Foundations and Trends® in Signal Processing

Signal Processing with Python

Suggested Citation: Charles Boncelet (2017), "Signal Processing with Python", Foundations and Trends® in Signal Processing: Vol. xx, No. xx, pp 1–16. DOI: 10.1561/XXXXXXXXX.

Charles Boncelet
University of Delaware
boncelet@udel.edu

now
the essence of knowledge
Boston — Delft

Contents

Signal Processing with Python

Charles Boncelet[1]

[1] *University of Delaware; boncelet@udel.edu*

ABSTRACT

Signal Processing with Python is a guidebook for how to do signal processing computations using Python and its core scientific libraries, numpy, scipy, and pandas. The book consists of a series of Jupyter notebooks, each focusing on a different topic. Example computations include audio, spectrum estimation, filtering, image processing, and data analysis.

The IEEE currently rates Python as the most important computer language (Matlab is 15th on the IEEE list). Whole scientific communities have adopted python as their language of choice, but not yet the signal processing community. This book may help tilt the signal processing community toward python.

Signal Processing with Python is targeted at researchers and practitioners in the signal processing field who want to learn how to use python in their work and to students studying signal processing at universities.

Charles Boncelet (2017), "Signal Processing with Python", Foundations and Trends® in Signal Processing: Vol. xx, No. xx, pp 1–16. DOI: 10.1561/XXXXXXXXX.

1

Introduction to Python

Python is a widely used, general purpose, computing language. It is the most common first language taught in US universities. The IEEE estimates that Python is the most popular programming language (ahead of C and Java, as of 2017). Whole scientific communities, e.g., astronomy and machine learning, have adopted Python as their language of choice. However, the signal processing community has been slow to adopt Python.

This book demonstrates how Python can be used for a myriad of signal processing tasks. First, we discuss the history of python and the basics of the language. Then, we move on to numerical computing and signal processing applications. We give Python examples in signal filtering and analysis, audio processing, image processing, control theory, statistical signal processing, and machine learning.

In summary, Python is an excellent language for signal processing teaching and research. We predict Python will soon become the dominant language used in signal processing teaching and research at universities worldwide.

To beginners: As you read this book, have a Jupyter notebook running. Type in the commands and see the results. The simple act of running the commands will make Python more accessible and give you confidence in writing your own programs.

1.1 History of Python

From Wikipedia,

> The Python programming language was conceived in the late 1980s, and its implementation was started in December 1989 by Guido van Rossum at CWI in the Netherlands as a successor to the ABC programming

language capable of exception handling and interfacing with the Amoeba operating system. Van Rossum is Python's principal author, and his continuing central role in deciding the direction of Python is reflected in the title given to him by the Python community, Benevolent Dictator for Life (BDFL). Python was named for the BBC TV show Monty Python's Flying Circus.

Python was initially written by Guido van Rossum, affectionally known in the Python community as the "Benevolent Dictator for Life" (BDFL), in the late 1980's and early 1990's. Python was quickly picked up by the open source community. It was new, simple, and directed toward beginners in programming.

Python 1.0 was released in 1994. Unlike many other software projects, the Python community was slow to increase version numbers. Despite many incremental improvements over the years, Python 2.0 was not released until 2000. Since that time, progress has been rapid and many substantial changes have been made.

The Python community introduced Python 3.0 in 2008. This was a major, non-backward compatible, release meant to correct some of the "flaws" that had crept in the language over the years. Unfortunately, this split meant the community had two concurrent versions of Python, the 2.X series and the 3.X series. Even today (2017) there are applications that still require a Python 2.X version.

As of this writing (2017), the current version of Python is 3.6. All examples in this book are done with Python 3.6 running in a Jupyter Notebook on a Macbook running OSX 10.12. The Python distribution is Anaconda 4.3.

1.2 Installing and Running Python

Python is an open source, interpreted language. The core language is small and designed to be simple to use. Much of Python's strength's are found in its many libraries. The core distribution available from www.python.org includes many dozens of libraries.

However, the standard distribution does not include many important libraries. We recommend all readers of this book install the Python 3.6 (or later) version of the *Anaconda* distribution available from www.continuum.io/downloads. All examples in the book are done with this distribution.

Anaconda includes many additional libraries, including the following essential ones for scientific computing:

- *numpy* gives Python vector and matrix methods.
- *matplotlib* gives Python Matlab-like plotting capabilities.
- *scipy* is a large collection of scientific computation libraries.
- *pandas* is a data processing library.
- *ipython* is an alternative interpreter to the standard python interpreter.

Anaconda installs the *Anaconda-Navigator* application. Start it, and click on the launch button for the *Jupyter* notebook. Jupyter starts web server and opens a webpage in your default browser. The *jupyter* webpage allows you to navigate the file system by clicking on the file icon (in the upper left corner). Click on *New* in the upper right corner and select *Python 3* notebook.

The notebook consists of a series of *cells*. By default the cells are interpreted by the *Ipython* interpreter. Ipython is an improved version of the standard python interpreter. Ipython is the recommended interpreter for Python programs.

Jupyter cells can be switched to *Markdown*, a simple language for documentation. In the toolbar, change the pulldown from *Code* to *Markdown*. Information about *Markdown* can be found here: jupyter-notebook.readthedocs.io. A common workflow is to alternate python code cells and documentation markdown cells.

This book is a series of Jupyter notebooks, one for each chapter. Each notebook is converted to LaTeX by (a modified version) of the Jupyter `nbconvert` command. The LaTeX files are combined into the final document.

In summary, install the latest version of Anaconda. Start the Anaconda-Navigator, launch Jupyter, and open a new Python 3 notebook. The notebook runs the Ipython interpreter. Use markdown cells to document your work.

1.3 First Steps, Using Python as a Calculator

A simple way to start is to use *Python* as a desktop calculator. This, in fact, is the approach the BDFL took many years ago. Type the following in the first cell and hit *Shift-Return* (hold the Shift key down and hit Return) to execute the cell and open a new cell below. (We will explain these lines later.)

Experiment with Python by inputting various mathematical commands and executing the cells. Also, experiment with *Markdown*. Type text, then hit Shift-Enter to execute (format) the text.

Most notebooks start with the following four lines. The first imports the `numpy` library (provides numerical functions), the second imports the plotting library, the third tells the notebook to print decimals to three digits, and the fourth tells the notebook to put plots within the notebook (the alternative is to put the plots in a separate window).

```
In [1]: import numpy as np
        import matplotlib.pyplot as plt
        np.set_printoptions(precision=3)
        %matplotlib inline
```

The notebook automatically prints the output of the last line of a cell (thus minimizing the need to use the `print` statement).

```
In [2]: 5+3
```

```
Out[2]: 8
```

Two or more things can be placed on the last line. This can be handy. (Technically, when two or more python expressions are separated by commas, Ipython interprets them as a tuple. Tuples are printed inside parentheses. That is why the output is in parentheses.)

```
In [3]: 5+3 == 8, 5+3 == 9 #comparison, not equality
```

```
Out[3]: (True, False)
```

```
In [4]: 2**16, np.log2(2**16), np.log(2**16)/np.log(2)
```

```
Out[4]: (65536, 16.0, 16.0)
```

Python has two different division operators, `a/b` and `a//b`. These are shown below. Note that `5/2` is not equal to `5//2`. Division by 0 results in an ZeroDivisionError.

```
In [5]: 6/2, 6//2, 5/2, 5.0/2, 5//2, 5.0//2
```

```
Out[5]: (3.0, 3, 2.5, 2.5, 2, 2.0)
```

Modulo arithmetic uses the `%` operator or the `np.mod` function (returns remainder) and the `divmod` function (returns both quotient and remainder).

```
In [6]: np.mod(16,3), np.mod(-16,3), 16//3, 16%3, divmod(16,3)
```

```
Out[6]: (1, 2, 5, 1, (5, 1))
```

Complex numbers use the electrical engineering convention, $j = \sqrt{-1}$. Note, use `1j` or `2j`, etc., not just `j`.

```
In [7]: x = 3+4j
        x, x.real, x.imag, np.abs(x), np.angle(x), x*np.conj(x)
```

```
Out[7]: ((3+4j), 3.0, 4.0, 5.0, 0.92729521800161219, (25+0j))
```

Variable names are strings of letters, numbers, and the underscore (though variables cannot start with a number).

```
In [8]: mass = 2 #kg
        acceleration = 9.8 #m/s
        force = mass * acceleration
        print("force = ", force, "newtons")
```

```
force =  19.6 newtons
```


1.4 Sample Function to Compute Zeller's Rule

Here is an example of *Zeller's Rule* for calculating the day of the week for any date (after 1752 in the Gregorian calendar). Reference mathforum.org.

$$f = \big(k + \lfloor(13m - 1)/5\rfloor + D + \lfloor D/4\rfloor + \lfloor C/4\rfloor - 2C\big) \bmod 7$$

where f = the day of the week (0=Sunday, 1=Monday, etc.), k = day of the month, m = number of the month (1=March, 2=April, etc., 11=January of the *previous* year, 12=February of the *previous* year), D = last two digits of the year, and C = first two digits of the year.

The notation $\lfloor x\rfloor$ is the *floor* function, e.g., $\lfloor 3.7\rfloor = 3$. We use the `np.floor` function.

```
In [9]: np.floor(3.7), np.floor(-3.7)
```

```
Out[9]: (3.0, -4.0)
```

For example, let us calculate the day of the week for January 1, 2018 (It is a Monday).

```
In [10]: k = 1
         m = 11
         C = 20
         D = 17 #the previous year
         f = (k + np.floor((13*m-1)/5) + D + np.floor(D/4) +
              np.floor(C/4) - 2*C) % 7
         f
```

```
Out[10]: 1.0
```

Having done this once, we might want to do it many times. Let us write a function that calculates the day of the week.

```
In [11]: def zeller(k, m, D, C):
             """calculate day of the week for a given date

             where

             k = day of month
             m = number of month, 1=Mar, 2=Apr, ..., 11=Jan, 12=Feb
             D = last two digits of year (previous year for Jan and Feb)
             C = first two digits of year"""
             day = int(np.mod(k + np.floor((13*m-1)/5) + D + np.floor(D/4) +
                              np.floor(C/4) - 2*C, 7))
             return day
```

```
In [12]: zeller(1,11,17,20)
```

```
Out[12]: 1
```

However, the function as written is not user friendly. It requires the user to use the peculiar month and year correction. Let us wrap the zeller function with a more user friendly one.

```
In [13]: def day_of_week(day, month, year):
             """calculate day of week using Zeller's Rule

             day = day of month, 1-31
             month = month of year, January = 1, Feb = 2, etc
             year = four digit year"""
             #apply January and February correction
             if month == 1 or month == 2:
                 month += 10
                 year -= 1
             else:
                 month -= 2
             C, D = divmod(year, 100)
             return zeller(day, month, D, C)
```

```
In [14]: day_of_week(1, 1, 2018) #It's a Monday
```

```
Out[14]: 1
```

Let us calculate the day of the week for some historical dates. `weekdays` is a *dictionary* (*dict* for short), `dates` is a *list* of tuples where each *tuple* is of the form `(day, month, year, name)`. We loop over dates, compute the day of the week for each date, and use the dict to convert integer days to names of the days.

```
In [15]: weekdays = {0:'Sunday', 1:'Monday', 2:'Tuesday', 3:'Wednesday',
                     4:'Thursday', 5:'Friday', 6:'Saturday'}

         dates = [(4,7,1776,'US Declaration of Independence'),
                  (8,5,1945,'VE Day in WWII'),
                  (11,11,1919, 'First Armistice Day')]

         for date in dates:
             day, month, year, name = date
             day_number = day_of_week(day, month, year)
             day_name = weekdays[day_number]
             print("%d/%d/%d %s: %s" % (month, day, year, name, day_name))
```

```
7/4/1776 US Declaration of Independence: Thursday
5/8/1945 VE Day in WWII: Tuesday
11/11/1919 First Armistice Day: Tuesday
```

Experienced Python programmers would shorten the code a bit.

```
In [16]: for day, month, year, name in dates:
             day_name = weekdays[day_of_week(day, month, year)]
             print("%d/%d/%d %s: %s" % (month, day, year, name, day_name))
```

```
7/4/1776 US Declaration of Independence: Thursday
5/8/1945 VE Day in WWII: Tuesday
11/11/1919 First Armistice Day: Tuesday
```


1.5 A Quick Tour of Python

Computing the dates above shows the four most important data structures in Python: lists, dictionarys, tuples, and strings. We discuss each of these in more detail.

1.5.1 Lists

The *list* is an ordered collection of things. Lists are created with `[]`. The list below contains integers, floats, strings, and a list. `len(l)` returns the length of `l`, i.e., the number of items in the list. Note, the list below has four items even though the last item is itself a list with two items.

```
In [17]: l = [0,1.5,'two',[3,'three']]
         len(l)

Out[17]: 4
```

In python, indices start with 0. The first item is `l[0]` and the last is `l[3]`. Python also has the special syntax allowing us to index backwards from the last index to the first. The last index can be accessed as `l[-1]`, the next to last as `l[-2]`, etc.

```
In [18]: l[0], l[1], l[3], l[-1], l[-2]

Out[18]: (0, 1.5, [3, 'three'], [3, 'three'], 'two')
```

Python lists can be *sliced*, e.g., `l[first:last]` returns `l[first], l[first+1], ..., l[last-1]`. Note, `l[last]` is not part of the slice. Negative indices count backward from the last element.

```
In [19]: l[:2], l[2:4], l[:-1], l[1:-2], l[-4:4]

Out[19]: ([0, 1.5],
          ['two', [3, 'three']],
          [0, 1.5, 'two'],
          [1.5],
          [0, 1.5, 'two', [3, 'three']])
```

We can concatenate lists. We can append items to lists. We can sort lists. We can reverse lists.

```
In [20]: u = [1,2]+[3,4,5] #concatenate
         u

Out[20]: [1, 2, 3, 4, 5]

In [21]: u.append(9)
         u.append(-6)
         u

Out[21]: [1, 2, 3, 4, 5, 9, -6]

In [22]: u.sort()
         u

Out[22]: [-6, 1, 2, 3, 4, 5, 9]

In [23]: u.reverse()
         u

Out[23]: [9, 5, 4, 3, 2, 1, -6]
```

The slicing notation can be confusing. `l[m:n]` is the subset of items starting with index `m` up to, but not including, index `n`. Empty `m` or `n` are replaced by first and last, respectively. Thus, `l=l[:n]+l[n:]`, i.e., the whole list is the first part concatenated with the second part.

```
In [24]: l[:2]+l[2:]

Out[24]: [0, 1.5, 'two', [3, 'three']]
```

1.5.2 For Loops

The `for` loop iterates over the elements of a list, tuple, string, or other iterable object.

```
In [25]: for x in l:
             print(x)
```

```
0
1.5
two
[3, 'three']
```

Use the `zip` function to simultaneously iterate over two or more lists.

```
In [26]: l1 = ['a', 'b', 'c']
         l2 = [1, 2, 3]
         for x,y in zip(l1,l2):
             print(x,y)
```

```
a 1
b 2
c 3
```

A common programming idiom is to loop over the integers. The special iterator `range` is handy for this purpose. It has three behaviors:

- `range(n)` acts like `[0, 1, ..., n-1]`
- `range(m,n)` acts like `[m, m+1, ..., n-1]`
- `range(m,n,s)` acts like `[m, m+s, m+2s, ..., m+ks]` where the last element is less than `n`

(In early versions of Python, `range(n) = [0, 1, ..., n-1]`, i.e., it equalled the actual list; now, `range` does not create the whole sequence at once, but rather returns the next element on each iteration. For large `n` the new behavior is faster.)

It is easy to get the sequence as a list, if that is desired.

```
In [27]: print(list(range(4)))
         print(list(range(1,8,2)))
         print(list(range(4,-2,-1)))
```

```
[0, 1, 2, 3]
[1, 3, 5, 7]
[4, 3, 2, 1, 0, -1]
```

A frequent paradigm is to loop over consecutive integers and do something with each one. For instance, here is a loop to create a sequence of squares.

```
In [28]: squares = []
         for i in range(5):
             squares.append(i**2)
         squares
```

```
Out[28]: [0, 1, 4, 9, 16]
```

This sequence occurs so often that Python has a special syntax called *list comprehensions* to make it easier, not just to write but also to read. In this example, the list comprehension accomplishes in one line what the standard method does in three.

```
In [29]: squares = [i**2 for i in range(5)]
         squares
```

```
Out[29]: [0, 1, 4, 9, 16]
```

Use the `enumerate` function to iterate over a list and also obtain the integer indices.

```
In [30]: for i,x in enumerate(l1):
             print(i,x)
```

```
0 a
1 b
2 c
```

```
In [31]: for i, x1 in enumerate(zip(l1,l2)):
             print(i,x1)
```

```
0 ('a', 1)
1 ('b', 2)
2 ('c', 3)
```


1.5.3 Tuples

Tuples are like lists but they are *immutable* (cannot be changed). Tuples are created with `()`.

```
In [32]: date = (4,7,1776,'US Declaration of Independence')
         date[0], date[2:4], date[-2]
```

```
Out[32]: (4, (1776, 'US Declaration of Independence'), 1776)
```

Tuples can be unwrapped into the individual elements.

```
In [33]: day, month, year, name = date
         print("%s: %d/%d/%d" % (name,month,day,year))
```

```
US Declaration of Independence: 7/4/1776
```

The list of dates above could have been created as a list of lists `[[], [], []]`, but the list of tuples `[(), (), ()]` is easier to read and conveys to reader that the dates will not change.

A relatively recent addition to Python is the *namedtuple*. *Namedtuples* allow the programmer to assign names to the values in a tuple and help make the code self-documenting. The example above may be written as follows with *namedtuples*.

```
In [34]: from collections import namedtuple
         Date = namedtuple('Date', ['day', 'month', 'year', 'name'])
         dates = [ Date(4, 7, 1776, 'US Declaration of Independence'),
                 Date(8, 5, 1945, 'VE Day in WWII'),
                 Date(11, 11, 1919, 'First Armistice Day') ]
         for date in dates:
             day_number = day_of_week(date.day, date.month, date.year)
             name_of_day = weekdays[day_number]
             print("%d/%d/%d %s: %s" % (date.month, date.day,
                                        date.year, date.name, name_of_day))
```

```
7/4/1776 US Declaration of Independence: Thursday
5/8/1945 VE Day in WWII: Tuesday
11/11/1919 First Armistice Day: Tuesday
```

We can use keyword arguments in creating a namedtuple. Also, the *namedtuple* prints well.

```
In [35]: date = Date(day=29, month=2, year=2020, name='Next Leap Day')
         print(date)
```

```
Date(day=29, month=2, year=2020, name='Next Leap Day')
```

1.5.4 Dicts

Dictionaries (dicts) are created with `{}`. Dicts consist of a set of `label:value` pairs. Dicts behave like small databases within the program. The labels can be numbers, strings, or tuples; labels cannot be lists.

```
In [36]: d = {0:'Sunday', 1:'Monday', -1:'Saturday', 'night':'day'}
         d

Out[36]: {0: 'Sunday', 1: 'Monday', -1: 'Saturday', 'night': 'day'}

In [37]: d[0], d[-1], d['night']

Out[37]: ('Sunday', 'Saturday', 'day')

In [38]: #create some new values
         d['new'] = 'old'
         d['Old'] = 'New'
         d
```

```
Out[38]: {0: 'Sunday',
          1: 'Monday',
          -1: 'Saturday',
          'night': 'day',
          'new': 'old',
          'Old': 'New'}
```

We can change the dict values as well.

```
In [39]: d[0] = 'Domingo'
         d
```

```
Out[39]: {0: 'Domingo',
          1: 'Monday',
          -1: 'Saturday',
          'night': 'day',
          'new': 'old',
          'Old': 'New'}
```

There are several ways to iterate over the items in a dict. Note, dicts have no quaranteed order, i.e., the `for` loop might produce the items in a different order than how the items were added. Here is one common way:

```
In [40]: for label,value in d.items():
             print(label, value)
```

```
0 Domingo
1 Monday
-1 Saturday
night day
new old
Old New
```


1.5.5 Strings

Strings are sequences of letters, numbers, and various symbols. Python has several ways to create strings. Below we show some of these:

```
In [41]: s = 'single quotes'
         t = "double quote's with apostrophe"
         u = """multiline
              string
              with three lines"""

         print(s)
         print(t)
         print(u)
```

```
single quotes
double quote's with apostrophe
multiline
     string
     with three lines
```

Strings can be concatenated to build up bigger strings.

```
In [42]: s + ' ' + t
```

```
Out[42]: "single quotes double quote's with apostrophe"
```

```
In [43]: s += ' xyzzy'
         s
```

```
Out[43]: 'single quotes xyzzy'
```

Raw strings are handy when the string contains special characters. Raw strings look like `r"string"`. We often use raw strings when annotating plots.

```
In [44]: print('first line\nsecond line') #normal string
```

```
first line
second line
```

```
In [45]: print(r'first line\nsecond line') #raw string
```

```
first line\nsecond line
```

```
In [46]: #loop over characters in string using list comprehension
         [c for c in "it's"]
```

```
Out[46]: ['i', 't', "'", 's']
```


1.5.6 If Statements

Python `if` statements have the form:

```
if condition:
    statements
elif condition:
    statements
else:
    statements
```

There can be any number of `elif` parts (including none). The `else` part is also optional. The condition is a Python expression that evaluates to True (non-zero) or False (zero).

```
In [47]: x = 11
         if x % 2:
             print(True)
         else:
             print(False)
```

```
True
```

```
In [48]: x = 3
         if x == 0:
             print('Zero')
         elif x == 1:
             print('One')
         else:
             print('Infinity')
```

```
Infinity
```

1.5.7 Functions

Python functions have the form:

```
def function_name(arguments):
   """optional doc string"""
   statements
   optional return statement
```

E.g., the `zeller` function takes arguments `k, m, D, C` and returns `f`. Python is flexible in how functions are called. The variations below all work.

```
In [49]: dw = day_of_week #shorter name
         dw(1,1,2018), dw(day=1, month=1, year=2018), dw(year=2018, day=1, month=1)
```

```
Out[49]: (1, 1, 1)
```

While the doc string is optional, it is good practice to include one in your functions. It makes your functions more readable and Python automatically incorporates your function into its help function. E.g., `help(day_of_week)` or, with a bit less typing, simply `day_of_week?`.

```
In [50]: help(day_of_week)
```

```
Help on function day_of_week in module __main__:

day_of_week(day, month, year)
    calculate day of week using Zeller's Rule

    day = day of month, 1-31
```

```
month = month of year, January = 1, Feb = 2, etc
year = four digit year
```

1.5.8 Methods and Attributes

Python is an object oriented language. Many objects (things) have *methods* and *attributes*. Methods are functions that are performed by the object. Some methods compute outputs, some methods change the object, some do both. Attributes are object properties that be set and observed. Here are a few list methods:

- `list.append(x)`: add `x` to the end of the list
- `list.pop()`: remove the last item from the list and return it.
- `list.reverse()`: reverse the order of the elements in the list

Since they cannot be changed, tuples have only two methods:

- `tuple.count(x)`: returns the number of items in the tuple equalling `x`
- `tuple.index(x)`: returns the index of `x` in the tuple

The simplest way to learn what methods or attributes are available on an object is to type the object name followed by `.` and then hit Tab. Try it on a string. Strings have dozens of methods.

1.6 Miscellaneous Python Programming Comments

Python is an easy to read language. It is an important reason why Python is growing in popularity. Good Python programs are easy to read.

For beginners,

- Use the `print` statement to aid in debugging. Print out intermediate variables, see what is happening.
- Start small, get something working, then add features and complexity.
- Test your code on simple examples for which you know the correct answer. E.g., that January 1, 2018 is a Monday can be checked with an ordinary calendar.
- Make your code easy to read. Use meaningful variable names. Add comments when they aid readability.

To get help on a function, type the function name followed by a `?` and hit Shift-Return. To find out what methods and attributes are available for an object, type the object name followed by a `.` and hit the Tab key.

A confusing point for beginners in python is the difference between creating an object and accessing parts of the object. Lists are created inside `[]`, tuples inside `()`, and dicts inside `{}`. However, to access an item in a list, tuple, or dict use `x[k]` where `x` is the object and `k` is the index (or label). The notation `x(k)` is a function call.

The Python interpreter outputs a *Traceback* on errors. Examining the Traceback can help determine the problem (pay particular attention to the first and last parts of the Traceback).

The reason Python has both lists and tuples is that a tuple can be used as a label in a dict, but a list cannot. The first cell below works, but the second results in an error and a Traceback. This one is short enough to easily identify what is the cause of the error. It says "unhashable type: 'list' ". In Python-speak, that means since lists can change, lists cannot be the labels of a dict.

```
In [51]: t = (1,2)  #tuple
         d[t] = 'good' #can use tuple as label in dict

In [52]: l = [1,2]  #list
         d[l] = 'bad' #cannot use list as label in dict

        ---------------------------------------------------------------------------

        TypeError                                 Traceback (most recent call last)

        <ipython-input-52-b7b7f64ddf1f> in <module>()
          1 l = [1,2]  #list
    ----> 2 d[l] = 'bad' #cannot use list as label in dict


        TypeError: unhashable type: 'list'
```

1.7 Summary

Python is an increasingly popular language. It is general purpose and is extended by libraries. The main libraries for signal processing are `numpy`, `matplotlib`, `scipy`, and `pandas`. These are discussed in following chapters.

2

Numpy

Numpy is the library that gives Python vectors and matrices. In numpy parlance, vectors and matrices are *arrays*. Numpy has many functions for manipulating arrays. These functions are written in a compiled language (C, C++, or FORTRAN) for speed. We write our code in Python, but get access to compiled function's speed.

Numpy is the third generation numerical package for Python. The first package was *numeric*. It worked well for small arrays, but was slow for large arrays. Members of the astronomy community developed *numarray* to work with the huge image and data arrays they generate. *Numpy* was created to combine *numeric's* ease of use and *numarray's* speed.

This is an advantage of Python's design: the core language is small and is extended by libraries. The libraries can be rewritten if needed. In contrast, in languages like Matlab which have vector and matrix commands built in it is difficult (if not impossible) to fix any design mistakes that may be discovered later.

To use *numpy* (or any other library) we must import it. The preferred way to do this is below. This command imports the numpy library and introduces the alias *np* (shorter and easier to type then *numpy*).

```
In [1]: import numpy as np
        import matplotlib.pyplot as plt
        np.set_printoptions(precision=3)
        %matplotlib inline
```

2.1 Numpy Arrays

The core numpy data structure is the *array*. Arrays are composed of a number of elements of the same type. Unlike lists, which may hold many different types of

elements, arrays hold the same type of element. It is this sameness that allows fast compiled routines to process arrays.

The simplest way to create an array is convert a list to an array.

```
In [2]: l = [i**2 for i in range(6)] #list comprehension
        a = np.array(l)
        a

Out[2]: array([ 0,  1,  4,  9, 16, 25])
```

Arrays have many *methods* and *attributes* that can modify their behavior. This array is of type `int64`, meaning each element is a 64 bit wide integer. The array is one-dimensional of size 6.

```
In [3]: a.dtype, a.shape, a.size, a.ndim

Out[3]: (dtype('int64'), (6,), 6, 1)
```

Two important methods are `astype()` (converts from one type to another) and `reshape()` (changes shape of array without making a new copy).

```
In [4]: b = a.astype('float') #creates copy because dtype changed
        b, b.dtype

Out[4]: (array([  0.,   1.,   4.,   9.,  16.,  25.]), dtype('float64'))

In [5]: c = b.reshape(3,2)  #not a copy (discussed below)
        c

Out[5]: array([[  0.,   1.],
               [  4.,   9.],
               [ 16.,  25.]])

In [6]: c.shape

Out[6]: (3, 2)
```

Numpy arrays can be accessed similarly to lists. Individual elements can be accessed and changed. Slices can be used.

```
In [7]: a[0], a[1], a[-1] #first, second, last

Out[7]: (0, 1, 25)

In [8]: a[0] = -1
        a

Out[8]: array([-1,  1,  4,  9, 16, 25])

In [9]: b[0], b[1], b[1:3]

Out[9]: (0.0, 1.0, array([ 1.,  4.]))
```

Elements of two dimensional arrays can be accessed individually and through slices.

```
In [10]: c[0,0], c[0,1], c[1,0] #individual elements
```

```
Out[10]: (0.0, 1.0, 4.0)

In [11]: c[0], c[1], c[2] #rows

Out[11]: (array([ 0.,  1.]), array([ 4.,  9.]), array([ 16.,  25.]))
```

To access the columns we use a null slice in the first dimension. Each : matches all indices in that dimension.

```
In [12]: c[:,0], c[:,1] #columns

Out[12]: (array([  0.,   4.,  16.]), array([  1.,   9.,  25.]))
```

We can also use the null slice to access rows, if desired.

```
In [13]: c[0], c[0,:] #same rows

Out[13]: (array([ 0.,  1.]), array([ 0.,  1.]))

In [14]: c[:,0] = np.array([-1,-2,-3]) #assignment to column
         c

Out[14]: array([[ -1.,   1.],
                [ -2.,   9.],
                [ -3.,  25.]])
```

Above when we set `c = b.reshape(3,2)` numpy created a pointer called `c` to the same data elements as `b` (`c` also has some additional values for dimensions and data type). When we changed `c`, we also changed `b`. In other words, `b` is now different from what it was before.

In "numpy-speak", `c = b.reshape(3,2)` is a new *view* of the data. `b` sees the data as a one-dimensional array, while `c` sees the same data in two dimensions.

```
In [15]: b  #different view of same data as c

Out[15]: array([ -1.,   1.,  -2.,   9.,  -3.,  25.])
```

This is a major difference between numpy and Matlab. In Matlab, `c=b` creates a copy of `b` and assigns the copy to `c`; in numpy, `c=b` creates a pointer `c` to the same data elements as `b`. (Minimizing copies was one of the design goals of numarray and was incorporated into numpy. Matlab tries to avoid making copies, but with less success than numpy.)

To create a copy, use the `np.array` function.

```
In [16]: d = np.array(c) #creates a copy
         d

Out[16]: array([[ -1.,   1.],
                [ -2.,   9.],
                [ -3.,  25.]])

In [17]: c[0,0] = -6
         c
```

```
Out[17]: array([[ -6.,   1.],
                [ -2.,   9.],
                [ -3.,  25.]])

In [18]: d   #d != c

Out[18]: array([[ -1.,   1.],
                [ -2.,   9.],
                [ -3.,  25.]])
```

2.2 Larger Arrays

While one and two dimensional arrays are the most common, numpy allows arrays with up to 32 dimensions. Below we create a three dimensional array.

```
In [19]: a = np.arange(24) #like range, but returns array
         b = a.reshape(2,3,4)
         b

Out[19]: array([[[ 0,  1,  2,  3],
                 [ 4,  5,  6,  7],
                 [ 8,  9, 10, 11]],

                [[12, 13, 14, 15],
                 [16, 17, 18, 19],
                 [20, 21, 22, 23]]])
```

By default, numpy orders elements in an array across columns, then rows. This is called row-major order and is the default for C programs. The alternative order, called column-major order, is the default in FORTRAN programs. Numpy can use column-major ordering with appropriate options, but that is beyond this text. Consult the numpy documentation to learn how to efficiently manipulate arrays in FORTRAN order.

```
In [20]: n0, n1, n2 = b.shape
         for i in range(n0):
             for j in range(n1):
                 for k in range(n2):
                     print(b[i,j,k], end=' ')

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
```

2.3 Other Array Creating Functions

We have seen how to create arrays with `np.array` and `np.arange`. Numpy has many other functions for creating arrays. Most array creating functions accept the optional command `dtype = 'int'` or `dtype = 'float'` to specify the type of the array. (Other `dtype's` can be found in the numpy documentation.)

```
In [21]: a = np.arange(5)
         b = np.arange(5,dtype='float')
         a, b
Out[21]: (array([0, 1, 2, 3, 4]), array([ 0.,  1.,  2.,  3.,  4.]))
In [22]: threes = np.arange(0,15,3) #start, stop, step
         threes
Out[22]: array([ 0,  3,  6,  9, 12])
In [23]: x = np.ones((3,2))
         x
Out[23]: array([[ 1.,  1.],
                [ 1.,  1.],
                [ 1.,  1.]])
In [24]: z = np.zeros((2,4))
         z
Out[24]: array([[ 0.,  0.,  0.,  0.],
                [ 0.,  0.,  0.,  0.]])
```

Sometimes we need to create all zero or all one arrays of the same type and shape as another array. Numpy provides two convenience functions to do this, `np.ones_like(x)` and `np.zeros_like(x)`. Using these makes your code simpler and easier to read.

```
In [25]: a, np.ones_like(a), b, np.zeros_like(b)
Out[25]: (array([0, 1, 2, 3, 4]),
          array([1, 1, 1, 1, 1]),
          array([ 0.,  1.,  2.,  3.,  4.]),
          array([ 0.,  0.,  0.,  0.,  0.]))
In [26]: np.eye(4) #identity matrix
Out[26]: array([[ 1.,  0.,  0.,  0.],
                [ 0.,  1.,  0.,  0.],
                [ 0.,  0.,  1.,  0.],
                [ 0.,  0.,  0.,  1.]])
```

In signal processing, we often need equally spaced points spanning a range. The function `np.linspace(start, stop, num)` creates `num` equally spaced points starting at `start` and ending with `stop` (unlike `np.arange` the final value is included in `np.linspace`).

