Chapter 30
Introduction to Submodules

The competent programmer is fully aware of the limited size of his own skull. He therefore approaches his task with full humility, and avoids clever tricks like the plague
Edsger Dijkstra

Aims
The aims of this chapter is to provide a short introduction to submodules.

30.1 Introduction

Modules were introduced into Fortran in the 1990 standard. Over the next ten or so years a number of issues arose that lead to the TR on Enhanced Module Facilities, N1602, which was the starting point for the submodule facility in Fortran. A copy can be found at the WG5 site. Visit

```
https://wg5-fortran.org/
```

to obtain a copy.
The actual published technical report (TR 19767) can be found at the ISO site.

```
https://www.iso.org/standard/37995.html
```

The document discussed the fact that the module system of Fortran was adequate for a wide range of problems, but had shortcomings when one ended up with large modules.
Four areas of concern were identified in this document:

© Springer International Publishing AG, part of Springer Nature 2018
I. Chivers and J. Sleightholme, *Introduction to Programming with Fortran*,
https://doi.org/10.1007/978-3-319-75502-1_30

- Decomposing large and interconnected facilities. If an intellectual concept is large and internally interconnected, it requires a large module to implement it. Decomposing such a concept into components of tractable size using modules may require one to convert private data to public data. One problem occurs during maintenance, when one must then answer the question where is this entity used?
- Avoiding recompilation cascades. Once the design of a program is stable, few changes to a module occur in its interface, that is, in its public data, public types, the interfaces of its public procedures, and private entities that affect their definitions. We refer to the rest of a module, that is, private entities that do not affect the definitions of public entities, and the bodies of its public procedures, as its implementation. Changes in the implementation have no effect on the translation of other program units that access the module. The existing module facility, however, draws no structural distinction between the interface and the implementation. Therefore, if one changes any part of a module, most language translation systems have no alternative but to conclude that a change might have occurred that could affect the translation of other modules that access the changed module. This effect cascades into modules that access modules that access the changed module, and so on. This can cause a substantial expense to re-translate and re-certify a large program. Re-certification can be several orders of magnitude more costly than retranslation.
- Packaging proprietary software. If a module is used to package proprietary software, the source text of the module cannot be published as authoritative documentation of the interface of the module, without either exposing trade secrets, or requiring the expense of separating the implementation from the interface every time a revision is published.
- Easier library creation. Most Fortran translator systems produce a single file of computer instructions and data, frequently called an object file, for each module. This is easier than producing an object file for the specification part and one for each module procedure. It is also convenient, and conserves space and time, when a program uses all or most of the procedures in each module. It is inconvenient, and results in a larger program, when only a few of the procedures in a general purpose module are needed in a particular program.

We provide a brief technical background below and then look at an example based on the date class from the second object oriented chapter.

30.2 Brief Technical Background

The following is taken from Sect. 14.2.3 of the Fortran 2018 standard.

A submodule is a program unit that extends a module or another submodule. The program unit that it extends is its host, and is specified by the parent-identifier in the submodule-stmt.

A module or submodule is an ancestor program unit of all of its descendants, which are its submodules and their descendants. The submodule identifier is the ordered pair whose first element is the ancestor module name and whose second element is the submodule name; the submodule name by itself is not a local or global identifier.

A module and its submodules stand in a tree-like relationship one to another, with the module at the root. Therefore, a submodule has exactly one ancestor module and can have one or more ancestor submodules.

A submodule may provide implementations for separate module procedures (15.6.2.5), each of which is declared within that submodule or one of its ancestors, and declarations and definitions of other entities that are accessible by host association in its descendants.

Here is an example taken from N1602.

