Introduction to FORTRAN

Gang Liu, Hartmut Schmider
gang.liu@queensu.ca, hartmut.schmider@queensu.ca
CAC, Queen's University

Summer School, Compute Ontario
2017

Overview

- Some backgrounds
- FORTRAN elements
- Data types, variables, operations
- Conditional branches
- Repeating constructs
- Code structures: module, program, subroutines, and functions
- Arrays
- Intrinsic functions
- Input, output, and external files
- Application example: π calculation

Some backgrounds

Today's computers

- are very powerful and clever
- e.g. can help doctors to diagnose patient's problems (IBM Watson)
- e.g. can win world top players in board game GO (AlphaGo)
- however, machines/mechanics
- always need instruction to do next
- the instructions must be accurate, detailed, and complete
- computer code is for that purpose.

Programming/coding principle

- Logical
- Then feasible, no conflicts, no ambiguity.
- E.g. not cooking and playing volleyball at the same time.
- E.g. in a many-road intersection, can not ask one to walk some steps without indicating direction.
- E.g. not try to present your "fourth" apple to your friend when you have only three.
- E.g. not try to divide 92 by "Please accept this cruise for our anniversary!"

How to learn coding

- Testing
- Testing
- And testing …

Computers of bits

- Almost everything in computers is bit or bits.
- Each bit can be imagined as a simple circuit with current running or not, two states only.
- Normally the two states are represented with 0 and 1.
- Then do you mean a computer can only describe two states?
- No. Each bit can do that. But we have many many bits.
- E.g. 01011101100010001 may mean "I love you!"
- Actually unlimited number of bits can express anything.
- All data and code instructions are bits.
- And anything computers do is on bits essentially.

Minimum working unit in computers

- Not simply one bit, but
- 8 bits
- Called one BYTE.
- This means whatever you do anything, one or more BYTEs will be used.

What we get when we buy a computer?

Laptops & MacBooks

HP 15.6" Touchscreen Laptop (AMD A9-9410 7th Generation/1 TB HDD/8 GB RAM/Windows 10 Home) - Black

$499.99

Save: $200

Sale Ends: June 29, 2017

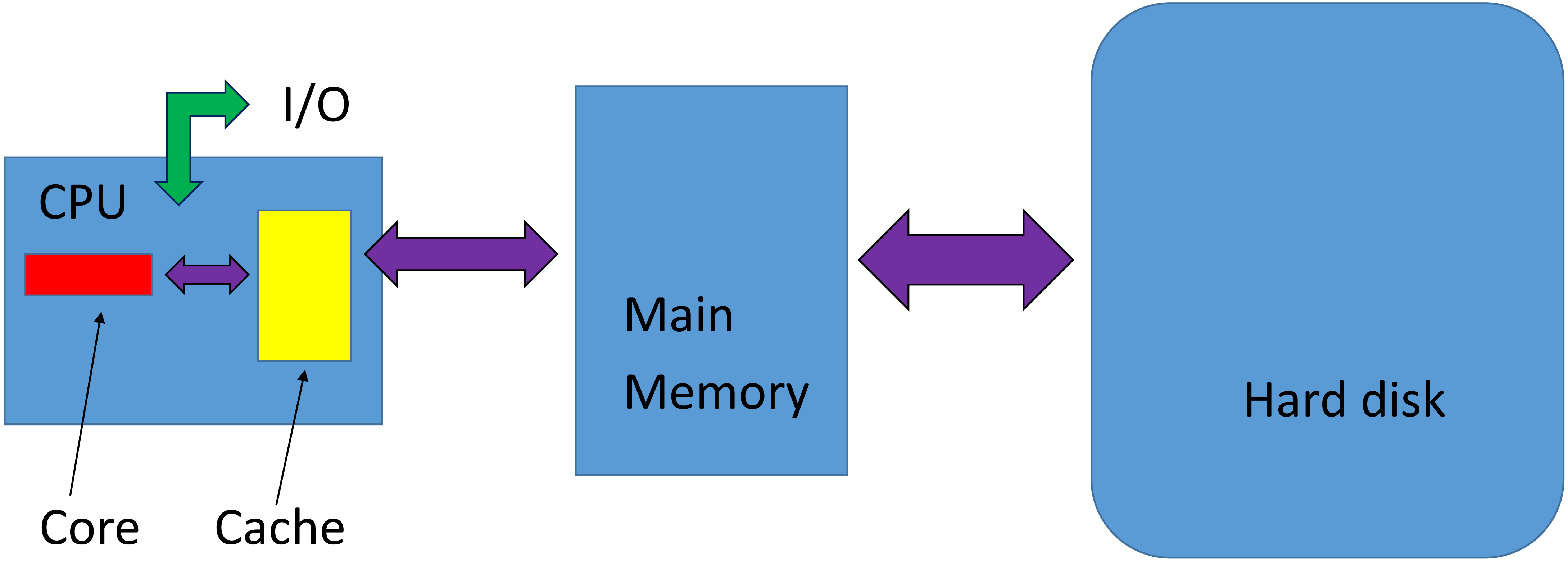
Shop Now ›

Computer key parts

- CPU. E.g. AMD A9-9410 2.9GHz
- Memory. 8GB is about 8,000,000,000 BYTES
- Hard disk: 1TB is about 1,000,000,000,000 BYTES

Accurately		
1KB	= 2^10 BYTES	= 1024 BYTES
1MB	= 2^10 KB	= 1024 KB
1GB	= 2^10 MB	= 1024 MB
1TB	= 2^10 GB	= 1024 GB
1PB	= 2^10 TB	= 1024 TB

A sketch of computer structure and data flow



CPU/Core can operate data only in cache at fixed frequency. Much additional time is spent in data transporting between the main memory and the cache. Fully making use of cache capacity reducing data movement between main memory and cache is critical for performance improvement.

Data and code

- are stored in hard disks as files in certain format.
- Files are placed in a hierarchy of directories.
- Although everything is bits/BYTEs, some files are stored in a way such that the BYTEs can be converted into characters, letter, numbers, and/or other symbols, then readable to people. Called text files.
- Other files are just bits, not indented to be converted. Called binary files. People can not read binary files. Meanwhile, computers can not run based on text files, but based on instructions in some binary files, called executable. Not all binary files are executable, e.g. movie data files.

