortran, *while loops* are formulated as special do loops:

Example 17.3 As an example, consider the Collatz problem, already discussed on several occasions, including Example 3.3 on page 28 and Example 6.2 on page 123:

```
1  program collatz
2    integer :: n = 100, nsteps = 0
3    do while (n > 1)
4      if (mod(n, 2) == 0) then
5        n = n/2
6      else
7        n = 3*n + 1
8      end if
9      nsteps = nsteps + 1
10   end do
11   print *, 'reached 1 in', nsteps, 'steps'
12 end program
```

As with if statements, the condition in the do-while heading in line 3 must be enclosed in parentheses.

The `exit` and `cycle` Statements

It is often necessary to interrupt the execution of a loop. The `exit` statement corresponds to the `break` statement in Python. It terminates the entire loop that contains it. The `cycle` statement corresponds to the `continue` statement. It is used to skip the rest of the code inside a loop for the current iteration only.

17.3 Functions

Fortran already provides a large collection of intrinsic built-in functions, for example the modulo function `mod` used in Example 1.3. In this section we show how to include self-written functions.

Example 17.4 As an introductory example, we generate the factorial function:

```
1 integer function fac(n); integer :: n
2   integer :: i
3   fac = 1
4   do i = 1, n
5     fac = i * fac
6   end do
7 end function
```

In line 1, we declare a function that returns a value of type `integer`, and specify that the input argument `n` also is of type `integer`. The output of the function is computed in lines 2 through 6. Note that the *name* `fac` of the function itself is used for the result variable. Line 7 is a required closing to the function declaration in line 1.

The function can then be tested in the following program `facTest`, which should be located in the same file as the function definition:

```
 8 program facTest
 9   integer :: fac
10   print *, fac(5)  ! Out: 120
11 end program
```

In line 9, note that we must tell the compiler that the result of `fac` is an integer.

Remark Note in line 1 that it is possible to write multiple statements on the same line if they are separated by a semicolon. We will often make use of this.

Recursive Functions

Example 17.5 We define a recursive variant of the factorial function:

```
1 recursive function fac(n) result(res)
2   integer :: n, res  ! type declaration for argument and result
3   res = 1
4   if (n == 1) return
5   res = n * fac(n - 1)
6 end function
```

In line 1 we declare `fac` as a recursive function. For recursive functions, the use of the function name as return variable is not allowed. Therefore, we must explicitly introduce a result variable, here `res`. Both argument and result variable are declared as `integer` in line 2.

The `return` statement in line 4 stops the computation when the base case is reached. Unlike in Python and C, for example, no value is returned here. What is returned as function result is the cumulated value in `res`.

Note the simple form of a conditional expression in line 4. It can be used when the block of statements to be executed consists of only a single statement.

Remark Note that also for non-recursive functions, the declaration of a specific result name can be used alternatively instead of the function name.

Subroutines

In Fortran a function without a return value is – mathematically correct – not called function. It is called *subroutine*.

Example 17.6 We define a subroutine to swap the values of two integer variables:

```
1 subroutine swap(x,y)
2   integer :: x, y, temp
3   temp = x; x = y; y = temp
4 end subroutine
```

We test it:

```
5 program testSwap
6   integer :: a = 1, b = 2
7   call swap(a,b)
8   print *, a, b  ! Out: 2 1
9 end program
```

In line 7 the subroutine is invoked with the `call` command. In function application such an explicit call mechanism is not necessary, since functions appear only in expressions and are called automatically when the expression is evaluated.

The `intent(in)` Attribute

There is a crucial point to note in Example 1.6. The reason subroutines work at all is that variables are passed by *reference*, not by *value*. The variables `x`, `y` in the `swap` definition are pure placeholders. When `swap` is called in line 7, they are associated with the actual argument variables `a`, `b`. What is then executed is actually

```
  temp = a; a = b; b = temp
```

In Fortran jargon, the place-holders are called *dummy* variables.

The same is true for functions. All arguments are passed by reference and can therefore be altered during execution. However, we can instruct the compiler to watch out for such potentially harmful side effects. For this purpose we add `intent(in)` to the variable declaration.

Example 17.7 Consider a simple square function:

```
1 integer function square(n)
2   integer :: n  ! if declared with attribute intent(in) ...
3   n = n**2      ! ... then cought by compiler
4   square = n
5 end function
```

In line 3 the dummy `n` is altered. In the main program we see the effect on the actual argument variable `m`:

```
6 program testSquare
7   integer square
8   integer :: m = 2
9   print *, square(m), m  ! Out:  4 4
10 end program
```

To catch such behavior we replace line 2 with

```
2 integer, intent(in) :: n
```

Here the keyword `intent(in)` instructs the compiler to abort with an error message if it detects an attempt to write to the variable `n`.

Remark It is a common pattern in Fortran to specify additional variable properties with such so-called *attributes*, which are written separated by commas after the type declaration. We will see more examples later.

Functions as Arguments

As in many other languages, functions can be passed as arguments to other functions.

Example 17.8 We define a second order function `integral` to approximate function integrals over the interval [0, 1] by the *midpoint rule*, already discussed in Sect. 12.3 and Example 13.1 on page 293. For background see page 278.

```
1  real function integral(fun)
2    interface
3      real function fun(x); real :: x; end function
4    end interface
5    integer, parameter :: n = 1000  ! constant
6    real, parameter :: h = 1.0/n    ! constant
7    integer :: i;
8    integral = 0.0
9    do i = 0, n-1
10     integral = integral + fun(h * (i + 0.5))
11   end do
12   integral = h * integral
13 end function
```

Line 1 declares `integral` to be a real valued function in one argument. As in our previous examples, the data type of the argument must be specified. Here we include the declaration in what is called an *explicit interface* block in lines 2 through 4. Line 3 states that `integral` expects real-valued functions in one real argument.

In lines 5 and 6, the variables `n` and `h` are declared with the attribute `parameter`, the Fortran term for constant. It tells the compiler that the initial value will not change during program execution. Here the declaration of `n` as constant is actually neccessary to be able to use it in the initialization in line 6.