The example module POINTS below declares a type POINT and a module procedure interface body for a module function POINT_DIST. Because the interface body includes the MODULE prefix, it accesses the scoping unit of the module by host association, without needing an IMPORT statement; indeed, an IMPORT statement is prohibited.

```
MODULE POINTS

  TYPE :: POINT
    REAL :: X, Y
  END TYPE POINT

  INTERFACE
    REAL MODULE FUNCTION POINT_DIST ( A, B ) &
  RESULT ( DISTANCE )
     TYPE(POINT), INTENT(IN) :: A, B
! POINT is accessed by host association
      REAL :: DISTANCE
    END FUNCTION POINT_DIST
  END INTERFACE

END MODULE POINTS
```

The example submodule POINTS A below is a submodule of the POINTS module. The type POINT and the interface POINT_DIST are accessible in the submodule by host association. The characteristics of the function POINT_DIST are redeclared in the module function body, and the dummy arguments have the same names. The function POINT_DIST is accessible by use association because its module procedure interface body is in the ancestor module and has the PUBLIC attribute.

```
SUBMODULE ( POINTS ) POINTS_A
```

```
    CONTAINS
      REAL MODULE FUNCTION POINT_DIST ( A, B ) &
    RESULT ( DISTANCE )
        TYPE(POINT), INTENT(IN) :: A, B
        DISTANCE = SQRT( (A%X-B%X)**2 + (A%Y-B%Y)**2 )
      END FUNCTION POINT_DIST
  END SUBMODULE POINTS_A
```

A complete example is given below.

30.3 Example 1: Rewrite of the Date Class Using Submodules

In this example we rewrite the base date module to have type declarations and interfaces for each of the contained module procedures.

The submodule will be based on the base date module and will have the implementations of the contained methods.

We have thus effectively decoupled the interface from the implementation.

The stages we followed are

- Duplicate the original module, creating an interface module and a implementation submodule
- Add interfaces for each function and subroutine to the interface module
- Add the new syntax to the interfaces in the module, i.e. add the MODULE keyword to each function and subroutine
- Remove all executable code from the interface module, in this example all code after the contains statement
- Remove all code before the contains statement in the implementation module
- Add the new submodule syntax
- Add the new syntax to each contained procedure, i.e. add the MODULE keyword to each function and subroutine
- Copy the module test program
- Change the test program to use the new module names

We can distribute the module interface, and effectively keep the implementation functions and subroutines hidden.

Here is the first source file. This is the base date class, but now rewritten just to have the interfaces, and no executable or implementation code.

```
module date_module_interface

  use day_and_month_name_module
```

```
implicit none

private

type, public :: date

  private

  integer :: day
  integer :: month
  integer :: year

contains

  procedure, pass (this) :: calendar_to_julian
  procedure, pass (this) :: date_to_day_in_year
  procedure, pass (this) :: &
date_to_weekday_number
  procedure, pass (this) :: get_day
  procedure, pass (this) :: get_month
  procedure, pass (this) :: get_year
  procedure, nopass :: julian_to_date
  procedure, nopass :: &
julian_to_date_and_week_and_day
  procedure, nopass :: ndays
  procedure, pass (this) :: print_date
  procedure, pass (this) :: set_day
  procedure, pass (this) :: set_month
  procedure, pass (this) :: set_year
  procedure, nopass :: year_and_day_to_date

end type date

interface date
  module procedure date_constructor
end interface date

interface
  module function calendar_to_julian(this) &
result (ival)
    implicit none
    integer :: ival
    class (date), intent (in) :: this
  end function calendar_to_julian
end interface
```

```
interface
  type (date) module function &
date_constructor(dd, mm, yyyy)

    implicit none
    integer, intent (in) :: dd, mm, yyyy
  end function date_constructor
end interface

interface
  integer module function &
date_to_day_in_year(this)
    implicit none
    class (date), intent (in) :: this
    intrinsic modulo
  end function date_to_day_in_year
end interface

interface
  integer module function &
date_to_weekday_number(this)
    implicit none
    class (date), intent (in) :: this
    intrinsic modulo
  end function date_to_weekday_number
end interface

interface
  module function get_day(this)
    implicit none
    integer :: get_day
    class (date), intent (in) :: this
  end function get_day
end interface

interface
  module function get_month(this)
    implicit none
    integer :: get_month
    class (date), intent (in) :: this
  end function get_month
end interface
```

```
interface
  module function get_year(this)
    implicit none
    integer :: get_year
    class (date), intent (in) :: this
  end function get_year
end interface

interface
  module function julian_to_date(julian)
    implicit none
    type (date) :: julian_to_date
    integer, intent (in) :: julian
  end function julian_to_date
end interface

interface
  module subroutine &
julian_to_date_and_week_and_day &
(jd, d, wd, ddd)
    implicit none
    integer, intent (in) :: jd
    type (date), intent (out) :: d
    integer, intent (out) :: wd, ddd
  end subroutine &
julian_to_date_and_week_and_day
end interface

interface
  module function ndays(date1, date2)
    implicit none
    integer :: ndays
    class (date), intent (in) :: date1, date2
  end function ndays
end interface

interface
  module function &
print_date(this, day_names, &
short_month_name, digits)
    implicit none
    class (date), intent (in) :: this
    logical, optional, intent (in) :: &
day_names, short_month_name, digits
    character (len=40) :: print_date
    end function print_date
  end interface

  interface
    module subroutine set_day(this, d)
      implicit none
      integer, intent (in) :: d
      class (date), intent (inout) :: this
    end subroutine set_day
  end interface

  interface
    module subroutine set_month(this, m)
      implicit none
      integer, intent (in) :: m
      class (date), intent (inout) :: this
    end subroutine set_month
  end interface

  interface
    module subroutine set_year(this, y)
      implicit none
      integer, intent (in) :: y
      class (date), intent (inout) :: this
    end subroutine set_year
  end interface

  interface
    module function &
  year_and_day_to_date(year, day_in_year)
      use day_and_month_name_module
      implicit none
      type (date) :: year_and_day_to_date
      integer, intent (in) :: day_in_year, year
    end function year_and_day_to_date
  end interface

  public :: calendar_to_julian, &
  date_to_day_in_year, &
    date_to_weekday_number, get_day, &
 get_month, &
 get_year, julian_to_date, &
     julian_to_date_and_week_and_day, &
 ndays, print_date, &
     set_day, set_month, set_year, &
    year_and_day_to_date

  end module date_module_interface
```

