José Unpingco

Python for Signal Processing

Featuring IPython Notebooks

Python for Signal Processing

José Unpingco

Python for Signal Processing

Featuring IPython Notebooks

Springer

José Unpingco
San Diego, CA
USA

ISBN 978-3-319-01341-1 ISBN 978-3-319-01342-8 (eBook)
DOI 10.1007/978-3-319-01342-8
Springer Cham Heidelberg New York Dordrecht London

Library of Congress Control Number: 2013946655

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

To Irene, Nicholas, and Daniella, for all their patient support.

Preface

This book will teach you the fundamentals of signal processing via the Python language and its powerful extensions for scientific computing. This is not a good *first* book in signal processing because we assume that you already had a course in signal processing at the undergraduate level. Furthermore, we also assume that you have some basic understanding of the Python language itself, perhaps through an online course (e.g., `codeacademy.com`). Having said that, this book is appropriate if you have a basic background in signal processing and want to learn how to use the scientific Python toolchain. On the other hand, if you are comfortable with Python, perhaps through working in another scientific field, then this book will teach you the fundamentals of signal processing. Likewise, if you are a signal processing engineer using a commercial package (e.g., MATLAB, IDL), then you will learn how to effectively use the scientific Python toolchain by reviewing concepts you are already familiar with.

The unique feature of this book is that everything in it is reproducible using Python. Specifically, all of the code, all of the figures, and (most of) the text is available in the downloadable supplementary materials that correspond to this book in the form of IPython Notebooks. IPython Notebooks are *live* interactive documents that allow you to change parameters, recompute plots, and generally tinker with all of the ideas and code in this book. I urge you to download these IPython Notebooks and follow along with the text to experiment with the signal processing topics covered. As an open-source project, the entire scientific Python toolchain, including the IPython Notebook, is freely available. Having taught this material for many years, I am convinced that the only way to learn is to experiment as you go. The text provides instructions on how to get started installing and configuring your scientific Python environment.

This book is not designed to be exhaustive and reflects the author's eclectic background in industry. The focus is on fundamentals for day-to-day work. Although Python supports many powerful constructs such as decorators, generators, and context managers, all the code here uses Python in the most straightforward way possible while encouraging good Python coding practices.

Acknowledgements I would like to acknowledge the help of Brian Granger and Fernando Peréz, two of the originators of the IPython Notebook, for all their great work, as well as the Python community as a whole, for all their contributions that made this book possible. Additionally, I would also like to thank Juan Carlos Chávez for his thoughtful review.

San Diego, CA, USA José Unpingco

Contents

Chapter 1
Introduction

1.1 Introduction

Python went mainstream years ago. It is well-established in web programming and is the platform for high-traffic sites like `YouTube`. Less well known is Python for scientific applications, what we are calling "Scientific Python" here. Scientific Python has been used in government, academia, and industry for at least a decade. NASA's Jet Propulsion Laboratory uses it for interfacing Fortran/C++ libraries for planning and visualization of spacecraft trajectories. The Lawrence Livermore National Laboratory uses scientific Python for a wide variety of computing tasks, some involving routine text processing, and others involving advanced visualization of vast data sets (e.g. VISIT [CBB+05]). Shell Research, Boeing, Industrial Light and Magic, Sony Entertainment, and Procter & Gamble use scientific Python on a daily basis for similar tasks. Python is well-established and continues to extend into many different fields.

Python is an interpreted language. This means that Python codes run on a Python *virtual machine* that provides a layer of abstraction between your code and the platform it runs on. This makes Python scripts portable, but slower than a compiled language such as Fortran. In a compiled language, the compiler takes the text of the code, studies it end-to-end, and then writes an executable that links against compiled system libraries. Once the executable is created, there is no further need for the compiler.

Python is different. Python codes *need* the Python interpreter, but the interpreter can also call the *same* system libraries as the compiler. This means that the time-consuming numerical algorithms are *not* implemented in the Python language itself (that would be too slow!). They are called from compiled libraries. Thus, Python provides a staging area for scientific computing, handing off the intensive work to compiled libraries, while providing many more benefits.

For scientific computing, the main benefit of Python is that it provides a common medium of exchange between many different kinds of scientific libraries. This makes it possible to mix-and-match routines from many different sources

J. Unpingco, *Python for Signal Processing*, DOI 10.1007/978-3-319-01342-8_1,

that were not otherwise designed to work together. Moreover, Python provides a multiplatform solution for scientific codes. As an open-source project, Python itself is available anywhere you can build it, even though it typically ships standard nowadays, as part of many operating systems. This means that once you have written your code in Python, you can just transfer the script to another platform and run it, as long as the compiled libraries are also available there. What if the compiled libraries are absent? Building and configuring compiled libraries across multiple systems used to be a painstaking job, but as scientific Python has matured, a wide range of libraries have now become available across all of the major platforms (i.e. Windows, MacOS, Linux, Unix) as prepackaged distributions.

Finally, scientific Python facilitates maintainability of scientific codes because Python syntax is clean, free of semi-colon litter and other visual clutter that makes code hard to read and easy to obfuscate. Python has many built-in testing, documentation, and deployment tools that ease maintenance. Scientific codes are usually written by scientists unschooled in software development, so having these development tools built into the language itself is a particular boon for scientific computing.

1.2 Installation and Setup

The easiest way to get started is to download the freely available Anaconda package from Continuum Analytics, which is available for all the major platforms. If you need it, Continuum also offers support for an additional fee. On Windows workstations, the free `PythonXY` distribution is a one-click installation of the scientific Python toolchain, as well as many other packages. On the major Linux distributions, the toolchain is usually available via the package manager (i.e. `apt-get`, `yum`). Regardless of your platform, we recommend the Python version 2.7 distributions. Python 2.7 is the "last" of the Python 2.x series and guarantees backwards compatibility with legacy codes. Python 3.x makes no such guarantees. Although all of the key components of the Python toolchain are available in version 3.x, the safest bet is to stick with version 2.7 until further notice. Enthought Canopy is the next generation of the Enthought Python Distribution (EPD), which is an enterprise-level scientific Python integrated development environment. The "Express" version of Canopy is available for free and works on all the major platforms.

You may later want to use a Python module that is not part of the major scientific Python distributions mentioned above and that you have to build yourself. The Windows platform presents singular difficulties because of the lack of freely available compilers (especially for Fortran) and the general difficulty of working with this complicated platform. A great place to look for high quality one-click installers is Christoph Gohlke's laboratory site at the University of California, Irvine where he kindly makes a long list of scientific modules available. Otherwise, Python packages on other platforms can be installed using `easy_install` or `pip`.

1.3 Numpy

To use a compiled scientific library, the memory allocated in the Python interpreter must somehow reach this library as input. Furthermore, the output from these libraries must likewise return to the Python interpreter. This two-way exchange of memory is essentially the core function of the Numpy (numerical arrays in Python) module. Numpy is the de-facto standard for numerical arrays in Python. It arose as an effort by Travis Oliphant and others to unify the many approaches to numerical arrays in Python. In this section, we provide an overview and some tips for using Numpy effectively, but for much more detail, Travis' book *Guide to Numpy* [Oli06] is still a great place to start and is available for free in PDF form.

Numpy provides specification of byte-sized arrays in Python. For example Listing **1.1** creates an array of three numbers, each of 4-bytes long (32 bits at 8 bits per byte) as shown by the `itemsize` property in Listing **1.1**. In addition to providing uniform containers for numbers, Numpy provides a long list if unary functions (*ufuncs*) that process arrays element-wise without additional looping semantics. Listing **1.2** shows how to compute the element-wise sine using Numpy,

Listing **1.2** shows computing the sine using Numpy where the input is a Numpy array and the output is the element-wise sine of that array. Numpy uses common-sense casting rules to resolve the output types. For example, if the inputs had been an integer-type, the output would still have been a floating point type. Further, observe that there was no Python looping construct (e.g. `for` loop) required to compute this element-wise because this happens in the compiled code that is invoked by the unary function. In this example, we provided a Numpy array as input to the sine function. We could have also used a plain Python list instead and Numpy would have built the intermediate Numpy array (e.g. `np.sin([1,1,1])`). The Numpy Documentation provides a comprehensive (and very long) list of available ufuncs.

Numpy arrays come in various shapes and dimensions. For example, Listing **1.3** shows a two-dimensional `2x3` array constructed from two conforming Python lists.

Note that you can have as many as 32 dimensions and if you need more then you can build Numpy to your specification.[1] Numpy arrays follow the usual Python slicing rules, but in more dimensions as shown in Listing **1.4** where the "`:`" character selects all elements along a particular axis. You can also select sub-sections of arrays by using slicing as shown in Listing **1.5**.

1.3.1 Numpy Arrays and Memory

Some interpreted languages implicitly allocate memory. For example, in MATLAB, you can extend a matrix by simply tacking on another dimension as in the MATLAB

[1] See `arrayobject.h`

session in Listing 1.6: This works because MATLAB arrays use pass-by-value semantics so that slice operations actually copy parts of the array as needed. By contrast, Numpy uses pass-by-reference semantics so that slice operations are *views* into the array without implicit copying. Numpy forces you to explicitly allocate memory for a new Numpy array. For example, to accomplish the same array extension in Numpy, you have to do something like Listing 1.7.

As shown in Listing 1.7, we cannot tack an extra dimension onto an array and instead must explicitly create a new array (line 12). One way to do this is with the indexing trick of using one of the dimensions twice as shown in line 12 above. It's important to understand that Numpy views point to the same elements in memory.

Continuing with the example in Listing 1.7, we can change one of the elements in `x` as shown in Listing 1.8, but this does not affect `y` because of the copy we forced earlier. However, if we start over and construct `y` by slicing (which makes it a view) as shown in Listing 1.9, then the change we made *does* affect `y`. If you want to explicitly force a copy without any indexing tricks, you can do `y=x.copy()`.

In addition to slicing, Numpy offers many advanced indexing facilities so that Numpy arrays can be indexed by other Numpy arrays and lists. Listing 1.10 shows an interesting contrast between indexing and slicing. In Listing 1.10, we created an array in `x` and created `z` by slicing. Then, we created `y` via integer indexing. Upon changing `x`, because only `z` is a view, only `z` was affected, not `y`, even though they coincidentally have the same elements. The `flags.ownsdata` property of Numpy arrays can help sort this out until you get used to it, as shown in Listing 1.11.

1.3.2 Numpy Matrices

Matrices in Numpy are very similar to Numpy arrays, but they implement the row-column matrix multiplication as opposed to element-wise multiplication. If you have two matrices you want to multiply, you can either create them directly or convert them from Numpy arrays. For example, Listing 1.12 shows how to create two matrices and multiply them. More commonly, you can convert Numpy arrays as in Listing 1.13. Everything until line 9 is a Numpy array and at this point we cast the Numpy array as a matrix with `np.matrix` which then uses row-column multiplication. Note that it is unnecessary to cast the `x` variable as a matrix because the left-to-right order of the evaluation takes care of that automatically. If we need to use `A` as a matrix elsewhere in the code then we should bind it to another variable instead of re-casting it every time. If you find yourself casting back and forth for large arrays, passing the `copy=False` flag to `matrix` avoids the expense of making a copy.

1.3.3 Numpy Broadcasting

Numpy broadcasting is a powerful way to make implicit multidimensional grids for expressions. It is probably the single most powerful feature of Numpy and the most difficult to grasp. Proceeding by example, consider the vertices of a two-dimensional unit square in Listing **1.14**.

Listing **1.14** shows the `X` and `Y` arrays whose corresponding entries match the coordinates of the vertices of the unit square (e.g. $(0, 0)$, $(0, 1)$, $(1, 0)$, $(1, 1)$). To add the x and y-coordinates, we could use `X` and `Y` as in `X+Y` shown below in Listing **1.15**, The output is the sum of the vertex coordinates of the unit square. It turns out we can skip a step here and not bother with `meshgrid` to implicitly obtain the vertex coordinates by using broadcasting as shown in Listing **1.16**.

Listing **1.16** shows how to avoid the intermediate `meshgrid` calculation by using Numpy broadcasting to create the intermediate arrays. This happens on line 7 where the `None` Python singleton tells Numpy to make copies of `y` along this dimension to create a conformable calculation. The following lines show that we obtain the same output as when we used the `X+Y` Numpy arrays. Note that without broadcasting `x+y=array([0, 2])` which is not what we are trying to compute. Let's continue with a more complicated example where we have differing array shapes.

Listing **1.17** shows that broadcasting works with different array shapes. For the sake of comparison, on line 3, `meshgrid` creates two conformable arrays, `X` and `Y`. On the last line, `x+y[:,None]` produces the same output as `X+Y` without the `meshgrid`. We can also put the `None` dimension on the `x` array as `x[:,None]+y` which would give the transpose of the result.

Broadcasting works in multiple dimensions also. Listing **1.18** shows broadcasting in three dimensions. The output shown has shape `(4,3,2)`. On the last line, the `x+y[:,None]` produces a two-dimensional array which is then broadcast against `z[:,None,None]`, which duplicates itself along the *two* added dimensions to accommodate the two-dimensional result on its left (i.e. `x + y[:,None]`). The caveat about broadcasting is that it can potentially create large intermediate arrays. There are methods for controlling this by re-using previously allocated memory but that is beyond our scope here. Some codes use a lot of broadcasting and some not at all. We use it occasionally in this text where it abbreviates superfluous looping.

1.4 Matplotlib

Matplotlib is the primary visualization tool for scientific graphics in Python. Like all great open-source projects, it originated to satisfy a personal need. At the time of its inception, John Hunter primarily used MATLAB for scientific visualization, but as he began to integrate data from disparate sources (e.g. internet, filesystems) using Python, he realized he needed a Python solution for visualization, so he single-handedly wrote Matplotlib. Since those early years, Matplotlib has displaced the

other competing methods for two-dimensional scientific visualization and today is a very actively maintained project, even without John Hunter, who sadly passed away in 2012.

John had a few basic requirements for Matplotlib:

- Plots should look publication quality with beautiful text.
- Plots should output Postscript for inclusion within LaTeX documents and publication quality printing.
- Plots should be embeddable in a Graphical User Interface (GUI) for application development.
- The code should be mostly Python to allow for users to become developers.
- Plots should be easy to make with just a few lines of code for simple graphs.

Each of these requirements has been completely satisfied, judging from the enthusiastic reception of Matplotlib by the scientific Python community. In the beginning, to ease the transition from MATLAB to Python, many of the Matplotlib functions were closely named after the corresponding MATLAB commands. The community has moved away from this style and, even though you will still find the old MATLAB-esque style used in the Matplotlib documentation. This book uses the more powerful and explicit Matplotlib interfaces.

Listing **1.19** shows the quickest way to draw a plot using Matplotlib and the plain Python interpreter. Later, we'll see how to do this even faster using IPython. The first line imports the requisite module as `plt` which is the recommended convention. The next line plots a sequence of numbers generated using Python's `range` function. Note the output list contains a `Line2D` object. This is an *artist* in Matplotlib parlance. Finally, the `plt.show()` function draws the plot in a GUI (i.e. *figure window*).

If you try this in your own plain Python interpreter (and you should!), you will see that you cannot type in anything further in the interpreter until the figure window is closed. This is because the `plt.show()` function preoccupies the interpreter with the controls in the GUI and *blocks* further interaction. As we discuss below, IPython provides ways to get around this blocking so you can simultaneously interact with the interpreter and the figure window.

The `plot` function returns a list containing the `Line2D` object as shown in Listing **1.19**. More complicated plots yield larger lists filled with *artists* in Matplotlib terminology. The suggestion is that artists draw on the *canvas* contained in the Matplotlib figure. The final line is the `plt.show` function that provokes the embedded artists to render on the Matplotlib canvas. The reason this is a separate function is that plots may have dozens of complicated artists and rendering may be a time-consuming task to only be undertaken at the end, when all the artists have been mustered. Matplotlib supports plotting types such as images, contours, and many others that we cover in detail in the following chapters.

Even though Listing **1.19** is the quickest way to draw a plot in Matplotlib, it is not recommended because there are no handles to the intermediate products of the plot such as the plot's axis. While this is okay for a simple plot like this, later on we will see how to construct complicated plots using the recommended style. There

is a close working relationship between Numpy and Matplotlib and you can load Matplotlib's plotting functions and Numpy's functions simultaneously using `pylab` as `from matplotlib.pylab import *`. Although importing everything this way as a standard practice is not recommended because of namespace pollution, there is no danger in this case due to the close working relationship between Numpy and Matplotlib.

One of the best ways to get started with Matplotlib is to browse the extensive on-line "gallery" of plots on the main Matplotlib site. Each of the plots comes with corresponding source code that you can use as a starting point for your own plots. In Sect. 1.6, we discuss special "magic" commands that make this particularly easy.

1.5 Alternatives to Matplotlib

Even though Matplotlib is unbeatable for script-based plotting, there are some alternatives for specialized scientific graphics that may be of interest. PyQwt is a set of bindings for the Qwt library, which contains useful GUI components and widgets for technical plotting. Matplotlib has some widgets for interactive plots, but they are pretty limited. The downside of PyQwt is that it relies on the Qwt documentation, which might be hard to read if you are not comfortable with C++ codes. Furthermore, the package does not seem to be currently maintained. The upside is that these libraries have been around for at least a decade, so they are well-tested and lighter-weight than Matplotlib.

Chaco is part of the Enthought Tool-Suite (ETS) and implements many real-time data visualization concepts and corresponding widgets. It is available on all of the major platforms and is also actively maintained and well-documented. Chaco is geared towards GUI application development, rather than script-based data visualization. It depends on the Traits package, which is also available in ETS and in the Enthought Python Distribution (EPD). If you don't want to use EPD, then you have to build Chaco and its dependencies separately. On Linux, this should be straight-forward, but potentially a nightmare on Windows if not for Christoph Gohlke's one-click installers.

If you require real-time data display and tools for volumetric data rendering and complicated 3D meshes with isosurfaces, then PyQtGraph is an option. PyQtGraph is a pure-Python graphics and GUI library that depends on Python bindings for the Qt GUI library (i.e. PySide or PyQt4) and Numpy. This means that the PyQtGraph relies on these other libraries (i.e. Qt's GraphicsView framework) for the heavy-duty numbercrunching and rendering. This package is actively maintained, but is still pretty new, with good (but not comprehensive) documentation. You also need to grasp a few Qt-GUI development concepts to use this effectively. Mayavi is another Enthought-supported 3D visualization package that sits on VTK (open-source C++ library for 3D visualization). Like Chaco, it is a toolkit for scientific GUI development as opposed to script-based plotting. To use it effectively, you need

to already know (or be willing to learn) about concepts like graphics pipelines. This package is actively supported and well-documented.

1.6 IPython

IPython originated as a way to enhance Python's basic interpreter for smooth interactive scientific development. In the early days, the most important enhancement was "tab-completion" for dynamic introspection of workspace variables. For example, you can start IPython at the commandline by typing `ipython` and then you should see something like the following in your terminal:

```
IPython 0.14.dev -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
\%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

In [1]:
```

Next, creating a string as shown and hitting the `TAB` key after the "dot" character initiates the introspection, showing all the functions and attributes of the `string` object in `x`.

```
In [1]: x = 'this is a string'

In [2]: x.<TAB>
x.capitalize x.format     x.isupper    x.rindex     x.strip
x.center     x.index      x.join       x.rjust      x.swapcase
x.count      x.isalnum    x.ljust      x.rpartition x.title
x.decode     x.isalpha    x.lower      x.rsplit     x.translate
x.encode     x.isdigit    x.lstrip     x.rstrip     x.upper
x.endswith   x.islower    x.partition  x.split      x.zfill
x.expandtabs x.isspace    x.replace    x.splitlines
x.find       x.istitle    x.rfind      x.startswith
```

To get help about any of these, you simply add the `?` character at the end as shown below,

```
In [2]: x.center?
Type:       builtin_function_or_method
String Form:<built-in method center of str object at 0x03193390>
Docstring:
S.center(width[, fillchar]) -> string

Return S centered in a string of length width. Padding is
done using the specified fill character (default is a space)
```

and IPython provides the built-in help documentation. Note that you can also get this documentation with `help(x.center)` which works in the plain Python interpreter as well.

The combination of dynamic tab-based introspection and quick interactive help accelerates development because you can keep your eyes and fingers in one place as you work. This was the original IPython experience, but IPython has since grown into a complete framework for delivering a rich scientific computing workflow that retains and enhances these fundamental features.

1.6.1 IPython Notebook

As you may have noticed investigating Python on the web, most Python users are web-developers, not scientific programmers, meaning that the Python toolchain is *very* well developed for web technologies. The genius move of the IPython development team was to leverage these technologies for scientific computing by embedding IPython in modern web-browsers. You can start the IPython Notebook, with all the Matplotlib and Numpy functions set up with the following command-line: `ipython notebook --pylab=inline`. The `inline` flag embeds Matplotlib graphics created in the IPython Notebook into the browser itself instead of popping open a separate GUI. After starting the notebook, you should see something like the following in the terminal,

```
[NotebookApp] Using existing profile dir: u'C:\\.ipython\\profile_default'
[NotebookApp] Serving notebooks from local directory: D:\
[NotebookApp] The IPython Notebook is running at: http://127.0.0.1:8888/
[NotebookApp] Use Control-C to stop this server and shut down all kernels.
```

The first line reveals where IPython looks for default settings. The next line shows where it looks for documents in the IPython Notebook format. The third line shows that the IPython Notebook started a web-server on the local machine (i.e. `127.0.0.1`) on port number `8888`. This is the address your browser needs to connect to the IPython session although your default browser should have opened automatically to this address. The port number and other configuration options are available either on the commandline or in the `profile_default` shown in the first line. If you are on a Windows platform and you do not get this far, then the Window's firewall is probably blocking the port. For additional configuration help, see the main IPython site (`ipython.org`) or e-mail the very responsive IPython mailing list (`ipython-dev@scipy.org`).

When IPython starts, it initiates many small Python processes that use the blazing-fast `ZeroMQ` message passing framework for interprocess-communication, along with the web-sockets protocol for back-and-forth communication with the browser. As of yet, because neither Internet Explorer nor Safari support web-sockets, you will have to use Firefox or Google Chrome to use the IPython Notebook. To start IPython and get around your default browser, you can use the additional `--no-browser` flag and then manually type in the local host address

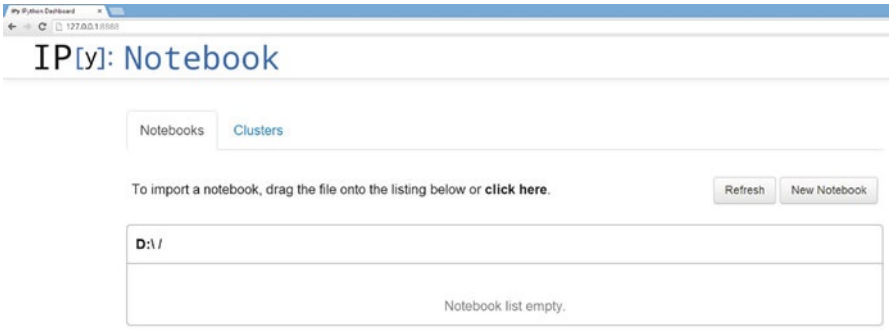


Fig. 1.1 The IPython Notebook dashboard

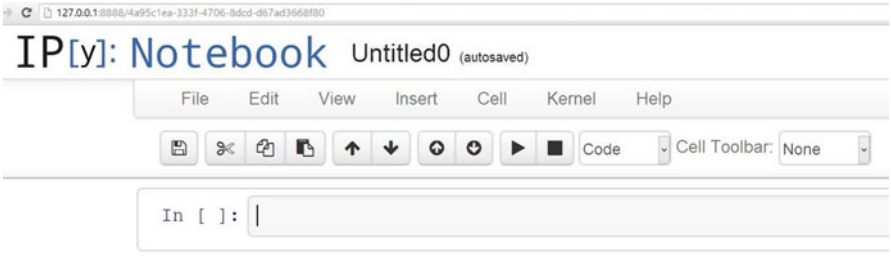


Fig. 1.2 A new IPython Notebook

`http://127.0.0.1:8888` into your favorite browser to get started. Once all that is settled, you should see something like the following Fig. 1.1.

You can create a new document by clicking the "New Notebook" button shown in Fig. 1.1. Then, you should see something like Fig. 1.2. To start using the IPython Notebook, you just start typing code in the shaded textbox and then hit `SHIFT+ENTER` to execute the code in that IPython *cell*. Figure 1.3 shows the dynamic introspection in the pulldown menu when you type the `TAB` key after the `x.`. Context-based help is also available as before by using the `?` suffix which opens a help panel at the bottom of the browser window. There are many amazing features including the ability to share notebooks between different users and to run IPython Notebooks in the Amazon cloud, but these features go beyond our scope here. Check the `ipython.org` website or peek at the mailing list for the lastest work on these fronts.