We test the integral function for the evaluation of the function $f(x) := 4/(1+x^2)$. Recall that the exact solution is π.

```
14 program integralTest
15   real :: integral, res
16   res = integral(test_fun)  ! test_fun defined below in line 21
17   print *, res              ! Out: 3.14159322
18   contains
19   real function test_fun(x)
20     real :: x
21     test_fun = 4.0/(1 + x*x)
22   end function
23 end program
```

In line 16, `integral` is invoked for the test function defined in lines 19 to 22. Line 17 outputs the result 3.14159322.

The following part, lines 18 to 22, beginning with the `contains` statement, shows that and how a function can be defined within the main program.

Remark If we wish to separate the definition of the test function from the main program, we can outsource it to a so-called *module*:

```
module testFun
  contains
  lines 19 to 22 go here
end module
```

Then we need to include the contents of the module into the main program. We get it by inserting

```
use testFun
```

directly after the program declaration in line 14.

We will see more examples of module usage later.

17.4 Arrays and Vectors

Arrays can store a fixed-size sequential collection of elements of the same type. As in Julia, arrays are used for lists, vectors, and matrices. In the following, we will focus on one-dimensional vectors and here in particular on the list aspects. Mathematical properties are discussed later in connection with matrices.

Vectors are declared with the `dimension` attribute. For example, to declare vectors named `v` and `w` of integer numbers, each containing 5 elements, we can write:

```
1 integer, dimension(5) :: v     ! indices by default begin at 1
2 integer, dimension(0:4) :: w   ! here indices set from 0 to 4
3 integer :: v(5), w(0:4)        ! equivalent definition
```

In line 1 we declare a vector `v` of length 5, with indices running from 1 to 5, as e.g. in Julia. The vector `w` also has a length of 5, but here the elements are accessed with indices from 0 to 4, as in Python or C. Line 3 shows an equivalent, commonly used declaration method.

In both cases, we can find the length of the vectors with the intrinsic function `size` in line 4:

```
4 print *, size(v), size(w)      ! Out: 5 5
5 print *, lbound(w), ubound(w)  ! Out: 0 4
```

Line 5 illustrates that you can use the built-in `lbound` and `ubound` functions to determine the lower and upper bound of an array, respectively.

To access the individual elements, we use the index notation `v(i)` with rounded parentheses.

The assignment of values to arrays can then be done element by element, for example:

```
6 do i = 1, 5                   ! assign values to elements
7   v(i) = i**2
8 end do
```

However, it is usually more convenient to use *array constructors*, such as

```
v = [1, 4, 9, 16, 25]          ! array constructor
```

with the same effect.

A very versatile method is to use an *implied do loop*, like

```
v = [(i**2, i = 1, 5)]         ! implied do loop constructor
```

The expression in parentheses returns the sequence of squares i^2, where the index i ranges over $1, \ldots, 5$. The sequence is then enclosed in an array constructor `[ ]`.

All array elements can be assigned the same value, for example

```
w = 0                          ! assign same value to all elements
```

As before, it is also possible to combine declaration and initialization, for example:

```
integer :: v(5) = [(i**2, i = 1, 5)], w(5) = 0
```

Note, however, that the size of the vector must still be explicitly specified.

Sieve of Eratosthenes

As a first example of the use of vectors, we write a program to compute all prime numbers below a given number n using the sieve of Eratosthenes, discussed in Example 3.4 on page 31 and Example 8.16 on page 169.

Example 17.9 In the corresponding Python and Julia programs in Sects. 3.4 and 8.16 we used dynamically extendable lists. Here we follow the approach of successively changing the entries in an array of truth values.

```
1  program eratos
2    integer, parameter :: n = 10
3    logical :: primes(2 : n) = .true.  ! array index starts at 2
4    integer i, j
5    do i = 2, n
6      do j = 2, n/i                    ! integer division n/i
7        primes(i * j) = .false.        ! product cannot be prime
8      end do
9    end do
10   do i = 2, n
11     if (primes(i)) print *, i
12   end do
13 end program
```

In line 2 we declare the upper bound `n` as `parameter`, i.e. as a constant. This is neccessary so that we can use it to specify the upper bound of the array `primes` in line 3. We initialize all entries `primes(i)` as `.true.`, which means that until proved otherwise, we assume all numbers i, $2 \le i \le n$ to be prime.

Line 7 is central. It states that the product $i \cdot j$, corresponding to the entry with index `i*j` in `primes` cannot be prime.

When passing through the nested do loops, only those entries in primes retain their initial `.true.` value whose index cannot be represented as a product.

Remark Note the program is full of computational redundancies. For example, it would suffice to run index i to $\sqrt{n}$. Moreover, the inner loop can clearly be skipped for indices i where `primes(i)` has already received the value `.false.`.

However, the example is intended to illustrate the use of vectors and is not intended as an attempt at efficiency.

Array Sections

As in Matlab we can use the colon operator to select a part of a given array:

```
1 integer :: v(5) = [3,1,4,1,5]
2 print '(4i3)', v(2:5)  ! Out: 1 4 1 5
3 print '(3i3)', v( :3)  ! Out: 3 1 4
4 print '(2i3)', v(4: )  ! Out: 1 5
```

In line 2 the section of `v`-entries `v(2)` to `v(5)` is selected. In line 3 the first elements up to position 3 are selected, in line 4 analogously the last elements from index 4.

Remark Note here the special form of the format descriptor string here. To format an integer, we can use the letter i followed by the number of digits that will be reserved to display the value. The number will then be right-justified in this field. To format multiple values with the same format string, the string can be preceded by a number, for example `4i3` in line 2. The format specification is then enclosed in parentheses.

We can also write all format specifications in lines 3, 4, and 5 as `'(*(i3))'`, where the asterisk stands for "as many as needed". Note the extra pair of parentheses.

Filtering with the `pack` Function

Filtering with the intrinsic `pack` function is a versatile method for selecting elements from vectors. Example:

```
1 integer :: v(5) = [3,1,4,1,5]
2 print '(*(i3))', pack(v, mask = (v /= 1))  ! Out: 3 4 5
```

What happens, is that the expression `v /= 1` is evaluated elementwise and returns a logical array:

```
3 print *, v /= 1  ! Out:  T F T F T
```

which is then used as a mask by the `pack` function to collect those elements from `v` for which the corresponding entry in the logical array is true. Note that the `mask` keyword is not required, so that we would usually simply write `pack(v, v/=1)`.

