Review of Vector Module

- We defined our data structure as `type vector`
  
  ```fortran
  integer :: length
  real(prec), pointer, dimension(:) :: oned_array
  
  end type vector
  ```

- To add functionality to the class we added
  - A manual constructor
  - A function to multiply a scalar times a vector and overload `*`
  - A function to add a real to a vector and overload `+`
  - Functions to overload operators such as `==`, `>`, etc.

- We discussed other routines we could add such as calculating norms, a destructor, etc.

- If we assume that we have completed the functionality, we now want to see how we can use our vector class to solve standard problems in linear algebra.
After we write our vector class which incorporates a constructor, destructor, and many of the operations we have looked at, we can test each of the operations that we program but we also want to look at common problems in linear algebra using vectors and see how our vector class can be used in these problems.
Matrix times vector operations

• Fortran 90 has a built-in intrinsic function for multiplying a matrix times another matrix or matrix times a vector; it is `matmul`. However, if we use the data structure `vector` then we can’t use this.

• Our first goal is to assume we have an $m \times n$ matrix $a$ (we could write a data structure for this too but at this point we won’t) and we want to multiply a member of our vector class (with length $n$) by this matrix. For example for a $2 \times 3$ matrix

\[
\begin{pmatrix}
1 & 2 & 3 \\
4 & 5 & 6
\end{pmatrix}
\begin{pmatrix}
-1 \\
0 \\
1
\end{pmatrix}
= \begin{pmatrix}
1 \cdot -1 + 2 \cdot 0 + 3 \cdot 1 \\
4 \cdot -1 + 5 \cdot 0 + 6 \cdot 1
\end{pmatrix}
= \begin{pmatrix}
2 \\
2
\end{pmatrix}
\]

• So in general to form $A\vec{x} = \vec{b}$ we have

We can write the general algorithm as

Multiplication of matrix times vector
Given an $m \times n$ matrix $A$ and a $n$-vector $x$,

$$
\text{for } i = 1, m \\
b(i) = \sum_{j=1}^{n} A(i, j) \ast x(j)
$$

We now want to translate this into a routine which will multiply a given $m \times n$ matrix times a member of our vector class with length $n$. 
function matrix_times_vector ( a, x, m ) result ( b )

integer :: m

real(prec), dimension(m,:) :: a

type (vector ) :: b, x

integer :: i, j, n

n = x % length

if (allocated (b % oned_array) .eqv. .false. ) b = make_vector(m)

do i = 1, m
    b % oned_array (i) = zero
    do j = 1, n
        b % oned_array (i) = b % oned_array (i) &
+ a (i,j) * x % oned_array ( j )

end do

end do

Note that in dimensioning the matrix we can’t use the assumed shape size a(:,,:); an assumed shape can only be used for the last dimension (so for a vector it’s okay).
Orthonormal Basis for $\mathbb{R}^n$

- A basis for $\mathbb{R}^n$ is a set of vectors $\vec{v}_j$, $j = 1, \ldots, n$ such that they are linearly independent (i.e., the only linear combination of the $\vec{v}_j$ which equals zero is the when the coefficients all are zero) and span the space $\mathbb{R}^n$ (i.e., any vector $\vec{w} \in \mathbb{R}^n$ can be written as a linear combination of the $\vec{v}_j$).

- Recall that if we want a basis in $\mathbb{R}^n$ we usually choose vectors of the form $(1, 0, \cdots, 0)^T$, $(0, 1, 0, \cdots, 0)^T$, etc. but there are an infinite number of choices for a basis for $\mathbb{R}^n$. 

• For example, in $\mathbb{R}^2$ each of the sets form a basis.