The IPython Notebook supports high-quality mathematical typesetting using MathJaX, which is a JavaScript version of most of LaTeX, as well as video and other rich content. The concept of consolidating mathematical algorithm descriptions and the code that implements those algorithms into a shareable document is more important than all of these amazing features. There is no understating the importance of this in practice because the algorithm documentation (if it exists) is usually in one format and completely separate from the code that implements it. This

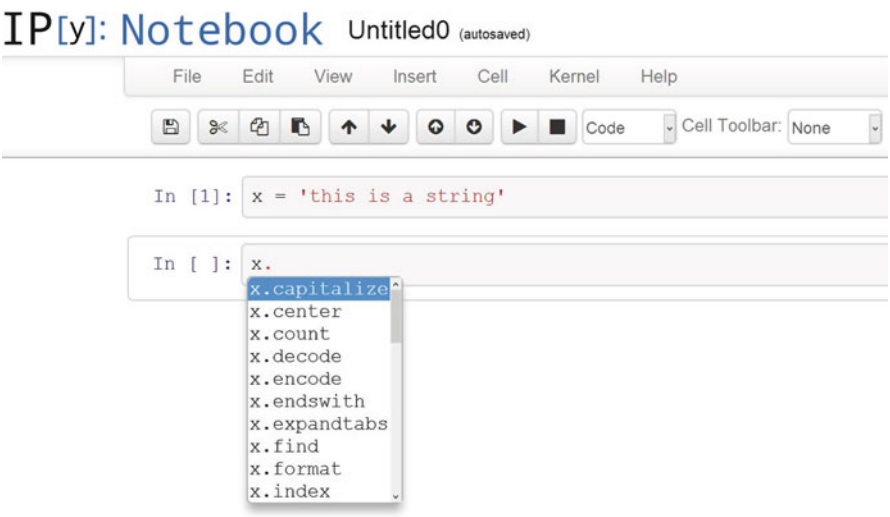


Fig. 1.3 Dynamic introspection in IPython Notebook with the pull-down menu

tragically common practice leads to un-synchronized documentation and code that renders one or the other useless. The IPython Notebook solves this problem by putting everything into a living shareable document based upon open standards and freely available software. IPython Notebooks can even be saved as static HTML documents for those without Python!

Finally, IPython provides a large set of "magic" commands for creating macros, profiling, debugging, and viewing codes. A full list of these can be found by typing in `%lsmagic` in IPython. Help on any of these is available using the `?` character suffix. Some frequently used commands include the `%cd` command that changes the current working directory, the `%ls` command that lists the files in the current directory, and the `%hist` command that shows the history of previous commands (including optional searching). The most important of these for new users is probably the `%loadpy` command that can load scripts from the local disk or from the web. Using this to explore the Matplotlib gallery is a great way to experiment with and re-use the plots there.

1.7 Scipy

Scipy (pronounced "Sigh Pie") was one of the first truly consolidated modules for a wide range of compiled libraries, all based on Numpy arrays. It contains the `signal` module that we focus on for this text and others that we occasionally sample. Note that some of the same functions appear in multiple places within Scipy itself as

well as in Numpy. Scipy includes numerous special functions (e.g. Airy, Bessel, elliptical) as well as powerful numerical quadrature routines via the QUADPACK Fortran library (see `scipy.integrate`), where you will also find other kinds of quadrature methods. Additionally, Scipy provides access to the ODEPACK library for solving differential equations. Lots of statistical functions, including random number generators, and a wide variety of probability distributions are included in the `scipy.stats` module. Interfaces to the Fortran MINPACK optimization library are provided via `scipy.optimize`. These include methods for root-finding, minimization and maximization problems, with and without higher-order derivatives. Methods for interpolation are provided in the `scipy.interpolate` module via the FITPACK Fortran package. Note that some of the modules are so big that you do not get all of them with `import scipy` because that would take too long. You may have to load some of these packages individually as `import scipy.interpolate`, for example.

As we discussed, the Scipy module is already packed with an extensive list of scientific codes. For that reason, the `scikits` modules were originally established as a way to try out candidates that could eventually make it into the already stuffed Scipy module, but it turns out that many of these modules became so successful in their own right that they will probably never be integrated into Scipy proper. Some examples include `sklearn` for machine learning and `scikit-image` for image processing.

1.8 Computer Algebra

Although we don't use it in this text, you should be aware of Sympy, which is a pure-Python module for computer algebra. It has benefited from "Google Summer of Code" sponsorship and grown into a powerful computer algebra system for Python. Furthermore, it has spawned many sub-projects that make it faster and integrate tighter with Numpy and IPython (among others). Alternatively, the Sage project is a consolidation of over 70 of the best open source packages for computer algebra and related computation. Although Sympy and Sage share code freely between them, Sage is a specialized build of the Python kernel itself as a way to hook into the underlying libraries. Thus, it is not a pure-Python solution for computer algebra and is really a superset of Python with its own extended syntax. From a portability standpoint, the main drawback of Sage is that it cannot be run on a Windows platform without running a virtual Linux machine that in turn runs Sage. Notwithstanding the lack of native Windows support, the choice between them depends on whether or not Sage implements a particular software package of interest.

The IPython Notebook is tightly integrated with Sympy and provides mathematical typesetting for Sympy so that the implemented computer algebra is readable as a publication-quality mathematical document. Use `ipython --profile=sympy`

`--pylab=inline` to start IPython Notebook with integrated Sympy support. The IPython Notebook also supports Sage.

1.9 Interfacing with Compiled Libraries

As we have discussed, Python for scientific computing really consists of gluing together different scientific libraries written in a compiled language like C or Fortran. Ultimately, you may want to use libraries not available with existing Python bindings. There are many, many options for doing this. The most direct way is to use the built-in `ctypes` module which provides tools for providing input/output pointers to the library's functions just as if you were calling them from a compiled language. This means that you have to know the function signatures in the library *exactly*—how many bytes for each input and how many bytes for the output. You are responsible for building the inputs exactly the way the library expects and collecting the resulting outputs. Even though this seems tedious, Python bindings for vast libraries have been built this way.

If you want an easier way, then `SWIG` is an automatic wrapper generating tool that can provide bindings to a long list of languages, not just Python; so if you need bindings for multiple languages, then this is your best and only option. Using `SWIG` consists of writing an interface file so that the compiled Python dynamically linked library (Python PYD) can be readily imported into the Python interpreter. Huge and complex libraries like Trilinos (Sandia National Labs) have been interfaced to Python using `SWIG`, so it is a well-tested option.

However, the `SWIG` model assumes that you want to continue developing primarily in C/Fortran and you are hooking into Python for usability or other reasons. On the other hand, if you start developing algorithms in Python and then want to speed them up, then `Cython` is an excellent option because it provides a mixed language that allows you to have both C-language and Python code intermixed. Like `SWIG`, you have to write additional files in this hybrid Python/C dialect to have `Cython` generate the C-code that you will ultimately compile. The best part of `Cython` is the profiler that can generate an HTML report showing where the code is slow and could benefit from translation to `Cython`. Both `SWIG` and `Cython` provide tools to easily connect to Numpy arrays. As you may have guessed, these work best for Linux/Unix development because of the ready availability of high-quality compilers.

`Cython` and `SWIG` are just two of the ways to create Python bindings for your favorite compiled libraries. Other notable options include `FWrap`, `f2py`, `CFFI`, and `weave`. It is also possible to use Python's own API directly, but this is an undertaking best left to professional coders. Although not mainly an interface option, the `Pypy` project is alternative implemention of Python that implements a just-in-time compiler (JIT) and other powerful optimizations that can substantially speed up pure Python codes. Extending `Pypy` to include Numpy and the rest of the scientific toolchain is well underway, but is still not appropriate for newcomers to Python.

1.10 Other Resources

The Python community is filled with super-smart and amazingly helpful people. One of the best places to get help with scientific Python is the `stackoverflow` site which hosts a competitive Q&A forum that is particularly welcoming for Python newbies. Several of the key Python developers regularly participate there and the quality of the answers is very high. The mailing lists for any of the key tools (e.g. Numpy, IPython, Matplotlib) are also great for keeping up with the newest developments. Anything written by Hans Petter Langtangen [Lan09] is excellent, especially if you have a physics background. The Scientific Python conference held annually in Austin is also a great place to see your favorite developers in person, ask questions, and participate in the many interesting sub-groups organized around niche topics. The `PyData` workshop is a semi-annual meeting focused on Python for large-scale data-intensive processing. The `PyVideo` site provides links to videos of talks and tutorials related to Python from around the world.

Appendix

```
>>> import numpy as np # recommended convention
>>> x = np.array([1,1,1],dtype=np.float32)
>>> x
array([ 1.,  1.,  1.], dtype=float32)
>>> x.itemsize
4
```

Listing 1.1: Line 1 imports Numpy as `np`, which is the recommended convention. The next line creates an array of 32 bit floating point numbers. The `itemize` property shows the number of bytes per item.

```
>>> np.sin(np.array([1,1,1],dtype=np.float32) )
array([ 0.84147096,  0.84147096,  0.84147096], dtype=float32)
```

Listing 1.2: This computes the sine of the input array of all ones, using Numpy's unary function, `np.sin`. There is another sine function in the built-in `math` module, but the Numpy version is faster because it does not require explicit looping (i.e. using a `for` loop) over each of the elements in the array. That looping happens in `np.sin` function itself.

```
>>> x=np.array([ [1,2,3],[4,5,6] ])
>>> x.shape
(2, 3)
```

Listing 1.3: Numpy arrays can have different shapes and number of dimensions.

```
>>> x=np.array([ [1,2,3],[4,5,6] ])
>>> x[:,0] # 0th column
array([1, 4])
>>> x[:,1] # 1st column
array([2, 5])
>>> x[0,:] # 0th row
array([1, 2, 3])
>>> x[1,:] # 1st row
array([4, 5, 6])
```

Listing 1.4: Numpy slicing rules extend Python's natural slicing syntax. Note the colon ":" character selects all elements in the corresponding row or column.

```
>>> x=np.array([ [1,2,3],[4,5,6] ])
>>> x
array([[1, 2, 3],
       [4, 5, 6]])
>>> x[:,1:] # all rows, 1st thru last column
array([[2, 3],
       [5, 6]])
>>> x[:,::2] # all rows, every other column
array([[1, 3],
       [4, 6]])
```

Listing 1.5: Numpy slicing can select sections of an array as shown.

```
>> x=ones(3,3)
x =
     1     1     1
     1     1     1
     1     1     1
>> x(:,4)=ones(3,1) % tack on extra dimension
x =
     1     1     1     1
     1     1     1     1
     1     1     1     1
>> size(x)
ans =
     3     4
```

Listing 1.6: MATLAB employs pass-by-value semantics meaning that slice operations like this automatically generate copies.

```
>>> x = np.ones((3,3))
>>> x
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
>>> x[:,[0,1,2,2]] # notice duplicated last dimension
array([[ 1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.]])
>>> y=x[:,[0,1,2,2]] # same as above, but do assign it
```

Listing 1.7: In contrast with MATLAB, Numpy uses pass-by-reference semantics so it creates *views* into the existing array, without implicit copying. This is particularly helpful with very large arrays because copying can be slow.

```
>>> x[0,0]=999 # change element in x
>>> x                          # changed
array([[ 999.,    1.,    1.],
       [   1.,    1.,    1.],
       [   1.,    1.,    1.]])
>>> y                          # not changed!
array([[ 1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.]])
```

Listing 1.8: Because we made a copy in Listing 1.7, changing the individual elements of `x` does *not* affect `y`.

```
>>> x = np.ones((3,3))
>>> y = x[:2,:2] # upper left piece
>>> x[0,0] = 999 # change value
>>> x
array([[ 999.,    1.,    1.], # see the change?
       [   1.,    1.,    1.],
       [   1.,    1.,    1.]])
>>> y
array([[ 999.,    1.], # changed y also!
       [   1.,    1.]])
```

Listing 1.9: As a consequence of the pass-by-reference semantics, Numpy views point at the same memory as their parents, so changing an element in `x` updates the corresponding element in `y`. This is because a view is just a window into the same memory.

```
>>> x = np.arange(5) # create array
>>> x
array([0, 1, 2, 3, 4])
>>> y=x[[0,1,2]] # index by integer list
>>> y
array([0, 1, 2])
>>> z=x[:3]      # slice
>>> z            # note y and z have same entries?
array([0, 1, 2])
>>> x[0]=999     # change element of x
>>> x
array([999,   1,   2,   3,   4])
>>> y            # note y is unaffected,
array([0, 1, 2])
>>> z            # but z is (it's a view).
array([999,   1,   2])
```

Listing 1.10: Indexing can also create copies as we saw before in Listing 1.7. Here, `y` is a copy, not a view, because it was created using indexing whereas `z` was created using slicing. Thus, even though `y` and `z` have the same entries, only `z` is affected by changes to `x`.

```
>>> x.flags.owndata
True
>>> y.flags.owndata
True
>>> z.flags.owndata # as a view, z does not own the data!
False
```

Listing 1.11: Numpy arrays have a built-in `flags.owndata` property that can help keep track of views until you get the hang of them.

```
>>> import numpy as np
>>> A=np.matrix([[1,2,3],[4,5,6],[7,8,9]])
>>> x=np.matrix([[1],[0],[0]])
>>> A*x
matrix([[1],
        [4],
        [7]])
```

Listing 1.12: Numpy arrays support elementwise multiplication, not row-column multiplication. You must use Numpy matrices for this kind of multiplication.

```
>>> A=np.ones((3,3))
>>> type(A) # array not matrix
<type 'numpy.ndarray'>
>>> x=np.ones((3,1)) # array not matrix
>>> A*x
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
>>> np.matrix(A)*x # row-column multiplication
matrix([[ 3.],
        [ 3.],
        [ 3.]])
```

Listing 1.13: It is easy and fast to convert between Numpy arrays and matrices because doing so need not imply any memory copying (recall the pass-by-value semantics). In the last line, we did not have to bother converting `x` because the left-to-right evaluation automatically handles that.

```
>>> X,Y=np.meshgrid(np.arange(2),np.arange(2))
>>> X
array([[0, 1],
       [0, 1]])
>>> Y
array([[0, 0],
       [1, 1]])
```

Listing 1.14: Numpy's `meshgrid` creates two-dimensional grids.

```
>>> X+Y
array([[0, 1],
       [1, 2]])
```

Listing 1.15: Because the two arrays have compatible shapes, they can be added together element-wise.

```
>>> x = np.array([0,1])
>>> y = np.array([0,1])
>>> x
array([0, 1])
>>> y
array([0, 1])
>>> x + y[:,None] # add broadcast dimension
array([[0, 1],
       [1, 2]])
>>> X+Y
array([[0, 1],
       [1, 2]])
```

Listing 1.16: Using Numpy broadcasting, we can skip creating compatible arrays using `meshgrid` and instead accomplish the same thing automatically by using the `None` singleton to inject an additional compatible dimension.

```
>>> x = np.array([0,1])
>>> y = np.array([0,1,2])
>>> X,Y = np.meshgrid(x,y)
>>> X
array([[0, 1], # duplicate by row
       [0, 1],
       [0, 1]])
>>> Y
array([[0, 0], # duplicate by column
       [1, 1],
       [2, 2]])
>>> X+Y
array([[0, 1],
       [1, 2],
       [2, 3]])
>>> x+y[:,None] # same as w/ meshgrid
array([[0, 1],
       [1, 2],
       [2, 3]])
```

Listing 1.17: In this example, the array shapes are different, so the addition of `x` and `y` is not possible without Numpy broadcasting. The last line shows that broadcasting generates the same output as using the compatible array generated by `meshgrid`.

```
>>> x = np.array([0,1])
>>> y = np.array([0,1,2])
>>> z = np.array([0,1,2,3])
>>> x+y[:,None]+z[:,None,None]
array([[[0, 1],
        [1, 2],
        [2, 3]],
       [[1, 2],
        [2, 3],
        [3, 4]],
       [[2, 3],
        [3, 4],
        [4, 5]],
       [[3, 4],
        [4, 5],
        [5, 6]]])
```

Listing 1.18: Numpy broadcasting also works in multiple dimensions. We start here with three one-dimensional arrays and create a three-dimensional output using broadcasting. The `x+y[:None]` part creates a conforming two-dimensional array as in Listing **1.17**, and due to the left-to-right evaluation order, this two-dimensional intermediate product is broadcast against the `z` variable, whose two `None` dimensions create an output three-dimensional array.

```
>>> import matplotlib.pyplot as plt
>>> plt.plot(range(10))
[<matplotlib.lines.Line2D object at 0x00CB9770>]
>>> plt.show() # unnecessary in IPython (discussed later)
```

Listing 1.19: The first line imports the Matplotlib module following the recommended naming convention. The next plots a range of numbers. The last line actually forces the plot to render. Because this example assumes you are in the plain Python interpreter, this last step is necessary. Otherwise, IPython (discussed later) makes this step unnecessary.

Chapter 2
Sampling Theorem

We enjoy the power and convenience of mobile communications today because of a very important exchange made early in the history of this technology. Once upon a time, radios were completely stunted by size, weight, and power limits. Just think of the backpack-sized radios that infantrymen carried around during World War II. The primary reason we enjoy tiny mobile radios (i.e. cellular phones) is that the analog design burdens were shifted to the digital domain which in turn placed the attendent digital algorithms onto integrated circuit technology, whose power vs. cost ratio accelerated favorably over the ensuing decades. The key bridge between the analog and the digital is sampling. In terms of basic electronics, an analog signal voltage is applied to a sample-and-hold circuit and held there by a tiny bit of capacitance while a bank of comparators marks off the proportional integer value. These values are the digital samples of the analog signal. The challenge is to analyze the original analog signal from its collected samples. The sampling theorem provides the primary mathematical mechanism for doing this.

2.1 Sampling Theorem

The following is the usual statement of the theorem from `Wikipedia`:

> If a function $x(t)$ contains no frequencies higher than B hertz, it is completely determined by giving its ordinates at a series of points spaced $1/(2B)$ seconds apart.

Because $x(t)$ is a function from the real line to the real line, there are infinitely many points between any two consecutive samples and thus sampling is a massive reduction of data because it only takes a finite number of points to completely characterize the function. We have seen the concept of reducing a function to a discrete set of numbers before in Fourier series expansions where (for periodic $x(t)$) we have,

$$a_n = \frac{1}{T}\int_0^T x(t)\exp(-j\omega_n t)dt \tag{2.1}$$

J. Unpingco, *Python for Signal Processing*, DOI 10.1007/978-3-319-01342-8_2,

with corresponding reconstruction as:

$$x(t) = \sum_n a_n \exp(j\omega_n t) \tag{2.2}$$

But here we generate discrete points a_n by integrating over the *entire* function $x(t)$, not just sampling it at a single point. This means we collect information about the entire function to compute a single discrete point a_n, whereas with sampling we are just taking individual points in isolation.

On the other hand, suppose we are given a set of samples $[x_1, x_2, .., x_N]$ and we are then asked to reconstruct the function. What would we do? Perhaps the most natural thing to do is draw a straight line between each of the points as in linear interpolation. Listing **2.1** takes samples of the sine over a single period and draws a line between samples.

The first line of Listing **2.1** ensures that we use floating-point division not Python's default integer division where `1/4=0` instead of `0.25`. You can embed this statement in your startup files (`ipython_config.py`) to avoid constantly putting it at the top of all your scientific codes, but *only* as long as you run your codes within IPython. See your `ipython_config.py` file in your `profile_default` directory that you noted when you first started the notebook, as we discussed in the previous chapter.

Line 3 sets up the figure and corresponding axes. This follows the modern convention for using Matplotlib. The `fig` is an object bound to the figure and `ax` is the axis that is drawn within that figure. The reason we want these separate is more complicated plots may have multiple axes embedded in the same figure. Because Matplotlib can render the plots in GUI windows, in files, or embedded in other applications (among others), it is not a good practice to assume that your plotting commands are operating on whatever the "currently active" figure is. Using `fig` and `ax` avoids this pitfall.

On line 6, the `arange` function creates a Numpy array of numbers starting at `-1` until `1+1/fs` in steps of `1/fs`. Recall that the built-in `range` command in Python yields a half-open interval (i.e. excluding right endpoint) and the same is true of `arange`. The next line computes the sine function on the array of points we just created and returns a Numpy array. Note that although Python itself comes with a sine function in the built-in `math` module, this sine is the Numpy version. The difference is that the Numpy version can ingest a Numpy array and produce a corresponding output without any extra looping that the `math` module's version would otherwise require.

The `ax.plot` command attaches the plot to the axis `ax`. The next two lines put text labels on the x-axis and y-axis. These labeling functions also provide many possibilities for text formatting including potential LaTeX fonts. Recall that Matplotlib artists are objects like axes, lines, markers, and text that actually *render* the given graphics.

Figure 2.1 shows that even where the function is curviest ($t = 1/(4f)$ and $t = 3/(4f)$), we have the same density of points as anywhere else. This is because the

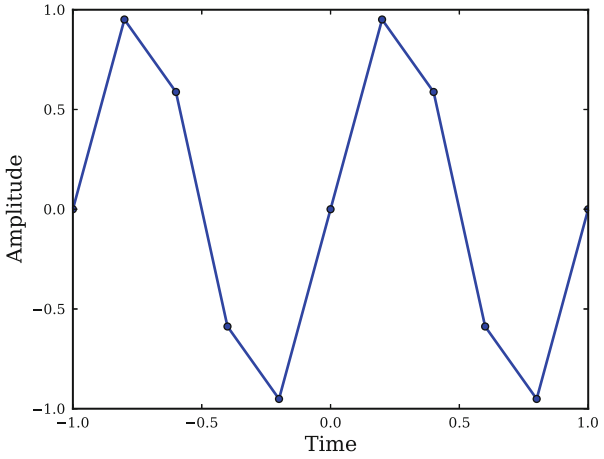


Fig. 2.1 Graphic generated by Listing 2.1. This shows the sine function and its samples. Note that sampling density is independent of the local curvature of the function. It may seem like it makes more sense to sample more densely where the function is curviest, but the sampling theorem has no such requirement

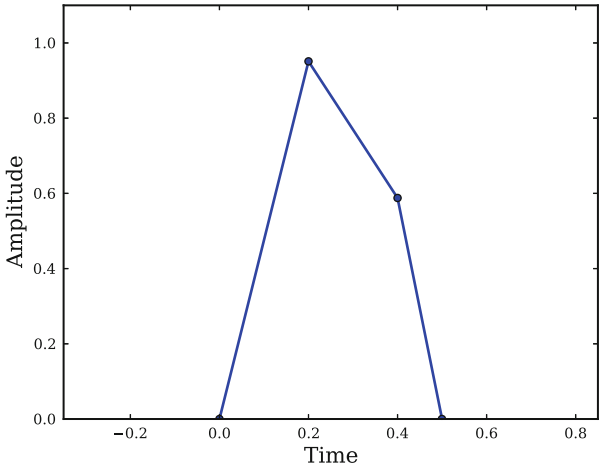


Fig. 2.2 Figure generated by Listing 2.2. This is a zoomed-in version of Listing 2.1

sampling theorem doesn't specify *where* we should sample as long as we sample at a periodic intervals. This means that on the up and down slopes of the sine, which are linear-looking and which would need fewer samples to characterize, we take the sample density as near the curvy peaks. Listing 2.2 zooms in to the first peak to illustrate this.

Listing 2.2 creates a new plot (e.g. lines, markers). The marker specification is the same as for MATLAB so `'o-'` means use a `'o'` marker and connect these with

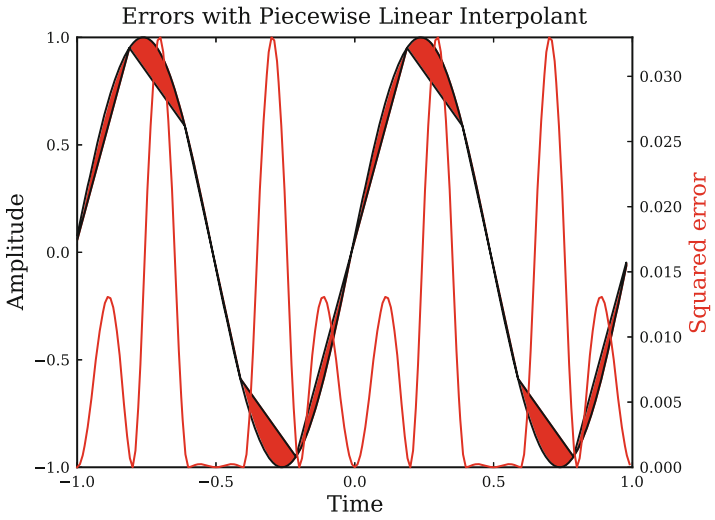


Fig. 2.3 Figure generated by Listing 2.4 showing the squared error of the sine function and its corresponding linear interpolant. Note that the piecewise approximation is worse where the sine is curviest and better where the sine is approximately linear. This is indicated by the *red-line* whose axis is on the *right side*

solid lines. The next line zooms into the plot by setting the axis limits. Note that we again use the keyword function style that makes it clear *what* we are changing on the axis. This is very useful for potential users of the code, including our forgetful future-selves. When possible, we recommend using this style for clarity.

