Review of Vectors

• We first stated that vectors or one-dimensional arrays will be a useful data structure. All entries of the array must be of the same type; e.g., integer, real, etc.

• Syntax for setting aside memory for our one-dimensional array:
  – Compile time examples (arrays with \( n \) entries)

\[
\text{real(prec), dimension(1:n) :: a}
\]
equivalently:

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

\[
\text{real(prec), dimension(0:n-1) :: b}
\]

\[
\text{real(prec), dimension(-1:n-2) :: c}
\]

where \( n \) has been defined.
Run time example

```fortran
real(prec), allocatable, dimension(:) :: a

once n has been defined allocate ( a (1:n) )
and to release the storage deallocate ( a )
```

- We accessed a particular element in our array by, for example
  ```fortran
  print *, a(5)
  or a section of an array
  print *, a(10:20)
  ```

- We can set the values of an array in several ways. We can read them from a file, hard wire them (as `a(1)=1.2`) or assign all entries using an intrinsic constructor
  ```fortran
  a = (/ 1.0, 2.0, -3.0 /)
  ```

- Next we became familiar with some simple operations with vectors where in Fortran 90 the standard operators `+`, `*`, `-` have been intrinsically overloaded.
  ```fortran
  c(1:n) = a(1:n) + b(1:n)
  ```
\[ c(1:n) = k \times a(1:n) \]

- We also looked at some intrinsic subprograms in fortran whose arguments are one-dimensional arrays. These include:
  - `dot_product(a(1:n), b(1:n))`
  - `maxval(a)`
  - `size(a)`
  - etc. where \( a \) is a one-dimensional array.

- In a completely analogous way we can have arrays of higher dimensions.

- For an \( m \times n \) matrix (\( m \) rows and \( n \) columns) we can set aside storage either at compile time or during execution (remember rows before columns)
  
  ```fortran
  real(prec), dimension(1:m, 1:n) :: a
  equivalently: real(prec), dimension(m,n) :: a
  real(prec), allocatable, dimension(:, :) :: a
  ```
  
  once \( m, n \) have been defined
  
  ```fortran
  allocate( a(1:m, 1:n) )
  ```
  
  and to release the storage
  
  ```fortran
  deallocate( a )
  ```

- Similarly we can add any two arrays of the same size using the + operator
and multiply any array by a scalar using $\times$.

- Later we will see how matrix times matrix or matrix times vector operations are achieved.
• An important part of scientific computing is validation that your program is working properly.

• One way to do this is to compare your computational errors to expected theoretical errors or rates of convergence. In the last section for nonlinear equations we looked at the rate of convergence of a method and could compare this with the theoretical rates.

• In many cases we will have an error vector instead of a single scalar error (which we had in our nonlinear problems). From this vector we would like to obtain a single non-negative number.

• There are many ways to do this and which measure of the error you use can depend on what error you are trying to control.

• A vector norm will be a quantification of a vector in some sense.
• Mathematically, how do we denote a norm?
  - Commonly we use the notation $\| \vec{a} \|$
  - This is an extension of the notation we used when the error is a scalar value, i.e., when we just take the absolute value, i.e., $|e|$.
  - A subscript is usually added to indicate which norm, e.g., $\| \vec{a} \|_2$.

• What properties do we want a norm to possess?
  - Since the norm will be some measure of the length of a vector, the norm should always be $\geq 0$ and only $= 0$ if the vector is the zero vector.
  - The norm of a scalar times a vector, i.e., $\| k\vec{a} \|$, must satisfy
    $$\| k\vec{a} \| = |k| \| \vec{a} \|$$
    So if we take a vector and multiply it by -3 then its length should change by 3.
  - The triangle inequality
    $$\| \vec{a} + \vec{b} \| \leq \| \vec{a} \| + \| \vec{b} \|$$
    This inequality derives its name from vectors in Euclidean space where in
If $a$ and $b$ are the sides of a triangle, the length of the hypotenuse is less than the sum of the lengths of the two sides.
Three Common Vector Norms

1. Probably the most commonly used norm is the standard Euclidean length (sometimes called the $\ell_2$ norm):

$$\|\vec{a}\|_2 = \sqrt{\sum_{i=1}^{n} a_i^2} = (a_1^2 + a_2^2 + \cdots + a_n^2)^{1/2}$$