Remark For the interested reader: On page 32 in the Python chapter we illustrated the concept of list comprehension with a simple example. Note that in Fortran we can use the `pack` function to emulate such behavior:

```
integer :: v(4) = [3,1,4,3]
print '(*(i3))', pack(v, v < 4)**2  ! Out:  9  1  9
```

Note that the quadratic operator `**2` is applied elementwise when applied to a vector.

Concatenation

Given vectors `v`, `w` of the same type, we can define the *concatenation* by enclosing the comma-separated vectors in square brackets, e.g. `[v,w]`. The concatenation can also contain scalar elements `a` of the same type, e.g. `[v,a,w]`. Example:

```
integer :: a = 3, v(2) = [1,2], w(3) = [2,1,0]
print '(*(i3))', [v,a,w]  ! Out:  1  2  3  2  1  0
```

The Quicksort Algorithm

As an application of the vector constructions, we write a program for sorting a list of numbers ‘a’ according to the quicksort algorithm.

Example 17.10 The program mimics the code in Example 3.13 on page 38 and Example 8.17 on page 170. Note that we always use the first array element as pivot.

The actual sorting function is again embedded in a module:

```
1  module quicksort
2    contains
3    recursive function qsort(a) result(b)
4      integer, dimension(:) :: a
5      integer :: b(size(a))
6      if (size(a) == 0) return
7      b = [ qsort( pack(a(2:), a(2:) < a(1)) ) ,   &
8            a(1),                                  &  ! pivot
9            qsort( pack(a(2:), a(2:) >= a(1)) ) ]
10   end function
11 end module
```

In line 4, the argument `a` is declared as a dummy variable for an integer vector. The expression ‘`(:)`’ means that the actual size is not specified. It is determined only when the function is called. In line 5 the result variable `b` is declared with the same size.

Lines 7 to 9 calculate the output for the current recursion level. In line 7, `pack(a(2:), a(2:)<a(1))` returns the array of all elements of `a` that are smaller than the pivot, except the pivot itself. Dually, `pack(a(2:), a(2:)>=a(1))` in line 9 returns the array with all elements greater than or equal to the pivot, again excluding the pivot itself. Both arrays are sorted by recursion and then concatenated with the pivot in the middle to get the output `b`.

The ‘&’ symbol in lines 7 and 8 is the Fortran line-continuation symbol. It tells the compiler that lines 7 – 9 should be considered as *one* line.

The program can then be tested:

```
12 program quicksortTest
13   use quicksort
14   integer :: a(6) = [4,5,7,3,8,3]
15   print '(*(i3))', qsort(a)  ! Out: 3  3  4  5  7  8
16 end program
```

The output in line 15 confirms that the `qsort` function works as desired.

Allocatable Arrays

The size of an array is usually specified at its declaration. However, arrays can also be declared as `allocatable`, which means that the size and storage space are determined and allocated at runtime.

Example 17.11 Suppose we wish to store the nonzero entries of an integer array ‘a’ in another array `arr`:

```
1 program allocExample
2   integer :: a(10) = [0,2,0,0,8,10,0,14,16,18]
3   integer, dimension(:), allocatable :: arr
4   integer :: m
5   m = size(pack(a, a /= 0))
6   allocate(arr(m))
7   arr = pack(a, a /= 0)
8   print '(*(i4))', arr  ! Out:  2   8  10  14  16  18
9 end program
```

In line 3 we declare `arr` to be an integer array of indeterminate size. The `allocatable` attribute means that no memory is reserved by the compiler, but is allocated at run time. In line 5, we determine the number of nonzero entries in `a`. Line 6 allocates memory space for an integer array of this size and associates this space with the variable `arr`. In line 7, `arr` is then filled with the nonzero entries in `a`.

17.5 Linear Algebra

Vectors

We have already seen how to define vectors as one-dimensional arrays. For mathematical use we mention that vectors can be multiplied by scalars and added when they have the same length:

```
integer :: v(3) = [1,2,3], w(3) = [3,2,1]
print '(*(i3))', 2 * v         ! Out:  2  4  6
print '(*(i3))', v + w         ! Out:  4  4  4
```

The scalar dot-product between vectors is computed with

```
print '(i3)', dot_product(v,w)  ! Out: 10
```

In contrast, the following calculation results in an elementwise multiplication:

```
print '(*(i3))', v * w          ! Out:  3  4  3
```

Matrices

Matrices are defined as rank 2 arrays:

```
integer, dimension(2,3) :: A  ! or ...
integer :: A(2,3)             ! ... equivalent
```

This declares `A` to be a matrix with 2 rows and 3 columns. The elements are accessed with `A(i,j)`.

Matrices can be initialized in various ways. The most versatile possibility is to *reshape* vectors:

```
A = reshape([1,2,3,4,5,6], [2,3], order=[2,1])
```

Note that in Fortran matrices are stored in *column-major* order. The `order=[2,1]` option now states that the matrix is to be filled in *row-by-row* order. The shape of the matrix is specifed by the entry `[2,3]`. The vector is converted into a 2×3 matrix so that the sequence of the first three entries 1, 2, 3 is assigned to the first row in `A` and 4, 5, 6 to the second.

Declaration and initialization can again be combined to:

```
integer :: A(2,3) = reshape([1,2,3,4,5,6], [2,3], order=[2,1])
```

Subblocks

As in NumPy and Matlab, indexing can access subblocks in a matrix using ranges or vectors of numbers instead of single numbers. This can for example be used to create a submatrix `A(p:q,r:s)` from a matrix `A` consisting of the intersection of the rows `p` through `q` and the columns `r` through `s`:

```
integer :: A(4,4) = reshape([(i, i=1,16)], [4,4], order=[2,1])
print '(2i4)', transpose(A(3:4, 2:3))
  10  11
  14  15
```

Note that the transpose operation is required to print the entries in a row-major form.

Lower and upper bounds can be left out: `A(3:,:2)` is equivalent to `A(3:4,1:2)`.