`np.logspace` creates equally spaced points on a logarithmic scale.

```
In [27]: np.linspace(0,1,11)
Out[27]: array([ 0. ,  0.1,  0.2,  0.3,  0.4,  0.5,  0.6,  0.7,  0.8,
                 0.9,  1. ])
In [28]: np.logspace(0,1,11) #from 10**0 to 10**1
Out[28]: array([  1.   ,   1.259,   1.585,   1.995,   2.512,   3.162,
                  3.981,   5.012,   6.31 ,   7.943,  10.   ])
```

2.4 Array Convenience Functions

Numpy includes two convenience functions for creating more complicated arrays, `np.r_[]` and `np.c_[]`. These combine the abilities of `np.arange` and `np.linspace` with the ability to concantenate smaller arrays to make larger ones.

In the examples below, note the peculiar syntax (using `[]`, not `()`) and the use of slice notation.

```
In [29]: a = np.r_[:6] #create 1d array
         a6a = np.r_[a,6,a] #concatenate arrays together along row
         print('a =  ',a)
         print('a6a = ',a6a)
```

```
a =   [0 1 2 3 4 5]
a6a =  [0 1 2 3 4 5 6 0 1 2 3 4 5]
```

```
In [30]: np.r_[a,6,a[::-1]]
```

```
Out[30]: array([0, 1, 2, 3, 4, 5, 6, 5, 4, 3, 2, 1, 0])
```

`np.c_[]` creates a two dimensional array. Note the "slice", `:4.`, using a floating point number, to create an array of floats.

```
In [31]: b = np.c_[:4.] #create 2d float array (column vector)
         bb = np.c_[b,b]
         print('b =  ', b)
         print('bb = ', bb)
```

```
b =   [[ 0.]
 [ 1.]
 [ 2.]
 [ 3.]]
bb =  [[ 0.  0.]
 [ 1.  1.]
 [ 2.  2.]
 [ 3.  3.]]
```

We can combine array creators to make complicated arrays.

```
In [32]: A = np.c_[np.r_[np.eye(3), np.ones((1,3))],
                   np.r_[2*np.ones(3),-1]]
         A
```

```
Out[32]: array([[ 1.,  0.,  0.,  2.],
                [ 0.,  1.,  0.,  2.],
                [ 0.,  0.,  1.,  2.],
                [ 1.,  1.,  1., -1.]])
```

Alternately, we can build the array piece by piece, using slicing notation.

```
In [33]: A = np.zeros((4,4))
         A[:3,:3] = np.eye(3)
         A[:3,-1] = 2*np.ones(3)
         A[-1,:3] = np.ones(3)
         A[-1,-1] = -1
         A

Out[33]: array([[ 1.,  0.,  0.,  2.],
                [ 0.,  1.,  0.,  2.],
                [ 0.,  0.,  1.,  2.],
                [ 1.,  1.,  1., -1.]])
```

To mimic `np.linspace` use the following. Note the `6j` as the "step"; the complex number is interpreted as the number of points.

```
In [34]: np.r_[0:1:6j] #6 values from 0 to 1 inclusive

Out[34]: array([ 0. ,  0.2,  0.4,  0.6,  0.8,  1. ])
```

2.5 Basic Operations

Basic operations, such as addition and multiplication, operate element by element. Simple methods like min, max, mean, var (variance), and sum operate on all elements.

```
In [35]: a = np.ones(4)
         b = np.arange(4)
         a, b

Out[35]: (array([ 1.,  1.,  1.,  1.]), array([0, 1, 2, 3]))

In [36]: a+b, a-b, 2*a, a+2, b**2, b*b*b

Out[36]: (array([ 1.,  2.,  3.,  4.]),
          array([ 1.,  0., -1., -2.]),
          array([ 2.,  2.,  2.,  2.]),
          array([ 3.,  3.,  3.,  3.]),
          array([0, 1, 4, 9]),
          array([ 0,  1,  8, 27]))

In [37]: b.min(), b.max(), b.mean(), b.var(), b.sum()

Out[37]: (0, 3, 1.5, 1.25, 6)
```

2.6 The Dot Product

Linear algebra calculations make frequent use of the *dot* product,

$$z = \sum_{k=0}^{n-1} x_k y_k$$

The `np.dot` function does this computation. Python 3.6 added the convenient notation, `x @ y`.

```
In [38]: x = np.ones(5)
         y = np.ones(5)
         np.dot(x,y), x @ y #same

Out[38]: (5.0, 5.0)

In [39]: A = np.array([[1,0,0],[1,1,0],[1,1,1]])
         b = np.array([1,2,3])
         A, b

Out[39]: (array([[1, 0, 0],
                 [1, 1, 0],
                 [1, 1, 1]]), array([1, 2, 3]))

In [40]: np.dot(A,b), A @ b #same

Out[40]: (array([1, 3, 6]), array([1, 3, 6]))

In [41]: b @ A

Out[41]: array([6, 5, 3])
```

Note the two answers above. Numpy interprets `b` as a column vector when computing `A @ b` and as a row vector when computing `b @ A`.

We can transpose arrays (flip them along the main diagonal).

```
In [42]: A.T

Out[42]: array([[1, 1, 1],
                [0, 1, 1],
                [0, 0, 1]])

In [43]: A.T @ b

Out[43]: array([6, 5, 3])
```

To compute the outer product of two arrays, use the `np.outer` function.

```
In [44]: np.outer(b,b)

Out[44]: array([[1, 2, 3],
                [2, 4, 6],
                [3, 6, 9]])
```

2.7 Cumsum and Cumprod

Cumsum and *cumprod* operate on arrays and return arrays. *cumsum(x)* produces the "cumulative sum" of the elements, `np.array([x[0], x[0]+x[1], x[0]+x[1]+x[2], ...])`. *cumprod* produces the "cumulative product" of the elements.

```
In [45]: integers = np.arange(1,8)
         sums = np.cumsum(integers)
         answer = integers*(integers+1)//2 #1+2+...+n=n(n+1)/2
         print(sums)
         print(answer)
```

```
[ 1  3  6 10 15 21 28]
[ 1  3  6 10 15 21 28]
```

```
In [46]: integers = np.arange(1,8)
         factorials = np.cumprod(integers)
         print(factorials)
```

```
[   1    2    6   24  120  720 5040]
```

2.8 Numpy Ufuncs

Numpy *ufuncs* are functions that operate element by element on numpy arrays. There are many ufuncs. Here we demonstrate a few of the more common ones.

A great many special functions can be found in `scipy.special`. Look there for Bessel functions, Legendre functions, and many others. Statistical functions can be found in `scipy.stats`.

```
In [47]: x = np.arange(1,5)
         x, np.exp(x), np.log(x), np.sqrt(x)
```

```
Out[47]: (array([1, 2, 3, 4]),
          array([  2.718,   7.389,  20.086,  54.598]),
          array([ 0.   ,  0.693,  1.099,  1.386]),
          array([ 1.   ,  1.414,  1.732,  2.   ]))
```

```
In [48]: np.modf(2.3*x) #fractional and integer parts
```

```
Out[48]: (array([ 0.3,  0.6,  0.9,  0.2]), array([ 2.,  4.,  6.,  9.]))
```

```
In [49]: np.cos(2*np.pi*x/5), np.sin(2*np.pi*x/5) #np.pi = 3.14159...
```

```
Out[49]: (array([ 0.309, -0.809, -0.809,  0.309]),
          array([ 0.951,  0.588, -0.588, -0.951]))
```

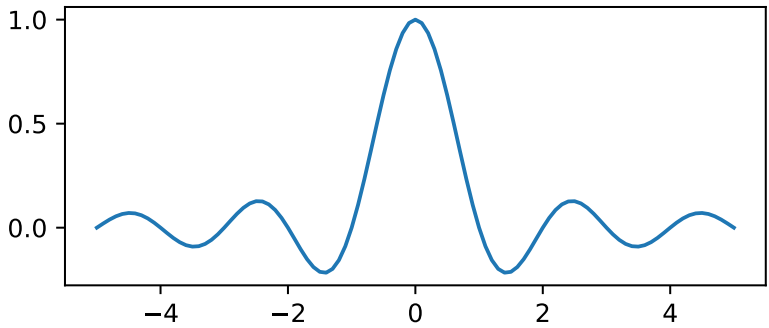
Arrays can be complex as well as real.

```
In [50]: np.exp(1j*x), np.cos(x)+1j*np.sin(x)
```

```
Out[50]: (array([ 0.540+0.841j, -0.416+0.909j, -0.990+0.141j,
                 -0.654-0.757j]),
          array([ 0.540+0.841j, -0.416+0.909j, -0.990+0.141j,
                 -0.654-0.757j]))
```

The `sinc` function is popular in signal processing. It is the Fourier transform of a perfect lowpass filter.

```
In [51]: from scipy.special import sinc
         t = np.linspace(-5,5,101)
         plt.plot(t,sinc(t));
```



2.9 Testing Calculations

Note, testing floating point computations for equality can be problematic. For example, here is a simple example that should be True but isn't. (Example taken from Antti Kaihola, talk at PyCon Finland 2016.)

```
In [52]: 2.2-2 == 1.2-1 #should be True

Out[52]: False

In [53]: print(2.2-2)
         print(1.2-1)

0.20000000000000018
0.19999999999999996
```

We can avoid the rounding problem above by using the `np.allclose(x,y)` function which returns True if all values of `x` are close to the corresponding value of `y`. Below we test Euler's equality.

```
In [54]: np.allclose(2.2-2, 1.2-1)

Out[54]: True

In [55]: #test Euler's formula
         t = np.linspace(0,2*np.pi,101)
         np.allclose(np.exp(1j*t), np.cos(t)+1j*np.sin(t))

Out[55]: True
```

2.10 Call by sharing

Before leaving this chapter, we must discuss a subtlety. Python functions pass variables using "call by sharing". The function can change a passed variable if that variable points to a mutable object; otherwise the function cannot change the passed value.

This is easier to understand by example. The function, `tmp(x)`, tries to change its argument. Watch what happens. Since `6` is immutable (cannot be changed), the first instance does not change `a`. However, lists and arrays are mutable (can be changed) and the variable `a` is changed.

```
In [56]: def tmp(x):
             x *= 2
             return x
         a = 6  #a points to immutable object
         y = tmp(a)
         y, a

Out[56]: (12, 6)

In [57]: a = [6] #2*a = [6,6], i.e., concatenation
         y = tmp(a)
         y, a

Out[57]: ([6, 6], [6, 6])

In [58]: a = np.array([6]) #2*a = array([12])
         y = tmp(a)
         y, a

Out[58]: (array([12]), array([12]))
```

This behavior minimizes unnecessary copies, but it can be confusing and can lead to subtle bugs when the function changes its arguments. If these "side effects" are undesirable, the function should copy its arguments before changing the copy.

2.11 Numpy Comments

Numpy gives Python vector operations. The core numpy data structure is the *array*. Arrays are sequences of floats or ints (and a few other types). Array can have many dimensions, but one and two dimensions are most common. Many array creating functions are available, with `np.array`, `np.arange`, `np.ones`, and `np.zeros` being the most popular.

Numpy differs from Matlab in many ways, but the two biggest "gotchas" are these:

- Numpy arrays start with index 0; Matlab arrays start in index 1.
- In numpy the statement `c=b` creates a pointer `c` to the same data as in `b`; in Matlab the statement `c=b` creates a copy of `b`.

3

Matplotlib for Plotting

Much of research and teaching in signal processing and computational science relies on graphics and visuals. In this chapter, we discuss the standard Python plotting package *Matplotlib*. We use the manipulation and display of *sinusoids* as the primary application.

Matplotlib is designed after the Matlab plotting capabilities. The matplotlib syntax for basic plots should be familiar to Matlab users.

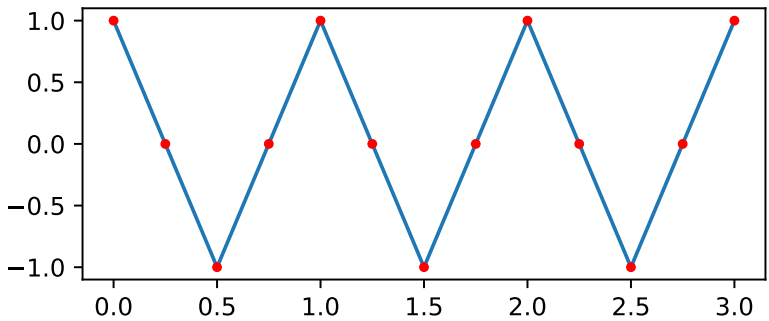
The first line below imports the numpy library, the second line imports the plotting library, and the third line tells the notebook to put the plots within the notebook (the alternative is to pop-up a separate plotting window). Setting `rcParams['figure.figsize']` sets the default size for `matplotlib` plots, in this case, to 5 inches by 2 inches.

```
In [1]: import numpy as np
        import matplotlib.pyplot as plt
        import matplotlib as mpl
        mpl.rcParams['figure.figsize']=(5,2.0)
        %matplotlib inline
```

Start with a plot of a simple sinusoid.

```
In [2]: secs = 3
        points_per_second = 4
        t = np.linspace(0,secs,points_per_second*secs+1)
        y = np.cos(2*np.pi*t)
        plt.plot(t,y)
        plt.plot(t,y,'r.')
```

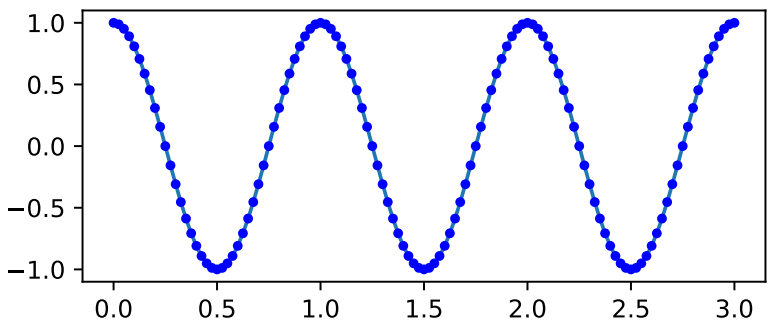
```
Out[2]: [<matplotlib.lines.Line2D at 0x1090f9c88>]
```



That plot is a poor representation of a sinusoid. The reason is we did not use enough points. The plotting routine "connects the dots" with straight line segments. We see this by plotting both lines and points. We get a better representation of a sinusoid by plotting more points per period.

Also, notice the notebook printed `[<matplotlib.lines.Line2D at ...>]`. The notebook prints the output of the last line. If we do not want this behavior, we have two options. One, rewrite the last line as `_ = plt.plot(t,y)` (since assignment statements do not generate any output). The simple variable `_` is used by Python programmers to designate a value that is not going to be used anywhere else. Two, end the command with a semicolon, i.e., `plt.plot(t,y);`. The notebook does not print the output of the final line if the line ends in a semicolon.

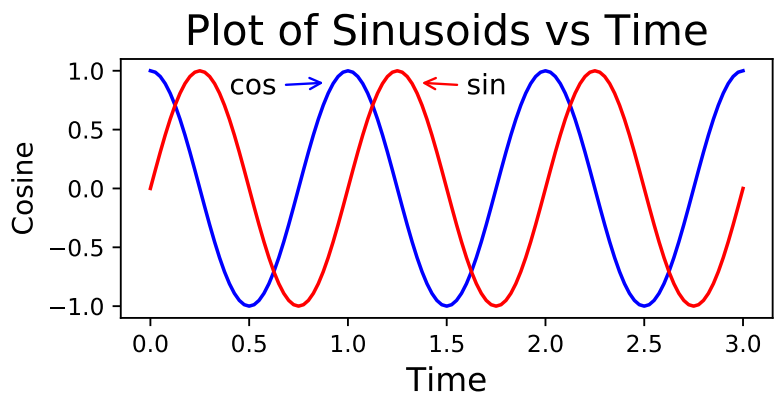

```
In [3]: secs = 3
        points_per_second = 40
        t = np.linspace(0,secs,points_per_second*secs+1)
        y = np.cos(2*np.pi*t)
        plt.plot(t,y) #smooth curve
        plt.plot(t,y,'b.'); #dots
```



We see our first plotting lesson: to get smooth (continuous) curves, use many closely spaced points. We might describe the above plot as a continuous function, but the underlying code is discrete.

Let us improve the appearance of plot by adding labels, annotations, and a title.

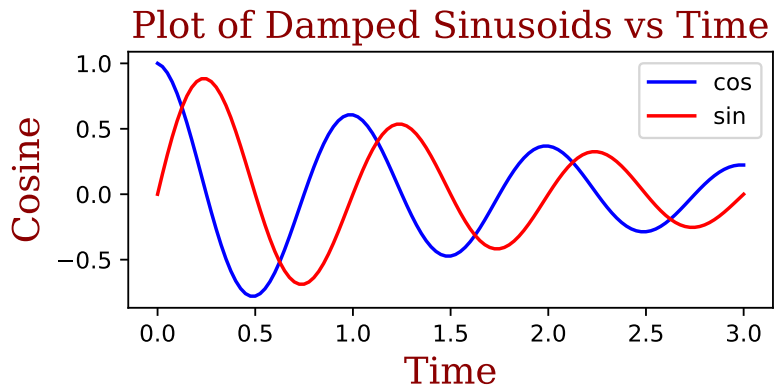
```
In [4]: secs = 3
        t = np.linspace(0,secs,40*secs+1)
        yc = np.cos(2*np.pi*t)
        ys = np.sin(2*np.pi*t)
        plt.plot(t,yc,'b',label='cos')
        plt.plot(t,ys,'r',label='sin')
        plt.xlabel('Time', fontsize=14) #make the font larger
        plt.ylabel('Cosine', fontsize=12)
        plt.title('Plot of Sinusoids vs Time',fontsize = 18)
        plt.annotate('cos',fontsize=12,xy=(0.9,0.9),xytext=(0.4,0.8),
                     arrowprops=dict(color='blue',arrowstyle="->"))
        plt.annotate('sin',fontsize=12,xy=(1.35,0.9),xytext=(1.6,0.8),
                     arrowprops=dict(color='red',arrowstyle="->"));
```



Below we encapsulate the plotting commands in a function. The function takes an optional dictionary argument `font`. We use this argument to change the format of the labels and the title.

```
In [5]: def plot_damped_sinusoids(secs=3, damp = 1, font={}):
            """plot cosine and sine"""
            t = np.linspace(0,secs,40*secs+1)
            damping = np.exp(-damp*t)
            yc = damping*np.cos(2*np.pi*t)
```

```
    ys = damping*np.sin(2*np.pi*t)
    plt.plot(t,yc,'b',label='cos')
    plt.plot(t,ys,'r',label='sin')
    plt.xlabel('Time', fontdict = font)
    plt.ylabel('Cosine', fontdict = font)
    plt.title('Plot of Damped Sinusoids vs Time',fontdict = font)
    plt.legend()

font = {'family': 'serif',
        'color':  'darkred',
        'weight': 'normal',
        'size': 16,
        }
plot_damped_sinusoids(secs=3, damp=0.5, font=font)
```



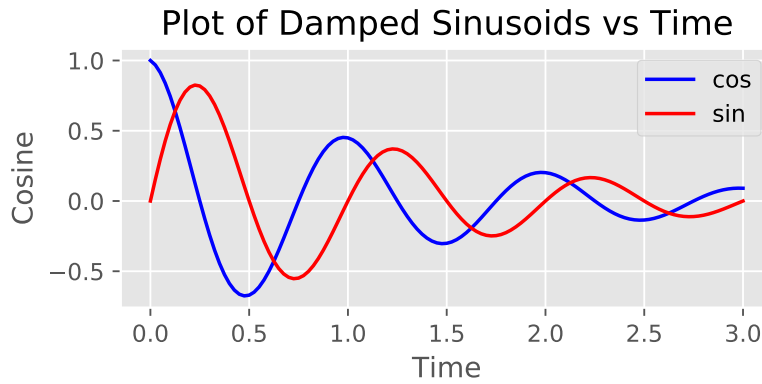
3.1 Changing the Plot Style

Wholesale changes to the plot can be done by using *styles*. Below, we use the `with` command to localize the changes to the commands that immediately follow. The command `plt.style.available` gives a list of available plotting styles. Here are some of them.

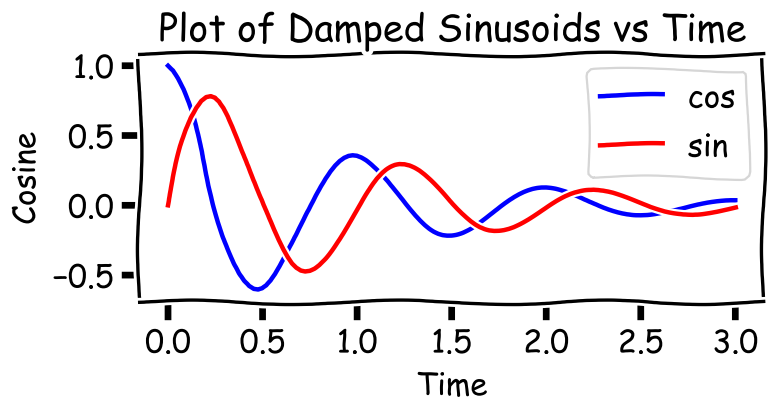
ggplot mimics the look of the R *ggplot* library.
seaborn mimics the look of the Python Seaborn statistical plotting library.
bmh mimics the look of the Bayesian Methods for Hackers online book.
fivethirtyeight mimics the look of the https://fivethirtyeight.com/ website.

Matplotlib also includes the fun style, `xkcd`, mimicing the "XKCD" comic, https://xkcd.com/.

```
In [6]: #ggplot style, replace 'ggplot' with the other styles
        with plt.style.context('ggplot'):
            plot_damped_sinusoids(secs=3, damp=0.8)
```




```
In [7]: #xkcd style, note the different command below
        with plt.xkcd():
            plot_damped_sinusoids(secs=3)
```

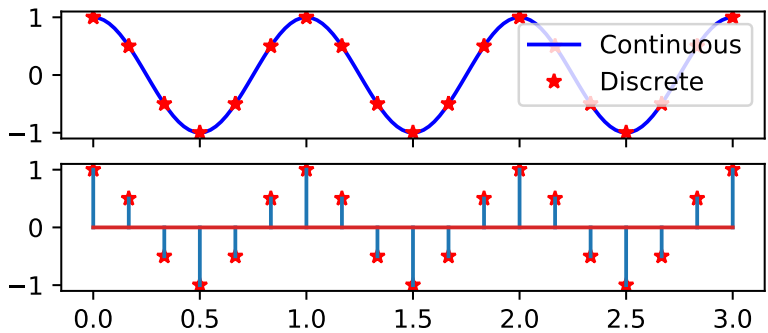


3.2 Discrete Time Sinusoids

Discrete signals can be obtained by sampling continuous signals. The simplest way to do this is have a slower discrete time clock. Let `Fs` be the sampling rate in samples per second.

The `subplot` command displays two or more plots together. In the example below, the first subplot plots the continuous and discrete signals together; the second plots the discrete signal as a stem plot.

```
In [8]: secs = 3
        Fcts = 40 #samples per second for continuous signal
        t = np.linspace(0,secs,secs*Fcts+1)
        Fs = 6 #samples per second
        n = np.linspace(0,secs,secs*Fs+1)
        yt = np.cos(2*np.pi*t)
        yn = np.cos(2*np.pi*n)
        fig, axes = plt.subplots(ncols=1, nrows=2, sharex=True)
        axes[0].plot(t,yt,'b-',label='Continuous')
        axes[0].plot(n,yn,'r*',label='Discrete')
        axes[0].legend(loc=1)
        axes[1].stem(n,yn,markerfmt='r*');
```

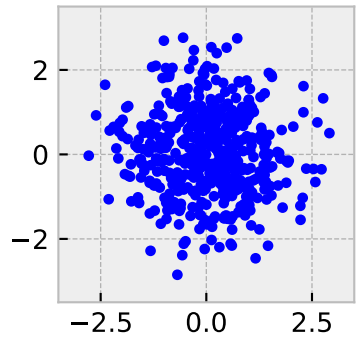


3.3 Scatter Plots

Scatter plots do not connect the dots. Below is a plot of 500 randomly selected points from a two dimensional Gaussian distribution.

```
In [9]: #500 Gaussian points
        n = 500
        x,y = np.random.randn(2*n).reshape(2,n)
```

```
with plt.style.context('bmh'): #bmh style
    plt.plot(x,y,'b.')
    plt.xlim([-3.5,3.5]) #use same size on both axes
    plt.ylim([-3.5,3.5])
    plt.axes().set_aspect('equal') #make plot square
```



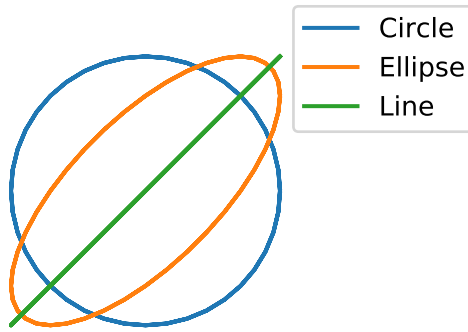
3.4 Parametric Curves and Lissajous Figures

Most plots plot y values versus x values, possibly as individual points or as connected line segments. *Parametric* plots plots y versus x but both x and y depend on a third value t. For each value of t, we plot $y(t)$ versus $x(t)$.

As examples, we consider *Lissajous* figures. In Lissajous figures both $x(t)$ and $y(t)$ are sinusoids in the variable t. Lissajous figures were commonly used in early oscilloscopes to measure the phase change between input and output sinusoids or to verify that the frequency of one sinusoid is an integer multiple of the frequency of a reference sinusoid.

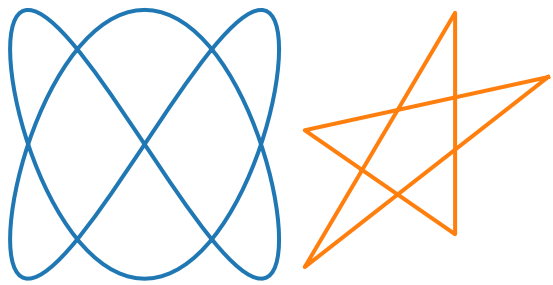
First, plot a circle, an ellipse, and a line. The only differences between the curves are the phase differences between the $x(t)$ and $y(t)$ signals. By looking at the shape of the curve, engineers could determine the phase difference between the two signals.

```
In [10]: x = np.cos(2*np.pi*t)
         plt.plot(x, np.cos(2*np.pi*t+np.pi/2),label='Circle')
         plt.plot(x, np.cos(2*np.pi*t+np.pi/4), label='Ellipse')
         plt.plot(x, np.cos(2*np.pi*t+0), label='Line')
         plt.legend(loc=(1,0.7))
         plt.axes().set_aspect('equal') #make circle round
         plt.axes().axis('off'); #eliminate box
```



Below we show two interesting variations. In the first, the sinusoids have different frequencies; in the second, the points are far apart.

```
In [11]: t = np.linspace(0,1,201)
         plt.figure(figsize=(5,2))
         plt.plot(np.cos(4*np.pi*t),np.cos(6*np.pi*t+np.pi/4))
         tt = 0.4*np.arange(0,6)
         plt.plot(2+np.cos(2*np.pi*tt),np.cos(2*np.pi*tt+np.pi/3))
         plt.axes().set_aspect('equal')
         plt.axes().axis('off');
```



4

Sympy for Symbolic Calculations

Sympy is a python library for symbolic calculations. While it is incomplete (features are added regularly), sympy is adept a routine algebra and calculus. Sympy is handy for producing LaTeX representations of mathematical expressions (for pasting into LaTex documents).

Unlike with `numpy`, here we run `from sympy import *` to import all `sympy` core functions. We also run `init_session()` which executes a number of helpful commands.

```
In [1]: import matplotlib.pyplot as plt
        import numpy as np
        from sympy import * #import all sympy functions
        #init_session() #start sympy session
        %matplotlib inline
```

Define some basic symbols and start fancy printing.

```
In [2]: x, y, z, t = symbols('x y z t')
        k, m, n = symbols('k m n', integer=True)
        f, g, h = symbols('f g h', cls=Function)
        init_printing()
```

4.1 Basic Operations

By default, sympy converts fractions to simplest form. If you want to keep it as a fraction, use the `S()` function.

```
In [3]: 1/2, S(1)/2
```

Out[3]:

$$\left(0.5, \quad \frac{1}{2}\right)$$

Sympy can evaluate functions to arbitrary precision.

```
In [4]: pi.evalf(32), E.evalf(10)
```

Out[4]:

$$(3.1415926535897932384626433832795, \quad 2.718281828)$$

One handy use of sympy is to create a LaTeX expression for insertion into another document. Note the use of "_" to refer to the previous expression (avoids a lot of typing).

```
In [5]: #declare some common Greek letters
        alpha, beta, tau = symbols('alpha beta tau', real=True)
        exp(-alpha*tau)*cos(2*pi*t)/sqrt(beta)
```

Out[5]:

$$\frac{1}{\sqrt{\beta}} e^{-\alpha \tau} \cos\left(2 \pi t\right)$$

```
In [6]: print(latex(_))
```

```
\frac{1}{\sqrt{\beta}} e^{- \alpha \tau} \cos{\left (2 \pi t \right )}
```

Let us demonstrate some simple calculus and polynomial functions. Common functions for manipulating polynomials are `factor` and `expand`. Functions for solving functions are `solve(f,x)` (gives the roots of `f`) and `roots(f)` (gives the roots and the multiplicity of each root).

```
In [7]: f = (x+1)*(x+2)*(x-3)**2
        expand(f), factor(expand(f))
```

Out[7]:

$$\left(x^4 - 3x^3 - 7x^2 + 15x + 18, \quad (x-3)^2 (x+1) (x+2)\right)$$

```
In [8]: solve(f,x), roots(f)
```

Out[8]:

$$([-2, \quad -1, \quad 3], \quad \{-2 : 1, \quad -1 : 1, \quad 3 : 2\})$$

```
In [9]: expand(f.diff(x)), expand(f.diff(x,x)) #first and second derivatives
```

Out[9]:

$$\left(4x^3 - 9x^2 - 14x + 15, \quad 12x^2 - 18x - 14\right)$$

```
In [10]: f.integrate(x) #no constant of integration
```

Out[10]:

$$\frac{x^5}{5} - \frac{3x^4}{4} - \frac{7x^3}{3} + \frac{15x^2}{2} + 18x$$

```
In [11]: #Integral yields unevaluated integral
         expr = Integral(f,(x,-5,5))
         Eq(expr, expr.doit()) #Eq = sympy equality expression
```

Out[11]:

$$\int_{-5}^{5} (x-3)^2 (x+1)(x+2)\, dx = \frac{2540}{3}$$

Evaluate the function at selected points using a list comprehension.

