10. Additional Intrinsic Data Types
Example:

You have written a test driver for SGESV in LAPACK consists of the following steps:

1. Generate the elements of the matrix \([A]\) and the vector \(\{x\}_{\text{exact}}\) using the random number generator.

2. Call the matrix-vector multiplication routine SGEMV in BLAS to compute \(\{B\} = [A] \times \{x\}_{\text{exact}}\).

3. Call the subroutine SGESV in LAPACK to solve the system of equations:
\[ [A] \times \{x\}_{\text{solve}} = \{B\}. \]

Now we would like to know how the error varies with the order of magnitude of the matrix elements, and also the dimension of the linear system. So modify the program you have written to perform the following tests:

1. The random values of the elements in \([A]\) and \(\{x\}\) are within \([-10^m, 10^m]\) where the integer \(m = 0 \sim 10\). The dimension of the linear system remains to be 300 for all cases.

2. The dimension of the linear system is \(2^n\), where the integer \(n = 5 \sim 11\). The random values of the elements are within \([-1, 1]\).

- Find both the maximum error and the root-mean-square error for all cases.
- Use allocatable arrays.
- Output the results in a proper format to show clearly the error property.
Dimension of the linear system is 300
Random values of the elements are within $[-10^m, 10^m]$ for $m = -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10$

<table>
<thead>
<tr>
<th>$m$</th>
<th>max error</th>
<th>rms error</th>
</tr>
</thead>
<tbody>
<tr>
<td>-5</td>
<td>0.28831E-09</td>
<td>0.91865E-10</td>
</tr>
<tr>
<td>-4</td>
<td>0.84292E-08</td>
<td>0.29087E-08</td>
</tr>
<tr>
<td>-3</td>
<td>0.17812E-07</td>
<td>0.64140E-08</td>
</tr>
<tr>
<td>-2</td>
<td>0.23558E-05</td>
<td>0.85145E-06</td>
</tr>
<tr>
<td>-1</td>
<td>0.57295E-05</td>
<td>0.19857E-05</td>
</tr>
<tr>
<td>0</td>
<td>0.54136E-04</td>
<td>0.19027E-04</td>
</tr>
<tr>
<td>1</td>
<td>0.11566E-02</td>
<td>0.40978E-03</td>
</tr>
<tr>
<td>2</td>
<td>0.15106E-01</td>
<td>0.50648E-02</td>
</tr>
<tr>
<td>3</td>
<td>0.84808E-01</td>
<td>0.31613E-01</td>
</tr>
<tr>
<td>4</td>
<td>0.40723E+00</td>
<td>0.14144E+00</td>
</tr>
<tr>
<td>5</td>
<td>0.55625E+01</td>
<td>0.17500E+01</td>
</tr>
<tr>
<td>6</td>
<td>0.76188E+02</td>
<td>0.25888E+02</td>
</tr>
<tr>
<td>7</td>
<td>0.11450E+04</td>
<td>0.41722E+03</td>
</tr>
<tr>
<td>8</td>
<td>0.48460E+04</td>
<td>0.15006E+04</td>
</tr>
<tr>
<td>9</td>
<td>0.31520E+05</td>
<td>0.11665E+05</td>
</tr>
<tr>
<td>10</td>
<td>0.10854E+07</td>
<td>0.40125E+06</td>
</tr>
</tbody>
</table>

Random values of the elements are within $[-1, 1]$ for $n = 5, 6, 7, 8, 9, 10, 11$
Dimension of the linear system is $2^n$

<table>
<thead>
<tr>
<th>$n$</th>
<th>max error</th>
<th>rms error</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0.58413E-05</td>
<td>0.24132E-05</td>
</tr>
<tr>
<td>6</td>
<td>0.16332E-04</td>
<td>0.70644E-05</td>
</tr>
<tr>
<td>7</td>
<td>0.54941E-04</td>
<td>0.18765E-04</td>
</tr>
<tr>
<td>8</td>
<td>0.31970E-04</td>
<td>0.11091E-04</td>
</tr>
<tr>
<td>9</td>
<td>0.10705E-03</td>
<td>0.28957E-04</td>
</tr>
<tr>
<td>10</td>
<td>0.12809E-03</td>
<td>0.39188E-04</td>
</tr>
<tr>
<td>11</td>
<td>0.58836E-03</td>
<td>0.17684E-03</td>
</tr>
</tbody>
</table>
**Errors in numerical computations**

1. errors *intrinsic* to the nature of computers
2. errors caused by inappropriate models, algorithms etc.

