Vectors

• You have probably been introduced to vectors in either a mathematics, a physics, or an engineering class. However, you may have just used vectors in $\mathbb{R}^2$ and $\mathbb{R}^3$.

• A general vector is simply an ordered $n$-tuple; a general vector in $\mathbb{R}^n$ is denoted by

$$\vec{x} = (x_1, x_2, \cdots, x_n) \in \mathbb{R}^n$$

where the $x_i$ are the components of the vector.

• Oftentimes we have a lot of data to store and manipulate; to solve this problem we need a data structure. Vectors or one-dimensional arrays are an ideal means to handle this.

• For example, we could have 100 data points $(x_i, y_i)$, $i = 1, 100$ and we could store the $x$-values in the vector $\vec{x} = (x_1, \cdots, x_{100})$ and the $y$-coordinates in $\vec{y} = (y_1, \cdots, y_{100})$. Alternately we could store the data in an $n \times 2$ array but we will look at arrays of dimension greater than one later.
Another example we could consider is the case of deterministic numerical integration formulas where we have integration points and weights and we store these in two separate vectors or one-dimensional arrays instead of having variables for each weight and point such as \texttt{weight1, weight2}, etc.

Another important use for arrays is when we approximate differential equations. In this situation we will overlay our domain with a finite grid or mesh and we will obtain an approximation to the solution of the differential equation at each mesh point.

Before we proceed to more complicated problems in scientific computing, we need to be able to store and use information in vectors. Later we will learn how to manipulate two dimensional arrays or matrices.
• What do we need to learn how to do in Fortran?
  – how to allocate storage for a 1-D array (i.e., a vector) when we compile
  – how to allocate storage for a 1-D array (i.e., a vector) during execution
  – how to set the entries of a vector
  – how to access an element or elements in an array
  – how to manipulate vectors; for example, do we have to write a routine for adding vectors or taking their dot product or does fortran have this capability.
Allocating Storage of One-Dimensional Arrays at Compiling

- As an example, suppose we have 100 floating point values (say, for example, daily temperatures) that we want to store in an array called \textit{temperatures}.

- When we declare the variable \textit{temperatures} which is the one-dimensional array or vector of length 100 we must tell the compiler to set aside 100 memory locations for this real array. This is done by using the \texttt{dimension} specifier. For example,

\begin{verbatim}
real(prec), dimension (100 ) :: temperatures
\end{verbatim}

This statement sets aside 100 memory locations when the program is compiled. It allows you to access elements 1 through 100 of this array.

- An equivalent statement would be

\begin{verbatim}
real(prec), dimension (1: 100 ) :: temperatures
\end{verbatim}

where we explicitly indicated the range of the indices of the vector.
• This statement could also be replaced by the following two statements where we dimension the array by a variable that has been specified in a parameter statement.

```fortran
integer, parameter :: n=100
real(prec), dimension (n) :: temperatures
```

• This is useful when we have several arrays dimensioned by the same values; to change the size of the arrays we simply redefine $n$.

• Remember that all entries of an array must be of the same type. The above example is a one-dimensional array whose components are all floating point numbers. However, components of arrays can be of other types too.

• Our arrays can be integers as well as real numbers. For example,

```fortran
integer, parameter :: n=100
integer, dimension(n) :: row
```

creates a vector of length 100 whose components are all integers.
• Similarly an array could consist of character strings. For example, we could set a vector of length 12 whose entries are the months of the year, i.e., each entry is a string.

    character(len=20), dimension(12) :: months

• We can also have a vector of logicals although this is less commonly used in scientific computing.

    logical, dimension(10) :: t_f
Accessing an Element of an Array

- When we use an array such as the one we defined by

  \[ \text{real(prec), dimension (n)) :: temperatures} \]

  we can refer to the entire array by using the array name \text{temperatures}; this is often used in situations such as passing it as an argument to a subprogram.

- In mathematics, to denote an element of a vector we use a subscript (or occasionally a superscript). For example the \(i\)th component of the vector \(\vec{x}\) is denoted \(x_i\). We can also access each individual component of a one-dimensional array by appending an index enclosed in parentheses to the array name. For example the statement

  \[ \text{temperatures}(5) = 88.5 \]

  sets the fifth element in the array to be 88.5
Accessing Sections of an Array

- Sometimes we may want to access a section of a given array. For example, we might want to print the first 10 temperatures in our array or print the temperatures in positions 20 through 30 of our temperature array.