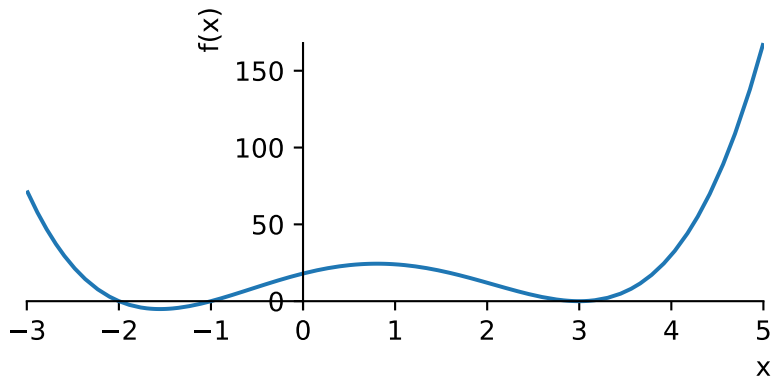
```
In [12]: [f.subs(x,v) for v in range(-3,6)]
```

Out[12]:

$$[72, \quad 0, \quad 0, \quad 18, \quad 24, \quad 12, \quad 0, \quad 30, \quad 168]$$

Sympy includes matplotlib plotting functionality, but some of the syntax is different.

```
In [13]: plot(f,(x,-3,5));
```



In addition to evaluating sympy functions, substitution is used to change variables.

```
In [14]: g = f.subs(x,y**2)
         g, solve(g,y)
```

Out[14]:

$$\left(\left(y^2-3\right)^2\left(y^2+1\right)\left(y^2+2\right),\quad \left[-\sqrt{3},\quad \sqrt{3},\quad -i,\quad i,\quad -\sqrt{2}i,\quad \sqrt{2}i\right]\right)$$

Sympy uses `oo` to represent infinity. The integral calculates the moments of a Gaussian density. The first sum calculates the finite sum of kz^{-k}; the second lets the upper limit go to ∞. It converges when $|1/z| < 1$ or, equivalently, when $|z| > 1$.

```
In [15]: k = symbols("k", integer=True, positive=True)
         gaussian = exp(-x**2/2)/sqrt(2*pi)
         Exk = integrate(x**k*gaussian,(x,-oo,oo))
         [Exk.subs(k,l) for l in range(0,5)]
```

Out[15]:

$$[1,\quad 0,\quad 1,\quad 0,\quad 3]$$

```
In [16]: summation(k*z**-k, (k,0,n))
```

Out[16]:

$$-\frac{z^{-n}}{(z-1)^2}\left(nz-n-zz^n+z\right)$$

```
In [17]: summation(n*z**-n, (n,0,oo))
```

Out[17]:

$$\begin{cases}\frac{1}{z\left(1-\frac{1}{z}\right)^2} & \text{for } \left|\frac{1}{z}\right|<1\\ \sum_{n=0}^{\infty} nz^{-n} & \text{otherwise}\end{cases}$$

4.2 Euler's Formula

Sympy can manipulate complex numbers and expressions. Unlike python and numpy, sympy uses `I` for $\sqrt{-1}$.

```
In [18]: t = symbols("t", real=True)
         exp(I*t), re(exp(I*t)), im(exp(I*t))
```

Out[18]:

$$\left(e^{it},\quad \cos(t),\quad \sin(t)\right)$$

```
In [19]: simplify(exp(I*t) - (cos(t)+I*sin(t)))
```

Out[19]:

$$0$$

4.3 Wilkinson polynomial

The Wilkinson polynomial is an example of the difficulty of computing the roots of a polynomial. The roots are the integers $1, 2, \ldots, n$, but are difficult to compute numerically when n is large. Sympy, however, can compute the roots exactly. Unfortunately, most polynomials of degree $n \geq 5$ cannot be solved exactly.

```
In [20]: n = 5
         wilk_poly = product(x-k,(k,1,n)) #product of terms
         wilk_poly
```

Out[20]:

$$(x-5)\,(x-4)\,(x-3)\,(x-2)\,(x-1)$$

```
In [21]: expand(wilk_poly), solve(wilk_poly)
```

Out[21]:

$$\left(x^5 - 15x^4 + 85x^3 - 225x^2 + 274x - 120,\quad [1,\quad 2,\quad 3,\quad 4,\quad 5]\right)$$

4.4 Quadratic and Cubic Expressions

Most people know the quadratic formula for finding the solutions of a second order polynomial. A few people know the formulas for third and fourth order polynomials. Sympy finds these easily.

```
In [22]: #quadratic polynomial
         a, b, c = symbols('a b c',real=True)
         quadratic = a*x**2 + b*x + c
         solve(quadratic, x)
```

Out[22]:

$$\left[\frac{1}{2a}\left(-b + \sqrt{-4ac + b^2}\right),\quad -\frac{1}{2a}\left(b + \sqrt{-4ac + b^2}\right)\right]$$

The formula for the cubic solutions is simpler if x is replaced by $y - a/3$. Notice the y^2 term is eliminated.

```
In [23]: cubic = x**3 + a*x**2 + b*x + c
         cubic_no_y2 = cubic.subs(x,y-a/3)
         cubic_no_y2.expand().collect(y)
```

Out[23]:

$$\frac{2a^3}{27} - \frac{ab}{3} + c + y^3 + y\left(-\frac{a^2}{3} + b\right)$$

It is common to eliminate the y^2 term, as we do below, but sympy can do the reverse substitution to obtain the solution to the original cubic (if desired).

In [24]:
```
#cubic without quadratic term
simple_cubic = y**3 + alpha*y + beta
roots = solve(simple_cubic, y)
roots[0] #not shown: roots[1] and roots[2]
```

Out[24]:

$$\frac{\alpha}{\sqrt[3]{\frac{27\beta}{2}+\frac{1}{2}\sqrt{108\alpha^3+729\beta^2}}} - \frac{1}{3}\sqrt[3]{\frac{27\beta}{2}+\frac{1}{2}\sqrt{108\alpha^3+729\beta^2}}$$

In [25]:
```
simplify(simple_cubic.subs(y, roots[0]))
```

Out[25]:

$$0$$

4.5 Partial Fraction Expansion

If the denominator of a rational function can be factored, then sympy can perform partial fraction expansion. `apart` takes the rational function apart, but, for complicated expressions, needs to be followed by `doit`.

Partial fraction expansion only works if the denominator polynomial can be factored.

In [26]:
```
rat = (x+1)/(x**2+1)
rat, rat.apart(full=True).doit()
```

Out[26]:

$$\left(\frac{x+1}{x^2+1}, \quad \frac{\frac{1}{2}+\frac{i}{2}}{x+i}+\frac{\frac{1}{2}-\frac{i}{2}}{x-i}\right)$$

4.6 Convolution

As an example of the benefits and pitfalls of symbolic computation consider the convolution of two exponential functions.

$$\begin{aligned} f(t) &= e^{-at}u(t) \\ g(t) &= e^{-bt}u(t) \\ h(t) &= (f*g)(t) = \int_0^t f(\tau)g(t-\tau)d\tau \end{aligned}$$

In [27]:
```
a, b = symbols("a b", real=True, positive=True)
f = exp(-a*t)
g = exp(-b*t)
h = integrate(f.subs(t,tau)*g.subs(t,t-tau), (tau,0,t))
h
```

Out[27]:

$$\begin{cases} te^{-bt} & \text{for } a = b \\ \frac{1}{ae^{bt} - be^{bt}} - \frac{1}{ae^{at} - be^{at}} & \text{otherwise} \end{cases}$$

Notice the answer depends on whether or not $a = b$. The answer can be separated into pieces as follows:

```
In [28]: h.args[0][0], h.args[1][0]
```

Out[28]:

$$\left(te^{-bt}, \quad \frac{1}{ae^{bt} - be^{bt}} - \frac{1}{ae^{at} - be^{at}}\right)$$

The second answer (for $a \neq b$) is not in the form usually presented. While it is possible to manipulate the expression (using some obscure sympy tricks), we can easily test for equality.

```
In [29]: preferred = (exp(-b*t)-exp(-a*t))/(a-b)
         preferred
```

Out[29]:

$$\frac{1}{a-b}\left(e^{-bt} - e^{-at}\right)$$

```
In [30]: simplify(h.args[1][0] - preferred)
```

Out[30]:

$$0$$

Since the answer is 0 the two expressions are equal.

4.7 Trigonometric Simplification

Sympy can simplify trigonometric expressions, but sometimes we want more complex representations. Currently, this aspect of sympy is being developed but is not complete. Nevertheless, sympy can manipulate trigonometic expressions using `expand(trig=True)` and `simplify`.

```
In [31]: sin(alpha+beta).expand(trig=True)
```

Out[31]:

$$\sin(\alpha)\cos(\beta) + \sin(\beta)\cos(\alpha)$$

```
In [32]: simplify(_)
```

Out[32]:

$$\sin(\alpha + \beta)$$

```
In [33]: cos(alpha+beta).expand(trig=True)
```

Out[33]:

$$-\sin(\alpha)\sin(\beta)+\cos(\alpha)\cos(\beta)$$

```
In [34]: cos(2*alpha).expand(trig=True)
```

Out[34]:

$$2\cos^2(\alpha)-1$$

```
In [35]: simplify(cos(alpha)**2 + sin(alpha)**2)
```

Out[35]:

$$1$$

4.8 Summary

While sympy is incomplete, it is still useful for many signal processing operations. Sympy is handy for basic calculus (integration and differentiation) and manipulating polynomials and trigonometric expressions. It can output plots and LaTeX expressions.

Besides the examples here, sympy functions can be converted to numpy functions (for numerical computation).

Some advice: rather than combine numpy and sympy in the same notebook, use two notebooks: one for sympy and one for numpy. Keeping them separate allows one to use the same name, e.g., `x` for the same thing in both sympy and numpy.

5

Sampling and Aliasing

Sampling is the process by which a continuous time signal is converted to a discrete time signal. The reverse process, *reconstruction*, converts a discrete time signal to a continuous time signal.

If the input frequency is too high compared to the sampling frequency, *aliasing* results. Aliasing refers to the phenomenon that the discrete sinusoid corresponds to a different continuous frequency than the original one. Sometimes aliasing is desired, but most often aliasing is a distortion that is best avoided.

The *sampling theorem* says that a continuous signal can be sampled and reconstructed *exactly* if the sampling rate is at least twice the highest frequency present in the signal. The sampling theorem is the theoretical justification for using digital computers for processing analog signals.

Note, an *analog-to-digital* converter includes two processes: sampling and quantization. The reverse, a *digital-to-analog* converter, includes reconstruction and inverse quantization. This discussion is about sampling and aliasing. It is not about reconstruction, quantization, and inverse quantization.

5.1 Sampling a Sinusoid

Sampling a single sinusoid illustrates how sampling works and how aliasing occurs.

We use the following notation:

- F is the frequency of the input sinusoid (Hertz, Hz, cycles per second). If the signal has multiple sinusoids, F_i is the frequency of the i'th sinusoid.
- $\Omega = 2\pi F$ is the continuous frequency in radians per second.
- f is the frequency of the discrete sinusoid in cycles per sample.

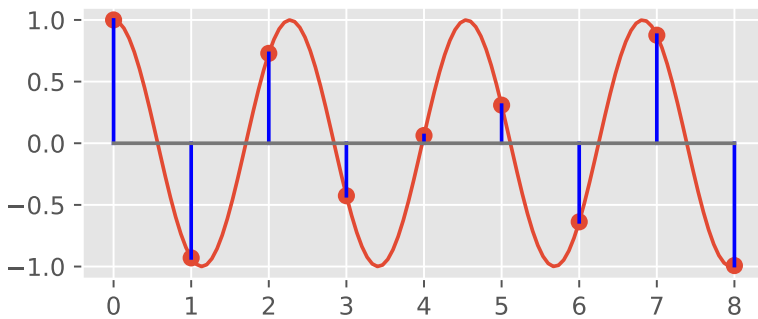
- $\omega = 2\pi f$ is the frequency of the discrete sinusoid in radians per sample.
- F_s is the sampling rate, the number of samples per second.
- $T = 1/Fs$ is the sampling time, in seconds per sample.

```
In [1]: import numpy as np
        import matplotlib.pyplot as plt
        plt.style.use('ggplot')
        from IPython.display import Audio
        import matplotlib
        matplotlib.rcParams['figure.figsize']=(5,2)
        %matplotlib inline
```

5.2 Example: sampling without aliasing

Aliasing is avoided when the sampling rate is at least twice the highest frequency in the input signal, i.e., when $F_s > 2\max(F)$.

```
In [2]: #continuous time sinusoid
        F = 0.44
        tmax = 8
        t = np.linspace(0,tmax,tmax*15+1)
        xt = np.cos(2*np.pi*F*t)
        plt.plot(t,xt);

        #sampled sinusoid
        Fs = 1
        T = 1.0/Fs
        n = np.linspace(0,tmax,tmax+1)
        xn = np.cos(2*np.pi*F*n)
        plt.stem(n,xn,'b');
```



5.3 Example: Sampling with Aliasing

In this example the original signal is at a frequency that is more than twice the sampling frequency. We show a second sinusoid that results in *exactly* the same samples. I.e., two different continuous time sinusoids are indistinguishable from the samples.

Let $x(t) = \cos(2\pi Ft)$ be the original sinusoid and let $x[n] = x(nT) = \cos(2\pi FTn)$ be the sampled discrete time signal. Then, $\omega = 2\pi FT$ and $f = FT$ are the discrete frequencies in radians per sample and cycles per sample, respectively.

Aliasing results from the 2π ambiguity of angles. The angular frequencies $\omega_k = \omega + k2\pi$ are equivalent to ω for all integer values of k, i.e., $\cos(\omega n) = \cos((\omega + k2\pi)n) = \cos(\omega_k n)$ for all integer k and all integer n.

The analog frequency corresponding to ω_k can be found as follows:

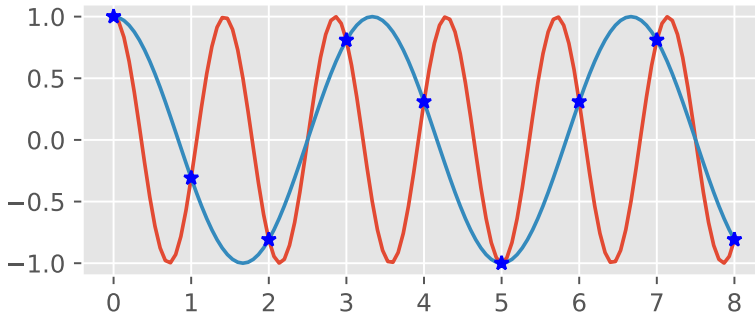
$$\omega_k = 2\pi FT + k2\pi = 2\pi(F + k/T)T = 2\pi(F + kF_s)T$$

The equivalent frequencies are $F + kF_s$ for all integer values k.

When $|F| > F_s/2$, one of the aliased frequencies is closer to 0 than is F. In the example below, $F = 0.7$ and $F_s = 1$. $F_k = 0.7 + kF_s = 0.7 + k$. For $k = -1$, $F_k = -0.3$ which is closer to 0 than is 0.7.

As a general rule, when $|F| > F_s/2$ the discrete signal is more easily interpreted as the aliased signal closer to 0 frequency than it is as the original frequency.

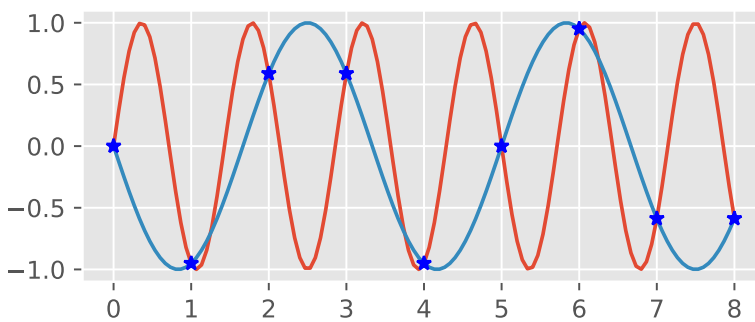
```
In [3]: F = 0.7
        Fk = -0.3
        t = np.linspace(0,tmax,tmax*15+1)
        xt1 = np.cos(2*np.pi*F*t)
        xt2 = np.cos(2*np.pi*Fk*t)
        plt.plot(t,xt1)
        plt.plot(t,xt2)

        Fs = 1
        n = np.linspace(0,tmax,tmax+1)
        xn = np.cos(2*np.pi*F*n)
        plt.plot(n,xn,'b*');
```



A phase change in the original signal complicates the analysis a bit, but does not change the final answer: sinusoids at multiples of F_s result in the same samples. For example, below we demonstrate the same example but with `sin` instead of `cos`. Note, $\sin(-\omega n) = -\sin(\omega n)$.

Notice both sinusoids are shifted, but the samples still occur at the crossing points.

```
In [4]: F = 0.7
        Fk = -0.3
        xt1 = np.sin(2*np.pi*F*t)
        xt2 = np.sin(2*np.pi*Fk*t)
        plt.plot(t,xt1)
        plt.plot(t,xt2)

        Fs = 1
        n = np.linspace(0,tmax,tmax+1)
        xn = np.sin(2*np.pi*F*n)
        plt.plot(n,xn,'b*');
```

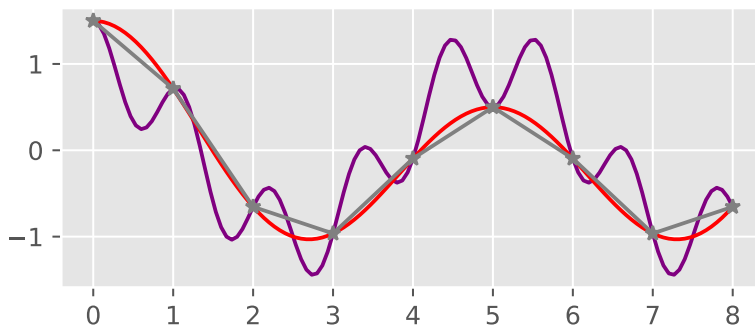


5.4 Sampling with Multiple Sinusoids

In the example below, the original signal is made from two sinusoids, one at frequency $f_1 = 0.2$ and the other at frequency $f_2 = 0.9$. The aliased signal has frequencies 0.2 and 0.1. The discrete signal is plotted by "connecting the dots". (This is known as *linear interpolation*.)

Notice how closely the reconstructed sampled signal matches the aliased signal, not the original signal. The high frequency component of the original signal is lost.

```
In [5]: w1 = 2*np.pi*0.2
        w2 = 2*np.pi*0.9
        w2prime = 2*np.pi*0.1
        xorig = np.cos(w1*t) + 0.5*np.cos(w2*t)
        xaliased = np.cos(w1*t) + 0.5*np.cos(w2prime*t)
        plt.plot(t,xorig, color='purple')
        plt.plot(t,xaliased, color='red')
        xn = np.cos(w1*n) + 0.5*np.cos(w2*n)
        plt.plot(n,xn,'*-',color='gray');
```



5.5 Sampling of Audio Rate Signals

We can hear aliasing. In the first cell below, we create a tone at frequency 7600 Hz and play it with a sampling rate of 16000 Hz. Since the frequency is less than half the sampling rate, there is no aliasing and we hear the tone at its natural frequency.

in the second cell, a tone at frequency 7600 Hz is played with a sampling rate of 8000 Hz. This frequency aliases to $|7600 - 8000| = 400$ Hz. We can hear a clear difference between the two tones, the difference being due to aliasing.

```
In [6]: Fs = 16000
        secs = 3
        f = 7600
        t = np.linspace(0,secs,secs*Fs+1)
        x = np.cos(2*np.pi*f*t)
        Audio(x,rate=Fs)
```

```
Out[6]: <IPython.lib.display.Audio object>
```

```
In [7]: Fs = 8000
        secs = 3
        f = 7600
        t = np.linspace(0,secs,secs*Fs+1)
        x = np.cos(2*np.pi*f*t)
        Audio(x,rate=Fs)
```

```
Out[7]: <IPython.lib.display.Audio object>
```

5.6 Sampling and Aliasing Summary

Sampling refers to the process of converting a continuous time signal to a discrete time signal by measuring the signal at regular points of time. The sampling rate, F_s, is the number of samples per second.

Aliasing occurs when the sampling rate is less than twice the highest frequency present in the signal. The aliased components appear to be at lower frequencies (closer to 0). These aliased components can be heard in audio signals.

The sampling theorem says the analog signal can be reconstructed exactly from its samples if the sampling rate is at least twice the highest frequency present. In practice, a sampling rate at least 10% higher is usually chosen. The sampling theorem is the theoretical justification for using digital computers to analyze analog signals.

Generally, aliasing is undesirable and should be avoided. The simplest way to avoid aliasing to sample fast enough (more than twice the highest frequency). If this is not possible, high frequency components can be filtered out before the signal is sampled. In practice, sampling systems usually put a filter before the sampler.

6

Audio Signals, Beats, and Modulation

Python can generate audio signals. The Jupyter notebook can play those signals. First, a simple trigonometric identity is used to generate *beat* frequencies. The identity is turned around to demonstrate *modulation*.

New libraries imported by this notebook are `Audio` (for putting an audio player in the notebook) and `scipy.signal` (contains a variety of signal processing functions).

```
In [1]: import numpy as np
        import matplotlib.pyplot as plt
        plt.style.use('ggplot')
        from IPython.display import Audio
        import scipy.signal as sig
        %matplotlib inline
```

6.1 Generate an Audio Sinusoid

The standard sampling rate for telephone quality audio is $F_s = 8000$ samples per second. Other sampling rates are used in other applications. For instance, CD quality audio uses 44,100 samples per second. High quality audio recording uses 48,000 or 96,000 samples per second.

The `Audio` command puts an audio widget in the notebook.

```
In [2]: Fs = 8000 #samples per second
        tmax = 3 #duration of signal
        t = np.linspace(0,tmax,tmax*Fs+1)
        f = 440 #note A above middle C on standard scale
```

```
x = np.cos(2*np.pi*f*t)
Audio(x,rate=Fs)
```

```
Out[2]: <IPython.lib.display.Audio object>
```

6.2 Beat Frequencies

Using the standard formula for addition of two cosines,

$$\cos(\alpha) + \cos(\beta) = 2\cos\left(\frac{\alpha+\beta}{2}\right)\cos\left(\frac{\alpha-\beta}{2}\right)$$

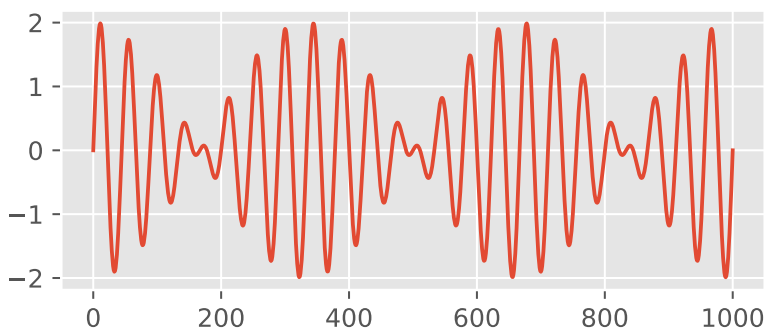
Using $\alpha = \omega_1 t$ and $\beta = \omega_2 t$, we get a formula for two sinusoids

$$\cos(\omega_1 t) + \cos(\omega_2 t) = 2\cos\left(\frac{\omega_1+\omega_2}{2}t\right)\cos\left(\frac{\omega_1-\omega_2}{2}t\right)$$

The *beat* frequency is $\omega_b = \omega_1 - \omega_2$. The left hand side of this equation is the sum of two sinusoids. The right hand side is a sinusoid at the average frequency amplitude modulated by a sinusoid at half the beat frequency. (The reason the beat frequency is $\omega_1 - \omega_1$ and not $(\omega_1 - \omega_2)/2$ is that the amplitude increases twice for each cycle of $\cos((\omega_1 - \omega_2)t/2)$.

Below is a plot of the sum of two sinusoids, one at frequency 7ω and the second at 8ω. The beat frequency is ω.

```
In [3]: omega = np.linspace(0,6*np.pi,1001)
        plt.plot(np.sin(7*omega)+np.sin(8*omega));
```



Humans can hear from about 20 Hz to 20,000 Hz. When the beat frequency is less than 20 Hz, the sum of two sinusoids is heard as an amplitude modulated sinusoid, not as two separate sinusoids.

For example, piano and guitar tuners simultaneously play a reference tone and the note to be tuned and listen for the beat. When the beat is gone, the instrument is properly tuned.

6.2.1 Large Beat Frequency

The beat frequency below, $540 - 440 = 100$ Hz, is within the human hearing range. We hear the two tones, not the beats.

```
In [4]: secs = 3
        t = np.linspace(0,secs,secs*Fs+1)
        w1 = 2*np.pi*440
        w2 = 2*np.pi*540
        sumsignal = np.cos(w1*t)+np.cos(w2*t)
        Audio(sumsignal, rate=Fs)
```

```
Out[4]: <IPython.lib.display.Audio object>
```

6.2.2 Small Beat Frequency

The beat frequency below, $444 - 440 = 4$ Hz, is outside the range of human hearing. We hear one amplitude modulated tone.

```
In [5]: secs = 3
        t = np.linspace(0,secs,secs*Fs+1)
        w1 = 2*np.pi*440
        w2 = 2*np.pi*444
        sumsignal = np.cos(w1*t)+np.cos(w2*t)
        Audio(sumsignal, rate=Fs)
```

```
Out[5]: <IPython.lib.display.Audio object>
```

6.3 Modulation

Modulation is the process of converting a signal's frequencies to new values. Modulation is commonly used in radio to transport a signal, e.g., audio, to new frequencies that can be transmitted electromagnetically.

The basic idea of modulation is to use the trigonometic identity in reverse:

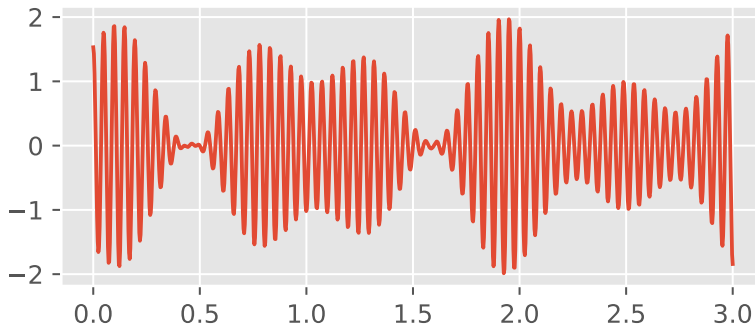
$$\cos(\omega_1 t)\cos(\omega_2 t) = \frac{\cos\big((\omega_1+\omega_2)t\big) + \cos\big((\omega_2-\omega_1)t\big)}{2}$$

The signal to be modulated is $\cos(\omega_1 t)$, the carrier is $\cos(\omega_2 t)$, and the resultant is the sum of two sinusoids. Generally, $\omega_2 \gg \omega_1$.

The example below illustrates *double sideband amplitude modulation* (DSB-AM), the technique used in broadcast AM radio. The signal is scaled so $|s(t)| \leq 1$ for all t. Thus, $1 + s(t) > 0$ for all t.

```
In [6]: w1, w2 = 2*np.pi, 41*np.pi
        t = np.linspace(0,3,1001)
        signal = np.cos(w1*t)+np.sin(1.7*w1*t)
        signal /= np.max(signal)
```

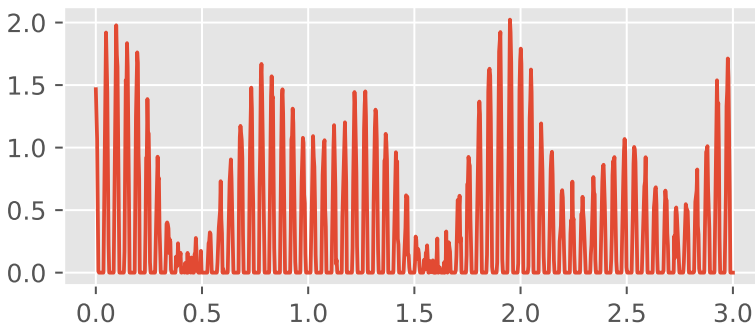
```
carrier = np.cos(w2*t)
sent = (1+signal)*carrier
plt.plot(t, sent);
```



DSB-AM is easily demodulated with basic electronics. (It was developed a century ago when electronics equipment was much more primitive than today. Nonetheless, it is still in use.) The received signal is passed through a diode to eliminate the negative going portions. The resulting signal is low pass filtered to eliminate the "wiggles".

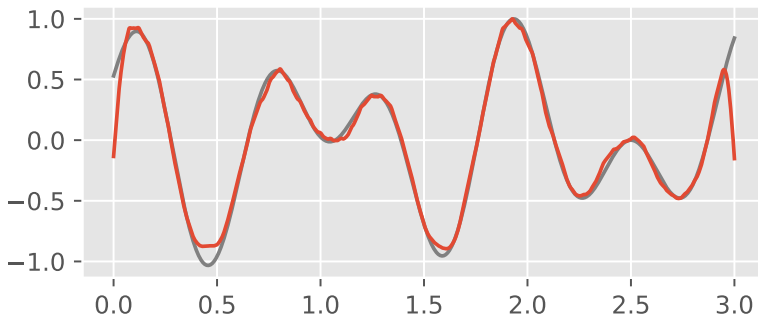
The received signal is usually corrupted by some additive noise.

```
In [7]: received = sent + 0.1* np.random.randn(len(sent))
        rectified = np.maximum(received,0) #mimic the diode
        plt.plot(t,rectified);
```



The rectified signal is low pass filtered. We say more about filtering in a later chapter, but here we use a simple FIR (finite impulse response) filter. The last step is to remove the DC bias (traditionally, this is done with a series capacitor).

```
In [8]: h = sig.firwin(51,0.02)
        demod = np.convolve(rectified, h, 'same')
        demod -= np.average(demod)
        demod /= np.max(np.abs(demod))
        plt.plot(t,signal,'grey')
        plt.plot(t,demod);
```



The demodulated signal is a reasonable approximation to the original (except for filtering transients at each end). Below we compute the demodulated signal's *signal to noise ratio* (SNR) ignoring the initial and final transients.

```
In [9]: MSE = np.var(signal[50:-50]-demod[50:-50])
        SNR = np.var(signal[50:-50])/MSE
        10*np.log10(SNR)  #dB's

Out[9]: 22.233448130160962
```

6.4 Summary

A simple trigonometric identity that says the sum of two sinusoids is equivalent to the product of two sinusoids yields two important applications: beat frequencies, treating the input as a sum of two sinusoids and the output as a product of two sinusoids, and modulation, doing the reverse with the input as the product of two sinusoids and the output as the sum of sinusoids.

The Jupyter notebook can output sounds with the `Audio` widget.

7

Convolution and Polynomials

A fundamental signal processing operation is the *convolution* of two signals. Convolution is a kind of multiplication of one signal by another and yields a third signal. Convolution is closely related to *polynomials* and polynomial multiplication.

We use the `seaborn` plotting style in this notebook.

```
In [1]: import numpy as np
        import matplotlib.pyplot as plt
        plt.style.use('seaborn')
        import matplotlib as mpl
        mpl.rcParams['figure.figsize'] = (5,2)
        %matplotlib inline
        np.set_printoptions(precision=3)
```

The convolution of two signals is denoted $y = x * h$. Note, if `h` and `x` are two numpy arrays, the Python command `h*x` is the element by element multiplication of the two arrays. This is *not* the convolution of the arrays.

In continuous time, the convolution of two signals is

$$y(t) = \int_{-\infty}^{\infty} h(t-\tau)x(\tau)d\tau$$

In discrete time, the convolution is

$$y[n] = \sum_{k=-\infty}^{\infty} h[n-k]x[k]$$

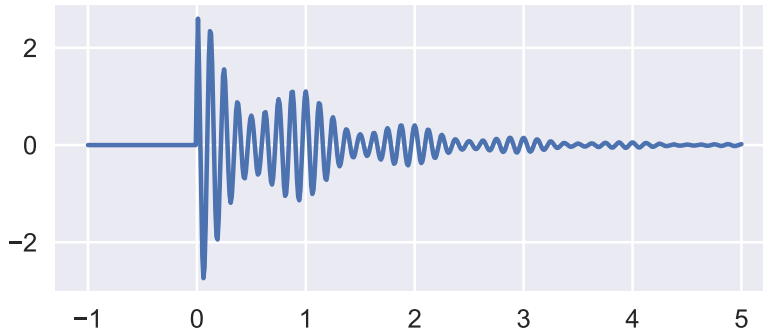
Convolution is commutative, $h * x = x * h$, associative, $(h * g) * x = h * (g * x)$, and distributive, $(h + g) * x = h * x + g * x$.

7.1 Continuous Time Convolution

To help understand convolution, consider the task of ringing a bell. Let $h(t)$ be in sound of the bell when struck. $h(t)$ is called the *impulse response* of the bell.