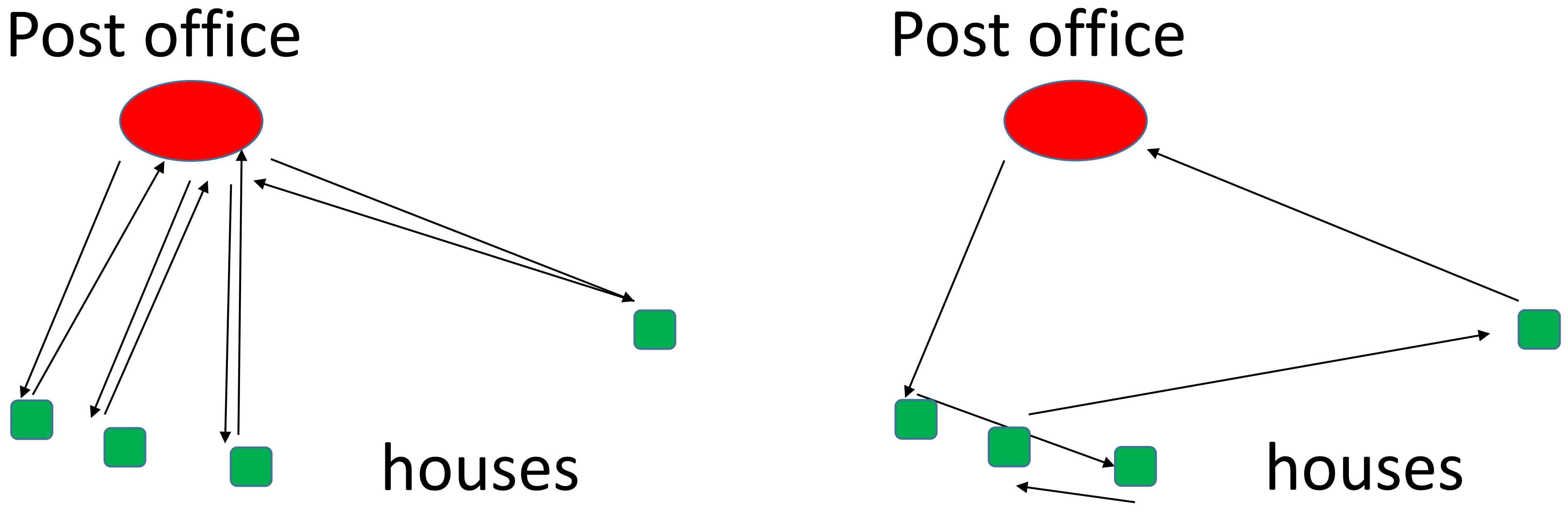
Computer languages

- Rules and facilities for writing (source) codes in text files, eventually converted into executable binaries to instruct computers what to do.
- Although they were created like natural languages as much as possible, their rules (syntax/grammar) are applied absolutely strictly. Any violation will be refused.
- There are a great number of computer languages.
- For application, especially computing/data science, high-level programming languages fall into two categories.
- Interpreted and compiled languages.

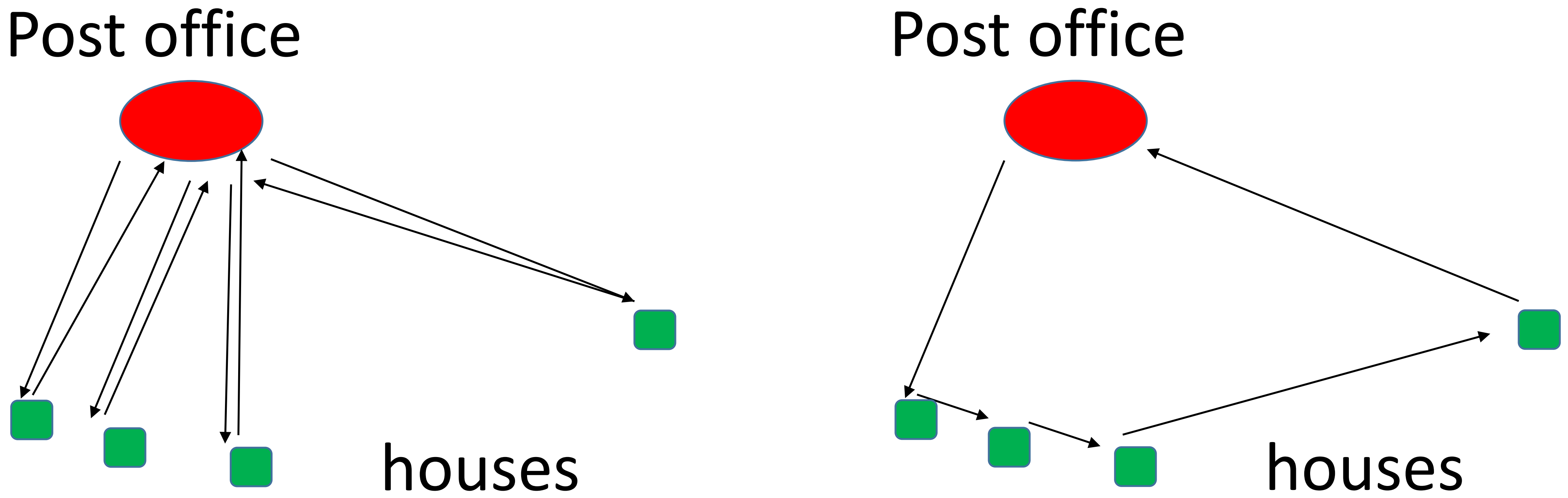
The difference

- In interpreted languages, like R, Python, and Matlab, one or more lines of source code is (are) converted into binary then executed. Then another section of source code. This procedure is repeated till end. Then computation is interrupted by conversions.
- In compiled languages, like C and FORTRAN, the whole source code is compiled into a big executable binary code. The compiled binary code can be run repeatedly later, forgetting the source code.

The difference



The difference



The interpreted, like a postman on the left, needs to go back and forth to deliver letters. The compiled (on the right) can deliver all together and further optimize since knows all tasks.

So

The compiled ones usually run much fast than the interpreted ones.

FORTRAN

is for fast (maybe also for easy) scientific computation on purpose.

Then it supports features for such purpose as much as possible, and in principle refused other feature. Sure this is different from C/C++, which likes to be most powerful.

As a compiled language, FORTRAN

was proposed and developed by John W. Backus's team in IBM, late 1953

(a contraction of FORmula TRANslation)

Versions	Year	Versions	Year
FORTRAN	1957	Fortran 90	1991
FORTRAN II	1958	Fortran 95	1997
FORTRAN III	1958	Fortran 2003	2004
FORTRAN IV	1961	Fortran 2008	2010
FORTRAN 66	1966	Fortran 2015	2018?
FORTRAN 77	1978	Fortran ???????	

