NumPy

NumPy [http://www.numpy.org/] is the core Python package for numerical computing. The main features of NumPy are:

- N-dimensional array object `ndarray`
- Vectorized operations and functions which broadcast across arrays for fast computation

To get started with NumPy, let's adopt the standard convention and import it using the name `np` :

```
import numpy as np
```

NumPy Arrays

The fundamental object provided by the NumPy package is the `ndarray` . We can think of a 1D (1-dimensional) `ndarray` as a list, a 2D (2-dimensional) `ndarray` as a matrix, a 3D (3-dimensional) `ndarray` as a 3-tensor (or a "cube" of numbers), and so on. See the NumPy tutorial [https://docs.scipy.org/doc/numpy/user/quickstart.html] for more about NumPy arrays.

Creating Arrays

The function `numpy.array` creates a NumPy array from a Python sequence such as a list, a tuple or a list of lists. For example, create a 1D NumPy array from a Python list:

```
a = np.array([1,2,3,4,5])
print(a)
```

```
[1 2 3 4 5]
```

Notice that when we print a NumPy array it looks a lot like a Python list except that the entries are separated by spaces whereas entries in a Python list are separated by commas:

```
print([1,2,3,4,5])
```

```
[1, 2, 3, 4, 5]
```

Notice also that a NumPy array is displayed slightly differently when output by a cell (as opposed to being explicitly printed to output by the `print` function):

```
a
```

```
array([1, 2, 3, 4, 5])
```

Use the built-in function `type` to verify the type:

```
type(a)
```

```
numpy.ndarray
```

Create a 2D NumPy array from a Python list of lists:

```
M = np.array([[1,2,3],[4,5,6]])
print(M)
```

```
[[1 2 3]
 [4 5 6]]
```

```
type(M)
```

```
numpy.ndarray
```

Create an n-dimensional NumPy array from nested Python lists. For example, the following is a 3D NumPy array:

```
N = np.array([ [[1,2],[3,4]] , [[5,6],[7,8]] , [[9,10],[11,12]] ])
print(N)
```

```
[[[ 1  2]
  [ 3  4]]

 [[ 5  6]
  [ 7  8]]

 [[ 9 10]
  [11 12]]]
```

There are several NumPy functions for creating arrays [https://docs.scipy.org/doc/numpy/user/quickstart.html#array-creation]:

Function	Description
`numpy.array(a)`	Create n-dimensional NumPy array from sequence `a`
`numpy.linspace(a,b,N)`	Create 1D NumPy array with `N` equally spaced values from `a` to `b` (inclusively)
`numpy.arange(a,b,step)`	Create 1D NumPy array with values from `a` to `b` (exclusively) incremented by `step`
`numpy.zeros(N)`	Create 1D NumPy array of zeros of length N
`numpy.zeros((n,m))`	Create 2D NumPy array of zeros with n rows and m columns
`numpy.ones(N)`	Create 1D NumPy array of ones of length N
`numpy.ones((n,m))`	Create 2D NumPy array of ones with n rows and m columns
`numpy.eye(N)`	Create 2D NumPy array with N rows and N columns with ones on the diagonal (ie. the identity matrix of size N)

Create a 1D NumPy array with 11 equally spaced values from 0 to 1:

```
x = np.linspace(0,1,11)
print(x)
```

```
[ 0.   0.1  0.2  0.3  0.4  0.5  0.6  0.7  0.8  0.9  1. ]
```

Create a 1D NumPy array with values from 0 to 20 (exclusively) incremented by 2.5:

```
y = np.arange(0,20,2.5)
print(y)
```

```
[  0.    2.5   5.    7.5  10.   12.5  15.   17.5]
```

These are the functions that we'll use most often when creating NumPy arrays. The function `numpy.linspace` works best when we know the *number of points* we want in the array, and `numpy.arange` works best when we know *step size* between values in the array.