- Just as we could use the : notation in the declaration of arrays, we can use it to get portions of an array. Recall that we have encountered the : notation before when we looked at the case construct.

- Suppose we want to initialize a vector $\vec{a}$ of length $n$ to zero. Of course this could be done by the following do loop:

  ```
  do i = 1, n
    a(i) = 0.0
  end do
  ```
• However, Fortran 90 allows us to use the simpler notation
  
  \[ a(1:n) = 0.0 \]

  where the \(1:n\) notation is an implied do loop.

• The statement
  
  \[ a = 0.0 \]

  also works but I prefer the one where we explicitly list the range of components to be set to zero.

• The notation \((1:n)\) is not only simpler but it allows the compiler to do this more efficiently if the architecture permits.

• If we wanted to set the first 10 (where \(10 < n\)) entries in the vector \(a\) to zero then we write
  
  \[ a(1:10) = 0.0 \]

• If we want to set the fifth through tenth entries to zero we use
  
  \[ a(5:10) = 0.0 \]
• We can also specify an increment in this notation. For example, if we want to set the 1, 3, 5, 7, 9, entries to zero we could write

\[ a(1:9:2) = 0.0 \]

or equivalently

\[ a(1:10:2) = 0.0 \]

where the final \( :2 \) is the increment.
Writing out a One-Dimensional Array

- If we want to write (in an unformatted write) the entire vector \( a \) to the screen then we could simply say

\[
\text{print } *, \ a(1:n) \quad \text{or} \quad \text{write}(\ast,\ast) \ a(1:n)
\]

Once again we could also omit \((1:n)\) and it will still work.

We can also use the syntax (which we used in Fortran 77)

\[
\text{write}(\ast,\ast) \ ( \ a(i), \ i=1,n) 
\]

All of these commands will produce output in a single line (if possible, otherwise wraps to the next line) as

\[
0.000000000 \quad - 1.20000000000 \quad 3.00000000000
\]

- If we want to print out positions 20 through 30 of our temperature array in an unformatted write then we could say

\[
\text{print } *, \ \text{temperatures}(20:30)
\]
• We can also specify an increment in this notation. For example, if we wanted to print every other temperature from 1 to 10 we could use

\[
\text{print } *, \text{ temperatures(1:10:2)}
\]

where the final :2 indicates the increment.

• If we want to write out the entire vector \( \mathbf{a} = (1, 3, 4, -3, 6) \) of length 5 using the format \( f12.5 \) for each component, we write

\[
\text{write(*,'(5f12.5)') a(1:5)}
\]

which produces the output

\[
1.00000 \quad 3.00000 \quad 4.00000 \quad -3.00000 \quad 6.00000
\]

Note that if the vector has been dimensioned by \( n \) where \( n \) has been set in a parameter statement we cannot write

\[
\text{write(*,'(nf12.5)') a(1:n)} \quad ! \text{doesn't work}
\]

We must use an actual number in front of \( f12.5 \).
• If we use the write statement

\[
\text{write(5, '}(f12.5)') \ a(1:n)
\]
then it produces the output where each component is on a single line; for example

1.00000
3.00000
4.00000
etc.

• If we use the write statement

\[
\text{write(5, '}(2f12.5)') \ a(1:n)
\]
then it produces the output where two components are on a single line; for example

1.00000 3.00000
4.00000 -3.00000
etc.
• If I have a vector of say length 100 then I don’t print it out with a format of \texttt{100f12.5} but instead think that with this \texttt{f12.5} format I can write say 8 numbers to a line so I use \texttt{8f12.5}. Then the first 8 entries are on the first line, next 8 on the next line, etc. If the length of the vector is not divisible by 8 then it simply leaves blanks on the last line.

• The syntax for read statements are analogous.
Setting the values of an array

- There are many ways that the entries of an array can be set.