Fortran 90 is a big jump

which makes FORTRAN a modern language.

We will talk about basic Fortran 90, which should be good for all later versions as well.

FORTRAN elements

Fortran character set

- the 26 letters of the English alphabet (case insensitive)
- the 10 Arabic numerals, 0 to 9
- the underscore, _,
- (all above is called alphanumeric characters)
- and the ones in the table of the next page

Special characters of Fortran

Character	Name	Characer	Name
=	Equals sign	:	Colon
+	Plus sign		Blank
-	Minus sign	!	Exclamation mark
*	Asterisk	"	Quotation mark
/	Slash	%	Percent
(	Left parenthesis	&	Ampersand
)	Right parenthesis	;	Semicolon
,	Comma	<	Less than
.	Decimal point	>	Greater than
$	Currency symbol	?	Question mark
'	Apostrophe		

Source form

- Source code is made of lines
- Each line may contain up to 132 characters
- (For versions before 90 up to 72)
- Each line usually contains one statement, e.g.
- x = (-y + root_of_discriminant)/(2.0*a)
- Anything after "!" in a given line is a comment, like
- ! Here is a comment to make some notes.
- x = y/a - b ! Solve the linear equation
- (For versions before 90, the first character must be "C" for comments)

Source form

If a line ends with "&", the next line is a continuation, thus a very long statement/line can be split into more lines, but limited to 39.

```
x  =                                                  &
     (-y + root_of_discriminant)                      &
     /(2.0*a)
```

If the first non-blank character in the next line is "&", those blank characters and the "&" are ignored:

```
x  =                                                  &
    & (-y + root_of_discriminant) / (2.0*a)
```

(For versions before 90, continuation lines are identified by a nonblank, nonzero character in column 6)

More than one short statements can be written into one line, separated with ";", e.g.

```
a = 3.5 ;  b = c + d ;  speed = distance / time
```

Names

Programmers need to create/name many names, especially variable names.

All names must consist of between 1 and 31 alphanumeric characters of which the first must be a letter.

A, a, ggg, f3d, alpha, KING8, second_generation, TRY_003 are all legal.

However,

delete them

2 students

$20

are all illegal.

Compiling and running

A Fortran source code must be compiled before running with a compiler.
Supposing a file called myfortran01.f90 contains

```
    program mywork
        print*, "My work is finished."
    end program mywork
```

```
ls -ltr
f90         myfortran01.f90
ifort       myfortran01.f90
gfortran    myfortran01.f90
ls -ltr
./a.out
```

Compiling many files for one executable

Supposing myfortran01.f90, myfortran02.f90, myfortran03.f90 are source

```
ifort      myfortran01.f90   myfortran02.f90   myfortran03.f90
./a.out
```

Compiling many files for one executable

```
ifort    -c    myfortran01.f90
ifort    -c    myfortran02.f90
ifort    -c    myfortran03.f90
ls -ltr
rm a.out
ifort   -o myexe  myfortran01.o  myfortran02.o  myfortran03.o
ls -ltr
./myexe
```

Compiling many files for one executable

```
ifort   -c   myfortran01.f90
ifort   -c   myfortran02.f90
ls -ltr
rm a.out
ifort   -o myexe  myfortran01.o  myfortran02.o  myfortran03.f90
ls -ltr
./myexe
```

Compiling many files for one executable

```
ifort    -c   -O3  myfortran01.f90
ifort    -c   -O3  myfortran02.f90
ifort    -c   -O3  myfortran03.f90
ls -ltr
rm a.out
ifort    -o myexe -O3 myfortran01.o  myfortran02.o  myfortran03.o
ls -ltr
./myexe
```

Data Types, Variables, and Operations

Data Types

- What and why types?
- Each type was defined, so that a reasonable amount of bits or BYTEs allocated in memory to store such values with certain format.
- Once a data type is defined, we can use it, forgetting its details.
- FORTRAN has the following types defined in the language as intrinsic types. Programmers can define more combined data types for their own projects.

Intrinsic Types

- INTEGER
- REAL
- DOUBLE PRECISION
- COMPLEX
- DOUBLE COMPLEX
- LOGICAL
- CHARACTER

INTEGER Type

- 2
- 5
- INTEGER :: I, J, K
- INTEGER :: LIMIT, SEAT_NUMBER = 34
- INTEGER, PARAMETER :: NUMBER_OF_MONTHS_A_YEAR = 12

INTEGER Type

- is for one integer.
- normally has 64 bits (8 BYTEs) in memory.
- one bit is used for sign, maybe 1 meaning +, 0 for -.
- the rest 63 bits are for the actual integer absolute value.
- the maximum positive value can be stored is 2**63 -1 = 9,223,3…,807
- Usually one would assume the type is big enough for holding values we are interested, then forget all above details. In case an integer is bigger than the type can express, an overflow error will be reported then the code running will be stopped automatically. All other types similar.

REAL Type

- 2.0
- 5.4e12
- REAL :: R, RG, SPEED = 90.0, DISTANCE
- REAL, PARAMETER :: MASS_OF_THE_OBJECT = 1.45

- A REAL usually has 32 bits (4 BYTEs) in memory (single precision).

DOUBLE PRECISION Type

- 2.0d0
- 5.4d12
- DOUBLE PRECISION :: R, RG, SPEED = 90.0d0

- A DOUBLE PRECISION usually has 64 bits (8 BYTEs) in memory (double precision).

COMPLEX Type

- (1.0, 2.0)
- (4.6e22, 5.4e12)
- COMPLEX :: R, RG, SPEED = (0.0, 90.0)

- A COMPLEX usually has 64 bits (8 BYTEs) in memory (two single precision).

DOUBLE COMPLEX Type

- (1.0d0, 2.0d0)
- (4.6d-22, -5.4d12)
- DOUBLE COMPLEX :: R, RG, SPEED = (0.0d0, 90.0d0)

- A DOUBLE COMPLEX usually has 128 bits (16 BYTEs) in memory (two double precision).

Although a kind number can be further specified

- INTEGER (KIND = 2)
- INTEGER (KIND = 4)
- REAL (KIND = 4)
- DOUBLE PRECISION (KIND = 4)
- COMPLEX (KIND = 6)
- DOUBLE COMPLEX (KIND =8)

- I personally do not suggest it, as kind numbers are usually processor or compiler dependent, rather than universal applicable.

Other ways to specify BYTEs

- INTEGER*8
- REAL*8 !double precision
- REAL*16 !quadruple precision
- COMPLEX*16 !double precision
- COMPLEX*32 !quadruple precision

- which are good enough for most scientific computations and universally applicable.