A special case is a single colon, which selects all rows or columns. For example, `A(:,j)` denotes the j-th column and `A(i,:)` denotes the i-th row of A.

Selections do not have to be contiguous. With the vectors `v(2) = [1,3]`, `w(3) = [2,4]` we get for example:

```
print '(2i4)', transpose(A(v,w))
```

```
  2   4
 10  12
```

Assignments can also be made to subblocks. This can be used to initialize matrices, here for example row by row:

```
A(1, :) = [ 1,  2,  3,  4]
A(2, :) = [ 5,  6,  7,  8]
A(3, :) = [ 9, 10 ,11, 12]
A(4, :) = [13, 14, 15, 16]
```

Matrix Product

The product of two matrices `A` and `B` of compatible shape is computed with the intrinsic function `matmul(A,B)`. It can also be used for matrix-vector products `matmul(A,v)`. Note that the vector `v` is automatically interpreted as a column vector.

Conjugate Gradient

We show that the linear algebraic tools developed so far are suffcient to implement the conjugate gradient method discussed previously on various occasions. For background we refer to Sect. 4.2 in the chapter Scientific Python.

Example 17.12 (Conjugate Gradient) Here is the Fortran version. We formulate the program core as a subroutine:

```
1  subroutine cgr(n, A, b, x)
2    integer :: n; real :: A(n,n), b(n), x(n)
3    real :: r(n), p(n), Ap(n), rs_old, alpha, rs_new; integer :: i
4    x = 0.0; r = b; p = b           ! start values
5    rs_old = dot_product(r,r)       !  - dito -
6    do i = 1, n
7      Ap = matmul(A,p)
8      alpha = rs_old / (dot_product(p, Ap))
9      x = x + alpha * p
10     r = r - alpha * Ap
11     rs_new = dot_product(r,r)
12     if (sqrt(rs_new) < 1e-6) exit  ! sqrt built-in function
13     p = r + (rs_new / rs_old) * p
14     rs_old = rs_new
15   end do
16 end subroutine
```

In line 1 we supply the problem size n as argument. This is not stricly necessary, but it makes for more readable code. In line 2 we specify the type and dimension of the input arguments. In line 3 we declare additional internal variables that are used in the computation.

The computation in the loop in lines 6 to 15 corresponds precisely to the one in the Python program, so the reader is asked to consult the comments there. Note only that we here use the Fortran `dot_product` and `matmul` functions introduced above.

We test the subroutine `cgr` for the same matrix A and vector b as in the Python example:

```
17 program testGradient
18   real :: A(4,4), b(4), x(4)
19   A(1, :) = [ 9,   3,  -6,  12]
20   A(2, :) = [ 3,  26,  -7, -11]
21   A(3, :) = [-6,  -7,   9,   7]
22   A(4, :) = [12, -11,   7,  65]
23   b = [18, 11, 3, 73]
24   call  cgr(4, A, b, x)  ! the result is stored in x
25   print '(*(f7.3))', x   ! Out:  1.000  1.000  1.000  1.000
26 end program
```

In the initialization of the matrix `A` and vector `b`, the integer entries are automatically converted to reals.

The format specification `f7.3` in line 25 causes a `real` number to be rounded to 3 decimal places, the number of digits to be set to 7 (including the decimal point), and the number to be printed right-justified.

Determinant

In Example 6.9 on page 128, we developed a recursive C function to compute the matrix determinant det(A) according to Laplace's formula, by expansion along the first row.

Example 17.13 Here is the Fortran version. Again, the inclusion of the function in a module seems to be the path of least resistance:

```
module laplace_det
  contains
  recursive function det(A) result(sum)
    real :: A(:,:), sum
    real :: A_sub( size(A,1)-1, size(A,1)-1 )
    integer :: col( size(A,1) ), n, j
    n = size(A,1)
    if (n == 1) then
      sum = A(1,1); return
    end if
    sum = 0.0
    col = [(j, j = 1, n)]
    do j = 1, n
      A_sub = A(2:n, pack(col, col /= j))
      sum = sum + (-1)**(j+1) * A(1,j) * det(A_sub)
    end do
```

```
17   end function
18 end module
```

In line 4 the type of the input and output variables is specified. `A` is declared as a matrix with indeterminate shape. It is set when the function is actually called.

In line 5 we declare a matrix `A_sub` to hold the submatrices needed in the expansion. The term '`size(A,1)`' denotes the number of rows (and also columns, since we assume the matrix is square) in `A`.

In line 6 we declare a vector `col`, which is filled with the column indices in line 12.

In line 9, the value `A(1,1)` is returned when the recursion base case is reached. Otherwise, in line 14, `A_sub` is loaded with the elements of `A`, deleting row 1 and column *j*.

In line 15, the determinant of the submatrix is computed recursively, multiplied by `A(1,j)`, and added to or subtracted from the sum according to the expansion rule.

The program is tested for the same matrix as in Example 6.9:

```
19 program detTest
20   use laplace_det
21   real :: A(3,3) = reshape([1,2,3,1,1,1,3,3,1], [3,3], order=[2,1])
22   print *, det(A)  ! Out: 2.00000000
23 end program
```

Remark We would have preferred to include a line like

```
integer, parameter :: n = size(A,1)
```

before line 3, so that we could use `n` as abbrevation in lines 5 and 6. However, this is not accepted by the compiler.

The Lapack Package

Lapack (for *L*inear *A*lgebra *Pack*age) is the standard software library for high performace numerical linear algebra in Fortran. Actually, also in many other programming environments, such as Matlab and SciPy, libraries for numerical linear algebra are built on top of Lapack. For details see `netlib.org/lapack`.

Example 17.14 (LU Decomposition) As a typical example we show how to use the Lapack subroutine `sgetrf` to compute the LU decomposition of a matrix A in the form $A = PLU$. For details see e.g. page 63. The Lapack naming convention still reflects the time when only 6-letter names were allowed: "s" stands for single precision, i.e. kind-4 reals, "ge" for general rectangular matrix, "tr" for triangular, "f" for factorization.