$\{(1, 0)^T, (0, 1)^T\}, \quad \{(1, 2)^T, (2, 3)^T\} \quad \{\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}})^T, -\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2})^T\}$

• Any vector in $\mathbb{R}^2$ can be written as a linear combination of the two vectors. For the first set, it is easy to see since

$$(x_1, x_2)^T = x_1(1, 0)^t + x_2(0, 1)^T$$

but for the other bases we have to show that the linear system (here I have used the second basis)

$$(x_1, x_2)^T = \alpha(1, 2)^T + \beta(2, 3)^T$$

or equivalently

$$\alpha + 2\beta = x_1$$
$$2\alpha + 3\beta = x_2$$

has a unique solution.

• The first and third bases have the specific properties that the Euclidean length of each vector is one and they are perpendicular (orthogonal).
• A basis \( \{ \vec{e}_j \}, \ j = 1, \ldots, n \) for \( \mathbb{R}^n \) is called an orthonormal basis if
  
  - \( \vec{e}_j \cdot \vec{e}_k = 0 \) for all \( j \neq k \)
  - \( \vec{e}_j \cdot \vec{e}_j = 1 \) for all \( j = 1, \ldots, n \).

• Clearly the standard basis \( (1, 0, \ldots, 0)^T, (0, 1, 0, \ldots, 0)^T, \ldots, (0, 0, \ldots, 0, 1)^T \) in \( \mathbb{R}^n \) is an orthonormal basis.

• How do we convert a basis into an orthonormal basis?
  
  - Gram-Schmidt Orthogonalization Method

• Example. Suppose we have two vectors \( \vec{v}^1 = (1, 2)^T \) and \( \vec{v}^2 = (2, 3)^T \) in \( \mathbb{R}^2 \) which form a basis and we want to use them to generate an orthonormal basis.
  
  - Normalizing the vector is easy, we just divide by its length. For example we can make \( \vec{v}^1 = (1, 2)^T \) have length one by dividing each term by \( \sqrt{5} \).
  
  - So our strategy will be to construct two orthogonal vectors, say \( \vec{u}_1, \vec{u}_2 \) (this means \( \vec{u}_1 \cdot \vec{u}_2 = 0 \)) from \( \vec{v}_1, \vec{v}_2 \) and then normalize them to get \( \vec{e}_1, \vec{e}_2 \).
Recall that in \( \mathbb{R}^2 \) the projection of a vector \( \vec{v} \) onto \( \vec{u} \) is a vector given by dropping a perpendicular from \( \vec{v} \) to \( \vec{u} \) and computing the length of this line segment and then multiplying it by a unit vector in the direction of \( \vec{u} \). If \( \theta \) is the angle between the two vectors then the length of this projection, denoted \( \text{proj}_{\vec{u}} \vec{v} \) is found from

\[
\cos \theta = \frac{\text{proj}_{\vec{u}} \vec{v}}{\| \vec{v} \|_2} \implies \text{proj}_{\vec{u}} \vec{v} = \| \vec{v} \|_2 \cos \theta
\]

Note that \( \vec{v} \) can be written as the sum of the projection vector plus a vector that is orthogonal to \( \vec{u} \).

Using the fact that \( \vec{u} \cdot \vec{v} = \| \vec{u} \|_2 \| \vec{v} \|_2 \cos \theta \) we get that the length of the projection is given by
\[ \text{proj}_{\vec{u}} \vec{v} = \|\vec{v}\|_2 \frac{\vec{u} \cdot \vec{v}}{\|\vec{u}\|_2 \|\vec{v}\|_2} = \frac{\vec{u} \cdot \vec{v}}{\|\vec{u}\|_2}. \]

- Then a vector with this magnitude is found by multiplying it by the unit vector \( \vec{u}/\|\vec{u}\|_2 \) so we have
  \[ \text{proj}_{\vec{u}} \vec{v} = \frac{\vec{u} \cdot \vec{v}}{\|\vec{u}\|_2} \frac{\vec{u}}{\|\vec{u}\|_2} = \frac{\vec{u} \cdot \vec{v}}{\vec{u} \cdot \vec{u}} \vec{u}. \]