Implicit rules

Fortran adopted an implicit rule for long period of time. It is called I-N rule, which means any variable beginning with I, J, K, L, M, N are assumed as integer type, otherwise real type. Then in such a case, variables can be used directly without declaration. This rule saved some typing effort for programming. But later people realized that even very heavy typing is ignorable compared with other effort in programming, e.g. debugging. Explicitly defining variables help programming to reduce errors, so more preferred now. In order to get rid of the implicit rule, the statement

```
IMPLICIT NONE
```

can be used.

LOGICAL Type

- Only has two possible values
- .TRUE.
- .FALSE.
- .TRUE. may also be denoted as T or 1
- .FALSE. may be denoted as F or 0.

- LOGICAL :: A1 = .TRUE., FFF = .FALSE.

CHARACTER Type

- "s"
- "Kingston is a great place to visit!"
- CHARACTER (LEN=11) :: AAA
- CHARACTER (LEN=11) :: AA1 = 'KINGSTON'
- CHARACTER (LEN=11) :: AAB = "KINGSTON IS"
- CHARACTER (LEN=11) :: B_2 = "KINGSTON IS GREAT!"
- CHARACTER (LEN=*), PARAMETER   :: &
-   BB_A = "Kingston is a great place to visit!"

TEST IT with file t1.f90

```
PROGRAM A_TEST
CHARACTER (LEN=11) :: AAA
CHARACTER (LEN=11) :: AA1 = "KINGSTON"
CHARACTER (LEN=11) :: AAB = "KINGSTON IS"
CHARACTER (LEN=11) :: B_2 = "KINGSTON IS GREAT!"
CHARACTER (LEN=*), PARAMETER   :: BB_A = "Kingston is a great place to visit!"
PRINT*, AA1
PRINT*, AAB
PRINT*, B_2
PRINT*, BB_A
STOP
END PROGRAM A_TEST
```

```
Ifort t1.f90
./a.out
```

Variables

- Variables are essentially the variables of mathematics.
- However they must be defined/declared as fixed data types at the beginning of the code unit where they will be assessed.
- Then the compiler will allocate enough space in memory for them, and of course specify the format to hold data.
- Must be initialized with values before being read. The values can be changed and read unlimitedly later. But the data type can not be changed.

Basic operations on numbers

```
Program test_02
integer  ::  i1,  i2, i3
real        ::  rl1, rl2, rl3
i1 = 4;  i2 = 3
i3 = i1 + i2
Print*, i1, i2, i3, i1-i2, i1*i2, i1/i2, i1**i2
rl1 = 4;  rl2 = 3
rl3 = rl1 + rl2
Print*, rl1, rl2, rl3, rl1-rl2, rl1*rl2, rl1/rl2, rl1**rl2
Stop
End Program test_02
```

Integer divisions

```
Program test_03
Print*, "Integer divisions: ", 1/3, 2/3, 3/3, 4/3, 5/3, 6/3
Print*, "Real divisions: ", 1.0/3, 2/3.0, 3./3, 4.0/3.0, 5./3., 6.0/3
Stop
End Program test_03
```

Operations on different types

```
Program test_04
Double precision :: dp = 4.5d0
Print*, 1 + 2.9, 3.0 + 3, 1 + dp, 4.3 + dp
Stop
End Program test_04
```

Integer => real or double precision

Low precision => high precision

Assignment with different types

```
Program test_06
Integer :: ii = 4, ii2, ii3
Real :: rr =5.123456789123456789e0, rr2, rr3
Double precision :: dp = 6.123456789123456789d0, dp2, dp3
ii2 = rr;  ii3 = dp
Print*, ii2, ii3
rr2 = ii;  rr3 = dp
Print*, rr2, rr3
dp2 = ii;  dp3 = rr
Print*, dp2, dp3
Stop
End Program test_06
```

Always converted into the type of the variable.

Precedence

```
Program test_07
Print*, 1+2-3, 1-3+2, 1+2*3, 1*2+3
Print*, 4*2/3, 4/2*3, 4*(2/3), 4/(2*3)
Print*, 2**3*5, 9*3**2
Stop
End Program test_07
```

Precedence:

** exponentiation

* / multiplication, division

+ - addition, subtraction

Use parentheses ( ) to change or make sure precedence.

Operations on logical variables

```
Program test_09
Print*, .not. .true.,  .not.  .false.
Print*, .true. .and.  .true.,   .true. .and.  .false.
Print*, .true. .or.  .true.,   .true. .or.  .false.
Print*, .true. .eqv.  .true.,   .true. .eqv.  .false.
Print*, .true. .neqv.  .true.,   .true. .neqv.  .false.
Stop
End Program test_09
```

Data comparisons resulting in logical values

```
Program test_10
Print*, 5 > 6,    7 < 8
Stop
End Program test_10
```

.lt.	or	<	less than
.le.	or	<=	less than or equal
.eq.	or	==	equal
.ne.	or	/=	not equal
.gt.	or	>	great than
.ge.	or	>=	great than or equal

Operations on characters

```
Program test_12
print*, "Queen's Univer"//"sity is in Kingston."
Stop
End Program test_12
```

Concatenation

Character substrings

```
Program test_14
Implicit none
Character (len=40) :: a_string = "Queen's University is in Kingston."
Print*, a_string
Print*, a_string(:6)
Print*, a_string(8:)
Print*, a_string(4:12)
Print*, a_string(9:9)
Stop
End Program test_14
```

Derived data types

```
type student
        character (len=20) :: name
        integer            :: id
        integer            :: grade
        real               :: age
        real               :: height
end type student

type(student)   ::   a_queens_student

a_queens_student%age
```

Conditional branches

IF statement

if (a condition is true)   (do something)

```
if (iii > 9) print*, iii
```

GO TO statement

- An example

```
if  (a condition  is  true)   go to 25
    (other statements)
25   c = a + b
    (other statements)
```

- Not suggested.

IF Construct

- It allows us to code sections of code to be executed under conditions.
- Most cases, different conditions execute different part/path of code.
- The general form

```
if  (condition1  is  true)   then
    (do some things accordingly)
else if  (condition2   is   true)    then
      (do some other things accordingly)
else if  (condition3   is   true)    then
      (do some other things accordingly)
else if  (...   is true)  then
       (do some other things accordingly)
else
       (do other things accordingly)
end if
```

- All else if and else constructs are optional and no limit on number of else if constructs.

IF Construct

- An example

```
var = 5
if  (var > 10)  then
    print*, "var has value ",  var, " and bigger than 10. "
else if  (var < 10) then
    print*, "var has value ",  var, " and less than 10. "
else
   print*, "var has value ",  var, " and is 10. "
end if
```

IF Construct

- can be unlimited nested.