The program begins with:

```
1 program luTest
2   real :: A(3,3) = reshape([1,2,3,1,1,1,3,3,1], [3,3], order=[2,1])
3   integer, parameter :: n = size(A,1)
4   real :: P(n,n) = 0, L(n,n) = 0, U(n,n) = 0, row(n)
5   integer :: ipiv(n), info
```

In line 2 we define the same example matrix as in the corresponding Python example on page 63. In line 3 we declare `n` as constant. This is necessary to be able to use it in the declarations in lines 4 and 5.

After these preparations we can call our Lapack subroutine:

```
6   call sgetrf(n, n, A, n, ipiv, info)
```

The parameter `n` describes the shape of the argument matrix. The vector `ipiv` stores information about the row permutations used in the factorization, `info` is used to pass possible error codes.

The factorization is performed in-place. On return, the upper triangular part of `A` including the diagonal contains the nonzero elements in U, the part below the diagonal contains the nonzero elements in L, with the 1-entry diagonal not needing to be stored.

In the following loop, the entries are separated between the matrix variables `L` and `U`. Furthermore, in line 10 the permutation matrix is initialized as identity matrix:

```
7   do i = 1, n
8     L(i,  :i-1) = A(i,  :i-1);  L(i, i) = 1
9     U(i, i:n)   = A(i, i:n )
10    P(i,i) = 1
11  end do
```

Then the specifc form of the permuation matrix is determined:

```
12  do i = 1, n
13    if (ipiv(i) /= i) then
14      row = P(i,:); P(i,:) = P(ipiv(i),:); P(ipiv(i),:) = row
15    end if
16  end do
```

In the present case, `ipiv` is `[3,3,3]`, which means that when A is factorized, first row 1 is swapped with row 3, then row 2 with row 3. The last entry in `ipiv` is again 3, which means that row 3 is swapped with itself, i.e. not moved at all.

At this point we have computed a decomposition $PA = LU$. As a last step we shift P over to the other side as P^{-1}. But for permutation matrices, the inverse is simply the transpose. So we get

```
17  P = transpose(P)
18  A = matmul(P, matmul(L,U))
```

We check that everything has worked properly:

```
19  do i = 1, n
20    print "(4(3f5.1'   '))", P(i,:), L(i,:), U(i,:), A(i,:)
21  end do
22 end program
```

To run the program we need the Lapack library. It can be installed with

```
$ brew install lapack
```

To compile it, we must include a link to the library:

```
$ gfortran  test.f90  -o test -llapack
$ ./test
```

As output of the print loop in lines 19 to 21 we get P, L, U, A in that order:

```
0.0  1.0  0.0     1.0  0.0  0.0     3.0  3.0  1.0     1.0  2.0  3.0
0.0  0.0  1.0     0.3  1.0  0.0     0.0  1.0  2.7     1.0  1.0  1.0
1.0  0.0  0.0     0.3 -0.0  1.0     0.0  0.0  0.7     3.0  3.0  1.0
```

This correspondes to the result obtained in Python on page 63.

17.6 Derived Types, Structures

Similar to C/C++ and Julia, we can define *derived data types*, also called *structures*, in Fortran. A derived data type can consist of data objects of different types. Derived types are often implemented in modules, so you can easily reuse them. They may also contain type-bound procedures that are intended to process the structure objects.

We illustrate the basic ideas along a structure for representing fractions. For background, we refer to Example 3.19 in the Python chapter.

We represent a fraction such as 3/4 as a pair (3, 4) of numerator and denominator. A corresponding structure can be defined by

```
1 type fract
2   integer :: num, den
3 end type
```

A variable ‘a’ of the `fract` type can then be declared with

```
4 type (fract) :: a
```

The component selector percent ‘%’ can be used to access components, e.g. `a % num` and `a % den`. Note that here both components are of the same type. Of course, this is generally not necessarily the case.

A `fract` variable `a` can be initalized componentwise, e.g. with

```
5 a % num = 3; a % den = 4
```

Another way is to initialize with the function `fract`, which has the same name as the type. It takes two arguments corresponding to the order in the declaration: `num`, `den`. The following is equivalent to line 5:

```
6 a = fract(3,4)
```

Declaration and initialization can again be combined to

```
7 type (fract) :: a = fract(3,4)
```

Fortran allows you to set a default (initial) value of any component in the derived type definition itself, such as

```
type fract
  integer :: num = 1, den
end type
type (fract) :: a
type (fract) :: b = fract(2,3)
print *, a % num, b % num  ! Out: 1 2
```

In this case the default setting applies (as in `a`), unless otherwise specified (as in `b`).

As in Python and Julia, we can define functions that are bound to a specific derived type. The following example shows how this can be used to emulate class constructions in object-oriented programming:

Example 17.15 Similar to Example 3.19, we define a module `fraction` that encapsulates type declaration and specific application functions in a a single entity.
We begin with the declaration part:

```
1  module fraction
2    type fract
3      integer :: num, den
4    end type
5    interface operator (+)
6      procedure add_fract
7    end interface
8    interface operator (==)
9      procedure eq_fract
10   end interface
```

The type declaration in lines 1 to 3 is as before. In lines 5 through 7, the '`+`' operator is overloaded so that we can use the fraction addition function defined below in infix form. Note that `add_fraction` is referred to in the interface declaration as `procedure`, a term that includes `function` and `subroutine`.
The specification in lines 8 to 10 is analogous.

We come to the implementation section:

```
11   contains
12   function add_fract(a, b) result(c)
13     type(fract), intent(in) :: a, b; type(fract) :: c
14     c % num = (a % num * b % den) + (b % num * a % den)
15     c % den = a % den * b % den
16   end function
17   function eq_fract(a, b) result(eq)
18     type(fract), intent(in) :: a, b; logical :: eq
19     eq = (a % num * b % den == b % num * a % den)
20   end function
21 end module
```

The only thing to note here is that the overload interface requires that the input arguments `a`, `b` in lines 13 and 18 are explicitly given the `intent(in)` attribute.

The construction can now be tested:

```
program fracTest
  use fraction
  type (fract) :: a = fract(3,4), b = fract(1,2), c, d
  logical :: eq
  c = a + b; d = b + a
  eq = (c == d)
  print '(2i3, l3)', c, eq  ! Out: 10 8 T
end program
```

Arrays of Derived Types

Just as with basic types, also derived-type objects can be stored in arrays.