**Intrinsic errors:** caused by finite *precision* and *range* of computers

The real or floating point data in a type of scientific notation of base 2 system:

\[
\text{value} = \text{mantissa} \times 2^{\text{exponent}}
\]

(precision) (range)

On most computers,

- **Single-precision real number** occupies 32 bits (4 bytes) of memory:
  
  24 bits mantissa: \( \pm 2^{23} = 8,388,608 \), i.e., about 6 or 7 significant digits
  
  8 bits exponent: \( \pm 2^7 = 128 \), \( 2^{128} \approx 3.4 \times 10^{38} \), \( 2^{-128} \approx 2.9 \times 10^{-39} \)
  
  i.e., range about \( 10^{-38} \sim 10^{38} \)

- **Double-precision real number:** 64 bits (8 bytes):
  
  53 bits mantissa: 14 or 15 significant digits
  
  11 bits exponent: range about \( 10^{-308} \sim 10^{308} \)

\( 12345678.9 \) will be round-off to \( 12345680.0 \) for single-precision real data (7 significant digits)

i.e., round-off error = 1.1
program pi
! The effects of finite precision.

write(*,* ) acos(-1.0)
write(*,10) acos(-1.0)
write(*,* ) acos(-1.0)/1000000.
write(*,10) acos(-1.0)/1000000.
write(*,* ) acos(-1.0)*1000000.
write(*,20) acos(-1.0)*1000000.
write(*,* ) 2./3.
write(*,10) 2./3.
write(*,* ) 2./3./1000000.
write(*,10) 2./3./1000000.
write(*,* ) 2./3.*1000000.
write(*,20) 2./3.*1000000.
10 format(f21.18)
20 format(f16.7)
end program

Output:

Example:

3.1415927
3.141592741012573242
3.1415927E-6
0.000003141592742395
3141592.8
3141592.7500000
0.6666667
0.666666686534881592
6.6666665E-7
0.000000666666664983
666666.69
666666.6875000
Alternative *kinds* of the **real** data type

- Fortran 95 and 2003 Standards guarantee that a compiler supports at least 2 kinds of real variables:
  - single-precision (usually 32 bits, 4 bytes)
  - double-precision (usually 64 bits, 8 bytes)

- Use *kind type parameters* to declare the data types:
  ```fortran
  real(kind=4) :: real_var_single,...
  real(kind=8) :: real_var_double,...
  ```

- Usually, a 32-bit real value is kind=4, and 64-bit is kind=8.

- But, each compiler vendor is free to assign any kind number to any size of real variable, so a 32-bit real value might be kind=1, and 64-bit is kind=2.

- So to make the program more portable between computers and compilers, one can assign kind number to a named constant:
  ```fortran
  integer,parameter :: single=4
  integer,parameter :: double=8
  real(kind=single) :: real_var_single,...
  real(kind=double) :: real_var_double,...
  ```
• Declare the kind of a real constant:
  
  \[
  x = 3.0 \quad ! \text{default kind} \\
  x = 3.0E0 \quad ! \text{single precision} \\
  x = 3.0D0 \quad ! \text{double precision}
  \]

• Intrinsic functions to determine the kind, precision & range of a variable:
  
  \[
  \text{kind(} \text{variable or constant}) \\
  \text{precision(} \text{variable or constant}) \\
  \text{range(} \text{variable or constant})
  \]

Example:

```
program kinds
implicit none
! To determine the kinds of single and double precision
! real values on a particular computer and compiler.
!
write(*,*) kind(0.), precision(0.), range(0.)
write(*,*) kind(0.e0), precision(0.e0), range(0.e0)
write(*,*) kind(0.d0), precision(0.d0), range(0.d0)
end program
```