- To create our orthonormal basis \( \vec{e}_1, \vec{e}_2 \) we set
  \[ \vec{u}_1 = \vec{v}_1 \quad \text{and} \quad \vec{e}_1 = \frac{\vec{u}_1}{\|\vec{u}_1\|_2} \]
  \[ \vec{u}_2 = \vec{v}_2 - \text{proj}_{\vec{u}_1} \vec{v}_2 \quad \text{and} \quad \vec{e}_2 = \frac{\vec{u}_2}{\|\vec{u}_2\|_2} \]

- Note that \( \vec{u}_1, \vec{u}_2 \) are orthogonal because we have subtracted off the part that is not orthogonal; this can be formally shown by
  \[ \vec{u}_1 \cdot \vec{u}_2 = \vec{u}_1 \cdot \vec{v}_2 - \vec{u}_1 \cdot \left( \frac{\vec{u}_1 \cdot \vec{v}_2}{\vec{u}_1 \cdot \vec{u}_1} \vec{u}_1 \right) = \vec{u}_1 \cdot \vec{v}_2 - (\vec{u}_1 \cdot \vec{v}_2) \frac{\vec{u}_1 \cdot \vec{u}_1}{\vec{u}_1 \cdot \vec{u}_1} = 0 \]
For our specific vectors we have
\[ \vec{e}_1 = \frac{1}{\sqrt{5}}(1, 2)^T = \left( \frac{1}{\sqrt{5}}, \frac{2}{5} \right)^T \]

and
\[ \vec{u}_2 = (2, 3)^T - \frac{(1, 2) \cdot (2, 3)}{5}(1, 2)^T = (2, 3)^T - \frac{8}{5}(1, 2)^T = \left( \frac{2}{5}, -\frac{1}{5} \right)^T \]

and thus
\[ \|(\frac{2}{5}, -\frac{1}{5})\| = \frac{1}{\sqrt{5}} \implies \vec{e}_2 = \frac{1}{\sqrt{5}}\left( \frac{2}{5}, -\frac{1}{5} \right)^T \]

Clearly our vectors are orthogonal since
\[ \frac{1}{\sqrt{5}}\left( \frac{1}{5}, \frac{2}{5} \right)^T \cdot \frac{1}{\sqrt{5}}\left( \frac{2}{5}, -\frac{1}{5} \right)^T = 0 \]

This algorithm can be extended to vectors in \( IR^3 \) easily by calculating \( \vec{u}_1, \vec{u}_2 \) as above and when calculating \( \vec{u}_3 \) we just subtract off the component of both \( \vec{u}_1 \) and \( \vec{u}_2 \) in the direction of \( \vec{v}_3 \), i.e.,
\[ \vec{u}_3 = \vec{v}_3 - \text{proj}_{\vec{u}_1} \vec{v}_3 - \text{proj}_{\vec{u}_2} \vec{v}_3 \]

Clearly this can be extended to \( IR^n \) in an analogous manner.
Implementing the Gram Schmidt Orthogonalization Procedure

– We first consider storage. Do we need to have $\vec{v}_i$, $\vec{u}_i$ and $\vec{e}_i$ or can we overwrite $\vec{v}_i$? This actually depends on our application. For example, we might need to save the vectors we input and the $\vec{e}_i$. This way we will need storage for $\vec{v}_i$ and $\vec{e}_i$. If we don’t want to save the initial vectors we could overwrite them with our orthonormal vectors.

– Our strategy will be to compute the orthogonal vectors sequentially and save the original vectors $\vec{v}_i$. Then we normalize. If we use the routine for the situation where we don’t need the original vectors, then we can simply deallocate them after the call to the routine.

– Since we are not overwriting, we set $\vec{e}_1 = \vec{v}_1$ initially.

