**Image Super-Resolution Using Deep Convolutional Networks**

**ABSTRACT:**

We propose a Deep learning technique for single picture Super Resolution (SR). Our strategy straightforwardly learns a start to finish mapping between the low/high-resolution pictures. The mapping is spoken to as a Deep convolutional neural system (CNN) that takes the low-resolution picture as the information and yields the high-resolution one. We further show that customary inadequate coding-based SR techniques can likewise be seen as a Deep convolutional arrange. In any case, dissimilar to conventional strategies that handle every segment independently, our strategy together streamlines all layers. Our Deep CNN has a lightweight structure, yet exhibits best in class reclamation quality, what's more, accomplishes quick speed for viable on-line utilization. We investigate diverse system structures and parameter settings to accomplish tradeoffs among execution and speed. In addition, we stretch out our system to adapt to three color channels at the same time, and show better overall remaking quality.

**INTRODUCTION:**

Single picture super-resolution (SR), which focuses on recuperating a high-resolution picture from a single low-resolution picture is an old style issue in Computer vision. This issue is naturally not well presented since an assortment of arrangements exist for some random low-resolution pixel. At the end of the day, it is an underdetermined backwards issue, of which arrangement isn't extraordinary. Such an issue is commonly alleviated by obliging the arrangement space by solid earlier data.

**EXISTING SYSTEM:**

Image Super resolution has been implemented in several different waysusing Technologies like Matlab. The system show a acceptable accuracy for image super resolution by implementing Neural Networks in the system.

**DISADVANTAGES:**

The existing systems are implemented on Matlab and hence are not opensource.

It takes up more resources and overall gives less accuracy.

**PROPOSED SYSTEM:**

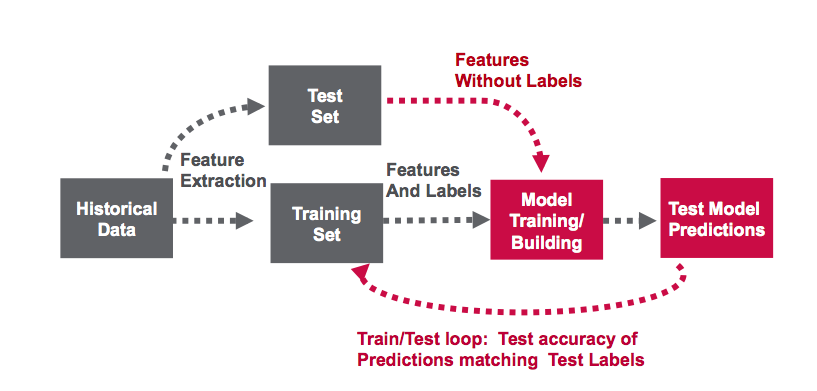
In our proposes system, Python based Deep learning algorithms are being implemented for making Super Resolution images using a Low-resolution image. The System is using Convolutional Neural Network for doing the task.

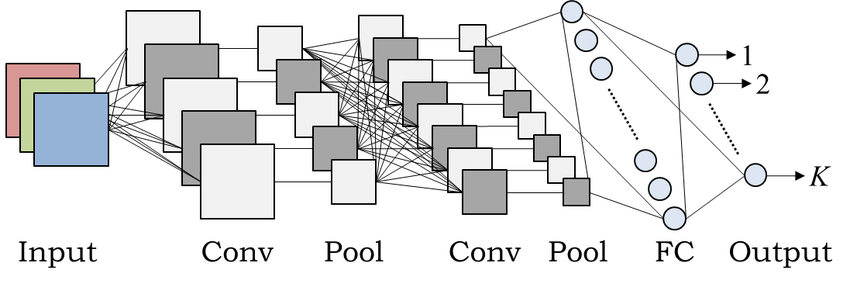
**ADVANTAGES:**

Usage of Machine Learning Algorithm makes the system more reliable and accurate.

Convolutional Neural Network algorithm is implemented.

**Algorithm: -**





Convolutional neural network architecture

Algorithm:

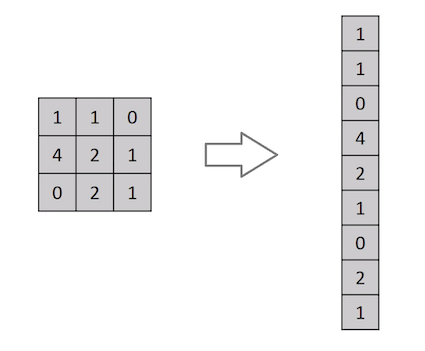


Example CNN sequence to classify handwritten digits

A **Convolutional Neural Network (ConvNet/CNN)** is a Deep Learning algorithm which can take in an input image, assign importance (learnable weights and biases) to various aspects/objects in the image and be able to differentiate one from the other. The pre-processing required in a ConvNet is much lower as compared to other classification algorithms. While in primitive methods filters are hand-engineered, with enough training, ConvNets have the ability to learn these filters/characteristics.

The architecture of a ConvNet is analogous to that of the connectivity pattern of Neurons in the Human Brain and was inspired by the organization of the Visual Cortex. Individual neurons respond to stimuli only in a restricted region of the visual field known as the Receptive Field. A collection of such fields overlaps to cover the entire visual area.

**Why ConvNets over Feed-Forward Neural Nets?**



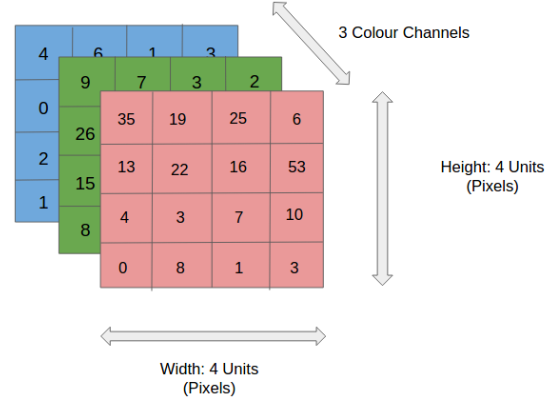
Flattening of a 3x3 image matrix into a 9x1 vector

An image is nothing but a matrix of pixel values, right? So why not just flatten the image (e.g. 3x3 image matrix into a 9x1 vector) and feed it to a Multi-Level Perceptron for classification purposes? Uh.. not really.

In cases of extremely basic binary images, the method might show an average precision score while performing prediction of classes but would have little to no accuracy when it comes to complex images having pixel dependencies throughout.

A ConvNet is able to **successfully capture the Spatial and Temporal dependencies** in an image through the application of relevant filters. The architecture performs a better fitting to the image dataset due to the reduction in the number of parameters involved and reusability of weights. In other words, the network can be trained to understand the sophistication of the image better.

**Input Image**



4x4x3 RGB Image

In the figure, we have an RGB image which has been separated by its three color planes — Red, Green, and Blue. There are a number of such color spaces in which images exist — Grayscale, RGB, HSV, CMYK, etc.

You can imagine how computationally intensive things would get once the images reach dimensions, say 8K (7680×4320). The role of the ConvNet is to reduce the images into a form which is easier to process, without losing features which are critical for getting a good prediction. This is important when we are to design an architecture which is not only good at learning features but also is scalable to massive datasets.

**Convolution Layer — The Kernel**



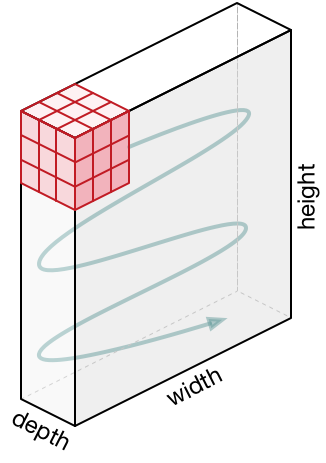
Convoluting a 5x5x1 image with a 3x3x1 kernel to get a 3x3x1 convolved feature

Image Dimensions = 5 (Height) x 5 (Breadth) x 1 (Number of channels, eg. RGB)

In the above demonstration, the green section resembles our **5x5x1 input image, I**. The element involved in carrying out the convolution operation in the first part of a Convolutional Layer is called the **Kernel/Filter, K**, represented in the color yellow. We have selected **K as a 3x3x1 matrix.**

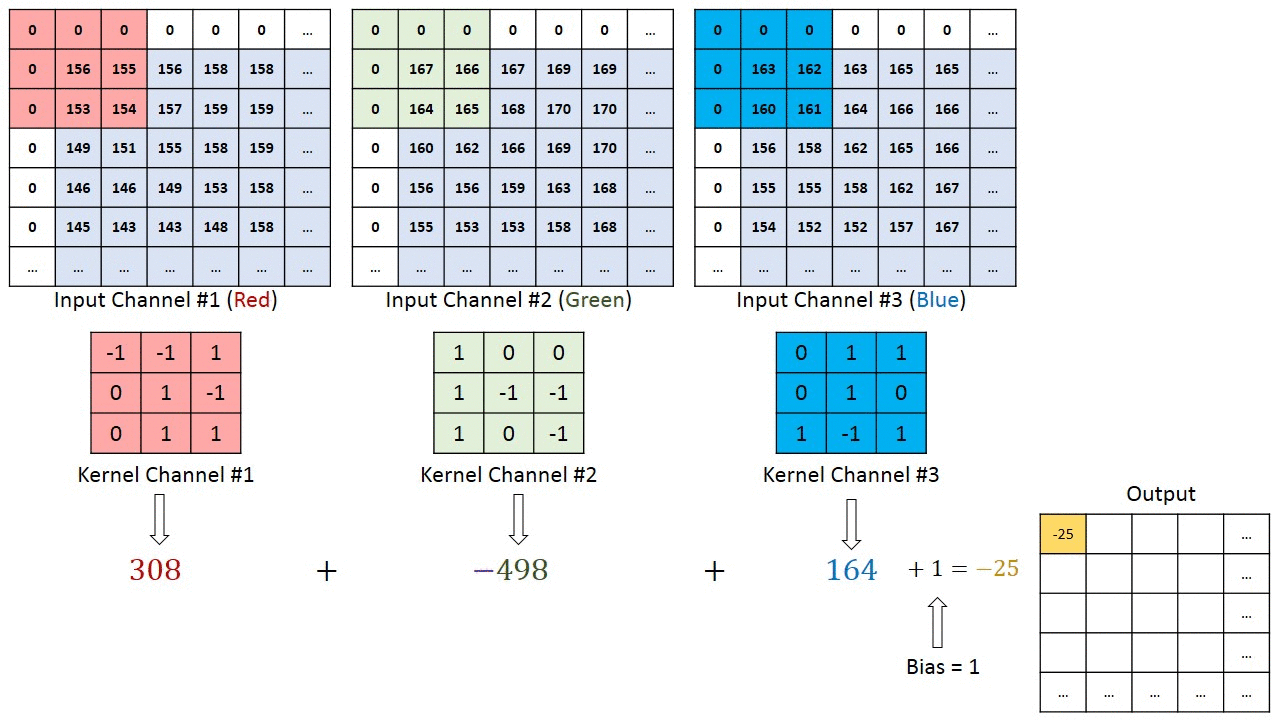
Kernel/Filter, K = 1 0 1  
0 1 0  
1 0 1

The Kernel shifts 9 times because of **Stride Length = 1 (Non-Strided)**, every time performing a **matrix multiplication operation between K and the portion P of the image** over which the kernel is hovering.