- Example

```
if (my_dad_is_at_home) then
        (my dad will cook)
else
        if (my_mom_is_at_home) then
              (my mom will cook)
        else
              (I will cook)
        end if
end if
```

CASE Construct

- Example

```
select case (an_integer)
case (:-1)
        (do things accordingly)
case (0)
        (do things accordingly)
case (1:7)
        (do things accordingly)
case (8:22)
        (do things accordingly)
case default
        (do things accordingly)
end select
```

Repeating constructs

If you have

- log(2.0) + log(3.0) + log(4.0) + log(5.0) + … + log(10000.0) to calculate,
- you would absolutely not like to write all such operations one by one.
- Loops are for such repeated works and can make life much easier.

It can be done by a do loop

```
program test_20
implicit none
integer*8 :: i
real*8       :: result
result = 0.0d0
do i = 2, 10000
      result = result + log(i * 1.0d0)
end  do
print*, result
stop
end program test_20
```

The do loop

- takes a form of

```
[name:]  do an_integer = beginning, ending, step
        (do something for the iteration)
end do [name]
```

- Example

```
integer :: a
do a = 2001, 2017
        print*, a, " is a year of this centrury. "
end do
```

CYCLE statement in do loop

- integer :: ii

  do ii = 1, 1000, 2

  (do something for the iteration)

  end do

- is equivalent to

```
ONLY_FOR_ODD_NUMBERS:   do ii = 1, 1000
        if ( (ii/2) == 0) cycle ONLY_FOR_ODD_NUMBERS
        (do something for the iteration)
end do ONLY_FOR_ODD_NUMBERS
```

EXIT statement in do loop

```
UNLIMITED_LOOP:   do
        if (some condition is met) exit UNLIMITED_LOOP
        (do something for the iteration)
    end do UNLIMITED_LOOP
```

Loops can be unlimited nested

```
program test_22
implicit none
integer :: i, j
do i = 1, 9
     do j = 1, 9
        print*, "The multiplication of ",i," and ",j," is ",i*j
     end do
end do
stop
end program test_22
```

Arrays

An array

- consists of a rectangular set of elements (scaler variables), all of the exactly same type.
- real, dimension(10) :: a
- then the successive elements of the array are a(1), a(2), …, a(10)
- real, dimension(-10:20) :: vt
- the elements are vt(-10), vt(-9), vt(-8), …, vt(20)
- real, dimension(5, 4) :: b
- then the successive elements of the array in memory are

  b(1, 1), b(2, 1), b(3, 1), b(4, 1), b(5, 1),

  b(1, 2), b(2, 2), b(3, 2), b(4, 2), b(5, 2),

  b(1, 3), b(2, 3), b(3, 3), b(4, 3), b(5, 3),

  b(1, 4), b(2, 4), b(3, 4), b(4, 4), b(5, 4)

they can also declared as

real :: a(10), vt(-10:20), b(5,4)

Arrays should be initialized before being used

- real :: a(10), vt(-10:20), b(5,4)
- a = 0.0d0
- vt = 0.0d0
- b = 0.0d0

- character (len=20) :: aaa(25,300)
- aaa = ' '

Subarrays and collective operations

```
real :: a(10),  vt(-10:20),  b(5,4)
vt(-2:2) = (/1.4, 3.6, 7.3, 8.9, 13.8/)
vt(:2) = 9.0
vt(3:) = 25.0
vt(-8:2:3) = 64.0
b = 24.0
a(2:5) = vt(-3:0) + b(1:4, 3)
```

Elemental operations on arrays

```
real :: b(5,4)
integer :: i, j
do i = 1, 4
     do j = 1, 5
          b (j, i) = (sin(i*1.0))**j
     end do
end do
```

Elemental operations on arrays

```
real :: b(5,4), c(4, 6), d(5,6)
integer :: i, j, k
b = 3.4
c = 5.9
d = 0.0
do i = 1, 6
    do j = 1, 5
        do k = 1, 4
            d (j, i) = d(j, i) + b(j, k) * c(k, i)
        end do
    end do
end do
```

Allocatable arrays

- Array sizes may not be known until run time

```
real, allocatable  :: arr1(:), arr2(:, :),  array8(:, :, :, :)
integer :: i=3, j=5, k=6, l=8
…
allocate(arr1(i))
allocate(arr2(k,j))
allocate(array8(i,j,k,l))
…
deallocate(arr1, arr2, array8)
```

Code structures:
modules, program, subroutines, and functions

Nowadays, people do everything step by step

Programming is not an exception

- A big computational task is usually cut into many smaller ones.
- With input and output assumed to some degrees, what and how to complete the inside of each smaller task is usually independent on any other smaller tasks. In other words, the inside is encapsulated.
- Functions/subroutines are used for such smaller tasks or steps.
- Normally when we code functions/subroutines, we can focus on the specific small task, forgetting the whole big complicated task.
- Functions/subroutines can be unlimitedly re-used and make code well structured.
- Additionally, modules are a great help in many respects.
- Functions/subroutines can only access their arguments, local variables, and data in the modules whey use.

The general form of functions

```
[type] function function_name(dummy_argument_list)
    ! function body (statements)
    ! function_name is the variable to be assigned new value
    ! to send back.
    return
end function function_name
```

A function example

```
real*8 function distance(x1, y1, x2, y2)
    real*8 :: x1, y1, x2, y2
    distance = sqrt ((x1 - x2)**2 + (y1 - y2)**2)
    return
end function distance

program test_30
    interface
        real*8 function distance(x1, y1, x2, y2)
            real*8 :: x1, y1, x2, y2
        end function distance
    end interface
    real*8 :: a1, b1, a2, b2
    a1 = 2.3d0;  b1 = 5.3d0;   a2 = 3.2d2;   b2 = 6.3d4
    print*, distance(a1, b1, a2, b2)
end  program test_30
```

The return data type can also be written as

```
function distance(x1, y1, x2, y2)
      real*8 :: x1, y1, x2, y2, distance
      distance = sqrt ((x1 - x2)**2 + (y1 - y2)**2)
      return
 end function distance

 program test_30
     interface
          function distance(x1, y1, x2, y2)
                   real*8 :: x1, y1, x2, y2, distance
           end function distance
     end interface
     real*8 :: a1, b1, a2, b2
     a1 = 2.3d0;  b1 = 5.3d0;   a2 = 3.2d2;   b2 = 6.3d4
     print*, distance(a1, b1, a2, b2)
 end  program test_30
```

Normally, when a function is called

all arguments should be provided in sequence, like

```
real*8  function my_functioin(ar1, ar2, ar3, … arn)
  …
end function my_functioin

aa = my_functioin(act1, act2, act3, … actn)
```

Argument names can also be used when a function is called

```
real*8  function my_function(ar1, ar2, ar3, … arn)
  …
end function my_function


aa = my_function(act1, act2, arn = act3, … ar3 = actn)
```

then the argument name is called keyword. Once a keyword argument is used, no more pure positional arguments allowed after it.

