ACM Computing Seminar Fortran Guide

Joseph P. McKenna

October 10, 2016

Contents

1 Introduction

This guide is intended to quickly get you up-and-running in scientific computing with Fortran.

1.1 About the language

Fortran was created in the 1950s for mathematical **FOR**-mula **TRAN**-slation, and has since gone through a number of revisions (FORTRAN 66, 77, and Fortran 90, 95, 2003, 2008, 2015). The language standards are put forth by the Fortran standards committee J3 in a document (ISO 1539-1:2010) available for purchase. The language syntax and intrinsic procedures make it especially suited for scientific computing. Fortran is a **statically-typed** and **compiled** language, like C++. You must declare the **type**, i.e. integer, real number, etc. of variables in programs you write. Your programs will be translated from human-readable *source code* into an executable file by software called a **compiler**. Fortran is **not case-sensitive**, so `matrix` and `MaTrIx` are translated to the same token by the compiler.

2 Getting started

The software that you need to get started comes prepackaged and ready to download on most Linux distributions. There are a few options for emulating a Linux environment in Windows or Mac OS, such as a virtual machine (VirtualBox) or package manager (MinGW or Cygwin on Windows and Brew on Mac OS).

2.1 Text editor

You will write the source code of your programs using a text editor. There are many options that have features designed for programming such as syntax highlighting and auto-completion. If you are an impossible-to-please perfectionist, you might want to check out Emacs. If you are easier to please, you might want to check out Sublime Text.

2.2 Compiler

To translate your source code into an executable, you will need a Fortran compiler. A free option is **gfortran**, part of the GNU compiler collection (gcc). The features of the Fortran language that are supported by the `gfortran` compiler are specified in the compiler manual. This is your most complete reference for the procedures intrinsic to Fortran that your programs can use. At the time of this writing, `gfortran` completely supports Fortran 95 and partially supports more recent standards.

2.3 Writing and compiling a program

A program is delimited by the `begin program` / `end program` keywords. A useful construct for keeping code that a program can use is called a **module**. A module is delimited by the `begin module` / `end module` keywords.

2.3.1 Hello world

Let's write a tiny program that prints "hello world" to the terminal screen in `hello.f90`.

```
program main
  print*, 'hello world'
end program main
```

To compile the program, execute the following command on the command line in the same directory as `hello.f90`

```
gfortran hello.f90
```

This produces an executable file named `a.out` by default (On Windows, this is probably named `a.exe` by default). To run, execute the file.

```
./a.out
```

```
hello world
```

We could have specified a different name for the executable file during compilation with the `-o` option of `gfortran`.

```
gfortran hello.f90 -o my_executable_file
```

On Windows, you should append the `.exe` extension to `my_executable_file`.

2.3.2 Template

Now let's write an empty source code template for future projects. Our source code template will consist of two files in the same directory (`./source/`). In the following files, the contents of a line after a `!` symbol is a comment that is ignored by the compiler. One file `header.f90` contains a module that defines things to be used in the main program.

```
module header
  implicit none
  ! variable declarations and assignments
contains
  ! function and subroutine definitions
end module header
```

This file should be compiled with the `-c` option of `gfortran`.

```
gfortran -c header.f90
```

This outputs the **object file** named `header.o` by default. An object file contains machine code that can be *linked* to an executable. A separate file `main.f90` contains the main program.

```
1 program main
2   use header
3   implicit none
4   ! variable declarations and assignments
5   ! function and subroutine calls
6 contains
7   ! function and subroutine definitions
8 end program main
```

On line 2 of `main.f90`, we instruct the main program to use the contents of `header.f90`, so we must link `header.o` when compiling `main.f90`.

```
gfortran main.f90 header.o -o main
```

To run the program, execute the output file `main`.

```
./main
```

As you get more experience, you may find it cumbersome to repeatedly execute `gfortran` commands with every modification to your code. A way around this is to use the `make` command-line utility. Using `make`, all the of the compilation commands for your project can be coded in a file named `makefile` in the same directory as your `.f90` source files. For example, the template above could use the following `makefile`.

```
 1 COMPILER = gfortran
 2 SOURCE = main.f90
 3 EXECUTABLE = main
 4 OBJECTS = header.o
 5
 6 all: $(EXECUTABLE)
 7 $(EXECUTABLE): $(OBJECTS)
 8     $(COMPILER) $(SOURCE) $(OBJECTS) -o $(EXECUTABLE)
 9 %.o: %.f90
10     $(COMPILER) -c $<
```

Then, to recompile both `header.f90` and `main.f90` after modifying either file, execute

```
make
```

in the same directory as `makefile`. The first four lines of the `makefile` above define the compiler command, file name of the main program, file name of the executable to be created, and file name(s) of linked object file(s), respectively. If you wrote a second module in a separate file `my_second_header.f90` that you wanted to `use` in `main.f90`, you would modify line 4 of `makefile` to `OBJ = header.o my_second_header.o`. The remaining lines of `makefile` define instructions for compilation.

2.4 Exercises

1. Compile and run `hello.f90`.

2. Execute `man gfortran` in any directory to bring up the manual for `gfortran`. Read the description and skim through the options. Do the same for `make`.

3 Data types

In both programs and modules, variables are declared first before other procedures. A variable is declared by listing its data type followed by `::` and the variable name, i.e. `integer :: i` or `real :: x`.

We will use the `implicit none` keyword at the beginning of each program and module as in line 2 of `header.f90` and line 3 of `main.f90` in Section 2.3.2. The role of this keyword is to suppress implicit rules for interpreting undeclared variables. By including it, we force ourselves to declare each variable we use, which should facilitate debugging when our program fails to compile. Without it, an undeclared variable with a name such as `i` is assumed to be of the `integer` data type whereas an undeclared variable with a name such as `x` is assumed to be of the `real` data type.

In addition to the most common data types presented below, Fortran has a `complex` data type and support for data types defined by the programmer (see Section 7.1).

3.1 The `logical` type

A `logical` data type can have values `.true.` or `.false.`. Logical expressions can be expressed by combining unary or binary operations.

```
logical :: a,b,c
a = .true.
b = .false.

! '.not.' is the logical negation operator
c = .not.a ! c is false

! '.and,' is the logical and operator
c = a.and.b ! c is false

! '.or.' is the logical or operator
c = a.or.b ! c is true

! '==' is the test for equality
c = 1 == 2 ! c is false

! '/=' is test for inequality
c = 1 /= 2 ! c is true
print*, c
```

Other logical operators include

- `<` or `.lt.`: less than
- `<=` or `.le.`: less than or equal
- `>` or `.gt.`: greater than
- `>=` or `.ge.`: greater than or equal

Logical expressions are often used in control structures.