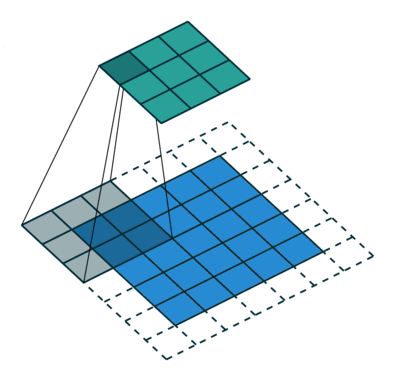
Movement of the Kernel

The filter moves to the right with a certain Stride Value till it parses the complete width. Moving on, it hops down to the beginning (left) of the image with the same Stride Value and repeats the process until the entire image is traversed.



Convolution operation on a MxNx3 image matrix with a 3x3x3 Kernel

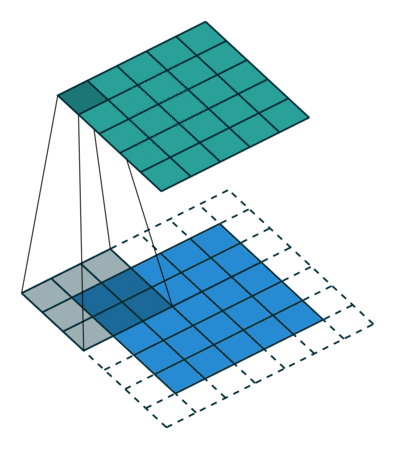
In the case of images with multiple channels (e.g. RGB), the Kernel has the same depth as that of the input image. Matrix Multiplication is performed between Kn and In stack ([K1, I1]; [K2, I2]; [K3, I3]) and all the results are summed with the bias to give us a squashed one-depth channel Convoluted Feature Output.



Convolution Operation with Stride Length = 2

The objective of the Convolution Operation is to **extract the high-level features** such as edges, from the input image. ConvNets need not be limited to only one Convolutional Layer. Conventionally, the first ConvLayer is responsible for capturing the Low-Level features such as edges, color, gradient orientation, etc. With added layers, the architecture adapts to the High-Level features as well, giving us a network which has the wholesome understanding of images in the dataset, similar to how we would.

There are two types of results to the operation — one in which the convolved feature is reduced in dimensionality as compared to the input, and the other in which the dimensionality is either increased or remains the same. This is done by applying **Valid Padding** in case of the former, or **Same Padding** in the case of the latter.



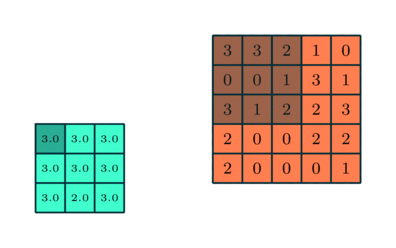
**SAME padding:** 5x5x1 image is padded with 0s to create a 6x6x1 image

When we augment the 5x5x1 image into a 6x6x1 image and then apply the 3x3x1 kernel over it, we find that the convolved matrix turns out to be of dimensions 5x5x1. Hence the name — **Same Padding**.

On the other hand, if we perform the same operation without padding, we are presented with a matrix which has dimensions of the Kernel (3x3x1) itself — **Valid Padding**.

The following repository houses many such GIFs which would help you get a better understanding of how Padding and Stride Length work together to achieve results relevant to our needs.

**Pooling Layer**



3x3 pooling over 5x5 convolved feature

Similar to the Convolutional Layer, the Pooling layer is responsible for reducing the spatial size of the Convolved Feature. This is to **decrease the computational power required to process the data** through dimensionality reduction. Furthermore, it is useful for **extracting dominant features** which are rotational and positional invariant, thus maintaining the process of effectively training of the model.

There are two types of Pooling: Max Pooling and Average Pooling. **Max Pooling** returns the **maximum value** from the portion of the image covered by the Kernel. On the other hand, **Average Pooling**returns the **average of all the values**from the portion of the image covered by the Kernel.

Max Pooling also performs as a**Noise Suppressant**. It discards the noisy activations altogether and also performs de-noising along with dimensionality reduction. On the other hand, Average Pooling simply performs dimensionality reduction as a noise suppressing mechanism. Hence, we can say that **Max Pooling performs a lot better than Average Pooling**.

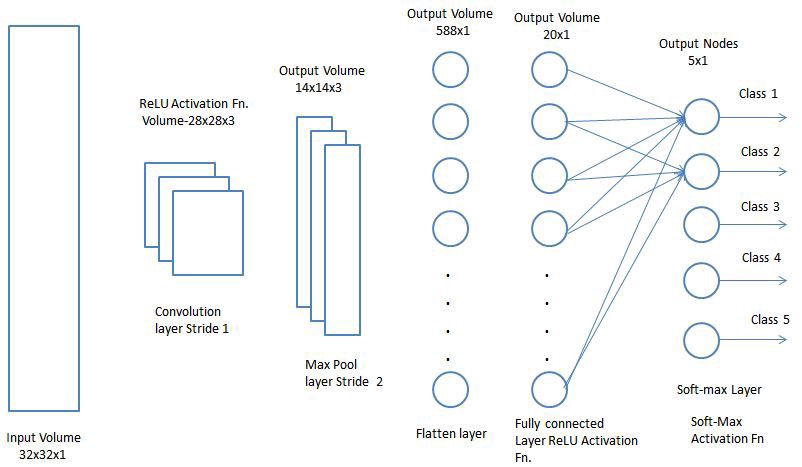


Types of Pooling

The Convolutional Layer and the Pooling Layer, together form the i-th layer of a Convolutional Neural Network. Depending on the complexities in the images, the number of such layers may be increased for capturing low-levels details even further, but at the cost of more computational power.

After going through the above process, we have successfully enabled the model to understand the features. Moving on, we are going to flatten the final output and feed it to a regular Neural Network for classification purposes.

**Classification — Fully Connected Layer (FC Layer)**



Adding a Fully-Connected layer is a (usually) cheap way of learning non-linear combinations of the high-level features as represented by the output of the convolutional layer. The Fully-Connected layer is learning a possibly non-linear function in that space.

Now that we have converted our input image into a suitable form for our Multi-Level Perceptron, we shall flatten the image into a column vector. The flattened output is fed to a feed-forward neural network and backpropagation applied to every iteration of training. Over a series of epochs, the model is able to distinguish between dominating and certain low-level features in images and classify them using the **SoftMax Classification** technique.

# **Modules Used**

## **TensorFlow**

TensorFlow is a free and open-source software library for dataflow and differentiable programming across a range of tasks. It is a symbolic math library, and is also used for machine learning applications such as neural networks. It is used for both research and production at Google. ‍ It is a standard expectation in the industry to have experience in TensorFlow to work in machine learning. TensorFlow was developed by the Google Brain team for internal Google use. It was released under the Apache 2.0 open-source license on November 9, 2015.

Starting in 2011, Google Brain built DistBelief as a proprietary machine learning system based on deep learning neural networks. Its use grew rapidly across diverse Alphabet companies in both research and commercial applications. Google assigned multiple computer scientists, including Jeff Dean, to simplify and refactor the codebase of DistBelief into a faster, more robust application-grade library, which became TensorFlow. In 2009, the team, led by Geoffrey Hinton, had implemented generalized backpropagation and other improvements which allowed generation of neural networks with substantially higher accuracy, for instance a 25% reduction in errors in speech recognition.

TensorFlow is Google Brain's second-generation system. Version 1.0.0 was released on February 11, 2017. While the reference implementation runs on single devices, TensorFlow can run on multiple CPUs and GPUs (with optional CUDA and SYCL extensions for general-purpose computing on graphics processing units). TensorFlow is available on 64-bit Linux, macOS, Windows, and mobile computing platforms including Android and iOS.

Its flexible architecture allows for the easy deployment of computation across a variety of platforms (CPUs, GPUs, TPUs), and from desktops to clusters of servers to mobile and edge devices.

TensorFlow computations are expressed as stateful dataflow graphs. The name TensorFlow derives from the operations that such neural networks perform on multidimensional data arrays, which are referred to as tensors. During the Google I/O Conference in June 2016, Jeff Dean stated that 1,500 repositories on GitHub mentioned TensorFlow, of which only 5 were from Google.

Eager Execution

TensorFlow's eager execution is an imperative programming environment that evaluates operations immediately, without building graphs: operations return concrete values instead of constructing a computational graph to run later. This makes it easy to get started with TensorFlow and debug models, and it reduces boilerplate as well. To follow along with this guide, run the code samples below in an interactive python interpreter.

Eager execution is a flexible machine learning platform for research and experimentation, providing:

* *An intuitive interface*—Structure your code naturally and use Python data structures. Quickly iterate on small models and small data.
* *Easier debugging*—Call ops directly to inspect running models and test changes. Use standard Python debugging tools for immediate error reporting.
* *Natural control flow*—Use Python control flow instead of graph control flow, simplifying the specification of dynamic models.

Eager execution supports most TensorFlow operations and GPU acceleration.

## **Tensor Values**

The central unit of data in TensorFlow is the **tensor**. A tensor consists of a set of primitive values shaped into an array of any number of dimensions. A tensor's **rank** is its number of dimensions, while its **shape** is a tuple of integers specifying the array's length along each dimension. Here are some examples of tensor values:

3. # a rank 0 tensor; a scalar with shape [],  
[1., 2., 3.] # a rank 1 tensor; a vector with shape [3]  
[[1., 2., 3.], [4., 5., 6.]] # a rank 2 tensor; a matrix with shape [2, 3]  
[[[1., 2., 3.]], [[7., 8., 9.]]] # a rank 3 tensor with shape [2, 1, 3]

TensorFlow uses numpy arrays to represent tensor **values**.

