Out[ ]:
```
5.5677643628300215
```

Triangle inequality. Let's check that the triangle inequality $\|A + B\| \leq \|A\| + \|B\|$ holds, for two specific matrices.

In [ ]:
```
A = np.array([[-1,0], [2,2]])
B = np.array([[3,1], [-3,2]])
print(np.linalg.norm(A + B))
print(np.linalg.norm(A) + np.linalg.norm(B))
```

```
4.69041575982343
7.795831523312719
```

Alternatively, we can write our own code to find the norm of `A`:

In [ ]:
```
A = np.array([[1,2,3], [4,5,6], [7,8,9], [10,11,12]])
m,n = A.shape
print(A.shape)
sum_of_sq = 0
for i in range(m):
    for j in range(n):
        sum_of_sq = sum_of_sq + A[i,j]**2
matrix_norm = np.sqrt(sum_of_sq)
print(matrix_norm)
print(np.linalg.norm(A))
```

```
(4, 3)
25.495097567963924
25.495097567963924
```

6.4. Matrix-vector multiplication

In Python, matrix-vector multiplication has the natural syntax `y = A @ x`. Alternatively, we can use the numpy function `np.matmul(A,x)`.

In [ ]:
```
A = np.array([[0,2,-1],[-2,1,1]])
x = np.array([2,1,-1])
A @ x
```

Out[ ]:
```
array([ 3, -4])
```

```
In [ ]: np.matmul(A,x)
```

```
Out[ ]: array([ 3, -4])
```

Difference matrix. An $(n-1) \times n$ difference matrix (equation (6.5) of VMLS) can be constructed in several ways. A simple one is the following.

```
In [ ]: diff_mat = lambda n: np.c_[-np.identity(n-1), np.zeros(n-1)] +
        ↪ np.c_[np.zeros(n-1), np.identity(n-1)]
        D = diff_mat(4)
        x = np.array([-1,0,2,1])
        D @ x
```

```
Out[ ]: array([ 1.,  2., -1.])
```

Since a difference matrix contains many zeros, this is a good opportunity to use sparse matrices. Here we use the `sparse.hstack()` function to stack the sparse matrices.

```
In [ ]: diff_mat = lambda n: sparse.hstack([-sparse.eye(n-1),
        ↪ sparse.coo_matrix((n-1,1))])+
        ↪ sparse.hstack([sparse.coo_matrix((n-1,1)), sparse.eye(n-1)])
        D = diff_mat(4)
        D @ np.array([-1,0,2,1])
```

```
Out[ ]: array([ 1.,  2., -1.])
```

Running sum matrix. The running sum matrix (equation (6.6) in VMLS) is a lower triangular matrix, with elements on and below the diagonal equal to one.

```
In [ ]: def running_sum(n):
            import numpy as np
            S = np.zeros((n,n))
            for i in range(n):
                for j in range(i+1):
                    S[i,j] = 1
            return S
        running_sum(4)
```

```
Out[ ]: array([[1., 0., 0., 0.],
               [1., 1., 0., 0.],
               [1., 1., 1., 0.],
               [1., 1., 1., 1.]])
```

```
In [ ]: running_sum(4) @ np.array([-1,1,2,0])
```

```
Out[ ]: array([-1.,  0.,  2.,  2.])
```

An alternative construction is `np.tril(np.ones((n,n)))`. This uses the function `np.tril`, which sets the elements of a matrix above the diagonal to zero.

Vandermonde matrix. An $m \times n$ Vandermonde matrix (equation (6.7) in VMLS) has entries t_i^{j-1} for $i = 1, \ldots, m$ and $j = 1, \ldots, n$. We define a function that takes an m-vector with elements $t_1, \ldots, t_m$ and returns the corresponding $m \times n$ Vandermonde matrix.

```
In [ ]: def vandermonde(t,n):
            m = len(t)
            V = np.zeros((m,n))
            for i in range(m):
                for j in range(n):
                    V[i,j] = t[i]**(j)
            return V
        vandermonde(np.array([-1,0,0.5,1]),5)
```

```
Out[ ]: array([[ 1.    , -1.    ,  1.    , -1.    ,  1.    ],
               [ 1.    ,  0.    ,  0.    ,  0.    ,  0.    ],
               [ 1.    ,  0.5   ,  0.25  ,  0.125 ,  0.0625],
               [ 1.    ,  1.    ,  1.    ,  1.    ,  1.    ]])
```

An alternative shorter definition uses numpy `np.column_stack` function.

```
In [ ]: vandermonde = lambda t,n: np.column_stack([t**i for i in
        ↪  range(n)])
        vandermonde(np.array([-1,0,0.5,1]),5)
```

```
Out[ ]: array([[ 1.    , -1.    ,  1.    , -1.    ,  1.    ],
               [ 1.    ,  0.    ,  0.    ,  0.    ,  0.    ],
               [ 1.    ,  0.5   ,  0.25  ,  0.125 ,  0.0625],
               [ 1.    ,  1.    ,  1.    ,  1.    ,  1.    ]])
```

6.5. Complexity

Complexity of matrix-vector multiplication. The complexity of multiplying an $m \times n$ matrix by an n-vector is $2mn$ flops. This grows linearly with both m and n. Let's check