3.2 The `integer` type

An `integer` data type can have integer values. If a real value is assigned to an `integer` type, the decimal portion is chopped off.

```
integer :: a = 6, b = 7 ! initialize a and b to 6 and 7, resp
integer :: c

c = a + b ! c is 13
c = a - b ! c is -1
c = a / b ! c is 0
c = b / a ! c is 1
c = a*b ! c is 42
c = a**b ! c is 6^7
c = mod(b,a) ! c is (b mod a) = 1
c = a > b ! c is 0 (logical gets cast to integer)
c = a < b ! c is 1 (logical gets cast to integer)
```

3.3 Floating point types

The two floating point data types `real` and `double precision` correspond to IEEE 32- and 64-bit floating point data types. A constant called *machine epsilon* is the least positive number in a floating point system that when added to 1 results in a floating point number larger than 1. It is common in numerical analysis error estimates.

```
real :: a ! declare a single precision float
double precision :: b ! declare a double precision float

! Print the min/max value and machine epsilon
! for the single precision floating point system
print*, tiny(a), huge(a), epsilon(a)

! Print the min/max value and machine epsilon
! for the double precision floating point system
print*, tiny(b), huge(b), epsilon(b)
```

```
1.17549435E-38   3.40282347E+38   1.19209290E-07
2.2250738585072014E-308   1.7976931348623157E+308   2.2204460492503131E-016
```

3.4 The `character` type

A `character` data type can have character values, i.e. letters or symbols. A character string is declared with a positive `integer` specifying its maximum possible length.

```
! declare a character variable s at most 32 characters
character(32) :: s

! assign value to s
s = 'file_name'

! trim trailing spaces from s and
! append a character literal '.txt'
print*, trim(s) // '.txt'
```

```
file_name.txt
```

3.5 Casting

An `integer` can be cast to a `real` and vice versa.

```
integer :: a = 1, b
real :: c, PI = 3.14159

! explicit cast real to integer
b = int(PI) ! b is 3

! explicit cast integer to real then divide
c = a/real(b) ! c is .3333...

! divide then implicit cast real to integer
c = a/b ! c is 0
```

3.6 The `parameter` keyword

The `parameter` keyword is used to declare constants. A constant must be assigned a value at declaration and cannot be reassigned a value. The following code is not valid because of an attempt to reassign a constant.

```
! declare constant variable
real, parameter :: PI = 2.*asin(1.) ! 'asin' is arcsine
```

```

PI = 3 ! not valid
```

The compiler produces an error like `Error: Named constant ‘pi’ in variable definition context (assignment)`.

3.7 Setting the precision

The `kind` function returns an `integer` for each data type. The precision of a floating point number can be specified at declaration by a literal or constant `integer` of the desired `kind`.

```
 1  ! declare a single precision
 2  real :: r
 3  ! declare a double precision
 4  double precision :: d
 5  ! store single precision and double precision kinds
 6  integer, parameter :: sp = kind(r), dp = kind(d)
 7  ! set current kind
 8  integer, parameter :: rp = sp
 9
10  ! declare real b in double precision
11  real(dp) :: b
12
13  ! declare real a with precision kind rp
14  real(rp) :: a
15
16  ! cast 1 to real with precision kind rp and assign to a
17  a = 1.0_rp
18
19  ! cast b to real with precision kind rp and assign to a
20  a = real(b,rp)
```

To switch the precision of each variable above with kind `rp`, we would only need to modify the declaration of `rp` on line 8.

3.8 Pointers

Pointers have the same meaning in Fortran as in C++. A pointer is a variable that holds the **memory address** of a variable. The implementation of pointers is qualitatively different in Fortran than in C++. In Fortran, the user cannot view the memory address that a pointer stores. A pointer variable is declared with the `pointer` modifier, and a variable that it points to is declared with the `target` modifier. The types of a `pointer` and its `target` must match.

```
! declare pointer
integer, pointer :: p
! declare targets
integer, target :: a = 1, b = 2

p => a ! p has same memory address as a
p = 2 ! modify value at address
print*, a==2 ! a is 2

p => b ! p has same memory address as b
p = 1 ! modify value at address
print*, b==1 ! b is 1

! is p associated with a target?
print*, associated(p)

! is p associated with the target a?
print*, associated(p, a)

! point to nowhere
nullify(p)
```

```
T
T
T
F
```

3.9 Arrays

The length of an array can be fixed or dynamic. The index of an array starts at 1 by default, but any index range can be specified.

3.9.1 Fixed-length arrays

An array can be declared with a single `integer` specifying its length in which cast the first index of the array is 1. An array can also be declared with an `integer` range specifying its first and last index.

Here's a one-dimensional array example.

```
! declare arrray of length 5
! index range is 1 to 5 (inclusive)
real :: a(5)

! you can work with each component individually
! set the first component to 1
a(1) = 1.0

! or you can work with the whole array
! set the whole array to 2
a = 2.0

! or you can with slices of the array
! set elements 2 to 4 (inclusive) to 3
a(2:4) = 3.0
```

And, here's a two-dimensional array example.

```
! declare 5x5 array
! index range is 1 to 5 (inclusive) in both axes
real :: a(5,5)

! you can work with each component individually
! set upper left component to 1
a(1,1) = 1.0

! or you can work with the whole array
! set the whole array to 2
a = 2.0

! or you can with slices of the array
! set a submatrix to 3
a(2:4, 1:2) = 3.0
```

Fortran includes intrinsic functions to operate on an array `a` such as

- `size(a)`: number of elements of `a`
- `minval(a)`: minimum value of `a`

- `maxval(a)`: maximum value of `a`
- `sum(a)`: sum of elements in `a`
- `product(a)`: product of elements in `a`

See the `gfortran` documentation for more.

3.9.2 Dynamic length arrays

Dynamic arrays are declared with the `allocatable` modifier. Before storing values in such an array, you must `allocate` memory for the array. After you are finished the array, you ought to `deallocate` the memory that it occupies.

Here's a one-dimensional array example.

```
 1  ! declare a one-dim. dynamic length array
 2  real, allocatable :: a(:)
 3
 4  ! allocate memory for a
 5  allocate(a(5))
 6
 7  ! now you can treat a like a normal array
 8  a(1) = 1.0
 9  ! etc...
10
11  ! deallocate memory occupied by a
12  deallocate(a)
13
14  ! we can change the size and index range of a
15  allocate(a(0:10))
16
17  a(0) = 1.0
18  ! etc...
19
20  deallocate(a)
```

Without the last `dellaocate` statement on line 20 the code above is valid, but the memory that is allocated for `a` will not be freed. That memory then cannot be allocated to other resources.