## **TensorFlow Core Walkthrough**

You might think of TensorFlow Core programs as consisting of two discrete sections:

1. Building the computational graph (a [tf.Graph](https://www.tensorflow.org/api_docs/python/tf/Graph)).
2. Running the computational graph (using a [tf.Session](https://www.tensorflow.org/api_docs/python/tf/Session)).

### **Graph**

A **computational graph** is a series of TensorFlow operations arranged into a graph. The graph is composed of two types of objects.

* [tf.Operation](https://www.tensorflow.org/api_docs/python/tf/Operation) (or "ops"): The nodes of the graph. Operations describe calculations that consume and produce tensors.
* [tf.Tensor](https://www.tensorflow.org/api_docs/python/tf/Tensor): The edges in the graph. These represent the values that will flow through the graph. Most TensorFlow functions return tf.Tensors.

Let's build a simple computational graph. The most basic operation is a constant. The Python function that builds the operation takes a tensor value as input. The resulting operation takes no inputs. When run, it outputs the value that was passed to the constructor. We can create two floating point constants a and b as follows:

a = tf.constant(3.0, dtype=tf.float32)  
b = tf.constant(4.0) # also tf.float32 implicitly  
total = a + b  
print(a)  
print(b)  
print(total)

The print statements produce:

Tensor("Const:0", shape=(), dtype=float32)  
Tensor("Const\_1:0", shape=(), dtype=float32)  
Tensor("add:0", shape=(), dtype=float32)

Notice that printing the tensors does not output the values 3.0, 4.0, and 7.0 as you might expect. The above statements only build the computation graph. These [tf.Tensor](https://www.tensorflow.org/api_docs/python/tf/Tensor) objects just represent the results of the operations that will be run.

Each operation in a graph is given a unique name. This name is independent of the names the objects are assigned to in Python. Tensors are named after the operation that produces them followed by an output index, as in "add:0" above.

### **TensorBoard**

TensorFlow provides a utility called TensorBoard. One of TensorBoard's many capabilities is visualizing a computation graph. You can easily do this with a few simple commands.

First you save the computation graph to a TensorBoard summary file as follows:

writer = tf.summary.FileWriter('.')  
writer.add\_graph(tf.get\_default\_graph())  
writer.flush()

This will produce an event file in the current directory with a name in the following format:

events.out.tfevents.{timestamp}.{hostname}

Now, in a new terminal, launch TensorBoard with the following shell command:

tensorboard --logdir .

Then open TensorBoard's [graphs page](http://localhost:6006/#graphs) in your browser, and you should see a graph similar to the following:



For more about TensorBoard's graph visualization tools see [TensorBoard: Graph Visualization](https://www.tensorflow.org/guide/graph_viz).

### **Session**

To evaluate tensors, instantiate a [tf.Session](https://www.tensorflow.org/api_docs/python/tf/Session) object, informally known as a **session**. A session encapsulates the state of the TensorFlow runtime, and runs TensorFlow operations. If a [tf.Graph](https://www.tensorflow.org/api_docs/python/tf/Graph) is like a .py file, a [tf.Session](https://www.tensorflow.org/api_docs/python/tf/Session) is like the python executable.

The following code creates a [tf.Session](https://www.tensorflow.org/api_docs/python/tf/Session) object and then invokes its run method to evaluate the total tensor we created above:

sess = tf.Session()  
print(sess.run(total))

When you request the output of a node with Session.run TensorFlow backtracks through the graph and runs all the nodes that provide input to the requested output node. So this prints the expected value of 7.0:

7.0

You can pass multiple tensors to [tf.Session.run](https://www.tensorflow.org/api_docs/python/tf/Session" \l "run). The run method transparently handles any combination of tuples or dictionaries, as in the following example:

print(sess.run({'ab':(a, b), 'total':total}))

which returns the results in a structure of the same layout:

{'total': 7.0, 'ab': (3.0, 4.0)}

During a call to [tf.Session.run](https://www.tensorflow.org/api_docs/python/tf/Session" \l "run) any [tf.Tensor](https://www.tensorflow.org/api_docs/python/tf/Tensor) only has a single value. For example, the following code calls [tf.random\_uniform](https://www.tensorflow.org/api_docs/python/tf/random/uniform) to produce a [tf.Tensor](https://www.tensorflow.org/api_docs/python/tf/Tensor) that generates a random 3-element vector (with values in [0,1)):

vec = tf.random\_uniform(shape=(3,))  
out1 = vec + 1  
out2 = vec + 2  
print(sess.run(vec))  
print(sess.run(vec))  
print(sess.run((out1, out2)))

The result shows a different random value on each call to run, but a consistent value during a single run (out1 and out2 receive the same random input):

[ 0.52917576  0.64076328  0.68353939]  
[ 0.66192627  0.89126778  0.06254101]  
(  
  array([ 1.88408756,  1.87149239,  1.84057522], dtype=float32),  
  array([ 2.88408756,  2.87149239,  2.84057522], dtype=float32)  
)

Some TensorFlow functions return tf.Operations instead of tf.Tensors. The result of calling run on an Operation is None. You run an operation to cause a side-effect, not to retrieve a value. Examples of this include the [initialization](https://www.tensorflow.org/guide/low_level_intro#Initializing%20Layers), and [training](https://www.tensorflow.org/guide/low_level_intro#Training) ops demonstrated later.

### **Feeding**

As it stands, this graph is not especially interesting because it always produces a constant result. A graph can be parameterized to accept external inputs, known as **placeholders**. A **placeholder** is a promise to provide a value later, like a function argument.

x = tf.placeholder(tf.float32)  
y = tf.placeholder(tf.float32)  
z = x + y

The preceding three lines are a bit like a function in which we define two input parameters (x and y) and then an operation on them. We can evaluate this graph with multiple inputs by using the feed\_dict argument of the [tf.Session.run](https://www.tensorflow.org/api_docs/python/tf/Session" \l "run) method to feed concrete values to the placeholders:

print(sess.run(z, feed\_dict={x: 3, y: 4.5}))  
print(sess.run(z, feed\_dict={x: [1, 3], y: [2, 4]}))

This results in the following output:

7.5  
[ 3.  7.]

Also note that the feed\_dict argument can be used to overwrite any tensor in the graph. The only difference between placeholders and other tf.Tensors is that placeholders throw an error if no value is fed to them.

## **Datasets**

Placeholders work for simple experiments, but [tf.data](https://www.tensorflow.org/api_docs/python/tf/data) are the preferred method of streaming data into a model.

To get a runnable [tf.Tensor](https://www.tensorflow.org/api_docs/python/tf/Tensor) from a Dataset you must first convert it to a [tf.data.Iterator](https://www.tensorflow.org/api_docs/python/tf/data/Iterator), and then call the Iterator's [tf.data.Iterator.get\_next](https://www.tensorflow.org/api_docs/python/tf/data/Iterator" \l "get_next) method.

The simplest way to create an Iterator is with the [tf.data.Dataset.make\_one\_shot\_iterator](https://www.tensorflow.org/api_docs/python/tf/data/Dataset" \l "make_one_shot_iterator) method. For example, in the following code the next\_item tensor will return a row from the my\_data array on each run call:

my\_data = [  
    [0, 1,],  
    [2, 3,],  
    [4, 5,],  
    [6, 7,],  
]  
slices = tf.data.Dataset.from\_tensor\_slices(my\_data)  
next\_item = slices.make\_one\_shot\_iterator().get\_next()

Reaching the end of the data stream causes Dataset to throw an [tf.errors.OutOfRangeError](https://www.tensorflow.org/api_docs/python/tf/errors/OutOfRangeError). For example, the following code reads the next\_item until there is no more data to read:

while True:  
  try:  
    print(sess.run(next\_item))  
  except tf.errors.OutOfRangeError:  
    break

If the Dataset depends on stateful operations you may need to initialize the iterator before using it, as shown below:

r = tf.random\_normal([10,3])  
dataset = tf.data.Dataset.from\_tensor\_slices(r)  
iterator = dataset.make\_initializable\_iterator()  
next\_row = iterator.get\_next()  
  
sess.run(iterator.initializer)  
while True:  
  try:  
    print(sess.run(next\_row))  
  except tf.errors.OutOfRangeError:  
    break

For more details on Datasets and Iterators see: [Importing Data](https://www.tensorflow.org/guide/datasets).

## **Layers**

A trainable model must modify the values in the graph to get new outputs with the same input. [tf.layers](https://www.tensorflow.org/api_docs/python/tf/layers) are the preferred way to add trainable parameters to a graph.

Layers package together both the variables and the operations that act on them. For example a [densely-connected layer](https://developers.google.com/machine-learning/glossary/#fully_connected_layer)performs a weighted sum across all inputs for each output and applies an optional [activation function](https://developers.google.com/machine-learning/glossary/#activation_function). The connection weights and biases are managed by the layer object.

### **Creating Layers**

The following code creates a [tf.layers.Dense](https://www.tensorflow.org/api_docs/python/tf/layers/Dense) layer that takes a batch of input vectors, and produces a single output value for each. To apply a layer to an input, call the layer as if it were a function. For example:

x = tf.placeholder(tf.float32, shape=[None, 3])  
linear\_model = tf.layers.Dense(units=1)  
y = linear\_model(x)

The layer inspects its input to determine sizes for its internal variables. So here we must set the shape of the x placeholder so that the layer can build a weight matrix of the correct size.

Now that we have defined the calculation of the output, y, there is one more detail we need to take care of before we run the calculation.

# Opencv

OpenCV (Open Source Computer Vision Library) is released under a BSD license and hence it’s free for both academic and commercial use. It has C++, Python and Java interfaces and supports Windows, Linux, Mac OS, iOS and Android. OpenCV was designed for computational efficiency and with a strong focus on real-time applications. Written in optimized C/C++, the library can take advantage of multi-core processing. Enabled with OpenCL, it can take advantage of the hardware acceleration of the underlying heterogeneous compute platform.