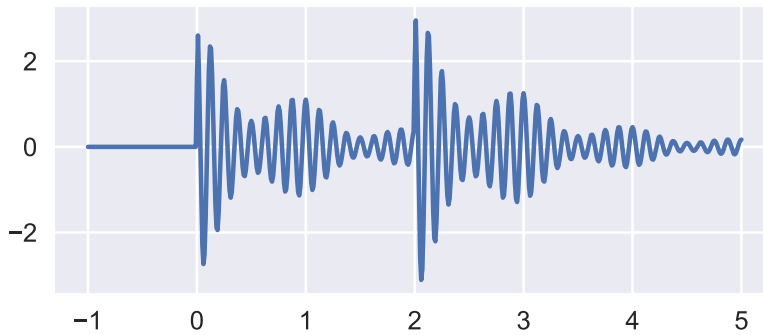
The `unit_step` function below works for numpy arrays (alternately, we could use the `np.heaviside` built-in function). A simple `if` statement does not work for numpy arrays.

```
In [2]: def unit_step(t):
            """compute Heaviside unit step function

            t = real number or numpy array"""
            return 1*(t>0)+0.5*(t==0)

        def bell(t, f = 8, ring = 1, decay = 1):
            """simplified bell impulse response

            f = frequency of bell (Hz)
            ring = ringing frequency of bell (Hz)
            decay = exponential decay constant (seconds)
            """
            tone = np.cos(2*np.pi*f*t)
            ringing = 2+np.cos(2*np.pi*ring*t)
            decaying = np.exp(-t/decay)
            return unit_step(t) * tone * ringing * decaying

In [3]: unit_step(np.linspace(-1,1,7))

Out[3]: array([ 0. ,  0. ,  0. ,  0.5,  1. ,  1. ,  1. ])

In [4]: help(bell)

Help on function bell in module __main__:

bell(t, f=8, ring=1, decay=1)
    simplified bell impulse response

    f = frequency of bell (Hz)
    ring = ringing frequency of bell (Hz)
    decay = exponential decay constant (seconds)



In [5]: t = np.linspace(-1,5,601)
        h = bell(t)
        plt.plot(t,h);
```



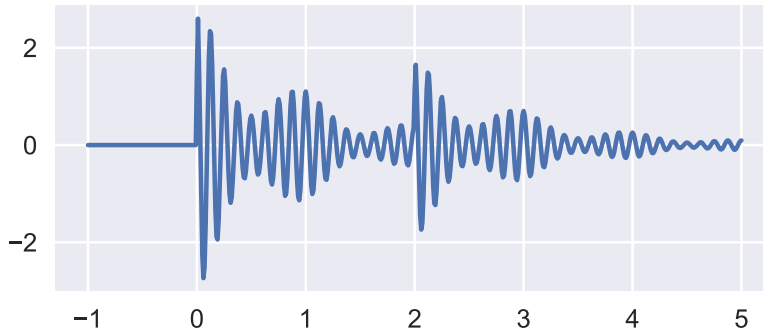
Now imagine the bell is struck a second time at $t = 2$ seconds. Furthermore, assume the bell is linear and the two sounds add together. (If you object to striking an already vibrating bell, imagine a second bell, identical to the first, is struck.)

```
In [6]: sound = bell(t) + bell(t-2)
        plt.plot(t,sound);
```



Perhaps the second strike has half the strength of the first.

```
In [7]: sound = bell(t) + 0.5*bell(t-2)
        plt.plot(t,sound);
```



In the example above, the input to the bell *system* consists of two *impulses*, one at $t = 0$ and one at $t = 2$.

$$x(t) = \delta(t) + 0.5\delta(t-2)$$

The output is the convolution of the impulse response (bell ringing) and the input.

$$\begin{aligned} y(t) &= \int_{-\infty}^{\infty} h(\tau)x(t-\tau)d\tau \\ &= \int_{-\infty}^{\infty} h(\tau)\big((\delta(t-\tau) + 0.5\delta(t-2-\tau)\big)d\tau \\ &= h(t) + 0.5h(t-2) \end{aligned}$$

In general, $x(t)$ can be described as a sum (integral) of many delta functions,

$$x(t) = \int_{-\infty}^{\infty} x(\tau)\delta(t-\tau)d\tau$$

Since an input of $\delta(t-\tau)$ produces an output $h(t-\tau)$, and $x(t)$ can be written as a sum of $\delta(t-\tau)$, the output can be written as a sum of $h(t-\tau)$. This is known as the *convolution* integral.

$$y(t) = \int_{-\infty}^{\infty} x(\tau)h(t-\tau)d\tau$$

The convolution integral describes the output of a linear, time invariant, system for any input $x(t)$.

7.2 Discrete Time Convolution

In discrete time, the integral is replaced by a sum.

$$y[n] = \sum_{k=-\infty}^{\infty} h[k]x[n-k]$$

Discrete time convolution obeys a simple length relation. Let `len(x)` be the length of `x`, i.e., the number of elements from the first nonzero element to the last nonzero element and let `len(h)` be the length of `h`. Then, `len(y) = len(x)+len(h)-1`.

Numpy provides a command for discrete convolution, `np.convolve`.

```
In [8]: x = np.array([1,2,3,4])
        h = np.array([1,1,2,-3])
        y = np.convolve(x,h)
        y
```

```
Out[8]: array([  1,   3,   7,   8,   4,  -1, -12])
```

```
In [9]: print(len(x), len(h), len(y), len(y) == len(x)+len(h)-1)
```

```
4 4 7 True
```

A simple way to implement convolution mimics the bell argument above. Each value of x adds a time shifted and scaled version of the impulse response.

Note, the function will be faster if we loop over the shorter of x and h. Of course, the `np.convolve` function is faster (and likely better) than anything we can write in Python since it is written in a compiled language (and has been extensively tested).

```
In [10]: lx, lh = len(x), len(h)
         ly = lx+lh-1
         y = np.zeros(ly)
         for i,v in enumerate(x):
             y[i:i+lh] += v*h
         y
```

```
Out[10]: array([  1.,   3.,   7.,   8.,   4.,  -1., -12.])
```

```
In [11]: def my_convolve(x,h):
             """compute the convolution of x and h"""
             if len(h) > len(x):
                 x,h  = h,x #swap x and h
             lh, lx = len(h), len(x)
             y = np.zeros(lh+lx-1).astype('int')
             for i,v in enumerate(h):
                 y[i:i+lx] += v*x #mimic the bell
             return y
```

```
In [12]: np.convolve(x,h), my_convolve(x,h), my_convolve(h,x)
```

```
Out[12]: (array([  1,   3,   7,   8,   4,  -1, -12]),
          array([  1,   3,   7,   8,   4,  -1, -12]),
          array([  1,   3,   7,   8,   4,  -1, -12]))
```

Another simple check is to compute the sum of the values. The sum of the output is the product of the sums of the two inputs.

```
In [13]: np.sum(y), np.sum(h), np.sum(x), np.sum(y)==np.sum(h)*np.sum(x)
```

```
Out[13]: (10.0, 1, 10, True)
```

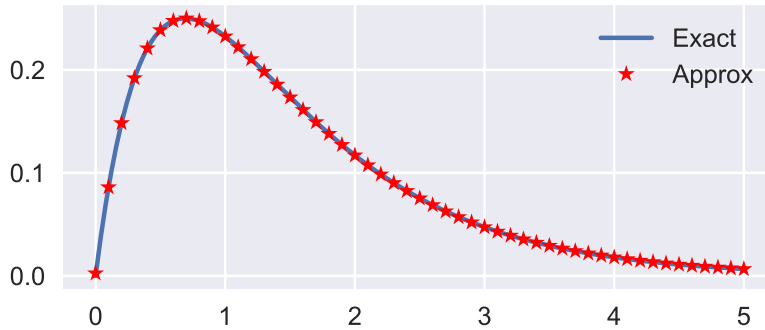
7.3 Using the Convolution Function to Approximate Continuous Time Convolution

Numpy computes numbers, not symbolic integrals. We can, however, use `np.convolve` to approximate the integral in the continuous convolution. The key is to discretize $x(t)$ and $h(t)$ at the same scale, i.e., the separation between samples is the same for both $x(t)$ and $h(t)$.

For example, if $x(t) = e^{-at}u(t)$ and $h(t) = e^{-bt}u(t)$ with a and b positive and $a \neq b$, then

$$y(t) = \int_0^t e^{-b\tau}e^{-a(t-\tau)}d\tau = \frac{e^{-at} - e^{-bt}}{b-a}u(t)$$

```
In [14]: a, b = 1, 2
         Fs = 100 #samples per second
         deltat = 1.0/Fs #time between samples
         tmax = 5
         t = np.linspace(0,tmax,tmax*Fs+1)
         x = np.exp(-a*t) * unit_step(t)
         h = np.exp(-b*t) * unit_step(t)

         yexact = (np.exp(-a*t)-np.exp(-b*t))/(b-a) * unit_step(t)
         yapprox = deltat*np.convolve(x,h)

         #plot every 10th point
         plt.plot(t,yexact, label='Exact')
         plt.plot(t[::10],yapprox[:len(t):10],'r*',label='Approx')
         plt.legend();
```



7.4 Polynomials

Numpy has a `Polynomial` class. Multiplication of polynomials corresponds to the convolution of the arrays of coefficients. For example, $(1 + 2x)(3 + 4x + 5x^2) = 1 \cdot 3 + (1 \cdot 4 + 2 \cdot 3)x + (2 \cdot 4 + 1 \cdot 5)x^2 + (2 \cdot 5)x^2 = 3 + 10x + 13x^2 + 10x^3$.

Polynomials are listed from lowest power (x^0) to highest power (x^n). Numpy polynomials consist of three parts: the *coefficients*, the *domain*, and the *window*. In this tutorial, we are interested only in the coefficients, not the domain or window.

Polynomials can be multiplied `p*q`, added `p+q`, differentiated `p.deriv`, integrated `p.integ`, divided `p//q`, evaluated `p(x)`, and solved `p.roots` (among other things).

```
In [15]: from numpy.polynomial import Polynomial as P
         p = P([1,2])
         q = P([3,4,5])
         print(p, q, p*q)
```

```
poly([ 1.  2.]) poly([ 3.  4.  5.]) poly([  3.  10.  13.  10.])
```

```
In [16]: np.convolve([1,2], [3,4,5]) #same as p*q coefficients
```

```
Out[16]: array([ 3, 10, 13, 10])
```

7.4.1 Division:

`q//p` returns the divisor polynomial of `q` when divided by `p`.
`q%p` returns the remainder polynomial.
`divmod(q,p)` returns both.

```
In [17]: quotient, remainder = divmod(q,p)
         print(q//p, quotient)
         print(q%p, remainder)
```

```
poly([ 0.75  2.5 ]) poly([ 0.75  2.5 ])
poly([ 2.25]) poly([ 2.25])
```

```
In [18]: print(remainder + quotient*p)
```

```
poly([ 3.  4.  5.])
```

7.4.2 Calculus with Polynomials

One derivative of q is $\frac{dq}{dx} = 4 + 10x$, two derivatives are 10.

```
In [19]: print(q.deriv(), q.deriv(m=2))
```

```
poly([  4.  10.]) poly([ 10.])
```

Integrating a polynomial results in a new polynomial of one higher degree. The integrating constant can be specified with `k`. If not specified, the constant is set to 0.

```
In [20]: print(p.integ(), p.integ(k=6))
```

```
poly([ 0.  1.  1.]) poly([ 6.  1.  1.])
```

The roots of polynomials with real coefficients are either real or occur in complex conjugate pairs. Polynomials can also be defined from a list of roots.

```
In [21]: p.roots(), q.roots()
```

```
Out[21]: (array([-0.5]), array([-0.4-0.663j, -0.4+0.663j]))
```

```
In [22]: print(P.fromroots(q.roots()))
```

```
poly([ 0.6+0.j  0.8+0.j  1.0+0.j])
```


7.4.3 Wilkinson's Polynomial

One must be careful when factoring large polynomials. A famous example is *Wilkinson's polynomial*.

$$W(x) = \prod_{k=1}^{20} (x - k)$$

The roots of the Wilkinson polynomial are the integers $1, 2, 3, \ldots, 20$. Below we create the Wilkinson polynomial and calculate its roots. Notice, many of the calculated roots are inaccurate.

```
In [23]: roots = np.arange(1,21)
         wilkinson = P.fromroots(roots)
         wilkinson.coef
```

```
Out[23]: array([  2.433e+18,  -8.753e+18,   1.380e+19,  -1.287e+19,
                  8.038e+18,  -3.600e+18,   1.207e+18,  -3.113e+17,
                  6.303e+16,  -1.014e+16,   1.308e+15,  -1.356e+14,
                  1.131e+13,  -7.561e+11,   4.017e+10,  -1.672e+09,
                  5.333e+07,  -1.257e+06,   2.062e+04,  -2.100e+02,
                  1.000e+00])
```

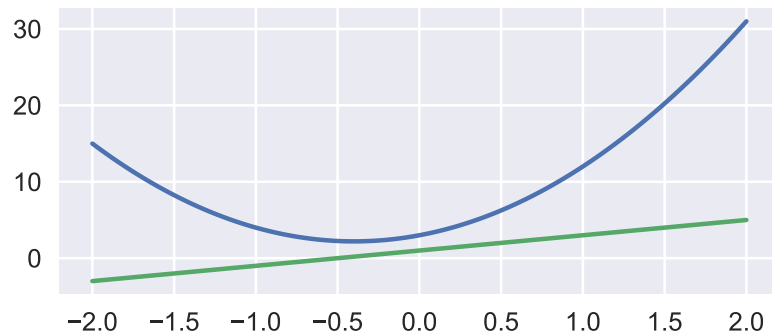
```
In [24]: wilkinson.roots()
```

```
Out[24]: array([  1.000+0.j   ,   2.000+0.j   ,   3.000+0.j   ,
                  4.000+0.j   ,   5.000+0.j   ,   6.000+0.j   ,
                  7.000+0.j   ,   8.002+0.j   ,   8.988+0.j   ,
                 10.047+0.j   ,  10.890+0.j   ,  12.341+0.j   ,
                 12.578+0.j   ,  14.518-0.208j,  14.518+0.208j,
                 16.206+0.j   ,  16.886+0.j   ,  18.030+0.j   ,
                 18.994+0.j   ,  20.001+0.j   ])
```

7.4.4 Evaluating and Plotting Polynomials

Plotting and evaluating polynomials work as expected.

```
In [25]: x = np.linspace(-2,2,101)
         plt.plot(x,q(x))
         plt.plot(x,p(x));
```



8

Filtering

In this chapter, we consider the problem of filtering a signal to emphasize or deemphasize specific components. The most common filters are lowpass (allows low frequencies to pass while stopping or reducing high frequencies), highpass (the reverse, stops low frequencies and passes high frequencies), bandpass (allows mid-frequencies), bandstop (blocks mid-frequencies), and allpass (changes phase while keeping magnitude constant).

The two most important classes of digital filters are FIR (finite impulse response) and IIR (infinite impulse response).

In addition to `numpy` and `matplotlib`, we import `scipy` (a collection of scientific libraries) and `scipy.signal` (signal processing functions). The `norm` library implements various Gaussian probability functions. Setting matplotlib's `rcParams['figure.figsize']` sets the default plot size (in inches) for this notebook.

```
In [1]: import numpy as np
        import matplotlib
        import matplotlib.pyplot as plt
        import scipy
        import scipy.signal as sig
        from scipy.stats.distributions import norm
        np.set_printoptions(precision=3,suppress=True)
        matplotlib.rcParams['figure.figsize']=(5.0,1.5)
        %matplotlib inline
```

8.1 FIR Filters

An FIR filter consists of a sequence of coefficients, $h(k)$ for $k = 0, 1, 2, \ldots, N-1$. The coefficients are the impulse response of the filter. The filter output is the convolution of the input sequence and the coefficient sequence.

$$y[n] = \sum_{k=0}^{N-1} x[n-k]h[k]$$

The filter's frequency response can be calculated from its Z-transform:

$$H(z) = \sum_{k=0}^{N-1} h[k]z^{-k}$$

$$H(\omega) = \sum_{k=0}^{N-1} h[k]e^{-j\omega k}$$

FIR filters have some advantages:

- If $h[k] = h[N-1-k]$ for all k (or $h[k] = -h[N-1-k]$ for all k), the filter is *linear phase*, meaning all frequency components are delayed equally. For instance, in many audio applications linear phase is desirable.
- Modern DSP processors can implement FIR filters easily and efficiently.
- FIR filters are always stable.

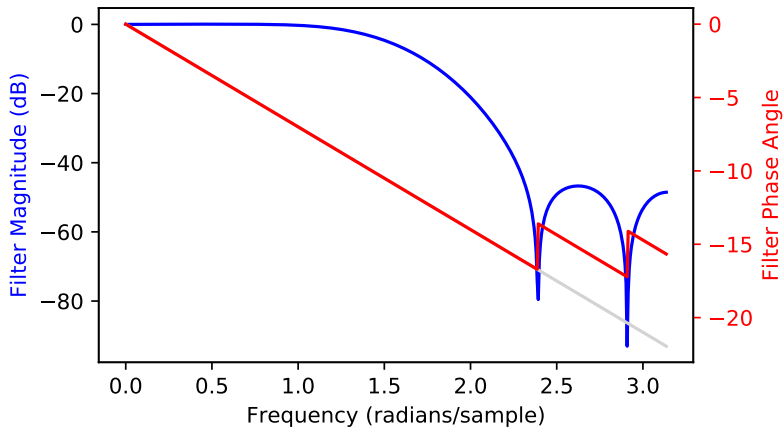
The code below uses `firwin` to compute the coefficients for a simple lowpass filter with cutoff frequency $f = 0.25$ cycles per sample. Note, `firwin` uses the peculiar convention (taken from Matlab) of relating frequencies to the Nyquist frequency. By default, `nyq=1.0`. E.g. `firwin(N,0.5)` creates an N point filter with cutoff frequency $0.5 \times 0.5 = 0.25$ cycles per sample. To use a different scaling, e.g., radians per sample, set `nyq=np.pi`.

Note the alternate coefficients are all approximately 0. The remaining coefficients are symmetric. Both these properties can be exploited to speed up an implementation. A careful implementation requires five multiplications and eight additions per output point.

```
In [2]: N = 15 #number of coefficients
        b = sig.firwin(N,0.5) #0.5 times Nyquist
        b

Out[2]: array([-0.004,  0.   ,  0.016, -0.   , -0.068,  0.   ,
                0.305,  0.502,  0.305,  0.   , -0.068, -0.   ,
                0.016,  0.   , -0.004])
```

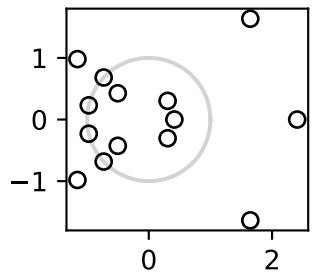
The filter's magnitude and phase response are plotted below. The magnitude is plotted in deciBels. Note the magnitude is approximately equal to 1 for $0 \leq \omega \leq \pi/2$. The phase is linear with a slope of -7 samples, i.e., the filter implements a delay of 7 samples for all frequencies. The phase plot has discontinuities at the two zero locations (where the frequency response changes sign), but it is still linear phase.

```
In [3]: f, w = sig.freqz(b) #f=radians per sample
        fig, ax1 = plt.subplots(figsize=(5,3))
        ax1.plot(f,20*np.log10(np.abs(w)),color='blue')
        ax1.set_ylabel('Filter Magnitude (dB)',color='blue')
        ax1.set_xlabel('Frequency (radians/sample)')
        ax2 = ax1.twinx()
        ax2.plot(f,-(N-1)*f/2,'lightgrey') #delay of (N-1)/2 samples
        ax2.plot(f,np.unwrap(np.angle(w)),'r')
        ax2.tick_params('y', colors='r')
        ax2.set_ylabel('Filter Phase Angle',color='r');
```



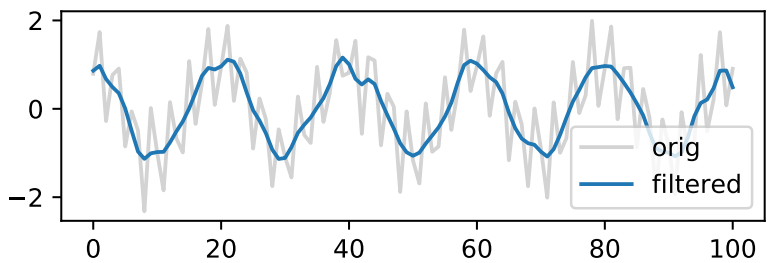
In addition to the magnitude and phase, filters can be described through *pole-zero* plots. FIR filters have only zeros and no poles. Zeros can be inside or outside the unit circle. Below are plotted the zero locations.

```
In [4]: h = sig.lti(b,1)
        theta = np.linspace(0,2*np.pi,101)
        plt.plot(np.cos(theta), np.sin(theta),'lightgrey')
        plt.plot(np.real(h.zeros),np.imag(h.zeros),'ko',
                 markerfacecolor='none')
        plt.axes().set_aspect('equal')
```



The signal below is a sum of two sinusoids, one at frequency 0.05 cycles per sample and one at 0.35, corrupted by some additive Gaussian noise. The filter passes the lower frequency sinusoid and attenuates the higher frequency sinusoid. High frequency noise is also attenuated.

```
In [5]: t = np.arange(101)
        f1, f2 = 0.05, 0.35 #cycles per sample
        x = np.cos(2*np.pi*f1*t) + np.sin(2*np.pi*f2*t)
        x += 0.2*np.random.randn(len(t))
        plt.plot(t,x,'lightgrey',label='orig')
        y = np.convolve(x,b,'same')
        plt.plot(t,y,label='filtered')
        plt.legend();
```



8.2 IIR Filters

An IIR (infinite impulse response) filter can be described by a difference equation, where $y[n]$ is the output sequence and $x[n]$ is the input sequence:

$$\sum_{k=0}^{N-1} a[k]y[n-k] = \sum_{k=0}^{N-1} b[k]x[n-k]$$

Often $a[0] = 1$. Then, $y[n]$ can be solved for as follows:

$$y[n] = -\sum_{k=1}^{N-1} a[k]y[n-k] + \sum_{k=0}^{N-1} b[k]x[n-k]$$

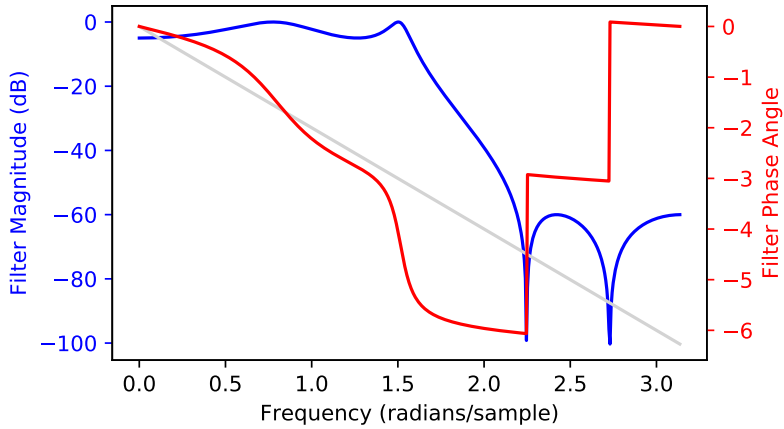
The current output is a sum of previous outputs and recent inputs.

In some applications, it is necessary to consider the initial conditions of the difference equation. These are generally given as values for $y[-1], y[-2], \ldots, y[-N+1]$. The initial values are set to 0 in many signal processing applications for the simple reason that the effects of the initial values die out quickly compared to the length of the input sequence. If this is not true, as in many control applications, the initial conditions must be accounted for.

Compared to FIR filters, IIR filters can achieve similar magnitude response with fewer terms, but usually at the expense of not having linear phase. All poles must be within the unit circle (for discrete systems) or in the left half plane (for continuous systems) to ensure the filter is stable; zeros can be anywhere without affecting stability.

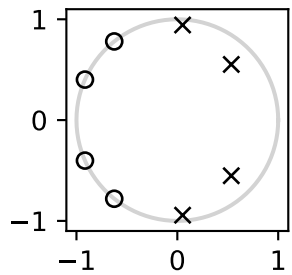
Below we calculate a simple IIR lowpass filter using the `iirfilter` function. Its magnitude and phase versus frequency are plotted below. Notice the following: (1) the magnitude response is not flat across the passband, (2) the magnitude drops quickly in the stop band, and (3) the phase response is not linear in the passband.

```
In [6]: b, a = sig.iirfilter(4, Wn=0.5, rp=5, rs=60,
                            btype='lowpass', ftype='ellip')
        f, w = sig.freqz(b,a) #f=radians per sample
        fig, ax1 = plt.subplots(figsize=(5,3))
        ax1.plot(f,20*np.log10(np.abs(w)),'b')
        ax1.set_ylabel('Filter Magnitude (dB)',color='blue')
        ax1.set_xlabel('Frequency (radians/sample)')
        ax1.tick_params('y', colors='b')
        ax2 = ax1.twinx()
        ax2.plot(f,-2*f,'lightgrey') #approx delay = 2 samples
        ax2.plot(f,np.unwrap(np.angle(w)),'r')
        ax2.tick_params('y', colors='r')
        ax2.set_ylabel('Filter Phase Angle',color='r');
```



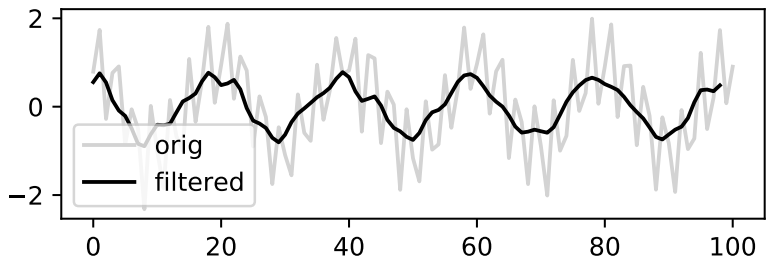
The pole-zero diagram is handy for understanding the IIR filter and verifying all poles are within the unit circle.

```
In [7]: h = sig.lti(b,a)
        theta = np.linspace(0,2*np.pi,101)
        plt.plot(np.cos(theta), np.sin(theta),'lightgrey')
        plt.plot(np.real(h.zeros),np.imag(h.zeros),'ko',
                 markerfacecolor='none')
        plt.plot(np.real(h.poles),np.imag(h.poles),'kx')
        plt.axes().set_aspect('equal')
```



The same sinusoidal signal as above filtered by the IIR filter. Notice the output is smaller than the output of the FIR filter. That is because this particular IIR filter has gain less than 1 (about 0.636) at $f = 0.05$ cycles per sample.

```
In [8]: t = np.arange(101)
        plt.plot(t,x,'lightgrey',label='orig')
        y = sig.lfilter(b,a,x)
        delay = 2
        plt.plot(t[:-delay],y[delay:],'k',label='filtered')
        plt.legend();
```



To find the gains at the two input frequencies, we need to search the `f` array and then find the value of the `w` array at those locations. The frequency array can be searched efficiently with the `bisect` command, which performs a bisection search to find where the second argument fits into the first argument (a list or array). Once we know where the value fits, we can look up the frequency response there. The low frequency sinusoid is reduced by a factor of 0.64 and the higher one by a factor of about 1000.

```
In [9]: from bisect import bisect
        i1, i2 = bisect(f,f1*2*np.pi), bisect(f,f2*2*np.pi)
        np.abs(w[i1]), np.abs(w[i2])
```

```
Out[9]: (0.63597667489061427, 0.00076082618308510506)
```

8.3 Example: Filtering an ECG Signal

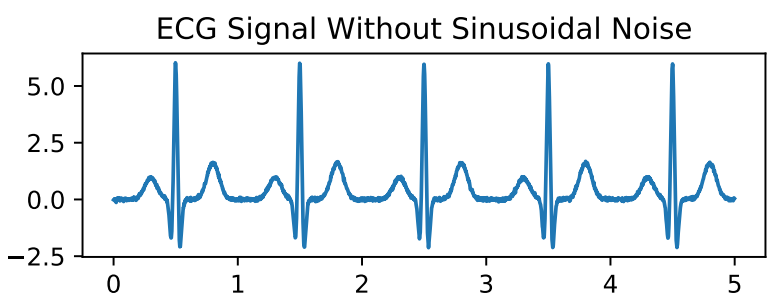
Electrocardiogram (ECG) signals measure the electrical activity across the heart. They are useful for detecting heart anomalies. Circuits to measure ECG signals are simple enough to be built in undergraduate circuits classes. One practical problem in the ECG measuring devices is eliminating 60 Hz (or 50 Hz, depending on your location) power line interference.

Healthy ECG signals consist of a repeating pattern of a P wave (a small hump), the QRS complex (a small downward spike, followed by a large positive spike, and then a small downward spike, and the T wave (another hump). Most hearts beat between 60 and 75 beats per minute.

We create an artificial ECG signal as follows: the P and T waves are modeled by the Gaussian density and the QRS complex by a modified "Mexican hat wavelet"

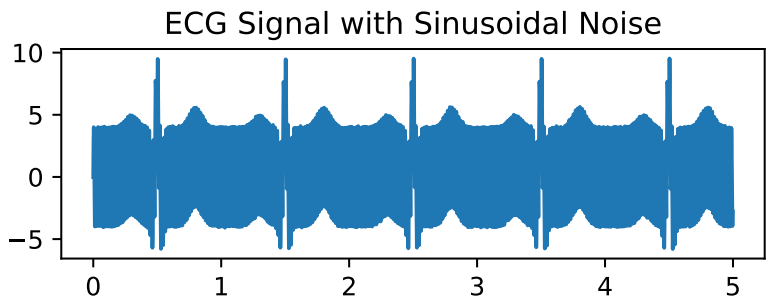
(the second derivative of the Gaussian density). The pattern is repeated five times and corrupted by a little Gaussian noise and a strong 60Hz sinusoid.

```
In [10]: T, Fs, copies = 1.0, 480, 5
         phi = norm.pdf
         t = np.linspace(0,T,int(T*Fs),endpoint=False) #one period
         x = 0.12*phi(t,0.3,0.05) #P wave
         x += (1.-600*(t-0.5)**2-1.4*t)*phi(t,0.5,0.02) #QRS
         x += 0.2*phi(t,0.8,0.05) #T wave
         t = np.linspace(0,T*copies,int(T*Fs*copies),endpoint=False)
         ecg = np.concatenate([x]*copies)  #make array of copies
         ecg += 0.04*np.random.randn(len(ecg)) #a little noise
         plt.title('ECG Signal Without Sinusoidal Noise')
         plt.plot(t, ecg);
```



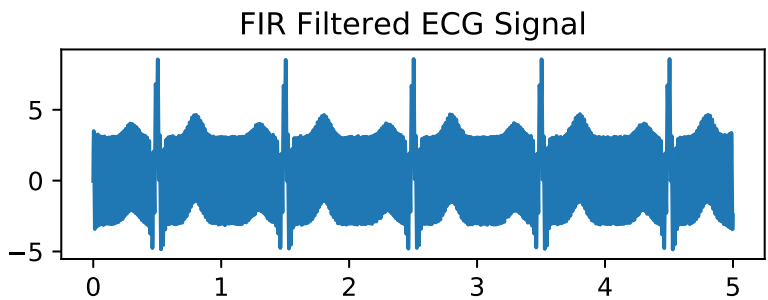
The signal corrupted by 60 Hz noise is useless for diagnostic purposes.

```
In [11]: ecg += 4*np.sin(2*np.pi*60*t) #60Hz noise
         plt.title('ECG Signal with Sinusoidal Noise')
         plt.plot(t,ecg);
```



FIR filters are not suited for removing strong sinusoidal interference. Even a long filter with 101 points performs poorly. The filter tries to maintain unity gain from 0 to 58 Hz, 0 gain at 60 Hz, and unity gain again from 62 to 240 Hz.

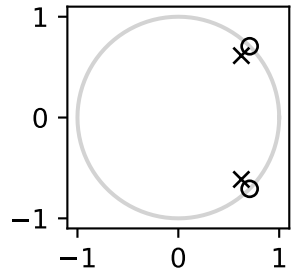
```
In [12]: h = sig.firwin2(101, [0.0,58.0,60,62.0,240], [1,1,0,1,1],
                         nyq=240)
         y = np.convolve(ecg, h,'same')
         plt.title('FIR Filtered ECG Signal')
         plt.plot(t,y);
```



However, as shown below, a simple second order IIR filter removes the interference and causes little distortion to the underlying ECG signal. The filter puts zeros on the unit circle at 60 Hz and places nearby poles inside the unit circle. The magnitude response is approximately unity everywhere except at 60 Hz.