Here's a two-dimensional array example.

```
! declare a two-dim. dynamic length array
```

```
real, allocatable :: a(:,:)

! allocate memory for a
allocate(a(5,5))

! now you can treat a like a normal array
a(1,1) = 1.0
! etc...

! deallocate memory occupied by a
deallocate(a)

! we can change the size and index range of a
allocate(a(0:10,0:10))

a(0,0) = 1.0
! etc...

deallocate(a)
```

4 Control structures

Control structures are used to direct the flow of code execution.

4.1 Conditionals

4.1.1 The `if` construct

The `if` construct controls execution of a single block of code. If the block of code is more than one line, it should be delimited by an `if` / `end if` pair. If the block of code is one line, it can be written on one line. A common typo is to forget the `then` keyword following the logical in an `if` / `end if` pair.

```
real :: num = 0.75

if (num < .5) then
   print*, 'num: ', num
   print*, 'num is less than 0.5'
end if

if (num > .5) print*, 'num is greater than 0.5'
```

```
num is greater than 0.5
```

4.1.2 Example: `if` / `else` and random number generation

The `if` / `else` construct controls with mutually exclusive logic the execution of two blocks of code.

The following code generates a random number between 0 and 1, then prints the number and whether or not the number is greater than 0.5

```
real :: num

! seed random number generator
call srand(789)

! rand() returns a random number between 0 and 1
num = rand()

print*, 'num: ', num

if (num < 0.5) then
   print*, 'num is less than 0.5'
else
   print*, 'num is greater then 0.5'
end if

! do it again
num = rand()

print*, 'num: ', num

if (num < 0.5) then
   print*, 'num is less than 0.5'
else
   print*, 'num is greater then 0.5'
end if
```

```
num:    6.17480278E-03
num is less than 0.5
num:   0.783314705
num is greater then 0.5
```

Since the random number generator was seeded with a literal integer, the above code will produce the *same* output each time it is run.

4.1.3 Example: `if` / `else if` / `else`

The `if` / `else if` / `else` construct controls with mutually exclusive logic the execution of three or more blocks of code. The following code generates a random number between 0 and 1, then prints the number and which quarter of the interval $[0,1]$ that the number is in.

```
real :: num

! seed random number generator with current time
call srand(time())

! rand() returns a random number between 0 and 1
num = rand()

print*, 'num:', num

if (num > 0.75) then
   print*, 'num is between 0.75 and 1'
else if (num > 0.5) then
   print*, 'num is between 0.5 and 0.75'
else if (num > 0.25) then
   print*, 'num is between 0.25 and 0.5'
else
   print*, 'num is between 0 and 0.25'
end if
```

```
num:  0.679201365
num is between 0.5 and 0.75
```

Since the random number generator was seeded with the current time, the above code will produce a *different* output each time it is run.

4.2 Loops

4.2.1 The `do` loop

A `do` loop iterates a block of code over a range of integers. It takes two `integer` arguments specifying the minimum and maximum (inclusive) of the range and takes an optional third `integer` argument specifying the iteration stride in the form `do i=min,max,stride`. If omitted, the stride is 1.

The following code assigns a value to each component of an array then prints it.

```
integer :: max = 10, i
real, allocatable :: x(:)

allocate(x(0:max))

do i = 0,max
   ! assign to each array component
   x(i) = i / real(max)

   ! print current component
   print "('x(', i0, ') = ', f3.1)", i, x(i)
end do

deallocate(x)
```

```
x(0) = 0.0
x(1) = 0.1
x(2) = 0.2
x(3) = 0.3
x(4) = 0.4
x(5) = 0.5
x(6) = 0.6
x(7) = 0.7
x(8) = 0.8
x(9) = 0.9
x(10) = 1.0
```

An *implicit* `do loop` can be used for formulaic array assignments. The following code creates the same array as the last example.

```
integer :: max = 10
real, allocatable :: x(:)

allocate(x(0:max))

```

```
! implicit do loop for formulaic array assignment
x = [(i / real(max), i=0, max)]

deallocate(x)
```

Example: row-major matrix The following code stores matrix data in a one-dimensional array named `matrix` in `row-major` order. This means the first `n_cols` elements of the array will contain the first row of the matrix, the next `n_cols` of the array will contain the second row of the matrix, etc.

```
integer :: n_rows = 4, n_cols = 3
real, allocatable :: matrix(:)
! temporary indices
integer :: i,j,k

! index range is 1 to 12 (inclusive)
allocate(matrix(1:n_rows*n_cols))

! assign 0 to all elements of matrix
matrix = 0.0

do i = 1,n_rows
   do j = 1,n_cols
      ! convert (i,j) matrix index to "flat" row-major index
      k = (i-1)*n_cols + j

      ! assign 1 to diagonal, 2 to sub/super-diagonal
      if (i==j) then
         matrix(k) = 1.0
      else if ((i==j-1).or.(i==j+1)) then
         matrix(k) = 2.0
      end if
   end do
end do

! print matrix row by row
do i = 1,n_rows
   print "(3(f5.1))", matrix(1+(i-1)*n_cols:i*n_cols)
end do

```

```
deallocate(matrix)
```

```
1.0  2.0  0.0
2.0  1.0  2.0
0.0  2.0  1.0
0.0  0.0  2.0
```

4.2.2 The `do while` loop

A `do while` loop iterates while a logical condition evaluates to `.true.`.

Example: truncated sum The following code approximates the geometric series

$$\sum_{n=1}^{\infty}\left(\frac{1}{2}\right)^n = 1.$$

The `do while` loop begins with $n = 1$ and exits when the current summand does not increase the current sum. It prints the iteration number, current sum, and absolute error

$$E = 1 - \sum_{n=1}^{\infty}\left(\frac{1}{2}\right)^n.$$

```
real :: sum = 0.0, base = 0.5, tol = 1e-4
real :: pow = 0.5
integer :: iter = 1

do while (sum+pow > sum)
   ! add pow to sum
   sum = sum+pow
   ! update pow by one power of base
   pow = pow*base

   print "('Iter: ', i3, ', Sum: ', f0.10, ', Abs Err: ', f0.10)", iter, sum, 1-su

   ! update iter by 1
   iter = iter+1
end do
```

```
Iter:   1, Sum: .5000000000, Abs Err: .5000000000
Iter:   2, Sum: .7500000000, Abs Err: .2500000000
```

```
Iter:   3, Sum: .8750000000, Abs Err: .1250000000
Iter:   4, Sum: .9375000000, Abs Err: .0625000000
Iter:   5, Sum: .9687500000, Abs Err: .0312500000
Iter:   6, Sum: .9843750000, Abs Err: .0156250000
Iter:   7, Sum: .9921875000, Abs Err: .0078125000
Iter:   8, Sum: .9960937500, Abs Err: .0039062500
Iter:   9, Sum: .9980468750, Abs Err: .0019531250
Iter:  10, Sum: .9990234375, Abs Err: .0009765625
Iter:  11, Sum: .9995117188, Abs Err: .0004882812
Iter:  12, Sum: .9997558594, Abs Err: .0002441406
Iter:  13, Sum: .9998779297, Abs Err: .0001220703
Iter:  14, Sum: .9999389648, Abs Err: .0000610352
Iter:  15, Sum: .9999694824, Abs Err: .0000305176
Iter:  16, Sum: .9999847412, Abs Err: .0000152588
Iter:  17, Sum: .9999923706, Abs Err: .0000076294
Iter:  18, Sum: .9999961853, Abs Err: .0000038147
Iter:  19, Sum: .9999980927, Abs Err: .0000019073
Iter:  20, Sum: .9999990463, Abs Err: .0000009537
Iter:  21, Sum: .9999995232, Abs Err: .0000004768
Iter:  22, Sum: .9999997616, Abs Err: .0000002384
Iter:  23, Sum: .9999998808, Abs Err: .0000001192
Iter:  24, Sum: .9999999404, Abs Err: .0000000596
Iter:  25, Sum: 1.0000000000, Abs Err: .0000000000
```