Create a 1D NumPy array of zeros of length 5:

```
z = np.zeros(5)
print(z)
```

```
[ 0.  0.  0.  0.  0.]
```

Create a 2D NumPy array of zeros with 2 rows and 5 columns:

```
M = np.zeros((2,5))
print(M)
```

```
[[ 0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.]]
```

Create a 1D NumPy array of ones of length 7:

```
w = np.ones(7)
print(w)
```

```
[ 1.  1.  1.  1.  1.  1.  1.]
```

Create a 2D NumPy array of ones with 3 rows and 2 columns:

```
N = np.ones((3,2))
print(N)
```

```
[[ 1.  1.]
 [ 1.  1.]
 [ 1.  1.]]
```

Create the identity matrix of size 10:

```
I = np.eye(10)
print(I)
```

```
[[ 1.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  1.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  1.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  1.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  1.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  1.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  1.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  1.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  1.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  1.]]
```

Array Datatype

NumPy arrays are *homogeneous*: all entries in the array are the same datatype. We will only work with numeric arrays and our arrays will contain either integers, floats, complex numbers or booleans. There are different kinds of datatypes provided by NumPy for different applications but we'll mostly be working with the default integer type `numpy.int64` and the default float type `numpy.float64`. These are very similar to the built-in Python datatypes `int` and `float` but with some differences that we won't go into. Check out the NumPy documentation on numeric datatypes [https://docs.scipy.org/doc/numpy/user/basics.types.html] for more information.

The most important point for now is to know how to determine if a NumPy array contains integers elements or float elements. We can access the datatype of a NumPy array by its `.dtype` attribute. For example, create a 2D NumPy array from a list of lists of integers:

```
A = np.array([[1,2,3],[4,5,6]])
print(A)
```

```
[[1 2 3]
 [4 5 6]]
```

We expect the datatype of `A` to be integers and we verify:

```
A.dtype
```

```
dtype('int64')
```

Most of the other NumPy functions which create arrays use the `numpy.float64` datatype by default. For example, using `numpy.linspace`:

```
u = np.linspace(0,1,5)
print(u)
```

```
[ 0.    0.25  0.5   0.75  1.  ]
```

```
u.dtype
```

```
dtype('float64')
```

Notice that numbers are printed with a decimal point when the datatype of the NumPy array is any kind of float.

Dimension, Shape and Size

We can think of a 1D NumPy array as a list of numbers, a 2D NumPy array as a matrix, a 3D NumPy array as a cube of numbers, and so on. Given a NumPy array, we can find out how many dimensions it has by accessing its `.ndim` attribute. The result is a number telling us how many dimensions it has.

For example, create a 2D NumPy array:

```
A = np.array([[1,2],[3,4],[5,6]])
print(A)
```

```
[[1 2]
 [3 4]
 [5 6]]
```

```
A.ndim
```

```
2
```

The result tells us that `A` has 2 dimensions. The first dimension corresponds to the vertical direction counting the rows and the second dimension corresponds to the horizontal direction counting the columns.

We can find out how many rows and columns `A` has by accessing its `.shape` attribute:

```
A.shape
```

```
(3, 2)
```

The result is a tuple `(3,2)` of length 2 which means that `A` is a 2D array with 3 rows and 2 columns.

We can also find out how many entries `A` has in total by accessing its `.size` attribute:

```
A.size
```

```
6
```

This is the expected result since we know that `A` has 3 rows and 2 columns and therefore 2(3) = 6 total entries.

Create a 1D NumPy array and inspect its dimension, shape and size:

```
r = np.array([9,3,1,7])
print(r)
```

```
[9 3 1 7]
```

```
r.ndim
```

```
1
```

```
r.shape
```

```
(4,)
```

```
r.size
```

```
4
```

The variable `r` is assigned to a 1D NumPy array of length 4. Notice that `r.shape` is a tuple with a single entry `(4,)`.

Slicing and Indexing

Accessing the entries in an array is called *indexing* and accessing rows and columns (or subarrays) is called *slicing*. See the NumPy documentation for more information about indexing and slicing [https://docs.scipy.org/doc/numpy/reference/arrays.indexing.html].