To drive this point home, we can construct the piecewise linear interpolant and compare the quality of the approximation using `numpy.piecewise`. Listing 2.3 builds the arguments required for Numpy `piecewise` using the `hstack` function, which "stacks" Numpy arrays horizontally into a bigger Numpy array. The Python list `tp` has the intervals for the piecewise approximation and the `apprx` list contains the corresponding linear approximations for each of those intervals. The `logical_and` is necessary because we want an element wise logical operation. Finally, `tp` and `apprx` are fed into `piecewise` which packages these together to be used as a function later.

With all that set up, we can examine the squared errors in the interpolant. Listing 2.4 plots the sine with the filled-in error of the linear interpolant we just constructed.

Listing 2.4 uses the `fill_between` function to fill the convex area between the linear interpolant and the sine function with red (`facecolor='red'`). The following `twinx()` call creates a duplicate of the current axis (i.e. `ax2`) with a y-axis label on the right-side. The `ax2.set_ylabel` call attaches the y-axis label on the right-side where `color` changes the font color to red. Having separate axis variables (i.e. `ax1` and `ax2`) allows us to refer to them separately in the code. Next, we use this newly-created axis to plot the squared-error (SE) in red (`'r'`).

I invite you to download the IPython Notebook corresponding to this chapter and see what happens when you change the f_s sampling rate in the code and rerun the IPython cell. How do the errors change with more/fewer sampling points?

Now, we could pursue this line of reasoning with higher-order polynomials instead of just straight lines, but this would lead us to the same conclusion: all such approximations improve as the density of sample points increases. But this is the *exact opposite* of what the sampling theorem says: there is *sparse* set of samples that will retrieve the original function. Furthermore, we observed that the quality of the piecewise linear interpolant depends on *where* the sample points are taken whereas the sampling theorem *has no such requirement*.

2.2 Reconstruction

Let's look at this another way by examining the Fourier Transform of a bandlimited signal satisfying the hypothesis of the sampling theorem: $X(f) = 0$ where $|f| > W$. The inverse Fourier transform of this is the following:

$$x(t) = \int_{-W}^{W} X(f) e^{j2\pi ft} df \tag{2.3}$$

We can take the $X(f)$ and expand it into a Fourier series by pretending it is periodic with period $2W$. Thus, we can formally write the following:

$$X(f) = \sum_k a_k e^{-j2\pi kf/(2W)} \tag{2.4}$$

and compute the coefficients a_k as

$$a_k = \frac{1}{2W} \int_{-W}^{W} X(f) e^{j2\pi kf/(2W)} df \tag{2.5}$$

These coefficients bear a striking similarity to the $x(t)$ integral we just computed in Eq. 2.3. In fact, by lining up terms, we can write:

$$a_k = \frac{1}{2W} x\left(t = \frac{k}{2W}\right) \tag{2.6}$$

Now, we can write out $X(f)$ in terms of this series and these a_k and then invert the Fourier transform to obtain the following:

$$x(t) = \int_{-W}^{W} \sum_k a_k e^{-j2\pi kf/(2W)} e^{j2\pi ft} df \tag{2.7}$$

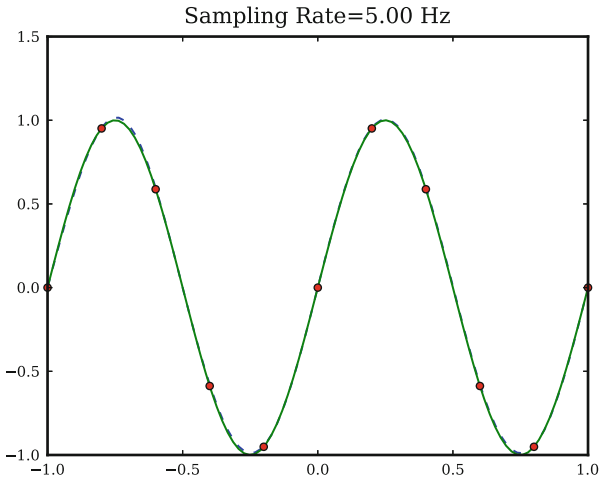


Fig. 2.4 Figure generated by Listing 2.5 showing the fit of `sinc`-based interpolator.

substitute for a_k,

$$x(t) = \int_{-W}^{W} \sum_k \frac{1}{2W} x\left(t = \frac{k}{2W}\right) e^{-j2\pi kf/(2W)} e^{j2\pi ft} df \tag{2.8}$$

switch summation and integration (usually dangerous, but okay here)

$$x(t) = \sum_k x\left(t = \frac{k}{2W}\right) \frac{1}{2W} \int_{-W}^{W} e^{-j2\pi kf/(2W)+j2\pi ft} df \tag{2.9}$$

which gives finally:

$$x(t) = \sum_k x\left(t = \frac{k}{2W}\right) \frac{\sin(\pi(k - 2tW))}{\pi(k - 2tW)} \tag{2.10}$$

And this what we have been seeking—a formula that reconstructs the function from its samples at $t = k/(2W)$. Note that the sinc function is defined as the following:

$$\text{sinc}(x) = \frac{\sin(\pi x)}{\pi x} \tag{2.11}$$

Because our samples are spaced at $t = k/f_s$, we'll use $W = f_s/2$ to line things up.

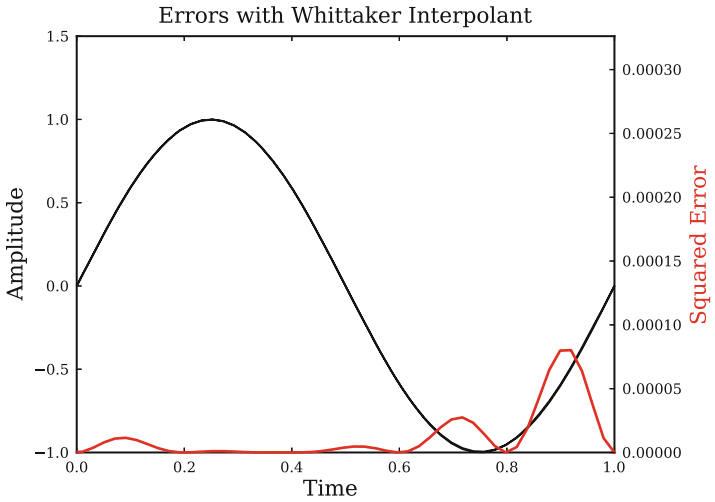


Fig. 2.5 Figure generated by Listing **2.6**. Note that the errors here are much smaller than those in Fig. 2.3

We can do the same evaluation of squared-error as we did for the linear interpolant above using the same code as Listing **2.4** with minor modifications shown in Listing **2.6**.

Figure 2.5 shows the massive reduction in squared-error resulting from using the *sinc* functions (scale on the right). This is *much better* than what we obtained from the linear interpolant in Fig. 2.3. In this context, the interpolating *sinc* functions are called the *Whittaker* interpolating functions. Let's examine these functions with the code in Listing **2.7**

Listing **2.7** introduces the `hlines`, `vlines`, and `ax.annotate` functions. The `hlines` and `vlines` functions draw horizontal and vertical lines on the plot, respectively. The `annotate` function draws arrows on the plot with corresponding text. This is a particularly powerful function because the arrow shapes and the text can be formatted separately with all of their respective options.

The first argument for `annotate` is the descriptive text itself. The keyword argument `xy` specifies the position of the tip of the arrowhead whereas the `xytext` argument indicates the position of the start of the text itself. Next, `arrowprops` is a Python dictionary that holds the properties for formatting the arrow. In this case, we specify the `facecolor` as `red` and the `shrink` factor as `0.05`, which moves the tip and base of the arrow a small percentage away from the text and annotation point.

Figure 2.6 shows three neighboring interpolation functions, one for each of three neighboring samples. Note that the peaks and zeros of these functions interleave. Therefore, at each of the sample points, there is no interference from any of the other functions because the others are all zero there. This is why the interpolated function matches the sample points exactly. In between sample points, the crown shape of the

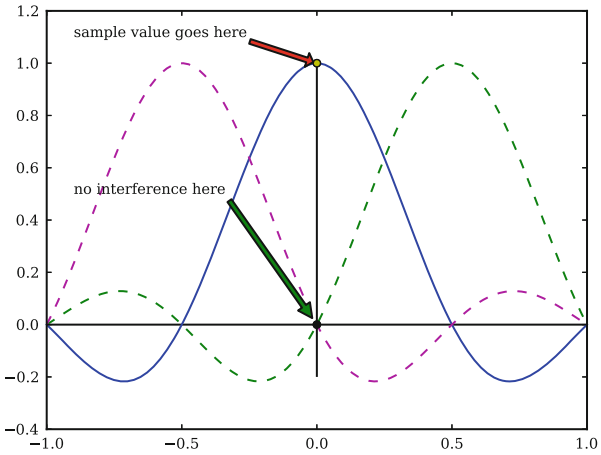


Fig. 2.6 Figure generated by Listing **2.7** showing how neighboring interpolating functions do not mutually interfere at the sample points

function fills in (i.e. interpolates) the missing values as shown in Fig. 2.6. Thus, the sinc functions provide the missing values, not straight lines as we presumed earlier. The width of these functions is directly related to the bandwidth of the signal which is part of the statement of the sampling theorem.

As an illustration, Listing **2.8** shows how the individual Whittaker functions (dashed lines) are assembled into the final approximation (black-line) using the given samples (blue-dots). We urge you to play with the sampling rate in the corresponding IPython Notebook and see what happens. Note the heavy use of Numpy broadcasting in this code instead of the multiple loops we used earlier.

Figure 2.5 shows the errors in the Whittaker interpolation, but why are there any errors? The sampling theorem didn't say anything about errors! Answering this question will take us deeper into the practical implications of the sampling theorem. This is the topic of our next section.

2.3 The Story So Far

Whatever conclusions we draw from our analysis in the digital domain must relate back to the original analog signal or they are not meaningful. The sampling theorem provides the mathematical justification for digital signal processing and thereby underpins the entire field. In this section, we started our investigation of the sampling theorem by asking if we could reverse-engineer it by reconstructing a signal from its discrete samples. This led us to derive the *Whittaker interpolator*, but this still did not *exactly* retrieve the original signal. The pursuit of an exact reconstruction is the topic of our next section. In the meantime, I invite you to

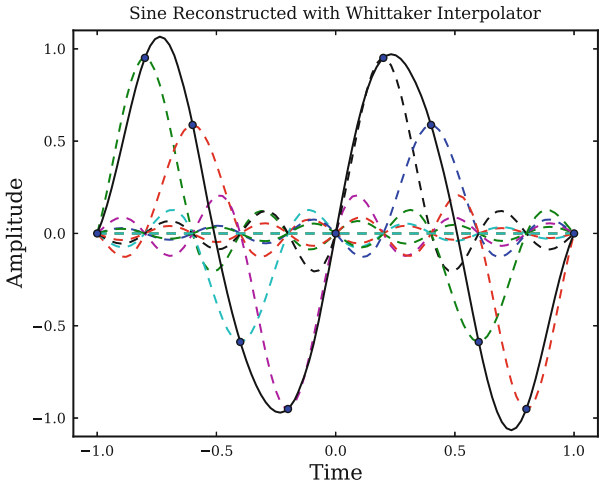


Fig. 2.7 Figure generated by Listing **2.8** showing how the sampled sine function is reconstructed (*solid line*) using individual Whittaker interpolators (*dashed lines*). Note each of the sample points sits on the peak of a sinc function

download the IPython Notebook corresponding to this section and play with the sampling frequency, and maybe even the sampled function and see what else you can discover about the sampling theorem.

2.4 Approximately Time-Limited-Functions

We left off with the disturbing realization that even though we satisfied the requirements of the sampling theorem, we still had errors in our approximating formula. We can resolve this by examining the Whittaker interpolating functions, which are used to reconstruct the signal from its samples.

Notice in Fig. 2.8 that the function extends to infinity in either direction. This basically means that the signals we can represent must also extend to infinity in either direction which then means that we have to sample forever to exactly reconstruct the signal! So, on the one hand, the sampling theorem says we only need a sparse density of samples, but this result says we need to sample forever.

This is a deep consequence of band-limited functions which, as we have just demonstrated, are *not* time-limited. Now, the new question is how to get these signals into a computer with finite memory? How can we use what we have learned about the sampling theorem with these finite-duration signals?

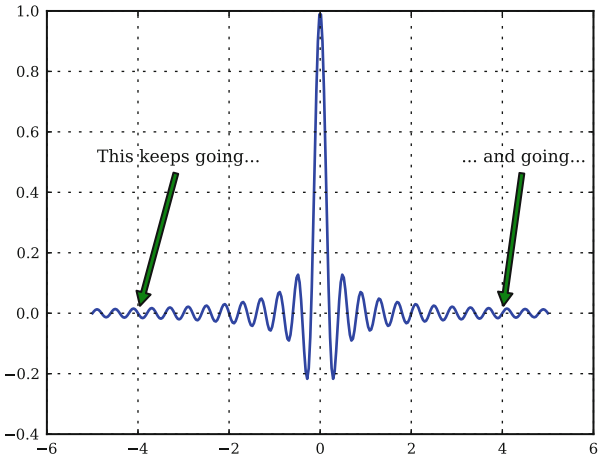


Fig. 2.8 Figure generated by Listing 2.9

Let's compromise and settle for functions that are *approximately* time-limited in the sense that almost all of their energy is concentrated in a finite time-window:

$$\int_{-\tau}^{\tau} |f(t)|^2 dt = E - \epsilon$$

where E is the total energy of the signal:

$$\int_{-\infty}^{\infty} |f(t)|^2 dt = E$$

Now, with this new definition, we can seek out functions that are band-limited but come very, very (i.e. within ϵ) close to being time-limited as well. In other words, we want functions $\phi(t)$ so that they are band-limited:

$$\phi(t) = \int_{-W}^{W} \Phi(\nu) e^{2\pi j \nu t} dt$$

and coincidentally maximize the following:

$$\int_{-\tau}^{\tau} |\phi(t)|^2 dt$$

After a complicated derivation, this boils down to solving the following eigenvalue equation:

$$\int_{-\tau}^{\tau} \phi(x) \frac{\sin(2\pi W(t-x))}{\pi(t-x)} dx = \lambda \phi(t)$$

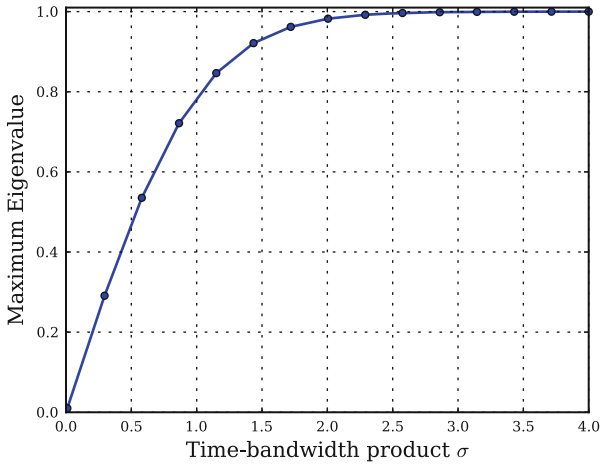


Fig. 2.9 Figure generated by Listing **2.10**

The set of $\phi_k(t)$ eigenfunctions form the basis for arbitrary approximately time-limited functions. In other words, we can express

$$f(t) = \sum_k a_k \phi_k(t)$$

Note that the $\phi_k(t)$ functions are not time-limited, but only time-concentrated in the $[-\tau, \tau]$ interval. With a change of variables, we can write this in normalized form as

$$\int_{-1}^{1} \psi(x) \frac{\sin(2\pi\sigma(t-x)/4)}{\pi(t-x)} dx = \lambda\psi(t)$$

where we define $\sigma = (2\tau)(2W)$ as the time-bandwidth product. The advantage of this change of variables is that τ and W are collected in a single term. Furthermore, this is the form of a classic problem where the ψ functions turn out to be the angular prolate spheroidal wave functions. Let's see what these ψ functions look like by solving this form of the eigenvalue problem in Listing **2.10**.

As shown in Fig. 2.9, the maximum eigenvalue quickly ramps up to almost one. The largest eigenvalue is the fraction of the energy contained in the interval $[-1, 1]$. Thus, this means that for $\sigma \gg 3$, $\psi_0(t)$ is the eigenfunction that is most concentrated in that interval. Now, let's look at this eigenfunction under those conditions shown in Fig. 2.10.

Note that we'll see the shape in Fig. 2.10 again when we take up window functions. What does this all mean? By framing our problem this way, we made a connection between the quality of our reconstruction via the Whittaker interpolant and the time-bandwidth product. Up until now, we did not have a concrete way of

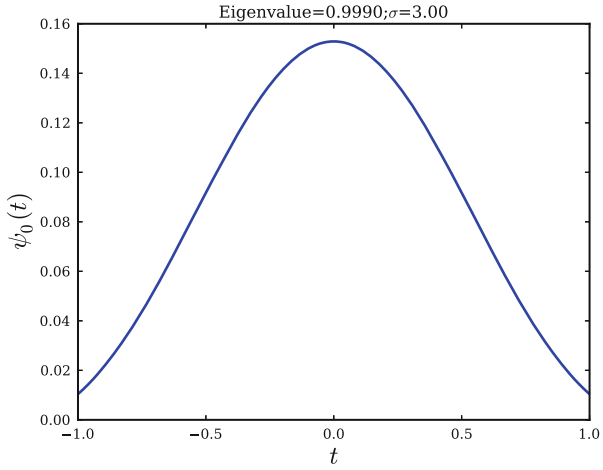


Fig. 2.10 Figure generated by Listing 2.10

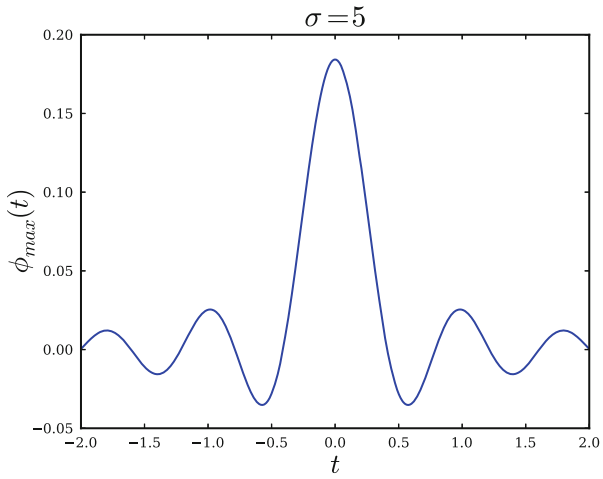


Fig. 2.11 Figure generated by Listing **2.12**

relating limitations in time to limitations in frequency. Now that we know how to use the time-bandwidth product, let's go back to the original formulation with the separate τ and W terms as in the following:

$$\int_{-\tau}^{\tau} \phi(x) \frac{\sin(2\pi W(t-x))}{\pi(t-x)} dx = \lambda \phi(t)$$

and then re-solve the eigenvalue problem.

Figure 2.11 looks suspicously like the sinc function. In fact, in the limit as $\sigma \rightarrow \infty$, the eigenfunctions devolve into time-shifted versions of the sinc function. These

are the same functions used in the Whittaker interpolant. Now we have a way to justify the interpolant by appealing to large σ values.

2.5 Summary

We started by investigating the residual error in the reconstruction formula using the Whittaker approximation functions. Then, we recognized that we cannot have signals that are simultaneously time-limited and band-limited. This realization drove us to investigate approximately time-limited functions. Through carefully examining the resulting eigenvalue problem, we determined the time-bandwidth conditions under which the Whittaker interopolant is asymptotically valid. As you can imagine, there is much more to this story, and many powerful theorems place bounds on the quality and dimensionality of this reconstruction, but for us, the qualifying concept of time-bandwidth product is enough for now.

Appendix

```
1  from __future__ import division
2
3  fig,ax = subplots()
4  f = 1.0  # Hz, signal frequency
5  fs = 5.0 # Hz, sampling rate (ie. >= 2*f)
6  t = arange(-1,1+1/fs,1/fs) # sample interval, symmetric
7                             # for convenience later
8  x = sin(2*pi*f*t)
9  ax.plot(t,x,'o-')
10 ax.set_xlabel('Time',fontsize=18)
11 ax.set_ylabel('Amplitude',fontsize=18)
```

Listing 2.1: The first line ensures that we use floating-point division instead of the default integer divide. Line 3 establishes the figure and axis bindings using `subplots`. Keeping these separate is useful for very complicated plots. The `arange` function creates a Numpy array of numbers. Then, we compute the sine of this array and plot it in the figure we just created. The `-o` is shorthand for creating a plot with solid lines and with points marked with `o` symbols. Attaching the `plot` to the `ax` variable is the modern convention for Matplotlib that makes it clear where the plot is to be drawn. The next two lines set up the labels for the x-axis and y-axis, respectively, with the specified font size.

```
1  fig,ax = subplots()
2  ax.plot(t,x,'o-')
3  ax.axis(xmin = 1/(4*f)-1/fs*3,
4          xmax = 1/(4*f)+1/fs*3,
5          ymin = 0,
6          ymax = 1.1 )
7  ax.set_xlabel('Time',fontsize=18)
8  ax.set_ylabel('Amplitude',fontsize=18)
```

Listing 2.2: Listing corresponding to Fig. 2.2. On the third line, we establish the limits of the axis using keyword arguments. This enhances clarity because the reader does not have to otherwise look up the positional arguments instead.

```
interval=[] # piecewise domains
apprx = []  # line on domains
# build up points *evenly* inside of intervals
tp = hstack([linspace(t[i],t[i+1],20,False) for i in range(len(t)-1)])
# construct arguments for piecewise
for i in range(len(t)-1):
   interval.append(logical_and(t[i] <= tp,tp < t[i+1]))
   apprx.append((x[i+1]-x[i])/(t[i+1]-t[i])*(tp[interval[-1]]-t[i]) + x[i])
x_hat = piecewise(tp,interval,apprx) # piecewise linear approximation
```

Listing 2.3: Code for constructing the piecewise linear approximation. The `hstack` function packs smaller arrays horizontally into a larger array. The remainder of the code formats the respective inputs for Numpy's piecewise linear interpolating function.

```
fig,ax1=subplots()
# fill in the difference between the interpolant and the sine
ax1.fill_between(tp,x_hat,sin(2*pi*f*tp),facecolor='red')
ax1.set_xlabel('Time',fontsize=18)
ax1.set_ylabel('Amplitude',fontsize=18)
ax2 = ax1.twinx() # create clone of ax1
sqe = (x_hat-sin(2*pi*f*tp))**2 #compute squared-error
ax2.plot(tp, sqe,'r')
ax2.axis(xmin=-1,ymax= sqe.max() )
ax2.set_ylabel('Squared error', color='r',fontsize=18)
ax1.set_title('Errors with Piecewise Linear Interpolant',fontsize=18)
```

Listing 2.4: Listing corresponding to Fig. 2.3. Line 2 uses `fill_between` to fill in the convex region between the `x_hat` and the `sin` function with the given `facecolor`. Because we want a vertical axis on both sides of the figure, we use the `twinx` function to create the duplicated axis. This shows the value of keeping separate variables for axes (i.e. `ax1`,`ax2`).

```
fig,ax=subplots()
t = linspace(-1,1,100)       # redefine this here for convenience
ts = arange(-1,1+1/fs,1/fs) # sample points
num_coeffs=len(ts)
sm=0
for k in range(-num_coeffs,num_coeffs): # since function is real, need both
   sides sm+=sin(2*pi*(k/fs))*sinc(k - fs*t)
ax.plot(t,sm,'--',t,sin(2*pi*t),ts, sin(2*pi*ts),'o')
ax.set_title('Sampling Rate=%3.2f Hz' % fs, fontsize=18 )
```

Listing 2.5: Code corresponding to Fig. 2.4 that shows that you can draw multiple lines with a single `plot` function. The only drawback is that you cannot later refer to the lines individually using the `legend` function. Note the squared-error here is imperceptible in this plot due to improved interpolant.

```
fig,ax1=subplots()
ax1.fill_between(t,sm,sin(2*pi*f*t),facecolor='red')
ax1.set_ylabel('Amplitude',fontsize=18)
ax1.set_xlabel('Time',fontsize=18)
ax2 = ax1.twinx()
sqe = (sm - sin(2*pi*f*t))**2
ax2.plot(t, sqe,'r')
ax2.axis(xmin=0,ymax = sqe.max())
ax2.set_ylabel('Squared Error', color='r',fontsize=18)
ax1.set_title(r'Errors with Whittaker Interpolant',fontsize=18)
```

