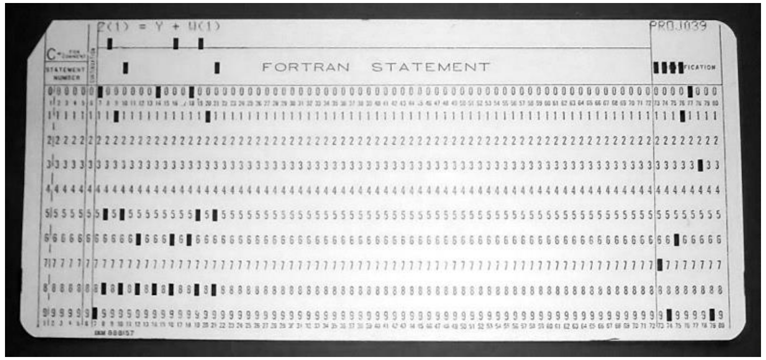
41525 Finite Element Methods

Introduction to Fortran

What is Fortran?

- A very old programming language
  - First version released by IBM in 1957 for their IBM 704 mainframe

LLNL(on flickr.com)



The Craft of Coding (on wordpress.com)

- Still updated and maintained, still widely used
  - Latest standard published in 2018

- Used especially for demanding scientific computation

wikipedia.org

Why Fortran in 41525 Finite Element Methods?

- Fortran is easy to get started with
- Fortran is heavily optimized and therefore fast
- Fortran is highly portable (can be compiled to run on Windows, Linux, etc.)
- Fortran’s array syntax is similar to MATLAB's

- **Why not use other common languages?**
- Of course this is possible! But for our purposes:
  – MATLAB and Python are too slow
  – C++ is too difficult to learn

Some key differences from MATLAB

- Fortran is statically compiled
  - Write the code, compile the code, run the code
  - Creates separate executables. Once you are done programming you are done with Fortran
- Fortran is strongly typed
  - Variables must be declared with a data type before use
- Fortran is case-insensitive: `x = X`
- Fortran is “bare bones”
  - Has some hundreds built-in functions

- MATLAB is just-in-time compiled
  - Write the code, run the code
  - Does not create executables. MATLAB must always be opened to run the code
- MATLAB is weakly typed
  - Any data type can be put in any variable (almost)
- MATLAB is case-sensitive: `x ≠ X`
- MATLAB is feature-rich
  - Has many expansions and thousands of built-in functions

How to program Fortran: The steps required

- Write the code
  - Fortran files are plain text files (called source files) containing Fortran instructions
  - Any text editor will do, but some (e.g. Code::Blocks, Notepad++, VS Code) provide nice, Fortran specific features
  - The code is just text, it can not be run yet!

- Compile the code
  - A compiler is a program that reads the plain text files containing Fortran code and turns them into an executable
  - Most compilers provide options for optimization which can dramatically speed up your code
  - Many compilers are available (e.g. GNU Fortran Compiler (gfortran), Intel Fortran Compiler (ifort), Silverfrost FTN95)

- Run the code
  - Typically from a shell (command window)

How to program Fortran: Integrated Development Environment (IDE)

- It seems a bit complicated to get started!?

- Don’t worry! All the previous steps can be collected in a single program called an Integrated Development Environment (IDE)

- Code::Blocks is a free, cross-platform IDE for Fortran (and C/C++)
  - Contains a feature-rich code editor
  - Manages your project and source files
  - Calls the compiler and creates the executable
  - Supports debugging and error checking

The Fortran language: Program structure

Try me!

```fortran
program AddTwoNumbers
   implicit none

   ! Variable declarations
   integer :: a, b, c

   ! Add two numbers
   a = 5
   b = 10
   c = a + b

   ! Print result to screen
   print*,'The sum is ', c

end program
```

Start and name of the program

Weird legacy quirk of Fortran:
Variable names starting with
`i,j,k,l,m` or `n`
are assumed to be **`integer`**, all others assumed **`real`**.
`implicit none`
removes this assumption