Create a 1D NumPy array:

```
v = np.linspace(0,5,11)
print(v)
```

```
[ 0.   0.5  1.   1.5  2.   2.5  3.   3.5  4.   4.5  5. ]
```

Access the entries in a 1D array using the square brackets notation just like a Python list. For example, access the entry at index 3:

```
v[3]
```

```
1.5
```

Notice that NumPy array indices start at 0 just like Python sequences.

Create a 2D array of integers:

```
B = np.array([[6, 5, 3, 1, 1],[1, 0, 4, 0, 1],[5, 9, 2, 2, 9]])
print(B)
```

```
[[6 5 3 1 1]
 [1 0 4 0 1]
 [5 9 2 2 9]]
```

Access the entries in a 2D array using the square brackets with 2 indices. In particular, access the entry at row index 1 and column index 2:

```
B[1,2]
```

```
4
```

Access the top left entry in the array:

```
B[0,0]
```

```
6
```

Negative indices work for NumPy arrays as they do for Python sequences. Access the bottom right entry in the array:

```
B[-1,-1]
```

```
9
```

Access a the row at index 2 using the colon `:` syntax:

```
B[2,:]
```

```
array([5, 9, 2, 2, 9])
```

Access the column at index 3:

```
B[:,3]
```

```
array([1, 0, 2])
```

Select the subarray of rows at index 1 and 2, and columns at index 2, 3 and 4:

```
subB = B[1:3,2:5]
print(subB)
```

```
[[4 0 1]
 [2 2 9]]
```

Slices of NumPy arrays are again NumPy arrays but possibly of a different dimension:

```
subB.ndim
```

```
2
```

```
subB.shape
```

```
(2, 3)
```

```
type(subB)
```

```
numpy.ndarray
```

The variable `subB` is assigned to a 2D NumPy array of shape 2 by 2.

Let's do the same for the column at index 2:

```
colB = B[:,2]
print(colB)
```

```
[3 4 2]
```

```
colB.ndim
```

```
1
```

```
colB.shape
```

```
(3,)
```

```
type(colB)
```

```
numpy.ndarray
```

The variable `colB` is assigned to a 1D NumPy array of length 3.

Stacking

We can build bigger arrays out of smaller arrays by stacking [https://docs.scipy.org/doc/numpy/user/quickstart.html#stacking-together-different-arrays] along different dimensions using the functions `numpy.hstack` and `numpy.vstack`.

Stack 3 different 1D NumPy arrays of length 3 vertically forming a 3 by 3 matrix:

```
x = np.array([1,1,1])
y = np.array([2,2,2])
z = np.array([3,3,3])
vstacked = np.vstack((x,y,z))
print(vstacked)
```

```
[[1 1 1]
 [2 2 2]
 [3 3 3]]
```

Stack 1D NumPy arrays horizontally to create another 1D array:

```
hstacked = np.hstack((x,y,z))
print(hstacked)
```

```
[1 1 1 2 2 2 3 3 3]
```

Use `numpy.hstack` and `numpy.vstack` to build the block matrix

$$T = \begin{bmatrix} 1 & 1 & 2 & 2 \\ 1 & 1 & 2 & 2 \\ 3 & 3 & 4 & 4 \\ 3 & 3 & 4 & 4 \end{bmatrix}$$

```
A = np.ones((2,2))
B = 2*np.ones((2,2))
C = 3*np.ones((2,2))
D = 4*np.ones((2,2))
A_B = np.hstack((A,B))
print(A_B)
```

```
[[ 1.  1.  2.  2.]
 [ 1.  1.  2.  2.]]
```

```
C_D = np.hstack((C,D))
print(C_D)
```

```
[[ 3.  3.  4.  4.]
 [ 3.  3.  4.  4.]]
```

```
T = np.vstack((A_B,C_D))
print(T)
```

```
[[ 1.  1.  2.  2.]
 [ 1.  1.  2.  2.]
 [ 3.  3.  4.  4.]
 [ 3.  3.  4.  4.]]
```