Listing 2.6: Listing for Fig. 2.5. Note that on Line 8, we scale the y-axis maximum using the Numpy unary function (`max()`) attached to the `sqe` variable.

```
fig,ax=subplots()
k=0
fs=2 # makes this plot easier to read
ax.plot(t,sinc(k - fs * t),
        t,sinc(k+1 - fs * t),'--',k/fs,1,'o',(k)/fs,0,'o',
        t,sinc(k-1 - fs * t),'--',k/fs,1,'o',(-k)/fs,0,'o'
)
ax.hlines(0,-1,1)  # horizontal lines
ax.vlines(0,-.2,1) # vertical lines
ax.annotate('sample value goes here',
            xy=(0,1),          # arrowhead position
            xytext=(-1+.1,1.1),# text position
            arrowprops={'facecolor':'red',
                        'shrink':0.05},
            )
ax.annotate('no interference here',
            xy=(0,0),
            xytext=(-1+.1,0.5),
            arrowprops={'facecolor':'green','shrink':0.05},
            )
```

Listing 2.7: Listing corresponding to Fig. 2.6 introducing the `annotate` function that draws arrows with indicative text on the plot. The `shrink` key moves the tip and base of the arrow a small percentage away from the annotation point.

```
fs=5.0 # sampling rate
k=array(sorted(set((t*fs).astype(int)))) # sorted coefficient list
fig,ax = subplots()
ax.plot(t,(sin(2*pi*(k[:,None]/fs))*sinc(k[:,None]-fs*t)).T,'--', # individual
        whittaker functions t,(sin(2*pi*(k[:,None]/fs))*sinc(k[:,None]-fs*t)).
        sum(axis=0),'k-', # whittaker interpolant k/fs,
        sin(2*pi*k/fs),'ob')# samples
ax.set_xlabel('Time',fontsize=18)
ax.set_ylabel('Amplitude',fontsize=18)
ax.set_title('Sine Reconstructed with Whittaker Interpolator')
ax.axis((-1.1,1.1,-1.1,1.1));
```

Listing 2.8: Listing for Fig. 2.7. Line 4 uses Numpy broadcasting to create an implicit grid for evaluating the interpolating functions. The `.T` suffix is the transpose. The `sum(axis=0)` is the sum over the rows.

```
t = linspace(-5,5,300) # redefine this here for convenience
fig,ax = subplots()

fs=5.0     # sampling rate
ax.plot(t,sinc(fs * t))
ax.grid()  # put grid on axes
ax.annotate('This keeps going...',
            xy=(-4,0),
            xytext=(-5+.1,0.5),
            arrowprops={'facecolor':'green',
                        'shrink':0.05},
            fontsize=14)
ax.annotate('... and going...',
            xy=(4,0),
            xytext=(3+.1,0.5),
            arrowprops={'facecolor':'green',
                        'shrink':0.05},
            fontsize=14)
```

Listing 2.9: Listing corresponding to Fig. 2.8.

```
def kernel(x,sigma=1):
    'convenient function to compute kernel of eigenvalue problem'
    x = np.asanyarray(x)            # ensure x is array
    y = pi*where(x == 0,1.0e-20, x)# avoid divide by zero
    return sin(sigma/2*y)/y

nstep=100                 # quick and dirty integral quantization
t = linspace(-1,1,nstep)  # quantization of time
dt = diff(t)[0]           # differential step size
def eigv(sigma):
    return eigvalsh(kernel(t-t[:,None],sigma)).max() # compute max eigenvalue

sigma = linspace(0.01,4,15) # range of time-bandwidth products to consider

fig,ax = subplots()
ax.plot(sigma, dt*array([eigv(i) for i in sigma]),'-o')
ax.set_xlabel('Time-bandwidth product $\sigma$',fontsize=18)
ax.set_ylabel('Maximum Eigenvalue',fontsize=18)
ax.axis(ymax=1.01)
ax.grid()
```

Listing 2.10: Listing corresponding to Fig. 2.9. The `eigvalsh` function comes from the LAPACK compiled library.

```
sigma=3  # time-bandwidth product
w,v=eigh(kernel(t-t[:,None],sigma)) # eigen-system
maxv=v[:, w.argmax()]               # eigenfunction for max eigenvalue
fig,ax=subplots()
ax.plot(t,maxv)
ax.set_xlabel('$t$',fontsize=22)
ax.set_ylabel('$\psi_0(t)$',fontsize=22)
ax.set_title('Eigenvalue=%3.4f;$\sigma$=%3.2f'%(w.max()*dt,sigma))
```

Listing 2.11: Listing corresponding to Fig. 2.10. The `argmax` function finds the array index corresponding to the array maximum.

```
def kernel_tau(x,W=1):
    'convenient function to compute kernel of eigenvalue problem'
    x = np.asanyarray(x)
    y = pi*where(x == 0,1.0e-20, x) # avoid divide by zero
    return sin(2*W*y)/y

nstep=300                # quick and dirty integral quantization
t = linspace(-1,1,nstep) # quantization of time
tt = linspace(-2,2,nstep)# extend interval
sigma = 5
W = sigma/2./2./t.max()
w,v=eig(kernel_tau(t-tt[:,None],W)) # compute e-vectors/e-values
maxv=v[:,w.real.argmax()].real      # take real part

fig,ax = subplots()
ax.plot(tt,maxv/sign(maxv[nstep/2])) # normalize for orientation
ax.set_xlabel('$t$',fontsize=24)
ax.set_ylabel(r'$\phi_{max}(t)$',fontsize=24)
ax.set_title('$\sigma=%d$'%(2*W*2*t.max()),fontsize=26)
```

Listing 2.12: Listing corresponding to Fig. 2.11. the `sign` function computes the sign of the argument.

```
def facet_filled(x,alpha=0.5,color='b'):
    'construct 3D facet from adjacent points filled to zero'
    a,b=x
    a0= a*array([1,1,0])
    b0= b*array([1,1,0])
    ve = vstack([a,a0,b0,b])       # create closed polygon facet
    poly = Poly3DCollection([ve]) # create facet
    poly.set_alpha(alpha)          # set transparency
    poly.set_color(color)
    return poly

def drawDFTView(X,ax=None,fig=None):
    'Draws 3D diagram given DFT matrix'
    a=2*pi/len(X)*arange(len(X))
    d=vstack([cos(a),sin(a),array(abs(X)).flatten()]).T
    if ax is None and fig is None:
        fig = plt.figure()
        fig.set_size_inches(6,6)

    if ax is None: # add ax to existing figure
        ax = fig.add_subplot(1, 1, 1, projection='3d')

    ax.axis([-1,1,-1,1])           # x-y limits
    ax.set_zlim([0,d[:,2].max()]) # z-limit
    ax.set_aspect(1)               # aspect ratio
    ax.view_init(azim=-30)         # camera view position
    a=FancyArrow(0,0,1,0,width=0.02,length_includes_head=True)
    ax.add_patch(a)
    b=FancyArrow(0,0,0,1,width=0.02,length_includes_head=True)
    ax.add_patch(b)
    art3d.patch_2d_to_3d(a) # format 2D patch for 3D plot
    art3d.patch_2d_to_3d(b)
    ax.axis('off')

    sl=[slice(i,i+2) for i in range(d.shape[0]-2)] # collect neighboring points
    for s in sl:
      poly=facet_filled(d[s,:])
      ax.add_collection3d(poly)

    # edge polygons
    ax.add_collection3d(facet_filled(d[[-1,0],:]))
    ax.add_collection3d(facet_filled(d[[-2,-1],:]))
```

```
def drawInOut(X,v,return_axes=False):
    fig = plt.figure()
    fig.set_size_inches(8,8)
    gs = gridspec.GridSpec(8,6)

    ax1 = plt.subplot(gs[3:5,:2])
    ax2 = plt.subplot(gs[:,2:],projection='3d')


    ax1.stem(arange(len(v)),v)
    ymin,ymax= ax1.get_ylim()
    ax1.set_ylim(ymax = ymax*1.2, ymin = ymin*1.2)
    ax1.set_title('Signal')
    ax1.set_xlabel('n')
    ax1.tick_params(labelsize=8)

    drawDFTView(X,ax2)
    if return_axes:
        return ax1,ax2
```

Listing 2.13: Setup code for gyphs. The `add_patch` function places graphics primitives ("patches") like rectangles and arrows onto the plot. The `FancyArrow` is one of the graphics primitives for an arrow. The `gridspec` is a more powerful tool than `subplots` for controlling placement of plots on a grid.

Chapter 3
Discrete-Time Fourier Transform

3.1 Fourier Transform Matrix

Let us start with the following DFT matrix

$$\mathbf{U} = \frac{1}{\sqrt{N}}\left[\exp\left(j\frac{2\pi}{N}nk\right)\right]_{n\in\{0,N_s-1\},k\in\{0,N-1\}} \tag{3.1}$$

where n counts the number of samples and k indexes the discrete frequencies as columns.

Figure 3.1 shows the discrete frequencies on the unit circle and their corresponding real and imaginary parts that are the columns of $\mathbf{U}$. The pinwheel on the left shows each discrete frequency on the unit circle corresponding to each of the columns of the $\mathbf{U}$ matrix shown on the right. These are color coded corresponding to the graphs on the right. For example, the $k = 1$ column of the $\mathbf{U}$ matrix (i.e. $\mathbf{u}_1$) corresponds to discrete frequency $\omega_1 = \frac{2\pi}{16}$ marked on the y-axis label which is shown in the second row down the middle column in the figure. The real part of $\mathbf{u}_1$ is plotted in bold and the corresponding imaginary part is muted in the background because it is just an out-of-phase version of the real part. These real/imaginary parts shown in the graphs correspond to the conjugacy relationships on the leftmost radial plot. For example, ω_1 and ω_{15} are complex conjugates and their corresponding imaginary parts are inverted as shown in the plots on the right. Figure 3.1 is the most important graphic in this *entire* book so please make sure to review it carefully.

As shown, N divides the unit circle into discrete frequencies between zero and 2π. It also divides the sample rate into sampled frequencies between zero and f_s. This is because sampling at $t = n/f_s$ means that

$$\exp(j2\pi ft) \Rightarrow \exp(j2\pi fn/f_s)$$

J. Unpingco, *Python for Signal Processing*, DOI 10.1007/978-3-319-01342-8_3,

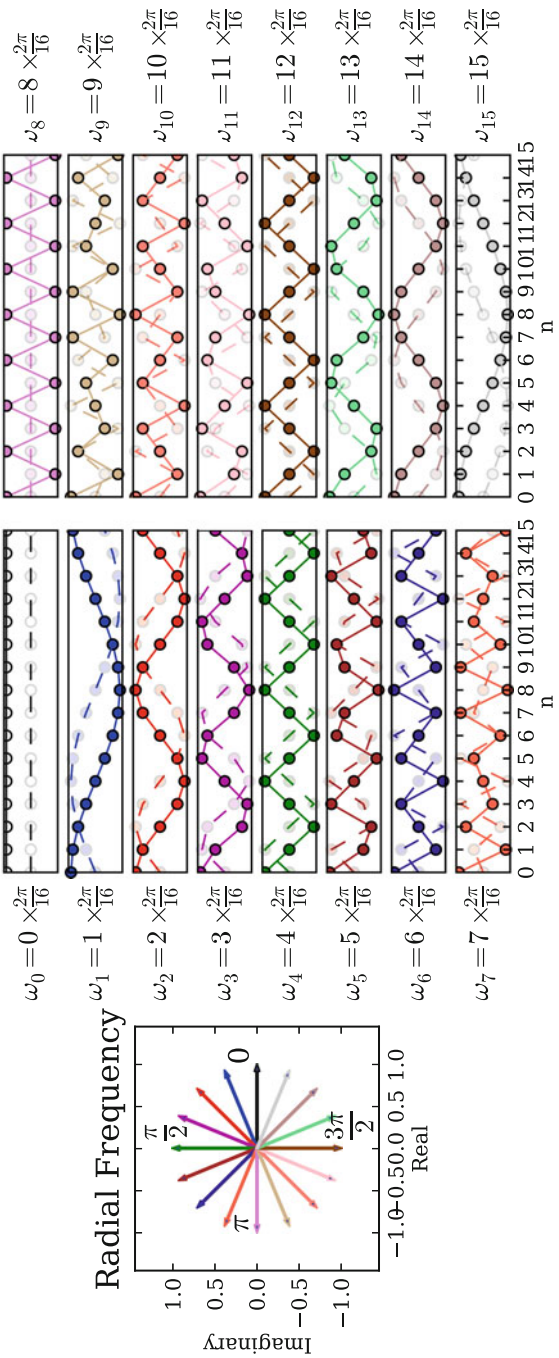


Fig. 3.1 Figure generated by Listing 3.1 showing the columns of the DFT matrix and their corresponding positions on the unit circle

and equating the argument of the exponential with Eq. 3.1 gives

$$f_k = \frac{k}{N} f_s$$

One immediate consequence of the one-to-one correspondence between ω_k and f_k is that when $k = N/2$, we have $\omega_{N/2} = \pi$ (halfway around the circle) and $f_{N/2} = f_s/2$ which is another way of saying that the Nyquist frequency occurs when $\omega_{N/2} = \pi$. Qualitatively, we can see this in Fig. 3.1 where the counterclockwise rotation around the unit circle towards π corresponds to increasingly jagged plots on the right. These plots get smoother as the rotation passes π and swings back towards zero. This is because the higher frequencies are those close to π and the lower frequencies are those close to zero on the complex plane. We will explore these crucial relationships more later, but let's first consider computing the DFT using this matrix.

3.2 Computing the DFT

To compute the DFT using the matrix, we calculate the following,

$$\hat{\mathbf{x}} = \mathbf{U}^H \mathbf{x}$$

which individually takes each of the columns of $\mathbf{U}$ and computes the complex inner product as the ith entry,[1]

$$\hat{\mathbf{x}}_i = \mathbf{u}_i^H \mathbf{x}$$

That is, we are measuring the *degree of similarity* between each column of $\mathbf{U}$ and the input vector. We can think of this as the coefficient of the projection of $\mathbf{x}$ onto $\mathbf{u}_i$. We can retrieve the original input from the DFT by calculating

$$\mathbf{x} = \mathbf{U}\mathbf{U}^H \mathbf{x}$$

because the columns of $\mathbf{U}$ are orthonormal (i.e. $\mathbf{u}_i^H \mathbf{u}_j = 0, \mathbf{u}_i^H \mathbf{u}_i = 1$). An important consequence of this is that

$$||\mathbf{x}||^2 = \mathbf{x}^H \mathbf{x} = \mathbf{x}^H \mathbf{U}\mathbf{U}^H \mathbf{x} = ||\hat{\mathbf{x}}||^2$$

[1] The H superscript indicates the conjugate transpose.

This is Parseval's theorem and it means that the DFT does not somehow "lose" signal energy as we transform back and forth between discrete frequency and sampled-time. As we'll see later, not losing energy in *total* doesn't account for how that energy may be spread out in frequency.

3.3 Understanding Zero-Padding

The only relationship between N, the size of the DFT, and the number of samples N_s is that $N \geq N_s$. In practice, because we use the Fast Fourier Transform (FFT) to compute this, we always choose N as a power of 2. Let's now turn to the consequences of choosing N larger than N_s. Figure 3.2 shows the magnitude of the DFT of $\mathbf{x} = \mathbf{1} \in \mathbb{R}^{16}$ with $N_s = 16$ samples with DFT-length, $N = 64$.

If you've been following closely, you may realize that for the above example we had $\mathbf{x} = \mathbf{1}$. But isn't this one of the columns of the $\mathbf{U}$ matrix? If all the columns of that matrix are orthonormal, then why is there more than one non-zero point on this graph? The subtle point here is that the DFT matrix has dimensions 64×16. This means that computationally,

$$\mathbf{U}_{16\times 64}^{H}\mathbf{x} = \mathbf{U}_{64\times 64}^{H}\,[\mathbf{x}, \mathbf{0}]^{T}$$

In other words, filling the original 16×1 vector $\mathbf{x}$ with zeros and using a larger compatible $\mathbf{U}_{64\times 64}$ matrix has the same effect as using the $\mathbf{U}_{16\times 64}$ matrix. The answer to the question is therefore that $\mathbf{x} = \mathbf{1}_{16\times 1} \neq [\mathbf{1}_{16\times 1}, \mathbf{0}]^{T}$ and the zero-augmented ones vector is *not* orthonormal to any columns in $\mathbf{U}_{64\times 64}$. This explains why there are so many non-zero points on the graph at different discrete frequencies.

As shown in Fig. 3.3, without zero-padding, $\mathbf{x}$ is the 0^{th} column of the 16-point DFT matrix and so all the coefficients except for the 0^{th} column are zero due to orthonormality (shown by the green squares). But, the zero-padded 64-element-long $\mathbf{x}$ vector is definitely *not* a column of the 64-point DFT matrix so we would *not*

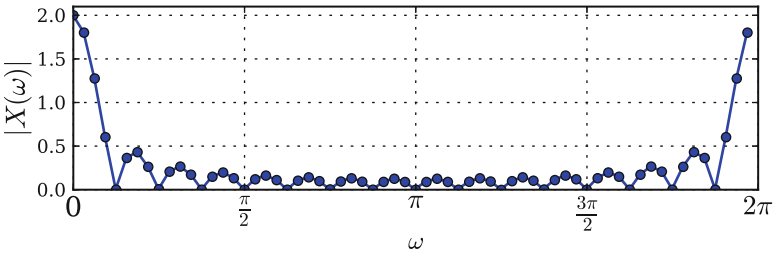


Fig. 3.2 Figure generated by Listing 3.2 showing the magnitude of the DFT of $\mathbf{x} = \mathbf{1} \in \mathbb{R}^{16}$ with $N_s = 16$ samples with DFT-length, $N = 64$

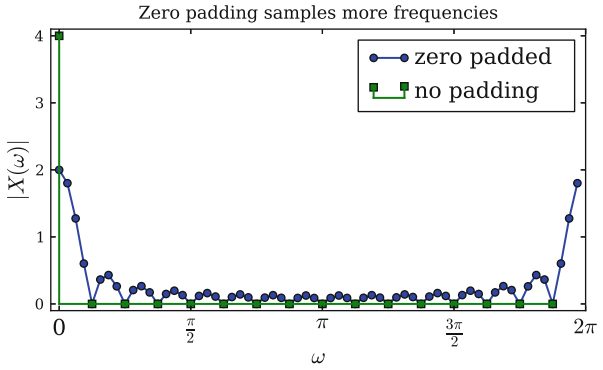
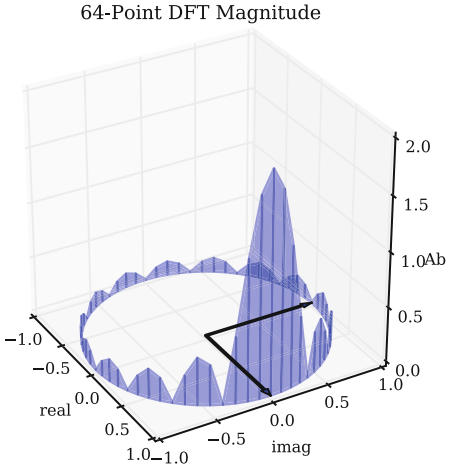


Fig. 3.3 Figure generated by Listing **3.3**

Fig. 3.4 Figure generated by Listing **3.6** showing the magnitude of the $X(\omega_k)$ 64-point DFT as in Fig. 3.2, but now plotted on the face of a cylinder, emphasizing its periodicity



expect all the other terms to be zero. In fact, the other terms account for the 63 other discrete frequencies that are plotted in Fig. 3.3.

In Fig. 3.3, note that for the 0^{th} frequency, the DFT magnitude is much smaller for the zero-padded constant signal compared to the unpadded version. Recall from Parseval's theorem that $||\mathbf{x}|| = ||\hat{\mathbf{x}}||$ so no energy is "lost", but this does not account for how that energy may be spread across frequency. In the unpadded case, *all* of the signal energy is concentrated in the $\mathbf{u}_0$ column of the DFT matrix because our constant signal is just a scalar multiple of $\mathbf{u}_0$. In the padded case, the signal's energy is spread out across many frequencies with smaller signal magnitudes per frequency, thus satisfying Parseval's theorem. In other words, the single non-zero term in the unpadded DFT is smeared out over all the other frequencies in the padded case.

The problem with Fig. 3.3 is that it does not emphasize that the discrete frequencies are periodic with period N. Figure 3.4 attempts to remedy this by

plotting the 64-point DFT on the face of a three-dimensional cylinder to emphasize the periodicity of the discrete frequencies. Figure 3.4 is very important and we will use it as a glyph later. It shows the same magnitude of the $X(\omega_k)$ 64-point DFT as in Fig. 3.2, but now plotted on the face of a cylinder, we can really see the periodic discrete frequencies. The two arrows in the xy-plane show the discrete frequencies zero and π, respectively, for reference. The code that draws this glyph is collected in Listing **3.6**.

Figure 3.5 shows the symmetric lobes of the DFT of a real signal. The plot on the left is the signal in the sampled time-domain and the plot on the right is its DFT-magnitude glyph. Because the input signal is real, the DFT is symmetric. Recall that in Fig. 3.1, every $\mathbf{u}_i$ had its complex conjugate, $\mathbf{u}_{N-i}$, and since the real parts of complex conjugates are the same and there is no imaginary part in the real-valued

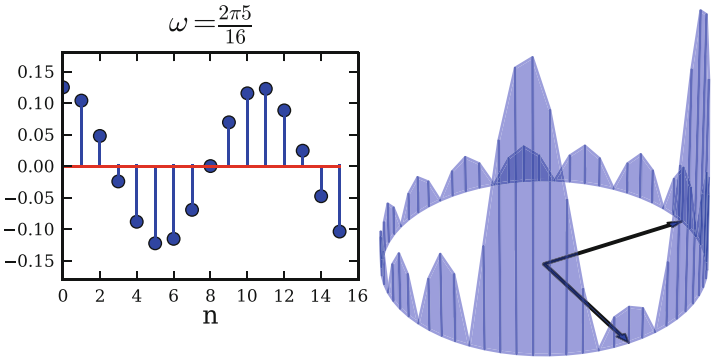


Fig. 3.5 Figure generated by Listing **2.13** showing the symmetric lobes of the DFT of a real signal

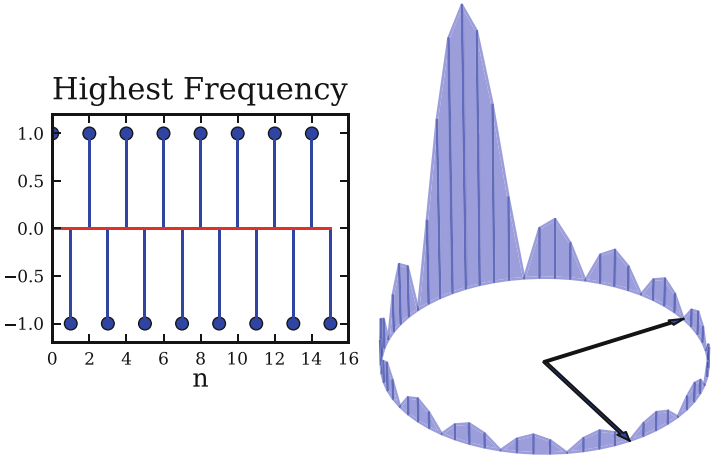


Fig. 3.6 Figure generated by Listing **3.5** showing the highest-frequency signal

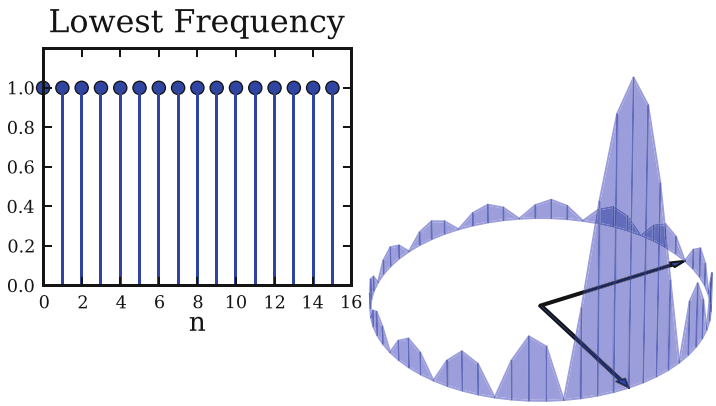


Fig. 3.7 Figure generated by Listing **3.5** showing the lowest-frequency signal

input signal, the resulting corresponding inner products are complex conjugates and thus have the same magnitudes. The block of code in Listing **3.4** illustrates this in Fig. 3.5. This fact has extremely important computational consequences for the fast implemention of the DFT (i.e. FFT), but that is a story for another day. For now, it's enough to recognize the symmetry of the DFT of real signals and how it arises.

Now that we have enough vocabulary defined, we can ask one more intuitive question: what does the highest frequency signal (i.e. $\omega_{N/2} = \pi$) look like in the sampled time-domain? This is shown on the left in Fig. 3.6 where a signal toggles back and forth positive and negative. Note that the amplitudes of this toggling are not important, it is the *rate* of toggling that defines the high frequency signal. At the other extreme, Fig. 3.7 shows the lowest frequency signal (i.e. $\omega_0 = 0$). Note that it is the mirror image of the high frequency signal. Please download the IPython notebook corresponding to this section and play with these plots to develop an intuition for where the various input signals appear.