Comments start with `!`

Variable type declarations before use

Assignment and addition

Note that semi-colons `;` are not used

Print result to screen

The asterisk `*` will be discussed later

End of program

The Fortran language: Built-in data types

- `logical` – boolean values `.true.` or `.false.`
- `integer` – integer values
- `real` – floating point values
- `complex` – complex numbers
- `character` – characters or strings (text variables)

- All *number* variables can be given a `kind` specifier, which sets the precision (number of bytes) used to store the number (defaults to `kind = 4`)
  – For instance: `integer(kind = 8) :: x`

- All *character* variables can be given a `len` specifier, which sets the number of characters stored in the variable (defaults to `len = 1`)
  – For instance:

```
character(len = 4) :: name
name = 'John'
```

The Fortran Language: More on the `kind` specifier

- The `kind` specifier can be set to 1, 2, 4, 8 or 16, whereby the variable will be stored using 1,2,4,8 or 16 bytes of memory
- For integer values `kind` determines the possible range of numbers. For instance:

`kind` specifier	Number of bits	Range	Reason
`kind = 4`	32	+/- ~2,147,483,647	(2^31)-1
`kind = 8`	64	+/- ~9,223,372,036,854,774,807	(2^63)-1

- For floating point values `kind` determines the possible range and precision of numbers. For instance:

`kind` specifier	Number of bits	Largest value	Smallest nonzero value	Precision
`kind = 4`	32	+/- ~1.7*10^38	+/- ~0.3*10^(-38)	6-9
`kind = 8`	64	+/- ~0.8*10^308	+/- ~0.5*10^(-308)	15-18

- Self study: In addition to the `kind` specifier, check out the `kind()` function used in the `types.f90` source file (part of the hand-out code from today)

Common pitfalls: Integer division

Dividing an integer by an integer always produces an integer result!

Try me!

```
program IntegerPitfall
   implicit none

   ! Variable declaration
   real :: x
   x = 1.0

   ! Print results to screen
   print*, 2/3
   print*, 2/3*x
   print*, 2.0/3.0

end program
```

Prints `0`. Result is a truncated integer

Prints `0.000000`. Result is a real but still truncated

Prints `0.666666687`

This behavior differs from MATLAB which assumes all numeric variables to be double precision floating point unless otherwise stated

Common pitfalls: Mixing kinds

Try me!

```
program MixingKinds
   implicit none

   real(kind = 4) :: x
   real(kind = 8) :: y

   x = 1.2
   y = 1.2

   print*, x
   print*, y

   y = 1.2d0

   print*, y
   print*, x + y

end program
```

Mixing reals of different kind can produce unexpected results!

Declare as *single precision*

Declare as *double precision*

Assign variables the same single precision (default) number

Prints `1.20000005`

Prints `1.2000000476837158`

Assign double precision number

Prints `1.2000000000000000`

Prints `2.4000000476837160`

When mixing reals of different kind, the result is only as precise as the lowest kind used allows

The Fortran language: Derived data types (structures)

```fortran
! Define a new data type
type Material
   real :: E, nu
   integer :: id
end type

! Variable declarations
type(Material) :: steel
real :: stress, strain

! Assign material parameters
steel%E = 200
...
! Recover stress
stress = steel%E*strain
```

Name of new data type

Derived data types can contain multiple data fields of varying kinds

Indexing is done using the percentage sign `%`

The Fortran Language: Operators: Arithmetic

- The basic arithmetic operators are

Description	Fortran operator	MATLAB operator
Addition	+	+
Subtraction	−	−
Multiplication	*	*
Division	/	/
Exponential	**	^

This one always causes trouble!