- You could read in their values, either from the screen or from a file. For example, to read in data from a file associated with unit 5 for the vector $x$ of length $n$, we could use (with unformatted read statement)

  `read ( 5, * ) x (1:n)`

  where we have used the : syntax.

- Alternately, we could use the syntax (which we had to use in Fortran 77)

  `read ( 5, * ) (x(i), i=1, n)`

  Note that we had to include the expression $x(i), i=1, n$ in parentheses; otherwise we will get an error.

- We could simply “hardwire” it into the code: $x(1)=2.0; x(2)=-3.4$, etc. Of course this is not practical for large arrays.
• An alternative is to assign the complete array by listing all of the elements between the symbols (/ and /). For example, for a vector of length 5

\[ x = (/ 2.0, -3.4, 7.1, 1.7, .65/) \]

**Warning:** There is no space allowed between the ( and the /.

• More commonly our entries are found as the result of some numerical computation and we are simply using the array for data storage.
A Second Look at our Routine to Print a Date

- Recall that when we wrote the routine to print out a date where we actually wrote off the name of the month we included a case construct with 12 cases. There is an easier way to do this now that we know about one-dimensional arrays.

- First let's make an array (declared as a character) of dimension 12 which contains the names of the month.

```fortran
character(len=20), dimension(12) :: months
months = (/ 'January', 'February', 'March', 'April', 'May', 'June', 'July', 'August', 'September', 'October', 'November', 'December' /)
```

Recall that the member of the date class was called `mdy` with components `month, day, year`. So to print we simply say

```fortran
print *, months ( mdy % month), mdy % day, mdy%year
```
An Example using a 1-D Array

Suppose we want to write a program which creates the first $n$ numbers in the Fibonacci sequence and stores them in an array for later use. In this case our array is an integer array of length $n$ where $n$ has been set as a parameter. Recall that the first two terms of the Fibonacci sequence are zero and one and the subsequent terms are found by summing the previous two terms.

```
program fibonacci
implicit none
integer, parameter :: n = 10
integer, dimension (n ) :: fib_sequence
integer :: k

fib_sequence(1) = 0
if ( n > 1 ) then
    fib_sequence(2) = 1
    do k = 3, n
```
fib_sequence(k) = fib_sequence(k-1) + fib_sequence(k-2)
end do
else
print *, "no terms in sequence computed"
end if
print *, fib_sequence(1: n )
end program fibonacci
In our examples, we have used positive subscripts for the array, e.g., $1, 2, 3, \ldots, n$ since this is what we commonly use.

Fortran allows a subscript to be any integer value: positive, negative or zero.

If we had points \( \{x_0, x_1, \ldots, x_n\} \) and we wanted to preserve this notation, then we could dimension our array \( x \) by

```fortran
integer, parameter :: n = 10
real, dimension(0:n) :: x
```

to allocate \( n + 1 \) storage locations.

The syntax to dimension a real array, say \( x \) where you begin indexing at say -10 and go to 5 (i.e., \( x_{-10}, x_{-9}, \ldots, x_{-1}, x_0, x_1, \ldots x_4, x_5 \) is

```fortran
real(prec), dimension ( -10 : 5 ) :: x
```

You can then access an element with syntax such as \( x (-5) \) or \( x(0) \). Note that when you use the syntax \( \text{dimension} (-10,5) \) it includes 0.

It is probably best not to do too much of starting a vector at a negative index because it is not how we actually think about vectors in most cases.
Run time arrays – Allocatable Arrays

– Sometimes we don’t know what the size of the array should be dimensioned until we do some calculations or read in the value to dimension it.

– Recall that the way we have dimensioned arrays so far, the compiler sets aside the storage at the time of compilation.

– In Fortran 77 if we didn’t know the size of the array we used to just estimate the size (and usually make it a little larger for safety).

– Fortran 90 allows us to allocate storage to an array during execution.

– We still must declare the array as real, integer, etc. and we must tell the compiler that we are going to allocate its size during execution. Also we must declare the shape of the array. The declaration statement now becomes
real(prec), dimension (:), allocatable :: temperatures

- The specifier `allocatable` tells the compiler we will be setting aside memory for this array during execution.