3.4 Summary

In this section, we considered the Discrete-Time Fourier Transform (DFT) using a matrix/vector approach. We used this approach to develop an intuitive visual vocabulary for the DFT with respect to high/low frequency and real-valued signals. We used zero-padding to enhance frequency domain signal analysis and examined the consequences of Parseval's theorem.

Appendix

```
# must start notebook with --pylab flag

from matplotlib.patches import FancyArrow
import mpl_toolkits.mplot3d.art3d as art3d
from mpl_toolkits.mplot3d.art3d import Poly3DCollection
import matplotlib.gridspec as gridspec

def dftmatrix(Nfft=32,N=None):
    'construct DFT matrix'
    k= np.arange(Nfft)
    if N is None: N = Nfft
    n = arange(N)
    U = matrix(exp(1j* 2*pi/Nfft *k*n[:,None])) # use numpy broadcasting to
    create matrix return U/sqrt(Nfft)

Nfft=16
v = ones((16,1))
U = dftmatrix(Nfft=Nfft,N=16)
# ---
# hardcoded constants to format complicated figure

gs = gridspec.GridSpec(8,12)
gs.update( wspace=1, left=0.01)

fig =figure(figsize=(10,5))
ax0 = subplot(gs[:,:3])
fig.add_subplot(ax0)

ax0.set_aspect(1)
a=2*pi/Nfft*arange(Nfft)

colors = ['k','b','r','m','g','Brown','DarkBlue','Tomato','Violet', 'Tan',
          'Salmon','Pink','SaddleBrown', 'SpringGreen', 'RosyBrown','Silver',]
for j,i in enumerate(a):
  ax0.add_patch(FancyArrow(0,0,cos(i),sin(i),width=0.02,
                           length_includes_head=True,edgecolor=colors[j]))

ax0.text(1,0.1,'0',fontsize=16)
ax0.text(0.1,1,r'$\frac{\pi}{2}$',fontsize=22)
ax0.text(-1,0.1,r'$\pi$',fontsize=18)
ax0.text(0.1,-1.2,r'$\frac{3\pi}{2}$',fontsize=22)
ax0.axis(array([-1,1,-1,1])*1.45)
ax0.set_title('Radial Frequency',fontsize=18)
ax0.set_xlabel('Real')
ax0.set_ylabel('Imaginary')
```

```
# plots in the far right column
for i in range(8):
  ax=subplot(gs[i,8:])
  ax.set_xticks([]);  ax.set_yticks([])
  ax.set_ylabel(r'$\omega_{%d}=%d\times\frac{2\pi}{16}$'%(i+8,i+8),fontsize=16,
                rotation='horizontal')
  ax.plot(U.real[:,i+8],'-o',color=colors[i+8])
  ax.plot(U.imag[:,i+8],'--o',color=colors[i+8],alpha=0.2)
  ax.axis(ymax=4/Nfft*1.1,ymin=-4/Nfft*1.1)
  ax.yaxis.set_label_position('right')
ax.set_xticks(arange(16))
ax.set_xlabel('n')
```

Listing 3.1: Listing corresponding to Fig. 3.1. The `yaxis.set_label_position` function places the y-label on the right side of the figure instead of on the default left side. Using `set_xticks` to set the tick labels list to the empty list (`[]`) removes all the tick marks on the x-axis. The `set_xticklabels` functions sets the labels on the tickmarks to the specified list of LaTeX formatted strings. Matplotlib's `gridspec` is a generalization of subplot that allows more precise control of the layout of nested plots. The `art3d` module holds the codes for converting 2D elements into 3D elements that can be added to 3D axes. The `Poly3DCollection` object is a container for 3D elements.

```
U = dftmatrix(64,16)
x = ones((16,1))
X = U.H*x

fig,ax=subplots()
fig.set_size_inches((8,4))
ax.set_aspect(0.8)
ax.grid()
ax.plot(arange(0,64)*2*pi/64.,abs(X),'o-')
ax.set_ylabel(r'$|X(\omega)|$',fontsize=18)
ax.set_xticks([0, pi/2., pi, 3*pi/2,2*pi])
ax.set_xlabel(r'$\omega$',fontsize=16)
ax.axis([0, 2*pi,0,2.1])
ax.set_xticklabels(['0',r'$\frac{\pi}{2}$', r'$\pi$',r'$\frac{3\pi}{2}$',
                    r'$2\pi$'], fontsize=18);
```

Listing 3.2: Listing corresponding to Fig. 3.2. The `set_xticklabels` function changes the labeling of the tick marks to the given string, here with extra LaTeX formatting.

```
U = dftmatrix(64,16)
x = ones((16,1))
X = U.H*x

fig,ax=subplots()
fig.set_size_inches((8,4))

ax.set_aspect(0.8) # aspect ratio
ax.plot(arange(0,64)*2*pi/64.,abs(X),'o-',label='zero padded')
ax.stem(arange(0,16)*2*pi/16.,abs(dftmatrix(16).H*x),
        markerfmt='gs', basefmt='g-',linefmt='g-',
        label='no padding')
ax.set_xlabel(r'$\omega$',fontsize=18)
ax.set_ylabel(r'$|X(\omega)|$',fontsize=18)
ax.set_xticks([0, pi/2., pi, 3*pi/2,2*pi])
ax.axis([-.1, 2*pi,-.1,4.1])
ax.legend(loc=0,fontsize=18)
ax.set_xticklabels(['0',r'$\frac{\pi}{2}$', r'$\pi$',r'$\frac{3\pi}{2}$',
                    r'$2\pi$'], fontsize=18);
ax.set_title('Zero padding samples more frequencies');
```

Listing 3.3: Listing corresponding to Fig. 3.3.

```
v = U[:,6].real
ax1,ax2=drawInOut(U.H*v,v,return_axes=1)
ax1.set_title(r'$\omega=\frac{2\pi 5}{16}$')
```

Listing 3.4: Listing corresponding to Fig. 3.5.

```
v = matrix(cos(pi*arange(0,16))).T
ax1,ax2=drawInOut(U.H*v,v,return_axes=1)
ax1.set_title('Highest Frequency')
v = ones((16,1))
ax1,ax2=drawInOut(U.H*v,v,return_axes=1)
ax1.set_title('Lowest Frequency')
```

Listing 3.5: Listing corresponding to Fig. 3.7.

```
a=2*pi/64.*arange(64)
d=vstack([cos(a),sin(a),array(abs(X)).flatten()]).T

fig = plt.figure()
fig.set_size_inches(6,6)
ax = fig.add_subplot(1, 1, 1, projection='3d')
ax.axis([-1,1,-1,1])
ax.set_zlim([0,d[:,2].max()])
ax.set_aspect(1)
ax.view_init(azim=-30)

ax.set_xlabel('real')
ax.set_ylabel('imag')
ax.set_zlabel('Abs')
ax.set_title('64-Point DFT Magnitude')

def facet_filled(x,alpha=0.5,color='b'):
    'construct 3D facet from adjacent points filled to zero'
    a,b=x
    a0= a*array([1,1,0])
    b0= b*array([1,1,0])
    ve = vstack([a,a0,b0,b])       # create closed polygon facet
    poly = Poly3DCollection([ve]) # create facet
    poly.set_alpha(alpha)
    poly.set_color(color)
    return poly

sl=[slice(i,i+2) for i in range(d.shape[0]-2)] # collect neighboring points
for s in sl:
  poly=facet_filled(d[s,:])
  ax.add_collection3d(poly)

# edge polygons
ax.add_collection3d(facet_filled(d[[-1,0],:]))
ax.add_collection3d(facet_filled(d[[-2,-1],:]))

# add 0 and pi/2 arrows for reference
a=FancyArrow(0,0,1,0,width=0.02,length_includes_head=True)
ax.add_patch(a)
b=FancyArrow(0,0,0,1,width=0.02,length_includes_head=True)
ax.add_patch(b)
art3d.patch_2d_to_3d(a)
art3d.patch_2d_to_3d(b)
plt.show()
```

Listing 3.6: Listing corresponding to Fig. 3.4.

Chapter 4
Introducing Spectral Analysis

In this section, we try to separate two nearby tones using the Discrete Fourier Transform (DFT). This problem is fundamental to signal processing, and we will develop the circular convolution as a tool to understand it, as we once again confront the uncertainty principle. Separating tones is important because scientific apparatus are often designed to indirectly measure physical quantities by associating them with frequencies. For example, radar converts relative velocity into frequency so that by resolving nearby frequencies, you can measure differential velocity.

4.1 Seeking Better Frequency Resolution with Longer DFT

The top plot in Fig. 4.1 above shows the magnitude of the DFT for the sum of two equal-amplitude tones separated by 2 Hz. Using the parameters we have chosen for the DFT, we can easily see there are two distinct frequencies in the input signal. However, in the bottom plot, the same tones are only separated by 0.5 Hz and cannot be distinguished in the DFT. Because the frequency resolution is f_s/N, could we increase N and thereby resolve the two tones? Let's try it as shown in Fig. 4.2.

As Fig. 4.2 shows, increasing the size of the DFT did not help separate our two tones. Didn't we increase the frequency resolution using a longer DFT? Why can't we separate frequencies now?

4.2 The Uncertainty Principle Strikes Back!

The resolution problem is a consequence of the uncertainty principle we discussed in Chap. 2. Recall that the uncertainty principle basically states that we cannot have arbitrarily fine frequency-resolution unless we have a longer time-durations (i.e. worse time-resolution). Because we always have a *finite* slice of signal to analyze, what we really have are samples of the product of a signal $x(t)$ and a

J. Unpingco, *Python for Signal Processing*, DOI 10.1007/978-3-319-01342-8_4,

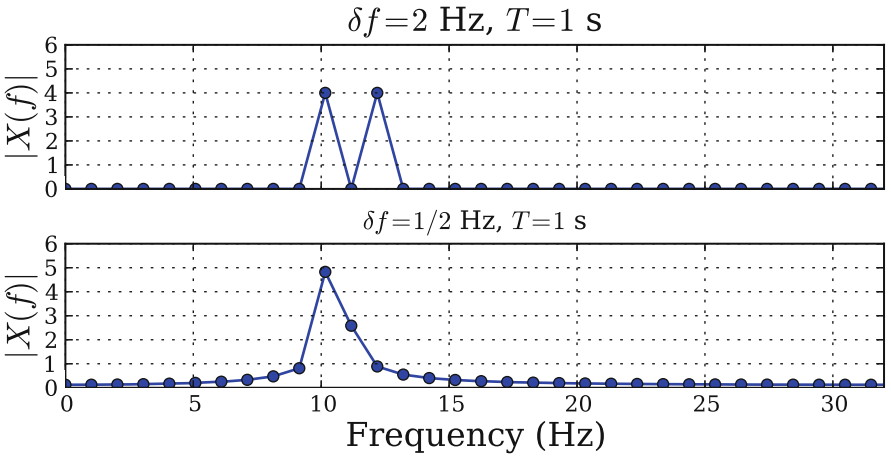


Fig. 4.1 Figure generated by Listing 4.1. In the *top plot*, we can clearly distinguish two tones separated by 2 Hz by the magnitude of the DFT shown. However, when these tones are separated by only 0.5 Hz, we can no longer do so (*bottom plot*)

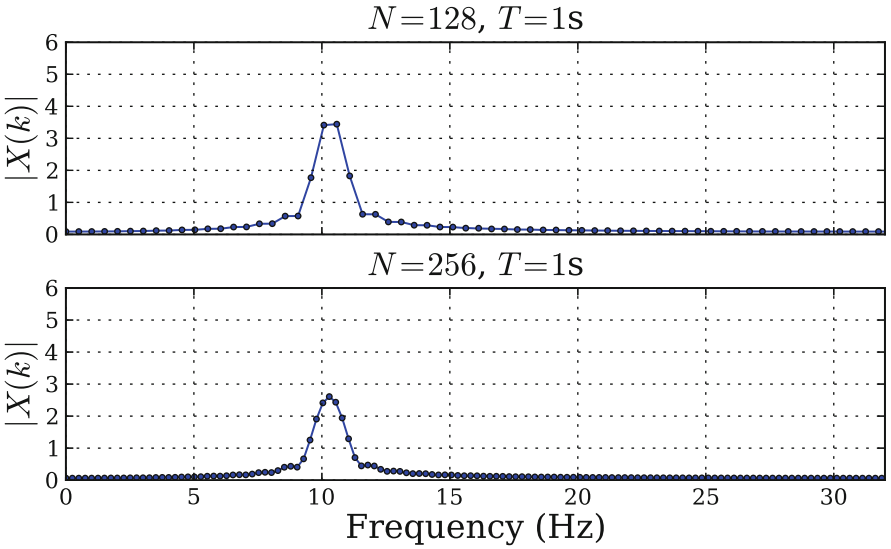


Fig. 4.2 Figure generated by Listing 4.2. In the *top plot*, we keep everything as in Fig. 4.1, but increase the DFT length to $N = 128$. Unfortunately, we still cannot clearly distinguish the two tones. Going up to $N = 256$ in the *bottom plot* does no better

rectangular time-window, $r(t)$, that is always zero except $r(t) = 1 \Leftrightarrow t \in [0, 1]$. To understand frequency resolution, we have to understand how this rectangular window influences the DFT.

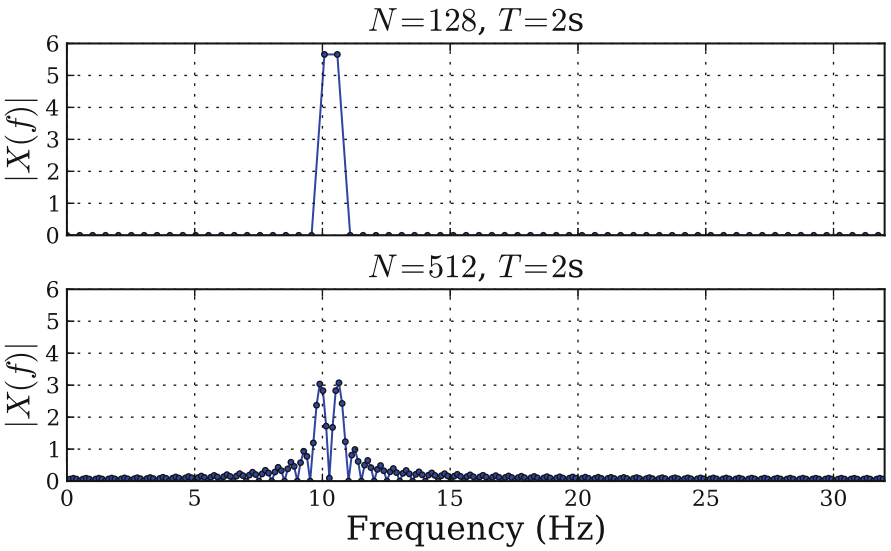


Fig. 4.3 Figure generated by Listing **4.3**. By sampling through a longer duration, we can now distinguish the two tones using a longer DFT (*bottom plot*), but still not so in the *top plot* with the shorter DFT. Clearly, we need both larger DFTs and longer sampling durations to resolve these two nearby tones, not just longer DFTs

The $N = 128$ DFT in Fig. 4.2 is updated with a longer duration rectangular window and shown at the top of Fig. 4.3. The bottom plot shows the DFT of the same pair of signals $N = 512$, but this time we can clearly distinguish two frequencies. Thus, in this case, a longer DFT *did* resolve the nearby frequencies, but only by using a longer duration signal. Why is this? Consider the DFT of a rectangular window of length N_s,

$$R_k = \frac{1}{\sqrt{N}} \sum_{n=0}^{N_s-1} \exp\left(\frac{2\pi}{N} k n\right)$$

after some re-arrangement, this reduces to

$$|R_k| = \frac{1}{\sqrt{N}} \left| \frac{\sin\left(N_s \frac{2\pi}{N} k\right)}{\sin\left(\frac{2\pi}{N} k\right)} \right| \tag{4.1}$$

which bears a strong resemblance to our original `sinc` function. Figure 4.4 plots this function.

Note that the DFT grows taller and narrower as the rectangular window grows longer (i.e. sampling duration increases). The amplitude growth occurs because the longer window accumulates more energy than the shorter window. Functionally, this appears in the argument of the sine function in the numerator of Eq. 4.1. The length of the DFT is the same for both lines in Fig. 4.4, so only the length of the

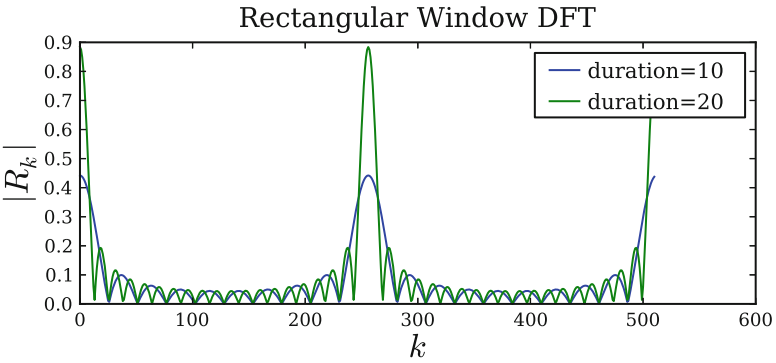


Fig. 4.4 Figure generated by Listing 4.4 showing how longer windows narrow the mainlobe so that it can resolve two nearby tones

rectangular window varies. Looking at the formula in Eq. 4.1, the null-to-null width of the mainlobe in frequency terms is the following

$$\delta f = 2\frac{N}{2N_s}\frac{f_s}{N} = \frac{f_s}{N_s}$$

Thus, two frequencies that differ by at least this amount should be resolvable in these plots. In our last example, we had $N_s = 128$, $f_s = 64 \Rightarrow \delta f = 1/2$ Hz and we were trying to separate two frequencies 0.5 Hz apart so we were right on the edge in that case. I invite you to download this IPython notebook and try longer or shorter signal durations to see how these plots change. This where some define *frequency bin* as the DFT resolution (f_s/N) divided by this minimal resolution (f_s/N_s), which gives N_s/N. Unfortunately, sometimes *bin* is used as slang for the kth discrete frequency index. Fundamentally, the DFT measures frequency in discrete units of this minimal resolution, N_s/N. We will see this terminology again when we take up window functions.

Beware that sampling over a longer duration only helps when the signal frequencies are *stable* over the longer duration. If these frequencies drift during the longer interval or otherwise become contaminated with other signals, then advanced techniques become necessary. Now that we have built up enough intuition, let's consider the mechanics of how the DFT of the rectangular window affects resolution by developing the circular convolution.

4.3 Circular Convolution

Suppose we want to compute the DFT of a product $z_n = x_n y_n$ as shown below,

$$Z_k = \frac{1}{\sqrt{N}}\sum_{n=0}^{N-1}(x_n y_n)W_N^{-nk}$$

where

$$W_N^{nk} = \exp\left(\frac{j2\pi nk}{N}\right)$$

in terms of the respective DFTs of x_n and y_n, X_k and Y_k, respectively, where

$$x_n = \frac{1}{\sqrt{N}} \sum_{p=0}^{N-1} X_p W_N^{-np}$$

and

$$y_n = \frac{1}{\sqrt{N}} \sum_{m=0}^{N-1} Y_m W_N^{-nm}$$

Then, substituting back in gives,

$$Z_k = \frac{1}{\sqrt{N}} \frac{1}{N} \sum_{p=0}^{N-1} X_p \sum_{m=0}^{N-1} Y_m \sum_{n=0}^{N-1} W_N^{-nk-np-nm}$$

The last term evaluates to

$$\sum_{n=0}^{N-1} W_N^{-nk-np-nm} = \frac{1 - W_N^{N(-k-p-m)}}{1 - W_N^{-k-p-m}} = \frac{1 - e^{j2\pi(-k-p-m)}}{1 - e^{j2\pi(-k-p-m)/N}}$$

This is always zero except where $k + p + m = qN$ ($q \in \mathbb{Z}$), in which case it is N. Substituting all this back into our expression gives the *circular convolution* usually denoted as

$$Z_k = \frac{1}{\sqrt{N}} \sum_{p=0}^{N-1} X_p Y_{((k-p))_N} = X_k \otimes_N Y_k$$

where the double subscripted parenthesis emphasizes the periodic nature of the index. The circular convolution lets us to compute the DFT Z_k directly from the corresponding DFTs X_k and Y_k.

Let's work through an example to see this in action. Figure 4.5 shows the rectangular window DFT in blue, R_k against the sinusoid input signal in red, X_k, for each value of k as the two terms slide past each other from left to right, top to bottom. In other words, the kth term in Z_k, the DFT of the product $x_n r_n$, can be thought of as the inner-product of the red and blue lines. This is not *exactly* true because we are just plotting magnitudes and not the real/imaginary parts, but it's enough to understand the mechanics of the circular convolution.

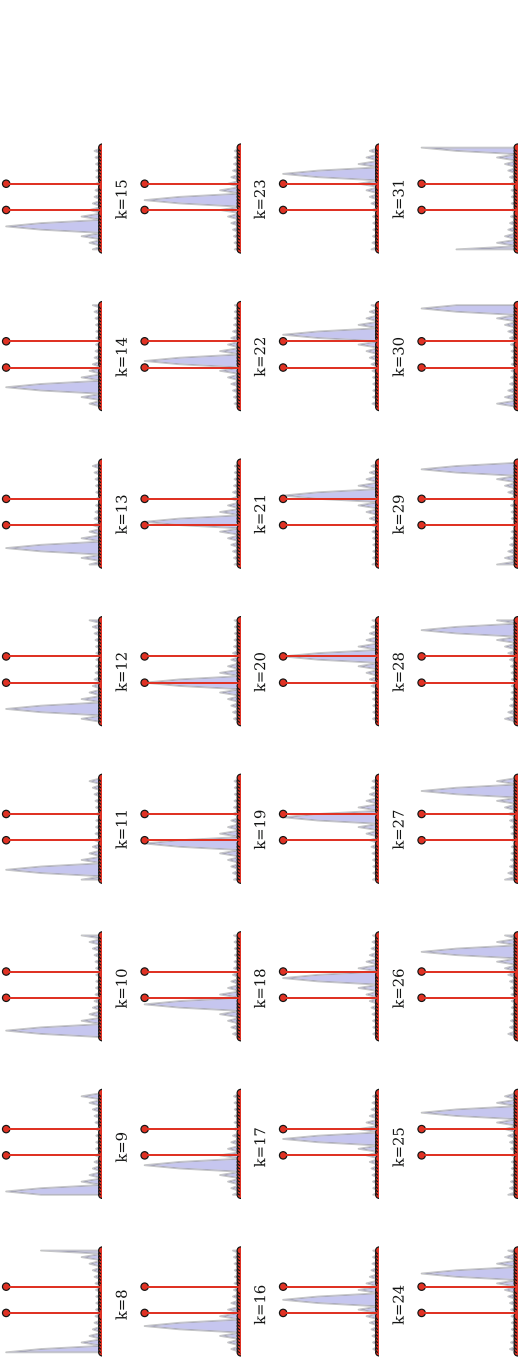


Fig. 4.5 Figure generated by Listing 4.5 showing the step-by-step operation of the circular convolution. The title of each of the subplots shows the DFT index. The DFT of the rectangular window is shown in *blue* and the two tones are shown in *red*. When the DFT of the rectangular window (i.e. sinc function) rolls up to the peak of one of the tones, the resulting inner-product is maximized. For example, for $k = 12$, the rectangular window hits the peak of the leftmost tone and the resulting DFT for $k = 12$ is shown in Fig. 4.6. This illustrates that when the rectangular window's DFT is wide enough to encompass both tones for a particular DFT index, the two tones can no longer be distinguished. This effect is the mechanism for frequency resolution using the DFT

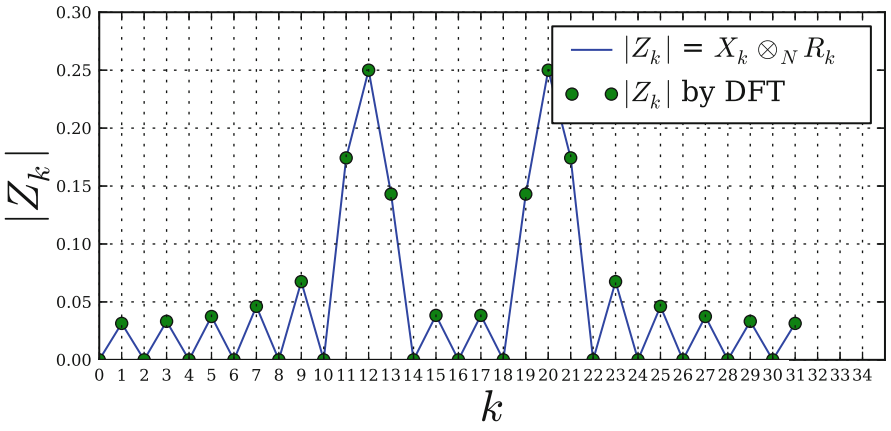


Fig. 4.6 Figure generated by Listing **4.6**. Each point on this plot corresponds to one of the frames shown in Fig. 4.5. For example, at $k = 12$, the DFT achieves a peak corresponding to one of the two tones in the input. This corresponds to the $k = 12$ frame in Fig. 4.5 where the DFT of the rectangular window lines up exactly with one of the two tones

The best way to think about the rectangular window's sinc-shaped DFT as it slides past the input signal is as a *probe* with a resolution defined by its mainlobe width. For example, in frame $k = 12$, we see that the peak of the rectangular window coincides with the peak of the input frequency so we should expect a large value for $Z_{k=12}$ which is shown in Fig. 4.6. However, if the rectangular window were shorter, corresponding to a wider mainlobe, then two nearby frequencies would be draped in the same mainlobe and would then be indistinguishable in the resulting inner-product of the two overlapping graphs. Figure 4.6 shows the direct computation of the DFT of Z_k matches the circular convolution method using X_k and R_k.