In Example 17.11, we discussed how to filter out zero entries from an array. Here we extend this idea and create an array representation for sparse vectors. To make provisions for the case where the number of entries is not known a priori, we use allocatable arrays.

Example 17.16 (Sparse Vectors) We write a program to convert an array of integers to a sparse representation in an allocatable array.

We start with the same array as in Example 17.11:

```
program sparseVectors
  integer, parameter :: n = 10; integer :: i, j, m
  integer :: a(n) = [0,2,0,0,8,10,0,14,16,18]
```

Index-value pairs are represented as objects of a derived type `node`:

```
  type node
    integer :: idx, val
  end type
```

We introduce two arrays of type `node`, `v` of fixed size `n`, and `w` allocatable:

```
  type (node) :: v(n)
  type (node), allocatable :: w(:)
```

In `v` we store all index-value pairs of elements in `a`:

```
  v = [(node(i, a(i)), i = 1, n)]  ! implied do loop constructor
```

Then we compute the number `m` of entries in `v` that have a nonzero value component, allocate the vector `w` with that size, and fill it with the index-value pairs:

```
  m = size(pack(a, a /= 0))
  allocate(w(m))
  w = pack(v, v % val /= 0)
```

To check the construction we print the sequence of index-value pairs in `w`:

```
  do j=1, m
    print '(2i4)', w(j)
  end do
end program
```

17.7 Pointers

As in other programming languages, a pointer variable in Fortran stores the memory address of an object. In Fortran, however, the explicit use of pointers is rarely necessary. One important application is in the construction of linked lists that can be extended dynamically.

In this section, we first present a simple program that demonstrates some of the key concepts of Fortran pointers. We then use pointers to create a linked list for storing sparse vectors.

Example 17.17 The following program illustrates the central pointer concepts:

```
 1 program pointerExample
 2   integer, pointer :: p, q
 3   integer, target :: a
 4   ! print *, p
 5   p => null(); print *, p, associated(p)  ! Out: 0 F
 6   allocate(p); print *, p, associated(p)  ! Out: 0 T
 7   p = 1;       print *, p                 ! Out: 1
 8   q => p;      print *, q, associated(q)  ! Out: 1 T
 9   a = 2
10   q => a;      print *, q                 ! Out: 2
11   print *, p + a, p + q                   ! Out: 3 3
12 end program
```

In line 2 we declare p, q as integer pointers. Note that the pointer declaration is written just like any other attribute. In fact, once p, q are properly initialized, we can use them like normal objects. In Fortran dereferencing is automatic; there is no need to write for example, *p as in C.

However, some precautions are required. Note that line 4 is commented out. Otherwise, the print command would try to access the integer value at the memory location pointed to by p. But p is still a so-called "dangling pointer" which is not connected to such a memory. Therefore the program would crash.

The symbol '=>' in line 5 stands for an *address* assignment: p gets the address of a special null value. The print command now returns this special value, but says that p is still not associated to a memory location that can contain a valid `integer` object. Note that the assignment in line 5 can again be combined with the declaration in line 2.

In line 6 such a memory location is allocated and its adress is stored in p. The status "assocciated" has changed to *true*.

Line 7 shows that p can now be used like any integer variable.

In line 8, the address stored in p is passed to q so that both now point to the same integer location. In particular `associated(q)` returns *true*.

We return to line 3 and declare an integer variable a with the attribute `target`. This means that it can be referenced by a pointer. In line 9, we assign a value to it. In line 10, the pointer q is given its adress. Dereferenced in the print command, it then returns the value stored there.

Line 11 shows that there is no difference between pointers and normal variables when they are used in actual expressions.

Sparse Vectors as Linked Lists

In Example 6.14 we discussed a C program to represent sparse vectors as linked lists. The construction was based on structures and pointers. Here we present a corresponding Fortran version.

Example 17.18 The program starts with the integer array to be converted. Note that we consider the same array as in Example 17.16 above:

```
1 program linkedList
2   integer, parameter :: n = 10; integer :: i
3   integer :: a(n) = [0,2,0,0,8,10,0,14,16,18]
```

We then introduce our data type `node` to store nonzero values, their index and a pointer to the next node:

```
4   type node
5     integer :: index, value
6     type (node), pointer :: next => null()  ! zero initialized
7   end type
```

We declare two `node` pointers `root` and `current`. In line 9, we allocate memory to store a `node` variable and assign the adress to `root`. In line 10, we pass this address also to `current`:

```
8   type (node), pointer :: root, current
9   allocate(root)
10  current => root
```

In the following main loop, we connect all entries with a nonzero value one by one to form a linked list:

```
11  do i = 1, n
12    if (a(i) == 0) cycle  ! skip to next index
13    current % index = i
14    current % value = a(i)
15    allocate(current % next)
16    current => current % next
17  end do
```

If `a` in line 12 contains a zero value at index `i`, we skip it and continue with the next index. If the value is $\neq 0$, we assign index and value to the current node, allocate a new node, and set `current` to point to it. In lines 13 and 14 again note that the pointer `current` is automatically dereferenced.

The result can then be checked by iterating through all nodes, starting with `root`:

```
18   current => root
19   do while (associated(current % next))
20     print *, current % index, current % value
21     current => current % next
22   end do
23 end program
```

The end is reached when `current%next` in line 19 is not connected to any successor node.

17.8 Parallel Fortran

Modern high-performance computing, for example in numerical weather and climate prediction or simulations in particle physics, can now use many thousands of computational cores. Many of the applications in these areas are written in Fortran.

This section covers two major approaches to parallel computing in Fortran, MPI and OpenMP, which have already been introduced in the corresponding Python and C/C++ chapters 12 and 13. The next section then deals with Fortran's own *coarrays* approach.

We need to install the required components, in our macOS for example with

```
$ brew install opencoarrays
```

For Linux and Windows we again refer to [2].

MPI and openMP

We begin with a simple MPI program already discussed in Example 12.1 on page 273:

Example 17.19 Here is the Fortran version:

```
1 program hello_mpi
2   use mpi_f08
3   integer :: rank, size, ierror
4   call mpi_init(ierror)
5   call mpi_comm_size(mpi_comm_world, size, ierror)
6   call mpi_comm_rank(mpi_comm_world, rank, ierror)
7   print *, 'Hello World from process', rank, 'of',  size
8   call mpi_finalize(ierror)
9 end program
```

In line 2 we import the current version of the Fortran MPI module `mpi_f08`. Note that the Fortran versions of the MPI commands require one additional variable `ierror` to pass a possible error code.