Here is the submodule that actually has the implementation.

```
  submodule (date_module_interface) &
    date_module_implementation


  contains

     module function &
       calendar_to_julian(this) &
       result (ival)
       implicit none
       integer :: ival
       class (date), intent (in) :: this

      ival = this%day - 32075 + 1461*&
        (this%year+ &
        4800+(this%month-14)/12)/4 + &
        367*(this%month-2-((this%month- &
        14)/12)*12)/12 - 3*&
        ((this%year+4900+(this% &
        month-14)/12)/100)/4
    end function calendar_to_julian

    type (date) module function &
      date_constructor(dd, mm, &
      yyyy)

      implicit none
      integer, intent (in) :: dd, mm, yyyy

      date_constructor%day = dd
      date_constructor%month = mm
      date_constructor%year = yyyy

    end function date_constructor

    integer module function &
      date_to_day_in_year(this)
      implicit none
      class (date), intent (in) :: this
      intrinsic modulo

      date_to_day_in_year = 3055*&
          (this%month+2)/ &
        100 - (this%month+10)/13*2 - &
          91 + &
        (1-(modulo(this%year,4)+3)/4+&
          (modulo(this% &
        year,100)+99)/100-(modulo(this%year, &
        400)+399)/400)*(this%month+10)/13 + &
        this%day
    end function date_to_day_in_year

    integer module function &
      date_to_weekday_number(this)
      implicit none
      class (date), intent (in) :: this
      intrinsic modulo

      date_to_weekday_number = modulo((13*( &
        this%month+10-&
          (this%month+10)/13*12)-1)/5+ &
        this%day+77+5*(this%year+(this%month- &
        14)/12-(this%year+&
          (this%month-14)/12)/100* &
        100)/4+(this%year+(this%month- &
        14)/12)/400-(this%year+(this%month- &
        14)/12)/100*2, 7)
    end function date_to_weekday_number

    module function get_day(this)
      implicit none
      integer :: get_day
      class (date), intent (in) :: this

      get_day = this%day
    end function get_day

    module function get_month(this)
      implicit none
      integer :: get_month
      class (date), intent (in) :: this

      get_month = this%month
    end function get_month
```

```
module function get_year(this)
  implicit none
  integer :: get_year
  class (date), intent (in) :: this

  get_year = this%year
end function get_year

module function julian_to_date(julian)
  implicit none
  type (date) :: julian_to_date
  integer, intent (in) :: julian

  integer :: l, n

  l = julian + 68569
  n = 4*l/146097
  l = l - (146097*n+3)/4
  julian_to_date%year = (4000*(l+1)/1461001)
  l = l - 1461*julian_to_date%year/4 + 31
  julian_to_date%month = (80*l/2447)
  julian_to_date%day = &
    (l-2447*julian_to_date% &
    month/80)
  l = julian_to_date%month/11
  julian_to_date%month = &
     (julian_to_date%month &
    +2-12*l)
  julian_to_date%year = (100*(n-49)+ &
    julian_to_date%year+1)
end function julian_to_date

module subroutine &
  julian_to_date_and_week_and_day(jd, &
  d, wd, ddd)
  implicit none
  integer, intent (in) :: jd
  type (date), intent (out) :: d
  integer, intent (out) :: wd, ddd

  d = julian_to_date(jd)
  wd = date_to_weekday_number(d)
  ddd = date_to_day_in_year(d)
end subroutine &
```

```
    julian_to_date_and_week_and_day

  module function ndays(date1, date2)
    implicit none
    integer :: ndays
    class (date), intent (in) :: date1, date2