– We compute $\vec{e}_2 = \vec{v}_2 - (\vec{v}_2 \cdot \vec{e}_1)\vec{e}_1$ or equivalently, set $\vec{e}_2 = \vec{v}_2$ and compute $\vec{e}_2 = \vec{e}_2 - (\vec{e}_2 \cdot \vec{e}_1)\vec{e}_1$.
- We must compute
\[ \vec{e}_3 = \vec{v}_3 - (\vec{v}_3 \cdot \vec{e}_1)\vec{e}_1 - (\vec{v}_3 \cdot \vec{e}_2)\vec{e}_2 \]
which is equivalent to the steps: set \( \vec{e}_3 = \vec{v}_3 \) then set
\[ \vec{e}_3 = \vec{e}_3 - (\vec{e}_3 \cdot \vec{e}_1)\vec{e}_1 \]
and finally set
\[ \vec{e}_3 = \vec{e}_3 - (\vec{e}_3 \cdot \vec{e}_2)\vec{e}_2 \]

- Why is this equivalent because it is not exactly our formula above which in the last term says take the dot product of \( \vec{v}_3 \) with \( \vec{e}_2 \) not the dot product of \( \vec{v}_3 - (\vec{v}_3 \cdot \vec{e}_1)\vec{e}_1 \) with \( \vec{e}_2 \)?

- The reason we write it like this is because these calculations look like a loop where
\[ \vec{e}_3 = \vec{v}_3 \]
\[ \vec{e}_3 = \vec{e}_3 - (\vec{e}_3 \cdot \vec{e}_k)\vec{e}_k \quad \text{for } k = 1, 2 \]

- To continue in this manner, to calculate \( \vec{e}_j \) we need to set
\[ \vec{e}_j = \vec{v}_j \]
and then sum from $k = 1, j - 1$ and compute the value

$$
\vec{e}_j = \vec{e}_j - (\vec{e}_j \cdot \vec{e}_k) \vec{e}_k
$$

**Gram-Schmidt Orthogonalization Algorithm**

Given $n$ linearly independent vectors $\vec{v}_j$,

for $j = 1, n$

$$
e_j = \vec{v}_j
$$

for $k = 1, j - 1$

$$
\vec{e}_j = \vec{e}_j - (\vec{e}_j \cdot \vec{e}_k) \vec{e}_k
$$

end do

$$
\vec{e}_j = \frac{1}{\|\vec{e}_j\|_2} \vec{e}_j, \text{ provided } \|\vec{e}_j\|_2 \neq 0
$$

end do
• Now we want to implement the algorithm using the functionality of our vector class.

• We will write a subroutine in our program test_vector_class.f90 to implement this algorithm.

• The first question that arises is how do we pass in a generic number of vectors. For example, we don’t want our calling statement to look like

```f90
subroutine gram_schmidt(vec1, vec2, vec3, vec4, vec5, ortho1, ortho2, ortho3, ortho4, ortho5, n_vectors)
```

Imagine what it would look like if we had 100 vectors! We want the same calling statement to work whether we have 5 members of our vector class or 100 members.

One way to handle this is to make a one-dimensional array each of whose elements are a member of the vector class. That is, we have an array

```
(vec1, vec2, vec3, ..., vecn)
```

Now each element of this array is a member of our vector class which means it is a one-dimensional array plus an integer length.
• How do declare such one-dimensional arrays whose entries are members of our vector class? In our main program we could have

```fortran
  type(vector) :: ortho (k )
```

which declares the elements of the array `ortho` to be members of our class `vector` and the length of this array to be `n` (assuming it has been previously defined). Of course `k` is not the length of each vector but rather the number of elements we are passing.

• Suppose we have the following subroutine and the declarations

```fortran
  subroutine gram_schmidt ( vectors_in, ortho_vectors, n_vectors)
    use common_data
    use class_vector
    implicit none

    type (vector), intent(in) :: vectors_in (n_vectors)
    type (vector), intent(out) :: ortho_vectors (n_vectors)
    integer :: n_vectors
  end subroutine gram_schmidt
```
Here `vectors_in` contain the linearly independent vectors (each contained in data structure `vector`) we are given (i.e., $v_j$ in our algorithm), `ortho_vectors` contain the resulting orthonormalized set of vectors (i.e., $e_j$ in our algorithm) and `n_vectors` is the number of vectors we have.