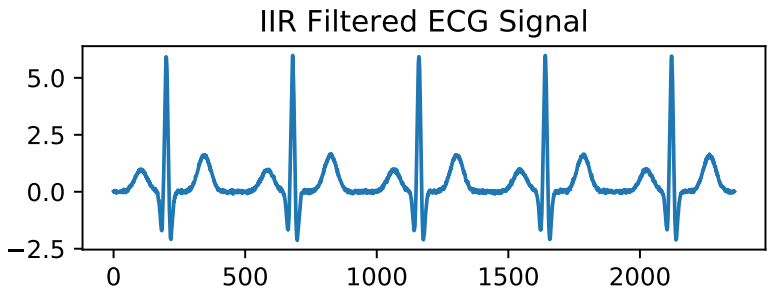
The pole-zero diagram shows how the notch filter is designed. Zeros on the unit circle to eliminate the 60 Hz interference and nearby poles to cause the gain to be approximately unity everywhere else. The `Q` parameter determines how close the poles are to the unit circle.

```
In [13]: b, a = sig.iirnotch(60*2/Fs, Q=3)
         h = sig.lti(b,a)
         theta = np.linspace(0,2*np.pi,101)
         plt.plot(np.cos(theta), np.sin(theta),'lightgrey')
         plt.plot(np.real(h.zeros),np.imag(h.zeros),'ko',
                  markerfacecolor='none')
         plt.plot(np.real(h.poles),np.imag(h.poles),'kx')
         plt.axes().set_aspect('equal');
```



The IIR filter does a great job. The filtered output (aside from an initial transient which can be ignored) is almost identical to the original signal.

```
In [14]: y = sig.lfilter(b,a,ecg)
         plt.title('IIR Filtered ECG Signal')
         plt.plot(y[40:]); #drop initial transient
```



8.4 Summary

FIR and IIR filters are the two main types of digital filters. FIR filters can be designed with linear phase and can be implemented easily with modern DSP processors. FIR filters are widely used in audio applications. IIR filters often have better performance than FIR filters with fewer coefficients. The IIR filter is dramatically better than the FIR filter in the ECG sinusoidal removal problem, for instance.

It pays to consider several examples of both types of filters before selecting one for a specific application. There are many practical considerations not considered here: implementation speed (and speed of writing and testing the code), numerical accuracy, and floating point versus fixed point implementations, to name a few.

9

Linear Systems

In this chapter, we consider a subset of linear time-invariance systems that can be represented by linear constant coefficient differential equations. The `scipy.signal` library includes functions for both continuous time and discrete time linear systems. Here we consider continous time.

As usual, begin by importing the usual libraries.

```
In [1]: import numpy as np
        import matplotlib.pyplot as plt
        import scipy.signal as sig
        import matplotlib
        matplotlib.rcParams['figure.figsize'] = (5,2)
        %matplotlib inline
```

9.1 Second Order Systems

General linear differential equations can be factored into a product of first and second order differential equations. We consider a standard second order system as presented in *Oppenheim, Willsky, and Hamid*, equations (6.31) and (6.33),

$$\frac{d^2y(t)}{dt^2} + 2\psi\omega_n\frac{dy(t)}{dt} + \omega_n^2 y(t) = \omega_n^2 x(t)$$

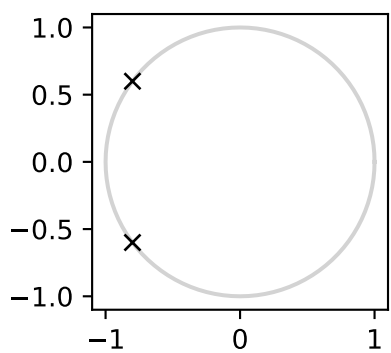
$$H(j\omega) = \frac{\omega_n^2}{(j\omega)^2 + 2\psi\omega_n(j\omega) + \omega_n^2}$$

Stability requires that $\psi > 0$ and ω_n real.

The `signal.lti` function represents a continuous time linear time invariant system. It can be initialized with the numerator and denominator coefficients or with the poles and zeros or with a matrix A, B, C, D representation (which is beyond this text, but is frequently encountered in control applications). Here we use the numerator and denominator polynomials. The coefficients are ordered from highest power to lowest power.

After initializing the system, we check the poles and zeros. This system has no zeros and has two poles. When $\psi < 1$, the poles are on the left side of unit circle, when $\psi = 1$, both poles are at $s = -1$, and when $\psi > 1$, the poles are real with one inside the unit circle and one outside. These values can be checked by varying `psi` in the cell below.

```
In [2]: #O&W Fig 6.22 on pg 453
        wn, psi = 1, 0.8
        num = [wn**2]
        den = [1, 2*psi*wn, wn**2]
        h = sig.lti(num, den)
        h.zeros, h.poles, np.abs(h.poles)

Out[2]: (array([], dtype=float64),
         array([-0.8+0.6j, -0.8-0.6j]),
         array([ 1.,  1.]))

In [3]: theta = np.linspace(0,2*np.pi,101)
        plt.plot(np.cos(theta), np.sin(theta),'lightgrey')
        plt.plot(np.real(h.zeros),np.imag(h.zeros),'ko',markerfacecolor='none')
        plt.plot(np.real(h.poles),np.imag(h.poles),'kx')
        plt.axes().set_aspect('equal')
```

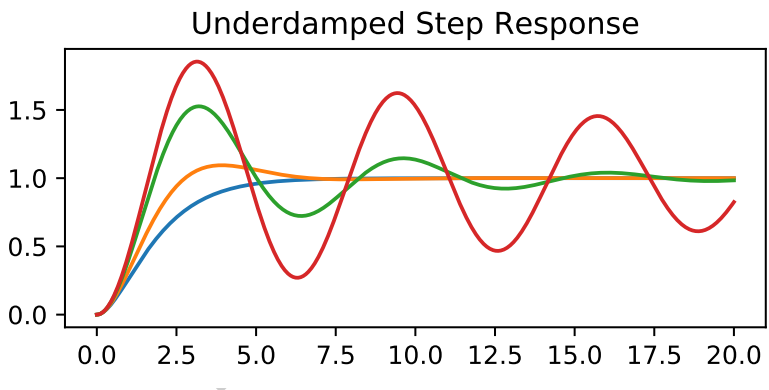


9.2 Underdamped systems

Underdamped systems "ring". Think of a swinging door slowly settling down, or a "bobble head" doll.

We plot the step and impulse responses for various values of the damping parameter, `psi`. In all the plots following below, the critically damped system `psi=1` is plotted in blue.

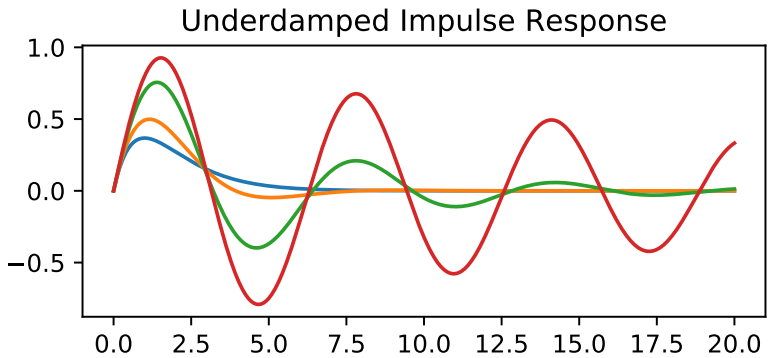
```
In [4]: wn = 1
        psis = [1.0, 0.6, 0.2, 0.05]
        T = np.linspace(0,20,1001)
        for psi in psis:
            num = [wn**2]
            den = [1, 2*psi*wn, wn**2]
            h = sig.lti(num,den)
            T, yout = sig.step2(h,T=T)
            plt.plot(T,yout)
        plt.title('Underdamped Step Response');
```



The ringing can be undesirable. For example, consider an automobile cruise control system when the intended speed is increased. An underdamped system (the red curve above) overshoots (speeds up beyond the set point), then undershoots (slows down below the set point), until eventually settling down to the desired speed. Such a system would not work well in practice.

```
In [5]: wn = 1
        psis = [1.0, 0.6, 0.2, 0.05]
        T = np.linspace(0,20,1001)
        for psi in psis:
            num = [wn**2]
            den = [1, 2*psi*wn, wn**2]
```

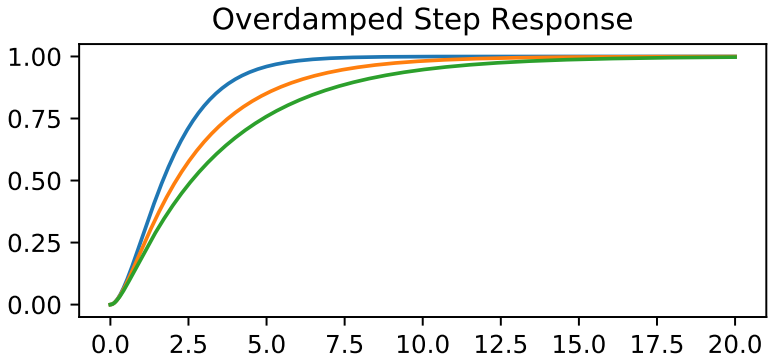
```
        h = sig.lti(num,den)
        T, yout = sig.impulse2(h, T=T)
        plt.plot(T,yout)
    plt.title('Underdamped Impulse Response');
```



9.3 Overdamped systems

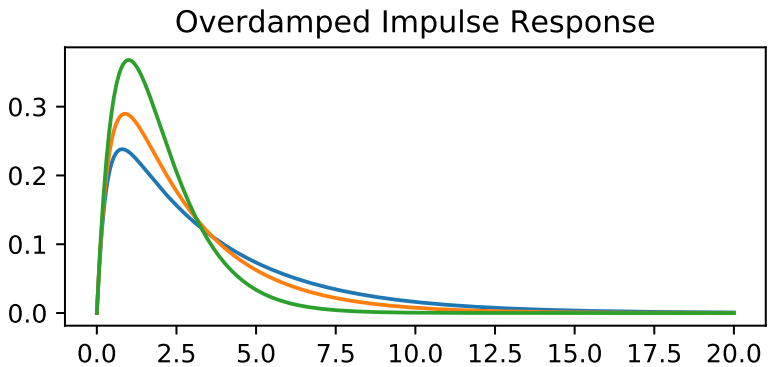
Overdamped systems do not oscillate, but they take their time to settle down to the final value, i.e., as `psi` increases, the settling time increases.

```
In [6]: wn = 1
        psis = [1.0, 1.4, 1.8]
        T = np.linspace(0,20,1001)
        for psi in psis:
            num = [wn**2]
            den = [1, 2*psi*wn, wn**2]
            h = sig.lti(num,den)
            T, yout = sig.step2(h, T=T)
            plt.plot(T,yout)
        plt.title('Overdamped Step Response');
```



An overdamped system is the opposite of underdamped. If the damping parameter is too high, the system responds too slowly. For instance, a cruise control system would take too long to respond to driver set point changes.

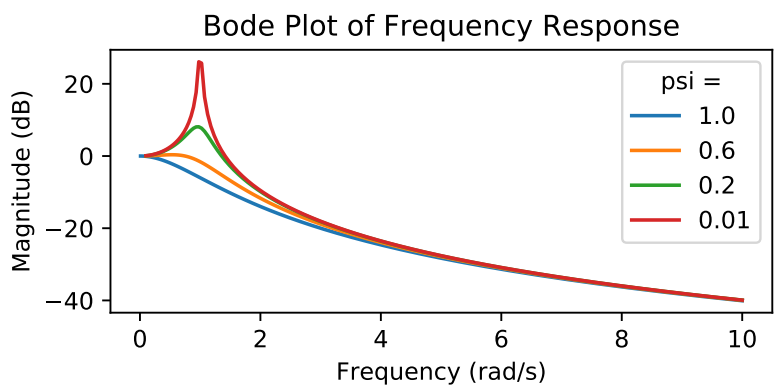
```
In [7]: wn = 1
        psis = [1.8, 1.4, 1.0]
        T = np.linspace(0,20,1001)
        for psi in psis:
            num = [wn**2]
            den = [1, 2*psi*wn, wn**2]
            h = sig.lti(num,den)
            T, yout = sig.impulse2(h, T=T)
            plt.plot(T,yout)
        plt.title('Overdamped Impulse Response');
```

9.4 Bode plot of frequency response

Note the peak of the frequency response occurs at approximately $\omega_n\sqrt{1-\psi^2}$. The Q of this system is $Q = 1/(2\psi)$. To get a sharp bandpass filter, make Q large, i.e., make ψ small.

```
In [8]: wn = 1
        psis = [1.0, 0.6, 0.2, 0.01]
        for psi in psis:
            num = [wn**2]
            den = [1, 2*psi*wn, wn**2]
            h = sig.lti(num,den)
            w, mag, phase = sig.bode(h)
            plt.plot(w,mag,label=psi)
        plt.legend(title='psi =')
        plt.ylabel('Magnitude (dB)')
        plt.xlabel('Frequency (rad/s)')
        plt.title('Bode Plot of Frequency Response');
```



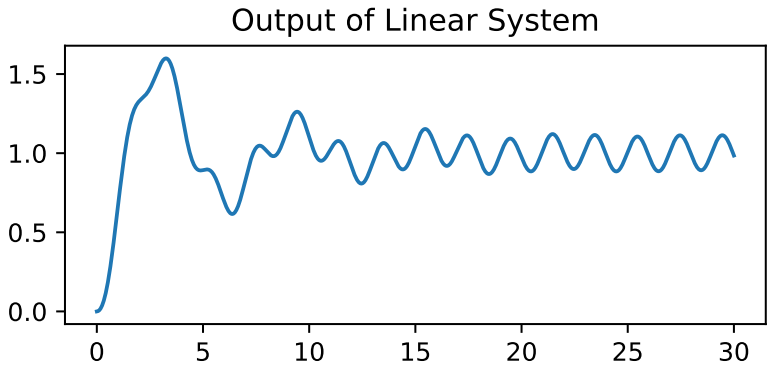
9.5 Output for General Inputs

The `sig.lsim` function integrates the differential equation for general inputs. In the example below, an underdamped system with `psi=0.2` and input consisting of a sum of a sinusoid and a unit step is simulated for $0 \le t \le 30$. Notice the initial overshoot (to about 1.6) and the initial ringing. Eventually the system settles down and the output is sinusoidal at the input frequency ($f = 0.5$) around $y = 1$.

```
In [9]: psi, wn, Fs = 0.2, 1.0, 100
        num, den = [wn**2], [1, 2*psi*wn, wn**2]
```

```
h = sig.lti(num,den)
t = np.linspace(0,30,int(30*Fs+1))
U = np.sin(2*np.pi*0.5*t)+1
t, yout, xout = sig.lsim(h,U,t)
print(np.max(yout))
plt.plot(t,yout)
plt.title('Output of Linear System');
```

```
1.59922097824
```



9.6 Summary

Many linear systems are modeled as differential equations (in continuous time) or difference equations (in discrete time). This chapter demonstrated some typical calculations on linear systems. Step and impulse responses are calculated and plotted. Bode plots plot the frequency response of the system. Finally, outputs are calculated (the differential equation is integrated) for a general input.

10

The Discrete Fourier Transform and FFT

The Discrete Fourier Transform (DFT) is one of the workhorse algorithms of signal processing. In this chapter we introduce the DFT and discuss the Fast Fourier Transform (FFT).

We emphasize the DFT and FFT are the same transform. Given the same inputs, they compute the same numbers. The FFT is a faster implementation of the DFT (for large input sizes).

Below we import a few new libraries, *time* and *timeit* for timing Python code, and the standard FFT and inverse functions, *fft* and *ifft*.

```
In [1]: import numpy as np
        import matplotlib.pyplot as plt
        import time
        import timeit
        from numpy.fft import fft, ifft
        plt.style.use('ggplot')
        %precision 3
        %matplotlib inline
```

10.1 DFT Definition and Inverse

Given a possibly complex input $x = [x_0, x_1, \ldots, x_{N-1}]$ the Discrete Fourier Transform for $k = 0, 1, \ldots, N-1$ is

$$X[k] = \sum_{n=0}^{N-1} x[n] \exp(-j2\pi nk/N) = \sum_{n=0}^{N-1} x[n] W_N^{-nk}$$

We made the usual substitution $W_N = e^{j2\pi/N}$ is the Nth root of unity, i.e., $W_N^{kN} = 1$ for all integer k. The X_k are known as *frequency coefficients*. The DFT converts a time signal into a frequency based signal.

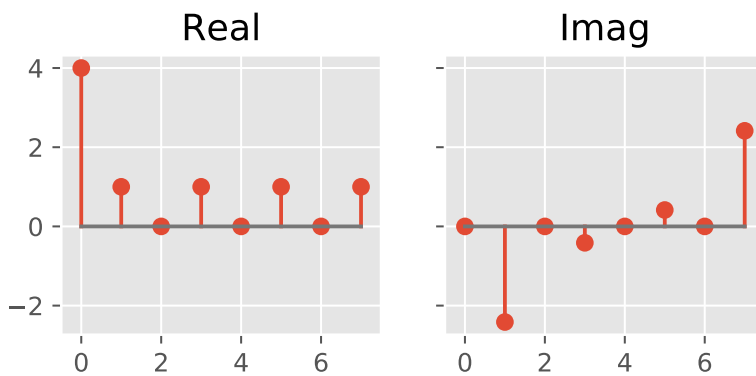
The inverse transform (IDFT) does the reverse. It takes a frequency based signal and returns a time signal. The interpretation of the inverse transform is important: it says an arbitrary length N time signal can be written as a sum of sinusoids:

$$x[n] = \frac{1}{N} \sum_{k=0}^{N-1} X[k] W_N^{nk}$$

Note the similarity between the DFT and the IDFT. The differences are in the sign of the exponent and the $(1/N)$ in the IDFT.

Here is a simple example of a short signal and its transform. The input `x` can be changed and the cell rerun to see how `y` changes.

```
In [2]: #vary x and see how y changes
        x = np.array([1,1,1,1,0,0,0,0]) + 1j*np.array([0,0,0,0,0,0,0,0])
        y = fft(x)
        fig, axs = plt.subplots(1,2,sharey=True,figsize=(5,2))
        axs[0].set_title('Real')
        axs[0].stem(np.real(y))
        axs[0].set_xlim(-0.5,7.5)
        axs[1].set_title('Imag')
        axs[1].stem(np.imag(y))
        axs[1].set_xlim([-0.5,7.5]);
```



```
In [3]: #numerical results
        x = np.arange(8)
        fft(x), ifft(fft(x))

Out[3]: (array([ 28.+0.j   ,  -4.+9.657j,  -4.+4.j   ,  -4.+1.657j,
                 -4.+0.j   ,  -4.-1.657j,  -4.-4.j   ,  -4.-9.657j]),
         array([ 0. +0.000e+00j,  1. -1.715e-15j,  2. +1.554e-15j,
                 3. +1.826e-15j,  4. +0.000e+00j,  5. +6.123e-17j,
                 6. -1.554e-15j,  7. -1.723e-16j]))
```

The final result is not exactly the same as the input (note the small imaginary components in the answer). This is an unfortunate result of the limited precision of floating point computations. The `np.allclose` function ignores these small errors.

```
In [4]: np.allclose(x, ifft(fft(x)))

Out[4]: True
```

We also set a numpy option to suppress printing of small numbers.

```
In [5]: np.set_printoptions(suppress=True) #don't print really small numbers
```

10.2 Matrix-Vector DFT Implementation

A straightforward (but, as we will see, slow) implementation of the DFT uses matrix and vector multiplication. Define the *Fourier* matrix $F = W_N^{-nk}$ for $n = 0, 1, \ldots, N-1$ and $k = 0, 1, \ldots, N-1$. Then the DFT is

$$X = Fx$$

Sometimes F is written as F_N to emphasize the size of the input and output vectors.

The IDFT is $F^{-1}x$. Using properties of the Fourier matrix, it can be shown $F^{-1} = F^*/N$ where F^* is the conjugate of F.

The fastest way in Python to define a matrix function is to use the `np.ogrid` function.

```
In [6]: r,c = np.ogrid[0:2,0:4]
        r*c

Out[6]: array([[0, 0, 0, 0],
               [0, 1, 2, 3]])

In [7]: N = 4
        rows, cols = np.ogrid[0:N,0:N]
        rows*cols

Out[7]: array([[0, 0, 0, 0],
               [0, 1, 2, 3],
               [0, 2, 4, 6],
               [0, 3, 6, 9]])
```

```
In [8]: F = np.exp(-2j*np.pi*rows*cols/N)
        F

Out[8]: array([[ 1.+0.j,  1.+0.j,  1.+0.j,  1.+0.j],
               [ 1.+0.j,  0.-1.j, -1.-0.j, -0.+1.j],
               [ 1.+0.j, -1.-0.j,  1.+0.j, -1.-0.j],
               [ 1.+0.j, -0.+1.j, -1.-0.j,  0.-1.j]])

In [9]: np.allclose(F,F.T)

Out[9]: True
```

Compute the DFT using matrix-vector multiplication. Calculate the inverse Fourier matrix and compute the IDFT.

```
In [10]: x = np.random.randn(N)
         X = F @ x  #DFT
         x, X

Out[10]: (array([ 1.264, -0.235, -1.065, -0.2  ]),
          array([-0.237+0.j   ,  2.329+0.035j,  0.633-0.j   ,
                  2.329-0.035j]))

In [11]: Finv = F.conj()/N
         np.allclose(F @ Finv, np.eye(N))

Out[11]: True

In [12]: xhat = Finv @ X #IDFT
         xhat

Out[12]: array([ 1.264+0.j, -0.235+0.j, -1.065+0.j, -0.200+0.j])
```

Note, `xhat != x` due to roundoff errors in the calculation, but `xhat` $\approx$ `x`.

```
In [13]: xhat == x, np.allclose(x, xhat)

Out[13]: (array([False, False, False, False], dtype=bool), True)
```

Even when the answer is known to be real, there may be nonzero imaginary parts due to roundoff error. It is common to take the real part of the answer.

```
In [14]: xhat.real

Out[14]: array([ 1.264, -0.235, -1.065, -0.2  ])
```

10.3 Development of the FFT Algorithm

The standard calculation of the DFT using a matrix-vector representation requires approximately N^2 complex multiplications and additions. Some of those multiplications are trivial (by ± 1 or $\pm j$) and can be eliminated, but, for large values of N, non-trivial multiplications dominate the calculation.

The Fast Fourier Transform (FFT) requires approximately $(N/2)\log(N)$ complex multiplications when N is a power of 2, i.e., when $N = 2^b$. The reduction of

computation time from N^2 to $(N/2)\log(N)$ can be dramatic. E.g., when $N = 1024$, the FFT reduces computation by a factor on the order of $N/\log_2(N) \approx 100$.

Below we describe the Cooley-Tukey Decimation in Time algorithm. It was first described in 1965 and is the most popular FFT algorithm. Other popular algorithms are the Decimation in Frequency algorithm and the Split-Radix algorithm.

In deriving the FFT, we use the following properties of W:

$$\begin{aligned} W_N^{2m} &= W_{N/2}^m \\ W_N^{N/2} &= -1 \\ W_N^{k+N/2} &= -W_N^k \end{aligned}$$

Assume, for the moment, $0 \le k < N/2$. Divide the data into odd and even segments and use the properties of W.

$$\begin{aligned} X[k] &= \sum_{n=0}^{N-1} x[n] W_N^{-kn} \\ &= \sum_{m=0}^{N/2-1} x[2m] W_N^{-2mk} + \sum_{m=0}^{N/2-1} x[2m+1] W_N^{-(2m+1)k} \\ &= \sum_{m=0}^{N/2-1} x[2m] W_{N/2}^{-mk} + W_N^{-k} \sum_{m=0}^{N/2-1} x[2m+1] W_{N/2}^{-mk} \end{aligned}$$

The first sum is the $N/2$ length DFT of the even sequence and the second sum is the $N/2$ length DFT of the odd sequence. The W_N^{-k} are known as "twiddle" factors.

Define even and odd signals as follows: $x_e[m] = x[2m]$ and $x_o[m] = x[2m+1]$, both for $m = 0, 1, \ldots, N/2-1$. Then, the kth term, for $k = 0, 1, \ldots, N/2-1$, of the DFT can be written as in matrix vector notation as

$$X[k] = \sum_{m=0}^{N/2-1} W_{N/2}^{-mk} x_e[m] + W_N^{-k} \sum_{m=0}^{N/2-1} W_{N/2}^{-mk} x_o[m]$$

Now consider $k > N/2 - 1$,

$$\begin{aligned} X[k + N/2] &= \sum_{m=0}^{N/2-1} W_{N/2}^{-m(k+N/2)} x_e[m] + W_N^{-k-N/2} \sum_{m=0}^{N/2-1} W_{N/2}^{-m(k+N/2)} x_o[m] \\ &= \sum_{m=0}^{N/2-1} W_{N/2}^{-mk} x_e[m] - W_N^{-k} \sum_{m=0}^{N/2-1} W_{N/2}^{-mk} x_o[m] \end{aligned}$$

Let $X_e[k]$ and $X_o[k]$ denote the kth terms of the DFT's of the even and odd sequences, respectively. Then, we get the basic "butterfly" operation of the FFT, for $k = 0, 1, \ldots, N/2 - 1$,

$$X[k] = X_e[k] + W_N^{-k} X_o[k]$$
$$X[k + N/2] = X_e[k] - W_N^{-k} X_o[k]$$

In the example code below, the $N = 8$ DFT is divided into two $N = 4$ DFTs and combined using the butterfly operation. First, we define a function to compute the Fourier matrix. Then, we implement a one stage butterfly and test the answer against the library `fft` function.

```
In [15]: def Fourier_matrix(N=4):
             """compute NxN Fourier matrix"""
             r,c = np.ogrid[0:N,0:N]
             return np.exp(-2j*np.pi*r*c/N)

In [16]: N=8
         x = np.random.randn(N) #random signal
         X = fft(x) #compare to standard library function

         F4 = Fourier_matrix(N=4)
         W8 = np.exp(-2j*np.pi*np.r_[:N/2]/N) #twiddle

         xe = x[::2]    #even sequence
         xo = x[1::2]   #odd sequence
         Xe = F4 @ xe   #DFT of even sequence
         Xo = F4 @ xo   #DFT of odd sequence
         W8Xo = W8 * Xo #mult by twiddle factor
         X8 = np.r_[Xe+W8Xo, Xe-W8Xo] #butterfly
         np.allclose(X,X8)

Out[16]: True
```


10.4 FFT Operation Count

The division into halves can be continued until $N = 1$, for which the DFT is trivial. In practice, the division usually stops at a convenient value, e.g., $N = 8$, at which point the direct calculation is used.

Let $T(N)$ be the number of complex multiplications required to compute an N point FFT. Then $T(N)$ satisfies the following equation:

$$T(N) = 2T(N/2) + N/2$$

The solution to this equation is $T(N) = cN \log_2(N)$,

$$\begin{aligned} cN\log_2(N) &= 2c(N/2)\log_2(N/2) + N/2 \\ &= cN\log_2(N) - cN + N/2 \end{aligned}$$

Clearly, $c = 1/2$.

Below, we generalize our one stage butterfly calculation to a recursive FFT implementation that uses the butterfly until $N = 256$ (determined empirically to be a good transition value), when we switch to the matrix-vector calculation.

```
In [17]: #precompute these matrices
         F256 = Fourier_matrix(N=256)
         F128 = Fourier_matrix(N=128)
         F64 = Fourier_matrix(N=64)

         def recursive_fft(x):
             """compute FFT(x) using recursive algorithm

             assumes N=len(x) is a power of 2"""
             N = len(x)
             assert(int(np.log2(N)) == np.log2(N)) #N=2**b
             if N > 256:
                 xe, xo = x[::2], x[1::2]
                 Wk = np.exp(-2j*np.pi*np.r_[0:N/2]/N)
                 Xe = recursive_fft(xe)
                 WXo = Wk*recursive_fft(xo)
                 return np.r_[Xe+WXo,Xe-WXo]
             elif N == 256:
                 return F256 @ x
             elif N == 128:
                 return F128 @ x
             elif N == 64:
                 return F64 @ x
             else:
                 F = Fourier_matrix(N)
                 return F @ x
```

Test the function with a simple example. Then test on a long random input.

```
In [18]: XR = recursive_fft([1,1,1,1,0,0,0,0])
         X = fft([1,1,1,1,0,0,0,0])
         print(np.allclose(X,XR))
```

```
True
```

```
In [19]: x1024 = np.random.randn(1024)
         X1024 = fft(x1024)
         XR1024 = recursive_fft(x1024)
         np.allclose(X1024, XR1024)
```

```
Out[19]: True
```

10.5 Timing the Various DFT implementations.

The matrix-vector DFT method requires approximately N^2 operations. The FFT requires approximately $(N/2)\log_2(N)$ operations. The library `fft` function has been carefully written to minimize its computation time. It is implemented in a compiled language. It is the fastest of the three implementations when N is large.

There are several ways to time the execution of Python programs. The notebook permits a simple timing mechanism. (Recall, the assignment `_ = fft(x)` supresses the printing of the output.)

```
In [20]: %time _ = fft(x1024)

CPU times: user 265 µs, sys: 124 µs, total: 389 µs
Wall time: 380 µs


In [21]: %time _ = recursive_fft(x1024)

CPU times: user 1.77 ms, sys: 870 µs, total: 2.64 ms
Wall time: 4.58 ms
```

In normal Python code, the simplest way is to use the `time` library. It returns the "wall clock" time.

```
In [22]: import time
         F1024 = Fourier_matrix(N=1024)
         t0 = time.time()
         fft(x1024)
         t1 = time.time()
         recursive_fft(x1024)
         t2 = time.time()
         F1024 @ x1024
         t3 = time.time()
         print("%.3e %.3e %.3e" % (t1-t0, t2-t1, t3-t2))

2.048e-04 8.152e-04 2.463e-03
```

Unfortunately, "wall clock" measurements are not always accurate. If the computer does something else (switches to a different task) while doing the calculation, the returned time may not be accurate. The best way to measure execution time is to use the `timeit` library. Below, we show a simple way to time the `fft` library function. By default, `timeit.Timer` repeats the command three times, with each repeat running the command `number=10` times. The best estimate is usually the minimum execution time.

```
In [23]: number = 10
         times = timeit.Timer('fft(x1024)',
                              globals=globals()).repeat(number=number)
         times, np.min(times)/number
```

```
Out[23]: ([0.002, 0.001, 0.001], 0.000)
```

The notebook has a "line magic" function to make calling the `timeit` library easier.

```
In [24]: t = %timeit -o -n1 -r5 fft(x1024)
```

```
The slowest run took 4.09 times longer than the fastest. This could mean that an ir
196 µs ± 134 µs per loop (mean ± std. dev. of 5 runs, 1 loop each)
```

Below we run a short script to estimate the time required for the three implementations for various values of N. Note, the matrix-vector method is much slower than the FFT methods for large N and, therefore, we limit the size of N for which we run the slower matrix-vector method.

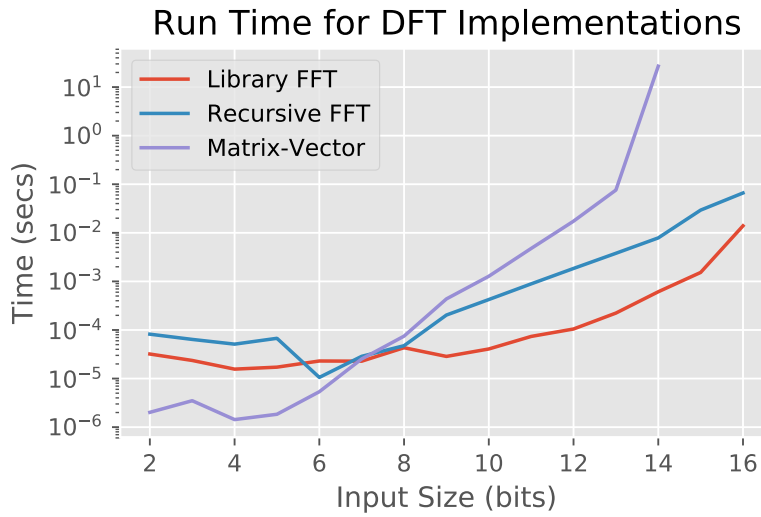
Because the timing runs take awhile, we use two cells. The first runs the various DFT algorithms; the second cell plots the answer. This way the plotting parameters can be iterated until we get the plot just right.

```
In [25]: bmax, dbmax = 16, 14
         ffts, rffts, dfts = [], [], []
         for b in range(2,bmax+1):
             N = 2**b
             x = np.random.randn(N)

             #library fft
             t = %timeit -o -q -n1 -r5 fft(x)
             ffts.append(t.best)

             #recursive fft
             t = %timeit -o -q -n1 -r5 recursive_fft(x)
             rffts.append(t.best)

             #matrix DFT
             if b <= dbmax:
                 F = Fourier_matrix(N)
                 t = %timeit -o -q -n1 -r5 F@x
                 dfts.append(t.best)
```

Now that we have computed the times, plot them. Dividing the task allows us to tweak the plot a few times without having to recompute the numbers.

```
In [26]: bits = range(2,bmax+1)
         fbits = range(2,bmax+1)
         dbits = range(2,dbmax+1)
```

```
plt.figure(figsize=(5,3))
plt.semilogy(fbits,ffts,label='Library FFT')
plt.semilogy(fbits,rffts,label='Recursive FFT')
plt.semilogy(dbits,dfts,label='Matrix-Vector')
plt.legend()
plt.xlabel('Input Size (bits)')
plt.ylabel('Time (secs)')
plt.title('Run Time for DFT Implementations');
```



The matrix-vector implementation time rises much faster than either of the FFT implementations. The libary `fft` function is about 10 times faster than the recursive implementation, but both FFT implementations have the same asymptotic behavior (same slope in the graph). (That the recursive python implementation is only ten times slower than the library `fft` function is actually impressive. The library function unwinds the recursion, is carefully optimized to eliminate unnecessary calculations, and is written in a compiled language. Also, the recursive implementation took only a few minutes to write.)