Copies versus Views

Under construction

Operations and Functions

Array Operations

Arithmetic operators [https://docs.scipy.org/doc/numpy/user/quickstart.html#basic-operations] including addition `+` , subtraction `-` , multiplication `*` , division `/` and exponentiation `**` are applied to arrays *elementwise*. For addition and substraction, these are the familiar vector operations we see in linear algebra:

```
v = np.array([1,2,3])
w = np.array([1,0,-1])
```

```
v + w
```

```
array([2, 2, 2])
```

```
v - w
```

```
array([0, 2, 4])
```

In the same way, array multiplication and division are performed element by element:

```
v * w
```

```
array([ 1,  0, -3])
```

```
w / v
```

```
array([ 1.        ,  0.        , -0.33333333])
```

Notice that the datatype of both `v` and `w` is `numpy.int64` however division `w / v` returns an array with datatype `numpy.float64` .

The exponent operator `**` also acts element by element in the array:

```
v ** 2
```

```
array([1, 4, 9])
```

Let's see these operations for 2D arrays:

```
A = np.array([[3,1],[2,-1]])
B = np.array([[2,-2],[5,1]])
```

```
A + B
```

```
array([[ 5, -1],
       [ 7,  0]])
```

```
A - B
```

```
array([[ 1,  3],
       [-3, -2]])
```

```
A / B
```

```
array([[ 1.5, -0.5],
       [ 0.4, -1. ]])
```

```
A * B
```

```
array([[ 6, -2],
       [10, -1]])
```

```
A ** 2
```

```
array([[9, 1],
       [4, 1]])
```

Notice that array multiplication and exponentiation are performed elementwise.

In Python 3.5+, the symbol `@` computes matrix multiplication for NumPy arrays:

```
A @ B
```

```
array([[11, -5],
       [-1, -5]])
```

Matrix powers [https://docs.scipy.org/doc/numpy/reference/generated/numpy.linalg.matrix_power.html] are performed by the function `numpy.linalg.matrix_power`. It's a long function name and so it's convenient to import it with a shorter name:

```
from numpy.linalg import matrix_power as mpow
```

Compute A^3:

```
mpow(A,3)
```

```
array([[37,  9],
       [18,  1]])
```

Equivalently, use the `@` operator to compute A^3:

```
A @ A @ A
```

```
array([[37,  9],
       [18,  1]])
```

Broadcasting

We know from linear algebra that we can only add matrices of the same size. Braodcasting [https://docs.scipy.org/doc/numpy/user/basics.broadcasting.html] is a set of NumPy rules which relaxes this constraint and allows us to combine a smaller array with a bigger when it makes sense.

For example, suppose we want to create a 1D NumPy array of y values for $x = 0.0, 0.25, 0.5, 0.75, 1.0$ for the function $y = x^2 + 1$. From what we've seen so far, it makes sense to create `x`, then `x**2` and then add an array of ones `[1. 1. 1. 1. 1.]`:

```
x = np.array([0,0.25,0.5,0.75,1.0])
y = x**2 + np.array([1,1,1,1,1])
print(y)
```

```
[ 1.      1.0625  1.25    1.5625  2.    ]
```

An example of broadcasting in NumPy is the following equivalent operation:

```
x = np.array([0,0.25,0.5,0.75,1.0])
y = x**2 + 1
print(y)
```

```
[ 1.      1.0625  1.25    1.5625  2.    ]
```

The number 1 is a scalar and we are adding it to a 1D NumPy array of length 5. The braodcasting rule in this case is to broadcast the scalar value 1 across the larger array. The result is a simpler syntax for a very comman operation.