Example: estimating machine epsilon The following code finds machine epsilon by shifting the rightmost bit of a binary number rightward until it falls off. Think about how it does this. Could you write an algorithm that finds machine epsilon using the function `rshift` that shifts the bits of float rightward?

```
double precision :: eps
integer, parameter :: dp = kind(eps)
integer :: count = 1

eps = 1.0_dp
do while (1.0_dp + eps*0.5 > 1.0_dp)
   eps = eps*0.5
   count = count+1
end do

```

```
print*, eps, epsilon(eps)
print*, count, digits(eps)
```

```
2.2204460492503131E-016   2.2204460492503131E-016
          53          53
```

4.2.3 Example: the `exit` keyword

The `exit` keyword stops execution of code within the current scope.

The following code finds the *hailstone sequence* of $a_1 = 6$ defined recursively by

$$a_{n+1} = \begin{cases} a_n/2 & \text{if } a_n \text{ is even} \\ 3a_n + 1 & \text{if } a_n \text{ is odd} \end{cases}$$

for $n \geq 1$. It is an open conjecture that the hailstone sequence of any initial value a_1 converges to the periodic sequence $4, 2, 1, 4, 2, 1 \ldots$. Luckily, it does for $a_1 = 6$ and the following infinite `do` loop exits.

```
integer :: a = 6, count = 1

! infinite loop
do
   ! if a is even, divide by 2
   ! otherwise multiply by 3 and add 1
   if (mod(a,2)==0) then
      a = a/2
   else
      a = 3*a+1
   end if

   ! if a is 4, exit infinite loop
   if (a==4) then
      exit
   end if

   ! print count and a
   print "('count: ', i2, ', a: ', i2)", count, a

   ! increment count
   count = count + 1
end do
```

```
count:  1, a:  3
count:  2, a: 10
count:  3, a:  5
count:  4, a: 16
count:  5, a:  8
```

5 Input/Output

5.1 File input/output

5.1.1 Reading data from file

The contents of a data file can be read into an array using `read`. Suppose you have a file `./data/array.txt` that contains two columns of data

```
1 1.23
2 2.34
3 3.45
4 4.56
5 5.67
```

This file can be opened with the `open` command. The required first argument of `open` is an `integer` that specifies a *file unit* for `array.txt`. Choose any number that is not in use. The unit numbers `0`, `5`, and `6` are reserved for system files and should not be used accidentally. Data are read in **row-major** format, i.e. across the first row, then across the second row, etc.

The following code reads the contents of `./data/array.txt` into an array called `array`.

```
! declare array
real :: array(5,2)
integer :: row

! open file and assign file unit 10
open (10, file='./data/array.txt', action='read')

! read data from file unit 10 into array
do row = 1,5
   read(10,*) array(row,:)
end do
```

```

! close file
close(10)
```

5.1.2 Writing data to file

Data can be written to a file with the `write` command.

```
real :: x
integer :: i, max = 5

! open file, specify unit 10, overwrite if exists
open(10, file='./data/sine.txt', action='write', status='replace')

do i = 0,max
   x = i / real(max)

   ! write to file unit 10
   write(10,*) x, sin(x)
end do
```

This produces a file `sine.txt` in the directory `data` containing

```
 0.00000000       0.00000000
0.200000003      0.198669329
0.400000006      0.389418334
0.600000024      0.564642489
0.800000012      0.717356086
 1.00000000      0.841470957
```

5.2 Formatted input/output

The format of a `print`, `write`, or `read` statement can be specified with a `character` string. A format character string replaces the `*` symbol in `print*` and the second `*` symbol in `read(*,*)` or `write(*,*)`. A format string is a list of literal character strings or character descriptors from

- `a`: character string
- `iW`: integer
- `fW.D`: float point

- `esW.DeE`: scientific notation
- `Wx`: space

where `W`, `D`, and `E` should be replaced by numbers specifying width, number of digits, or number of exponent digits, resp. The width of a formatted integer or float defaults to the width of the number when `W` is `0`.

```
character(32) :: fmt, a = 'word'
integer :: b = 1
real :: c = 2.0, d = 3.0

! character string and 4 space-delimited values
print "('four values: ', a, 1x i0, 1x f0.1, 1x, es6.1e1)", trim(a), b, c, d

! character string and 2 space-delimited values
fmt = '(a, 2(f0.1, 1x))'
print fmt, 'two values: ', c, d
```

```
four values: word 1 2.0 3.0E+0
two values: 2.0 3.0
```

5.3 Command line arguments

Arguments can be passed to a program from the command line using `get_command_argument`. The first argument received by `get_command_argument` is the program executable file name and the remaining arguments are passed by the user. The following program accepts any number of arguments, each at most 32 characters, and prints them.

```
program main
  implicit none

  character(32) :: arg
  integer :: n_arg = 0

  do
     ! get next command line argument
     call get_command_argument(n_arg, arg)

     ! if it is empty, exit
```

```
     if (len_trim(arg) == 0) exit

     ! print argument to screen
     print"('argument ', i0, ': ', a)", n_arg, trim(arg)

     ! increment count
     n_arg = n_arg+1
  end do

  ! print total number of arguments
  print "('number of arguments: ', i0)", n_arg

end program main
```

After compiling to `a.out`, you can pass arguments in the executing command.

```
./a.out 1 2 34

argument 0: ./a.out
argument 1: 1
argument 2: 2
argument 3: 34
number of arguments: 4
```

6 Functions/Subroutines

Functions and subroutines are callable blocks of code. A `function` returns a value from a set of arguments. A `subroutine` executes a block of code from a set of arguments but does not explicitly return a value. Changes to arguments made within a `function` are not returned whereas changes to arguments made within a `subroutine` can be returned to the calling program. Both functions and subroutines are defined after the `contains` keyword in a `module` or `program`.