Assume the program is stored in a file `test.f90`. It can then be compiled and run with

```
$ mpif90 test.f90  -o test
$ mpiexec -n 8 ./test
```

The adaptation of OpenMP programs is also straightforward:

Example 17.20 Here is the Fortran version of Example 13.4 on page 298:

```
1 program hello_omp
2   use omp_lib  ! Fortran module for parallel OpenMP execution
3   call omp_set_num_threads(8)
4   !$omp parallel
5     print *, 'Hello from thread', omp_get_thread_num(), &
6              'of', omp_get_num_threads()
7   !$omp end parallel
8 end program
```

Just note that in line 3 we set the number of threads to 8. Without this specification, the number of threads defaults to one per available core.

Again assume the program is stored in `test.f90`. It can then be compiled and run with

```
$ gfortran  -fopenmp test.f90  -o test
$ ./test
```

Hybrid Message Passing and Shared Memory Programming

Also in Fortran we can write programs that use both MPI and OpenMP.

Example 17.21 We write a Fortran version of the distributed integral approximation in Sect. 13.8 on page 306.

We begin with the integrand function:

```
real function f(x); real :: x
  f = 4.0/(1.0 + x*x)
end function
```

This is followed by the function for calculating the integral parts, distributed according to the OMP protocol:

```
function partial_pi(n, start, step) result(sum)
  integer :: n, start, step, i
  real :: h, sum, f
  h = 1.0/n
  !$omp parallel do reduction(+:sum)
    do i = start, n-1, step  ! note step argument in loop
      sum = sum + h*f(h*(i+0.5))
    end do
  !$omp end parallel do
  end function
```

Then the main computation is distributed among a number of processes according to the MPI protocol. The calculations of the local integral parts are then in turn distributed to all available threads in line 23:

```
14 program integralTest
15   use mpi_f08
16   real :: partial_pi, pi, pi_loc
17   integer  ::  rank, size, ierror, tag, n
18   call mpi_init(ierror)
19   call mpi_comm_size(mpi_comm_world, size, ierror)
20   call mpi_comm_rank(mpi_comm_world, rank, ierror)
21   if (rank == 0)  n = 1000
22   call mpi_bcast(n, 1, mpi_int, 0, mpi_comm_world, ierror)
23   pi_loc = partial_pi(n,rank,size)  ! OMP used here
24   call mpi_reduce(pi_loc, pi, 1, mpi_float, mpi_sum, 0,  &
25             mpi_comm_world, ierror)
26   if (rank == 0) print *, pi
27   call mpi_finalize(ierror)
28 end program
```

The program is then compiled and spawned to 4 processes:

```
$ mpif90  -fopenmp test.f90  -o test
$ mpiexec -n 4 ./test
  3.14159274
```

17.9 Coarray Fortran

Coarrays and images are the fundamentel concepts in Fortran's own approach to parallel processing.

On it's own, a Fortran image is very similar to an MPI process.

Example 17.22 We demonstrate the idea with a simple "Hello World" program:

```
program hello_world
  print *, 'Hello World from image ', this_image(),'of', num_images()
end program
```

Here `num_images()` corresponds to the `size` parameter in MPI, `this_image()` to `rank`. Just remember that indices in Fortran start at 1.

Assuming the program is stored in the file `test.f90`, we can compile and run it on 3 images in parallel with

```
$ caf test.f90 -o test
$ cafrun -n 3 ./test
  Hello World from image   3  of  3
  Hello World from image   1  of  3
  Hello World from image   2  of  3
```

Note that `caf` is simply a wrapper around our usual compiler `gfortran`, with some required additional features. `cafrun` is actually just an alias for `mpiexec`.

Communication through Coarrays

Suppose we have several images and a variable ‘a’ that can contain a local component for each image. If we assume that the images are stacked on top of each other, we can imagine that the local components form a vertical array that goes through all the layers. This array can then be used to transfer values between images. And that is exactly the idea of a Fortran coarray.

Example 17.23 We illustrate the idea with a

```
program coArrayExample
```

for the distributed computation of π according to the Monte-Carlo method already discussed several times.

The following declaration of a coarray ‘hits’ to store the local contributions is central:

```
 integer, codimension[*] :: hits = 0
```

The codimension attribute declares the hits variable as a coarray, more precisely as an integer variable with a local instance on each image. The asterisk means that the actual length is not determined until runtime. The individual array entries hits[i] are then used to refer to the local variable instance in image number *i*. Note that we use square brackets instead of parentheses to distinguish coarrays from “normal” arrays. All instances are initialized with 0.

Then we set the total number of random trials. Here we choose $n = 10^6$:

```
integer, parameter :: n = 10**6
```

The computation is distributed among the images:

```
integer i; real :: x,y
do i = 1, n/num_images()  ! assume n divisible by number of images
  call random_number(x); call random_number(y)
  if (x*x + y*y  <= 1.0) hits = hits + 1  ! local to each image
end do
```

In the loop, the local hits count is computed in each image. Note that hits without brackets refers to the image’s own instance.

We wait for all images to finish their computation:

```
  sync all
```

Then we let image 1 collect the other partial results and add them to its own local contribution:

```
  if (this_image() == 1) then  ! executed only in image 1
    do i=2, num_images()       ! start at 2, collect from helpers
      hits = hits + hits[i]    ! collected
    end do
    print *, 4 * hits/real(n)  ! convert to real number
  end if
end program
```

The program is then compiled and distributed to 8 images:

```
$ caf test.f90 -o test
$ cafrun -n 8 ./test
   3.14115596
```

Remark Note that image 1 plays a double role here, similar to root process 0 in MPI: it participates in the distributed computation, but also takes overall control in lines 10 to 15. It is a typical pattern to have image 1 do any setup and terminal processing. We therefore often speak of image 1 as the root image, or simply *root*, and the others then as *helpers*.