Output:

\[
4 \quad 6 \quad 37 \\
4 \quad 6 \quad 37 \\
8 \quad 15 \quad 307
\]
Selecting precision in a processor-independent manner

- specify the range and precision required
- use `selected_real_kind` intrinsic function to automatically select the proper kind of real value:
  
  \[
  \text{kind\_number} = \text{selected\_real\_kind}(\ p=\text{prec\_dig}, \ r=\text{range\_exp})
  \]

  \textit{prec\_dig}: precision, number of decimal digits of precision required
  \textit{range\_exp}: range, range of the exponent required in power of 10

Example:

```fortran
program select_kind_1
  ! Example of use of selected_real_kind function

  write(*,*) selected_real_kind(p=6, r=37)  ! (single precision, 4-byte)
  write(*,*) selected_real_kind(p=12)       ! (double precision, 8-byte)
  write(*,*) selected_real_kind(r=100)      ! (10-byte)
  write(*,*) selected_real_kind(r=1000)     ! (range is unavailable)
  write(*,*) selected_real_kind(r=10000)    ! (precision is unavailable)
  write(*,*) selected_real_kind(p=13,200)   ! (both range & precision are unavailable)
  write(*,*) selected_real_kind(p=13)      ! (precision is unavailable)
  write(*,*) selected_real_kind(p=17)      ! (precision is unavailable)
  write(*,*) selected_real_kind(p=63)      ! (both range & precision are unavailable)
  write(*,*) selected_real_kind(p=63, r=10000)
end program
```

Output:

4  
8  
10  
-2  
8  
8  
-1  
-3
program select_kind_2
  implicit none

  integer, parameter :: single = selected_real_kind(p=6,r=37)
  integer, parameter :: double = selected_real_kind(p=14,r=200)
  integer, parameter :: more   = selected_real_kind(p=18)
  integer, parameter :: extra  = selected_real_kind(p=24,r=1000)

  real (kind = single) :: x_s, two_s=2., three_s=3.
  real (kind = double) :: x_d, two_d=2., three_d=3.
  real (kind = more)   :: x_m, two_m=2., three_m=3.
  real (kind = extra)  :: x_e, two_e=2., three_e=3.

  write(*,*) kind(two_s), precision(two_s), range(two_s)
  write(*,*) kind(two_d), precision(two_d), range(two_d)
  write(*,*) kind(two_m), precision(two_m), range(two_m)
  write(*,*) kind(two_e), precision(two_e), range(two_e)

  x_s = two_s/three_s
  write(*,*) single
  write(*,*) x_s

  x_d = two_d/three_d
  write(*,*) double
  write(*,*) x_d

  x_m = two_m/three_m
  write(*,*) more
  write(*,*) x_m

  x_e = two_e/three_e
  write(*,*) extra
  write(*,*) x_e

end program
- **Mixed-mode arithmetic** (not recommended)

  $$1.d0/3. + 1/3 \quad \rightarrow 3.333333333333e-001$$

  $$1./3. + 1.d0/3. \quad \rightarrow 6.666666333333e-001$$

  $$1.d0/3. + 1./3.d0 \quad \rightarrow 6.666666666666e-001$$

- **Double-precision intrinsic function**

<table>
<thead>
<tr>
<th>Single precision</th>
<th>Double precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs(r)</td>
<td>dabs(r)</td>
</tr>
<tr>
<td>cos(r)</td>
<td>dcos(r)</td>
</tr>
<tr>
<td>exp(r)</td>
<td>dexp(r)</td>
</tr>
<tr>
<td>alog(r)</td>
<td>dlog(r)</td>
</tr>
<tr>
<td>alog10(r)</td>
<td>dlog10(r)</td>
</tr>
<tr>
<td>sqrt(r)</td>
<td>dsqrt(r)</td>
</tr>
</tbody>
</table>
**When to use high-precision data type ...**

- Over specify the precision of data type can increase the memory size and slow down execution.