When the vector represents an error and we don’t normalize this by the norm of the solution, then we must normalize by its length, i.e.,

$$\|\vec{a}\|_2 = \sqrt{\frac{1}{n} \sum_{i=1}^{n} a_i^2}$$

Why do we do this? Suppose we have a vector which represents components of our error and all of its components are exactly 0.1; whether the vector has length 10 or 100 we would like this measure of the error, i.e., $\|\vec{e}\|_2$ to be...
the same. This will be the case if we normalize as above. For example, for a vector of length 10
\[(.1^2 + \cdots + .1^2)^{1/2} = (.1)^{1/2} = 0.316 \quad \left(\frac{1}{10}(.1^2 + \cdots + .1^2)\right)^{1/2} = 0.1\]
and for a vector of length 100
\[(.1^2 + \cdots + .1^2)^{1/2} = (.1)^{1/2} = 1.0 \quad \left(\frac{1}{100}(.1^2 + \cdots + .1^2)\right)^{1/2} = 0.1\]
However, if we always measure the error in a relative sense by normalizing by the norm of the solution, then this is not necessary. For example, the relative $\ell_2$ error is
\[
\frac{\|\text{approximate} - \text{exact}\|_2}{\|\text{exact}\|_2}
\]
2. Another commonly used measure is the max norm or the infinity norm
\[
\|\vec{a}\|_{\infty} = \max_{i=1,\ldots,n} |a_i|
\]
3. A third commonly used measure is the one norm

\[ \|\vec{a}\|_1 = \sum_{i=1}^{n} |a_i| = (|a_1| + |a_2| + \cdots + |a_n|) \]

If we draw “the unit ball” in \( R^2 \) (i.e., \( \{(x, y) : \|x\vec{i} + y\vec{j}\| \leq 1\} \)) where we measure length in each of the three norms we get different results. Can you determine which norm is used in each of the sketches?
Example

Consider the vector

$$\vec{v} = \begin{pmatrix} 2. \\ 1. \\ -5. \end{pmatrix}$$

Then the Euclidean norm, called the $\ell_2$ norm and denoted by $\| \cdot \|_2$ is given by

$$\| \vec{v} \|_2 = \sqrt{2^2 + 1^2 + (-5)^2} = \sqrt{4 + 1 + 25} = \sqrt{30} = 5.477$$

The infinity or max norm, denoted $\| \cdot \|_\infty$ is given by

$$\| \vec{v} \|_\infty = \max_{i=1,2,3} |v_i| = \max\{2, 1, | - 5|\} = 5$$

The one-norm, denoted $\| \cdot \|_1$ is given by

$$\| \vec{v} \|_1 = |2| + |1| + | - 5| = 8.$$
• Even though we get different values for each of the norms there is a theorem about the equivalence of vector norms in $\mathbb{R}^n$.

• We first define what we mean by two norms being equivalent.

  **Definition.** The norm $\| \cdot \|_p$ is equivalent (or comparable) to the norm $\| \cdot \|_q$ if there exists positive constants $\alpha, \beta$ such that

  $$\alpha \|x\|_q \leq \|x\|_p \leq \beta \|x\|_q$$

  for all $x \in \mathbb{R}^n$.

• Note that this also says that $\| \cdot \|_q$ is equivalent to $\| \cdot \|_p$ since

  $$\frac{1}{\beta} \|x\|_p \leq \|x\|_q \leq \frac{1}{\alpha} \|x\|_p$$

• For example, by the definition of the $\| \cdot \|_\infty$ and $\| \cdot \|_1$ norms, we see that

  $$\|x\|_\infty \leq \|x\|_1 \leq n \|x\|_\infty$$
• Now we want to write functions to calculate the various norms of a vector.

• Suppose we want to write a routine to calculate the infinity norm of a vector. Recall that we can not use the intrinsic function `maxval` because this returns the maximum value but not in absolute value; however we could use `maxval(abs(a))`. Ultimately we will write a routine to calculate a norm of a member of a vector data structure.

• Analogous to the case of variables passed, the array must be defined in the calling unit and in the subprogram.