Let's try another example. What happens when we try to add a 1D NumPy array of length 4 to a 2D NumPy array of size 3 by 4?

```
u = np.array([1,2,3,4])
print(u)
```

```
[1 2 3 4]
```

```
A = np.array([[1,1,1,1],[2,2,2,2],[3,3,3,3]])
print(A)
```

```
[[1 1 1 1]
 [2 2 2 2]
 [3 3 3 3]]
```

```
result = A + u
print(result)
```

```
[[2 3 4 5]
 [3 4 5 6]
 [4 5 6 7]]
```

The 1D NumPy array is broadcast across the 2D array because the length of the first dimension in each array are equal!

Array Functions

There are *many* array functions [https://docs.scipy.org/doc/numpy/reference/routines.html] we can use to compute with NumPy arrays. The following is a partial list and we'll look closer at mathematical functions in the next section.

`numpy.sum`	`numpy.prod`	`numpy.mean`
`numpy.max`	`numpy.min`	`numpy.std`
`numpy.argmax`	`numpy.argmin`	`numpy.var`

Create a 1D NumPy array with random values and compute:

```
arr = np.array([8,-2,4,7,-3])
print(arr)
```

```
[ 8 -2  4  7 -3]
```

Compute the mean of the values in the array:

```
np.mean(arr)
```

```
2.7999999999999998
```

Verify the mean once more:

```
m = np.sum(arr) / arr.size
print(m)
```

```
2.8
```

Interestingly, the function `numpy.mean` introduced some rounding error.

Find the index of the maximum element in the array:

```
max_i = np.argmax(arr)
print(max_i)
```

```
0
```

Verify the maximum value in the array:

```
np.max(arr)
```

```
8
```

```
arr[max_i]
```

```
8
```

Array functions apply to 2D arrays as well (and N-dimensional arrays in general) with the added feature that we can choose to apply array functions to the entire array, down the columns or across the rows (or any axis).

Create a 2D NumPy array with random values and compute the sum of all the entries:

```
M = np.array([[2,4,2],[2,1,1],[3,2,0],[0,6,2]])
print(M)
```

```
[[2 4 2]
 [2 1 1]
 [3 2 0]
 [0 6 2]]
```

```
np.sum(M)
```

```
25
```

The function `numpy.sum` also takes a keyword argument `axis` which determines along which dimension to compute the sum:

```
np.sum(M,axis=0) # Sum of the columns
```

```
array([ 7, 13,  5])
```

```
np.sum(M,axis=1) # Sum of the rows
```

```
array([8, 4, 5, 8])
```

Mathematical Functions

Mathematical functions [http://docs.scipy.org/doc/numpy/reference/routines.math.html] in NumPy are called universal functions [https://docs.scipy.org/doc/numpy/user/quickstart.html#universal-functions]) and are *vectorized*. Vectorized functions operate *elementwise* on arrays producing arrays as output and are built to compute values across arrays *very* quickly. The following is a partial list of mathematical functions:

`numpy.sin`	`numpy.cos`	`numpy.tan`
`numpy.exp`	`numpy.log`	`numpy.log10`
`numpy.arcsin`	`numpy.arccos`	`numpy.arctan`

Compute the values $\sin(2\pi x)$ for $x = 0, 0.25, 0.5 \dots, 1.75$:

```
x = np.arange(0,1.25,0.25)
print(x)
```

```
[ 0.    0.25  0.5   0.75  1.  ]
```

```
np.sin(2*np.pi*x)
```

```
array([  0.00000000e+00,   1.00000000e+00,   1.22464680e-16,
        -1.00000000e+00,  -2.44929360e-16])
```

We expect the array `[0. 1. 0. -1. 0.]` however there is (as always with floating point numbers) some rounding errors in the result. In numerical computing, we can interpret a number such as 10^{-16} as 0.