- Calculation range: $> 10^{39}$ or $< 10^{-39}$

- Summation or subtraction of numbers of very different size
  
  $1000000.0 + 3.25 = 1000003.0$ (single)
  
  $= 1000003.25$ (double)

- Subtraction of numbers of nearly equal size (see next section)

- Solving large systems of simultaneous linear equations
  e.g., cumulative round-off errors in Gauss-Jordan elimination
  ➔ the larger the system, the bigger the error
**Example:** Condition of a system of linear equation

<table>
<thead>
<tr>
<th>3.0x − 2.0y = 3.0</th>
<th>x = 1.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.0x + 3.0y = 5.0</td>
<td>y = 0.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1.00x − 1.00y = −2.00</th>
<th>x = 1.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.03x − 0.97y = −2.03</td>
<td>y = 0.5</td>
</tr>
</tbody>
</table>

To compare the sensitivity of the systems, add small errors (1%) to the coefficients:

\[
(3.0 + 0.03)x − 2.0y = 3.0 \quad x = 0.995
\]

\[
5.0x + 3.0y = 5.0 \quad y = 0.008
\]

\[
(1.00 + 0.01)x − 1.00y = −2.00 \quad x = 1.789
\]

\[
1.03x − 0.97y = −2.03 \quad y = 0.193
\]

The solution is not too sensitive to the error in coefficient.

When solving a **ill-conditioned** system, it is necessary to use double-precision arithmetic to reduce cumulative round-off errors.
Exercise

Redo the exercise in the lecture Linear Algebra Computation using both single- and double-precision data types, and calling the corresponding subroutines in LAPACK.

1. Test the results of both data types by increasing the number of equations of the system (for example, 100, 1000, 2500). The random values of the elements in \([A]\) and \(\{x\}\) are within \([-1, 1]\).

2. Test the results of both data types by using random values of the elements in \([A]\) and \(\{x\}\) within \([-10^m, 10^m]\) where the integer \(m = -2 \sim 2\). The dimension of the linear system remains to be 300 for all cases.

3. Measure the CPU time for each case.

4. Design suitable output format to present the results. An example is shown below:

<table>
<thead>
<tr>
<th>number of equations</th>
<th>single precision</th>
<th>double precision</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>max. error</td>
<td>cpu time</td>
</tr>
<tr>
<td>100</td>
<td>.........</td>
<td>.........</td>
</tr>
<tr>
<td>1000</td>
<td>.........</td>
<td></td>
</tr>
<tr>
<td>2500</td>
<td>.........</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>max abs value of matrix</th>
<th>single precision</th>
<th>double precision</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>max. error</td>
<td>cpu time</td>
</tr>
<tr>
<td>0.0001</td>
<td>.........</td>
<td>.........</td>
</tr>
<tr>
<td>0.01</td>
<td>.........</td>
<td></td>
</tr>
<tr>
<td>1.0</td>
<td>.........</td>
<td></td>
</tr>
<tr>
<td>100.0</td>
<td>.........</td>
<td></td>
</tr>
<tr>
<td>10000.0</td>
<td>.........</td>
<td></td>
</tr>
</tbody>
</table>
**COMPLEX Data Type**

- **Complex constant and variable**

  \[
  (1., 0.) \quad 1 + 0i \\
  (0.7071, 0.7071) \quad 0.7071 + 0.7071i \\
  (0, -1) \quad -i \\
  (1.01e6, 0.6e2) \quad 1010000.0 + 60.0i
  \]

  \[
  \text{complex(kind=kind_num)} :: \text{var1, var2, etc.} \\
  \text{complex(kind=kind_num), dimension(n_dim)} :: \text{arr1, arr2, etc.}
  \]

- **Initializing complex variable**

  \[
  \text{complex dimensional(256)} :: \text{array} \\
  \text{array } = (0., 0.)
  \]

  \[
  \text{complex im_i :: im_i=(0., 1.)}
  \]
• Using complex numbers with relational operators

  can use: ==, /=

  cannot use: >, <, >=, <=

• Complex intrinsic function

  Type conversion functions:  CMPLX(a, b, kind)
                            CONJG(z), REAL(z), AIMAG(z),

  Absolute function:  CABS(c)

  Mathematical functions:  CSQRT(z)
                         CEXP(z), CLOG(z)
                         CSIN(z), CASIN(z)