In this section, we unpacked the issues involved in resolving two nearby frequencies using the DFT and once again confronted the uncertainty principle in action. We realized that longer DFTs cannot distinguish nearby frequencies unless the signal is sampled over a sufficient duration. We developed the circular convolution as a tool to visualize the exactly how a longer sampling duration helps resolve frequencies.

4.4 Spectral Analysis Using Windows

In the previous section, we discovered that resolving two nearby frequencies depends crucially on the rectangular window and its corresponding DFT. However, our two neighboring tones had *equal* amplitudes. What happens when this is no longer the case? Figure 4.7 shows the DFT of two nearby signals where one has ten times the amplitude of the other.

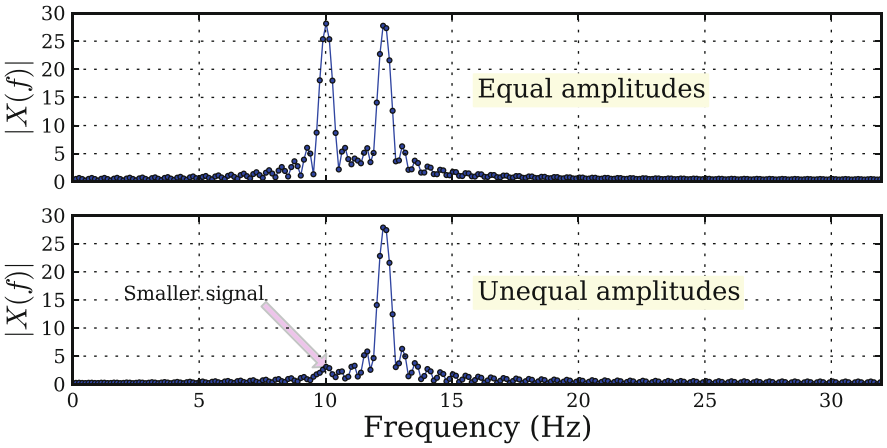


Fig. 4.7 Figure generated by Listing **4.7**. The two equal amplitude tones are easily distinguished in the *top plot*, but not in the *bottom plot*, where one has ten times the amplitude

In Fig. 4.7, the top plot shows the two peaks of neighboring tones when they both have the same amplitude. The bottom plot shows the case where the higher frequency signal has ten times the amplitude of the lower frequency signal. Note that the two frequencies are no longer resolvable because the sidelobes of the stronger signal have swamped the weaker signal. Thus, we need more than just longer sampling durations and longer DFTs to resolve these signals. This is where *windows* come in.

A window is a function that is multiplied sample-by-sample against the input signal (i.e. element-wise). For example, given an input signal, s_n, and a window, w_n, the new windowed signal is,

$$y_n = s_n w_n \tag{4.2}$$

known in vector notation as the *Hadamard* product,

$$\mathbf{y} = \mathbf{w} \odot \mathbf{s} \tag{4.3}$$

Figure 4.8 shows the input signal before (light blue line) and after (red line) applying the triangular window. The amplitudes are noted on the vertical scale on the left. The triangular window is drawn in the background with scale noted on the right side. The red line is the product of the blue line and the window function in the pink background. Because the triangular window starts at zero, peaks in the middle, and descends back to zero at the end, the amplitude of the windowed signal also peaks at the middle, and drops away towards either end. This means that we lose precious signal energy at the start and end of the signal. We certainly want something in exchange for this!

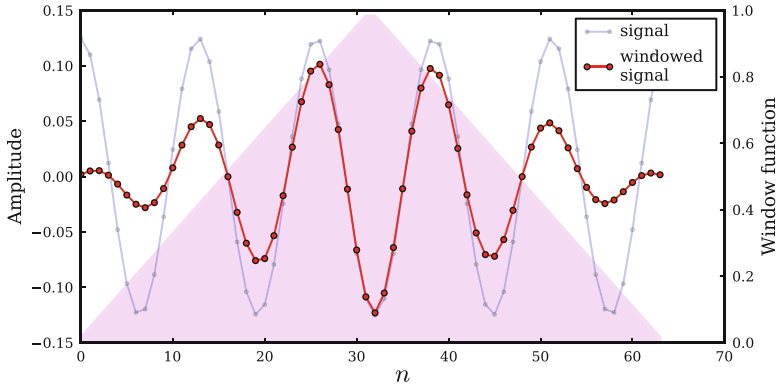


Fig. 4.8 Figure generated by Listing 4.8

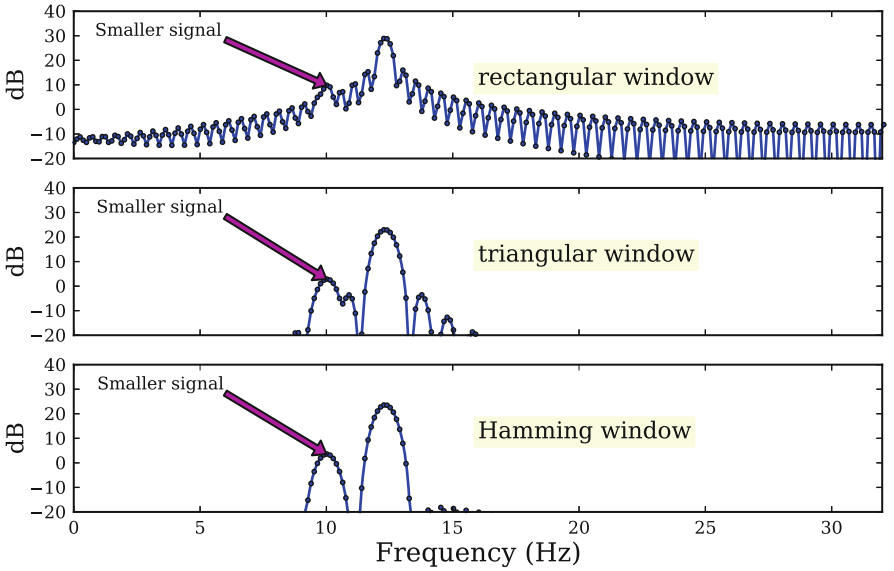


Fig. 4.9 Figure generated by Listing 4.9 showing a weaker signal buried in the sidelobes of a strong signal that is uncovered by window functions

Figure 4.9 shows the two input signals with rectangular, triangular, and Hamming windows applied. To enhance detail, Fig. 4.9 shows the $20\log_{10}|X|$ of the DFT of the input signal with noted windows applied on the same scale. In the top plot, we can barely make out the smaller signal in the sidelobes of the dominant larger signal. The middle plot shows the application using the triangular window with the weaker tone peeking out from the sidelobes. The bottom plot shows two distinct peaks due to the applied Hamming window. Note that the width of the mainlobe widened as the second weaker signal emerged from the sidelobes of the stronger signal. This is one

of the many trade-offs involved when using windows for spectral analysis. We just traded precious signal energy for resolution, but we need a systematic framework for evaluating different window functions. This framework will be the topic of our next section.

4.5 Window Metrics

In the last section, we used windows for spectral analysis and noted that, while windows helped uncover weak signals buried in the sidelobes of nearby stronger signals, there were trade-offs involved. In this section, we put together the standard framework for analyzing and categorizing windows. There are many, many windows used in signal processing, so we begin by considering their common characteristics and then develop the standard metrics for them.

In Chap. 2, we discussed the time-bandwidth product and how to choose signal durations to satisfy the underlying reconstruction, but satisfactory reconstruction is different from separating frequencies in a finite window using the DFT, even though many of the same issues apply. The DFT works best when the signal is periodic with period equal to the length of the DFT (N), and when an N-length section of signal, repeated end-to-end, has no discontinuities at the joining end-points. When this happens, it is possible to `exactly` isolate the signal's frequency, because it, like the columns of the DFT matrix, shares the same fundamental frequency, $2\pi/N$. More commonly, when this doesn't happen, the signal is smeared across all of the frequencies represented in the DFT matrix. This is *spectral leakage*.

Figure 4.10 shows the effect of spectral leakage. The top row shows the ideal situation where there is a perfect match between the period of the signal and the DFT-length. Furthermore, one of the DFT frequencies is exactly equal to the signal frequency. This means that the DFT perfectly isolates the signal frequency at the one component shown on the top right.

The second row of Fig. 4.10 shows when the signal's period is *slightly different* from the DFT-length and there *is* a discontinuity at the joining ends. When this happens, the DFT cannot perfectly isolate the signal's frequency because the signal frequency is not an integer-multiple of $2\pi/N$, which means it is *not* in the set of frequencies represented in the DFT (i.e. columns of the DFT matrix) and thus *leaks* energy into other frequencies. In other words, the signal frequency is somewhere between integer-multiples of $2\pi/N$ and thus cannot be perfectly isolated in any of the N discrete frequencies represented by the DFT. The third row of Fig. 4.10 uses a window to that eliminates the discontinuity at edges, but obviously distorts the amplitude of the original signal. The remainder of this section explores the many ways to analyze windows and their effects and provides some guidelines for choosing windows in practice.

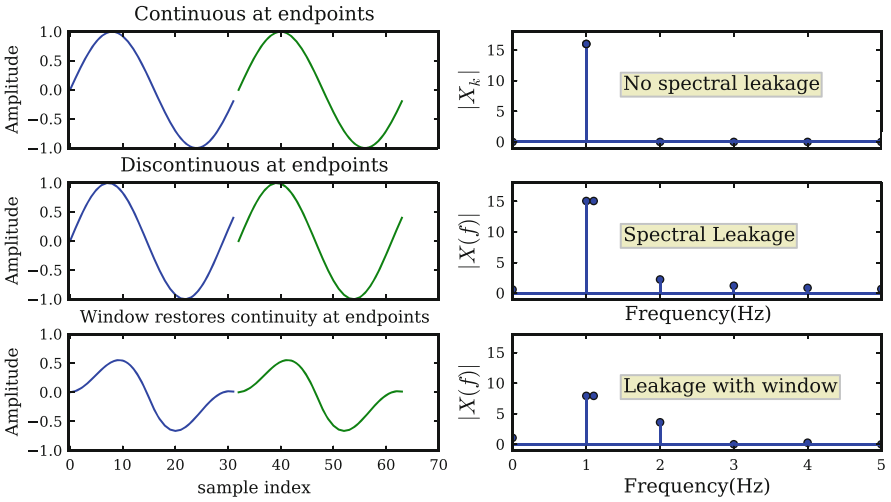


Fig. 4.10 Figure generated by Listing 4.12

4.5.1 Processing Gain

Figure 4.11 shows the loss in signal due to the window in the sample domain (top plot) and in the DFT-domain (bottom plot). The shaded region in the top plot shows the signal that has been attenuated by the window, which slopes to zero towards the edges. The bottom plot shows the DFTs of the signal before and after windowing. As indicated, the loss in power is the drop in the DFT at the signal frequency, f_o. Note that the mainlobe of the windowed DFT is *much* larger than before which makes it harder to separate two nearby frequencies that are separated by less than the width of the mainlobe. Losing signal energy is not good, but we must consider this in the context of ever-present noise. This brings us to our first formal window metric.

To analyze the effect of windows, we define the signal-to-noise ratio (SNR) and then compute it before and after applying the window function. For simplicity, we consider a perfect narrowband signal at frequency f_o with amplitude, A,

$$\mathbf{x} = A\mathbf{u}_o \tag{4.4}$$

where

$$\mathbf{u}_o = \frac{1}{\sqrt{N_s}}\left[\exp\left(j\frac{2\pi f_o}{f_s}n\right)\right]_{n=0}^{N_s-1} \tag{4.5}$$

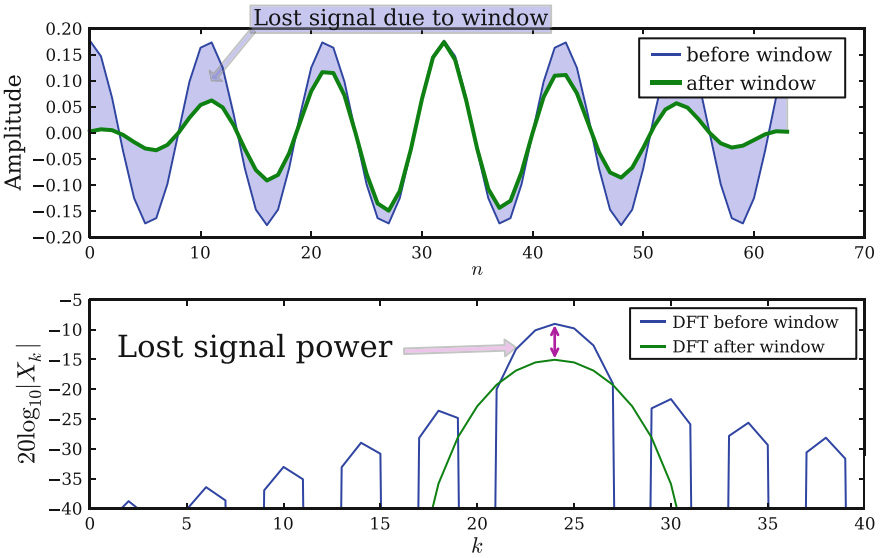


Fig. 4.11 Figure generated by Listing 4.10. The *shaded region* in the *top plot* shows the signal attenuated by the window resulting in a loss of signal energy. The same loss is shown in the *bottom plot* using the DFT

with signal energy equal to A^2. We'll assume the noise power is σ_ν^2. Thus, the pre-window signal-to-noise ratio is,

$$SNR_{pre} = \frac{A^2}{\sigma_\nu^2} \tag{4.6}$$

After applying the window, the updated signal power at $f = f_o$ becomes

$$|\mathbf{u}_o^H\,(\mathbf{w} \odot \mathbf{x})\,|^2 = |\mathbf{u}_o^H\,\mathrm{diag}(\mathbf{w})\mathbf{x}|^2 = A^2|\mathbf{1}^T\mathbf{w}|^2 \tag{4.7}$$

with corresponding noise power

$$\mathbb{E}|\mathbf{w} \odot \mathbf{n}|^2 = \mathrm{Trace}\left(\mathrm{diag}(\mathbf{w})\mathbb{E}\left(\mathbf{n}\mathbf{n}^T\right)\mathrm{diag}(\mathbf{w})\right) \tag{4.8}$$

$$= \mathrm{Trace}\left(\mathrm{diag}(\mathbf{w})\sigma_\nu^2\mathbf{I}\,\mathrm{diag}(\mathbf{w})\right) \tag{4.9}$$

$$= \sigma_\nu^2\mathbf{w}^T\mathbf{w} \tag{4.10}$$

where $\mathbb{E}\left(\mathbf{n}\mathbf{n}^T\right) = \sigma_\nu^2\mathbf{I}$ and $\mathbf{n}$ is a vector of mutually uncorrelated noise samples with variance σ_ν^2. Thus, the post-window signal-to-noise ratio is

$$SNR_{post} = \frac{A^2|\mathbf{1}^T\mathbf{w}|^2}{\sigma_v^2\mathbf{w}^T\mathbf{w}} \tag{4.11}$$

Finally, the ratio of the post-window to pre-window signal-to-noise ratios is defined as the *processing gain*,

$$G_p \triangleq \frac{SNR_{post}}{SNR_{pre}} = \frac{|\mathbf{1}^T\mathbf{w}|^2}{\mathbf{w}^T\mathbf{w}} \tag{4.12}$$

Note that the *coherent gain* comes from the numerator as defined as the following:

$$G_{coh} \triangleq \mathbf{1}^T\mathbf{w} \tag{4.13}$$

Thus, the window reduces signal power *and* noise power and the net effect is to increase the SNR. Processing gain summarizes this effect.

4.5.2 Equivalent Noise Bandwidth

Another way to understand window functions in terms of noise is to evaluate windows in terms of the amount of noise they *pass* through the mainlobe. Figure 4.12 illustrates this concept. Figure 4.12 shows the squared DFT of the window with the overlaid rectangle showing the idealized bandpass filter that would pass the same amount of noise power at the peak of the window's DFT. It's as if all the noise was packed into a rectangle centered at the peak of the mainlobe. This is the *Equivalent Noise Bandwidth* (ENBW) concept. The post-window noise power is the following:

$$\mathbb{E}||\mathbf{w} \odot \mathbf{n}||^2 = \sigma_v^2\mathbf{w}^T\mathbf{w} \tag{4.14}$$

We want to equate this power (which is spread over all frequencies) to the corresponding output noise power of a perfect bandlimited filter with height equal to W_0 and width B_{neq}.

$$B_{neq} \cdot W_0^2\sigma_v^2 = \sigma_v^2\mathbf{w}^T\mathbf{w} \tag{4.15}$$

and solving for B_{neq} gives,

$$B_{neq} = \mathbf{w}^T\mathbf{w}/W_0^2 \tag{4.16}$$

and since $W_0 = \mathbf{1}^T\mathbf{w}$, we can write this as the following,

$$B_{neq} \triangleq \frac{\mathbf{w}^T\mathbf{w}}{|\mathbf{1}^T\mathbf{w}|^2} = \frac{1}{G_p} \tag{4.17}$$

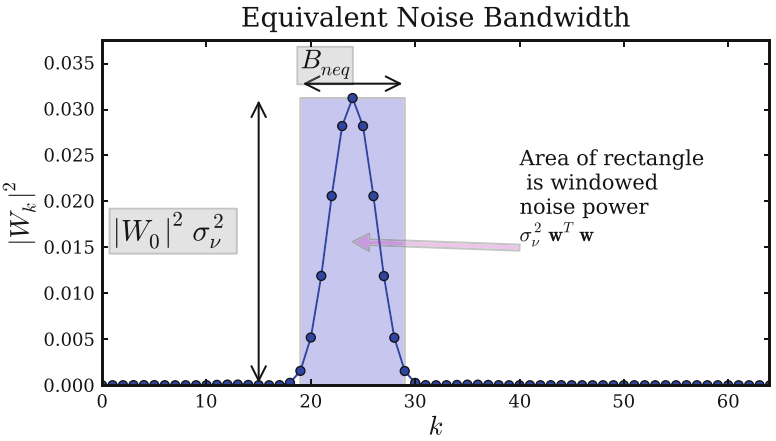


Fig. 4.12 Figure generated by Listing **4.11** showing the equivalent noise bandwidth concept

which shows its close relationship to processing gain. The intuition is that the larger the equivalent noise bandwidth, the more noise is passed through the mainlobe, thus competing more with the signal at the nominal center of that lobe, thereby reducing the processing gain of the window.

Now, we cover three additional metrics used in practice for evaluating and choosing between different window functions.

4.5.3 Peak Sidelobe Level

As the name suggests, Peak Sidelobe Level measures the gap between the mainlobe maximum and the nearest sidelobe peak. This captures the worst-case scenario where a nearby signal sits exactly on the highest sidelobe. This idea is illustrated in Fig. 4.13.

Figure 4.13 shows the DFT of the Hanning window and how far down the next sidelobe is from the peak (≈31 dB). The figure of merit considers the worst-case where an interfering single frequency sits exactly on the peak of this sidelobe.[1]

Now, let's consider the scenario with a signal on the highest sidelobe of the window function. The cartoon in Fig. 4.14 shows the DFT of the window function as it slides across the two signals in the circular convolution. In the top plot, the green signal sits on the peak of the first sidelobe of the window. Because the peak sidelobe level is 31 dB down, its contribution to the overall DFT at that discrete frequency is reduced by 31 dB. Note that at this stage, the signal on the left offers its maximum

[1]The code in Listing **4.14** uses complex roots to find the peaks of the sidelobes for window functions.

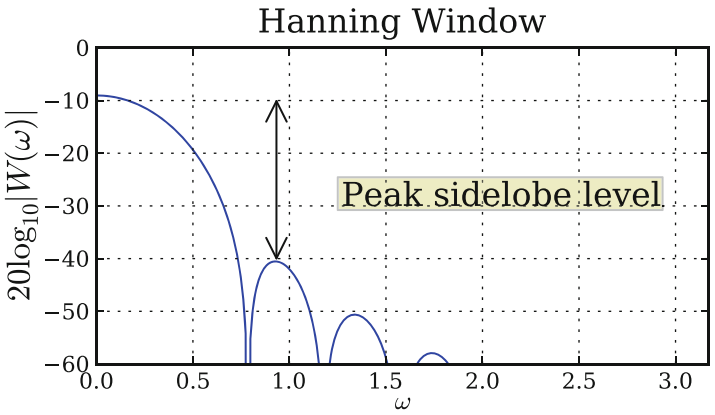


Fig. 4.13 Figure generated by Listing 4.13 showing the definition of "peak sidelobe level". The peak sidelobe level is the gap between the window's peak and the peak of the highest sidelobe

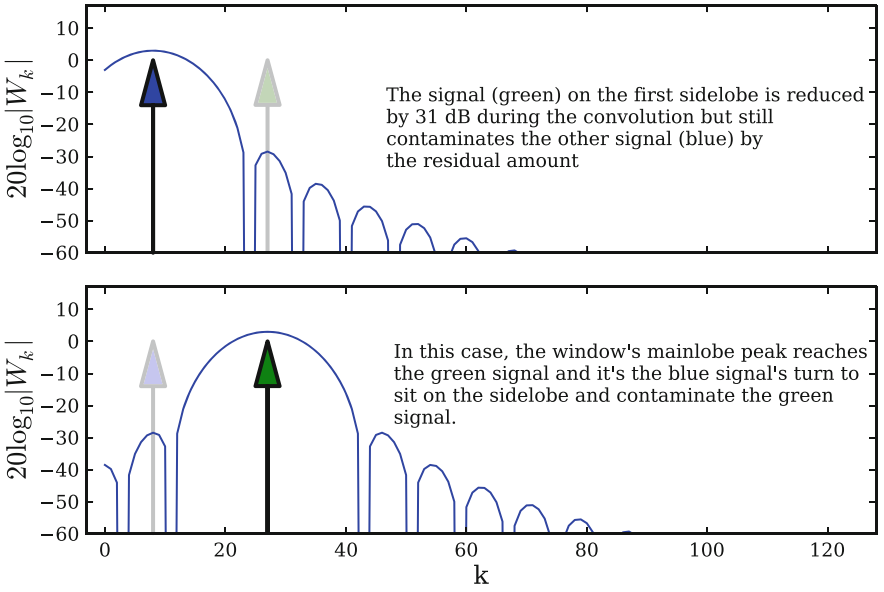


Fig. 4.14 Figure generated by Listing 4.15 illustrating the effect of peak sidelobe levels. As we discussed previously in the circular convolution, as the window slides past one of the two tones, the neighboring tone coincidentally hits the peak sidelobe and is only attenuated by the peak sidelobe level. Thus, with a high peak sidelobe level, interference from the neighboring tone can bleed into the signal at the *center* of the mainlobe and distort it. This happens in the *top plot* where the left-most tone is contaminated by the other tone. The *bottom plot* shows this happening again in the circular convolution, but now the right-most tone becomes contaminated. The concept of peaks sidelobe level quantifies this potential unhappy circumstance. Keep in mind that this figure is a cartoon because the circular convolution does uses complex linear terms, not the logarithmic magnitudes shown

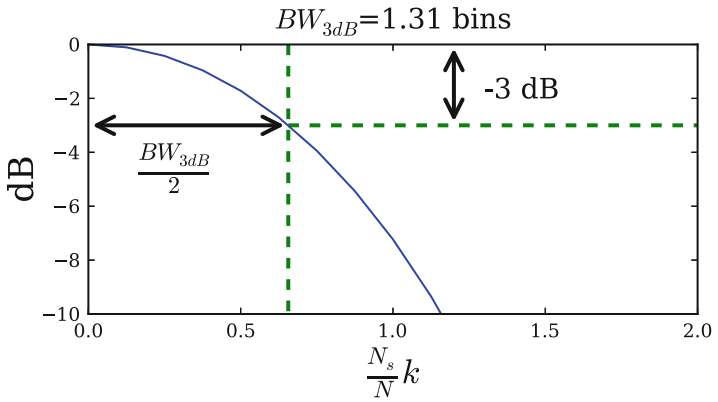


Fig. 4.15 Figure generated by Listing 4.16 illustrating the concept of 3-dB bandwidth. the distance (in frequency units) between the mainlobe's peak and the point on the mainlobe that is 3 dB down from the peak is half of the 3-dB bandwidth. This metric attempts to quantify how close two nearby tones can be before they become indistinguishable

contribution because it is squarely on the peak of the window's mainlobe. As shown in the bottom plot, the same situation happens again in the convolution as the sliding window reaches the green signal on the right, where it will be contaminated by signal on the left as that signal climbs onto the peak of the sidelobe on the left. Bear in mind this figure is a cartoon because the circular convolution uses complex linear terms, not the logarithmic magnitudes shown.