In practice, use the library `fft` and `ifft` functions. They are fast and well tested.

10.6 Convolution with FFTs

When the sizes of x and h are large, it is faster to compute convolutions with FFTs. If $X[k]$ is the DFT of $x[n]$ and $H[k]$ is the DFT of $h[n]$, then $Y[k] = X[k]H[k]$ is the DFT of $y[n] = x[n] \circledast h[n]$, where $\circledast$ denotes circular convolution. The circular

convolution is equivalent to the (normal) linear convolution if the FFT size N at least as big as the length of y. If that is true, $y = IDFT(DFT(x)DFT(h))$.

The steps are to choose an FFT size N such that $N \geq \text{len(y)}$, pad x and h with zeros to size N, compute DFTs of the padded x and h, multiply the resulting DFT coefficients together, then compute the IDFT.

The straightforward calculation of the convolution requires approximately N^2 operations (each value of x is multiplied by each value of h), but the FFT based calculation requires on the order of $N \log_2(N)$ operations. Each DFT requires $(N/2)\log(N)$ complex multiplications, multiplying $X[k]$ by $H[k]$ requires N operations, and the IDFT requires $(N/2)\log_2(N)$ operations.

For large N, the FFT calculation can be much faster than the direct method.

```
In [27]: x = np.random.randn(6000)
         h = np.arange(6000)
         lx, lh = len(x), len(h)
         ly = lx+lh-1
         N = 2**np.ceil(np.log2(ly)).astype('int')
         N
```

```
Out[27]: 16384
```

```
In [28]: %%time
         y = np.convolve(h,x)
```

```
CPU times: user 12.1 ms, sys: 757 µs, total: 12.9 ms
Wall time: 14.2 ms
```

```
In [29]: %%time
         xz = np.r_[x,np.zeros(N-lx)] #zero pad
         hz = np.r_[h,np.zeros(N-lh)]
         Xz = fft(xz)
         Hz = fft(hz)
         Yz = Hz*Xz
         yz = ifft(Yz)
```

```
CPU times: user 3.69 ms, sys: 3.35 ms, total: 7.04 ms
Wall time: 9.81 ms
```

```
In [30]: np.allclose(yz[:ly], y)
```

```
Out[30]: True
```


10.7 DFT and FFT Summary

The DFT converts a time domain signal to a frequency representation. The FFT is a faster implementation. Its speedup can be dramatic for large N.

The FFT is used in frequency and spectral estimation and computing convolutions. Since convolutions are related to polynomials, FFTs are also used in multiplying large polynomials together. For example, the algorithms which compute $\pi = 3.1415...$ to millions of digits use the FFT to multiply large numbers together.

11

Frequency and Spectral Analysis

A longstanding problem in science is to represent a signal by its frequency components. In 1672, Isaac Newton used a prism to separate white light into a rainbow of colors. In 1800, William Herschel discovered the sun's infrared radiation. Joseph Fraunhofer discovered dark lines in the sun's spectrum. These discoveries led to the development of the field of spectroscopy. Spectroscopy is used to measure chemical compositions of the sun, stars, and planets, and was crucial to demonstrating the atomic nature of matter, to name only two of its many successes.

In signal processing, frequency techniques are universally used to analyze signals. Human hearts beat about once per second, people hear from about 20 Hz to 20000 Hz, the note A above middle C is 440 Hz, bats emit chirp (variable frequency) signals in the ultrasonic band.

A stationary signal's *spectrum* is its energy versus frequency. In *spectrum estimation* the signal's energy is estimated. First we estimate energy versus frequency for stationary signals. Then we estimate energy versus frequency versus time for time-varying signals.

Let $x(n)$ be a discrete time signal. The Discrete Time Fourier Transform (DTFT) of $x(n)$ is

$$X(\Omega) = \sum_{n=-\infty}^{\infty} x(n) e^{-jn\Omega}$$

If $x(n) = e^{j\omega n}$, then $X(\Omega) = \delta(\Omega - \omega)$. Thus, each sinusoid corresponds to two delta functions in the DTFT, one at $\Omega = \omega$ and one at $\Omega = -\omega$.

Let $X(k)$ be the DFT of an N point sequence $x(n)$. The DFT differs from the DTFT due to the finite number of samples (and the DFT outputs a discrete spectrum

while the DTFT is a continuous spectrum). Define $w(n) = 1$ for $n = 0, 1, \ldots, N-1$ and $w(n) = 0$ otherwise. Then the finite sequence $x(n)$ corresponds to the infinite sequence $x(n)w(n)$. The DTFT of $x(n)w(n)$ is the circular convolution of $X(\Omega)$ and $W(\Omega)$. As we discuss below, $w(n)$ is known as a *window* sequence. Different window functions have different trade-offs in spectral estimation.

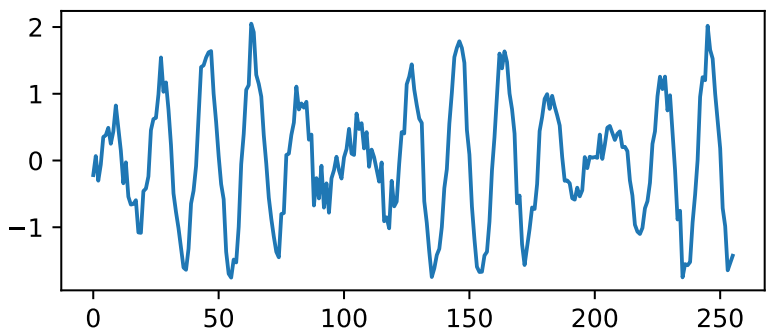
Below we import the `scipy.signal` library and the `fft` and `ifft` functions from the `scipy.fftpack` library. (The `scipy` version of `fft` is twice as fast as the `numpy` version for real inputs.)

```
In [1]: import numpy as np
        import matplotlib.pyplot as plt
        import scipy.signal as sig
        from scipy.fftpack import fft, ifft
        %matplotlib inline
```

11.1 Sinusoids in Noise

Our example signal consists of two sinusoids corrupted by additive Gaussian noise.

```
In [2]: N=256
        f1, f2 = 0.05, 0.06 #cycles per sample
        a1, a2 = 1, -0.7 #amplitudes
        sigma = 0.2 #strength of noise
        t = np.arange(N)
        x = a1*np.sin(2*np.pi*f1*t) + \
            a2*np.sin(2*np.pi*f2*t) + \
            sigma*np.random.randn(N)
        plt.plot(t,x);
```



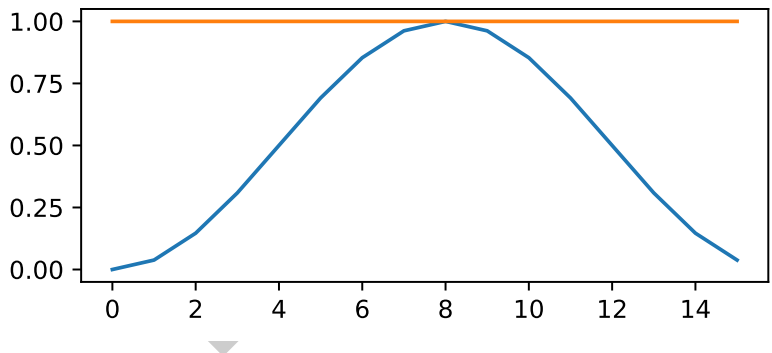
11.2 Window Functions

We consider two window functions: the *boxcar* (or *rectangular*) and the *Hann* window. The boxcar window is the one discussed above, $w(n) = 1$ for $n = 0, 1, \ldots, N-1$ and $w(n) = 0$ otherwise. The Hann window (like many other windows) weights center samples more heavily than first and last samples.

It is common for newcomers to spectrum estimation to wonder why use a window function at all. The answer is that window functions are unavoidable. The boxcar window function represents the finite nature of the data sequence. The Hann and other windows are introduced to compensate for the problems caused by the boxcar window.

Below we use a small value of N to illustrate the two windows. The Hann window has the values $w(n) = 0.5\big(1 - \cos(2\pi n/(N-1))\big)$.

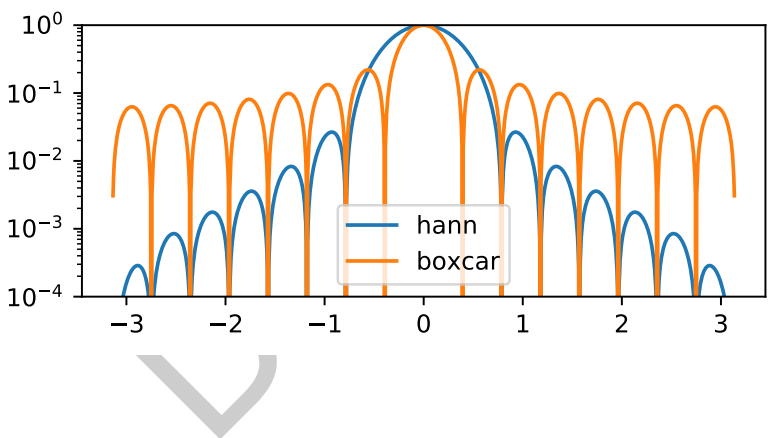
```
In [12]: N=16
         hann = sig.hann(N,sym=False) #sym=False for spectral estimation
         boxcar = sig.boxcar(N)
         plt.plot(hann)
         plt.plot(boxcar);
```



Below we calculate and plot the Fourier transform of both windows. The `scale_window` function normalizes the transform to a maximum value of 1 and presents both the negative and positive frequency portions for the plot.

The plot below illustrates the two most important features in window functions. The *main lobe* is the central (and highest) portion of the curve. The *side lobes* are the other peaks. The boxcar window has a narrow main lobe and high side lobes, while the Hann window has a wider main lobe and smaller side lobes. The main lobe determines the width of any sinusoidal components (narrower is better) and the side lobes (lower is better) determine how much energy *leaks* from the sinusoidal components into other frequencies and also how much noise is present in the final estimates. Thus the boxcar window is better at representing the sinusiodal components but is worse everywhere else.

```
In [15]: def scale_window(N,window='hann'):
             """scale window for double sided plots"""
             h = sig.get_window(window,N)
             w,p = sig.freqz(h)
             p /= np.max(np.abs(p))
             w = np.r_[-w[::-1],w[1:]]
             p = np.r_[p[::-1],p[1:]]
             return w,p

         N = 16
         w,p = scale_window(N,'hann')
         plt.semilogy(w,np.abs(p),label='hann')
         w,p = scale_window(N,'boxcar')
         plt.semilogy(w,np.abs(p),label='boxcar')
         plt.legend()
         plt.ylim(1e-4,1);
```



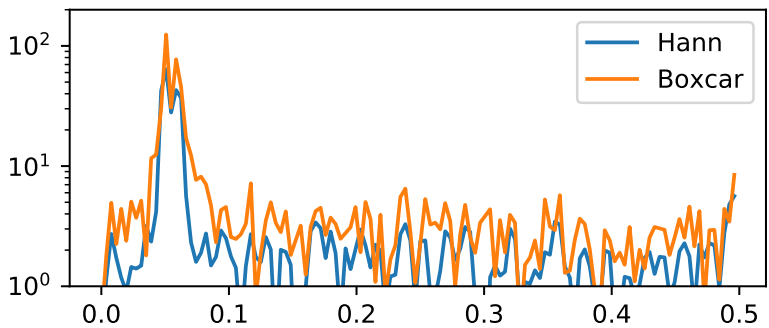
11.3 Periodogram Spectral Estimate

The *periodogram* spectral estimate consists of three steps:

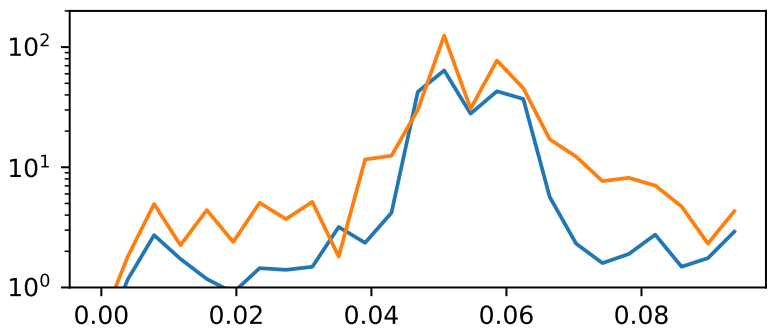
1. Multiply $x(n)$ by a suitably chosen window function (or use the default boxcar window).
2. Compute the DFT of $y(n) = x(n)w(n)$.
3. The final estimate is $|Y(k)|^2$.

In the example below, we see the boxcar window produces better peaks but has much higher noise everywhere else. In the zoomed in plot, we see the wider spread around the sinusoid peaks in the boxcar plot than in the Hann plot.

```
In [5]: N=len(x)
        f = np.linspace(0,1,N,endpoint=False)
        w = sig.hann(N, sym = False)
        Xw = fft(x*w)
        plt.semilogy(f[:N//2], np.abs(Xw[:N//2]), label='Hann')
        X = fft(x)
        plt.semilogy(f[:N//2], np.abs(X[:N//2]), label='Boxcar')
        plt.ylim(1,200)
        plt.legend();
```




```
In [6]: #zoom in on sinusoid peaks
        plt.semilogy(f[:N//10], np.abs(Xw[:N//10]))
        plt.semilogy(f[:N//10], np.abs(X[:N//10]))
        plt.ylim(1,200);
```

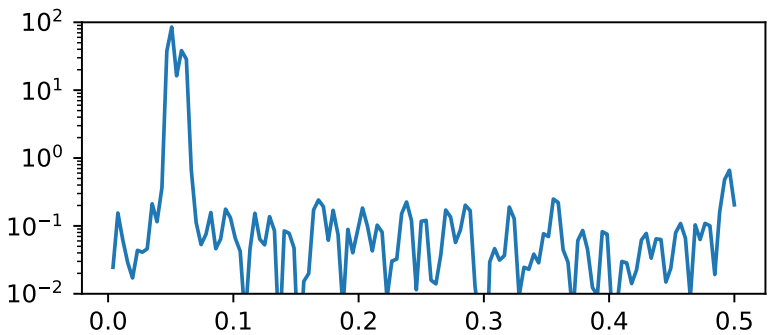


11.4 The Welch Spectrum Estimator

The Welch estimator trades-off precision of the sinusoidal components for lower noise elsewhere. The periodogram produces the same number of output points as input point and, hence, does not benefit from noise averaging. The Welch estimator divides the original sequence into smaller, possibly overlapping, sequences, computes a periodogram estimate of each sequence, and then averages the periodogram estimates.

Below we use the `scipy.signal.welch` function to compute a spectral estimate. If the data sequence is long enough, the Welch estimator is a good choice.

```
In [7]: f,w = sig.welch(x)
        plt.semilogy(f[1:],w[1:])
        plt.ylim([1e-2,1e2]);
```



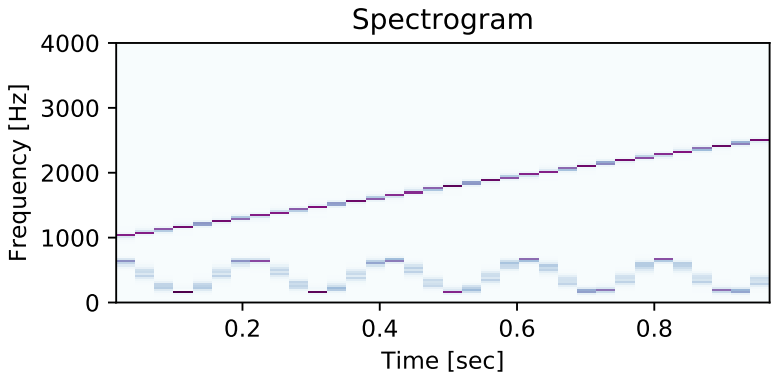
11.5 Spectrogram for Time-Varying Signals

Spectrum estimation methods designed for stationary signals perform poorly on time-varying signals. The *spectrogram* is visual technique for representing frequency versus time information for time-varying signals. The spectrogram breaks the signal into short segments (possibly overlapping), computes a periodogram estimate of each, and plots the estimated energy on the y-axis. The resulting picture shows signal energy per frequency versus time. Spectrograms are often used in voice signal analysis.

As an example, we build a time-varying signal consisting of an FM modulated sinusoid (as might appear in a music synthesizer) and a chirp signal corrupted by some Gaussian noise. We use the `scipy.signal.spectrogram` estimator.

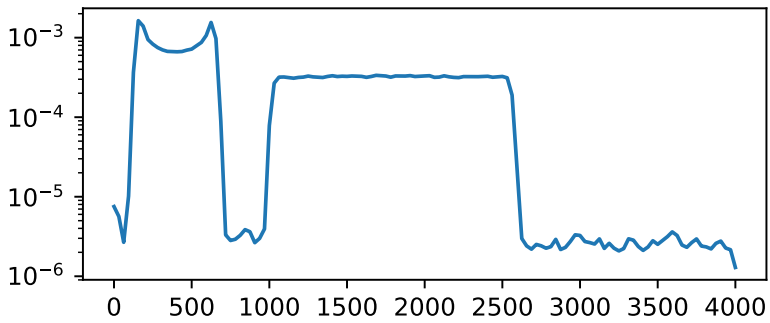
The plot below shows the two components. The chirp is the upward sloping line, showing the increasing frequency. The FM sinusoid is the sinusoid-like part at the bottom.

```
In [10]: Fs = 8000
         t = np.linspace(0,1,Fs+1)
         FMmod = np.cos(2*np.pi*400*t + 50*np.sin(2*np.pi*5*t))
         chirp = np.cos(2*np.pi*1000*t + 5000*t**2)
         noise = 0.1*np.random.randn(len(t))
         x = FMmod + chirp + noise
         f, times, Sxx = sig.spectrogram(x, Fs)
         plt.pcolormesh(times, f, Sxx,cmap='BuPu')
         plt.ylabel('Frequency [Hz]')
         plt.xlabel('Time [sec]')
         plt.title('Spectrogram')
         plt.show()
```



By comparison, below is the Welch spectrum estimator (which assumes the signal is stationary). It indicates signal energy centered around 440 Hz and between about 1000 Hz and 2500 Hz, but does not indicate the time dependence of these components.

```
In [9]: f,w = sig.welch(x,Fs)
        plt.semilogy(f,np.abs(w));
```



11.6 Summary

Spectral estimation is an important problem in many areas of science and engineering. There are numerous methods. In this chapter, we presented the most popular estimate, the periodogram, and a related estimator, the Welch estimator. For time-varying signals we show the spectrogram.

12

Fourier Series

Periodic signals repeat, again and again, running through the same values. The most famous (and important) repeating functions are sinusoids, but many other signals repeat, at least for portions. For instance, musical notes repeat for short durations. In this chapter, we consider *Fourier series*, a technique for representing or approximating periodic signals by a sum of sinusoids.

A periodic signal repeats with period $T > 0$ if $x(t) = x(t+T)$ for all t. Normally, T is chosen as the minimum value of T such at $x(t) = x(t+T)$. (If $x(t) = x(t+T)$, then $x(t) = x(t+T) = x(t+2T) = x(t+3T)$, etc.)

The *Fourier series* of $x(t)$ is the following:

$$x(t) = \sum_{k=0}^{\infty} a_k \cos(2\pi kt) + b_k \sin(2\pi kt)$$

The technical conditions allowing equality above are beyond this text. Instead, we focus on an approximation:

$$x(t) \approx \hat{x} = \sum_{k=0}^{K-1} a_k \cos(2\pi kt) + b_k \sin(2\pi kt)$$

Note, for $k = 0$, $\cos(2\pi kt) = 1$ and $\sin(2\pi kt) = 0$. Thus, b_0 is irrelevant and is usually taken to be 0.

The a_k and b_k can be chosen to minimize the squared error between $x(t)$ and $\hat{x}(t)$.

$$Q\big(x(t), \hat{x}(t)\big) = \int_0^T \big(x(t) - \hat{x}(t)\big)^2 dt$$

The optimal values of a_k and b_k can be found by taking derivatives of Q with respect to a_l or b_m and setting the derivatives to 0. The resulting equations are known as the *normal equations*. Since cos() and sin() are orthogonal on the interval $(0, T)$, the normal equations simplify to the following:

$$\begin{aligned} a_0 &= \frac{1}{T}\int_0^T x(t)dt \\ a_k &= \frac{2}{T}\int_0^T x(t)\cos(2\pi kt)dt \\ b_0 &= 0 \\ b_k &= \frac{2}{T}\int_0^T x(t)\sin(2\pi kt)dt \end{aligned}$$

For many known functions $x(t)$, the integrals can be computed exactly (e.g., using `sympy`). In this chapter we focus on numerical techniques for computing the Fourier coefficients.

In addition to the usual imports, we import `wavfile` (to read in a sound file) and `Audio` to output an audio player to the notebook.

```
In [1]: import numpy as np
        import matplotlib.pyplot as plt
        %matplotlib inline
        from scipy.io import wavfile
        import scipy.signal as sig
        from scipy.fftpack import fft, ifft
        from IPython.display import Audio
        np.set_printoptions(precision=3, suppress=True)
```

12.1 Numerical Calculation of Fourier Coefficents

When $x(t)$ is only known through its samples, $x(n)$, the integrals must be computed numerically. In the function below, the integrals are approximated by the `np.average` function. More complicated integral functions, such as the trapezoidal and Simpson's rules, are unnecessary due to the repeating nature of the signal.

```
In [2]: def fourier_coeffs(x, K=1):
            """numerically calculate fourier series coefficients
            inputs:
                x = array with 1 period of signal
                K = number of Fourier coeffs
```

```
        periods = number of periods in input
    returns:
        aks = cosine coeffs
        bks = sine coeffs"""
    omega = np.linspace(0,2*np.pi,len(x))
    aks, bks = [np.average(x)], [0] #k=0 values
    for k in range(1,K):
        ck,sk = np.cos(omega*k), np.sin(omega*k)
        aks.append(2*np.average(x*ck))
        bks.append(2*np.average(x*sk))
    return np.array(aks), np.array(bks)
```

The `fourier_approx` function creates $\hat{x}(t)$ from the calculated Fourier coefficients.

```
In [3]: def fourier_approx(aks, bks, nmax, periods=1):
            """calculate fourier series approximation"""
            n = np.linspace(0,periods*2*np.pi,nmax)
            xhat = np.zeros(nmax)
            for k,(ak,bk) in enumerate(zip(aks,bks)):
                xhat += ak*np.cos(k*n) + bk*np.sin(k*n)
            return xhat
```

12.2 Square Wave

The first example is a square wave, which alternates between -1 and 1. The Fourier coefficients are known to be the following: $a_k = 0$, $b_k = 4/(k\pi)$ for k odd, and $b_k = 0$ for k even.

Below $L = 10000$ is chosen to be large so the numerical calculation agrees closely with the theoretical. Rather than print out b_k directly, we scale the calculated numbers by $k\pi/4$. The scaled numbers should be (close to) 0 and 1.

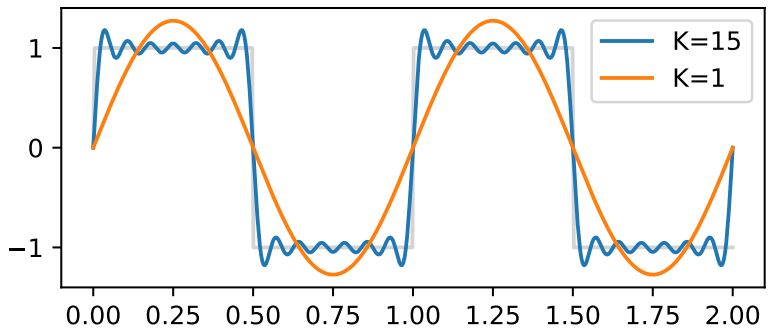
```
In [24]: L = 10000
         sq_wave = np.r_[np.ones(L//2), -np.ones(L//2)]
         aks, bks = fourier_coeffs(sq_wave, K=15)
         print(aks)
         print(bks*np.pi*np.arange(len(bks))/4)
```

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

The numerical calculations agree with the theoretical values. This lends some confidence that the `fourier_coeffs` function is correct. (More extensive tests should be done before trusting it completely.)

Below we plot approximations to the square wave. The $K = 1$ approximation is a single sinusoid. It is known as the *fundamental*. The $K = 15$ approximation is a sum of sinusoids ($k = 1, 3, 5, \ldots, 13$). It is starting to look like a square wave. The overshoot and undershoot at the edges (discontinuities) are known as Gibbs phenomena. Gibbs phenomena remain as long as the number of terms is finite.

```
In [27]: nmax, periods = 500, 2
         n = np.linspace(0,periods,nmax)
         sq_wave = np.sign(np.sin(2*np.pi*n))
         plt.plot(n,sq_wave,'lightgrey')
         sq_wave_hat = fourier_approx(aks, bks, nmax, 2)
         plt.plot(n,sq_wave_hat,label='K=15')
         fundamental = 4*np.sin(2*np.pi*n)/np.pi
         plt.plot(n,fundamental,label='K=1')
         plt.legend();
```



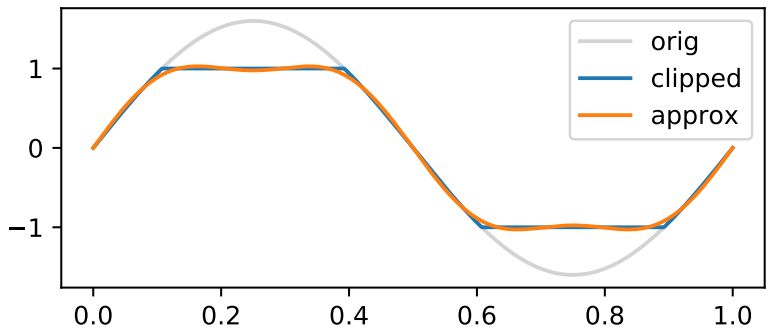
12.3 Clipped Sinusoid

An interesting example is the *clipped sinusoid*. Musicians sometimes clip their music because doing so results in more complicated sounds.

The code below plots three signals, the original sinusoid, the clipped sinusoid, and a Fourier series approximation using $k = 1$ (the fundamental) and $k = 3$ and $k = 5$ (third and fifth harmonics). Notice, clipping results in significant power in the third harmonic (and some in the fifth).

```
In [6]: nmax = 501
        n = np.linspace(0,1,nmax)
        orig = 1.6*np.sin(2*np.pi*n)
        clipped = np.clip(orig, -1, 1)
        plt.plot(n,orig,'lightgrey',label='orig')
        plt.plot(n,clipped,label='clipped')
        aks, bks = fourier_coeffs(clipped, K=6)
        xhat = fourier_approx(aks,bks,nmax)
        plt.plot(n, xhat, label='approx')
        plt.legend()
        print(bks)
```

```
[ 0.     1.182  0.     0.201  0.    -0.005]
```



12.4 Analysis of Vowel Sound

Vowel sounds are quasi-periodic, $x(t) \approx x(t+T)$. The example below is from a recording of my voice saying "aahh" for about two seconds. The sound was recorded using the microphone on my laptop (which likely is weak in low frequency response) using the free software *Audacity* (single channel, sampling rate = 8000 Hz).

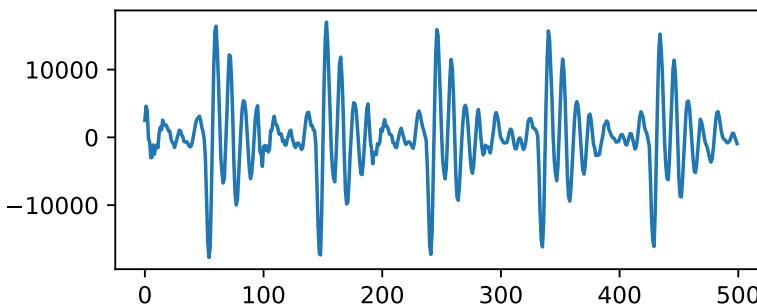
The examples below can be replicated with other voices and other vowel sounds.

The commands below read in the sound, set up the notebook `Audio` player, and then plots a short portion of the signal. Notice how complicated the repeating pattern is.

```
In [7]: Fs, aahh = wavfile.read('aahh.wav') #my voice
        Audio(aahh, rate=Fs)
```

```
Out[7]: <IPython.lib.display.Audio object>
```

```
In [8]: plt.plot(aahh[:500]);
```



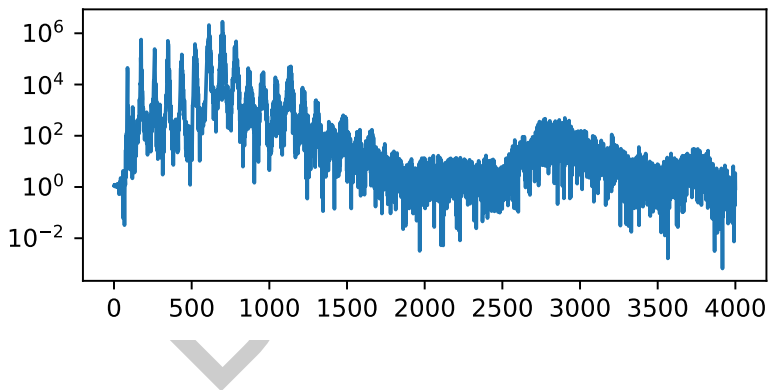
```
In [9]: aahh.size
```

```
Out[9]: 19249
```

The sound file has 19249 samples. The volume decreases slightly toward the end. So we will use the first 8192 samples to estimate the frequency content.

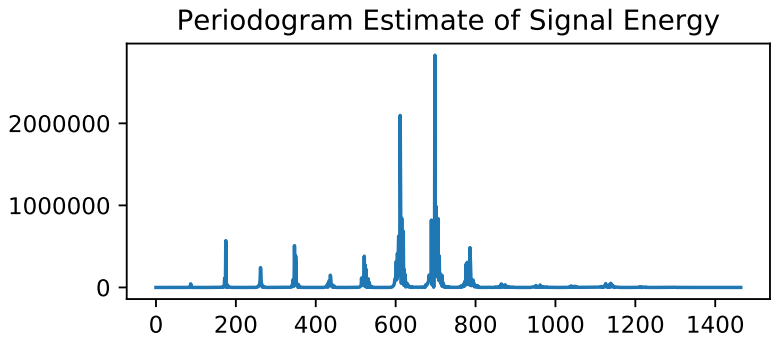
The signal is recorded at 8000 samples per second. The frequency variable is quantized to $8000/8192 = 0.98 \approx 1$ Hz per sample. The `periodogram` function calculates an estimate of the signal's spectrum. Notice two things: Most of the energy is in the frequencies below about 1500 Hz, and the "comb-like" structure indicates periodicity (sums of sinusoids at harmonic frequencies).

```
In [10]: x = aahh[:8192]
         N = len(x)
         f, Pxx = sig.periodogram(x, fs=8000)
         plt.semilogy(f,Pxx);
```



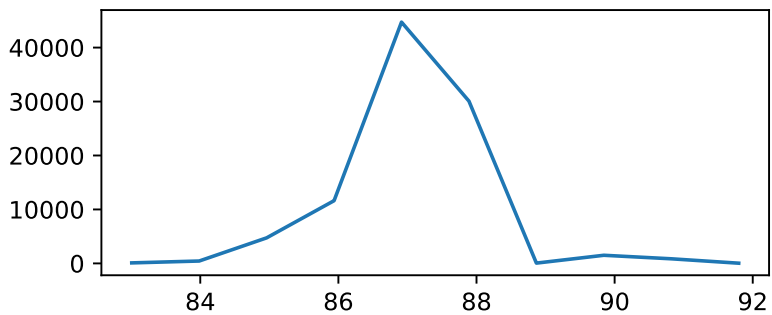
Zoom in on the low 1500 Hz. Curiously, the highest energy is in the seventh and eighth harmonics (indicating complex vibrations in the vocal tract).

```
In [20]: plt.plot(f[:1500],Pxx[:1500])
         plt.title('Periodogram Estimate of Signal Energy');
```



Zoom in on the fundamental. It is about 87 Hz, which corresponds to a length of about $L \approx 8000/87 = 92$ samples per period. The vowel signal repeats approximately every 92 samples.

```
In [12]: plt.plot(f[85:95],Pxx[85:95]); #fundamental = 87 Hz
```




```
In [13]: 8000/87 #length per period
```

```
Out[13]: 91.95402298850574
```

Scipy includes a function `signal.find_peaks_cwk` that helps study the periodicity of the signal. The function smooths the signal (with a wavelet filter) and applies a nonlinear peak finding algorithm.