OpenCV (Open Source Computer Vision Library) is an open source computer vision and machine learning software library. OpenCV was built to provide a common infrastructure for computer vision applications and to accelerate the use of machine perception in the commercial products. Being a BSD-licensed product, OpenCV makes it easy for businesses to utilize and modify the code.

The library has more than 2500 optimized algorithms, which includes a comprehensive set of both classic and state-of-the-art computer vision and machine learning algorithms. These algorithms can be used to detect and recognize faces, identify objects, classify human actions in videos, track camera movements, track moving objects, extract 3D models of objects, produce 3D point clouds from stereo cameras, stitch images together to produce a high resolution image of an entire scene, find similar images from an image database, remove red eyes from images taken using flash, follow eye movements, recognize scenery and establish markers to overlay it with augmented reality, etc. OpenCV has more than 47 thousand people of user community and estimated number of downloads exceeding 14 million. The library is used extensively in companies, research groups and by governmental bodies.

Along with well-established companies like Google, Yahoo, Microsoft, Intel, IBM, Sony, Honda, Toyota that employ the library, there are many startups such as Applied Minds, VideoSurf, and Zeitera, that make extensive use of OpenCV. OpenCV’s deployed uses span the range from stitching streetview images together, detecting intrusions in surveillance video in Israel, monitoring mine equipment in China, helping robots navigate and pick up objects at Willow Garage, detection of swimming pool drowning accidents in Europe, running interactive art in Spain and New York, checking runways for debris in Turkey, inspecting labels on products in factories around the world on to rapid face detection in Japan.

It has C++, Python, Java and MATLAB interfaces and supports Windows, Linux, Android and Mac OS. OpenCV leans mostly towards real-time vision applications and takes advantage of MMX and SSE instructions when available. A full-featured CUDAand OpenCL interfaces are being actively developed right now. There are over 500 algorithms and about 10 times as many functions that compose or support those algorithms. OpenCV is written natively in C++ and has a templated interface that works seamlessly with STL containers.

OpenCV has a modular structure, which means that the package includes several shared or static libraries. The following modules are available:

* **Core functionality** (**core**) - a compact module defining basic data structures, including the dense multi-dimensional array Mat and basic functions used by all other modules.
* **Image Processing** (**imgproc**) - an image processing module that includes linear and non-linear image filtering, geometrical image transformations (resize, affine and perspective warping, generic table-based remapping), color space conversion, histograms, and so on.
* **Video Analysis** (**video**) - a video analysis module that includes motion estimation, background subtraction, and object tracking algorithms.
* **Camera Calibration and 3D Reconstruction** (**calib3d**) - basic multiple-view geometry algorithms, single and stereo camera calibration, object pose estimation, stereo correspondence algorithms, and elements of 3D reconstruction.
* **2D Features Framework** (**features2d**) - salient feature detectors, descriptors, and descriptor matchers.
* **Object Detection** (**objdetect**) - detection of objects and instances of the predefined classes (for example, faces, eyes, mugs, people, cars, and so on).
* **High-level GUI** (**highgui**) - an easy-to-use interface to simple UI capabilities.
* **Video I/O** (**videoio**) - an easy-to-use interface to video capturing and video codecs.
* ... some other helper modules, such as FLANN and Google test wrappers, Python bindings, and others.

OpenCV was started at Intel in 1999 by **Gary Bradsky** and the first release came out in 2000. **Vadim Pisarevsky** joined Gary Bradsky to manage Intel’s Russian software OpenCV team. In 2005, OpenCV was used on Stanley, the vehicle who won 2005 DARPA Grand Challenge. Later its active development continued under the support of Willow Garage, with Gary Bradsky and Vadim Pisarevsky leading the project. Right now, OpenCV supports a lot of algorithms related to Computer Vision and Machine Learning and it is expanding day-by-day.

Currently OpenCV supports a wide variety of programming languages like C++, Python, Java etc and is available on different platforms including Windows, Linux, OS X, Android, iOS etc. Also, interfaces based on CUDA and OpenCL are also under active development for high-speed GPU operations.

OpenCV-Python is the Python API of OpenCV. It combines the best qualities of OpenCV C++ API and Python language.

## **OpenCV-Python**

Python is a general purpose programming language started by **Guido van Rossum**, which became very popular in short time mainly because of its simplicity and code readability. It enables the programmer to express his ideas in fewer lines of code without reducing any readability.

Compared to other languages like C/C++, Python is slower. But another important feature of Python is that it can be easily extended with C/C++. This feature helps us to write computationally intensive codes in C/C++ and create a Python wrapper for it so that we can use these wrappers as Python modules. This gives us two advantages: first, our code is as fast as original C/C++ code (since it is the actual C++ code working in background) and second, it is very easy to code in Python. This is how OpenCV-Python works, it is a Python wrapper around original C++ implementation.

And the support of Numpy makes the task more easier. **Numpy** is a highly optimized library for numerical operations. It gives a MATLAB-style syntax. All the OpenCV array structures are converted to-and-from Numpy arrays. So whatever operations you can do in Numpy, you can combine it with OpenCV, which increases number of weapons in your arsenal. Besides that, several other libraries like SciPy, Matplotlib which supports Numpy can be used with this.

So OpenCV-Python is an appropriate tool for fast prototyping of computer vision problems

**Automatic Memory Management**

OpenCV handles all the memory automatically.

First of all, std::vector, **cv::Mat**, and other data structures used by the functions and methods have destructors that deallocate the underlying memory buffers when needed. This means that the destructors do not always deallocate the buffers as in case of Mat. They take into account possible data sharing. A destructor decrements the reference counter associated with the matrix data buffer. The buffer is deallocated if and only if the reference counter reaches zero, that is, when no other structures refer to the same buffer. Similarly, when a Mat instance is copied, no actual data is really copied. Instead, the reference counter is incremented to memorize that there is another owner of the same data. There is also the Mat::clone method that creates a full copy of the matrix data.

**Automatic Allocation of the Output Data**

OpenCV deallocates the memory automatically, as well as automatically allocates the memory for output function parameters most of the time. So, if a function has one or more input arrays (**cv::Mat** instances) and some output arrays, the output arrays are automatically allocated or reallocated. The size and type of the output arrays are determined from the size and type of input arrays. If needed, the functions take extra parameters that help to figure out the output array properties.

The array frame is automatically allocated by the >> operator since the video frame resolution and the bit-depth is known to the video capturing module. The array edges is automatically allocated by the cvtColor function. It has the same size and the bit-depth as the input array. The number of channels is 1 because the color conversion code **cv::COLOR\_BGR2GRAY** is passed, which means a color to grayscale conversion. Note that frame and edges are allocated only once during the first execution of the loop body since all the next video frames have the same resolution. If you somehow change the video resolution, the arrays are automatically reallocated.

The key component of this technology is the Mat::create method. It takes the desired array size and type. If the array already has the specified size and type, the method does nothing. Otherwise, it releases the previously allocated data, if any (this part involves decrementing the reference counter and comparing it with zero), and then allocates a new buffer of the required size. Most functions call the Mat::create method for each output array, and so the automatic output data allocation is implemented.

Some notable exceptions from this scheme are **cv::mixChannels**, **cv::RNG::fill**, and a few other functions and methods. They are not able to allocate the output array, so you have to do this in advance.

**Saturation Arithmetics**

As a computer vision library, OpenCV deals a lot with image pixels that are often encoded in a compact, 8- or 16-bit per channel, form and thus have a limited value range. Furthermore, certain operations on images, like color space conversions, brightness/contrast adjustments, sharpening, complex interpolation (bi-cubic, Lanczos) can produce values out of the available range. If you just store the lowest 8 (16) bits of the result, this results in visual artifacts and may affect a further image analysis. To solve this problem, the so-called *saturation* arithmetics is used. For example, to store r, the result of an operation, to an 8-bit image, you find the nearest value within the 0..255 range:

I(x,y)=min(max(round(r),0),255)

Similar rules are applied to 8-bit signed, 16-bit signed and unsigned types. This semantics is used everywhere in the library. In C++ code, it is done using the **cv::saturate\_cast**<> functions that resemble standard C++ cast operations.

**Fixed Pixel Types. Limited Use of Templates**

Templates is a great feature of C++ that enables implementation of very powerful, efficient and yet safe data structures and algorithms. However, the extensive use of templates may dramatically increase compilation time and code size. Besides, it is difficult to separate an interface and implementation when templates are used exclusively. This could be fine for basic algorithms but not good for computer vision libraries where a single algorithm may span thousands lines of code. Because of this and also to simplify development of bindings for other languages, like Python, Java, Matlab that do not have templates at all or have limited template capabilities, the current OpenCV implementation is based on polymorphism and runtime dispatching over templates. In those places where runtime dispatching would be too slow (like pixel access operators), impossible (generic **cv::Ptr**<> implementation), or just very inconvenient (**cv::saturate\_cast**<>()) the current implementation introduces small template classes, methods, and functions. Anywhere else in the current OpenCV version the use of templates is limited.

Consequently, there is a limited fixed set of primitive data types the library can operate on. That is, array elements should have one of the following types:

* 8-bit unsigned integer (uchar)
* 8-bit signed integer (schar)
* 16-bit unsigned integer (ushort)
* 16-bit signed integer (short)
* 32-bit signed integer (int)
* 32-bit floating-point number (float)
* 64-bit floating-point number (double)
* a tuple of several elements where all elements have the same type (one of the above). An array whose elements are such tuples, are called multi-channel arrays, as opposite to the single-channel arrays, whose elements are scalar values. The maximum possible number of channels is defined by the **CV\_CN\_MAX** constant, which is currently set to 512.

For these basic types, the following enumeration is applied:

enum { CV\_8U=0, CV\_8S=1, CV\_16U=2, CV\_16S=3, CV\_32S=4, CV\_32F=5, CV\_64F=6 };

Multi-channel (n-channel) types can be specified using the following options:

* **CV\_8UC1** ... **CV\_64FC4** constants (for a number of channels from 1 to 4)
* **CV\_8UC(n)** ... **CV\_64FC(n)** or **CV\_MAKETYPE(CV\_8U, n)** ... **CV\_MAKETYPE(CV\_64F, n)** macros when the number of channels is more than 4 or unknown at the compilation time.

Examples:

Mat mtx(3, 3, CV\_32F); // make a 3x3 floating-point matrix

Mat cmtx(10, 1, CV\_64FC2); // make a 10x1 2-channel floating-point

// matrix (10-element complex vector)

Mat img(Size(1920, 1080), CV\_8UC3); // make a 3-channel (color) image

// of 1920 columns and 1080 rows.

