We cover the chapters of your Fortran text listed in the syllabus two at a time: 7&8, 9&10, 11&12, 13&14, 15&16, 17&18, 19&22, 23.
Reading assignment: chapters 9, 10, 11, 22.  Open SciTE, Lecture 8

**Matrix functions**

The intrinsic array functions are:

- \( a + b \), \( a \times b \) = componentwise addition and multiplication.
- \( \text{maxval}(a) \) = the maximum value.
- \( \text{minval}(a) \) = the minimum value.
- \( \text{maxloc}(a) \) = the index of maximum value.
- \( \text{minloc}(a) \) = the index of minimum value.
- \( \text{sum}(a) \) = the sum of the elements.
- \( \text{product}(a) \) = the product of the elements.
- \( \text{matmul}(a,b) \) = the matrix product.
- \( \text{dot_product}(a,b) \) = the dot product of vectors \( a, b \).
- \( \text{transpose}(a) \) = the transpose.
- \( \text{size}(a,1) \) = number of rows (1st dimension).
- \( \text{size}(a,2) \) = number of columns (2nd dimension).
In Matlab, \( a \times b \) is matrix multiplication, \\
\( a \cdot b \) is componentwise multiplication.  
In Fortran, \( a \times b \) is componentwise multiplication,  
\( \text{matmul}(a,b) \) is matrix multiplication.

**Allocation**

If the size of a vector or array is not known at the beginning of a program, declare it to be allocatable with dimension(:) for vectors and dimension(:,:) for arrays. Then allocate space later in the program with  
\( \text{allocate}(v(n)) \) or \( \text{allocate}(v(1:n)) \) for vectors, and  
\( \text{allocate}(a(n,m)) \) for arrays. For two vectors \( v, w \)  
\( \text{allocate}(v(n),w(n)) \) allocates \( n \) places for each.

Warning on console input: don’t backspace on console inputs. If you do, press <F5> to start again.
**Example** allocate.f95

```fortran
allocate.f95
program allocate
integer,dimension(:),allocatable :: v
integer :: i,n
print *,'How many numbers are there?'
read *,n
allocate(v(n))
print *,'Enter',n,'numbers.'
read *,v
print *,(v(i),i=n,1,-1)!goes backwards, step-size -1
endprogram
```

In a do-loop, `do i=10,0,-2` means start at 10, stop at 0, step-size = -2.

As an alternative to allocating space for a vector of unknown length, declare the vector to be of long enough (length=200) to handle any likely situation. See CLASSWORK 15.4

See **ENTERING LONG FORMULAS, COMMON ERRORS** at this lecture's end.
For a vector \( a = [a_1, a_2, ..., a_n] \), the magnitude \( \sqrt{a_1^2 + a_2^2 + ... + a_n^2} \)

**CLASSWORK 15.1(3) magnitude.f95**

```
real function magnitude(v,n)
integer::n; real::v(n)
magnitude= ______  !write the formula, use ** not ^
endfunction

program test_magnitude
integer::n; real::magnitude
__________ !declare v allocatable
print *, 'How many numbers are there?'
read *,n
__________ !allocate space for v(n)
print *, 'Enter',n,'numbers.'; read *,v
print'(a,200(f5.2))',"The vector ",v
print*, 'has magnitude',magnitude(v,n)
endprogram
```
Write a real function \texttt{distance(v,w,n)} which given an integer \( n \), and vectors \( v, w \) with \( n \) real numbers, returns the distance between of these two vectors:

\[
\sqrt{(v_1 - w_1)^2 + (v_2 - w_2)^2 + \ldots + (v_n - w_n)^2}.
\]

Write a program \texttt{test\_distance} which uses this function.

It asks for and reads the number \( n \) of values.
It allocates space for vectors \( v \) and \( w \).

\texttt{allocate(v(n),w(n))}

It asks for and reads the \( n \) values of vector \( v \) on one line.
It asks for and reads the \( n \) values of vector \( w \) in the next.
It prints “The distance = ” followed by the distance calculated by \texttt{distance(v,w,n)}.
For \( n = 3 \). \( v = [1, 2, 3] \), \( w = [1, 1, 1] \) the printout is:
The distance = 2.236.
**Internal subroutines vs. external subroutines**

An *external* subroutine lies outside the program. The variables the subroutine needs to know are listed as *arguments* after its name as in

```plaintext
subroutine printer(b,n,m)
```

All other variables are isolated from those of the main program. They won’t accidently corrupt variables in the main program but they can’t allocate space for program variables.