• When the subprogram is referenced, the first element of the actual array in the calling program is associated with the first element of the corresponding array in the subprogram. Successive array elements are then associated. Of course like variables, the name in the calling program doesn’t have to match the array name in the subprogram.
To write this routine we of course need to pass the vector whose norm we are going to compute. What about the length of the vector? We could always write

```fortran
function get_max_norm ( x, n)
  integer :: n
  real(prec), dimension(n), intent(in) :: x
where our declaration statement for the vector x is the usual compile-time declaration except for the intent option. It doesn’t matter whether the vector was an allocatable array or not in the calling program.
```

However, Fortran allows us an alternative to passing the size of a one-dimensional array argument to a subprogram. We can simply use an assumed-shape array. For our problem this would be

```fortran
function get_max_norm ( x )
  real(prec), dimension(:), intent(in) :: x
```

It is also allowed to use
real(prec), dimension(*), intent(in) :: x

because this was the syntax used in Fortran 77 because the : didn't exist.

- **Warning:** This works for one-dimensional arrays; for higher dimensional arrays we must specify the actual size (more precisely, we can only use an assumed shape for the last dimension)

- **Warning:** If you need the length of a vector for a do loop then you can not use the size function to get this if you have dimensioned the array in your subprogram with the assumed shape (:). Also when you use intrinsics like `dot_product` with two vectors defined through the assumed shape you must write the command as `dot_product ( a(1:n), b(1:n) )` instead of `dot_product (a, b)` since the subprogram does not have the explicit value of \( n \). The same is true using commands like `maxval`, etc.

- **Warning:** If an array is defined as double precision in the main program then it must be double precision in any subprogram it is passed to; otherwise this will give you a very strange error.

- When we define our vector class we will include the length of the vector as an attribute so that it won’t be an issue with us.
Creating a Vector Class

- Our goal is now to write a fairly complete vector class including constructors, destructors, copy, print and operations.
- Then we want to see how this vector class can be used to solve a standard problem in linear algebra – finding an orthonormal basis.
- To get experience with overloading operators we are going to write our own routines to, for example, add two members of our vector data structure.
- Another reason for doing this is to provide the capability for your code to have a user-friendly premature termination; i.e., instead of the compiler giving you an error, you check and print out your own error message before termination.
- The first decision you have to make in creating a derived data type is to decide what the components or attributes of an object in your data structure need to be.
- **What should the components of an object in the vector data structure be?**
Clearly it should be the vector itself and we have seen that oftentimes the length of the vector is also needed in a routine. When we define the data structure, we don’t know the length of the vector (because we want to use this in a generic way). So we will define our vector data structure as:

```fortran
  type vector
    integer :: length
    real(prec), pointer, dimension(:) :: oned_array
  end type vector
```

- The attribute `length` will give the length of the array.
- The attribute `oned_array` will give the one-dimensional array itself.
- Note that in the array we have used new syntax `pointer`. In the version of `gfortran` the statement
  ```fortran
  real(prec), allocatable, dimension(:) :: oned_array
  ```
  is not accepted as proper syntax although in other compilers (such as `ifort`) it is. We will talk about pointers later but for now in the `type` statement use `pointers` where you think `allocatable`.
• As before, when we declare a member of our vector data type we simply use

```fortran
  type (vector) :: a, b
```

because the members of our data structure contain an array which has already been dimensioned and an integer.

• Now when we have a member of the `vector` class and pass it to a subroutine, we are actually passing the length of the array in addition to the array itself. This will avoid problems about the necessity to pass the size of an array.

**A Manual Constructor**

• This function will have optional arguments: the length of the vector, the components of the vector. The output will be a new member of the vector class.

• If both arguments are present, it will first allocate memory for the component `oned_array` and then set the length of the member of the vector class and set the values of `oned_array` to be the input values.

• If the length is not present maybe we want to set the length arbitrarily to
one, allocate it and print out a warning message.

• The portion of the code where both arguments are present could look like the following.
function make_vector ( n, data_in ) result (vec )

integer, optional, intent(in) :: n ! length of vector
real(prec), optional, intent(in), dimension(:) :: data_in ! optional components of vector
type(vector) :: vec ! output member of vector class

if ( present(n) ) then

    vec % length = n
allocate ( vec % oned_array (1:n ) )

    if ( present ( data_in) ) then

        vec % oned_array = data_in

    else


print *, " vector of length ", n, " is allocated but has no entries"

stop

dend if
Overloading the operator *

• We can think of two ways to use the * operator.
  – Multiplying a member of the vector class times another member of the vector class; i.e., taking the dot or scalar product of two vectors. Here we will add a check to make sure the vectors are the same length.
  – Multiplying a scalar times a member of the vector class.