The Fortran Language: Operators: Relational

- The relational operators are

Description	Fortran operator	MATLAB operator
Equal	`==` or `.eq.`	`==`
Not equal	`/=` or `.ne.`	`~-`
Greater than	`>` or `.gt.`	`>`
Less than	`<` or `.lt.`	`<`
Greater or equal	`>=` or `.ge.`	`>=`
Less or equal	`<=` or `.le.`	`<=`

The Fortran Language: Operators: Logical

- The relational operators are

Description	Fortran operator	MATLAB operator
And	`.and.`	`&&`
Or	`.or.`	`||`
Not	`.not.`	`~`
Equivalent	`.eqv.`	`==`
Not equivalent	`.neqv.`	`~=`

The Fortran language: Flow control: `if`

```fortran
if (a == 1) then
   ! Do some stuff
else if (a < 0) then
   if (b > 0.1) then
      ! Do other stuff
   end if
else
   ! Default option
end if

if (b < 0.6) b = -b
```

- `then` statement is required
- Block `if`
- Parentheses are required
- Nested block `if`
- `end if` is required after block `if`
- In-line if

The Fortran language: Flow control: `if`

There are some differences in syntax as compared to MATLAB

```fortran
if (a == 1) then
   ! Do some stuff
else if (a < 0) then
   if (b > 0.1) then
      ! Do other stuff
   end if
else
   ! Default option
end if

if (b < 0.6) b = -b

if (100) print*, 'hello'
```

Error: Condition must be boolean in Fortran!

```matlab
if a == 1
   % Do some stuff
elseif a < 0
   if b > 0.1
      % Do other stuff
   end
else
   % Default option
end

if b < 0.6; b = -b; end

if 100; disp('hello'); end
```

In MATLAB this is fine

Self study

The Fortran language:
Flow control: `select case`

```
select case(a)
case(1)
   ! Do some stuff
case(2, 4, 10:20)
   ! Do other stuff
case default
   ! Default option
end select


select case(str)
case('hello')
   ! Do some stuff
case('world')
   ! Do other stuff
end select
```

Switch between multiple cases based on the value of `a`

Each case can contain a single option

Or multiple options. Even ranges

A default case is allowed

Switch between multiple cases using a string

The Fortran language: Flow control: do loops

```fortran
do
   ! Do stuff forever
end do

do i = 1, 10
   ! Do some stuff 10 times
end do

do i = n, 1, -1
   ! Count down from n
end do

do while (x > 0)
   ! Do stuff until x <= 0
end do
```

Endless loop

Remember to initialize loop variable before use (`integer :: i`)

Finite loop with predetermined length

Finite loop with variable length and specified increment size

Conditional loop

The Fortran language: Flow control: do loops

There are some differences in syntax as compared to MATLAB

```fortran
do
   ! Do stuff forever
end do

do i = 1, 10
   ! Do some stuff 10 times
end do

do i = n, 1, -1
   ! Count down from n
end do

do while (x > 0)
   ! Do stuff until x <= 0
end do
```

After the loop: `i = 11`

```matlab
while true
   % Do stuff forever
end

for i = 1:10
   % Do some stuff 10 times
end

for i = n:-1:1
   % Count down from n
end

while x > 0
   % Do stuff until x <= 0
end
```

After the loop: `i = 10`

Note the ordering

Self study

The Fortran language: Flow control: do loops

```fortran
do...
   ...
   cycle
   ...
   exit
   ...
end do

increments: do...
   iterations: do...
      ...
      exit increments
      ...
   end do iterations
end do increments
```

Jump to next loop iteration

Exit current loop

Named do loops

Exit specified loop

Names are required after the end do statement

Let's get more advanced!
How to structure programs as complexity increases

- Main program stored in the primary .f90 file
- Additional *procedures* stored in *modules* in separate .f90 files
- A module file can contain many variables, procedures and data types
- Data can be passed to procedures as arguments or made available multiple places using modules

- Main program stored in the primary .m file
- Additional *functions* stored in separate .m files
- A function file can contain a single function
- Data should almost always be passed to functions as arguments. Globals are not recommended

Modules are essential to organizing Fortran projects. Think of them as libraries or packages where you can keep procedures, derived data types, variables to be used in multiple places, etc.