Mat grayscale(image.size(), CV\_MAKETYPE(image.depth(), 1)); // make a 1-channel image of

// the same size and same

// channel type as img

Arrays with more complex elements cannot be constructed or processed using OpenCV. Furthermore, each function or method can handle only a subset of all possible array types. Usually, the more complex the algorithm is, the smaller the supported subset of formats is. See below typical examples of such limitations:

* The face detection algorithm only works with 8-bit grayscale or color images.
* Linear algebra functions and most of the machine learning algorithms work with floating-point arrays only.
* Basic functions, such as **cv::add**, support all types.
* Color space conversion functions support 8-bit unsigned, 16-bit unsigned, and 32-bit floating-point types.

The subset of supported types for each function has been defined from practical needs and could be extended in future based on user requests.

### InputArray and OutputArray

Many OpenCV functions process dense 2-dimensional or multi-dimensional numerical arrays. Usually, such functions take cppMat as parameters, but in some cases it's more convenient to use std::vector<> (for a point set, for example) or **cv::Matx**<> (for 3x3 homography matrix and such). To avoid many duplicates in the API, special "proxy" classes have been introduced. The base "proxy" class is **cv::InputArray**. It is used for passing read-only arrays on a function input. The derived from InputArray class **cv::OutputArray** is used to specify an output array for a function. Normally, you should not care of those intermediate types (and you should not declare variables of those types explicitly) - it will all just work automatically. You can assume that instead of InputArray/OutputArray you can always use Mat, std::vector<>, **cv::Matx**<>, **cv::Vec**<> or **cv::Scalar**. When a function has an optional input or output array, and you do not have or do not want one, pass **cv::noArray()**.

### Error Handling

OpenCV uses exceptions to signal critical errors. When the input data has a correct format and belongs to the specified value range, but the algorithm cannot succeed for some reason (for example, the optimization algorithm did not converge), it returns a special error code (typically, just a boolean variable).

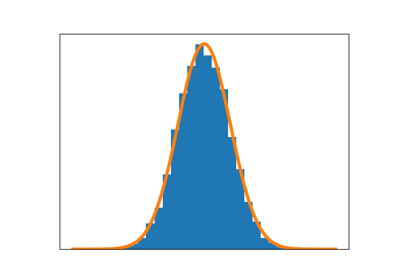
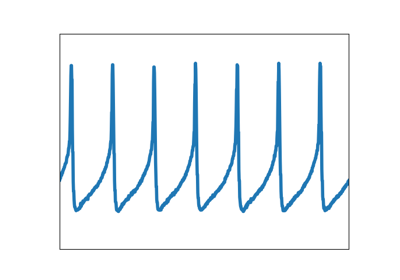
The exceptions can be instances of the **cv::Exception** class or its derivatives. In its turn, **cv::Exception** is a derivative of std::exception. So it can be gracefully handled in the code using other standard C++ library components.

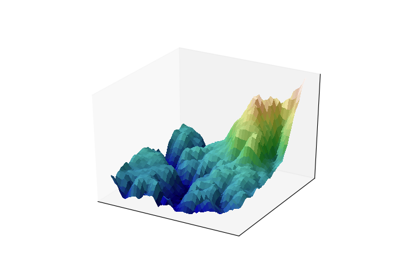
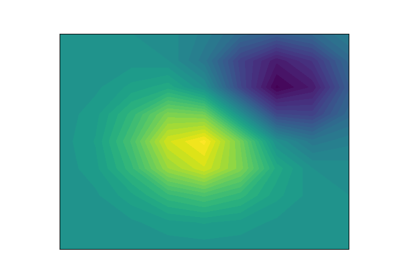
**Multi-threading and Re-enterability**

The current OpenCV implementation is fully re-enterable. That is, the same function or the same methods of different class instances can be called from different threads. Also, the same Mat can be used in different threads because the reference-counting operations use the architecture-specific atomic instructions.

# Matplotlib

Matplotlib is a Python 2D plotting library which produces publication quality figures in a variety of hardcopy formats and interactive environments across platforms. Matplotlib can be used in Python scripts, the Python and IPython shells, the Jupyter notebook, web application servers, and four graphical user interface toolkits.

[](https://matplotlib.org/3.0.3/tutorials/introductory/sample_plots.html)

[](https://matplotlib.org/3.0.3/tutorials/introductory/sample_plots.html)

Matplotlib tries to make easy things easy and hard things possible. You can generate plots, histograms, power spectra, bar charts, errorcharts, scatterplots, etc., with just a few lines of code. For examples, see the sample plots and thumbnail gallery.

For simple plotting the pyplot module provides a MATLAB-like interface, particularly when combined with IPython. For the power user, you have full control of line styles, font properties, axes properties, etc, via an object oriented interface or via a set of functions familiar to MATLAB users.

**Installing an official release**

Matplotlib and its dependencies are available as wheel packages for macOS, Windows and Linux distributions:

python -m pip install -U pip

python -m pip install -U matplotlib

Although not required, we suggest also installing IPython for interactive use. To easily install a complete Scientific Python stack, see Scientific Python Distributions below.

**macOS**

To use the native OSX backend you will need a framework build build of Python.

**Test data**

The wheels (\*.whl) on the PyPI download page do not contain test data or example code.

If you want to try the many demos that come in the Matplotlib source distribution, download the \*.tar.gz file and look in the examples subdirectory.

To run the test suite:

* extract the lib/matplotlib/tests or lib/mpl\_toolkits/tests directories from the source distribution;
* install test dependencies: pytest, Pillow, MiKTeX, GhostScript, ffmpeg, avconv, ImageMagick, and Inkscape;
* run python -mpytest.

**Third-party distributions of Matplotlib**

**Scientific Python Distributions**

Anaconda and Canopy and ActiveState are excellent choices that "just work" out of the box for Windows, macOS and common Linux platforms. WinPython is an option for Windows users. All of these distributions include Matplotlib and *lots* of other useful (data) science tools.

**Linux: using your package manager**

If you are on Linux, you might prefer to use your package manager. Matplotlib is packaged for almost every major Linux distribution.

* Debian / Ubuntu: sudo apt-get install python3-matplotlib
* Fedora: sudo dnf install python3-matplotlib
* Red Hat: sudo yum install python3-matplotlib
* Arch: sudo pacman -S python-matplotlib

**Installing from source**

If you are interested in contributing to Matplotlib development, running the latest source code, or just like to build everything yourself, it is not difficult to build Matplotlib from source. Grab the latest *tar.gz* release file from the PyPI files page, or if you want to develop Matplotlib or just need the latest bugfixed version, grab the latest git version Install from source.

The standard environment variables CC, CXX, PKG\_CONFIG are respected. This means you can set them if your toolchain is prefixed. This may be used for cross compiling.

export CC=x86\_64-pc-linux-gnu-gcc

export CXX=x86\_64-pc-linux-gnu-g++

export PKG\_CONFIG=x86\_64-pc-linux-gnu-pkg-config

Once you have satisfied the requirements detailed below (mainly Python, NumPy, libpng and FreeType), you can build Matplotlib.

cd matplotlib

python -mpip install .

We provide a setup.cfg file which you can use to customize the build process. For example, which default backend to use, whether some of the optional libraries that Matplotlib ships with are installed, and so on. This file will be particularly useful to those packaging Matplotlib.

If you have installed prerequisites to nonstandard places and need to inform Matplotlib where they are, edit setupext.py and add the base dirs to the basedir dictionary entry for your sys.platform; e.g., if the header of some required library is in /some/path/include/someheader.h, put /some/path in the basedir list for your platform.

**Dependencies**

Matplotlib requires the following dependencies:

* Python (>= 3.5)
* FreeType (>= 2.3)
* libpng (>= 1.2)
* NumPy (>= 1.10.0)
* setuptools
* cycler (>= 0.10.0)
* dateutil (>= 2.1)
* kiwisolver (>= 1.0.0)
* pyparsing

Optionally, you can also install a number of packages to enable better user interface toolkits. See What is a backend? for more details on the optional Matplotlib backends and the capabilities they provide.

* tk (>= 8.3, != 8.6.0 or 8.6.1): for the Tk-based backends;
* PyQt4 (>= 4.6) or PySide (>= 1.0.3): for the Qt4-based backends;
* PyQt5: for the Qt5-based backends;
* PyGObject or pgi (>= 0.0.11.2): for the GTK3-based backends;
* wxpython (>= 4): for the WX-based backends;
* cairocffi (>= 0.8) or pycairo: for the cairo-based backends;
* Tornado: for the WebAgg backend;

For better support of animation output format and image file formats, LaTeX, etc., you can install the following:

* ffmpeg/avconv: for saving movies;
* ImageMagick: for saving animated gifs;
* Pillow (>= 3.4): for a larger selection of image file formats: JPEG, BMP, and TIFF image files;
* LaTeX and GhostScript (>=9.0) : for rendering text with LaTeX.

**Building on Linux**

It is easiest to use your system package manager to install the dependencies.

If you are on Debian/Ubuntu, you can get all the dependencies required to build Matplotlib with:

sudo apt-get build-dep python-matplotlib

If you are on Fedora, you can get all the dependencies required to build Matplotlib with:

sudo dnf builddep python-matplotlib

If you are on RedHat, you can get all the dependencies required to build Matplotlib by first installing yum-builddep and then running:

su -c "yum-builddep python-matplotlib"

These commands do not build Matplotlib, but instead get and install the build dependencies, which will make building from source easier.

**Building on macOS**

The build situation on macOS is complicated by the various places one can get the libpng and FreeType requirements (MacPorts, Fink, /usr/X11R6), the different architectures (e.g., x86, ppc, universal), and the different macOS versions (e.g., 10.4 and 10.5). We recommend that you build the way we do for the macOS release: get the source from the tarball or the git repository and install the required dependencies through a third-party package manager. Two widely used package managers are Homebrew, and MacPorts. The following example illustrates how to install libpng and FreeType using brew:

brew install libpng freetype pkg-config

If you are using MacPorts, execute the following instead:

port install libpng freetype pkgconfig

After installing the above requirements, install Matplotlib from source by executing:

python -mpip install .

Note that your environment is somewhat important. Some conda users have found that, to run the tests, their PYTHONPATH must include /path/to/anaconda/.../site-packages and their DYLD\_FALLBACK\_LIBRARY\_PATH must include /path/to/anaconda/lib.