6.1 Writing a function

The definition of a function starts with the name of the function followed by a list of arguments and return variable. The data types of the arguments and return variable are defined within the `function` body.

6.1.1 Example: `linspace`: generating a set of equally-space points

The following program defines a function `linspace` that returns a set of equidistant points on an interval. The main function makes a call to the function.

```
program main
  implicit none

  real :: xs(10)

  ! call function linspace to set values in xs
  xs = linspace(0.0, 1.0, 10)

  ! print returned value of xs
  print "(10(f0.1, 1x))" , xs

contains

  ! linspace: return a set of equidistant points on an interval
  ! min: minimum value of interval
  ! max: maximum value of interval
  ! n_points: number of points in returned set
  ! xs: set of points
  function linspace(min, max, n_points) result(xs)
    real :: min, max, dx
    integer :: n_points
    integer :: i
    real :: xs(n_points)

    ! calculate width of subintervals
    dx = (max-min) / real(n_points-1)

    ! fill xs with points
    do i = 1,n_points
       xs(i) = min + (i-1)*dx
    end do

  end function linspace

end program main
```

```
.0 .1 .2 .3 .4 .6 .7 .8 .9 1.0
```

6.2 Writing a subroutine

The definition of a subroutine begins with the name of the subroutine and list of arguments. Arguments are defined within the `subroutine` body with one of the following intents

- `intent(in)`: changes to the argument are not returned
- `intent(inout)`: changes to the argument are returned
- `intent(out)`: the initial value of the argument is ignored and changes to the argument are returned.

Subroutines are called using the `call` keyword followed by the subroutine name.

6.2.1 Example: polar coordinates

The following code defines a subroutine `polar_coord` that returns the polar coordinates (r, θ) defined by $r = \sqrt{x^2 + y^2}$ and $\theta = \arctan(y/x)$ from the rectangular coordinate pair (x, y).

```
program main

  real :: x = 1.0, y = 1.0, rad, theta

  ! call subroutine that returns polar coords
  call polar_coord(x, y, rad, theta)
  print*, rad, theta

contains

  ! polar_coord: return the polar coordinates of a rect coord pair
  ! x,y: rectangular coord
  ! rad,theta: polar coord
  subroutine polar_coord(x, y, rad, theta)
    real, intent(in) :: x, y
    real, intent(out) :: rad, theta

    ! compute polar coord
```

```
    ! hypot = sqrt(x**2+y**2) is an intrinsic function
    ! atan2 = arctan with correct sign is an intrinsic function
    rad = hypot(x, y)
    theta = atan2(y, x)

    end subroutine polar_coord

end program main
```

```
1.41421354      0.785398185
```

6.3 Passing procedures as arguments

An `inteface` can be used to pass a function or subroutine to another function or a subroutine. For this purpose, an `interface` is defined in the receiving procedure essentially the same way as the passed procedure itself but with only declarations and not the implementation.

6.3.1 Example: Newton's method for rootfinding

Newton's method for finding the root of a function $f : \mathbb{R} \to \mathbb{R}$ refines an initial guess x_0 according to the iteration rule

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

for $n \geq 1$ until $f(x)$ is less than a chosen tolerance or a maximum number of iterations.

The following code defines a subroutine `newton_root` that returns a root of an input function as well as the number of iterations of Newton's method used to find the root. It is called by the main program to approximate the positive root of $f(x) = x^2 - 2$ from an initial guess $x_0 = 1$.

```
program main
  implicit none

  character(64) :: fmt
  real :: x = 1.0
  integer :: iter = 1000

  ! call newton rootfinding function
```

```
  call newton_root(f, df, x, iter, 1e-6, .true.)

  ! print found root and number of iterations used
  fmt = "('number of iterations: ', i0, ', x: ', f0.7, ', f(x): ', f0.7)"
  print fmt, iter, x, f(x)

contains

  ! function f(x) = x^2 - 2
  function f(x) result(y)
    real :: x, y
    y = x*x - 2
  end function f

  ! function df(x) = 2x
  function df(x) result(dy)
    real :: x, dy
    dy = 2*x
  end function df

  ! newton_root: newtons method for rootfinding
  ! f: function with root
  ! df: derivative of f
  ! x: sequence iterate
  ! iter: max number of iterations at call, number of iterations at return
  ! tol: absolute tolerance
  ! print_iters: boolean to toggle verbosity
  subroutine newton_root(f, df, x, iter, tol, print_iters)

    ! interface to function f
    interface
       function f(x) result(y)
         real :: x, y
       end function f
    end interface

    ! interface to function df
    interface
       function df(x) result(dy)
         real :: x, dy
```

```
       end function df
    end interface

    real, intent(inout) :: x
    real, intent(in) :: tol
    integer, intent(inout) :: iter
    logical, intent(in) :: print_iters
    integer :: max_iters

    max_iters = iter
    iter = 0

    ! while f(x) greater than absolute tolerance
    ! and max number of iterations not exceeded
    do while (abs(f(x))>tol.and.iter<max_iters)
       ! print current x and f(x)
       if (print_iters) print "('f(', f0.7, ') = ', f0.7)", x, f(x)

       ! Newton's update rule
       x = x - f(x)/df(x)

       ! increment number of iterations
       iter = iter + 1
    end do

  end subroutine newton_root

end program main
```

```
f(1.0000000) = -1.0000000
f(1.5000000) = .2500000
f(1.4166666) = .0069444
f(1.4142157) = .0000060
number of iterations: 4, x: 1.4142135, f(x): -.0000001
```

6.3.2 Example: The midpoint rule for definite integrals

The midpoint rule approximates the definite integral $\int_a^b f(x)\,dx$ with integrand $f : \mathbb{R} \to \mathbb{R}$ by

$$\Delta x \sum_{i=1}^{n} f(\bar{x}_i) \tag{1}$$

where $\Delta x = (b-a)/n$, $x_i = a + i\Delta x$ and $\bar{x}_i = (x_{i-1} + x_i)/2$.