*Internal* subroutines lie inside the program after a *contains* statement as in

```plaintext
contains
subroutine printer
...
endsubroutine
endprogram
```

Internal subroutines can see and allocate (and possibly corrupt) the program’s variables. They can have, don’t need to have lists of arguments and don’t have to declare their variables. But they are hard to reuse in other programs.
Write an external subroutine which nicely prints real matrices with 2-place decimals. This is a version of the internal integer printer of Classwork 14.2(4).

!c15_2_1printer.f95
subroutine printer(b,n,m)
  real:: b(n,m) ! n = # rows, m = # columns
... delete this, write the subroutine
endsubroutine ! always print decimals with two places
program printer_test
  real:: a(2,2)
a(1,:)=(/10.25,200./)
a(2,:)=(/3.,4.5/)
... delete this, call printer to print the 2x2 matrix a
endprogram
Rewrite the above subroutine `printer` as an internal subroutine with the two lines

```
subroutine printer(b)
real::b(:,:);integer::n,m
n=size(b,1);m=size(b,2)
```

Also appropriately change the call `printer` statement.
Write an internal subroutine **array_read** which declares `a` to be of dimension(:,::) and allocatable, then asks the user to enter the n×m array `a` of reals. First ask for `n`, `m`, then allocate space for `a` and then ask for and read `n` rows of `m` entries into `a`.

If `n` = 2, `m` = 3, and `a` = (the real-number version of)

\[
\begin{pmatrix}
1 & 1 & 1 \\
2 & 2 & 2 \\
\end{pmatrix}
\]

1) Use **array_read** to read in the array `a` one row at a time. (2) Call **printer** to print the matrix `a`.

```fortran
!c15_3_4array_read.f95

... delete this, copy the printer subroutine here

program test_array_read
real, ____ , ____ :: a   !need dimension, need allocatable
call ____ , ____ :: a
call array_read

call printer(a,n,m)
contains
 subroutine array_read
   ... delete this, write the subroutine
endsubroutine
endprogram
```
**Numbers in Different Bases**

The digits of a base-$b$ number will be written as a sequence $\text{seq}$ of base $b$ digits declared with `integer::seq(200)`.

If the base-5 sequence is $[1, 3, 4, 2]$, the decimal value is $d = 1 \times 5^3 + 3 \times 5^2 + 4 \times 5^1 + 2 \times 5^0 = 222$.
For $5 = 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$, the base-2 sequence is $[1,0,1]$.
For $2 = 2 \times 3^0$, the base-3 sequence is $[2]$. Note that the remainder $\text{mod}(3, 2)=2$ and the quotient $\text{quo}(3, 2)=0$.

For $d=2345 = 2 \times 10^3 + 3 \times 10^2 + 4 \times 10 + 5$, the base-10 sequence of digits is $\text{seq} = [2, 3, 4, 5]$. $\text{mod}(2345, 10)=5$ and $\text{quo}(2345, 10)=234$. The last digit 5 is $\text{mod}(d, 10)$, the previous digits $[2,3,4]$ come from $\text{quo}(d, 10)$. For any base $b$, the last digit is $r=\text{mod}(d, b)$, the previous digits come from $q=\text{quo}(d, b) = d/b$. If $\text{quo}(d, b)=0$, the sequence is just $[r]$. 
Given an integer \( d \), find its base-\( b \) sequence \( \text{seq} \) of digits and the length \( n \) of \( \text{seq} \). Instead of allocating \( \text{seq} \), declare it to be long enough (length=200) to handle likely applications.

```
c15_4_4decimal2base.f95
!
d= decimal (base 10) integer, b=new base, 
seq=vector of n base b digits
recursive subroutine decimal2base(d,b,seq,n)
integer::d,b,seq(200),r,q,seqq(200)
r=mod(d,b); q=d/b
 ... fill in the missing lines.
endsubroutine
```

```
program test_decimal2base
integer::d=6,b,seq(200)
do b=2,8
 call decimal2base(d,b,seq,n)
print'(i0,a,i2,a,200i1)',d,' in base ',b,' is ',seq(1:n)
enddo; print*
endprogram
```
Write a subroutine `base2decimal(b, seq, n, d)` which, given a base-`b` sequence `seq` of `n` digits, finds its integer `d`. If `b=2` and `seq = [1,1,0]` then `d = 1 \times 2^2 + 1 \times 2 + 0 = 6`.

```fortran
!base2decimal.f95
!d= decimal (base 10) integer, b=new base,
!seq=vector of n base b digits
subroutine base2decimal(b, seq, n, d)
integer:: d, b, seq(200)
d=0
 do i=___,___ ;d=d+seq(___)*b**(___);enddo
endsubroutine
program test_decimal2base
integer:: d, b, seq(200), n=3
seq(1:n)=[1,1,0]
do b=2,10
call base2decimal(b, seq, n, d)
print*,'"Base Sequence Value"
print'(i4,i5,2i1,i8)',b,seq(1:n),d
enddo; print*
endprogram
```
**Polynomials**