Optional arguments in functions

should be declared at the end of argument list

```
real*8  function my_function(ar1, ar2, ar3, … arn, p1, p2, …, pn)
      real*8, optional :: p1, p2, …, pn
      if (present(p1)) then
          …
      end if
      …
end function my_function

aa = my_function(act1, act2, act3, … actn, 3.5d0)
```

Functions as arguments in functions

```
real*8  function minimum(ar1, ar2, a_function)
    real*8 :: ar1, ar2
    interface
        real*8  function a_function(x)
                real*8 :: x
        end function a_function
    end interface
    …
end function minimum
```

```
real*8  function  pain(x)
          real*8 :: x
          pain = 72.0d0 + 3.0d0*x + 5.0d0*x**2
end function pain


aa = minimum(10.d0, 100.d0, pain)
```

Actually

```
program program_name
    …
end  program program_name
```

is also a function, but called the unique main function, starting point of code running.

Subroutines are very similar to functions

but everything is in the argument list including the return variable, then may return many variables.

Subroutine example

```
real*8 function distance(x1, y1, x2, y2)
    real*8 :: x1, y1, x2, y2
    distance = sqrt ((x1 - x2)**2 + (y1 - y2)**2)
    return
end function distance

subroutine get_distance(x1, y1, x2, y2, distance)
    real*8 :: x1, y1, x2, y2, distance
    distance = sqrt ((x1 - x2)**2 + (y1 - y2)**2)
    return
end subroutine get_distance

aa = distance(a1, b1, a2, b2)
call get_distance(x1, y1, x2, y2, distance)
```

Argument intent

```
subroutine get_distance(x1, y1, x2, y2, distance)
    real*8, intent(in)    :: x1, y1, x2, y2
    real*8, intent(out) :: distance
    distance = sqrt ((x1 - x2)**2 + (y1 - y2)**2)
    return
end subroutine get_distance
```

```
real*8, intent(inout) :: aaa, bbb
```

Modules

can be used to declare global data and other specification statements (like interface block). It can be accessible when a "use" statement of it is coded.

Modules can use other previous modules.

Module example

```
module  factorials
implicit none
integer, parameter :: size_of_factls = 10
real*8  ::  factls(size_of_factls)
end module factorials

subroutine ini_factorials()
use factorials
implicit none
integer :: i
factls(1) = 1.0d0
do i = 2, size_of_factls
        factls(i) = factls(i-1) * i
end do
return
end subroutine ini_factorials
```

```
subroutine print_a_factorial(i)
use factorials
implicit none
integer :: i
print*, factls(i)
end subroutine print_a_factorial
```

```
program test_40
use factorials
implicit none
call ini_factorials()
call print_a_factorial(8)
stop
end program test_40
```

Module, functions, subroutines, and the main function

can contain internal subprograms (functions and/or subroutines), after a

contains

statement.

An internal subprogram automatically has access to all the host's entities, including the ability to call its other internal subprograms.

Module, functions, subroutine, and the main function

```
module  factorials
implicit none
integer, parameter :: size_of_factls = 10
real*8  ::  factls(size_of_factls)

contains
subroutine ini_factorials()
integer :: i
factls(1) = 1.0d0
do i = 2, size_of_factls
        factls(i) = factls(i-1) * i
end do
return
end subroutine ini_factorials
end module factorials
```

```
subroutine print_a_factorial(i)
use factorials
implicit none
integer :: i
print*, factls(i)
end subroutine print_a_factorial
```

```
program test_40
use factorials
implicit none
call ini_factorials()
call print_a_factorial(8)
stop
end program test_40
```

Overloading

A group of functions/subroutines usually of the same functionality but with different dummy argument list of types, can be "renamed" the same in a interface block, although they originally have different names.

```
module  my_renaming
interface  the_new_universal_name
      function old_name_001(…)
      …
      function old_name_002(…)
      …
      function old_name_003(…)
      …
end interface  the_new_universal_name
end module  my_renaming

use  my_renaming
```

Automatic objects

```
subroutine swap(a, b)
    real*8 :: a(:),  b(:)
    real*8 :: work(size(a))
    work = a
    a = b
    b = work
end subroutine swap
```

```
real*8 :: a(10), b(10)
real*8 :: cc(800), dd(800)
…
call swap(a, b)
call swap(cc, dd)
```

```
subroutine word_process(word1)
character(len=*) :: word1
character(len=len(word1)) :: word2
…
end subroutine word_process
```

```
character(len=20) :: aa1
character(len=436) :: ggt
call word_process(aa1)
call word_process(ggt)
```

Intrinsic procedures

Elemental numeric functions

abs(a), aimag(z), aint(a), anint(a), ceiling(a),
cmplx(x [,y]), floor(a), int(a), nint(a), real(a)

conjg(z), dim(x, y), max(a1, a2 [, a3, …]),
min(a1, a2 [, a3, …]), mod(a, p),
modulo(a, p), sign(a, b)

Elemental mathmatical functions

acos(x), asin(x), atan(x), atan2(y, x), cos(x), cosh(x), exp(x), log(x), log10(x), sin(x), sinh(x), sqrt(x), tan(x), tanh(x)

Character-integer conversions

achar(i), char(i), iachar(c), ichar(c)

String-handling functions

len(string)

adjustl(string), adjustr(string),

index(string, substring [, back])

len_trim(string), scan(string, set [, back])

verify(string, set [, back])

repeat(string, ncopies)

trim(string)

Array operations

dot_product(vector_a, vector_b)

matmul(matrix_a, matrix_b)

all(mask), any(mask), count(mask)

maxval(array), minval(array),

product(array), sum(array)

allocated(array)

lbound(array [, dim]), ubound(array [, dim])

shape(array), size(array [, dim])

transpose(matrix)

Time

call date_and_time([date] [, time] [, zone] [, values])

call system_clock([count] [, count_rate] [, count_max])

call cpu_time(time)

Random numbers

```
call random_number(harvest)
call random_seed([size] [put] [get])
```

Input/output and external files

Keyboard input and terminal output

```
read(*, *) variables

print*, variables
write(*,*) variables
```