Compute the values $\log_{10}(x)$ for $x = 1, 10, 100, 1000, 10000$:

```
x = np.array([1,10,100,1000,10000])
print(x)
```

```
[    1    10   100  1000 10000]
```

```
np.log10(x)
```

```
array([ 0.,  1.,  2.,  3.,  4.])
```

Note that we can also evaluate mathematical functions with scalar values:

```
np.sin(0)
```

```
0.0
```

NumPy also provides familiar mathematical constants such as π and e:

```
np.pi
```

```
3.141592653589793
```

```
np.e
```

```
2.718281828459045
```

For example, verify the limit

$$\lim_{x \to \infty} \arctan(x) = \frac{\pi}{2}$$

by evaluating $\arctan(x)$ for some (arbitrary) large value x:

```
np.arctan(10000)
```

```
1.5706963267952299
```

```
np.pi/2
```

```
1.5707963267948966
```

Random Number Generators

The subpackage `numpy.random` contains functions to generate NumPy arrays of random numbers [https://docs.scipy.org/doc/numpy/reference/routines.random.html] sampled from different distributions. The following is a partial list of distributions:

Function	Description
`numpy.random.rand(d1,...,dn)`	Create a NumPy array (with shape `(d1,...,dn)`) with entries sampled uniformly from `[0,1)`
`numpy.random.randn(d1,...,dn)`	Create a NumPy array (with shape `(d1,...,dn)`) with entries sampled from the standard normal distribution
`numpy.random.randint(a,b,size)`	Create a NumPy array (with shape `size`) with integer entries from `low` (inclusive) to `high` (exclusive)

Sample a random number from the uniform distribution [https://en.wikipedia.org/wiki/Uniform_distribution_%28continuous%29]:

```
np.random.rand()
```

```
0.07229262326288399
```

Sample 3 random numbers:

```
np.random.rand(3)
```

```
array([ 0.4077374 ,  0.35378315,  0.10577014])
```

Create 2D NumPy array of random samples:

```
np.random.rand(2,4)
```

```
array([[ 0.1571874 ,  0.69865762,  0.3492962 ,  0.51449223],
       [ 0.55514344,  0.42532896,  0.71434093,  0.15352856]])
```

Random samples from the standard normal distribution [https://en.wikipedia.org/wiki/Normal_distribution]:

```
np.random.randn()
```

```
0.1031075276853907
```

```
np.random.randn(3)
```

```
array([-0.70740941, -3.21927869, -0.79699647])
```

```
np.random.randn(3,1)
```

```
array([[-2.01060258],
       [-0.21597249],
       [ 0.07654851]])
```

Random integers sampled uniformly from various intervals:

```
np.random.randint(-10,10)
```

```
-5
```

```
np.random.randint(0,2,(4,8))
```

```
array([[0, 0, 0, 1, 1, 1, 1, 1],
       [0, 0, 1, 1, 1, 0, 1, 0],
       [0, 0, 0, 1, 1, 0, 1, 0],
       [0, 0, 1, 1, 1, 0, 0, 1]])
```

```
np.random.randint(-9,10,(5,2))
```

```
array([[ 1, -2],
       [ 2,  9],
       [-5,  6],
       [ 5,  7],
       [-3, -7]])
```

Examples

Brute Force Optimization

Find the absolute maximum and minimum [https://en.wikipedia.org/wiki/Maxima_and_minima] values of the function

$$f(x) = x\sin(x) + \cos(4x)$$

on the interval $[0, 2\pi]$. We know that the maximum and minimum values must occur at either the endpoints $x = 0, 2\pi$ or at critical points where $f'(x) = 0$. However, the derivative is given by

$$f'(x) = \sin(x) + x\cos(x) - 4\sin(4x)$$

and the equation $f'(x) = 0$ is impossible to solve explicitly.