The Fortran language: Modules: A simple example

Try me!

mod.f90

```fortran
module Constants
   implicit none
   real :: pi = 3.1415
end module
```

The module contains a single variable

main.f90

```fortran
program MainProgram
   use Constants
   implicit none
   real :: area, radius

   radius = 2
   area = pi*radius**2
   print*,'Area = ', area
end program
```

Make the module `Constants` available for use in the program

The parameter `pi` can now be used

Procedures:
Functions and subroutines

- There are two types of *procedures* in Fortran
  - Functions
    - Take one or more input arguments and return a single output
    - Functions that return numeric values can be used in variable declarations such as `y = f(x)` or in arithmetic as `z = 2 - 3*f(x)`
    - There are many built-in functions (intrinsics) such as `sqrt(x)`, `sin(x)`, etc.
    - Generally used for small, simple operations

  - Subroutines
    - Take one or more input arguments and return one or more outputs
    - Are invoked using the **`call`** statement, e.g. `call ComputeArea(area, size, type)`
    - Generally used for larger parts of the code, e.g. stiffness matrix assembly

- Recommendation: Procedures should generally be placed in modules, not in the main program file, to avoid errors

The Fortran language: Procedures: Functions

Try me!

mod.f90

```
module CircleModule
  implicit none
  real :: pi = 4*atan(1.0)

  contains

  function area(r) result(a)
    real :: r, a
    a = pi*r**2
  end function

end module
```

main.f90

```
program MainProgram
  use CircleModule
  implicit none

  print*,'A = ', area(3.0)
end program
```

Variables defined here are available to all procedures in the module

More robust way to define `pi`

Procedure definitions must be preceded by the `contains` statement

Function name, input argument(s) and result variable definitions

Variables defined here are only available in this function

Calling the function with input argument `3.0`

The Fortran language: Procedures: Subroutines

Try me!

mod.f90

```
module CircleModule
  implicit none
  contains

  subroutine area(a, r)
    real :: a, r, pi = 3.141
    a = pi*r**2
  end subroutine

end module
```

main.f90

```
program MainProgram
  use CircleModule
  implicit none
  real :: a

  call area(a, 3.0)
  print*,'Area = ', a
end program
```

Subroutine name and input/output arguments

If a subroutine returns a value, it simply modifies its arguments

Declare variable `a`

Call the subroutine. The subroutine updates the variable `a`

The variable `a` now contains the area

Common pitfalls: Wrong use of procedures

Functions and subroutines are *not* the same thing and are not called in the same way

```
program ProcedureConfusion
  use MyModule
  implicit none

  ...

  a = 1 + call MyRoutine(x)
  call MyFunction(x)
  MyFunction(x)
  print*, MyFunction(x)
  [a,b] = MyFunction2(x,y)

end program
```

Subroutines cannot be used inline in arithmetic

A function cannot be invoked using the `call` statement

The function output must go somewhere

This is OK. If you want to test a function, use the `print` statement

This is MATLAB syntax. A function in Fortran can have only one output

Use functions for simple things like `sqrt(x)` or computing an area. Use subroutines for complex tasks like building the stiffness matrix

Common pitfalls: Implicit save

Declaring and initializing a variable at the same time implicitly adds the *save* attribute

mod.f90

```
module MyModule
  implicit none
  subroutine Sub
    implicit none
    integer :: a = 0

    a = a + 1

  end subroutine
end module
```

Initialization only happens once. The variable is saved between calls

Each call to the subroutine causes `a` to increment by 1

main.f90

```
program MainProgram
  implicit none
  call Sub
  call Sub
end program
```

After this call `a = 1`

After this call `a = 2`

If the variable is not a constant, always separate the declaration and initialization steps

```
integer :: a
a = 0
```

The Fortran language: Procedures: Argument intent

```
module NumTools
  implicit none
  contains

  subroutine Solver(A, b, i)
    real, intent(in) :: A
    real, intent(inout) :: b
    real, intent(out) :: i

    ...

  end subroutine
end module
```