**Building on Windows**

The Python shipped from https://www.python.org is compiled with Visual Studio 2015 for 3.5+. Python extensions should be compiled with the same compiler, see e.g. https://packaging.python.org/guides/packaging-binary-extensions/#setting-up-a-build-environment-on-windows for how to set up a build environment.

Since there is no canonical Windows package manager, the methods for building FreeType, zlib, and libpng from source code are documented as a build script at matplotlib-winbuild.

There are a few possibilities to build Matplotlib on Windows:

* Wheels via matplotlib-winbuild
* Wheels by using conda packages (see below)
* Conda packages (see below)

**Wheel builds using conda packages**

This is a wheel build, but we use conda packages to get all the requirements. The binary requirements (png, FreeType,...) are statically linked and therefore not needed during the wheel install.

Set up the conda environment. Note, if you want a qt backend, add pyqt to the list of conda packages.

conda create -n "matplotlib\_build" python=3.7 numpy python-dateutil pyparsing tornado cycler tk libpng zlib freetype msinttypes

conda activate matplotlib\_build

For building, call the script build\_alllocal.cmd in the root folder of the repository:

build\_alllocal.cmd

## General Concepts

matplotlib has an extensive codebase that can be daunting to many new users. However, most of matplotlib can be understood with a fairly simple conceptual framework and knowledge of a few important points.

Plotting requires action on a range of levels, from the most general (e.g., 'contour this 2-D array') to the most specific (e.g., 'color this screen pixel red'). The purpose of a plotting package is to assist you in visualizing your data as easily as possible, with all the necessary control -- that is, by using relatively high-level commands most of the time, and still have the ability to use the low-level commands when needed.

Therefore, everything in matplotlib is organized in a hierarchy. At the top of the hierarchy is the matplotlib "state-machine environment" which is provided by the matplotlib.pyplot module. At this level, simple functions are used to add plot elements (lines, images, text, etc.) to the current axes in the current figure.

**Note**

Pyplot's state-machine environment behaves similarly to MATLAB and should be most familiar to users with MATLAB experience.

The next level down in the hierarchy is the first level of the object-oriented interface, in which pyplot is used only for a few functions such as figure creation, and the user explicitly creates and keeps track of the figure and axes objects. At this level, the user uses pyplot to create figures, and through those figures, one or more axes objects can be created. These axes objects are then used for most plotting actions.

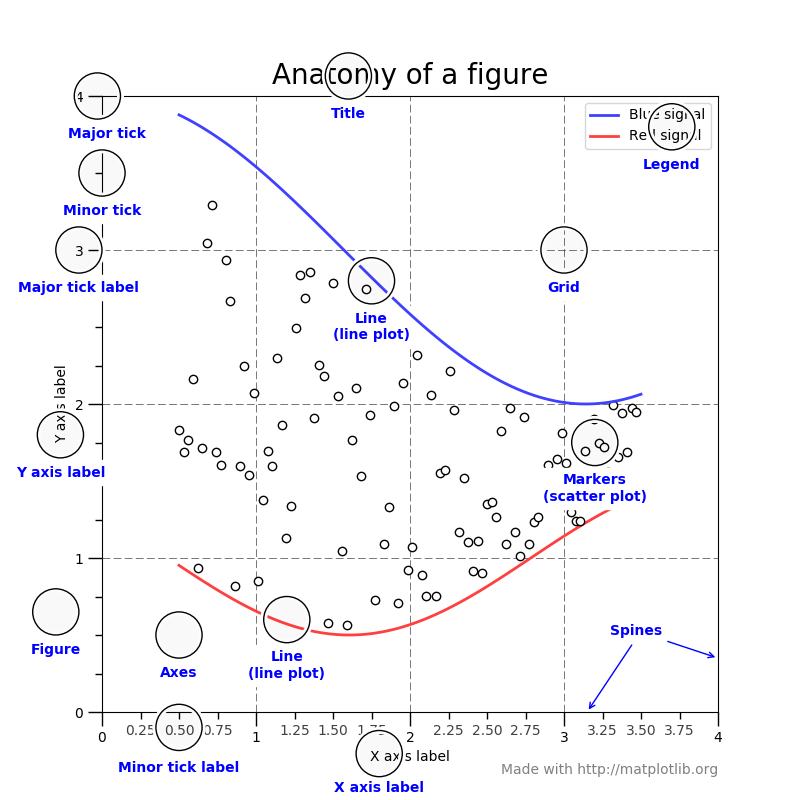
For even more control -- which is essential for things like embedding matplotlib plots in GUI applications -- the pyplot level may be dropped completely, leaving a purely object-oriented approach.

*# sphinx\_gallery\_thumbnail\_number = 3*

**import** **matplotlib.pyplot** **as** **plt**

**import** **numpy** **as** **np**

## Parts of a Figure

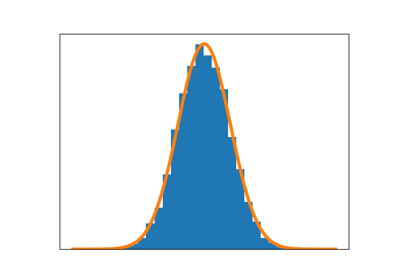
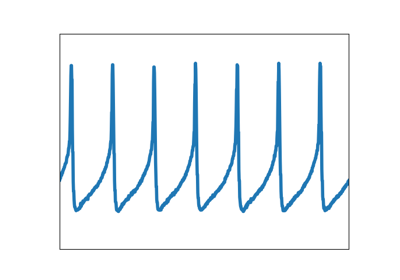


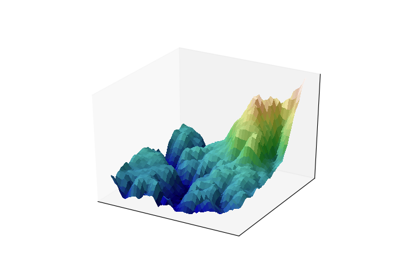
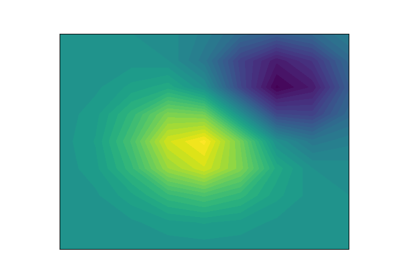
### Figure

The **whole** figure. The figure keeps track of all the child Axes, a smattering of 'special' artists (titles, figure legends, etc), and the **canvas**. (Don't worry too much about the canvas, it is crucial as it is the object that actually does the drawing to get you your plot, but as the user it is more-or-less invisible to you). A figure can have any number of Axes, but to be useful should have at least one.

# Matplotlib

Matplotlib is a Python 2D plotting library which produces publication quality figures in a variety of hardcopy formats and interactive environments across platforms. Matplotlib can be used in Python scripts, the Python and IPython shells, the Jupyter notebook, web application servers, and four graphical user interface toolkits.

[](https://matplotlib.org/3.0.3/tutorials/introductory/sample_plots.html)

[](https://matplotlib.org/3.0.3/tutorials/introductory/sample_plots.html)

Matplotlib tries to make easy things easy and hard things possible. You can generate plots, histograms, power spectra, bar charts, errorcharts, scatterplots, etc., with just a few lines of code. For examples, see the sample plots and thumbnail gallery.

For simple plotting the pyplot module provides a MATLAB-like interface, particularly when combined with IPython. For the power user, you have full control of line styles, font properties, axes properties, etc, via an object oriented interface or via a set of functions familiar to MATLAB users.

**Installing an official release**

Matplotlib and its dependencies are available as wheel packages for macOS, Windows and Linux distributions:

python -m pip install -U pip

python -m pip install -U matplotlib

Although not required, we suggest also installing IPython for interactive use. To easily install a complete Scientific Python stack, see Scientific Python Distributions below.

**macOS**

To use the native OSX backend you will need a framework build build of Python.

**Test data**

The wheels (\*.whl) on the PyPI download page do not contain test data or example code.

If you want to try the many demos that come in the Matplotlib source distribution, download the \*.tar.gz file and look in the examples subdirectory.

To run the test suite:

* extract the lib/matplotlib/tests or lib/mpl\_toolkits/tests directories from the source distribution;
* install test dependencies: pytest, Pillow, MiKTeX, GhostScript, ffmpeg, avconv, ImageMagick, and Inkscape;
* run python -mpytest.

**Third-party distributions of Matplotlib**

**Scientific Python Distributions**

Anaconda and Canopy and ActiveState are excellent choices that "just work" out of the box for Windows, macOS and common Linux platforms. WinPython is an option for Windows users. All of these distributions include Matplotlib and *lots* of other useful (data) science tools.

**Linux: using your package manager**

If you are on Linux, you might prefer to use your package manager. Matplotlib is packaged for almost every major Linux distribution.

* Debian / Ubuntu: sudo apt-get install python3-matplotlib
* Fedora: sudo dnf install python3-matplotlib
* Red Hat: sudo yum install python3-matplotlib
* Arch: sudo pacman -S python-matplotlib

**Installing from source**

If you are interested in contributing to Matplotlib development, running the latest source code, or just like to build everything yourself, it is not difficult to build Matplotlib from source. Grab the latest *tar.gz* release file from the PyPI files page, or if you want to develop Matplotlib or just need the latest bugfixed version, grab the latest git version Install from source.

The standard environment variables CC, CXX, PKG\_CONFIG are respected. This means you can set them if your toolchain is prefixed. This may be used for cross compiling.

export CC=x86\_64-pc-linux-gnu-gcc

export CXX=x86\_64-pc-linux-gnu-g++

export PKG\_CONFIG=x86\_64-pc-linux-gnu-pkg-config

Once you have satisfied the requirements detailed below (mainly Python, NumPy, libpng and FreeType), you can build Matplotlib.

cd matplotlib

python -mpip install .

We provide a setup.cfg file which you can use to customize the build process. For example, which default backend to use, whether some of the optional libraries that Matplotlib ships with are installed, and so on. This file will be particularly useful to those packaging Matplotlib.

If you have installed prerequisites to nonstandard places and need to inform Matplotlib where they are, edit setupext.py and add the base dirs to the basedir dictionary entry for your sys.platform; e.g., if the header of some required library is in /some/path/include/someheader.h, put /some/path in the basedir list for your platform.