- Let’s look at our algorithm line by line. We first have

  ```
  for \( j = 1, n \)
  
  \( e_j = v_j \)
  ```

- Now remember for us, we are working with members of our vector class. We add the do loop and to set $e_j = v_j$ we use the `copy_vector` routine which allocates memory for `ortho_vectors(j)` and sets it equal to `vectors_in(j)`. We won’t use $v_j$ again; remember we said we wouldn’t overwrite the input vectors.
• The next part of our algorithm is

\[
\text{for } k = 1, j - 1 \\
\quad \vec{e}_j = \vec{e}_j - (\vec{e}_j \cdot \vec{e}_k)\vec{e}_k \\
\text{end do}
\]

• This is not performed when \( j = 1 \); the first time it is used is when \( j = 2 \). At that point we already have computed \( \vec{e}_1 \) and have set \( \vec{e}_2 = \vec{v}_2 \).

• Now the dot product \( (\vec{e}_j \cdot \vec{e}_k) \) (when \( j = 2 \) this is \( (\vec{e}_2 \cdot \vec{e}_1) \)) is performed by our routine \texttt{scalar_product} but we have overloaded the operator \* to call this routine. So we might have

\[
\text{dot} = \text{ortho_vectors}(j) * \text{ortho_vectors}(k)
\]

• Now \texttt{dot} is a scalar so if we type \texttt{dot * ortho_vectors(k)} then it will use the routine \texttt{real_times_vector} since we overloaded the operator \*.

• Similarly, to subtract this value from \texttt{ortho_vectors(j)} we use \- since we have overloaded this operator and it will call \texttt{subtract_real_from_vector}.  

• Summarizing, we have the statements

\[
\begin{array}{l}
do \ k = 1, \ j-1 \\
\quad \text{dot} = \text{ortho\_vectors}(j) \ast \text{ortho\_vectors}(k) \\
\quad \text{ortho\_vectors}(j) = \text{ortho\_vectors}(j) - \text{dot} \ast \text{ortho\_vectors}(k) \\
\end{array}
\]

\end{do}

• To make our vector have Euclidean length one we first compute \( \| \vec{e}_j \|_2 \) using a call to our routine \text{compute\_ltwo\_norm} which checks to make sure the resulting norm is nonzero.

• We now form

\[
\vec{e}_j = \frac{1}{\| \vec{e}_j \|_2} \vec{e}_j
\]

by using the operator overload \( \ast \) where we multiply a vector times a real (\text{function vector\_times\_real}) by coding

\[
\begin{align*}
\text{norm} & = \text{compute\_ltwo\_norm} (\text{ortho\_vectors}(j)) \\
\text{ortho\_vectors}(j) & = \text{ortho\_vectors}(j) \ast (\text{one} / \text{norm})
\end{align*}
\]
• Of course we could have written routines to overload `/` too and used that.
• Now let's see the completed subroutine and what our routine would look like if we didn’t use operator overload.
subroutine gram_schmidt ( vectors_in, ortho_vectors, n_vectors)

use common_data

use class_vector

implicit none

type (vector), intent(in) :: vectors_in (n_vectors)

type (vector), intent(out) :: ortho_vectors (n_vectors)

integer :: n_vectors

real(prec) :: dot, norm

do j = 1, n_vectors

    call copy_vector (vectors_in(j),ortho_vectors(j) )
do k = 1, j-1
  
  dot = ortho_vectors(j) * ortho_vectors(k)
  
  ortho_vectors(j) = ortho_vectors(j) - dot * ortho_vectors(k)
  