4.5.4 3-dB Bandwidth

At the 3-dB bandwidth point, the mainlobe has lost half of its peak power. This figure of merit provides a sense of how far off a signal can be from the mainlobe before losing half its energy. This is because $10\log_{10}(1/2) \approx -3$. Figure 4.15 shows the DFT of the hamming window and its corresponding half-power point on the mainlobe. In general, there is no closed form solution to the half-power level so we must compute it numerically. Figure 4.15 is the schematic for the mainlobe of the window.

4.5.5 Scalloping Loss

The DFT divides the sampling frequency f_s into N discrete frequencies, but the signal of interest may not conveniently lie on one of these sampled frequencies. The *scalloping loss* accounts for the reduction in signal energy. Scalloping loss is defined

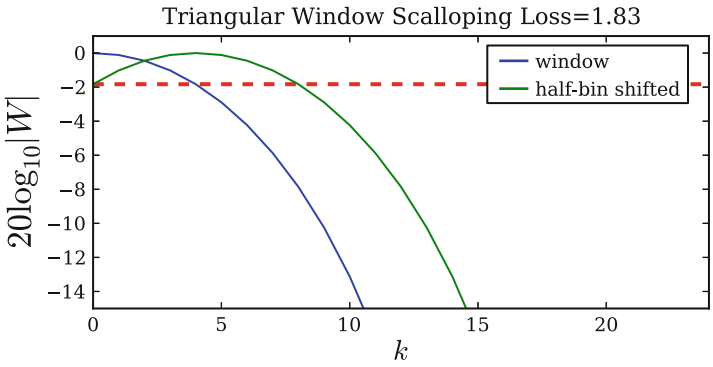


Fig. 4.16 Figure generated by Listing **4.17** illustrating the concept of "scalloping loss". This metric quantifies the loss that occurs when the DFT index frequency does not match the signal frequency

as the ratio of coherent gain for a tone located half a bin from a DFT sample point to the coherent gain,

$$ScallopingLoss = \frac{|\sum_{n=0}^{N_s-1} w_n \exp(-j\pi n/N_s)|}{\sum_{n=0}^{N_s-1} w_n}$$

The horizontal line in Fig. 4.16 shows the level of the scalloping loss for the triangular window.

4.6 Summary

In this section, we explained the shape of windows functions in terms of spectral leakage and developed the concept of processing gain and equivalent noise bandwidth as closely related metrics for categorizing windows. The exhaustive 1978 paper by Harris [Haran] is the primary reference work for many more windows. Note that the window functions are implemented in the `scipy.signal.windows` submodule but sometimes the normalization factors and defined parameters are slightly different from Harris' paper. Sometimes the term *tapers* is used instead of *window functions* in certain applications. Window functions are also fundamental to antenna analysis for many of the same reasons, especially with respect to linear arrays.

We illustrated the major issues involved in using window functions for spectral analysis and derived the most common figures of merit for some popular windows. There are many more windows available and Harris [Haran] provides an exhaustive list and many more detailed figures of merit. In fact, I have seen rumpled versions of this 1978 paper in just about every engineering lab I have worked in. In practice, some window functions are preferred because they have coefficients that are easy

to implement on fixed point arithmetic hardware without excessive sensitivity to quantization errors. The main idea is that windows with favorable sidelobe levels are always wider (larger equivalent noise bandwidth and 3-dB bandwidth) so these pull more signal noise into their mainlobes and make it harder to distinguish nearby signals which may fit into the same mainlobe. Thus, there is a tension between signal-to-noise (more noise in mainlobe reduces signal-to-noise) and resolution (wider mainlobes mask nearby signals). Unfortunately, it is not possible to simultaneously have very low sidelobes and a very narrow mainlobe. This is due to the *waterbed effect* where "pushing" down the window's DFT at any one place causes it to pop up somewhere else.

Outside of two-tone separability, there is the issue of wideband signals. In that case, you may prefer a wide mainlobe that encompasses the signal bandwidth with very low sidelobes that reduce extraneous signals. The bottom line is that there are many engineering trade-offs involved in choosing window functions for a particular application. Understanding how window functions are used in spectral analysis is fundamental to the entire field of signal processing because it touches all of the key issues encountered in practice.

Appendix

```
from numpy import fft

Nf = 64                 # N- DFT size
fs = 64                 # sampling frequency
f = 10                  # one signal
t = arange(0,1,1/fs)  # time-domain samples
deltaf = 1/2.           # second nearby frequency

# keep x and y-axes on same respective scale
fig,ax = subplots(2,1,sharex=True,sharey=True)
fig.set_size_inches((8,3))

x=cos(2*pi*f*t) + cos(2*pi*(f+2)*t) # 2 Hz frequency difference
X = fft.fft(x,Nf)/sqrt(Nf)
ax[0].plot(linspace(0,fs,Nf),abs(X),'-o')
ax[0].set_title(r'$\delta f = 2$ Hz, $T=1$ s',fontsize=18)
ax[0].set_ylabel(r'$|X(f)|$',fontsize=18)
ax[0].grid()
x=cos(2*pi*f*t) + cos(2*pi*(f+deltaf)*t) # delta_f frequency difference
X = fft.fft(x,Nf)/sqrt(Nf)
ax[1].plot(linspace(0,fs,Nf),abs(X),'-o')
ax[1].set_title(r'$\delta f = 1/2$ Hz, $T=1$ s',fontsize=14)
ax[1].set_ylabel(r'$|X(f)|$',fontsize=18)
```

```
ax[1].set_xlabel('Frequency (Hz)',fontsize=18)
ax[1].set_xlim(xmax = fs/2)
ax[1].set_ylim(ymax=6)
ax[1].grid()
```

Listing 4.1: Listing corresponding to Fig. 4.1. The `fft` module is a front-end to the FFTPACK compiled library. The `abs` computes the real-valued magnitude of its complex argument.

```
Nf = 64*2 # FFT size
fig,ax = subplots(2,1,sharex=True,sharey=True)
fig.set_size_inches((8,4))

X = fft.fft(x,Nf)/sqrt(Nf)
ax[0].plot(linspace(0,fs,len(X)),abs(X),'-o',ms=3.) # marker size=3
ax[0].set_title(r'$N=%d$, $T=1$s'%Nf,fontsize=18)
ax[0].set_ylabel(r'$|X(k)|$',fontsize=18)
ax[0].grid()

Nf = 64*4 # FFT size
X = fft.fft(x,Nf)/sqrt(Nf)
ax[1].plot(linspace(0,fs,len(X)),abs(X),'-o',ms=3.)
ax[1].set_title(r'$N=%d$, $T=1$s'%Nf,fontsize=18)
ax[1].set_ylabel(r'$|X(k)|$',fontsize=18)
ax[1].set_xlabel('Frequency (Hz)',fontsize=18)
ax[1].set_xlim(xmax=fs/2)
ax[1].set_ylim(ymax=6)
ax[1].grid()
```

Listing 4.2: Listing corresponding to Fig. 4.2. Note the explicit setting for the marker size `ms=3`.

```
t = arange(0,2,1/fs)
x=cos(2*pi*f*t) + cos(2*pi*(f+deltaf)*t)

Nf = 64*2 # FFT size
fig,ax = subplots(2,1,sharex=True,sharey=True)
fig.set_size_inches((8,4))

X = fft.fft(x,Nf)/sqrt(Nf)
ax[0].plot(linspace(0,fs,len(X)),abs(X),'-o',ms=3.)
ax[0].set_title(r'$N=%d$, $T=2$s'%Nf,fontsize=18)
ax[0].set_ylabel(r'$|X(f)|$',fontsize=18)
ax[0].grid()

Nf = 64*8 # FFT size
X = fft.fft(x,Nf)/sqrt(Nf)
ax[1].plot(linspace(0,fs,len(X)),abs(X),'-o',ms=3.)
ax[1].set_title(r'$N=%d$, $T=2$s'%Nf,fontsize=18)
ax[1].set_ylabel(r'$|X(f)|$',fontsize=18)
ax[1].set_xlabel('Frequency (Hz)',fontsize=18)
ax[1].set_xlim(xmax = fs/2)
ax[1].set_ylim(ymax=6)
ax[1].grid()
```

Listing 4.3: Listing corresponding to Fig. 4.3.

```
def abs_sinc(k=None,N=64,Ns=32):
    'absolute value of sinc'
    if k is None: k = arange(0,N-1)
    y = where(k == 0, 1.0e-20, k)
    return abs(sin(Ns*2*pi/N*y)/sin(2*pi*y/N))/sqrt(N)

fig,ax=subplots()
fig.set_size_inches((8,3))

ax.plot(abs_sinc(N=512,Ns=10),label='duration=10')
ax.plot(abs_sinc(N=512,Ns=20),label='duration=20')
ax.set_xlabel('$k$',fontsize=22)
ax.set_ylabel(r'$|R_k|$',fontsize=22)
ax.set_title('Rectangular Window DFT',fontsize=18)
ax.legend(loc=0,fontsize=14);
```

Listing 4.4: Listing corresponding to Fig. 4.4. The `label` keyword in the plot function assigns the text to the line for later use in the `legend` function. You can assign text without this labeling later in `legend`, but doing it this ways makes it easier to keep track of which line goes with which text.

```
def dftmatrix(Nfft=32,N=None):
    'construct DFT matrix'
    k= np.arange(Nfft)
    if N is None: N = Nfft
    n = arange(N)
    # use numpy broadcasting to create matrix
    U = matrix(exp(1j* 2*pi/Nfft *k*n[:,None]))
    return U/sqrt(Nfft)

Nf = 32 # DFT size
U = dftmatrix(Nf,Nf)
x = U[:,12].real      # input signal
X = U.H*x             # DFT of input
rect = ones((Nf/2,1)) # short rectangular window
z = x[:Nf/2] # product of rectangular window and x (i.e. chopped version of x)
R = dftmatrix(Nf,Nf/2).H*rect # DFT of rectangular window
Z = dftmatrix(Nf,Nf/2).H*z    # DFT of product of x_n and r_n

# use numpy broadcasting to setup summand's indices
idx=arange(Nf)-arange(Nf)[:,None]
idx[idx<0]+=Nf # add periodic Nf to negative indices for wraparound
a = arange(Nf) # k^th frequency index

fig,ax = subplots(4,8,sharex=True,sharey=True)
fig.set_size_inches((12,5))
for i,j in enumerate(ax.flat):
    j.fill_between(arange(Nf),1/sqrt(Nf)*abs(R[idx[:,i],0]).flat,0,alpha=0.3)
    # separate stem parts
    markerline, stemlines, baseline =j.stem(arange(Nf),abs(X))
    setp(markerline, 'markersize', 4.)
    setp(markerline,'markerfacecolor','r')
    setp(stemlines,'color','r')
    j.axis('off')
    j.set_title('k=%d'%i,fontsize=8)
```

Listing 4.5: Listing corresponding to Fig. 4.5. The `flat` suffix converts the multidimensional Numpy arrays to one-dimensional Numpy arrays. The `setp` function is a Matplotlib convenience function to set the individual properties of the given objects.

```
fig,ax=subplots()
fig.set_size_inches((7,3))
ax.plot(a,abs(R[idx,0]*X)/sqrt(Nf),label=r'$|Z_k|$ = $X_k\otimes_N R_k$')
ax.plot(a,abs(Z),'o',label=r'$|Z_k|$ by DFT')
ax.set_xlabel('$k$',fontsize=22)
ax.set_ylabel(r'$|Z_k|$',fontsize=22)
ax.set_xticks(arange(ax.get_xticks().max()))
ax.tick_params(labelsize=8)
ax.legend(loc=0,fontsize=14)
ax.grid()
```

Listing 4.6: Listing corresponding to Fig. 4.6. The `tick_params` function changes the size of the labels of all of the tick marks. The `set_xticks` function changes the set of tick marks that are drawn.

```
from scipy import signal
from numpy import fft

# some useful functions
def dftmatrix(Nfft=32,N=None):
    'construct DFT matrix'
    k= np.arange(Nfft)
    if N is None: N = Nfft
    n = arange(N)
    U = matrix(exp(1j* 2*pi/Nfft *k*n[:,None])) # use numpy broadcasting
    to create matrix return U/sqrt(Nfft)

def db20(W,Nfft=None):
    'Given DFT, return power level in dB'
    if Nfft is None: # assume W is DFT
        return 20*log10(abs(W))
    else: # assume time-domain passed, so need DFT
        return 20*log10(abs( fft.fft(W,Nfft)/sqrt(Nfft) ) )

fs = 64 # sampling frequency
t = arange(0,2,1/fs)
f = 10  # one signal
deltaf = 2.3 # second nearby frequency

Nf = 512
fig,ax = subplots(2,1,sharex=True,sharey=True)
fig.set_size_inches((9,4))
```

```
x=10*cos(2*pi*f*t) + 10*cos(2*pi*(f+deltaf)*t) # equal amplitudes
X = fft.fft(x,Nf)/sqrt(Nf)
ax[0].plot(linspace(0,fs,len(X)),abs(X),'-o',ms=3.)
ax[0].set_ylabel(r'$|X(f)|$',fontsize=18)
ax[0].set_xlim(xmax = fs/2)
ax[0].grid()
ax[0].text(0.5,0.5,'Equal amplitudes',
            transform=ax[0].transAxes,
            backgroundcolor='Lightyellow',
            fontsize=16)
x=cos(2*pi*f*t) + 10*cos(2*pi*(f+deltaf)*t) # one has 10x the amplitude
X = fft.fft(x,Nf)/sqrt(Nf)
ax[1].plot(linspace(0,fs,len(X)),abs(X),'-o',ms=3.)
ax[1].set_ylabel(r'$|X(f)|$',fontsize=18)
ax[1].set_xlabel('Frequency (Hz)',fontsize=18)
ax[1].set_xlim(xmax = fs/2)
ax[1].grid()


ax[1].text(0.5,0.5,'Unequal amplitudes',
                transform=ax[1].transAxes,
                backgroundcolor='lightyellow',
                fontsize=16)
ax[1].annotate('Smaller signal',
            fontsize=12,xy=(f,abs(X)[int(f/fs*Nf)]),
            xytext=(2,15),
            arrowprops={'facecolor':'m','alpha':.3});
```

Listing 4.7: Listing corresponding to Fig. 4.7. The `text` function puts text in the middle of the plot. Setting the `transform` to `transAxes` means that the `(0.5,0.5)` coordinates of the text are relative to the axis and not to the data in the plot. Thus, `(0.5,0.5)` refers to the center of the plot. Using this coordinate system makes it easy to place things in the plot regardless of the data that is plotted.

```
fig,ax = subplots()
fig.set_size_inches((10,5))

Nf = 128
nsamp = 64
window = signal.triang(nsamp)
rectwin = ones(nsamp)

x=array(dftmatrix(64,64)[:,5].real).flatten() # convert to numpy array
n = arange(len(x))
window = signal.triang(len(x))

ax.plot(n,x,'-o',label='signal',ms=3.,alpha=0.3)
ax.plot(n,window*x,'-or',label='windowed\nsignal',ms=5.)
ax.set_ylabel('Amplitude',fontsize=16)
ax.set_xlabel('$n$',fontsize=22)
ax.legend(loc=0,fontsize=14)
ax2 = ax.twinx()
ax2.fill_between(n,window,alpha=0.2,color='m')
ax2.set_ylabel('Window function',fontsize=16);
```

Listing 4.8: Listing corresponding to Fig. 4.8.

```
Nf = 512
fig,ax = subplots(3,1,sharex=True,sharey=True)
fig.set_size_inches((10,6))

x=cos(2*pi*f*t) + 10*cos(2*pi*(f+deltaf)*t)
X = fft.fft(x,Nf)/sqrt(Nf)
ax[0].plot(linspace(0,fs,len(X)),db20(X),'-o',ms=3.)
ax[0].set_xlim(xmax = fs/2)
ax[0].set_ylabel('dB',fontsize=16)
ax[0].text(0.5,0.5,'rectangular window',
                transform=ax[0].transAxes,
                backgroundcolor='lightyellow',
                fontsize=16)
ax[0].annotate('Smaller signal',
            fontsize=12,xy=(f,db20(X)[int(f/fs*Nf)]),
            xytext=(1,30),
            arrowprops={'facecolor':'m'})

w = signal.triang(len(x))
X = fft.fft(x*w,Nf)/sqrt(Nf)
ax[1].plot(linspace(0,fs,len(X)),db20(X),'-o',ms=3.)
ax[1].set_xlim(xmax = fs/2)
ax[1].set_ylabel('dB',fontsize=16)
ax[1].text(0.5,0.5,'triangular window',
                transform=ax[1].transAxes,
                backgroundcolor='lightyellow',
                fontsize=16)
                ax[1].annotate('Smaller signal',
                         fontsize=12,xy=(f,db20(X)[int(f/fs*Nf)]),
            xytext=(1,30),
            arrowprops={'facecolor':'m'})
                w = signal.hamming(len(x))
X = fft.fft(x*w,Nf)/sqrt(Nf)
ax[2].plot(linspace(0,fs,len(X)),db20(X),'-o',ms=3.)
ax[2].set_xlabel('Frequency (Hz)',fontsize=18)
ax[2].set_xlim(xmax = fs/2)
ax[2].set_ylabel('dB',fontsize=16)
ax[2].set_ylim(ymin=-20)
ax[2].text(0.5,0.5,'Hamming window',
                transform=ax[2].transAxes,
                backgroundcolor='lightyellow',
                fontsize=16)
ax[2].annotate('Smaller signal',
            fontsize=12,xy=(f,db20(X)[int(f/fs*Nf)]),
            xytext=(1,30),
            arrowprops={'facecolor':'m'});
```

Listing 4.9: Listing.

```
fo = 2*pi/64*6 # in radians/sec
nz=randn(64,1) # noise samples
w=signal.triang(64) # window function

fig,ax= subplots(2,1)
fig.set_size_inches((10,6))
subplots_adjust(hspace=.3)
n = arange(len(u))
ax[0].plot(n,u.real,label='before window',lw=2)
ax[0].set_ylabel('Amplitude',fontsize=16)
ax[0].plot(n,diag(w)*u.real,label='after window',lw=3.)
ax[0].fill_between(n,array(u).flat, array(diag(w)*u).flat,alpha=0.3)
ax[0].legend(loc=0,fontsize=14)
ax[0].set_xlabel('$n$',fontsize=14)
ax[0].annotate('Lost signal due to window',
                fontsize=16,
                bbox={'fc':'b','alpha':.3}, # alpha is transparency
                xy=(11,0.1),
                xytext=(30,40),
                textcoords='offset points', # coordinate system for xytext
                arrowprops={'facecolor':'b','alpha':.3})
N=256 # DFT size for plot
idx = int(fo/(2*pi/N))
ax[1].plot(db20(u,N),label='DFT before window')
ax[1].plot(db20(diag(w)*u,N),label='DFT after window')
ax[1].set_ylim(ymin=-40,ymax=-5)
ax[1].set_xlim(xmax=40)
ax[1].set_ylabel(r'$20\log_{10}|X_k|$',fontsize=18)
```

```
ax[1].set_xlabel(r'$k$',fontsize=16)
ax[1].annotate('Lost signal power',
                fontsize=22,
                xy=(22,-13),
                xytext=(2,-15),
                arrowprops={'facecolor':'m','alpha':.3})
pkU = db20(u,N)[idx]
pkW = db20(diag(w)*u,N)[idx]
ax[1].annotate('',xy=(idx,pkW),
               xytext=(idx,pkU),
               fontsize=12,
               arrowprops={'arrowstyle':'<->','color':'m'})
ax[1].legend(loc=0,fontsize=12)
```

Listing 4.10: Listing corresponding to Fig. 4.11. The `subplots_adjust` function tweaks the subplot layout by changing the horizontal/vertical whitespace and relative positions of the subplot elements.

```
from matplotlib.patches import Rectangle

fig,ax = subplots()
fig.set_size_inches((8,4))

N = 256 # DFT size
idx = int(fo/(2*pi/N))
Xm = abs(fft.fft(array(diag(w)*u).flatten(),N)/sqrt(N))**2
ax.plot(Xm,'-o')
ax.add_patch(Rectangle((idx-10/2,0),width=10,height=Xm[idx],alpha=0.3))
ax.set_xlim(xmax = N/4)
ax.set_ylabel(r'$|W_k|^2$',fontsize=18)
ax.set_xlabel(r'$k$',fontsize=18)
ax.set_title('Equivalent Noise Bandwidth',fontsize=18)
ax.annotate('Area of rectangle\n is windowed\nnoise power\n'\
              +r'$\sigma_\nu^2 \mathbf{w}^T \mathbf{w}$',
            fontsize=14,
            xy=(idx,Xm.max()/2.),
            xytext=(40,Xm.max()/2.),
            arrowprops={'facecolor':'m','alpha':.3});
ax.annotate('',ha='center',fontsize=24,
             xy=(idx+10/2,Xm.max()*1.05),
             xytext=(idx-10/2,Xm.max()*1.05),
             arrowprops=dict(arrowstyle='<->'))
             ax.annotate('',ha='center',fontsize=24,
             xy=(15,0),
             xytext=(15,Xm.max()),
             arrowprops=dict(arrowstyle='<->'))
             ax.text( 1, Xm.max()/2,r'$ |W_0|^2\sigma_\nu^2 $',fontsize=20,
bbox={'fc':'gray','alpha':.3}) ax.text( idx-5, Xm.max()*1.1,r'$B_{neq}$',
fontsize=18,bbox={'fc':'gray','alpha':.3}) ax.set_ylim(ymax = Xm.max()*1.2)
```

Listing 4.11: Listing corresponding to Fig. 4.12.

```
from scipy import signal

fo = 1 # signal frequency
fs = 32 # sample frequency
Ns = 32 # number of samples
x = sin(2*pi*fo/fs*arange(Ns)) # sampled signal
fig,axs= subplots(3,2,sharex='col',sharey='col')
fig.set_size_inches((12,6))
subplots_adjust(hspace=.3)

ax=axs[0,0]
ax.plot(arange(Ns),x,label='signal')
ax.plot(arange(Ns)+Ns,x,label='extension')
ax.set_ylabel('Amplitude',fontsize=14)
ax.set_title('Continuous at endpoints',fontsize=16)

ax=axs[0,1]
N=Ns  #chosen so DFT bin is exactly on fo
Xm = abs(fft.fft(x,N))
idx = int(fo/(fs/N))
ax.stem(arange(N)/N*fs,Xm,basefmt='b-')
ax.plot(fo, Xm[idx],'o')
ax.set_ylabel(r'$|X_k|$',fontsize=18)
ax.set_xlim(xmax=5)
ax.set_ylim(ymin=-1)
ax.text(0.3,0.5,'No spectral leakage',fontsize=16,
        transform=ax.transAxes,
        bbox={'fc':'y','alpha':.3})

fo = 1.1                        # signal frequency
x = sin(2*pi*fo/fs*arange(Ns)) # sampled signal

ax=axs[1,0]
ax.plot(arange(Ns),x,label='signal')
ax.plot(arange(Ns)+Ns,x,label='extension')
ax.set_ylabel('Amplitude',fontsize=14)
ax.set_title('Discontinuous at endpoints',fontsize=16)
ax=axs[1,1]
Xm = abs(fft.fft(x,N))
idx = int(fo/(fs/N))
ax.stem(arange(N)/N*fs,Xm,basefmt='b-')
ax.plot(fo, Xm[idx],'o')
```

```
ax.set_xlabel('Frequency(Hz)',fontsize=16)
ax.set_ylabel(r'$|X(f)|$',fontsize=18)
ax.text(0.3,0.5,'Spectral Leakage',fontsize=16,
        transform=ax.transAxes,
        bbox={'fc':'y','alpha':.3})
ax.set_xlim(xmax=5)
ax.set_ylim(ymin=-1)


x = x*signal.triang(Ns,2)
ax=axs[2,0]
ax.plot(arange(Ns),x,label='signal')
ax.plot(arange(Ns)+Ns,x,label='extension')
ax.set_xlabel('sample index',fontsize=14)
ax.set_ylabel('Amplitude',fontsize=14)
ax.set_title('Window restores continuity at endpoints',fontsize=14)

ax=axs[2,1]
Xm = abs(fft.fft(x,N))
idx = int(fo/(fs/N))
ax.stem(arange(N)/N*fs,Xm,basefmt='b-')
ax.plot(fo, Xm[idx],'o')
ax.set_xlabel('Frequency(Hz)',fontsize=16)
ax.set_ylabel(r'$|X(f)|$',fontsize=18)
ax.text(0.3,0.5,'Leakage with window',fontsize=16,
        transform=ax.transAxes,
        bbox={'fc':'y','alpha':.3})
ax.set_xlim(xmax=5)
ax.set_ylim(ymin=-1)
```