`A` is transferred from the caller to the subroutine, where it is read-only. It is not returned to the caller

`b` is transferred from the caller to subroutine, where it can be overwritten and returned to the caller

`i` is not transferred from the caller. It is used only as an output and is returned to the caller

Specifying argument intent is not required but recommended. It is checked by the compiler which helps to find and eliminate mistakes

The Fortran language: Modules: Public or private

mod.f90

```fortran
module NumTools
  implicit none
  private :: PreProc
  real, private :: c
  contains

  subroutine Solver(A, b, i)
    call PreProc(b)
    ...
  end subroutine

  subroutine PreProc(b)
    ! Do something using c
  end subroutine

end module
```

Define the subroutine `PreProc` as private. It is only available *inside* the module

Define the variable `c` as private

Define the subroutine `Solver`. By default everything in a module is public, so `Solver` is public

`Solver` can call `PreProc`

`PreProc` can use `c`

main.f90

```fortran
program MainProgram
  use NumTools
  ...
  call Solver(A, b, i)
  call PreProc(b)
  print*, c
end program
```

OK

Error

Error

The Fortran language:
Arrays: 1 dimensional (vectors)

```fortran
! Vector with 8 elements
real, dimension(8) :: v

! Initialize with all zeros
v = 0

! Set first three elements
v(1:3) = (/1.0, 2.0, -41.9/)

! Add 3 to an element
v(7) = v(7) + 3.0
```

```matlab
% Vector with 8 zeros
v = zeros(8,1);

% Initialize (not needed)
v(:) = 0;

% Set first three elements
v(1:3) = [1, 2, -41.9];

% Add 3 to an element
v(7) = v(7) + 3;
```

Array indexing in Fortran is very similar to MATLAB except for minor syntax differences. Both languages use 1-indexing

The Fortran language:
Arrays: 2 dimensional (matrices)

```fortran
! 3x3 matrix
real, dimension(3,3) :: A

! Initialize with all zeros
A = 0

! Set element [1,2]
A(1,2) = 3.0

! Set second row
A(2,:) = (/1.0, 2.0, 3.0/)
```

```matlab
% 3x3 matrix
A = zeros(3);

% Initialize (not needed)
A(:) = 0;

% Set element [1,2]
A(1,2) = 3;

% Set second row
A(2,:) = [1, 2, 3];
```

The Fortran language: Arrays: Dynamic

```fortran
! Declare vector with unknown size
real, dimension(:), allocatable :: v

! Declare matrix with unknown size
real, dimension(:,:), allocatable :: A

! Allocate at run time
allocate(v(n))
allocate(A(n,m))

! Zero all elements
v = 0
A = 0
```

Declare arrays with sizes that are unknown at compile time

Allocate the arrays once the sizes are known at run time (`n` and `m` are integer variables)

The Fortran language: Arrays: Operations

```fortran
! Vector dot product
x = dot_product(v1, v2)

! Matrix multiplication
C = matmul(A, B)

! Elementwise multiplication
C = A*B
```

```matlab
% Vector dot product
x = dot(v1, v2);

% Matrix multiplication
C = A*B;

% Elementwise multiplication
C = A.*B;
```

Important to remember:

Fortran defaults to elementwise operations, MATLAB defaults to vector/matrix operations

Common pitfalls: Wrong dummy array shape

Specifying a *dummy shape* for an array in a subroutine can cause unexpected behavior!

main.f90

```
program MainProgram
  ...
  real, dimension(3,4):: A
  call subBad(A)
  call subNice(A)
end program
```

mod.f90

```
module ArrayModule
  ...
  subroutine subBad(A)
    real, dimension(2,2):: A
  end subroutine

  subroutine subNice(A)
    real, dimension(:,:):: A
  end subroutine

end module
```

Let us assume

$$A = \begin{matrix} 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 \end{matrix}$$