It takes a bit of trial and error to get the arguments right (`range(10,100)`). The code below finds peaks and prints out the first few. `np.diff` computes the difference between successive values of its input. The differences are in the low 90's.

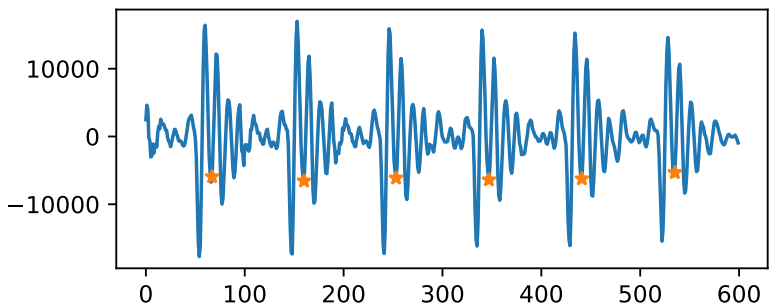
```
In [14]: peaks = sig.find_peaks_cwt(x,range(10,100))
         peaks[:8], np.diff(peaks[:9])

Out[14]: (array([ 67, 160, 253, 347, 441, 535, 628, 721]),
          array([93, 93, 94, 94, 94, 93, 93, 93]))
```

`np.bincount` counts the number of occurrences of each value. The values not shown below are 0; the nonzero values are displayed below. The most frequent length is 92, with 91 and 93 second and third, respectively.

```
In [15]: np.bincount(np.diff(peaks))[90:]

Out[15]: array([ 6, 27, 30, 21,  3])
```

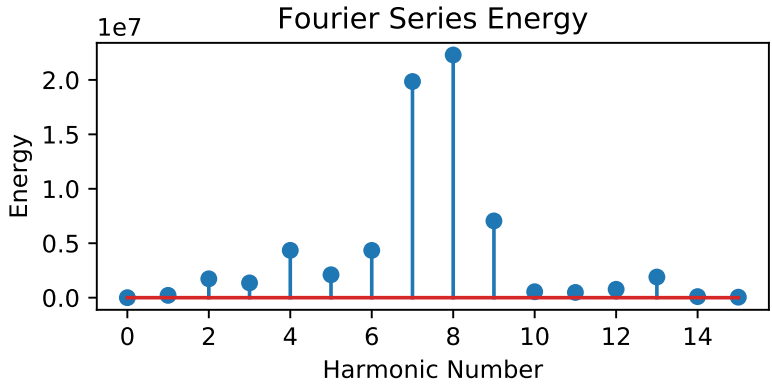
As a check, the located peaks are plotted against the original signal. The "peaks" are located in the middle of the signal, not at the highest points.

```
In [16]: plt.plot(x[0:600])
         plt.plot(peaks[:6],x[peaks[:6]],'*');
```



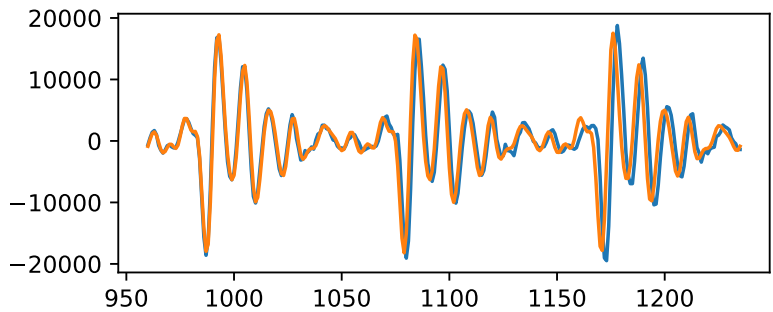
Take a period starting at time 960 (an arbitrary value) of length 92 and compute Fourier coefficients. Plot the energy (squared value) of each Fourier component (harmonics). The seventh and eighth harmonics have the most energy. This plot looks similar to the periodogram of the original signal (as it should).

```
In [28]: L, start = 92, 960
         sample = x[start:start+L]
         aks, bks = fourier_coeffs(sample,K=16)
         plt.title('Fourier Series Energy')
         plt.xlabel('Harmonic Number')
         plt.ylabel('Energy')
         plt.stem(aks**2 + bks**2);
```



Below we plot the Fourier series approximate signal against the original. The first period was used to generate the Fourier coefficients. Even though the signal is only quasi-periodic, the approximate signal closely fits the next two periods.

```
In [18]: xhat = fourier_approx(aks, bks, 3*92, 3)
         n = np.arange(start,start+3*92)
         plt.plot(n,x[start:start+3*92])
         plt.plot(n,xhat);
```



Generate a long approximation. The sound is more realistic if some noise is added.

```
In [19]: xhat = fourier_approx(aks,bks,100*92,100)
         xhat += 1500*np.random.rand(len(xhat))
         Audio(xhat,rate=Fs)
```

```
Out[19]: <IPython.lib.display.Audio object>
```

Many high performance audio compression algorithms employ similar methods to the example here. Using 31 Fourier coefficients (16 a_k's and 15 b_k's), we created an approximation of length 9200 samples to the original signal, i.e., 9200 samples are approximated by 31 Fourier coefficients. Other vowel sounds can be treated similarly.

12.5 Summary

Periodic signals can be written as sums of sinusoids. The Fourier coefficients are computed as integrals of the signal multiplied by the sinusoid. These integrals can be computed numerically and approximate signals generated. One application for these approximations is in voice coding.

13

Let's Make Music

We use signal processing techniques to make a simple music synthesizer and discuss some of the ways it can be improved.

Import `Audio` (for putting an audio widget in the notebook), `Image` (for displaying an image within the notebook), and `scipy.signal` (for filtering).

```
In [1]: import numpy as np
        import matplotlib.pyplot as plt
        plt.style.use('ggplot')
        from IPython.display import Audio, Image
        import scipy.signal as sig
        %matplotlib inline
```

13.1 Generate an Audio Sinusoid

The standard sampling rate for telephone quality audio is $F_s = 8000$ samples per second. Other sampling rates are used in other applications. For instance, CD quality audio uses 44,100 samples per second. High quality audio recording uses 48,000 or 96,000 samples per second.

According to Nyquist theory, the highest frequency that can be represented is half the sampling rate. In this notebook, we use `Fs=22050`, half the CD rate and allowing signal frequencies up to about 11 KHz, but other rates can be used.

The `Audio` command puts an audio widget in the notebook.

```
In [2]: Fs = 22050
        f = 440 #Hz, A above middle C
        t = np.linspace(0,1,Fs)
```

```
        sound = np.sin(2*np.pi*f*t)
        Audio(sound,rate=Fs)
```

```
Out[2]: <IPython.lib.display.Audio object>
```

Music synthesizers use signal processing techniques to mimic the sound of traditional instruments or to create new sounds not heard from traditional instruments.

This chapter builds a simple synthesizer to play "Mary Had A Little Lamb" and offers some suggestions for improvements.

13.2 Music Theory

Music frequencies are exponential. Each octave is a doubling in frequency. For example, note A is 440 Hz in the standard middle octave. In general, A can be 55 Hz, 110 Hz, 220 Hz, 440 Hz, 880 Hz, etc., depending on which octave is chosen, $A = 440 \times 2^{m-4}$ where m is octave number. The standard middle octave is $m = 4$.

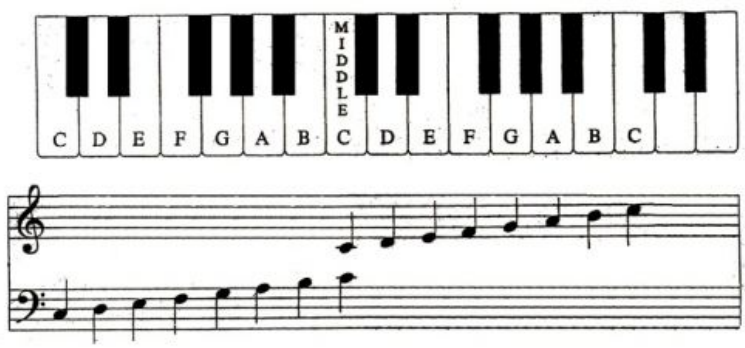
The standard music scale in western music divides each octave into twelve *semitones*). The frequencies of successive semitones are in ratios of $2^{1/12} = 1.06$. There are twelve semitones per octave, $(2^{1/12})^{12} = 2$.

The major notes (white piano keys) are C, D, E, F, G, A, B, C5 (C5 is note C in the next octave.) The MIDI note number for A above middle C is $n = 69$. The minor notes (black piano keys) are the missing values: 61, 63, 66, 68, 70. Some notes are one semitone apart (adjacent keys on the piano) while other notes are separated by two semitones.

```
In [3]: C, D, E, F, G, A, B, C5 = 60, 62, 64, 65, 67, 69, 71, 72
        scale = [C, D, E, F, G, A, B, C5]
```

```
In [4]: #from Piano-Keyboard-Guide.com
        Image('piano_keyboard_picture.jpg')
```

```
Out[4]:
```



```
In [5]: #play A above Middle C
        Fs = 22050 #samplerate
        dur = 1 #note duration in seconds
        t = np.linspace(0,dur,dur*Fs,endpoint=False)
        f = 440
        Audio(np.cos(2*np.pi*f*t), rate=Fs)
```

```
Out[5]: <IPython.lib.display.Audio object>
```

```
In [6]: #play a scale
        basenote = 440
        sound = []
        for note in scale:
            f = basenote * 2**((note-69)/12)
            sound.append(np.sin(2*np.pi*f*t))
        sound = np.concatenate(sound)
        Audio(sound, rate=Fs)
```

```
Out[6]: <IPython.lib.display.Audio object>
```

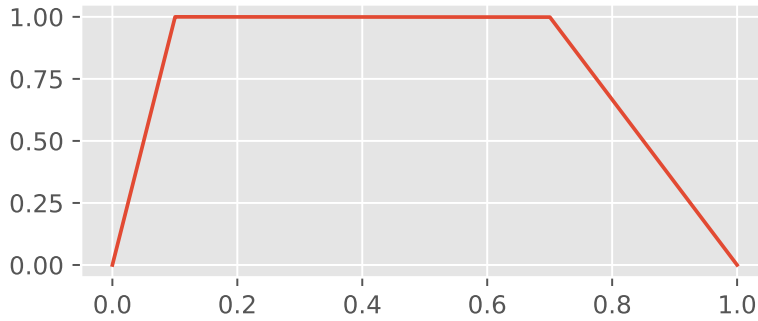
13.3 Envelope Functions

The notes sound better if an envelope function is applied to soften the note's rise and fall. Professional envelopes use an ADSR (Attack, Delay, Sustain, Release) shape.

The function below implements a simple envelope function. It uses the `np.select` function to implement the trapezoidal shape. The `condlist` is a list of conditions and `choicelist` is a list of values. The function selects the first element of `choicelist` whose `condlist` is True.

The gradual rise and fall in volume results in much less popping.

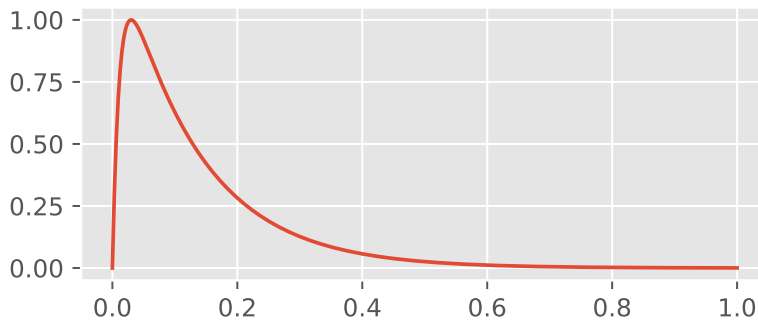
```
In [7]: def trap_env(t, dur=1, up = 0.1, down = 0.3):
            """simple trapezoid shaped envelope function,
            starts at 0, rises to 1,stays there,
            then drops back to 0"""
            condlist = [t<up, t<dur-down, True]
            choicelist = [t/up, 1, (dur-t)/down]
            return np.select(condlist, choicelist)
        plt.plot(t,trap_env(t,dur=1)); #test trap_env
```



A stringed instrument has a different shape. The note rises quickly and decays exponentially (as the vibrating string loses energy).

The function below include the argument `dur=1` even though the function does not use `dur`. We do this for two reasons: (1) the function may be modified in the future to incorporate `dur`, and (2) other envelope functions do use `dur` and including it here results in the same calling sequence for the various envelope functions.

```
In [8]: def string_env(t,dur=1):
            """envelope function for vibrating string"""
            env = (1-np.exp(-80*t))*np.exp(-8*t)
            return env/np.max(env)
        plt.plot(t,string_env(t));
```



13.4 Mary Had a Little Lamb

We use a simple song, "Mary Had a Little Lamb", to illustrate the basics of music synthesis. "Mary" is often the first song a piano student learns. The song consists of a list of notes, each note containing a frequency and duration. The notes are played consecutively, one note at a time.

For instance, the first note is E played for a half unit of time, the second note is D also played for a half unit of time. We take the basic unit of time to be 1 second, but shorter or longer times can be used. Below we play "Mary" with the default parameters.

```
In [9]: mary = [ (E, 1/2), (D, 1/2), (C, 1/2), (D, 1/2), (E, 1/2), (E, 1/2),
                 (E, 1), (D, 1/2), (D, 1/2), (D, 1), (E, 1/2), (G, 1/2),
                 (G, 1), (E, 1/2), (D, 1/2), (C, 1/2), (D, 1/2), (E, 1/2),
                 (E, 1/2), (E, 1/2), (E, 1/2), (D, 1/2), (D, 1/2), (E, 1/2),
                 (D, 1/2), (C, 1)]

        def playsong(song, env=trap_env, basenote=440, Fs=22500, time=1):
            """play song"""
            sounds = []
            for note in song:
                fnum, dur = note
                t = np.linspace(0,dur*time,int(dur*time*Fs),endpoint=False)
                f = basenote * 2**((fnum-69)/12)
                sinusoid = np.sin(2*np.pi*f*t)
                sounds.append(env(t,dur*time) * sinusoid)
            return np.concatenate(sounds)

        Audio(playsong(mary,Fs=Fs),rate=Fs)
```

```
Out[9]: <IPython.lib.display.Audio object>
```

Change the envelope to the `string_env`. Notice how the sound changes even though the notes are the same.

```
In [10]: string_mary = playsong(mary,env=string_env,Fs=Fs)
         Audio(string_mary, rate=Fs)
```

```
Out[10]: <IPython.lib.display.Audio object>
```

Change the `basenote=880` to play the song in the next highest octave.

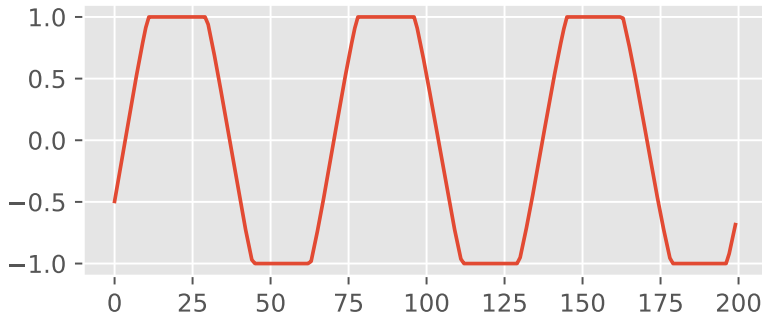
```
In [11]: Audio(playsong(mary,env=string_env,basenote=880,Fs=Fs),rate=Fs)
```

```
Out[11]: <IPython.lib.display.Audio object>
```

13.5 Clipping

Nonlinear clipping is used to create new sounds, e.g., in electric guitars. The `np.clip` command implements this.

```
In [12]: clipped = np.clip(string_mary,-0.6,0.6)/0.6
         plt.plot(clipped[1000:1200]);
```



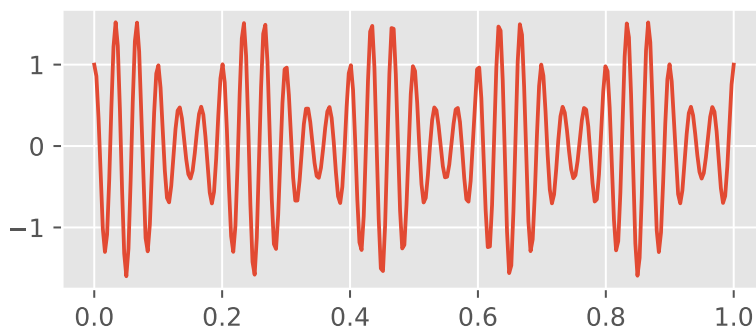

```
In [13]: Audio(clipped, rate=Fs)
```

```
Out[13]: <IPython.lib.display.Audio object>
```

13.6 Tremolo

Tremolo is a bit of amplitude modulation applied to the notes. Tremolo depends on two parameters: depth and frequency. Depth determines how much the volume varies and frequency determines how often the volume changes. The tremolo frequency must be below the threshold for human hearing (20 Hz).

```
In [14]: s = np.linspace(0,1,300)
         tremolo = 1+0.6*np.sin(2*np.pi*5*s)
         plt.plot(s,tremolo*np.cos(2*np.pi*30*s));
```



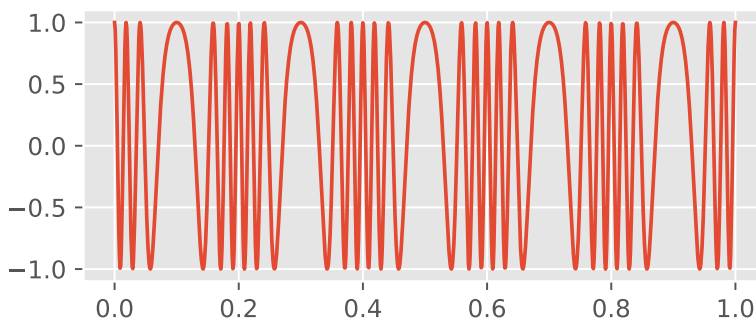
```
In [15]: depth = 0.6
         tremfreq = 8
         Fs = 22050

         sounds = []
         for note in mary:
             fnum, dur = note
             t = np.linspace(0,dur,int(dur*Fs),endpoint=False)
             tremolo = 1 + depth*np.sin(2*np.pi*tremfreq*t)
             f = basenote * 2**((fnum-69)/12)
             sinusoid = np.sin(2*np.pi*f*t)
             sounds.append(trap_env(t,dur) * tremolo * sinusoid)
         sound = np.concatenate(sounds)
         Audio(sound, rate=Fs)
```

```
Out[15]: <IPython.lib.display.Audio object>
```

13.7 FM Modulation

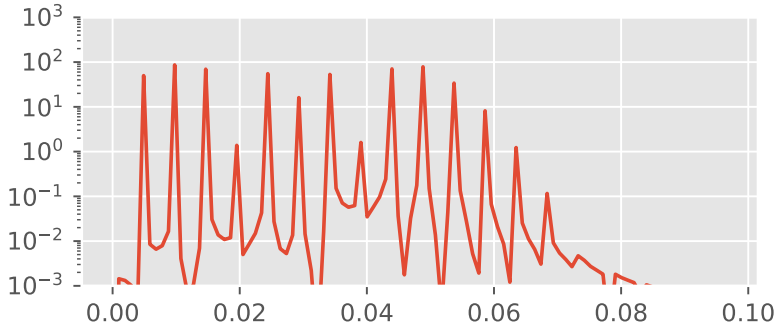
Frequency modulation (FM) is a change in the frequency of the signal. It has been applied to music synthesis since the 1970's. Below an FM signal is plotted.

```
In [16]: s = np.linspace(0,1,1024)
         fm = np.cos(2*np.pi*30*s + 5*np.sin(2*np.pi*5*s))
         plt.plot(s, fm);
```



The spectrum of an FM signal is rich in harmonics, as shown below.

```
In [17]: f, Pxx = sig.periodogram(fm)
         plt.semilogy(f[:100],Pxx[:100])
         plt.ylim(1e-3,1e3);
```




```
In [18]: fmsound = []
         for note in mary:
             fnum, dur = note
             t = np.linspace(0,dur,int(dur*Fs),endpoint=False)
             f = basenote * 2**((fnum-69)/12)
             FMsinusoid = np.cos(2*np.pi*f*t + 10*np.sin(4.1*np.pi*f*t))
             fmsound.append(string_env(t) * FMsinusoid)
         fmsound = np.concatenate(fmsound)
         Audio(fmsound, rate=Fs)
```

```
Out[18]: <IPython.lib.display.Audio object>
```

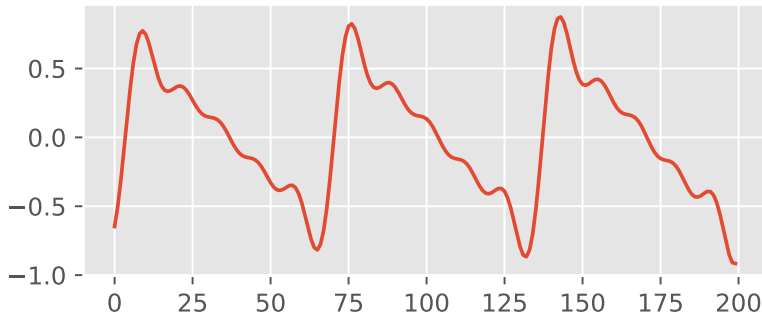
13.8 Add Harmonics

One of the best ways to add complexity to the sounds and to mimic actual instruments is to add harmonics (sinusoids at integer multiples of the fundamental).

The code below uses some numpy tricks: `np.outer(ks,t)` computes a 2D array with k running across the rows and t running across the columns, `np.sin()` applies the sine function to each element of the array, `harmonics @ np.sin()` multiplies each row by the harmonic coefficient and adds up the rows.

```
In [19]: harmonics = [1.0,0.5,0.4,0.3,0.2] #organ-like
         ks = np.arange(1,len(harmonics)+1)
         sounds = []
         for note in mary:
             fnum, dur = note
             t = np.linspace(0,dur,int(dur*Fs),endpoint=False)
             f = basenote * 2**((fnum-69)/12)
             sound = harmonics @ np.sin(2*np.pi*f*np.outer(ks,t))
             sounds.append(trap_env(t,dur,down=0.1) * sound)
         sound = np.concatenate(sounds)
         Audio(sound, rate=Fs)
```

```
Out[19]: <IPython.lib.display.Audio object>

In [20]: plt.plot(sound[1000:1200]);
```



13.9 Extensions and Summary

The primitive music synthesizer developed here demonstrates some of the techniques used in better synthesizers. It can be improved many ways:

- Rewrite the code to have multiple instruments and overlapping notes.
- Choose harmonic coefficients to mimic actual instruments, e.g., organs, brass instruments, etc.
- Build a guitar synthesizer using the Karplus-Strong algorithm.
- Add percussion instruments (often not based on harmonics).
- Add models based on the physics of instruments, e.g., standing waves in woodwind and brass instruments.

One interesting experiment is to play the song with a rich set of harmonics, then play the same song with the fundamental removed (i.e., set `harmonics = [0, ...]`) or greatly reduced. It is surprising how easily the ear believes the fundamental is present even when it is not.

14

Image Processing

Most applications considered so far in this book are one-dimensional, a signal that varies over time. In this chapter, we consider *image processing*. An image is a two-dimensional signal that is constant in time, i.e., a picture. A *video* is a two (or sometimes three) dimensional signal that varies in time, but video processing is beyond this text.

Python handles image processing through several libraries. In this chapter, we use `scipy.ndimage` which provides numpy related functions for processing images and `skimage`, short for *scikit-image*, which provides specialized image processing routines. Not considered in this chapter, but popular for computer vision applications, is *OpenCV*. OpenCV understands numpy arrays and interacts well with `ndimage` and `skimage`.

Digital images are large rectangular arrays of *pixels* (short for "picture elements"). Modern cameras acquire pictures with tens of millions of pixels. In this chapter, we use smaller images (about a quarter million pixels). These are faster and easier to manipulate, but the processing techniques are the same for small or large images.

Images consist of color bands. Grayscale images have a single color band. Each pixel is quantified by a single number. Pixels vary from black to white. Color images consist of multiple color bands. Each pixel is represented by multiple numbers. Images that represent light typically use three color bands, usually red, green, and blue (RGB images). Images that represent printed pictures typically use four to six color bands.

It must be emphasized that images representing light are *additive*. Red, green, and blue light combine to make white light. Black is the absence of light. Printed images, however, are *subtractive*. The page starts out white. As ink is applied, colors are absorbed and the spot gets darker.

In summary, in additive images, larger signal values lead to white; in subtractive images, more ink leads to black. Digital images must be properly interpreted whether they represent additive or subtractive colors. The two images in this notebook are both additive images.

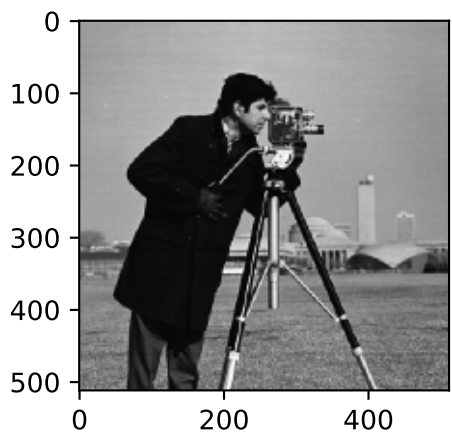
```
In [1]: import numpy as np
        import matplotlib.pyplot as plt
        import scipy.ndimage as nd
        import skimage as ski
        from skimage import data
        np.set_printoptions(precision=3)
        %matplotlib inline
```

14.1 Scikit-Image Example Images

`Scikit-Image` includes several example images. Here we load the `camera` image, a classic image that has appeared in numerous examples and research papers. `camera` is a 512×512 greyscale image (a quarter million pixels). Each pixel (short for "picture element") is an 8 bit "uint8" (one byte) value. Typically, black has value 0, white has value 255, and greys are between 0 and 255.

```
In [2]: camera = ski.data.camera()
        plt.figure(figsize=(2.5,2.5))
        plt.imshow(camera, cmap='gray')
        print(camera.shape, camera.dtype)
```

```
(512, 512) uint8
```



14.2 Image Plotting Routines

Below we present two python functions for plotting images within the notebook. While the matplotlib command `plt.imshow` prints an image, it does not have the best defaults. For one color images to be plotted as grayscale, the `cmap` (color map) option must be set to `gray` (the default is to plot a pseudo-color image). Also, the axes are not useful and we want to easily plot multiple images side-by-side.

```
In [3]: from itertools import zip_longest
        def plotimage(image,title='',cmap='gray', figsize=(6,4), **kws):
            """plot image with useful defaults"""
            fig, ax = plt.subplots(1,1,figsize=figsize)
            plt.imshow(image, cmap=cmap, **kws)
            if title: ax.set_title(title)
            ax.axis('off') #no x-y axes

        def plotimages(images, titles=[], cmaps=[], figsize=(6,4), **kws):
            """plot images with useful defaults"""
            fig, axs = plt.subplots(1, len(images), figsize=figsize)
            for im, ti, cm, ax in zip_longest(images,titles,cmaps,axs):
                if not cm: cm = 'gray'
                ax.imshow(im,cmap=cm)
                if ti: ax.set_title(ti)
                ax.axis('off')
                ax.set_adjustable('box-forced')
```

14.3 Digital Images are Numpy Arrays

`scipy.ndimage` and `skimage` use numpy arrays to represent images. A grayscale image is a two dimensional array and a color (RBG) image is a three dimensional array. The most popular data types for images are `uint8` (8 bit positive integers) and `float` (8 byte floating point numbers generally between 0 and 1).

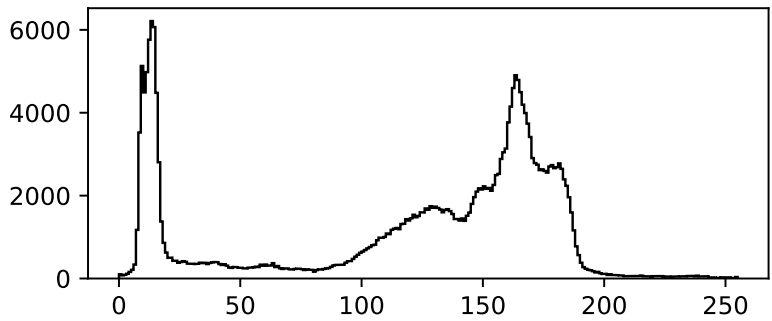
The cell below shows various numpy techniques to alter the image. The first fifty rows are set to a mid gray, all dark values (below 87) are set to white, black dots are added, and a circular mask is applied.

```
In [4]: #example adapted from scikit-image documentation
        #set first fifty rows to mid gray
        camera[:50] = 100

        #set all dark values to white
        mask = (camera < 87)
        camera[mask] = 255

        #dotted black dots on diagonals
        inds_x = np.arange(len(camera))
        inds_y = (4 * inds_x) % len(camera)
```

```
camera[inds_x, inds_y] = 0

#circular mask, outside to 0
l_x, l_y = camera.shape
X, Y = np.ogrid[:l_x, :l_y]
outer_disk_mask = (X - l_x / 2)**2 + (Y - l_y / 2)**2 > (l_x / 2)**2
camera[outer_disk_mask] = 0

plotimage(camera)
```

The image histogram displays the distribution of light and dark pixels. The histogram shows a peak of dark pixels (near 0), a broader peak of gray values roughly between 100 and 200, and relatively few bright white pixels (greater than 200).

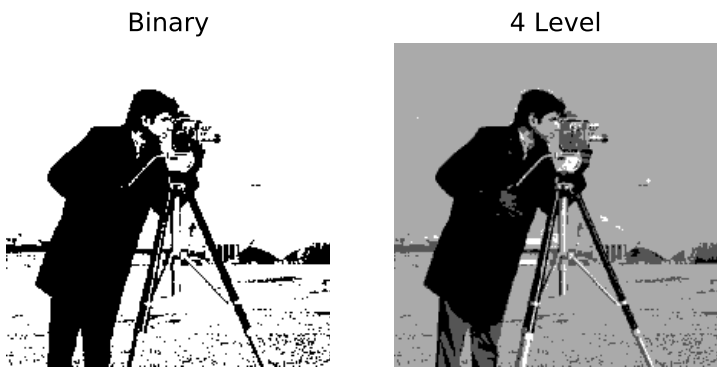
```
In [5]: camera = ski.data.camera() #reload camera image
        plt.hist(camera.flatten(), bins=255, histtype='step', color='black');
```



Below are two ways to quantize the image to fewer gray levels. The first `camera>100` sets values above 100 to True, and below 100 to False. `plt.imshow` maps True to white and False to black. The second method uses `np.digitize` to assign four levels based on histogram values. `np.bincount` gives the counts of each integer value, i.e., there are 52739 pixels digitized to 0 (black). `levels.flatten()` converts the array to one dimension.

```
In [6]: binary = (camera>100) #binary values, True=1, False=0
        levels = np.digitize(camera,[30,100,200,256]) #4 levels
        print(levels.min(), levels.max(), np.bincount(levels.flatten()))
        plotimages([binary,levels],['Binary','4 Level'])
```

```
0 3 [ 52739  21433 185138   2834]
```

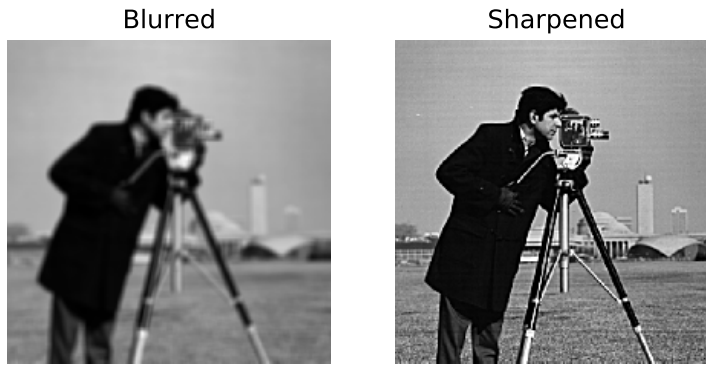
14.4 Filters, Blurring, and Sharpening

The basic filtering operation on digital images is the 2D convolution of a (small) window over the image. To exaggerate the blurring below, a window of size 11×11 is convolved with the image. At each location in the output image, the filter computes the mean of the 121 pixels centered at that location, i.e., it computes the mean of the 121 neighboring pixels.