• If we declare

```fortran
  type(vector) :: a, b
  real(prec) :: lambda
```

and do not overload the operator * then we can NOT write (as we did when we were testing vector operations)

```fortran
  b = lambda * a
```

or

```fortran
  lambda = dot_product (a, b)
```
• Why?
• We could write (which is fairly cumbersome)
  \[ b \ % \ \text{oned\_array} = \lambda \ * \ a \ % \ \text{oned\_array}(1:n) \]
  \[ \lambda = \text{dot\_product}(a \ % \ \text{oned\_array}, b\ % \ \text{oned\_array}) \]
• If we provide routines to overload the operator \(*\) then we can use the shorthand notation above.
• Let’s look at a routine for overloading \(*\) to compute a real times a member of our vector class and output the result in a new array.
• As input it will have the vector and the scalar and as output it will have the resultant vector.
function real_times_vector (lambda, vec) result ( vec_out )

type ( vector ), intent(in) :: vec

type ( vector ) :: vec_out

real(prec), intent (in) :: lambda

integer :: n

n = vec % length

if ( n >= 1 ) then

    if ( allocated( vec_out % oned_array ) .eqv. .false. ) then

        vec_out = make_vector( n )

    end if

end if
vec_out % oned_array ( 1: n) = lambda *vec % oned_array ( 1: n)

else

    print *, "trying to multiply a real times a vector whose size is not >=1"; stop

end if

end function real_times_vector

We add the interface construct to our module:

interface operator (*)

    module procedure real_times_vector

end interface
• We have used a new intrinsic array processing function, \texttt{allocated} which returns true if memory has been allocated to the array and false otherwise.

• Recall that when testing a logical (i.e., true or false) the syntax is \texttt{.true.} not \texttt{true}.

• When testing a logical we must use \texttt{.eqv.} instead of \texttt{==}.

• For example, if we have a real variable \texttt{a} then we use

\begin{verbatim}
if ( a == four ) then
\end{verbatim}

but if we have a logical \texttt{tf} then we can’t say

\begin{verbatim}
if ( tf == .true. ) then
\end{verbatim}

but rather must use

\begin{verbatim}
if ( tf .eqv. .true. ) then
\end{verbatim}

• The statement

\begin{verbatim}
vec_out = make_vector( n )
\end{verbatim}

is equivalent to the statements

\begin{verbatim}
allocate ( vec_out % oned_array (n) ); vec_out % length = n
\end{verbatim}
Testing Overloading $\ast$ for scalar times a member of our vector data structure

- Now we want to test our operator overload to multiply a real times a member of our vector data structure. For example, what will happen in each of the following cases
  
  - $b = 5.\_prec \ast a$
  - $b = 5 \ast a$
  - $b = a \ast 5.\_prec$

  where $a, b$ have been declared as members of the derived data structure vector
Overloading the operator * again

- We can associate several different routines with the operator * if the arguments are different.

- In the previous case we saw that if you type
  
  ```
  b= lambda * a ! lambda real, a vector
  ```

  it will call the routine we wrote but if you reverse the order of `lambda` and `a` then you get an error. To remedy this you can either always remember to write the real number first or you can write a second routine

  ```
  function vector_times_real (vec, lambda) result ( vec_out)
  ```

  and change our interface statement to

  ```
  interface operator (*)
      module procedure real_times_vector , vector_times_real
  end interface
  ```

- Now it doesn’t matter if you put the real number first or second; both work.
• What about using the * for the usual dot product? We can do this by writing a routine

```fortran
function scalar_product ( vec1, vec2 ) result (scalar)

where vec1, vec2 are members of the data type vector and scalar is real. Now the arguments are two members of the data type vector so when we want to use this function we can write

```fortran
type(vector):: a,b
real(prec) :: value

value = a * b
```

where the function scalar_product is called.

• Note that we can’t call our function dot_product because this name is already an intrinsic fortran function.
• We modify our interface statement as follows