The following code defines a function `midpoint` that computes the approximation eq. 1 given a, b, and n. The main program calls `midpoint` to approximate the definite integral of $f(x) = 1/x$ on $[1, e]$ for a range of n.

```
program main
  implicit none

  real, parameter :: E = exp(1.)
  integer :: n
  real :: integral

  ! Approximate the integral of 1/x from 1 to e
  ! with the midpoint rule for a range of number of subintervals
  do n = 2,20,2
     print "('n: ', i0, ', M_n: ', f0.6)", n, midpoint(f, 1.0, E, n)
  end do

contains

  ! function f(x) = 1/x
  function f(x) result(y)
    real :: x, y
    y = 1.0/x
  end function f

  ! midpoint: midpoint rule for definite integral
  ! f: integrand
  ! a: left endpoint of interval of integration
  ! b: right endpoint of interval of integration
  ! n: number of subintervals
  ! sum: approximate definite integral
  function midpoint(f, a, b, n) result(sum)

```

```
    ! interface to f
    interface
       function f(x)
         real :: x, y
       end function f
    end interface

    real :: a, b, min, xi, dx, sum
    integer :: n, i

    ! subinterval increment
    dx = (b-a)/real(n)
    ! minimum to increment from
    min = a - dx/2.0

    ! midpoint rule
    do i = 1,n
       xi = min + i*dx
       sum = sum + f(xi)
    end do
    sum = sum*dx
  end function midpoint

end program main
```

```
n: 2, M_n: .976360
n: 4, M_n: .993575
n: 6, M_n: .997091
n: 8, M_n: .998353
n: 10, M_n: .998942
n: 12, M_n: .999264
n: 14, M_n: .999459
n: 16, M_n: .999585
n: 18, M_n: .999672
n: 20, M_n: .999735
```

6.4 Polymorphism

An `interface` can be used as an entry into two different implementations of a subroutine or function with the same name so long as the different implementations have different argument signatures. This may be particularly useful for defining both a single precision and double precision version of a function or subroutine.

6.4.1 Example: machine epsilon

The following code implements two versions of a function that computes machine epsilon in either single or double precision. The different implementations are distinguished by their arguments. The single precision version `mach_eps_sp` accepts one single precision float and the double precision version `mach_eps_dp` accepts one double precision float. Both functions are listed in the `interface` and can be called by its name `mach_eps`.

```
program main
  implicit none

  integer, parameter :: sp = kind(0.0)
  integer, parameter :: dp = kind(0.d0)

  interface mach_eps
     procedure mach_eps_sp, mach_eps_dp
  end interface mach_eps

  print*, mach_eps(0.0_sp), epsilon(0.0_sp)
  print*, mach_eps(0.0_dp), epsilon(0.0_dp)

contains

  function mach_eps_sp(x) result(eps)
    real(sp) :: x, eps
    integer :: count = 0

    eps = 1.0_sp
    do while (1.0_sp + eps*0.5 > 1.0_sp)
       eps = eps*0.5
       count = count+1
    end do
  end function mach_eps_sp

  function mach_eps_dp(x) result(eps)
```

```
    real(dp) :: x, eps
    integer :: count = 0

    eps = 1.0_dp
    do while (1.0_dp + eps*0.5 > 1.0_dp)
       eps = eps*0.5
       count = count+1
    end do
  end function mach_eps_dp

end program main
```

```
1.19209290E-07   1.19209290E-07
2.2204460492503131E-016   2.2204460492503131E-016
```

6.5 Recursion

A function or subroutine that calls itself must be defined with the `recursive` keyword preceding the construct name.

6.5.1 Example: factorial

The following code defines a recursive function `factorial` that computes $n!$. If $n > 1$, the function call itself to return $n(n-1)!$, otherwise the function returns 1. The main program calls `factorial` to compute 5!.

```
program main
  implicit none

  ! print 5 factorial
  print*, factorial(5)

contains

  ! factorial(n): product of natural numbers up to n
  ! n: integer argument
  recursive function factorial(n) result(m)
    integer :: n, m

    ! if n>1, call factorial recursively
    ! otherwise 1 factorial is 1
```

```
    if (n>1) then
       m = n*factorial(n-1)
    else
       m = 1
    end if

  end function factorial

end program main
```

```
120
```

7 Object-oriented programming

7.1 Derived types

Data types can be defined by the programmer. Variables and procedures that belong to a defined data type are declared between a `type` / `end type` pair. Type-bound procedures, i.e. functions and subroutines, are defined by the `procedure` keyword followed by `::` and the name of the procedure within the `type` / `end type` pair after the `contains` keyword. A variable with defined type is declared with the `type` keyword and the name of the type. The variables and procedures of a defined type variable can be accessed by appending a `%` symbol to the name of the variable.

```
! define a 'matrix' type
! type-bound variables: shape, data
! type-bound procedures: construct, destruct
type matrix
   integer :: shape(2)
   real, allocatable :: data(:,:)
 contains
   procedure :: construct
   procedure :: destruct
end type matrix

! declare a matrix variable
type(matrix) :: mat

! assign value to type-bound variable
mat%shape = [3,3]
```

7.2 Modules

A type-bound procedure can be defined after the `contains` keyword in the same program construct, i.e. a `module`, as the type definition. The first argument in the definition of a type-bound procedure is of the defined type and is declared within the procedure body with the `class` keyword and the name of the type.

```
module matrix_module
  implicit none

  type matrix
     integer :: shape(2)
     real, allocatable :: data(:,:)
   contains
     procedure :: construct
     procedure :: destruct
  end type matrix

contains

  ! construct: populate shape and allocate memory for matrix
  ! m,n: number of rows,cols of matrix
  subroutine construct(this, m, n)
    class(matrix) :: this
    integer :: m, n
    this%shape = [m,n]
    allocate(this%data(m,n))
  end subroutine construct

  ! destruct: deallocate memory that matrix occupies
  subroutine destruct(this)
    class(matrix) :: this
    deallocate(this%data)
  end subroutine destruct

end module matrix_module
```

To define variables of the `matrix` type in the main program, tell it to `use` the module defined above with `use matrix_module` immediately after the `program main` line. The procedures bound to a defined type can be access through variables of that type by appending the `%` symbol to the name of the variable.

```
program main
  use matrix_module
  implicit none

  type(matrix) :: mat
  mat%shape = [3,3]

  ! create matrix
  call mat%construct(3,3)

  ! treat matrix variable 'data' like an array
  mat%data(1,1) = 1.0
  ! etc...

  ! destruct matrix
  call matrix%destruct()
end program main
```