**Dependencies**

Matplotlib requires the following dependencies:

* Python (>= 3.5)
* FreeType (>= 2.3)
* libpng (>= 1.2)
* NumPy (>= 1.10.0)
* setuptools
* cycler (>= 0.10.0)
* dateutil (>= 2.1)
* kiwisolver (>= 1.0.0)
* pyparsing

Optionally, you can also install a number of packages to enable better user interface toolkits. See What is a backend? for more details on the optional Matplotlib backends and the capabilities they provide.

* tk (>= 8.3, != 8.6.0 or 8.6.1): for the Tk-based backends;
* PyQt4 (>= 4.6) or PySide (>= 1.0.3): for the Qt4-based backends;
* PyQt5: for the Qt5-based backends;
* PyGObject or pgi (>= 0.0.11.2): for the GTK3-based backends;
* wxpython (>= 4): for the WX-based backends;
* cairocffi (>= 0.8) or pycairo: for the cairo-based backends;
* Tornado: for the WebAgg backend;

For better support of animation output format and image file formats, LaTeX, etc., you can install the following:

* ffmpeg/avconv: for saving movies;
* ImageMagick: for saving animated gifs;
* Pillow (>= 3.4): for a larger selection of image file formats: JPEG, BMP, and TIFF image files;
* LaTeX and GhostScript (>=9.0) : for rendering text with LaTeX.

**Building on Linux**

It is easiest to use your system package manager to install the dependencies.

If you are on Debian/Ubuntu, you can get all the dependencies required to build Matplotlib with:

sudo apt-get build-dep python-matplotlib

If you are on Fedora, you can get all the dependencies required to build Matplotlib with:

sudo dnf builddep python-matplotlib

If you are on RedHat, you can get all the dependencies required to build Matplotlib by first installing yum-builddep and then running:

su -c "yum-builddep python-matplotlib"

These commands do not build Matplotlib, but instead get and install the build dependencies, which will make building from source easier.

**Building on macOS**

The build situation on macOS is complicated by the various places one can get the libpng and FreeType requirements (MacPorts, Fink, /usr/X11R6), the different architectures (e.g., x86, ppc, universal), and the different macOS versions (e.g., 10.4 and 10.5). We recommend that you build the way we do for the macOS release: get the source from the tarball or the git repository and install the required dependencies through a third-party package manager. Two widely used package managers are Homebrew, and MacPorts. The following example illustrates how to install libpng and FreeType using brew:

brew install libpng freetype pkg-config

If you are using MacPorts, execute the following instead:

port install libpng freetype pkgconfig

After installing the above requirements, install Matplotlib from source by executing:

python -mpip install .

Note that your environment is somewhat important. Some conda users have found that, to run the tests, their PYTHONPATH must include /path/to/anaconda/.../site-packages and their DYLD\_FALLBACK\_LIBRARY\_PATH must include /path/to/anaconda/lib.

**Building on Windows**

The Python shipped from https://www.python.org is compiled with Visual Studio 2015 for 3.5+. Python extensions should be compiled with the same compiler, see e.g. https://packaging.python.org/guides/packaging-binary-extensions/#setting-up-a-build-environment-on-windows for how to set up a build environment.

Since there is no canonical Windows package manager, the methods for building FreeType, zlib, and libpng from source code are documented as a build script at matplotlib-winbuild.

There are a few possibilities to build Matplotlib on Windows:

* Wheels via matplotlib-winbuild
* Wheels by using conda packages (see below)
* Conda packages (see below)

**Wheel builds using conda packages**

This is a wheel build, but we use conda packages to get all the requirements. The binary requirements (png, FreeType,...) are statically linked and therefore not needed during the wheel install.

Set up the conda environment. Note, if you want a qt backend, add pyqt to the list of conda packages.

conda create -n "matplotlib\_build" python=3.7 numpy python-dateutil pyparsing tornado cycler tk libpng zlib freetype msinttypes

conda activate matplotlib\_build

For building, call the script build\_alllocal.cmd in the root folder of the repository:

build\_alllocal.cmd

## General Concepts

matplotlib has an extensive codebase that can be daunting to many new users. However, most of matplotlib can be understood with a fairly simple conceptual framework and knowledge of a few important points.

Plotting requires action on a range of levels, from the most general (e.g., 'contour this 2-D array') to the most specific (e.g., 'color this screen pixel red'). The purpose of a plotting package is to assist you in visualizing your data as easily as possible, with all the necessary control -- that is, by using relatively high-level commands most of the time, and still have the ability to use the low-level commands when needed.

Therefore, everything in matplotlib is organized in a hierarchy. At the top of the hierarchy is the matplotlib "state-machine environment" which is provided by the matplotlib.pyplot module. At this level, simple functions are used to add plot elements (lines, images, text, etc.) to the current axes in the current figure.

**Note**

Pyplot's state-machine environment behaves similarly to MATLAB and should be most familiar to users with MATLAB experience.

The next level down in the hierarchy is the first level of the object-oriented interface, in which pyplot is used only for a few functions such as figure creation, and the user explicitly creates and keeps track of the figure and axes objects. At this level, the user uses pyplot to create figures, and through those figures, one or more axes objects can be created. These axes objects are then used for most plotting actions.

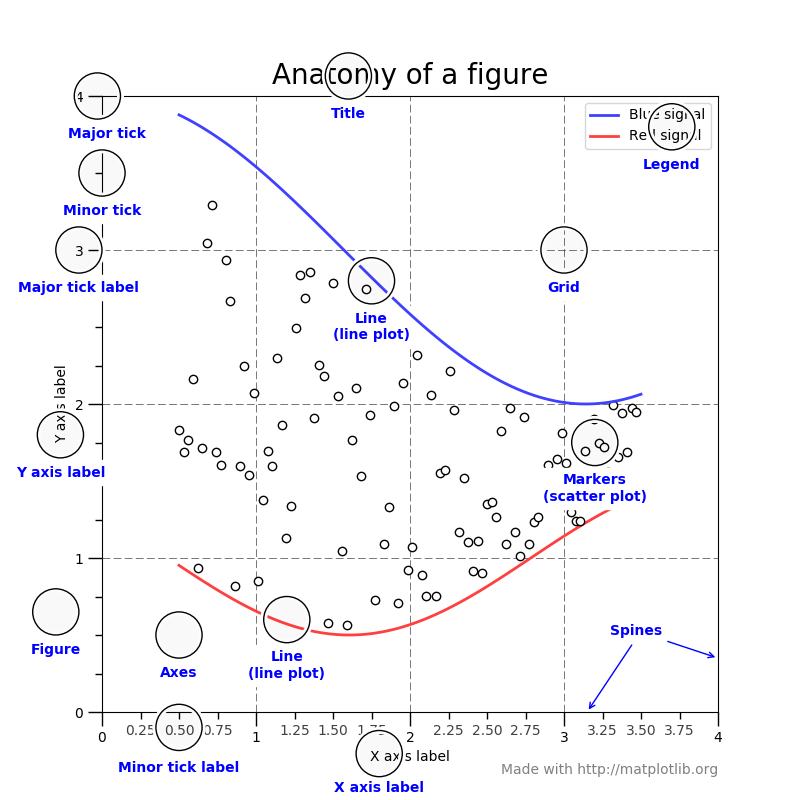
For even more control -- which is essential for things like embedding matplotlib plots in GUI applications -- the pyplot level may be dropped completely, leaving a purely object-oriented approach.

*# sphinx\_gallery\_thumbnail\_number = 3*

**import** **matplotlib.pyplot** **as** **plt**

**import** **numpy** **as** **np**

## Parts of a Figure



### Figure

The **whole** figure. The figure keeps track of all the child Axes, a smattering of 'special' artists (titles, figure legends, etc), and the **canvas**. (Don't worry too much about the canvas, it is crucial as it is the object that actually does the drawing to get you your plot, but as the user it is more-or-less invisible to you). A figure can have any number of Axes, but to be useful should have at least one.

# **SciPy**

SciPy is a collection of mathematical algorithms and convenience functions built on the Numpy extension of Python. It adds significant power to the interactive Python session by providing the user with high-level commands and classes for manipulating and visualizing data. With SciPy an interactive Python session becomes a data-processing and system-prototyping environment rivaling systems such as MATLAB, IDL, Octave, R-Lab, and SciLab.

The additional benefit of basing SciPy on Python is that this also makes a powerful programming language available for use in developing sophisticated programs and specialized applications. Scientific applications using SciPy benefit from the development of additional modules in numerous niches of the software landscape by developers across the world. Everything from parallel programming to web and data-base subroutines and classes have been made available to the Python programmer. All of this power is available in addition to the mathematical libraries in SciPy.

This tutorial will acquaint the first-time user of SciPy with some of its most important features. It assumes that the user has already installed the SciPy package. Some general Python facility is also assumed, such as could be acquired by working through the Python distribution’s Tutorial. For further introductory help the user is directed to the Numpy documentation.

For brevity and convenience, we will often assume that the main packages (numpy, scipy, and matplotlib) have been imported as:

>>>

**>>> import** **numpy** **as** **np**

**>>> import** **matplotlib** **as** **mpl**

**>>> import** **matplotlib.pyplot** **as** **plt**

These are the import conventions that our community has adopted after discussion on public mailing lists. You will see these conventions used throughout NumPy and SciPy source code and documentation. While we obviously don’t require you to follow these conventions in your own code, it is highly recommended.

[**SciPy Organization**](https://docs.scipy.org/doc/scipy/reference/tutorial/general.html#id2)

SciPy is organized into subpackages covering different scientific computing domains. These are summarized in the following table:

| **Subpackage** | **Description** |
| --- | --- |
| **cluster** | Clustering algorithms |
| **constants** | Physical and mathematical constants |
| **fftpack** | Fast Fourier Transform routines |
| **integrate** | Integration and ordinary differential equation solvers |
| **interpolate** | Interpolation and smoothing splines |
| [**io**](https://docs.python.org/dev/library/io.html#module-io) | Input and Output |
| **linalg** | Linear algebra |
| **ndimage** | N-dimensional image processing |
| **odr** | Orthogonal distance regression |
| **optimize** | Optimization and root-finding routines |
| [**signal**](https://docs.python.org/dev/library/signal.html#module-signal) | Signal processing |
| **sparse** | Sparse matrices and associated routines |
| **spatial** | Spatial data structures and algorithms |
| **special** | Special functions |
| **stats** | Statistical distributions and functions |

Scipy sub-packages need to be imported separately, for example:

>>>

**>>> from** **scipy** **import** linalg, optimize

Because of their ubiquitousness, some of the functions in these subpackages are also made available in the **scipy** namespace to ease their use in interactive sessions and programs. In addition, many basic array functions from **[numpy](https://docs.scipy.org/doc/numpy/reference/index.html" \l "module-numpy" \o "(in NumPy v1.16))** are also available at the top-level of the **scipy** package. Before looking at the sub-packages individually, we will first look at some of these common functions.