External files

```
unit_number = 25
open(unit_number, file = '…/file1.dat')
read(unit_number, *) variables
close(unit_number)

open(unit_number, file = '…/file2.dat')
write(unit_number, *) variables
close(unit_number)
```

read/write formats

```
read(unit_number, 10) x, y, z
write(unit_number, 10) x, y, z
10 format(3e20.12)
```

or

```
read(unit_number, '(3e20.12)') x, y, z
write(unit_number, '(3e20.12)') x, y, z
```

read/write formats

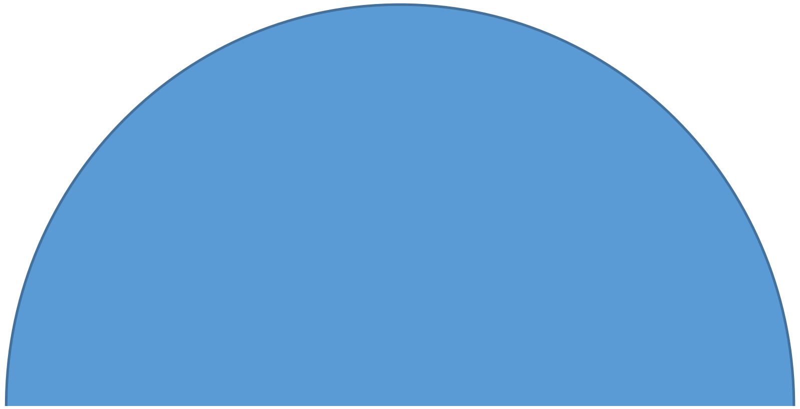
```
read(unit_number, '(a, i8, f20.12)') x, y, z
write(unit_number, '(a, i8, f20.12)') x, y, z
```

Many formats, inquiries, and various operations can be done on formatted, unformatted, direct-access files.

Application example: π calculation

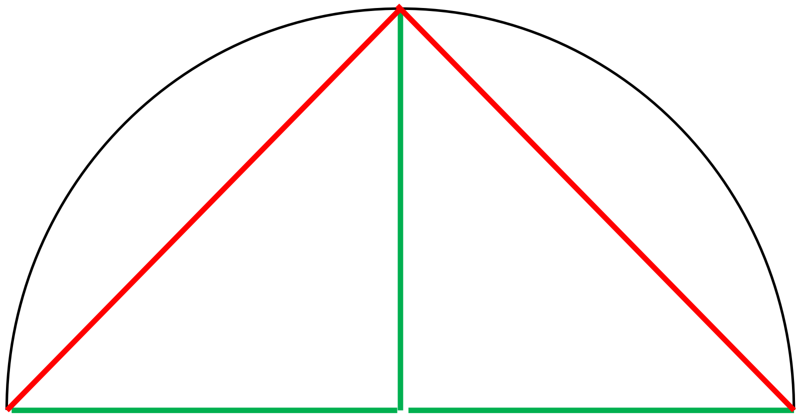
π calculation based on half circle

$$\pi = \frac{half\_circle}{r}$$



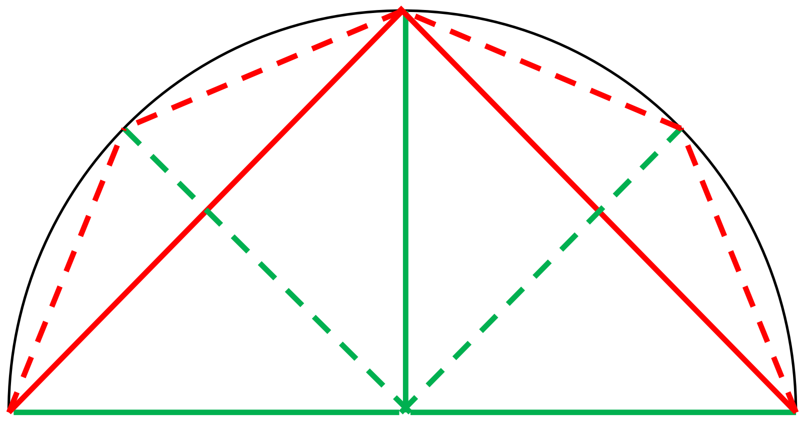
π calculation based on half circle

$$\pi \approx \frac{\sum chords}{r}$$



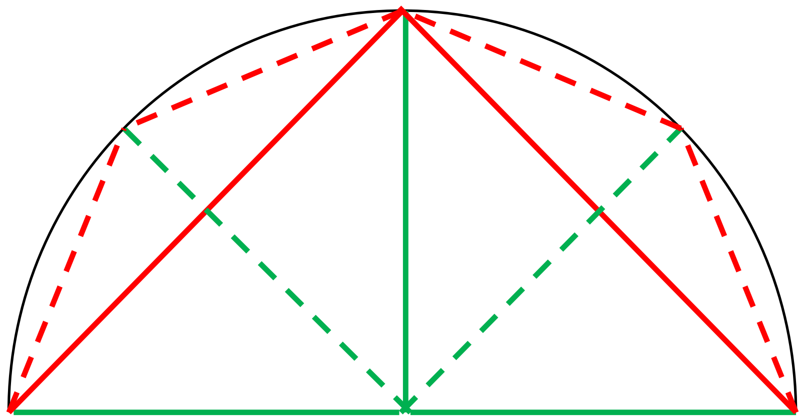
π calculation based on half circle

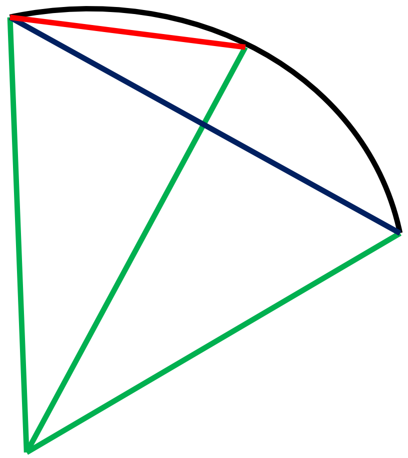
$$\pi \approx \frac{\sum chords}{r}$$



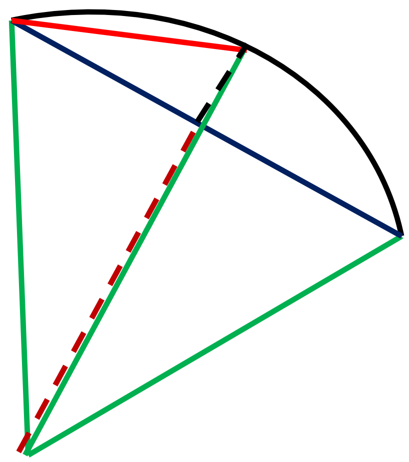
π calculation based on half circle

$$\pi \approx \frac{\sum chords}{r}$$





π calculation based on half circle



$$\pi \approx \frac{\sum chords}{r}$$

h: old chord in blue
$v1$: dark red dashed part
$v1 = \sqrt{r^2 - (h/2)^2}$
$v2$: black dashed part