    ndays = calendar_to_julian(date1) - &
      calendar_to_julian(date2)
  end function ndays

  module function &
    print_date(this, day_names, &
    short_month_name, digits)
    implicit none
    class (date), intent (in) :: this
    logical, optional, intent (in) :: &
      day_names, &
      short_month_name, digits
    character (40) :: print_date
    integer :: pos
    logical :: want_day, &
      want_short_month_name, &
      want_digits
    intrinsic len_trim, present, trim

    want_day = .false.
    want_short_month_name = .false.
    want_digits = .false.
    print_date = ' '
    if (present(day_names)) then
      want_day = day_names
    end if
    if (present(short_month_name)) then
      want_short_month_name = short_month_name
    end if
    if (present(digits)) then
      want_digits = digits
    end if
    if (want_digits) then
      write (print_date(1:2), '(i2)') this%day
      print_date(3:3) = '/'
      write (print_date(4:5), '(i2)') &
        this%month
      print_date(6:6) = '/'
      write (print_date(7:10), '(i4)') this%year
    else
      if (want_day) then
        pos = date_to_weekday_number(this)
        print_date = trim(day(pos)) // ' '
        pos = len_trim(print_date) + 2
      else
        pos = 1
        print_date = ' '
      end if
      write (print_date(pos:pos+1), '(i2)') &
        this%day
      if (want_short_month_name) then
        print_date(pos+3:pos+5) = month(this% &
          month)(1:3)
        pos = pos + 7
      else
        print_date(pos+3:) = month(this%month)
        pos = len_trim(print_date) + 2
      end if
      write (print_date(pos:pos+3), '(i4)') &
        this%year
    end if

    return
  end function print_date

  module subroutine set_day(this, d)
    implicit none
    integer, intent (in) :: d
    class (date), intent (inout) :: this

    this%day = d
  end subroutine set_day

  module subroutine set_month(this, m)
    implicit none
    integer, intent (in) :: m
    class (date), intent (inout) :: this

    this%month = m
  end subroutine set_month

  module subroutine set_year(this, y)
    implicit none
    integer, intent (in) :: y
    class (date), intent (inout) :: this

    this%year = y
  end subroutine set_year

  module function year_and_day_to_date(year, &
    day_in_year)
    use day_and_month_name_module
    implicit none
    type (date) :: year_and_day_to_date
    integer, intent (in) :: day_in_year, year
    integer :: t
    intrinsic modulo

    year_and_day_to_date%year = year
    t = 0
    if (modulo(year,4)==0) then
      t = 1
    end if
    if (modulo(year,400)/=0 .and. &
      modulo(year,100)==0) then
      t = 0
    end if
    year_and_day_to_date%day = day_in_year
    if (day_in_year>59+t) then
      year_and_day_to_date%day = &
        year_and_day_to_date%day + 2 - t
    end if
    year_and_day_to_date%month = &
      ((year_and_day_to_date%day+91)*100)/3055
    year_and_day_to_date%day = ( &
      year_and_day_to_date%day+91) - &
      (year_and_day_to_date%month*3055)/100
    year_and_day_to_date%month = &
      year_and_day_to_date%month - 2
    if (year_and_day_to_date%month>=1 .and. &
      year_and_day_to_date%month<=12) then
      return
    end if
    write (unit=*, fmt='(a,i11,a)') &
      '$$year_and_d&
      &ay_to_date: day of the year input &
      &=', day_in_year, ' is out of range.'
  end function year_and_day_to_date
```

```
 end submodule date_module_implementation
```