Instead, create a 1D NumPy array of x values from 0 to 2π of length N (for some arbitrarily large value N) and use the functions `numpy.min` and `numpy.max` to find maximum and minimum y values, and the functions `numpy.argmin` and `numpy.argmax` to find the indices of the corresponding x values.

```
N = 10000
x = np.linspace(0,2*np.pi,N)
y = x * np.sin(x) + np.cos(4*x)
y_max = np.max(y)
y_min = np.min(y)
x_max = x[np.argmax(y)]
x_min = x[np.argmin(y)]
print('Absolute maximum value is y =',y_max,'at x =',x_max)
print('Absolute minimum value is y =',y_min,'at x =',x_min)
```

```
Absolute maximum value is y = 2.59927260729 at x = 1.6281361267
Absolute minimum value is y = -5.12975203918 at x = 5.34187001664
```

Riemann Sums

Write a function called `exp_int` which takes input parameters b and N and returns the (left) Riemann sum [https://en.wikipedia.org/wiki/Riemann_sum]

$$\int_0^b e^{-x^2} \, dx \approx \sum_{k=0}^{N-1} e^{-x_k^2} \Delta x$$

for $\Delta x = b/N$ and the partition $x_k = k \, \Delta x$, $k = 0, \dots, N$.

```
def exp_int(b,N):
    "Compute left Riemann sum of exp(-x^2) from 0 to b with N
subintervals."
    x = np.linspace(0,b,N+1)
    x_left_endpoints = x[:-1]
    Delta_x = b/N
    I = Delta_x * np.sum(np.exp(-x_left_endpoints**2))
    return I
```

The infinite integral satisfies the beautiful identity [https://en.wikipedia.org/wiki/Gaussian_integral]

$$\int_0^{\infty} e^{-x^2} \, dx = \frac{\sqrt{\pi}}{2}$$

Compute the integral with large values of b and N:

```
exp_int(100,100000)
```

```
0.886726925452758
```

Compare to the true value:

```
np.pi**0.5/2
```

```
0.8862269254527579
```

Infinite Products

The cosine function has the following infinite product representation

$$\cos x = \prod_{k=1}^{\infty} \left(1 - \frac{4x^2}{\pi^2 (2k-1)^2} \right)$$

Write a function called `cos_product` which takes input parameters x and N and returns the Nth partial product

$$\prod_{k=1}^N \left(1 - \frac{4x^2}{\pi^2 (2k - 1)^2} \right)$$

```
def cos_product(x,N):
    "Compute the product \\prod_{k=1}^N (1 - 4x^2/(pi^2 (2k -
1)^2)."
    k = np.arange(1,N+1)
    terms = 1 - 4*x**2 / (np.pi**2 * (2*k - 1)**2)
    return np.prod(terms)
```

Verify our function using values for which we know the result. For example, $\cos(0) = 1$, $\cos(\pi) = -1$ and $\cos(\pi/4) = \frac{1}{\sqrt{2}}$.

```
cos_product(0,10)
```

```
1.0
```

```
cos_product(np.pi,10000)
```

```
-1.0001000050002433
```

```
cos_product(np.pi/4,10000000)
```

```
0.7071067856245614
```

```
1/2**0.5
```

```
0.7071067811865475
```

Matrix Multiplication

Under construction

Exercises

1. The natural log satisfies the following definite integral

$$\int_1^e \frac{\ln x \, dx}{(1 + \ln x)^2} = \frac{e}{2} - 1$$

   Write a function called `log_integral` which takes input parameters c and N and returns the value of the (right) Riemann sum

$$\int_1^c \frac{\ln x \, dx}{(1 + \ln x)^2} \approx \sum_{k=1}^N \frac{\ln x_k \, \Delta x}{(1 + \ln x_k)^2} \ , \ \ \Delta x = \frac{c-1}{N}$$

   for the partition $x_k = 1 + k\Delta x$, for $k = 0, \dots, N$.

2. Write a function called `k_sum` which takes input parameters `k` and `N` and returns the partial sum

$$\sum_{n=1}^N \frac{1}{n^k}$$

   Verify your function by comparing to the infinite series identity

$$\sum_{n=1}^\infty \frac{(-1)^{n+1}}{n^2} = \frac{\pi^2}{12}$$

3. Write a function called `dot` which takes 3 inputs `M`, `i` and `j` where `M` is a square NumPy array and the function returns the dot product of the ith row and the jth column of M.