```fortran
interface operator (*)
    module procedure real_times_vector, vector_times_real,&
    scalar_product
end module
end interface
```

• Remember that we can do this because each of the two subprograms `real_times_vector`, `vector_times_real` have different sets of arguments; if they had the same it wouldn’t work.
Overloading the operators +, -

- As in the case of the operator * we can overload +, - by writing a routine to add/subtract two vectors and to check that they are of the same length.
- We can also write a routine to add/subtract a real number to each component of the one-d array in a member of the vector class.
- Note that as in the previous case, each function must have different sets of arguments.
- For the scalar plus a vector the order is important so to avoid this we could write two routines, one with the scalar first and the other with the scalar as the second argument.
Function to add a real number to a vector

• Function add_real_to_vector to overload the operator +

• What structure do we want it to have?
  – As arguments, pass in the real first and member of vector class second (we can have an analogous function add_vector_to_real with arguments reversed).
  – Output is our new member of vector
  – If the output result has not been defined, then you need to allocate the space and set the dimension, e.g., make_vector(n)
  – Perform the addition
  – Add the interface statement to beginning of module.
  – Add statements to test_vector to test out your routine.
Overloading the assignment =

- How might we overload this assignment? We could have it as a default in case we wanted our vector to be a scalar of this value or we could use it to set the entire vector to a constant.
- We often want to “zero out” an array or set its values all to some real number so this might be the most useful.
- In this case we would like to overload the assignment = so that we could write

```fortran
  type (vector) :: a
  a = zero
```

instead of writing

```fortran
  a % oned_array (1:n) = zero
```
- We would need the scalar and the vector. However, if we allocate the size of the vector then we would need to pass its length also.
- Should we have a function or a subroutine?
subroutine set_equal_real (vec, lambda,n)
type (vector), intent(inout) :: vec
real(prec), intent(in) :: lambda
integer, intent(in) :: n

and add an interface statement to overload the assignment =, then we get the error

The number of subroutines arguments must be 2 in a defined ASSIGNMENT.

• In order to correctly perform the assignment, the compiler needs to know what is assigned to what so we need to just have two arguments.

• If we take the other approach in this assignment overload (i.e., just to set the vector to have dimension 1 and set it to our scalar) then we could write a separate routine to call to zero out a vector.
• Taking this approach if we have the statements

```fortran
subroutine set_equal_real (vec, lambda )
  type (vector) :: vec
  real(prec) :: lambda
  we still get an error. Why?
• The error we get is

  Standard F90 requires an explicit INTENT(OUT) or INTENT(INOUT) for a generic ASSIGNMENT procedure.
• Clearly the compiler needs to know which quantity is on the right side of the equal sign.
• If we add the intent specifier then it compiles.
• Probably the most useful approach is to write a separate routine, e.g., zero_array which allocates memory and sets the vector to zero instead of overloading an operator.
Overloading the operator `==`

- As in the rational class we wrote, the overloading of this operator is commonly used to test if two vectors are equal.
- In this case the function is a **logical function** whose output is true or false.
- So the two vectors are the input and the output is the logical.
- We first check to see if the length of the two vectors is the same; if not return with a false value. For example
  
  ```fortran
  t_f = .false.
  if ( vec1 % length .ne. vec2 % length) then
    print out an error message and return. Here
    type(vector):: vec1, vec2
  endif
  ```
- If the two vectors have the same length, then we must check to see if all entries of the two vectors are the same.
• This can be done with a loop or we can use the intrinsic logical function `all`.

• Here is how this logical function works. Suppose we have defined two actual vectors (not members of our vector class) say
  \[ a = (1., 2., 3.) \]; \[ b = (1., 2., -3.) \]

  and we use the command

  \[ \text{print *}, \text{all}( a == b) \]

  Then in our case it will print false.

• You can use other expressions than `==`; for example, you can write

  \[ \text{print *}, \text{all}( a >= b) \]

  which checks to see if each entry of \( a \) is greater than or equal to the corresponding entry of \( b \).

• Of course in our routine we can’t say `all( vec1 == vec2 )` because it wants an array as an argument, not a member of the vector class. What do we write instead?
Adding additional functionality to our class_vector

- In our rational class we found that a routine to make a copy of a member of our class is useful.

  function copy (vec_in) result(vec_out)

  Remember that you will need to allocate the one-d array in vec_out.

- A routine to delete a member of the vector class. This should deallocate memory and set the length to zero.

- Routines to calculate various norms of a vector.

- A routine to calculate the maximum/minimum values.

- A routine to print out the vector and its dimension in a pleasing format.

- In Project III you are asked to complete the module class_vector adding this functionality in addition to the previous overloading of operators and assignments that we have discussed. In addition you will write a program to test out all the functionality that you have incorporated in your module.
• The last thing we will do with this module for now is to see how we can use it to perform a Gram Schmidt orthonormalization of a set of linearly independent vectors.

• Next time we will look at the Gram Schmidt orthonormalization and see how we might modify our vector class to create a sparse vector class.