Here is the Fortran driving program to test the submodule out.

```
include 'day_and_month_name_module.f90'
include 'date_module_interface.f90'
include 'date_module_implementation.f90'

program ch3001

  use date_module_interface , only : &
  calendar_to_julian, &
    date, date_to_day_in_year, &
    date_to_weekday_number, get_day, get_month, &
    get_year, julian_to_date, &
    julian_to_date_and_week_and_day, ndays, &
    print_date, year_and_day_to_date

  implicit none
  integer :: dd, ddd, i, mm, ndiff, wd, yyyy
  integer :: julian
  integer :: val(8)
  intrinsic date_and_time
  type (date) :: date1, date2, x, tx1, tx2

  call date_and_time(values=val)
  yyyy = val(1)
  mm = 10
  do i = 31, 26, -1
    x = date(i, mm, yyyy)
    if (x%date_to_weekday_number()==0) then
      print *, 'Turn clocks  back to EST on: ', &
        i, ' October ', x%get_year()
      exit
    end if
  end do
  call date_and_time(values=val)
  yyyy = val(1)
  mm = 4
  do i = 1, 8
    x = date(i, mm, yyyy)
    if (x%date_to_weekday_number()==0) then
      print *, 'Turn clocks ahead to DST on: ', &
        i, ' April   ', x%get_year()
      exit
    end if
  end do
  call date_and_time(values=val)
  yyyy = val(1)
  mm = 12
  dd = 31
  x = date(dd, mm, yyyy)
  if (x%date_to_day_in_year()==366) then
    print *, x%get_year(), ' is a leap year'
  else
    print *, x%get_year(), ' is not a leap year'
  end if
  x = date(1, 1, 1970)
  call julian_to_date_and_week_and_day &
    (calendar_to_julian(x), x, wd, ddd)
  if (x%get_year()/=1970 .or. x%get_month()/=1 &
    .or. x%get_day()/=1 .or. wd/=4 .or. ddd/=1) &
    then
    print *, &
      'julian_to_date_and_week_and_day failed'
    print *, ' date, wd, ddd = ', x%get_year(), &
      x%get_month(), x%get_day(), wd, ddd
    stop
  end if
  date1 = date(22, 5, 1984)
  date2 = date(22, 5, 1983)
  ndiff = ndays(date1, date2)
  yyyy = 1970

  x = year_and_day_to_date(yyyy, ddd)

  if (ndiff/=366) then
    print *, 'ndays failed; ndiff = ', ndiff
  else
    if (x%get_month()/=1 .and. x%get_day()/=1) &
      then
      print *, 'year_and_day_to_date failed'
      print *, ' mma, dda = ', x%get_month(), &
        x%get_day()
    else
      print *, ' calendar_to_julian OK'
      print *, ' date_ OK'
      print *, ' date_to_day_in_year OK'
      print *, ' date_to_weekday_number OK'
      print *, ' get_day OK'
      print *, ' get_month OK'
      print *, ' get_year OK'
      print *, &
        ' julian_to_date_and_week_and_day OK'
      print *, ' ndays OK'
      print *, ' year_and_day_to_date OK'
    end if
  end if

  tx1 = date(1, 1, 1970)
  julian = tx1%calendar_to_julian()
  tx2 = julian_to_date(julian)
  if (tx1%get_day()==tx2%get_day() .and. &
    tx1%get_month()==tx2%get_month() .and. &
    tx1%get_year()==tx2%get_year()) then
    print *, ' calendar_to_julian and '
    print *, ' julian_to_date worked'
  end if

  x = date(11, 2, 1952)

  print *, ' print_date test'
  print *, ' Single parameter       ', &
    x%print_date()
  print *, &
    ' day_names=false short_month_name=false ', &
    x%print_date(day_names=.false., &
    short_month_name=.false.)
  print *, &
    ' day_names=true  short_month_name=false ', &
    x%print_date(day_names=.true., &
    short_month_name=.false.)
  print *, &
    ' day_names=false short_month_name=true  ', &
    x%print_date(day_names=.false., &
    short_month_name=.true.)
  print *, &
    ' day_names=true  short_month_name=true  ', &
    x%print_date(day_names=.true., &
    short_month_name=.true.)
  print *, ' digits=true            ', &
    x%print_date(digits=.true.)

  print *, ' Test out a month'
  yyyy = 1970
  do dd = 1, 31
    x = year_and_day_to_date(yyyy, dd)
    print *, x%print_date(day_names=.false., &
      short_month_name=.true.)
  end do

end program ch3001
```