end do

norm = compute_ltwo_norm ( ortho_vectors(j) )

ortho_vectors(j) = ortho_vectors(j) * (one/norm)

end do

The code is very readable and is quite short.
If we don’t use the operator overloading we would replace

\[
\text{dot} = \text{ortho_vectors}(j) \times \text{ortho_vectors}(k) \quad \text{with}
\]

\[
n = \text{ortho_vectors}(k) \% \text{length}
\]

\[
\text{dot} = \text{dot_product} ( \text{ortho_vectors}(j) \% \text{oned_array}(1:n) ), \&
\]

\[
\quad \text{ortho_vectors}(k)\% \text{oned_array}(1:n) )
\]

We would replace

\[
\text{ortho_vectors}(j) = \text{ortho_vectors}(j) - \text{dot} \times \text{ortho_vectors}(k)
\]

with

\[
\text{ortho_vectors}(j)\% \text{oned_array}(1:n) = \text{ortho_vectors}(j)\% \text{oned_array}(1:n)
\]

\[
- \text{dot} \times \text{ortho_vectors}(k)\% \text{oned_array}(1:n)
\]

The other expressions would have to be modified in a similar way.
• **What is a sparse vector?** This is a vector which has predominantly zero entries. For example

\[
\begin{pmatrix}
0. \\
1. \\
0. \\
0. \\
0. \\
-5. \\
0. \\
0. \\
0. \\
0. \\
\end{pmatrix}
\]

• In storing these vectors we only store the nonzero entries. In this case we would store the vector \((1., -5.)^T\).

• How can we recover the full vector, if we need it to perform operations with
this vector?

- We have to know where the nonzero entries we have stored actually occur in the full vector.

- So what we need is an integer array which associates each element of our stored vector with the row it appears in if it were a full vector. For our example, this pointer array would be

\[
\begin{pmatrix}
2 \\
6
\end{pmatrix}
\]

This says that the first element in the stored array actually occurs in row 2 of the full vector and the second stored entry occurs in row 6 of the full vector. All other entries in the full vector are zero.

- So to define a sparse vector derived type we clearly need (i) the number of nonzero entries in the vector, (ii) the stored vector, (iii) the number of entries in the full vector, and (iv) the integer array indicating which row each stored entry appears in the full vector. With this information we should be able to recover a full vector.
We define the sparse vector data type by

```fortran
type sparse_vector
  integer :: n_nonzero
  real(prec), pointer dimension(:) :: nonzero_entries
  integer :: length
  integer, pointer dimension(:) :: row
end type sparse_vector
```

Now some of the routines, like multiplying a real times our vector, will be easy to write. Others like adding a nonzero real to a sparse vector will create a full vector so its output should be a member of our vector class. However, routines like taking the dot product of two sparse vectors will require considerably more logic because the vectors may possess different nonzero patterns.

Sparse vector arithmetic is useful but the most useful is sparse matrix arithmetic (due to the increase in storage of a matrix over a vector).
A Brief Look at a Sparse Matrix

- What is a sparse matrix? This is a matrix which has predominantly zero entries. For example

\[
\begin{pmatrix}
4 & -1 & 0 & 0 & 0 \\
-1 & 2 & -1 & 0 & 0 \\
0 & -1 & 8 & -1 & 0 \\
0 & 0 & -1 & 4 & -2 \\
0 & 0 & 0 & -2 & 4
\end{pmatrix}
\]

- Matrices with a large number of zero entries arise often in discretizing differential equations.

- In storing these matrices one option is to only store the nonzero entries. In this case we see that the matrix has three diagonal bands so we could store the nonzero entries in three 1-d arrays or an $5 \times 3$ matrix. (Actually because it is symmetric, i.e., $A_{ij} = A_{ji}$ we only need 2 vectors for storage.)