$$v2 = r - v1$$

new chord in red

$$\sqrt{(h/2)^2 + (v2)^2}$$

π calculation based on half circle

```
MODULE BASIC_DATA_MDL
    IMPLICIT NONE
    REAL*8, PARAMETER :: RADIUS  = 1.0D0
    REAL*8, PARAMETER :: RADIUS_SQUIRED = RADIUS ** 2
    REAL*8, PARAMETER :: REQUIRED_ACCURACY  = 1.0D-12
END MODULE BASIC_DATA_MDL


REAL*8 FUNCTION NEXT_CHORD(CHORD_TRIED)
    USE BASIC_DATA_MDL
    REAL*8 :: CHORD_TRIED, HALF, VT1, VT2
    HALF = CHORD_TRIED/2
    VT1 = SQRT(RADIUS_SQUIRED - HALF * HALF)
    VT2 = RADIUS - VT1
    NEXT_CHORD = SQRT(HALF * HALF + VT2*VT2)
END FUNCTION NEXT_CHORD
```

$$\pi = \frac{\sum chords}{r}$$

h: old chord in blue

$v1$: dark red dashed part

$v1 = \sqrt{r^2 - (h/2)^2}$

$v2$: black dashed part

$$v2 = r - v1$$

new chord in red

$$\sqrt{(h/2)^2 + (v2)^2}$$

π calculation based on half circle

```
MODULE INTERFACE_MDL
INTERFACE
REAL*8 FUNCTION NEXT_CHORD(CHORD_TRIED)
   REAL*8 :: CHORD_TRIED
END FUNCTION NEXT_CHORD
END INTERFACE
END MODULE INTERFACE_MDL
```

π calculation based on half circle

$$\pi = \frac{\sum chords}{r} \qquad r=1.0$$

```
PROGRAM PI_CALCULATION
   USE BASIC_DATA_MDL
   USE INTERFACE_MDL
   INTEGER     :: EFFORT
   REAL*8      :: CHORD, NUMBER_OF_CHORDS, PREVIOUS_PI, CURRENT_PI, RELATIVE_ERROR
   EFFORT = 1; CHORD = SQRT(RADIUS_SQUIRED + RADIUS_SQUIRED)
   NUMBER_OF_CHORDS = 2.0D0
   PREVIOUS_PI = 8.0D10;      CURRENT_PI = CHORD * NUMBER_OF_CHORDS / RADIUS
   RELATIVE_ERROR = ABS(CURRENT_PI - PREVIOUS_PI) / CURRENT_PI
   WORKING_HARD: DO
         EFFORT = EFFORT + 1;    CHORD = NEXT_CHORD(CHORD)
         NUMBER_OF_CHORDS = 2.0D0 * NUMBER_OF_CHORDS
         PREVIOUS_PI = CURRENT_PI;    CURRENT_PI = CHORD * NUMBER_OF_CHORDS / RADIUS
         RELATIVE_ERROR = ABS(CURRENT_PI - PREVIOUS_PI) / CURRENT_PI
         IF(RELATIVE_ERROR < REQUIRED_ACCURACY) EXIT WORKING_HARD
   END DO WORKING_HARD
   PRINT*,  EFFORT,   CURRENT_PI,   PREVIOUS_PI
END PROGRAM PI_CALCULATION
```

π calculation for higher accuracy

Change all real*8 into real*16 and

REQUIRED_ACCURACY = 1.0D-12 into 1.0D-32

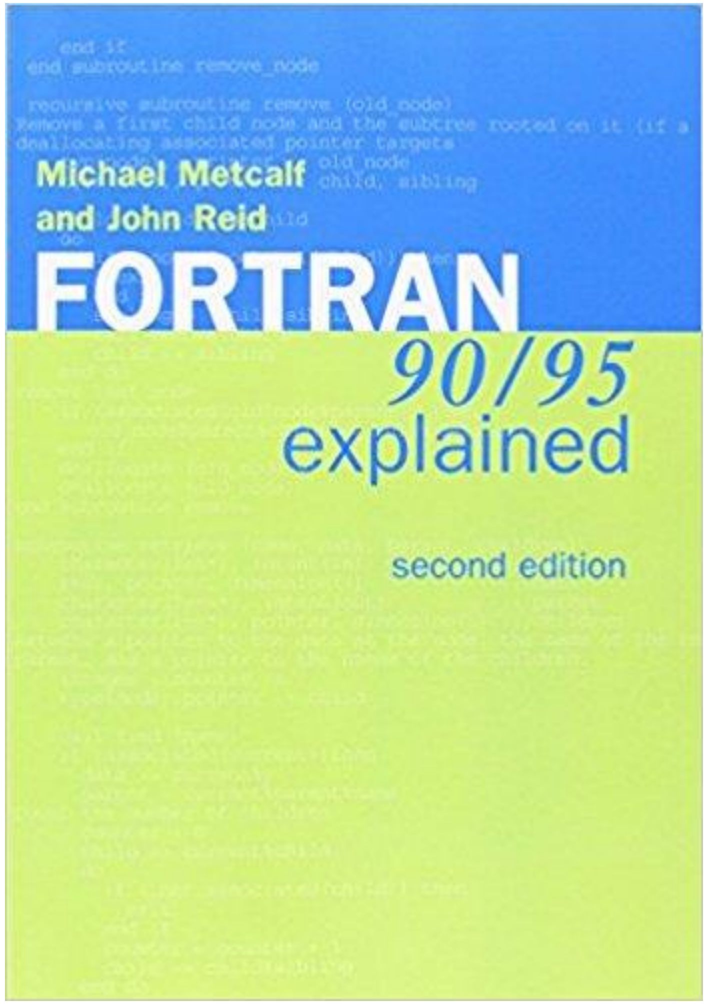
π in wiki

- https://en.wikipedia.org/wiki/Pi

- The first 50 decimal digits are 3.14159265358979323846264338327950288419716939937510

References

http://j3-fortran.org/



http://j3-fortran.org/doc/year/10/10-007r1.pdf

Thank you very much
for your attention!
Have a nice day!