As can be seen the test or driving program is identical to the earlier, non submodule version.

30.4 Example 2: Rewrite of the First Order RKM ODE Solver Using Modules

The module `rkm_module` from Chap. 26 contained the `runge_kutta_merson` subroutine which was an implementation of the Runge Kutta Merson (RKM) algorithm.

We have now introduced a submodule called `rkm_module_implementation` which contains the `runge_kutta_merson` subroutine. By moving the body of the procedure into a submodule any subsequent changes to the body will typically only require recompilation of the submodule. Here is the new RKM module code.

```
module rkm_module

interface

  module subroutine &
    runge_kutta_merson(y, fun, ifail, n, a, b, tol)

  use precision_module, wp=> dp

    implicit none

    real (wp), intent (inout), dimension (:) :: y
    real (wp), intent (in) :: a, b, tol
    integer, intent (in) :: n
    integer, intent (out) :: ifail

    interface

      subroutine fun(t, y, f, n)
```

```
        use precision_module, wp => dp
        implicit none
        real (wp), intent (in), dimension (:) :: y
        real (wp), intent (out), dimension (:) :: f
        real (wp), intent (in) :: t
        integer, intent (in) :: n
      end subroutine fun

    end interface

  end subroutine runge_kutta_merson

end interface

end module rkm_module
```

Here is the RKM submodule.

```
submodule (rkm_module) rkm_module_implementation

 contains

  module subroutine &
  runge_kutta_merson(y, fun, ifail, n, a, b, tol)
  use precision_module, wp => dp

!   runge-kutta-merson method for the solution
!   of a system of n 1st order initial value
!   ordinary differential equations.
!   the routine tries to integrate from
!   t=a to t=b with initial conditions in y,
!   subject to the condition that the
!   absolute error estimate <= tol. the step
!   length is adjusted automatically to meet
!   this condition.
!   if the routine is successful it returns with
!   ifail = 0, t=b and the solution in y.

    implicit none

!   define arguments

    real (wp), intent (inout), &
  dimension (:) :: y
    real (wp), intent (in) :: a, b, tol
    integer, intent (in) :: n
    integer, intent (out) :: ifail

    interface
      subroutine fun(t, y, f, n)
        use precision_module, wp => dp
        implicit none
        real (wp), intent (in), &
  dimension (:) :: y
        real (wp), intent (out), &
  dimension (:) :: f
        real (wp), intent (in) :: t
        integer, intent (in) :: n
      end subroutine fun
    end interface

!   local variables

    real (wp), dimension (1:size(y)) :: &
 s1, s2, s3, s4, s5, new_y_1, new_y_2, error
    real (wp) :: &
 t, h, h2, h3, h6, h8, factor = 1.e-2_wp
    real (wp) :: &
 smallest_step = 1.e-6_wp, max_error
    integer :: no_of_steps = 0

    ifail = 0

!   check input parameters

    if (n<=0 .or. a==b .or. tol<=0.0) then
      ifail = 1
      return
    end if

!   initialize t to be start of interval and
!   h to be 1/100 of interval

    t = a
    h = (b-a)/100.0_wp
    do

!     ##### beginning of
!     ##### repeat loop
```

```
        h2 = h/2.0_wp
        h3 = h/3.0_wp
        h6 = h/6.0_wp
        h8 = h/8.0_wp

  !     calculate s1,s2,s3,s4,s5 !
        s1=f(t,y)

        call fun(t, y, s1, n)
        new_y_1 = y + h3*s1

  !     s2 = f(t+h/3,y+h/3*s1)

        call fun(t+h3, new_y_1, s2, n)
        new_y_1 = y + h6*s1 + h6*s2

  !     s3=f(t+h/3,y+h/6*s1+h/6*s2)

        call fun(t+h3, new_y_1, s3, n)
        new_y_1 = y + h8*(s2+3.0_wp*s3)

  !     s4=f(t+h/2,y+h/8*(s2+3*s3))

        call fun(t+h2, new_y_1, s4, n)
        new_y_1 = y + h2*(s1-3.0_wp*s3+4.0_wp*s4)

  !     s5=f(t+h,y+h/2*(s1-3*s3+4*s4))

        call fun(t+h, new_y_1, s5, n)

  !     calculate values at t+h

        new_y_1 = y + h6*(s1+4.0_wp*s4+s5)
        new_y_2 = y + h2*(s1-3.0_wp*s3+4.0_wp*s4)

  !     calculate error estimate

        error = abs(0.2_wp*(new_y_1-new_y_2))
        max_error = maxval(error)
        if (max_error>tol) then

  !       halve step length and try again

          if (abs(h2)<smallest_step) then
            ifail = 2
            return
          end if
          h = h2
        else

  !       accepted approximation so overwrite
  !       y with y_new_1, and t with t+h

          y = new_y_1
          t = t + h

  !       can next step be doubled?

          if (max_error*factor<tol) then
            h = h*2.0_wp
          end if

  !       does next step go beyond interval end b,
  !       if so set h = b-t

          if (t+h>b) then
            h = b - t
          end if
          no_of_steps = no_of_steps + 1
        end if
        if (t>=b) exit

  !     ##### end of
  !     ##### repeat loop

      end do
    end subroutine runge_kutta_merson

  end submodule rkm_module_implementation
```