- The specifier `dimension (:)` tells the compiler that this is going to be a one-dimensional array, i.e., it gives the shape of the array. Later we will be looking at two-dimensional and higher arrays.

- Being able to allocate and release memory during execution is an extremely valuable tool because it allows us to save on storage.

- **How do we allocate the memory during execution?**
  * through the use of the `allocate` statement

- **How do we release memory during execution?**
  * through the use of the `deallocate` statement
The allocate/deallocate statements

– When we declare an array as allocatable, then we are telling the compiler that we will be allocating space during execution.

– Suppose that we have done some calculations and have determined that the entries of our array temperatures should range from 1 to \( n \) where \( n \) is the result of the calculations. Then to allocate the array we use

\[
\text{allocate ( } \text{temperatures (n) )}
\]

or equivalently

\[
\text{allocate ( } \text{temperatures (1:n) )}
\]

– If we want to allocate more than one array we can put them in separate statements or we can put them in one allocate statement by making a list such as

\[
\text{allocate ( } a(1:n), b(1:n) \ )
\]

– Each of these arrays must be initially declared as allocatable and we must use the dimension specifier \textit{dimension ( : )} . Otherwise you will
receive an error message.

– **Where do we put the allocate statement?** Basically, anyplace in the executable statements before we need to access the array.

– Since storage is an issue when we do large scale computations, it is good programming practice to release the storage of an array when we no longer need it. If we don’t do this, the storage will be used until the execution of the program is terminated.

– To do this, we use the `deallocate` statement. For example,

```plaintext
deallocate ( temperatures, a, b )
```

deallocates the 3 arrays listed. Note that we do not have to indicate their size when we deallocate them.
Our Fibonacci example revisited