As with Matlab, polynomials are written in the order of increasing powers. Instead of \( x^2 - 2x + 3 \), write \( 3 - 2x + x^2 \). Represent \( (3)x^0 + (-2)x^1 + (1)x^2 \) with the vector \( p=[3, -2, 1] \) of its coefficients. Declare it as \texttt{real::p(0:2)}\, , since the degrees range from 0 to 2. Thus \( p(0)=3 \), \( p(1)=-2 \), \( p(2)=1 \). Write 0 for any missing coefficient: write \( x^2 - 1 \) as \( -1 + 0x + 1x^2 \) with vector \( p=[-1, 0, 1] \) of degree \( n=3 \). Evaluating a polynomial \( p \) at a real \( r \) means the result of substituting \( r \) for \( x \). \( x^2 - 2x + 3 \) evaluated at \( x=10 \) is \( 10^2 - 2(10) + 3 = 100 - 20 + 3 = 83 \)

**Classwork:** 15.1(3) magnitude 15.2(2) printer 15.3(4) array_read 15.4(4) decimal2base

**Email:** dale@math.hawaii.edu **Subject:** 190 c15(13)

**Homework:** 15.1 distance 5.2 printer2 15.3 base2decimal

Sent separately.
Write a real function \texttt{polyeval} \((p, n, r)\) which evaluates a polynomial \(p\) of degree \(n\) at \(x = r\).

\begin{verbatim}
!c16_1_2polyeval.f95
!p=a polynomial of degree n, polyeval = p(r)
real function polyeval(p,n,r) result(y)
real::p(0:n),r
y=0
do i=____,____ ;y=y+p(____)*r**____;enddo
endfunction
program test_decimal2base
real::p(0:2),polyeval,r,y
p=[-1,0,1] !x^2-1
do i=1,4
 y=polyeval(p,2,real(i))
 print*,'r=',real(i),'p=x^2-1, value=',y
 enddo; print*
endprogram  !change print* so decimals have two places
\end{verbatim}
Write a subroutine \texttt{polyproduct}(p, n, q, m, r) which gives the coefficient vector for the polynomial \( r \) which is the product of the two polynomials \( p \) of degree \( n \) and \( q \) of degree \( m \).
subroutine polyproduct (p,n,q,m,r)
real :: p(0:n), q(0:m), r(0:n+m)
r(0:n+m)=0
do i=0,n; do j=0,m
  r(i+j)=r(i+j)+ ______
enddo; enddo
endsubroutine

program test_polyproduct
real :: p(0:1), q(0:2), r(0:4)
p=[1, -1] ! p=(-1+x), q=(-1+x)^2, r=(-1+x)^4
print *, "(-1+x), (-1+x)^2, (-1+x)^4"
print '(a,2f4.0,a)', '(p(i),i=1,0,-1)', ']
call polyproduct (p,1,p,1,q)
print '(a,3f4.0,a)', '(q(i),i=2,0,-1)', ']
call polyproduct (q,2,q,2,r)
print '(a,5f4.0,a)', '(r(i),i=4,0,-1)', ']
endprogram
An estate has $M$ dollars which is to divided among $n$ heirs $1, 2, 3, \ldots, n$. Let $a = [a(1), a(2), \ldots, a(n)]$ where $a(i)$ is the amount (in integer dollars, no change) the $i$th heir gets. Write a subroutine `estate(M, n, a)` which calculates $a$ by dividing the $M$ dollars among the $n$ heirs (at most 12 heirs) so that all get the same amount plus/minus one dollar, $a(i)$ equals $a(j)$ plus or minus 1. $[3, 4, 4], [4, 3, 4]$ are acceptable divisions of $\$11$ among 3 heirs; but not $[3, 3, 5], [5, 4, 2]$. Find the quotient $q$ and remainder $r$ of $M/n$. Give each heir $q$ dollars. Then give $r$ of the $n$ heirs one more dollar.
subroutine estate(M,n,a)
!
integer::M,n,a(n),r,q