Nested Communication

In Sect. 12.5 in the Python MPI chapter, we discussed an example where nested communication was required during a parallel computation, namely the iterative approximate solution for a Laplace equation. For background and details, we refer to that section.

Here we develop a correspoding Fortran coarray program. We begin with a sequential subroutine to compute the iteration steps for the inner points according to equation (4) on page 283:

```
1  module stencilComp
2    contains
3    subroutine stencil(U); real :: U(:,:)
4      real, allocatable :: Tmp(:,:)
5      integer :: n, m, i, j
6      Tmp = U  ! automatic allocation
7      n = size(U,1); m = size(U,2)
8      do i = 2, n-1
9        do j = 2, m-1
10         Tmp(i,j) = 0.25 * (U(i-1,j) + U(i,j-1) + U(i+1,j) + U(i,j+1))
11       end do
12     end do
13     U = Tmp
14   end subroutine
15 end module
```

In line 3, we declare the argument `U` in `stencil` to be a `real` matrix whose dimensions are set when the subroutine is called.

In line 4, we introduce an auxiliary matrix `Tmp`. The assignment in line 6 automatically allocates it with the same shape as `U` and then copies the `U` values.

In line 7, we store the number of rows and columns in `U` in the variables `n` and `m`. In line 10, the new value for each inner point is calculated and written to `Tmp`. Finally, `U` is replaced with the updated values in line 13.

Our actual computation program starts with

```
16 program laplace
17   use stencilComp
18   real :: U(10,10)  ! matrix to hold solution
```

As in Sect. 12.5, we divide the matrix `U` vertically into two parts with 5 columns each and distribute the parts to two images, which then compute the update of the stencil points for their local part. We collect the parts in a coarray `coU`. To calculate the values in the border columns, we also need the current values from the other image, the so-called "ghost points". Therefore, the coarray must also provide an additional column for them.

We arrive at the following declaration:

```
19   real, codimention[*] :: coU(10,6)
```

Then we set up the vector needed to define the boundary condition, and two index variables:

```
20   real :: x(10)
21   integer :: i, j
```

In the following, the root image 1 again assumes a double role: it organizes the computation, and additionally also computes the stencil updates for the left part.

First, the root image initiates the computation:

```
22   if (this_image() == 1) then
23     U = 0.0
24     x = [(i * 1./9, i = 0, 9)]  ! implied do loop constructor
25     U(1,:) = x * (1-x)          ! elementwise multiplication
26     coU(:,:)    = U(:, :6)      ! local instance
27     coU(:,:)[2] = U(:, 5:)      ! sent to helper image 2
28   end if
```

In line 23, the matrix is zero initialized. In lines 24 and 25 the boundary condition is computed and inserted as row 1 in `U`.

Then, in line 26, root assigns the left half of `U` plus the initial values of ghost column 6 to its own local covariable instance. In line 27 it sends the right part of `U` plus the ghost column 5 to the helper image 2.

The actual iteration is performed by both images in the following loop:

```
29   do i = 1, 40
30     call stencil(coU)
31     sync all
32     if (this_image() == 1) coU(:, 1)[2] = coU(:, 5)
33     if (this_image() == 2) coU(:, 6)[1] = coU(:, 1)
34   end do
```

In line 30, both images compute the stencil values of the next iteration for their inner points. In lines 32 and 33 they send the updated ghost columns to their neighbor. This process is also floridly called "halo exchange". In general, halo cells refer to array elements that belong to neighbor images, but are needed locally to perform the calculation.

When the computation is complete, the ghost columns are discarded and the root image combines the partial contributions to the overall result:

```
35   if (this_image() == 1) then
36     U(:,  :5) = coU(:,  :5)
37     U(:, 6: ) = coU(:, 2:)[2]
38   end if
```

Now that we have the solution, let's visualize it. Fortran does not offer any options for this. However, one standard is the use of the open source plotting program *Gnuplot*.

What remains to be done in Fortran is to store the solution matrix `U` in a form understood by Gnuplot. We again leave this to the root image 1:

```
39 if (this_image() == 1) then
40     U = transpose(U)
41     open(1, file="data.plt")
42     do i = 1, 10
43       do j = 1, 10
44         write(1,*) i, j, U(i,j)
45       end do
46       write(1,*)
47     end do
48     close(1)
49   end if
```

For technical reasons we transpose the matrix in line 40. In line 41, we open (or create it, if it does not exist) a file `data.plt` and assign it a reference number, here 1.

Then, in lines 42 to 47, we write the matrix to the file, according to the syntax required by Gnuplot. The first argument 1 in the write statements again refers to the file opened in line 41. The connection to the file is closed in line 48.

The only thing left to do is to exit the program:

```
50 end program
```

We can then compile it and spread the execution over two images:

```
$ caf test.f90 -o test
$ cafrun -n 2 ./test
```

We finally come to the visualization of the result with Gnuplot.

Gnuplot can be installed with

```
$ brew install gnuplot
```

Once installed, we start Gnuplot with the Terminal command:

```
$ gnuplot
```

and then at the Gnuplot prompt enter

```
gnuplot> splot "data.plt" using 1:2:3 with pm3d
```

The result is shown in Fig. 17.1.

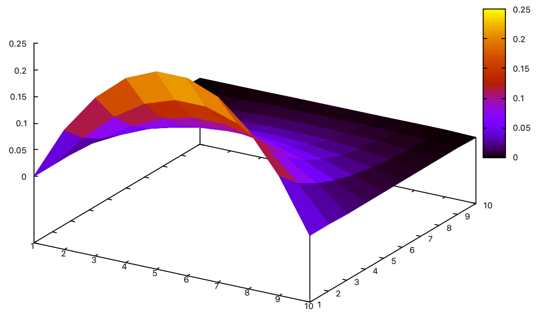

Fig. 17.1 Laplace Equation computed by Fortran coarray program

References

1. W. S. Brainerd *Guide to Fortran 2008 Programming.* (Springer-Verlag, 2015)
2. M. Curcic *Modern Fortran.* (Manning Publications, 2020)
3. M. Metcalf, J. Reid, M. Cohen *Modern Fortran Explained: Incorporating Fortran 2018.* (Oxford University Press, 2018)