Here we are going to rewrite our example where we now are going to make our array containing the sequence allocatable.

```fortran
program fibonacci
implicit none
integer, dimension (:), allocatable :: fib_sequence
integer :: k, n_terms

print *, " enter number of terms in Fibonacci sequence" 
read *, n_terms 
allocate ( fib_sequence(1:n_terms) )
fib_sequence(1) = 0
if ( n_terms > 1 ) then
    fib_sequence(2) = 1
    do k = 3, n_terms
        fib_sequence(k) = fib_sequence(k-1) + fib_sequence(k-2)
    end do
end if
```

```
else
  print *, "number of terms in sequence set to 1"
end if
deallocaat ( fib_sequence)
end program fibonacci
Operations on Vectors

- Addition and subtraction of vectors.
  - This is done by adding/subtracting the corresponding components, i.e., for $\vec{a}, \vec{b} \in \mathbb{R}^n$
    $$\vec{c} = \vec{a} + \vec{b} = (a_1 + b_1, a_2 + b_2, \cdots, a_n + b_n)$$
  - In fortran we can write
    ```fortran
    c = a + b
    ```
    where the arrays have been dimensioned by
    ```fortran
    real, dimension(n) :: a, b, c
    ```
  - However, I prefer the following so that we are explicit that these are arrays and which entries we are modifying.
    $$c(1:n) = a(1:n) + b(1:n)$$
- If \( \mathbf{a}, \mathbf{b}, \mathbf{c} \) are vectors of length \( n \) and we just want to add the first \( m \) components with \( m < n \) then we can write
  \[
  \mathbf{c}(1:m) = \mathbf{a}(1:m) + \mathbf{b}(1:m)
  \]

- **Multiplication by a scalar**
  - To multiply the vector \( \mathbf{a} \) by a scalar \( k \), we simply multiply each component by \( k \), i.e.,
  \[
  \mathbf{c} = k\mathbf{a} = (ka_1, ka_2, \cdots, ka_n)
  \]
  - In fortran we write
    \[
    \mathbf{c} = k*\mathbf{a}
    \]
    or more explicitly
    \[
    \mathbf{c}(1:n) = k*\mathbf{a}(1:n)
    \]
  - Here \( k \) needs to be declared as a real number since the array \( \mathbf{a} \) was declared as real (we don’t use mixed arithmetic).
• The dot product of two vectors
  – Recall that the dot or scalar product of two vectors is a scalar which can be found by summing the product of their components, i.e.,
    \[
    \vec{c} = \vec{a} \cdot \vec{b} = \sum_{i=1}^{n} a_i b_i = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n
    \]
  – This dot product is also useful when we want to determine the Euclidean length of a vector, i.e., a measure of its size
    \[
    \sqrt{\vec{a} \cdot \vec{a}}
    \]
    We will be especially interested in this length if \( \vec{a} \) represents a vector whose components represent components of the error we are making.
  – Fortran has an intrinsic subprogram for computing the dot product of two one-dimensional arrays of the same length.
    \[
    c = \text{dot_product} \left( a, b \right)
    \]
  – If arrays \( a \) and \( b \) are not of the same length, then you will get an error message.
  – If arrays \( a \) and \( b \) are not one-dimensional arrays you will get an error.
Fortran 90 has several other intrinsic functions for processing one-dimensional arrays.

- **maxval (a)**  This returns the maximum (not in absolute value) value in the given array. For example
  
  ```fortran
  a= (/ 1.0, 3.0, -2.0, -5.0, 3.0 /)
  value = maxval (a)
  ```
  
  returns the value 3.0

- **maxloc (a)**  This returns a one-dimensional array containing one element whose value is the position of the first occurrence of the maximum value in a. This is an integer. For example
integer, dimension(1) :: location
location = maxloc(a)

returns the integer location 2 since this is the first occurrence of the max value 3.0

• \texttt{minval(a)}, \texttt{minloc(a)} are defined analogously.
• \texttt{product(a)} returns the product (a scalar) of the elements of the array, i.e.,

\[ a_1 a_2 \cdots a_n \]

• \texttt{size(a)} returns the number of elements in the array. This can be useful because it can give us the option of not passing the size of a one-dimensional array to a subprogram.
• \texttt{sum(a)} returns the sum (a scalar) of the entries of the array, i.e.,

\[ a_1 + a_2 + \cdots + a_n \]
Other Operations on Vectors

• For vectors in $\mathbb{R}^3$ we can take the cross product which results in another vector

$$\vec{c} = \vec{a} \times \vec{b} = (a_2b_3 - a_3b_2, a_3b_1 - a_1b_3, a_1b_2 - a_2b_1)$$

Fortran does not have an intrinsic routine for the cross product.

• We can find the angle, $\theta$, between two vectors from the formula

$$\cos \theta = \frac{\vec{a} \cdot \vec{b}}{|\vec{a}| \ |\vec{b}|}$$

• There are a myriad of vector identities that can also be determined.

• We have seen that we can find the Euclidean length of a vector by taking the square root of the dot product of the vector with itself. There are other ways to quantify the length of a vector.
Classwork

• Write a program `test_vectors.f90` to set $n$ as an integer parameter, say $n = 4$ and dimension real arrays $a, b, c$ by $n$.

• Set the values of array $a$ to be $(1., -2., 3., -5.)$ by using the syntax $a = (/\text{one, -two, \ldots/})$

• Print out only the first 2 components of $a$ in an unformatted write.

• Print out all of $a$ using the format $4f12.4$

• Print out all of $a$ using the format $3f12.4$. Note the difference is the output.

• Set the one-dimensional array $b$ to zero. Then reset the fourth entry to one and print out.

• Dimension an array $x$ starting at 0 and having 5 storage locations. Set $x = (1., 2., 3., 4., 5.)$. Print out $x(0)$ and $x(4)$.

• Make one of your arrays allocatable and read in the value for the dimension of the array and then allocate it.
• Add code to set some values for the vector $\vec{b}$; set $\vec{c} = 2\vec{a} - \frac{1}{2}\vec{b}$ and print out $\vec{c}$

• Add code to modify the vector $\vec{a}$ so that its first three components are multiplied by 2. Print out $\vec{a}$ to verify that the change has been properly made.

• We defined a vector $x$ with dimension $(0:4)$. Add this to $\vec{a}$ using the notation $\vec{a} + x$. Why do you get an error?

• Set $\vec{c} = \vec{a} \cdot \vec{b}$ and print out $\vec{c}$.

• Add code to compute the length of $\vec{c} = \sqrt{\vec{c} \cdot \vec{c}}$ and print the length.