- For a general sparse matrix we want to store only the nonzero entries and
we typically do this in a one-dimensional array. For example, we could have

\[(4, -1, -1, 2, -1, -1, 8, -1, -1, 4, -2, -2, 4)\]

- If, for example, we want to multiply this sparse matrix times a vector then we need to know in which row and column a particular element of the one-d array actually lives.

- So if we have e.g., the 5th entry in our stored matrix (i.e., the 5th entry in the 1-d array), how do we know where it lives in the full matrix?

- We could have two integer 1-d arrays of length \texttt{n\_nonzero} (the number of nonzero entries) and the 5th position in one could point to the row and the 5th position in the other could point to the column of the original matrix.

- However, we can actually recover the information with less storage and this is typically what is done in standard library software. First assume we have a 1-d array of length \texttt{n\_nonzero} that points to the column each entry is in; e.g.,

\[(1, 2, 1, 2, 3, 2, 3, 4, 3, 4, 5, 4, 5)\]

- Now we have another 1-d integer array dimensioned by the number of rows
in the original matrix, say $n$. This array will tell us where each row begins; e.g.,
\[(1, 3, 6, 9, 12)\]

• Now for our matrix
\[
\begin{pmatrix}
4 & -1 & 0 & 0 & 0 \\
-1 & 2 & -1 & 0 & 0 \\
0 & -1 & 8 & -1 & 0 \\
0 & 0 & -1 & 4 & -2 \\
0 & 0 & 0 & -2 & 4
\end{pmatrix}
\]
we store \((4, -1, -1, 2, -1, -1, 8, -1, -1, 4, -2, -2, 4)\) and have the pointer arrays \((1, 2, 1, 2, 3, 2, 3, 4, 3, 4, 5, 4, 5)\) and \((1, 3, 6, 9, 12)\). If we have, e.g., the 4th entry in our 1-d array then we know that it goes in column 2 of the original matrix. Then we have to look at the row array and determine that the 2nd row begins with the third element in stored array and the 3rd row begins at the 6th element so our 4th entry must be in the second row.

• Conversely, if we want to know where the \((4, 5)\) entry (i.e., -2) of our original matrix is stored then from the row pointer array we know it is in entries 9,
10 or 11 (because row 4 starts at entry 9 and row 5 at entry 12). So we need to check entries 9, 10, 11 in column pointer array and see which is 5; in our case it is 11 so -2 is stored in the 11th entry in the 1-d array.

• So to define a sparse matrix derived type we clearly need (i) the number of nonzero entries in the vector, (ii) the stored vector, (iii) the dimension of the matrix (assuming it is $n \times n$), (iv) the integer array indicating which column each stored entry appears in the full matrix and (v) the row pointer array which indicates the entry in the stored 1-d array where each row begins. With this information we should be able to recover the full matrix and perform operations with it such as a matrix times a full vector.
• We define the sparse matrix data type (assuming the matrix is $n \times n$) by

```fortran
  type sparse_matrix
    integer :: n_nonzero
    real(prec), pointer, dimension(:) :: nonzero_entries
    integer :: n
    integer, pointer dimension(:) :: column_pointer
    integer, pointer dimension(:) :: row_pointer
  end type sparse_matrix
```
A routine to multiply a sparse matrix times a full vector

For simplicity I have left off some of the declaration statements for internal variables

function sparse_matrix_times_vector ( a, x ) result ( b )

type(sparse_matrix) :: a

type (vector ) :: b, x

doi = 1, a % n
    row_begin = a % row_pointer(i)
    row_end = a % row_pointer(i+1) - 1
    doj = row_begin, row_end
\[ k = a \% \text{column}_{\text{pointer}}(j) \]
\[ b \% \text{oned}_{\text{array}}(i) = b \% \text{oned}_{\text{array}}(i) \& + a \% \text{nonzero}_{\text{entries}}(j) * x \% \text{oned}_{\text{array}}(k) \]

end do

end do
Homework

• Project III - writing a vector class and a sparse vector class