# Special functions ([scipy.special](https://docs.scipy.org/doc/scipy-1.2.1/reference/special.html" \l "module-scipy.special" \o "scipy.special))

The main feature of the **[scipy.special](https://docs.scipy.org/doc/scipy-1.2.1/reference/special.html" \l "module-scipy.special" \o "scipy.special)** package is the definition of numerous special functions of mathematical physics. Available functions include airy, elliptic, bessel, gamma, beta, hypergeometric, parabolic cylinder, mathieu, spheroidal wave, struve, and kelvin. There are also some low-level stats functions that are not intended for general use as an easier interface to these functions is provided by the stats module. Most of these functions can take array arguments and return array results following the same broadcasting rules as other math functions in Numerical Python. Many of these functions also accept complex numbers as input. For a complete list of the available functions with a one-line description type >>> help(special). Each function also has its own documentation accessible using help. If you don’t see a function you need, consider writing it and contributing it to the library. You can write the function in either C, Fortran, or Python. Look in the source code of the library for examples of each of these kinds of functions.

## Bessel functions of real order(jn, [jn\_zeros](https://docs.scipy.org/doc/scipy-1.2.1/reference/generated/scipy.special.jn_zeros.html" \l "scipy.special.jn_zeros" \o "scipy.special.jn_zeros))

Bessel functions are a family of solutions to Bessel’s differential equation with real or complex order alpha:

x2d2ydx2+xdydx+(x2−α2)y=0

Among other uses, these functions arise in wave propagation problems such as the vibrational modes of a thin drum head. Here is an example of a circular drum head anchored at the edge:

>>>

>>> **from** **scipy** **import** special

>>> **def** drumhead\_height(n, k, distance, angle, t):

... kth\_zero = special.jn\_zeros(n, k)[-1]

... **return** np.cos(t) \* np.cos(n\*angle) \* special.jn(n, distance\*kth\_zero)

>>> theta = np.r\_[0:2\*np.pi:50j]

>>> radius = np.r\_[0:1:50j]

>>> x = np.array([r \* np.cos(theta) **for** r **in** radius])

>>> y = np.array([r \* np.sin(theta) **for** r **in** radius])

>>> z = np.array([drumhead\_height(1, 1, r, theta, 0.5) **for** r **in** radius])

>>>

>>> **import** **matplotlib.pyplot** **as** **plt**

>>> **from** **mpl\_toolkits.mplot3d** **import** Axes3D

>>> **from** **matplotlib** **import** cm

>>> fig = plt.figure()

>>> ax = Axes3D(fig)

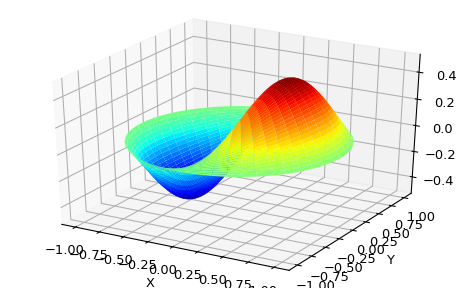
>>> ax.plot\_surface(x, y, z, rstride=1, cstride=1, cmap=cm.jet)

>>> ax.set\_xlabel('X')

>>> ax.set\_ylabel('Y')

>>> ax.set\_zlabel('Z')

>>> plt.show()



## Cython Bindings for Special Functions ([scipy.special.cython\_special](https://docs.scipy.org/doc/scipy-1.2.1/reference/special.cython_special.html" \l "module-scipy.special.cython_special" \o "scipy.special.cython_special))

Scipy also offers Cython bindings for scalar, typed versions of many of the functions in special. The following Cython code gives a simple example of how to use these functions:

**cimport** **scipy.special.cython\_special** **as** **csc**

**cdef**:

double x = 1

double complex z = 1 + 1j

double si, ci, rgam

double complex cgam

rgam = csc.gamma(x)

**print**(rgam)

cgam = csc.gamma(z)

**print**(cgam)

csc.sici(x, &si, &ci)

**print**(si, ci)

(See the [Cython documentation](http://docs.cython.org/en/latest/src/reference/compilation.html) for help with compiling Cython.) In the example the function csc.gamma works essentially like its ufunc counterpart [**gamma**](https://docs.scipy.org/doc/scipy-1.2.1/reference/generated/scipy.special.gamma.html#scipy.special.gamma), though it takes C types as arguments instead of NumPy arrays. Note in particular that the function is overloaded to support real and complex arguments; the correct variant is selected at compile time. The function csc.sici works slightly differently from **[sici](https://docs.scipy.org/doc/scipy-1.2.1/reference/generated/scipy.special.sici.html" \l "scipy.special.sici" \o "scipy.special.sici)**; for the ufunc we could write ai, bi = sici(x) whereas in the Cython version multiple return values are passed as pointers. It might help to think of this as analogous to calling a ufunc with an output array: sici(x, out=(si, ci)).

There are two potential advantages to using the Cython bindings:

* They avoid Python function overhead
* They do not require the Python Global Interpreter Lock (GIL)

The following sections discuss how to use these advantages to potentially speed up your code, though of course one should always profile the code first to make sure putting in the extra effort will be worth it.

### **Avoiding Python Function Overhead**

For the ufuncs in special, Python function overhead is avoided by vectorizing, that is, by passing an array to the function. Typically this approach works quite well, but sometimes it is more convenient to call a special function on scalar inputs inside a loop, for example when implementing your own ufunc. In this case the Python function overhead can become significant. Consider the following example:

**import** **scipy.special** **as** **sc**

**cimport** **scipy.special.cython\_special** **as** **csc**

**def** python\_tight\_loop():

**cdef**:

int n

double x = 1

**for** n **in** range(100):

sc.jv(n, x)

**def** cython\_tight\_loop():

**cdef**:

int n

double x = 1

**for** n **in** range(100):

csc.jv(n, x)

On one computer python\_tight\_loop took about 131 microseconds to run and cython\_tight\_loop took about 18.2 microseconds to run. Obviously this example is contrived: one could just call special.jv(np.arange(100), 1) and get results just as fast as incython\_tight\_loop. The point is that if Python function overhead becomes significant in your code then the Cython bindings might be useful.

### **Releasing the GIL**

One often needs to evaluate a special function at many points, and typically the evaluations are trivially parallelizable. Since the Cython bindings do not require the GIL, it is easy to run them in parallel using Cython’s prange function. For example, suppose that we wanted to compute the fundamental solution to the Helmholtz equation:

ΔxG(x,y)+k2G(x,y)=δ(x−y),

where k is the wavenumber and δ is the Dirac delta function. It is known that in two dimensions the unique (radiating) solution is

G(x,y)=i4H0(1)(k|x−y|),

where H0(1) is the Hankel function of the first kind, i.e. the function [**hankel1**](https://docs.scipy.org/doc/scipy-1.2.1/reference/generated/scipy.special.hankel1.html#scipy.special.hankel1). The following example shows how we could compute this function in parallel:

**from** **libc.math** **cimport** fabs

**cimport** **cython**

**from** **cython.parallel** **cimport** prange

**import** **numpy** **as** **np**

**import** **scipy.special** **as** **sc**

**cimport** **scipy.special.cython\_special** **as** **csc**

**def** serial\_G(k, x, y):

**return** 0.25j\*sc.hankel1(0, k\*np.abs(x - y))

**@cython**.boundscheck(False)

**@cython**.wraparound(False)

**cdef** void \_parallel\_G(double k, double[:,:] x, double[:,:] y,

double complex[:,:] out) **nogil**:

**cdef** int i, j

**for** i **in** prange(x.shape[0]):

**for** j **in** range(y.shape[0]):

out[i,j] = 0.25j\*csc.hankel1(0, k\*fabs(x[i,j] - y[i,j]))

**def** parallel\_G(k, x, y):

out = np.empty\_like(x, dtype='complex128')

\_parallel\_G(k, x, y, out)

**return** out

(For help with compiling parallel code in Cython see [here](http://docs.cython.org/en/latest/src/userguide/parallelism.html#compiling).) If the above Cython code is in a file test.pyx, then we can write an informal benchmark which compares the parallel and serial versions of the function:

**import** **timeit**

**import** **numpy** **as** **np**

**from** **test** **import** serial\_G, parallel\_G

**def** main():

k = 1

x, y = np.linspace(-100, 100, 1000), np.linspace(-100, 100, 1000)

x, y = np.meshgrid(x, y)

**def** serial():

serial\_G(k, x, y)

**def** parallel():

parallel\_G(k, x, y)

time\_serial = timeit.timeit(serial, number=3)

time\_parallel = timeit.timeit(parallel, number=3)

**print**("Serial method took {:.3} seconds".format(time\_serial))

**print**("Parallel method took {:.3} seconds".format(time\_parallel))

**if** \_\_name\_\_ == "\_\_main\_\_":

main()

On one quad-core computer the serial method took 1.29 seconds and the parallel method took 0.29 seconds.

## Functions not in [scipy.special](https://docs.scipy.org/doc/scipy-1.2.1/reference/special.html" \l "module-scipy.special" \o "scipy.special)

Some functions are not included in special because they are straightforward to implement with existing functions in NumPy and SciPy. To prevent reinventing the wheel, this section provides implementations of several such functions which hopefully illustrate how to handle similar functions. In all examples NumPy is imported as np and special is imported as sc.

The [binary entropy function](https://en.wikipedia.org/wiki/Binary_entropy_function):

**def** binary\_entropy(x):

**return** -(sc.xlogy(x, x) + sc.xlog1py(1 - x, -x))/np.log(2)

A rectangular step function on [0, 1]:

**def** step(x):

**return** 0.5\*(np.sign(x) + np.sign(1 - x))

Translating and scaling can be used to get an arbitrary step function.

The [ramp function](https://en.wikipedia.org/wiki/Ramp_function):

**def** ramp(x):

**return** np.maximum(0, x)