7.3 Example: determinant of random matrix

The following module defines a `matrix` type with two variables: an `integer` array `shape` that stores the number of rows and columns of the matrix and a `real` array `data` that stores the elements of the matrix. The type has four procedures: a subroutine `construct` that sets the shape and allocates memory for the data, a subroutine `destruct` that deallocates memory, a subroutine `print` that prints a matrix, and a function `det` that computes the determinant of a matrix. Note `det` is based on the definition of determinant using cofactors, and is very inefficient. A function `random_matrix` defined within the module generates a matrix with uniform random entries in $[-1, 1]$.

```
module matrix_module
  implicit none

  type matrix
     integer :: shape(2)
     real, allocatable :: data(:,:)
   contains
```

```
     procedure :: construct
     procedure :: destruct
     procedure :: print
     procedure :: det
  end type matrix

contains

  subroutine construct(this, m, n)
    class(matrix) :: this
    integer :: m,n
    this%shape = [m,n]
    allocate(this%data(m,n))
  end subroutine construct

  subroutine destruct(this)
    class(matrix) :: this
    deallocate(this%data)
  end subroutine destruct

  ! print: formatted print of matrix
  subroutine print(this)
    class(matrix) :: this
    ! row_fmt: format character string for row printing
    ! fmt: temporary format string
    character(32) :: row_fmt, fmt = '(a,i0,a,i0,a,i0,a)'
    ! w: width of each entry printed
    ! d: number of decimal digits printed
    integer :: w, d = 2, row
    ! find largest width of element in matrix
    w = ceiling(log10(maxval(abs(this%data)))) + d + 2
    ! write row formatting to 'row_fmt' variable
    write(row_fmt,fmt) '(',this%shape(2),'(f',w,'.',d,',1x))'
    ! print matrix row by row
    do row = 1,this%shape(1)
       print row_fmt, this%data(row,:)
    end do
  end subroutine print

  ! det: compute determinant of matrix
```

```
! using recursive definition based on cofactors
recursive function det(this) result(d)
  class(matrix) :: this
  type(matrix) :: submatrix
  real :: d, sgn, element, minor
  integer :: m, n, row, col, i, j

  m = this%shape(1)
  n = this%shape(2)
  d = 0.0

  ! compute cofactor
  ! if 1x1 matrix, return value
  if (m==1.and.n==1) then
     d = this%data(1,1)
  ! if square and not 1x1
  else if (m==n) then
     ! cofactor sum down the first column
     do row = 1,m
        ! sign of term
        sgn = (-1.0)**(row+1)
        ! matrix element
        element = this%data(row,1)
        ! construct the cofactor submatrix and compute its determinant
        call submatrix%construct(m-1,n-1)
        if (row==1) then
           submatrix%data = this%data(2:,2:)
        else if (row==m) then
           submatrix%data = this%data(:m-1,2:)
        else
           submatrix%data(:row-1,:) = this%data(:row-1,2:)
           submatrix%data(row:,:) = this%data(row+1:,2:)
        end if
        minor = submatrix%det()
        call submatrix%destruct()

        ! determinant accumulator
        d = d + sgn*element*minor
     end do
  end if
```

```
  end function det

  ! random_matrix: generate matrix with random entries in [-1,1]
  ! m,n: number of rows,cols
  function random_matrix(m,n) result(mat)
    integer :: m,n,i,j
    type(matrix) :: mat
    ! allocate memory for matrix
    call mat%construct(m,n)
    ! seed random number generator
    call srand(time())
    ! populate matrix
    do i = 1,m
       do j = 1,n
          mat%data(i,j) = 2.0*rand() - 1.0
       end do
    end do
  end function random_matrix

end module matrix_module
```

The main program uses the `matrix_module` defined above to find the determinants of a number of random matrices of increasing size.

```
program main
  use matrix_module
  implicit none

  type(matrix) :: mat
  integer :: n

  ! compute determinants of random matrices
  do n = 1,5
     ! generate random  matrix
     mat = random_matrix(n,n)

     ! print determinant of matrix
     print "('n: ', i0, ', det: ', f0.5)", n, det(mat)

     ! destruct matrix
```

```
      call mat%destruct()
   end do

end program main
```

```
./main
```

```
n: 1, det: -.68676
n: 2, det: .45054
n: 3, det: .37319
n: 4, det: -.27328
n: 5, det: .26695
```

7.4 Example: matrix module

```
module matrix_module
  implicit none

  public :: zeros
  public :: identity
  public :: random

  type matrix
     integer :: shape(2)
     real, allocatable :: data(:,:)
   contains
     procedure :: construct => matrix_construct
     procedure :: destruct => matrix_destruct
     procedure :: norm => matrix_norm
  end type matrix

  type vector
     integer :: length
     real, allocatable :: data(:)
   contains
     procedure :: construct => vector_construct
     procedure :: destruct => vector_destruct
     procedure :: norm => vector_norm
  end type vector

```

```
! assignments
interface assignment(=)
   procedure vec_num_assign, vec_vec_assign, mat_num_assign, mat_mat_assign
end interface assignment(=)

! operations
interface operator(+)
   procedure vec_vec_sum, mat_mat_sum
end interface operator(+)

interface operator(-)
   procedure vec_vec_diff, mat_mat_diff
end interface operator(-)

interface operator(*)
   procedure num_vec_prod, num_mat_prod, mat_vec_prod, mat_mat_prod
end interface operator(*)

interface operator(/)
   procedure vec_num_quot, mat_num_quot
end interface operator(/)

interface operator(**)
   procedure mat_pow
end interface operator(**)

! functions
interface norm
   procedure vector_norm, matrix_norm
end interface norm

! structured vectors/matrices
interface zeros
   procedure zeros_vector, zeros_matrix
end interface zeros

interface random
   procedure random_vector, random_matrix
end interface random

```

```
contains

  subroutine matrix_construct(this, m, n)
    class(matrix) :: this
    integer :: m,n
    this%shape = [m,n]
    allocate(this%data(m,n))
  end subroutine matrix_construct

  subroutine vector_construct(this, n)
    class(vector) :: this
    integer :: n
    this%length = n
    allocate(this%data(n))
  end subroutine vector_construct

  subroutine matrix_destruct(this)
    class(matrix) :: this
    deallocate(this%data)
  end subroutine matrix_destruct

  subroutine vector_destruct(this)
    class(vector) :: this
    deallocate(this%data)
  end subroutine vector_destruct

  ! assignment
  subroutine vec_num_assign(vec,num)
    type(vector), intent(inout) :: vec
    real, intent(in) :: num
    vec%data = num
  end subroutine vec_num_assign

  subroutine vec_vec_assign(vec1,vec2)
    type(vector), intent(inout) :: vec1
    type(vector), intent(in) :: vec2
    vec1%data = vec2%data
  end subroutine vec_vec_assign

  subroutine mat_num_assign(mat,num)
```

```
    type(matrix), intent(inout) :: mat
    real, intent(in) :: num
    mat%data = num
  end subroutine mat_num_assign

  subroutine mat_mat_assign(mat1,mat2)
    type(matrix), intent(inout) :: mat1
    type(matrix), intent(in) :: mat2
    mat1%data = mat2%data
  end subroutine mat_mat_assign

  ! operations
  function vec_vec_sum(vec1,vec2) result(s)
    type(vector), intent(in) :: vec1, vec2
    type(vector) :: s
    call s%construct(vec1%length)
    s%data = vec1%data + vec2%data
  end function vec_vec_sum

  function mat_mat_sum(mat1,mat2) result(s)
    type(matrix), intent(in) :: mat1, mat2
    type(matrix) :: s
    call s%construct(mat1%shape(1),mat1%shape(2))
    s%data = mat1%data+mat2%data
  end function mat_mat_sum

  function vec_vec_diff(vec1,vec2) result(diff)
    type(vector), intent(in) :: vec1, vec2
    type(vector) :: diff
    call diff%construct(vec1%length)
    diff%data = vec1%data-vec2%data
  end function vec_vec_diff

  function mat_mat_diff(mat1,mat2) result(diff)
    type(matrix), intent(in) :: mat1, mat2
    type(matrix) :: diff
    call diff%construct(mat1%shape(1),mat1%shape(2))
    diff%data = mat1%data-mat2%data
  end function mat_mat_diff