Specifying the shape can cause implicit (without warning) reshape at run time such that inside the subroutine the matrix is

$$A = \begin{matrix} 1 & 3 \\ 2 & 1 \end{matrix}$$

Using dynamic sizing in the input arguments is the safe way

Output:
Print and write

- Fortran has a rich, but sometimes complicated/confusing, input/output system
- The are two main functions for output in Fortran
  - `print`
    - Generally used for debugging
    - Always prints to the default output unit (the screen in most cases)
  - `write`
    - Generally used for outputting data
    - Can write to any available output unit (usually the screen or an output file)

Recommendation:

Use `print` for debugging only. It is then easy to search for and remove all `print` statements once the program is working

The Fortran language: Output: Print

Try me!

```
program PrintTemp
  implicit none
  real :: T = 21.5

  ! Print using default formatting
  print*, T

  ! Print using a format specifier
  print'(f5.2)', T

  ! Print using default formatting
  print*, 'Temp. is ', T

  ! Print using a format specifier
  print'(Af5.2)', 'Temp. is ', T

end program
```

The asterisk * means default formatting. Often gives usable results

Prints
`21.5000000`

Print a floating point number with 2 decimals using at most 5 characters

Prints
`21.50`

Prints
`Temp. is      21.5000000`

Print a string of unspecified length followed by a floating point number with 2 decimals using at most 5 characters

Prints
`Temp. is 21.50`

The Fortran language: Output: Print

```fortran
real, dimension(3) :: vec
real, dimension(2, 4) :: mat

...

! Print vec as row vector
print*, vec

! Print vec as column vector
print'(f20.8)', vec

! Print mat as matrix
print'(4f20.8)', transpose(mat)
```

Arrays are printed as rows when using default formatting

The format specifier is applied once per line printed, and thus the array is printed as a column

Fortran cycles through each element in each column first, but `print` displays rows. Transpose is needed for correct display of matrix

The format specifier can be applied multiple times per line printed to display a matrix. Here 4 times since the matrix has 4 columns

The Fortran language: Output: Write

Try me!

```
program WriteTemp
  implicit none
  real :: T = 21.5

  ! Print to screen
  write(*, *) 'Temp. is ', T

  ! Save to file
  open(unit=20, file='result.txt')
  write(20, *) 'Temp. is ', T
  close(20)

end program
```

Write using default formatting to the default output unit (screen)

Open a file and write the output (using default formatting) to the file instead of the screen

Files are identified by a unit number. Numbers less than 10 may be reserved - don't use them. Files have read/write/create-behavior by default

Common pitfalls:
Too small field width

Too small field width turns output into asterisks.

Try me!

```
program main

  print'(f5.3)', 12.345

  print'(f6.3)', 12.345

  print'(f7.3)', 12.345

  print*, 12.345

end program
```

Prints (without " ")
`"*****"`

Prints (without " ")
`"12.345"`

Prints (without " ")
`" 12.345"`

Prints (without " ")
`" 12.3450003 "`

Simple solution: Use default formatting

Concluding remarks: Coding style

- Use meaningful variable, procedure and module names
  – Good naming increases readability of you code drastically

- Make your code flexible
  – Don’t hard code material parameters, dimensions and other "magic numbers"

- Take the time to clean up your code after you have completed an assignment
  – Old clutter inevitably leads to bugs

- Use subroutines and functions extensively to organize your code

- Check if an intrinsic (built-in function) does what you want before you start coding
  – https://fortranwiki.org/fortran/show/Intrinsic+procedures

Concluding remarks: Documentation and getting help

- The documentation for the source code hand out is available at
  - www.student.dtu.dk/~clfe
- Extensive documentation for Fortran is available at
  - https://fortranwiki.org/fortran/show/HomePage
- The compiler often gives useful insight in case of bugs
  - Make sure to read the compiler output when you have issues
- Fortran debugging is very useful and supported in Code::Blocks
  - You will have to set this up yourself. Check out the gdb debugger if you are interested
- Fortran has been in use for more than half a century
  - If you have a problem, someone else has had it as well. Google is your friend!