NumPy/SciPy Notes

Aaron Bestick

September 15, 2013

1 What is NumPy/SciPy?

NumPy and SciPy are libraries which, together, provide MATLAB-like functionality in Python. NumPy provides a multidimensional array datatype and some basic linear algebra functionality. SciPy builds on this by providing more sophisticated linear algebra functions, as well as a variety of other functions for data processing and visualization.

NumPy/SciPy have a couple advantages over MATLAB for some programming tasks: First, they're free(!), and second, they're built on top of a *real* programming language (Python), which makes it easier to write complex software without being confined to the set of tools available in MATLAB.

Since we're already using Python to write code for ROS, we'll use it in the homeworks as well so you'll be able to recycle homework code for labs and vice-versa.

1.1 Getting NumPy/SciPy

The easiest ways to access NumPy/SciPy for this class are to use either the lab computers or the Ubuntu+ROS virtual machine available on the course web page. If you'd like to install them on your own computer, instructions are available at `http://scipy.org/install.html`.

2 Usage

2.1 Creating arrays

First, import the NumPy and SciPy libraries:

```
import numpy as np
import scipy as sp
```

Then create new arrays using the `numpy.array()` function:

```
#1-D array
newArray1 = np.array([1,2,3,4])

#2-D array
newArray2 = np.array([[1,2,3],[4,5,6],[7,8,9]])

#3-D array
newArray3 = np.array([[[1,2],[3,4]],[[5,6],[7,8]]])
```

We can print arrays too:

```
print(newArray1)
print(newArray2)
print(newArray3)

#Output:
#[1 2 3 4]
```

```
#
#[[1 2 3]
# [4 5 6]
# [7 8 9]]
#
#[[[1 2]
#  [3 4]]
#
# [[5 6]
#  [7 8]]]
```

Other functions create frequently used arrays quickly:

```
identityArray = np.eye((3,3))
onesArray = np.ones((2,3))
zerosArray = np.zeros((3,2))

print(identityArray)
print(onesArray)
print(zerosArray)

#Output:
#[[ 1.  0.  0.]
# [ 0.  1.  0.]
# [ 0.  0.  1.]]
#
#[[ 1.  1.  1.]
# [ 1.  1.  1.]]
#
#[[ 0.  0.]
# [ 0.  0.]
# [ 0.  0.]]
```

2.2 Array properties

Once you've created an array, it has a number of properties you can access. These include

- Total number of elements:

  ```
  print(newArray1.size)
  print(newArray2.size)
  print(newArray3.size)

  #Output:
  #4
  #9
  #8
  ```

- Number of dimensions:

  ```
  print(newArray1.ndim)
  print(newArray2.ndim)
  print(newArray3.ndim)

  #Output:
  #1
  #2
  #3
  ```

- Size:

```
print(newArray1.shape)
print(newArray2.shape)
print(newArray3.shape)

#Output:
#(4,)
#(3,3)
#(2,2,2)
```

2.3 Array indexing

You can access individual elements of arrays using syntax similar to that in MATLAB (note that Python uses zero-based indexing as opposed to MATLAB's one-based indexing):

```
print(newArray2[1,2])

#Output:
#6
```

You can also take "slices" from arrays using MATLAB-like syntax:

```
print(newArray2[0:2,1:2])

#Output:
#[[2]
# [5]]
```

The `.flat` attribute allows an array to be indexed as one long vector (like linear indexing in MATLAB):

```
print(newArray2.flat[7])

#Output:
#8
```

The syntax for setting array values is the same as for reading them:

```
newArray2[1,2] = 26
print(newArray2)
newArray2[0:2,0:1] = [[15],[25]]
print(newArray2)

#Output:
#[[ 1  2  3]
# [ 4  5 26]
# [ 7  8  9]]
#
#[[15  2  3]
# [25  5 26]
# [ 7  8  9]]
```

2.4 SciPy linear algebra functions

Once you've created an array, SciPy provides a library of common linear algebra functions you can use with it. First, import the linear algebra library:

```
from scipy import linalg
```

Some examples of useful functions are:

- Matrix multiplication:

  ```
  mult = linalg.dot(newArray2,newArray2)
  #or, equivalently
  mult = newArray2.dot(newArray2)
  ```

- Inverse:

  ```
  inv = linalg.inverse(newArray2)
  ```

- Norm:

  ```
  norm = linalg.norm(newArray1)
  #Calculates the 2-norm by default
  ```

- Matrix exponential:

  ```
  exp = linalg.expm(newArray2)
  ```

2.5 Tips

A few additional notes...

- Don't forget that Python differentiates between integer and floating point operations:

  ```
  print(3/4)
  print(3.0/4)

  #Output:
  #0
  #0.75
  ```

- Your code will be easier to debug if you add some basic error checking to functions which use arrays:

  ```
  def rot3D(omega, theta):
      #Check that omega is a 3-vector and throw an exception if it isn't
      if not omega.shape == (3,):
          raise TypeError("omega must be a 3-vector")

      #Compute the rotation matrix...

      #Return the result
      return rotationMatrix
  ```

- NumPy offers a `matrix` type in addition to the `ndarray` type we've used here. To avoid confusion, we'll always use `ndarray`.