```

```
  function num_vec_prod(num,vec) result(prod)
    real, intent(in) :: num
    type(vector), intent(in) :: vec
    type(vector) :: prod
    call prod%construct(vec%length)
    prod%data = num*vec%data
  end function num_vec_prod

  function num_mat_prod(num,mat) result(prod)
    real, intent(in) :: num
    type(matrix), intent(in) :: mat
    type(matrix) :: prod
    call prod%construct(mat%shape(1),mat%shape(2))
    prod%data = num*mat%data
  end function num_mat_prod

  function mat_vec_prod(mat,vec) result(prod)
    type(matrix), intent(in) :: mat
    type(vector), intent(in) :: vec
    type(vector) :: prod
    call prod%construct(mat%shape(1))
    prod%data = matmul(mat%data,vec%data)
  end function mat_vec_prod

  function mat_mat_prod(mat1,mat2) result(prod)
    type(matrix), intent(in) :: mat1, mat2
    type(matrix) :: prod
    call prod%construct(mat1%shape(1),mat2%shape(2))
    prod%data = matmul(mat1%data,mat2%data)
  end function mat_mat_prod

  function vec_num_quot(vec,num) result(quot)
    type(vector), intent(in) :: vec
    real, intent(in) :: num
    type(vector) :: quot
    call quot%construct(vec%length)
    quot%data = vec%data/num
  end function vec_num_quot

  function mat_num_quot(mat,num) result(quot)
```

```
    type(matrix), intent(in) :: mat
    real, intent(in) :: num
    type(matrix) :: quot
    call quot%construct(mat%shape(1),mat%shape(2))
    quot%data = mat%data/num
  end function mat_num_quot

  function mat_pow(mat1,pow) result(mat2)
    type(matrix), intent(in) :: mat1
    integer, intent(in) :: pow
    type(matrix) :: mat2
    integer :: i
    mat2 = mat1
    do i = 2,pow
       mat2 = mat1*mat2
    end do
  end function mat_pow

  ! functions
  function vector_norm(this,p) result(mag)
    class(vector), intent(in) :: this
    integer, intent(in) :: p
    real :: mag
    integer :: i
    ! inf-norm
    if (p==0) then
       mag = 0.0
       do i = 1,this%length
          mag = max(mag,abs(this%data(i)))
       end do
    ! p-norm
    else if (p>0) then
       mag = (sum(abs(this%data)**p))**(1./p)
    end if
  end function vector_norm

  function matrix_norm(this, p) result(mag)
    class(matrix), intent(in) :: this
    integer, intent(in) :: p
    real ::  mag, tol = 1e-6
```

```
    integer :: m, n, row, col, iter, max_iters = 1000
    type(vector) :: vec, last_vec
    m = size(this%data(:,1)); n = size(this%data(1,:))

    ! entry-wise norms
    if (p<0) then
       mag = (sum(abs(this%data)**(-p)))**(-1./p)
    ! inf-norm
    else if (p==0) then
       mag = 0.0
       do row = 1,m
          mag = max(mag,sum(abs(this%data(row,:))))
       end do
    ! 1-norm
    else if (p==1) then
       mag = 0.0
       do col = 1,n
          mag = max(mag,sum(abs(this%data(:,col))))
       end do
    ! p-norm
    else if (p>0) then
       vec = random(n)
       vec = vec/vec%norm(p)
       last_vec = zeros(n)
       mag = 0.0
       do iter = 1,max_iters
          last_vec = vec
          vec = this*last_vec
          vec = vec/vec%norm(p)
          if (vector_norm(vec-last_vec,p)<tol) exit
       end do
       mag = vector_norm(this*vec,p)
    end if
  end function matrix_norm

  ! structured vectors/matrices
  function random_matrix(m,n) result(mat)
    integer :: m,n
    type(matrix) :: mat
    call mat%construct(m,n)
```

```
    call random_seed()
    call random_number(mat%data)
  end function random_matrix

  function random_vector(n) result(vec)
    integer :: n
    type(vector) :: vec
    call vec%construct(n)
    call random_seed()
    call random_number(vec%data)
  end function random_vector

  function zeros_vector(n) result(vec)
    integer :: n
    type(vector) :: vec
    call vec%construct(n)
    vec = 0.0
  end function zeros_vector

  function zeros_matrix(m,n) result(mat)
    integer :: m,n
    type(matrix) :: mat
    call mat%construct(m,n)
    mat = 0.0
  end function zeros_matrix

  function identity(m,n) result(mat)
    integer :: m,n,i
    type(matrix) :: mat
    call mat%construct(m,n)
    do i = 1,min(m,n)
       mat%data(i,i) = 1.0
    end do
  end function identity

end module matrix_module
```

```
program main
  use matrix_module
  implicit none
```

```

  type(vector) :: vec1, vec2
  type(matrix) :: mat1, mat2
  real :: x
  integer :: i

  ! 0s, id, random
  mat1 = zeros(3,3)
  call mat1%destruct()
  mat1 = identity(3,3)
  mat2 = random(3,3)
  mat1 = mat1*mat1
  vec1 = zeros(3)
  call vec1%destruct()
  vec1 = random(3)
  vec2 = random(3)
  ! +,-,*,/,**
  mat1 = mat1+mat2
  vec1 = vec1+vec2
  mat1 = mat1-mat2
  vec1 = vec1-vec2
  vec1 = mat1*vec2
  mat1 = mat2*mat1
  mat1 = 2.0*mat1
  vec1 = 2.0*vec1
  mat1 = mat1/2.0
  vec1 = vec1/2.0
  mat2 = mat1**3
  ! norm
  x = norm(vec1,0)
  x = norm(vec1,1)
  x = norm(mat1,-1)
  x = norm(mat1,0)
  x = norm(mat1,1)
  x = norm(mat1,2)
  call vec1%destruct
  call vec2%destruct
  call mat1%destruct
  call mat2%destruct
end program main
```

```
./main
```