Unsharp masking is a common technique for sharpening in image. It uses the simple idea: the difference between the original image and a blurred version enhances the edges and small details. The sharpened image is the original plus the edge enhanced version.

Filtering and sharpening (and many other operations) work best if the image is converted to `float`. The `np.clip` functions guarantees the output image is between 0 and 1. If necessary, the output image can be converted back to `uint8`.

```
In [7]: orig = ski.img_as_float(camera) #original camera image
        window = np.ones((11,11))/121 #mean filter
        blurred = nd.convolve(orig,window)
        sharpened = orig + 0.7*(orig-blurred) #unsharp masking
        sharpened = sharpened.clip(0,1)
        plotimages([blurred,sharpened],['Blurred','Sharpened'])
```



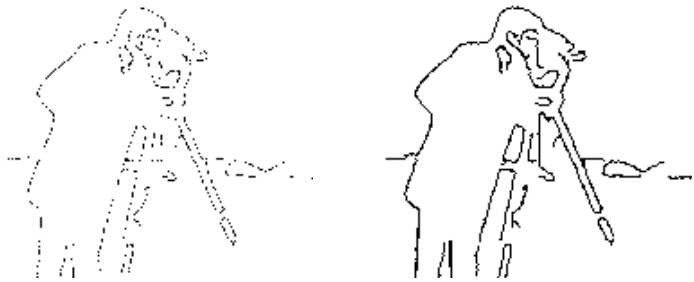
The `canny` function estimates edges in images. (Another choice is the `sobel` function.) On the left is displayed the estimated edges by the `canny` function. On the right, the edges are filled in and thickened by the morphological `binary_dilation` operation. Each black dot in the canny edge image is replaced by a diamond of five black dots.

```
In [8]: from skimage import morphology
        from skimage.feature import canny
```

```
        edges = canny(camera,sigma=5)
        diamond = morphology.diamond(1)
        diamond
```

```
Out[8]: array([[0, 1, 0],
               [1, 1, 1],
               [0, 1, 0]], dtype=uint8)
```

```
In [9]: thick_edges = morphology.binary_dilation(edges, diamond)
        plotimages([edges,thick_edges],cmaps=['Greys','Greys'],figsize=(5,3))
```



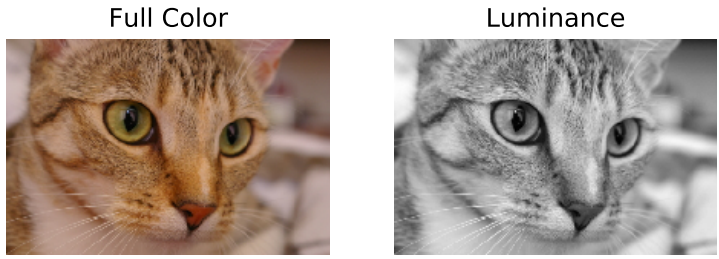
14.5 Color Images

Standard color images (RGB) can be thought of two different ways: first, as a rectangular array of pixels with each pixel taking three different values, and, second, as three rectangular arrays, one for red values, one for green values, and one for blue values.

`chelsea` is included with `skimage`. The `rgb2gray` function converts an RGB color image to a grayscale image. It uses the equation $Y = 0.2125R + 0.7154G + 0.0721B$. Notice the large emphasis given to the green color. This is because the human eye is more sensitive to green than to either red or blue.

```
In [10]: chelsea = ski.img_as_float(ski.data.chelsea())
         chelsea_y = ski.color.rgb2gray(chelsea)
         plotimages([chelsea,chelsea_y],['Full Color','Luminance'])
         print(chelsea.shape,chelsea_y.shape)
```

```
(300, 451, 3) (300, 451)
```

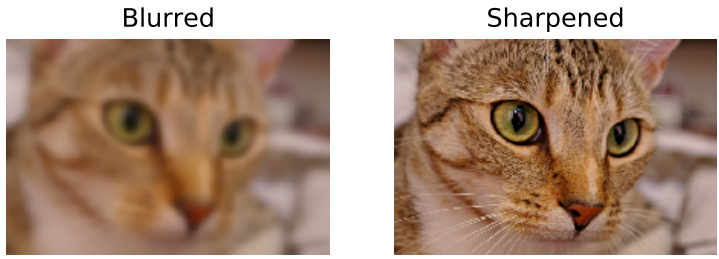


Below the `chelsea` image is separated in `R`, `G`, and `B` components. The pictures show two different color maps. The first maps large numbers to bright red. It results in a *negative* image, the opposite of what is normally done. The second uses a reverse color map and results in a more normal looking image.

```
In [11]: R, G, B = chelsea[:,:,0], chelsea[:,:,1], chelsea[:,:,2]
         plotimages([R,R],['Reds','Reds_r'],['Reds','Reds_r'])
```

Repeat the blurred and unsharp masking for the color image. In this example, we switch to a Gaussian blur filter using the `filters.gaussian` function. A gaussian blur mimics the blurring from out of focus optics. The `multichannel=True` option tells the filter to treat the color components separately.

```
In [12]: from skimage import filters
         chelsea_blurred = ski.filters.gaussian(chelsea, 5, multichannel=True)
         chelsea_sharp = chelsea + 0.5*(chelsea - chelsea_blurred)
         chelsea_sharp = chelsea_sharp.clip(0,1)
         plotimages([chelsea_blurred, chelsea_sharp],['Blurred','Sharpened'])
```



14.6 Artistic Examples

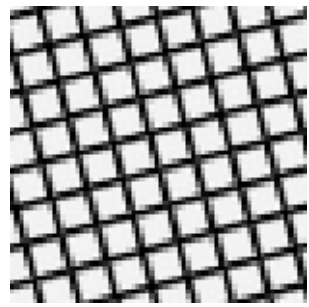
We create two artistic versions of chelsea. The first mimics a pen and ink drawing and the second is in comic book style. The first step is to find edges. Edge detection is usually done on the luminance image. Below are the edges as found by the `canny` function and the thickened edges as improved by the morphological filter.

```
In [13]: ch_edge = canny(chelsea_y,sigma=1.9)
         ch_edge_m = morphology.binary_dilation(ch_edge,
                                                morphology.diamond(1))
         plotimages([ch_edge,ch_edge_m],cmaps=['Greys','Greys'])
```

For the pen and ink drawing, we build a "crosshatch" image. The crosshatch is used to mimic gray levels. A portion of the crosshatch is shown below.

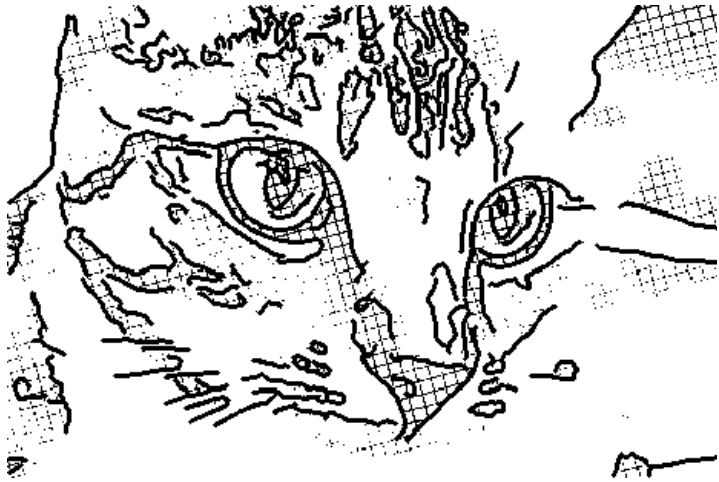
```
In [14]: lines = np.zeros([1024,1024])
         lines[::8] = 1
         lines[1::8] = 0.75
         lines15 = nd.rotate(lines, 15, reshape=False)
```

```
lines105 = nd.rotate(lines, 105, reshape=False)
crosshatch = np.maximum(lines15, lines105)
crosshatch = crosshatch[256:-256,256:-256]
plotimage(crosshatch[0:64,0:64],cmap='Greys',figsize=(2,2))
```



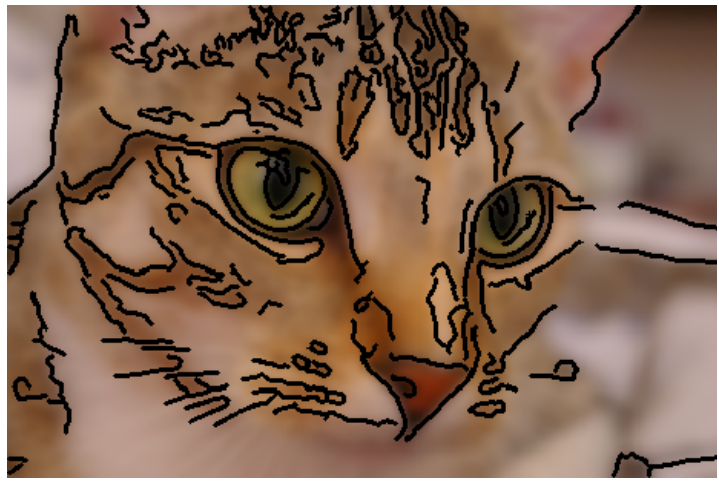
Halftoning is a process that gives binary images the illusion of multiple gray levels. A common halftoning technique is to compare the image to a *dither* pattern. The printing industry tries to hide the dithering, but here the crosshatch is visible, as it is in traditional drawings.

```
In [15]: r, c = chelsea_y.shape
         ch_dither = (crosshatch[:r,:c]-chelsea_y) > 0.7
         chelsea_ink = np.maximum(ch_dither, ch_edge_m)
         plotimage(chelsea_ink,cmap='Greys')
```



The comic image uses a blurred image as background and the edges as foreground.

```
In [16]: chelsea_blurred = ski.filters.gaussian(chelsea, 3,
                                                multichannel=True)
         chelsea_comic = np.minimum(chelsea_blurred,
                                    1-ch_edge_m[:,:,np.newaxis])
         plotimage(chelsea_comic)
```



14.7 Summary

Image processing is the application of signal processing techniques to digital images and videos. The python libraries, `scipy.ndimage` and `skimage` (scikit-image), represent images as numpy arrays. Any techniques to manipulate arrays can be used to manipulate images. However, both libraries supply specialized functions for images.

In this chapter, we showed basic image manipulations on grayscale and color images. More advanced functions, especially for image understanding, can be found in the *OpenCV* library.

15

Pandas for Data Analysis

Pandas is the preferred library for data analysis. Pandas provides two related data structures: *series* and *dataframes*. A series is a one-dimensional data structure with an *index* and *values*. A dataframe is a two-dimensional structure with an index and one or more columns of values. Pandas is built on top of numpy arrays and shares some characteristics with numpy; however, there are many differences between pandas and numpy.

Both series and dataframe values can be accessed by row and column indices (similarly to numpy arrays). Unlike numpy arrays, however, pandas series and dataframes can also be accessed by index values and column names.

Simply put, if your data looks like a table or a spreadsheet, use a pandas dataframe. If your data is a timeseries (especially with a complicated time structure) use a pandas series.

Data analysis consists of three main steps:

1. Find and acquire the data.
2. Manipulate the data into a form that can be analyzed. This step includes dealing with common problems, including missing or erroneous data and disjoint or disparate datasets.
3. Analyze the data and present the results.

Many data analysts will point out that step two above is often the least interesting, yet most time-consuming, of the three steps. It is this step where pandas excels.

This chapter features two examples of data analysis. The first is an analysis of average monthly temperatures for Paris, France. The second is an analysis of a time series of sunspot counts.

We import two new libraries: pandas and *datetime*. Datetime is a library for manipulating date and time stamps. We use datetime in analyzing the sunspot data.

From the `numpy.linalg` library, we use the `lstsq` function to compute the least squares (linear regression) estimator of the Paris temperature data.

```
In [1]: import datetime #for date calcuations
        import numpy as np
        import numpy.linalg as la
        import matplotlib.pyplot as plt
        plt.rcParams['figure.figsize'] = (5,1.75)
        import pandas as pd
        import scipy.signal as sig
        np.set_printoptions(precision=3)
        %matplotlib inline
```

15.1 Paris Temperatures

We acquired average minimum and maximum monthly temperatures for Paris, France, from weather.com. The dataset is simple and can be analyzed with `numpy` and `scipy.signal`. We use this dataset to illustrate some of pandas features.

One simple way to create a dataframe is to start with a dictionary. This dataframe has two columns, `Min` and `Max`. We use the conventional `df` to refer to the dataframe. The command, `df.head()`, shows the first five rows. At this point, the index of the dataframe is the set of numbers $0, 1, \ldots, 11$.

```
In [2]: paris = {'Min': [34, 34, 38, 42, 49, 54, 58, 57, 52, 46, 39, 36],
                 'Max': [43, 45, 51, 57, 64, 70, 75, 75, 69, 59, 49, 45]}
        df = pd.DataFrame(paris)
        df.head() #first five rows
```

Out[2]:

	Max	Min
0	43	34
1	45	34
2	51	38
3	57	42
4	64	49

If the data is more naturally viewed as rows, the dataframe can be transposed.

```
In [3]: newdf = df.T  #transpose
        newdf
```

Out[3]:

	0	1	2	3	4	5	6	7	8	9	10	11
Max	43	45	51	57	64	70	75	75	69	59	49	45
Min	34	34	38	42	49	54	58	57	52	46	39	36

It is common in weather to record daily high and low temperatures and compute a daily *average* temperature as the average of the high and low temperatures. Note how easy it is to add a new column to the dataframe.

```
In [4]: df['Ave'] = (df['Min']+df['Max'])/2.0
        df.head() #note new column
```

```
Out[4]:    Max  Min   Ave
        0   43   34  38.5
        1   45   34  39.5
        2   51   38  44.5
        3   57   42  49.5
        4   64   49  56.5
```

Compute basic statistics.

```
In [5]: df.describe() #basic statistics on columns
```

```
Out[5]:              Max        Min        Ave
        count  12.000000  12.000000  12.000000
        mean   58.500000  44.916667  51.708333
        std    11.950656   8.928589  10.423876
        min    43.000000  34.000000  38.500000
        25%    48.000000  37.500000  43.125000
        50%    58.000000  44.000000  51.000000
        75%    69.250000  52.500000  60.875000
        max    75.000000  58.000000  66.500000
```

15.1.1 Replace the Index

The cell below replaces the index with more descriptive month names.

```
In [6]: months = ['Jan', 'Feb', 'Mar', 'Apr', 'May',
            'Jun', 'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec']
        df.index = months  #reset the index
        df.head()
```

```
Out[6]:      Max  Min   Ave
        Jan   43   34  38.5
        Feb   45   34  39.5
        Mar   51   38  44.5
        Apr   57   42  49.5
        May   64   49  56.5
```

15.1.2 Accessing Pandas Elements

There are many ways to access the elements of a dataframe. The table below is taken (and modified slightly) from the Pandas documentation at pandas.pydata.org. It shows the most common ways to access dataframe elements.

Operation	Syntax	Result
Select column	df[col]	Series
Select columns	df[[col1,col2]]	DataFrame
Select row by label	df.loc[label]	Series
Select rows by labels	def.loc[[label1, label2]]	DataFrame

Operation	Syntax	Result
Select row by integer location	df.iloc[loc]	Series
Slice rows	df[5:10]	DataFrame
Select rows by boolean vector	df[bool_vec]	DataFrame

The slice on rows below includes both the first and last months (unlike python and numpy which both exclude the last value).

```
In [7]: df.loc['Apr':'Jun'] #df.loc to access by index, note the slice

Out[7]:      Max  Min   Ave
        Apr   57   42  49.5
        May   64   49  56.5
        Jun   70   54  62.0

In [8]: df.iloc[3:6] #df.iloc to access by row numbers

Out[8]:      Max  Min   Ave
        Apr   57   42  49.5
        May   64   49  56.5
        Jun   70   54  62.0

In [9]: df.iloc[3,0], df.iloc[3][0] #access by row and column numbers

Out[9]: (57, 57.0)

In [10]: df[['Min','Ave']][0:3] #combine row and column access

Out[10]:      Min   Ave
         Jan   34  38.5
         Feb   34  39.5
         Mar   38  44.5

In [11]: df[df['Max']>60] #select rows based on values

Out[11]:      Max  Min   Ave
         May   64   49  56.5
         Jun   70   54  62.0
         Jul   75   58  66.5
         Aug   75   57  66.0
         Sep   69   52  60.5
```

15.1.3 Using Numpy to Process the Data

The values of a dataframe can be accessed as a numpy array through the `.values` attribute.

```
In [12]: df['Ave'][:5].values #convert to numpy array

Out[12]: array([ 38.5,  39.5,  44.5,  49.5,  56.5])
```

Numpy functions can process pandas dataframes. The first example shows how to use a dataframe as input to a numpy function. The second example, using the pandas `.apply()` method, is the preferred way to do it.

```
In [13]: np.log10(df['Ave'][:5]) #numpy function on df

Out[13]: Jan    1.585461
         Feb    1.596597
         Mar    1.648360
         Apr    1.694605
         May    1.752048
         Name: Ave, dtype: float64

In [14]: df['Ave'][:5].apply(np.log10) #preferred

Out[14]: Jan    1.585461
         Feb    1.596597
         Mar    1.648360
         Apr    1.694605
         May    1.752048
         Name: Ave, dtype: float64
```


15.1.4 Analyze the Paris Temperatures

Temperatures are seasonal and repeat yearly. Accordingly we model the temperature as a constant plus a sinusoid at a frequency of one cycle per year.

$$\hat{y}(t) = a + b\cos(2\pi t) + c\sin(2\pi t)$$

where $0 \le t \le 1$.

The temperature values represent the values over the whole month. In the code below, we represent the midpoints of the months as $0.5, 1.5, \ldots, 11.5$.

We find the *best* values for a, b, and c by finding the least squared error predictor, comparing the model to the measurements. The `lstsq` function computes the least squares (linear regression) estimator. As an alternative, the `statsmodels` library can be used to compute the least squares estimator (and many other estimators).

```
In [15]: y = df['Ave'].values
         mos = np.arange(12)+0.5
         X = np.c_[np.ones(12), np.cos(2*np.pi*mos/12),
                   np.sin(2*np.pi*mos/12)]
         X, y

Out[15]: (array([[ 1.   ,  0.966,  0.259],
                 [ 1.   ,  0.707,  0.707],
                 [ 1.   ,  0.259,  0.966],
                 [ 1.   , -0.259,  0.966],
                 [ 1.   , -0.707,  0.707],
                 [ 1.   , -0.966,  0.259],
                 [ 1.   , -0.966, -0.259],
```

```
       [ 1.   , -0.707, -0.707],
       [ 1.   , -0.259, -0.966],
       [ 1.   ,  0.259, -0.966],
       [ 1.   ,  0.707, -0.707],
       [ 1.   ,  0.966, -0.259]]),
array([ 38.5,  39.5,  44.5,  49.5,  56.5,  62. ,  66.5,
        66. ,  60.5,  52.5,  44. ,  40.5]))
```

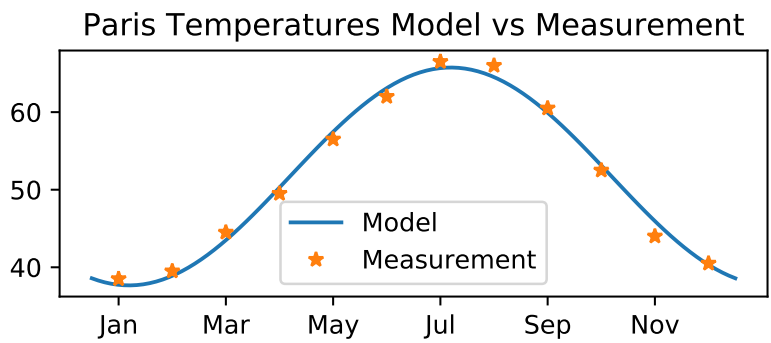
Compute the estimate.

```
In [16]: betahat, sum_squares, rank, sing_values = la.lstsq(X,y)
         betahat
```

```
Out[16]: array([ 51.708, -13.126,  -4.989])
```

Plot the model as a smooth curve and the measurements as asterisks. Note how close the model is to the measurements. The sinusoidal curve fits the temperature measurements well.

```
In [17]: t = np.linspace(0,12,101)
         Xt = np.c_[np.ones_like(t),np.cos(2*np.pi*t/12),
                    np.sin(2*np.pi*t/12)]
         yhat = Xt @ betahat
         plt.plot(t,yhat,label='Model')
         plt.plot(mos,y,'*',label='Measurement')
         plt.xticks(mos[::2], months[::2]) #every other month
         plt.legend()
         plt.title('Paris Temperatures Model vs Measurement');
```

15.2 Sunspot Observations

The second example is to plot and analyze sunspot data. Sunspots represent one of the longest, continually measured time series. Sunspots were first observed with telescopes in the early 1600's (there are isolated records of sunspots observed by naked eye observers throughout recorded history). Sunspots are the first indicators of changing behavior of the sun. That changing behavior may be related to changes observed on the earth. For instance, periods of low sunspot number, e.g., the Maunder minimum (1645-1715) and the Dalton minimum (1790-1830) are associated with cold temperatures. The current sunspot numbers are low, leading some scientists to wonder whether colder temperatures are coming.

We access a record of sunspots from www.esrl.noaa.gov. The file is a text file with a simple format. The first line is a title, the middle lines each consist of a year and twelve numbers separated by whitespace, with each number representing the average daily sunspot count for that month, followed by six more lines of text. The initial and final text can be ignored (for our analysis, but help in understanding what the file represents).

The pandas command, `pd.read_csv(file)`, can download the file from the internet and (with appropriate options) convert it to a dataframe. However, rather than download the file every time we run this notebook, we use a helper function that downloads the file and stores it for future use. The helper function relies on three standard libraries: `io` for processing the text, `requests` for accessing the internet, and `os.path` for manipulating the file's path.

```
In [18]: import io
         import requests
         import os.path

         def web_get_file(fileurl, force=False):
             """Get file from filesystem or download it.
             It assumes file is stored in the current folder.

             force=True to force download of the file
             """
             filename = os.path.basename(fileurl)
             if os.path.exists(filename) and not force:
                 return open(filename, 'r')
             else:
                 f = requests.get(fileurl)
                 with open(filename, 'w') as outfile:
                     outfile.write(f.text)
                 return io.StringIO(f.text) #look like a file
```

The `pd.read_csv()` command is the preferred way to convert a *CSV* file to a dataframe. Technically, the sunspot file is not a CSV file (it delimits entries with whitespace, not commas) but it is close enough. (The `web_get_file()` command is optional and can be eliminated.)

(The URL below is broken into two pieces to fit within the margins of the text.)

```
In [19]: url0 = 'https://www.esrl.noaa.gov/psd/gcos_wgsp'
         url1 = '/Timeseries/Data/sunspot.long.data'
         sunspot_file = url0 + url1
         col_names = ["Year", "Jan", "Feb", "Mar", "Apr", "May", "Jun",
                      "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"]
         df = pd.read_csv(web_get_file(sunspot_file),
                 skiprows=1,          #skip first row of text
                 header=None,         #file does not have a header row
                 delim_whitespace=True, #delimiters are whitespace
                 names = col_names,   #use these as column names
                 index_col = 'Year', #the 'Year' column is the index
                 engine='python',     #use this to allow 'skipfooter'
                 skipfooter=6)        #skip last six lines of text
         df.loc[1749:1751,'Jan':'Jun'] #display a few rows and columns
```

Out[19]:

Year	Jan	Feb	Mar	Apr	May	Jun
1749	58.0	62.6	70.0	55.7	85.0	83.5
1750	73.3	75.9	89.2	88.3	90.0	100.0
1751	70.0	43.5	45.3	56.4	60.7	50.7

As of this writing (summer 2017) the file contains observations from 1749 to 2015.

```
In [20]: df.index.min(), df.index.max()
```

Out[20]: (1749, 2015)

Look at the last few columns in the last rows. Notice the `-99.9` values used to represent missing observations. I.e., the last actual observation in this dataset is for May 2015.

```
In [21]: df.loc[2011:2015,'Apr':'Dec']
```

Out[21]:

Year	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
2011	54.4	41.5	37.0	43.8	50.6	78.0	88.0	96.7	73.0
2012	55.2	69.0	64.5	66.5	63.0	61.4	53.3	61.8	40.8
2013	72.4	78.7	52.5	57.0	66.0	37.0	85.6	77.6	90.3
2014	84.7	75.2	71.0	72.4	74.6	87.6	60.6	70.2	76.7
2015	54.4	58.8	-99.9	-99.9	-99.9	-99.9	-99.9	-99.9	-99.9

15.2.1 Convert Data to Series

The data is not in the format we want. There is no physical reason why sunspots should vary monthly (the earth's revolution is too tiny to affect the sun). Better would be a single time series, not a two-dimensional table. The `df.stack()` command reorders the data into a single series with a *MultiIndex* consisting of year and month name.

```
In [22]: s = df.stack() #s = series
         s.name = "Sunspots"
         s.head() #show first five lines

Out[22]: Year
         1749  Jan    58.0
               Feb    62.6
               Mar    70.0
               Apr    55.7
               May    85.0
         Name: Sunspots, dtype: float64
```

Replace the MultiIndex with a single index of year and month. The `datetime.date()` function returns a date (year, month, day) in the format the `pd.period_range()` command needs. Also, clean the data by eliminating the rows with fake observations of `-99.9`.

```
In [23]: start = datetime.date(df.index.min(),1,1) #Jan 1 of first year
         end = datetime.date(df.index.max(),12,1) #Dec 1 of last year
         s.index = pd.period_range(start=start, end=end, freq='M')
         s = s[s>=0] #eliminate negative values
         s.tail()

Out[23]: 2015-01    67.0
         2015-02    44.8
         2015-03    38.4
         2015-04    54.4
         2015-05    58.8
         Freq: M, Name: Sunspots, dtype: float64
```


15.2.2 Sunspot Statistics

Look at some elementary statistics.

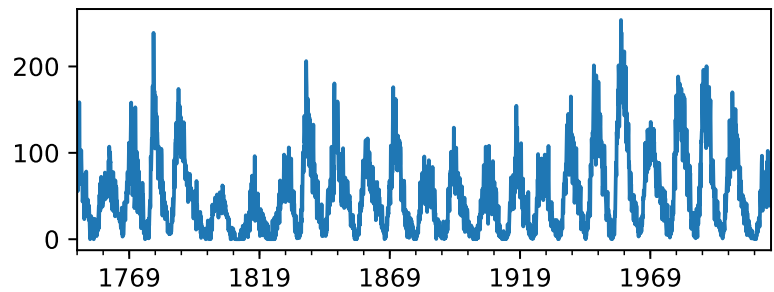
```
In [24]: s.describe()

Out[24]: count    3197.000000
         mean       52.097685
         std        44.036410
         min         0.000000
         25%        15.700000
         50%        42.400000
         75%        76.500000
         max       253.800000
         Name: Sunspots, dtype: float64
```

While the mean daily sunspot count is 52, there are months where the mean is 0. How many are there? List the most recent ones. Except for one recent month, all the zero sunspot months occur over a century ago. This may indicate a change in the sun's behavior, or it may indicate a change in the counting procedure (better equipment today), or it may be a statistical fluke (negligible chance).

```
In [25]: s[s==0].count() #67 months with zero sunspots

Out[25]: 67

In [26]: s[s==0][-5:] #most recent 5 sunspot-free months

Out[26]: 1902-04    0.0
         1912-02    0.0
         1913-05    0.0
         1913-06    0.0
         2009-08    0.0
         Freq: M, Name: Sunspots, dtype: float64
```

Pandas includes matplotlib plotting built-in (though the syntax is sometimes different). The sunspot counts vary with a period of approximately 11 years.

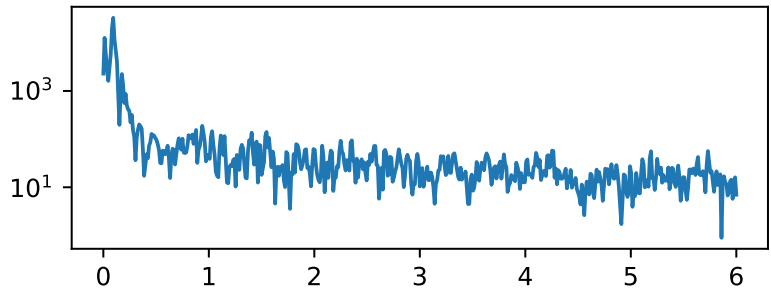
```
In [27]: s.plot(); #notice the automatic xtick labels
```



15.2.3 The Sunspot Count Spectrum

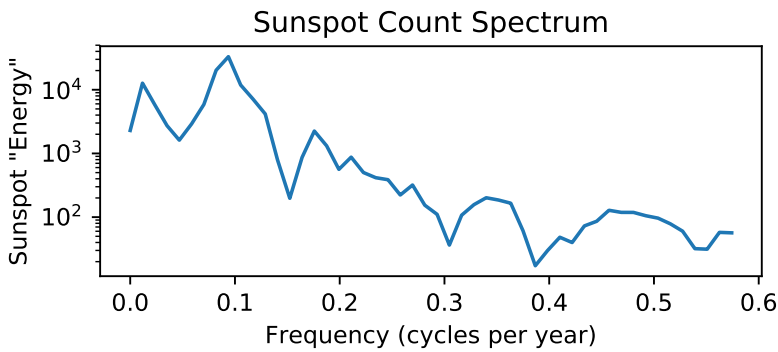
Signal processing tools can estimate the sunspot count spectrum. The data is sampled monthly, `fs=12`. The 11 year period corresponds to a frequency of $f = 1/11 = 0.09$ cycles per year. Less obvious from the plot above is another cycle with period approximately 60 years, corresponding to a frequency of $1/60 = 0.016$ cycles per year. To observe this cycle, we need to use an FFT of length at least $60 \cdot 12 = 720$ samples. That is why `nperseg=1024`.

```
In [28]: f, w = sig.welch(s.values-s.mean(), fs=12, nperseg=1024,
                          noverlap=256)
         plt.semilogy(f,np.abs(w));
```



Zoom in on the left side to see the two cycles. The large central peak corresponds to the eleven year cycle, the smaller peak to the left to the sixty year cycle.

```
In [29]: plt.semilogy(f[:50],np.abs(w[:50]))
         plt.title('Sunspot Count Spectrum')
         plt.ylabel('Sunspot "Energy"')
         plt.xlabel('Frequency (cycles per year)');
```



15.3 Pandas Summary and Comments

Pandas is the preferred package for data processing, especially the intermediate steps of massaging the data into a form that can be analyzed. Simple analyses can be done within pandas, e.g., basic statistics, but other tools, such as `numpy`, `statsmodels` (statistical models), and `scikit-learn` (machine learning), can be used.

Pandas is a *big* package. This chapter is only an introduction. Consult the on-line documentation or the many books featuring pandas for more information.

Appendices

A

Bibliography

Recommended Signal Processing Books:

1. Lathi B. and Green R., *Essentials of Digital Signal Processing*, Cambridge University Press, 2014.
2. Mitra, S., *Digital Signal Processing: A Computer-Based Approach*, McGraw-Hill, 2011.
3. Oppenheim, A., Willsky, A. and Nawab, S., *Signals & Systems*, Prentice-Hall, 1996.

Recommended Books on Python, Numpy, Scipy, and Pandas:

1. Lawson, R. and Jarmul, K., *Python Web Scraping, 2nd Ed.*. Packt Publishing, 2017.
2. Lutz, M., *Learning Python*, O'Reilly Media, 2013.
3. McKinney, W., *Python for Data Analysis, 2nd Ed.*, O'Reilly Media, 2017.
4. Mehta, H., *Mastering Python Scientific Computing*, Packt Publishing, 2015.
5. VanderPlas, J., *Python Data Science Handbook*, O'Reilly Media, 2016.

Recommended Websites for Python, Numpy, Scipy, and Pandas:

1. *https://www.python.org/doc/* Official documentation for python.
2. *https://scipy.org/docs.html* Documentation for Numpy, Scipy, Pandas, IPython.
3. *http://matplotlib.org/* Documentation for Matplotlib.
4. *http://pandas.pydata.org/* Documentation for Pandas.

Specific Citations in this book:

1. *http://www.piano-keyboard-guide.com/*, piano-keyboard image.
2. Kaihola, A. *Unit Testing in the Python Stack*, PyCon Finland, 2016.
3. http://spectrum.ieee.org/computing/software/the-2017-top-programming-languages, IEEE, 2017.