... finish the subroutine
endsubroutine

program test_estate
integer::M,n,a(12)
do M=11,13; do n=3,4
   call estate(M,n,a)
print'(a,i2,i2,a,20i2)','$',M,n,' heirs, each gets ',a(1:n)
endo; enddo
endprogram

Homework 16.2(4) estate2.f95

e-mail: dale@math.hawaii.edu subject line: 190 h16.2(4)

Write a subroutine estate2(M,n,a) as in the classwork problem but without quotients or remainders (no mod) and using only one do-loop do j=1,M where j ranges over the M dollars in the estate. Initially set the vector a to all 0's. Give each heir in turn one dollar until the money runs out. Let i be the heir whose turn it is to get a dollar. Start with heir i=1. After heir i has been given a dollar, the next heir to get
a dollar is heir \( i+1 \), unless \( i=n \) in which case the next heir is 1. Include the test_estate2 program of CLASSWORK 15.3(2). Writing a program that works isn't enough, your code must use the steps just described: give each heir one dollar in turn until the money runs out.

!estate2.f95
subroutine estate2 (M, n, a)
!M = inheritance dollars, n= # heirs, a(i)= ith heir's inheritance.
integer::M,n,a(n)
i=1; a=0
... finish the subroutine, no mod, no quotients
endsubroutine
program test_estate2
integer:: M,n,a(12)
do M=11,13; do n=3,4
  call estate2 (M,n,a)
print'(a,i2,i2,a,20i2)','$',M,n,' heirs, each gets ',a(1:n)
enddo; enddo
endprogram

Don’t write more than 10 lines without testing. To test your most recent lines, you may need to add a print
statement to see the results. After testing, delete these lines or comment them out.

**COMMON ERRORS**

Failure to declare the function in the program that uses it. If the value of variable is garbage, say 3552748, quite likely it’s value was never set in the program.

Before writing code, do the steps by hand.

For hard problems, start with a simplified version and evolve it one step and test at a time. Think about what steps should be solved first. Typically, do the printing part first (you can’t see what is happening without printing) and do the user input part last (testing takes longer when you have to supply input). If you are spending more than an hour on a program, maybe you should start over and do it one-step-and-test at a time.

When declaring a function or vector or a matrix it is useful to add a comment which says what it is. Instead of

```cpp
real::a(4,4)
```

write

```cpp
real::a(4,4) !matrix of scores + column of totals + row of totals
```

or use a more descriptive name, say `scores`, instead of `a`.

```cpp
real::scores(4,4)
```
ENTERING LONG FORMULAS

For complicated formulas, define them inside out to assure the parentheses are correct. Consider the following complicated formula:

\[
\sqrt{\frac{1}{a_1^2} + \frac{1}{a_2^2} + \ldots + \frac{1}{a_n^2}}
\]

Start with the vector \( a = [a_1, a_2, \ldots, a_n] \)
Square it: \( a**2 \)
Take the reciprocal: \( 1/(a**2) \)
Sum it: \( \text{sum}(1/(a**2)) \)
Take the square root: \( \text{sqrt}(\text{sum}(1/(a**2))) \)

Since we added a matching pair of ( )’s at each step, the number of left (’s will equal the number of right )’s.

DEBUGGING (Correcting programs)

First check that there are no grammar errors:
Once grammar mistakes are corrected, compilation gives
Exit code: 0. If it runs but gives the wrong answer, you have a logical mistake, a \textbf{bug}. Here is a faulty function for determining primes along with a test program.
To see what is happening inside your function `prime`, add debugging statements to your program which print the values of your variables at each step of your program. The statements in green are the debugging statements. Once the program is corrected, delete them. <ctrl-q> uncomments.