Here is the `fun1_module`, which is the same code as in Chap. 26.

```
module fun1_module

  implicit none

contains

  subroutine fun1(t, y, f, n)
    use precision_module, wp => dp
    implicit none
    real (wp), intent (in), dimension (:) :: y
    real (wp), intent (out), dimension (:) :: f
    real (wp), intent (in) :: t
    integer, intent (in) :: n
    f(1) = tan(y(3))
    f(2) = -0.032_wp*f(1)/y(2) - &
                0.02_wp*y(2)/cos(y(3))
    f(3) = -0.032_wp/(y(2)*y(2))
  end subroutine fun1

end module fun1_module
```

Here is the main program, which is the same code as in Chap. 26.

```
include 'precision_module.f90'
include 'ch3002_fun1_module.f90'
include 'ch3002_rkm_interface_module.f90'
include 'ch3002_rkm_implementation_module.f90'

program ch3002

  use precision_module, wp => dp
  use rkm_module
  use fun1_module

  implicit none

  real (wp), dimension (:), allocatable :: y
  real (wp) :: a, b, tol
  integer :: n, ifail, all_stat

  print *, 'input no of equations'
  read *, n

! allocate space for y - checking to see that it
! allocates properly

  allocate (y(1:n), stat=all_stat)
  if (all_stat/=0) then
    print *, ' not enough memory'
    print *, ' array y is not allocated'
    stop
  end if
  print *, ' input start and end of interval over'
```

```
    print *, ' which equations to be solved'
    read *, a, b
    print *, 'input initial conditions'
    read *, y(1:n)
    print *, 'input tolerance'
    read *, tol
    print 100, a
  100 format &
      ('at t= ', f5.2, ' initial conditions are :')
    print 110, y(1:n)
  110 format (4(f5.2,2x))
    call &
    runge_kutta_merson(y, fun1, ifail, n, a, b, tol)
    if (ifail/=0) then
      print *, &
   'integration stopped with ifail = ', ifail
    else
      print 120, b
  120 format ('at t= ', f5.2, ' solution is:')
      print 110, y(1:n)
    end if

  end program ch3002
```

30.5 Problems

30.1 Compile and run the above example. Compare the output to the previous version.

30.2 Convert an earlier module example to use submodules, with an interface module and an implementation submodule.

30.6 Bibliography

ISO/IEC DIS 1539-1 Information technology – Programming languages – Fortran – Part 1: Base language

- Fortran 2018 draft standard.

```
  https://www.iso.org/standard/72320.html
```