Listing 4.12: Listing corresponding to Fig. 4.10 illustrating the effect of end-to-end discontinuity and the resulting spectral leakage.

```
fig, ax = subplots()
fig.set_size_inches((6,3))

Ns= 16
Nf = 256*2
freqs = arange(Nf)*2*pi/Nf
w = signal.hanning(Ns,False)
W = db20(w,Nf)

ax.plot(freqs,W,'-b',ms=4.)
ax.set_ylim(ymin = -60)
ax.set_xlim(xmax = pi*1.01)
ax.set_xlabel(r'$\omega$',fontsize=14)
ax.set_ylabel(r'$20\log_{10}|W(\omega)|$',fontsize=18)
ax.grid()
ax.set_title('Hanning Window',fontsize=18)
ax.annotate('',fontsize=28,
            xy=(76/Nf*2*pi,W[0]),
            xytext=(76/Nf*2*pi,W[0]-32),
            arrowprops={'facecolor':'b','arrowstyle':'<->'},
            )
ax.text(0.4,0.5,'Peak sidelobe level',
        fontsize=18,
        transform=ax.transAxes,
        bbox={'fc':'y','alpha':.3})
```

Listing 4.13: Listing corresponding to Fig. 4.13. The `bbox` creates a bounding box for the text with a yellow face-color (i.e. `'fc':'y'`) that is semi-transparent (`alpha`).

```
def peak_sidelobe(w,N=256,return_index=False, return_all=False):
    '''Given window function, return peak sidelobe level and bin index of
       all (return_all=True) or some sidelobe peaks if desired
       (return_index=True). Note that this method fails when the window
       function has no roots on the unit circle (e.g. exponential window).
       The return index is in units of DFT-bin (k/N).
    '''
    assert (len(w)<=N) # need longer DFT otherwise
    r=np.roots(w)      # find complex roots of window function
    r = r[np.where(np.round(abs(r),3)==1)] # keep only those on unit circle
    (approx)
    y=log(r).imag/2./pi*N     # get k^th bin index
    y=y[y>0].astype(np.int32) # keep positive half only as integer roundoff
    y=np.unique(y)            # dump repeated
    y.sort()                  # sort in-place
    W = 20*log10(abs(fft.fft(w,N))) #compute DFT
    # loop through slices and pick out max() as peak for that slice's sidelobe
```

```
18      sidelobe_levels = []
19      sidelobe_idx =[]
20      for s in [slice(i,j) for i,j in zip(y[:-1],y[1:])]:
21          imx= s.start+W[s].argmax() # bin index of max
22          peak= W[imx]-W[0]          # relative to global peak
23          sidelobe_levels.append( peak ) # store sidelobe level for later
24          sidelobe_idx.append(imx/N)       # ... with corresponding bin
25      if return_all:
26          return zip(sidelobe_levels, sidelobe_idx)
27      if return_index:
28          return (sidelobe_levels[0], sidelobe_idx[0])
29      return sidelobe_levels[0]
30
31  def dftmatrix(N=32,Ns=None):
32      'construct DFT matrix of size N give Ns time-samples'
33      k= np.arange(N)
34      if Ns is None: Ns = N
35      n = arange(Ns)
36      U = matrix(exp(1j* 2*pi/N *k*n[:,None])) # use numpy broadcasting to create
37      matrix return U/sqrt(N)
```

Listing 4.14: Code to compute peak sidelobe level.

```
Ns = 64
N= 512

U=dftmatrix(N=N,Ns=Ns)
offset=8 # place DFT near middle of plot for readability
u=array(U[:,offset]).flatten()*sqrt(N) # phase shifts

w = signal.hanning(Ns,False)
level,idx = peak_sidelobe(w,N,return_index=True)
x0 = u*ones(Ns)
x1=u*exp(1j*2*pi*arange(Ns)*(idx))          # signal on peak of sidelobe

fig,axs = subplots(2,1,sharex=True,sharey=True)
fig.set_size_inches((9,6))

ax=axs[0]
ax.plot(db20(w*x0,N))
ax.arrow(offset+idx*N,-60,0,60,
         length_includes_head=True,lw=2.,
         head_length=14,head_width=4,fc='g',alpha=0.3)
#ax.arrow( idx*N,0,0,3,length_includes_head=True,lw=1.5,head_width=2,fc='g')
ax.arrow(offset,-60,0,60,
         length_includes_head=True,
         lw=2.,head_length=14,head_width=4,fc='b')
#ax.legend(loc=0)
ax.set_xlim(xmax=N/4.,xmin=-3)
ax.set_ylim(ymax = 17,ymin=-60)
ax.set_ylabel(r'$20\log_{10}|W_k|$',fontsize=18)
ax.text(0.4,.5,'''The signal (green) on the first sidelobe is reduced
by 31 dB during the convolution but still
contaminates the other signal (blue) by
the residual amount''',va='center',fontsize=12,transform=ax.transAxes);

ax=axs[1]
ax.plot(db20(w*x1,N))
ax.arrow(offset+idx*N,-60,0,60,
         length_includes_head=True,lw=2.,
         head_length=14,head_width=4,fc='g')
ax.arrow(offset,-60,0,60,
         length_includes_head=True,lw=2.,
         head_length=14,head_width=4,fc='b',alpha=0.3)
#ax.legend(loc=0)
ax.set_xlim(xmax=N/4.,xmin=-3)
ax.set_ylim(ymax = 17,ymin=-60)
ax.set_ylabel(r'$20\log_{10}|W_k|$',fontsize=18)
ax.set_xlabel('k',fontsize=16)
ax.text(0.4,.6,'''In this case, the window's mainlobe peak reaches
the green signal and it's the blue signal's turn to
sit on the sidelobe and contaminate the green
signal.''',va='center',fontsize=12,transform=ax.transAxes);
```

Listing 4.15: Listing corresponding to Fig. 4.14.

```
fig,ax = subplots()
fig.set_size_inches((7,3))

N=512
w=signal.windows.hamming(Ns)
W=db20(w,N)

m =10
p=np.polyfit(arange(m)/N*Ns,W[:m]-W[0]+3.01,2) # fit quadratic polynomial
width = np.roots(p)[0]*2 # 3-dB beamwidth

ax.plot(arange(N)/N*Ns,W-W[0]) # normalize to peak
ax.set_ylim(ymin=-10)
ax.set_xlim(xmax = 2)

ax.vlines(width/2,0,-60,lw=2.,linestyle='--',color='g')
ax.set_ylabel('dB',fontsize=22)
ax.set_title(r'$ BW_{3dB}$=%3.2f bins'%width,fontsize=18)
ax.set_xlabel(r'$\frac{N_s}{N} k$',fontsize=22)
ax.annotate('',fontsize=28,xy=(0,-3),
            xytext=(width/2,-3),
            arrowprops=dict(arrowstyle="<->",lw=3))
ax.annotate('',fontsize=28,xy=(1.2,0),
            xytext=(1.2,-3),
            arrowprops=dict(arrowstyle="<->",lw=3))
ax.hlines(-3,width/2,2,linestyle='--',color='g',lw=2.)
ax.text( width/2/4,-5,r'$\frac{BW_{3dB}}{2}$',fontsize=22)
ax.text( 1.3,-2,'-3 dB',fontsize=18)
```

Listing 4.16: Listing corresponding to Fig. 4.15.

```
fig,ax = subplots()
fig.set_size_inches((7,3))

N=256
Ns = 32
w=signal.windows.triang(Ns)
W=db20(w,N)

W0 = db20(exp(1j*2*pi/Ns*arange(Ns)*1/2.)*w,N)-W[0]
W=W-W[0] # rescale for plot

ax.plot(W,label='window')
ax.plot(W0,label='half-bin shifted')
scalloping_loss = W[0]-W0[0]
ax.axis(ymin=-15,ymax=1,xmax=24)

ax.set_title('Triangular Window Scalloping Loss=%3.2f'%(scalloping_loss))
ax.set_xlabel('$k$',fontsize=18)
ax.set_ylabel(r'$20\log_{10}|W|$',fontsize=22)
ax.hlines(-scalloping_loss,0,24,color='red',linestyle='--',lw=2.)
ax.legend(loc=0)
```

Listing 4.17: Listing corresponding to Fig. 4.16.

Chapter 5
Finite Impulse Response Filters

5.1 FIR Filters as Moving Averages

Filtering means preserving certain favored signal frequencies without distorting them while simultaneously suppressing others. Although there are many, many approaches to digital filtering, we focus on Finite Impulse Response (FIR) filters. As the name suggests, these filters have no feedback loops, which means that they stop producing output when the input runs out. These are very popular in practice, with blazing-fast on-chip implementations, and easy-to-understand design specifications. This section introduces the main concepts of FIR filter design.

Finite Impulse Response (FIR) filters have the following form:

$$y_n = \sum_{k=0}^{M-1} h_k x_{n-k} \tag{5.1}$$

with real input x_n and real output y_n. These are called *finite* impulse response because there is no feedback to keep them going indefinitely. These are also sometimes called *moving average* filters or *all-zero* filters. The word *taps* is used for M so a *10-tap* filter has $M = 10$ coefficients. For example, given the two-tap filter, $h_0 = h_1 = 1/2$, we have

$$y_n = x_n/2 + x_{n-1}/2$$

For example, for input $x_n = 1 \forall n \geq 0$, the corresponding output is $y_n = 1 \forall n \geq 1$. Note that we have to wait one sample to fill the filter for $n = 0$ which means we have to wait one sample for a valid filter output. This is the filter's *transient* state. In this case, the output y_n is equal to the input x_n except for this transient shift. Because the constant x_n input corresponds to $\omega = 0$ radial frequency, and the filter leaves this unchanged, we conclude that this two-tap moving-average filter *preserves* the $\omega = 0$ signal.

J. Unpingco, *Python for Signal Processing*, DOI 10.1007/978-3-319-01342-8_5,

As another example, consider $\omega = \pi$ input $x_n = \exp(j\pi n)\,\forall n \geq 0$, with corresponding output, $y_n = 0 \forall n \geq 1$. Unlike the last example, the filter zeros-out this input. These two cases show that this moving average filter eliminates the highest frequency signal ($\omega = \pi$) and preserves the lowest frequency signal ($\omega = 0$). But what about all those frequencies in between? To completely characterize filters by frequency, we need to review the continuous-frequency version of the Fourier Transform.

5.2 Continuous-Frequency Filter Transfer Function

Thus far, we have considered samples of the Fourier transform at discrete frequences ($\omega_k = \frac{2\pi}{N}k$). Now, we want to consider the Fourier Transform of the discrete input for *continuous* frequency defined as the following:

$$H(\omega) = \sum_{n\in\mathbb{Z}} h_n \exp(-j\omega n)$$

Note that this is periodic, $H(\omega) = H(\omega + 2\pi)$. By keeping track of summation indicies, it is not hard to show that

$$Y(\omega) = H(\omega)X(\omega)$$

where $H(\omega)$ is called the *transfer function* or the *frequency response* of the filter h_n. This product of transforms is much easier to work with than convolution and narrows our focus to the properties of $H(\omega)$ that determine filter performance. Now, we can reconsider our two-tap moving average filter by plotting the magnitude and phase of $H(\omega)$.

The top plot in Fig. 5.1 shows the magnitude response of the filter (in dB) and the bottom plot shows the phase response in degrees. At $\omega = 0$, we have $|H(\omega = 0)| = 1$ (i.e. unity-gain) which says that our moving average filter does not change the amplitude of signals at $\omega = 0$. We observed this already with the input $x_n = 1$ that produced $y_n = 1$ as output. On the graph at the other extreme ($\omega = \pi$), we have $|H(\omega = \pi)| = 0$ which we observed earlier for input $x_n = \exp(j\pi n)\,\forall n \geq 0$ that produced all-zero output.

Now, let's consider a signal halfway between the two extremes $0 < \omega = \pi/2 < \pi$ as input and see if we can make sense of the filter's output in the time-domain (shown in Fig. 5.2) using Fig. 5.1. The input signal is then,

$$x_n = \exp(j\pi n/2)\,\forall n \geq 0$$

According to the top plot in Fig. 5.3, the magnitude of the filter at $\omega = \pi/2$ is about -3 dB, which corresponds to $|H(\omega = \pi/2)|^2 = 1/2$ meaning the signal energy at this frequency has been cut in half which is indicated by the lower amplitude of the output signal as compared to the input. According to the bottom plot in Fig. 5.1 the signal phase has been shifted by 45°. To see this, note that the input signal repeats every four samples (360°). A phase shift of 45° which is

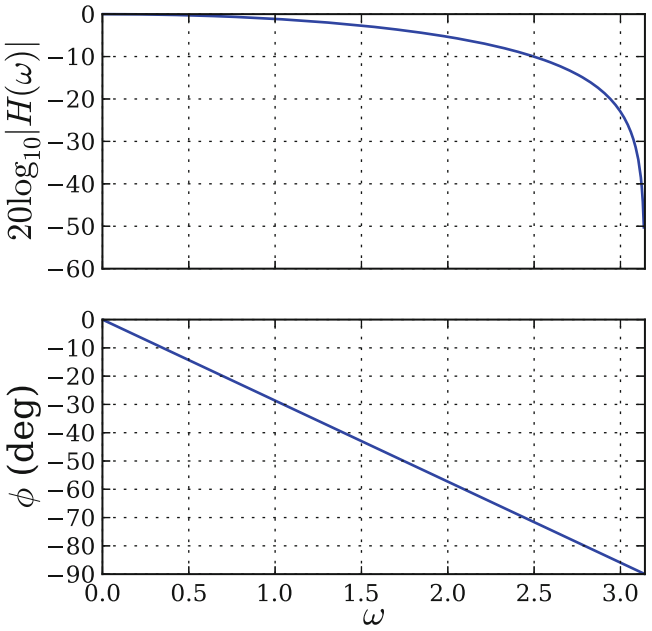


Fig. 5.1 Figure generated by Listing 5.3. The *top plot* shows the magnitude response of the two-tap moving average filter in decibels. The *bottom plot* shows the phase response in degrees

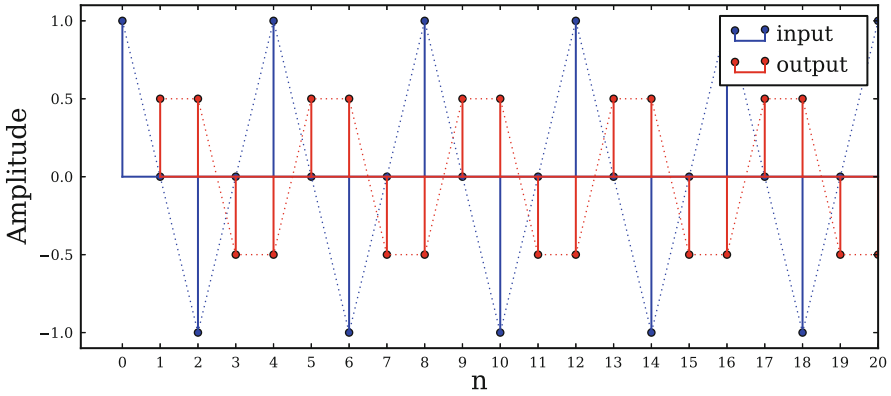


Fig. 5.2 Figure generated by Listing 5.1. This shows the input/output time-domain response of the two-tap moving average filter. Note that the output is delayed by one sample because of the filter's transient. The frequency domain response for this filter is shown in Fig. 5.1. The input frequency is $\omega = \pi/2$

equivalent to a shift of one-half sample. Note that `signal.lfilter` automatically inserts a zero initial condition so we had to drop that one incomplete output point to align the input and output.

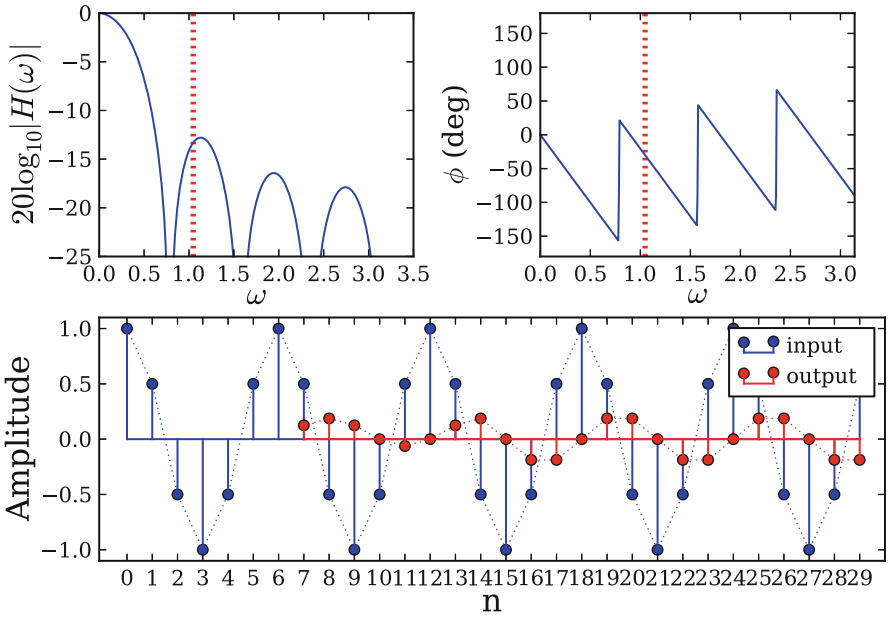


Fig. 5.3 Figure generated by Listing **5.2**. The *top left* shows the magnitude response of the eight-tap filter. The *top left* shows the corresponding phase response. The *bottom plot* shows the input/output time-domain response for the input at frequency $\omega = \pi/3$

What happens when we lengthen our moving average filter to average over eight samples instead of two? Figure 5.3 shows the magnitude (top left) and phase (top right) responses of the longer moving average filter. Note that the magnitude plot shows multiple lobes and dips at discrete frequencies where the output is zero-ed out by the filter. The zig-zag lines of the phase plot are due to the wrap-around of the phase as it coils around 180° mark. The bottom plot shows the input/output sequences. Note that the output shown in Fig. 5.3 is delayed by the length of the input filter. Because the frequency of the input signal is $2\pi/6$, its period is $T = 6$ samples, and the input signal repeats every six samples. According to the phase plot in Fig. 5.1, the phase at this discrete frequency is approximately 30° plus the 180° jump, so the output sequence is shifted over by half a sample ($30/360 = 0.5/6$) plus the three samples (half the six sample period, $180/360 = 1/2$).

By simply increasing the length of the moving average filter, we obtained many more cases (i.e. zeros) where $H(\omega) = 0$. Because our filters produce real outputs y_n given real inputs, x_n, the zeros of the $H(\omega)$ must be in complex conjugate pairs. To analyze the impact of these zeros, we need to generalize the Fourier Transform to the *z-transform*.

5.3 Z-Transform

The filter's z-transform is defined as the following:

$$H(z) = \sum_n h_n z^{-n}$$

The Fourier transform is a special case of the z-transform evaluated on the unit circle ($z = \exp(j\omega)$), but z more generally spans the entire complex plane. Thus, to understand how our moving average filter removes frequencies, we need to compute the complex roots of the z-transform of h_n. Thus, for our eight-tap moving average filter, we have

$$H(z) = \sum_{n=0}^{M-1} h_n z^{-n} = \frac{1}{8}\sum_{n=0}^{7} z^{-n} = \frac{1}{8}(1+z)(1+z^2)(1+z^4)/z^7$$

The first zero occurs when $z = \exp(j\omega) = -1 \Rightarrow \omega = \pi$. The next pair of zeros occurs when $z = \pm j$ which corresponds to $\omega = \pm\pi/2$. Finally, there are more zeros at $\omega = \pm\pi/4$ and at $\omega = \pm 3\pi/4$. Notice that any filter with this $z + 1$ term will eliminate the $\omega = \pi$ (highest) frequency. Likewise, the term $z - 1$ means that the filter zeros out the $\omega = 0$ frequency. In general, the roots of the z-transform *do not* lie on the unit circle. One way to understand FIR filter design is as the judicious placement of these zeros in the complex plane so the resulting transfer function $H(z)$ evaluated on the unit circle satisfies our design specifications. However, the complex-plane is a big place so we need practical considerations to corral this idea.

5.4 Causality

Recall a special case of the Fourier Transform when when the input sequence is symmetric,

$$x_n = x_{-n}$$

that leads to a real-valued (i.e. zero-phase) Fourier transform property,

$$H(\omega) = x_0 \sum_{n \geq 0} 2x_n \cos(\omega n)$$

When the input is anti-symmetric,

$$x_n = -x_{-n}$$

$$H(\omega) = j \sum_{n > 0} 2x_n \sin(\omega n)$$

The Fourier transform is purely imaginary. Observe that $x_n = -x_{-n}$ for $n = 0$ means that $x_0 = 0$.

With that in mind, by changing the indexing in our first moving average filter example from $h_0 = h_1 = 1/2$ to $h_{-1} = h_1 = 1/2$, we can get symmetry around zero with the resulting Fourier transform,

$$H(\omega) = \frac{1}{2}\exp(j\omega) + \frac{1}{2}\exp(-j\omega) = \cos(\omega)$$

which is a *real* function of frequency (with zero-phase). While this is nice theoretically, it is not practical because it requires future-knowledge of the input sequence as shown below

$$y_n = \sum_k h_k x_{n-k} = y_n = h_{-1}x_{n+1} + h_1 x_{n-1} = (x_{n+1} + x_{n-1})/2$$

which shows that y_n depends on the $n + 1$ future value, x_{n+1}. This is what non-causal means and we must omit this kind of symmetry about zero from our class of admissible filter coefficients. Causality (or the lack thereof) is an artifact of the FIR definition in Eq. 5.1. Non-causal filters, such as smoothing filters, do exist, but they use a buffering mechanism outside of our definition.

To enforce causality, we can scoot the symmetric point to the center of the sequence at the cost of introducing a linear phase factor, $\exp(-j\omega(M-1)/2)$. Filters with linear phase do not distort the input phase across frequency. This means that all frequency components of the signal emerge at the other end of the filter in the same order they entered it. Otherwise, it would be very hard to retrieve any information embedded in the signal's phase in later processing. This is the concept of *group delay*. Thus, we can build linear phase causal filters with symmetric coefficients,

$$h_n = h_{M-1-n}$$

or anti-symmetric coefficients,

$$h_n = -h_{M-1-n}$$

by putting the point of symmetry at $(M-1)/2$. Note that this symmetry means that efficient hardware implementations can re-use stored filter coefficients.

5.5 Symmetry and Anti-symmetry

Now that we know how to build linear phase filters with symmetric or anti-symmetric coefficients and enforce causality by centering the point of symmetry, we can collect these facts and examine the resulting consequences. Given

$$h_n = \pm h_{M-1-n}$$

with $h_n = 0 \ \ \forall n \geq M \wedge \forall n < 0$. For even M, the z-transform then becomes,

$$H(z) = \sum_{n=0}^{M-1} z^{-n} h_n = h_0 + h_1 z^{-1} + \ldots + h_{M-1} z^{-M+1}$$

$$= z^{-(M-1)/2} \sum_{n=0}^{M/2-1} h_n \left(z^{(M-1-2n)/2} \pm z^{-(M-1-2n)/2} \right)$$

Likewise, for odd M,

$$H(z) = z^{-(M-1)/2} \left\{ h_{(M-1)/2} + \sum_{n=0}^{(M-3)/2} h_n \left(z^{(M-1-2n)/2} \pm z^{-(M-1-2n)/2} \right) \right\}$$

By substituting $1/z$ and multiplying both sides by $z^{-(M-1)}$, we obtain

$$z^{-(M-1)} H(z^{-1}) = \pm H(z)$$

This equation shows that if z is a root, then so is $1/z$, and because we want a real-valued impulse response, complex roots must appear in conjugate pairs. Thus, if z_1 is a complex root, then its complex conjugate, z_1^*, is also a root, and so is $1/z_1$ and $1/z_1^*$. One complex root generates *four* roots. This means that having M taps on the filter does not imply M independent choices of the filter's roots, or, equivalently, of the filter's coefficients. The symmetry conditions reduce the number of degrees of freedom available in the design.

5.6 Extracting the Real Part of the Filter Transfer Function

We can evaluate these z-transforms on the unit circle when $h_n = +h_{M-n-1}$ to obtain the following,

$$H(\omega) = H_{re}(\omega) \exp\left(-j\omega(M-1)/2\right)$$

where $H_{re}(\omega)$ is a real-valued function that can be written as

$$H_{re}(\omega) = 2 \sum_{n=0}^{(M/2)-1} h_n \cos\left(\omega \frac{M-1-2n}{2} \right)$$

for even M and as

$$H_{re}(\omega) = h_{(M-1)/2} + 2\sum_{n=0}^{(M-3)/2} h_n \cos\left(\omega\frac{M-1-2n}{2}\right)$$

for odd M. Similar results follow when $h_n = -h_{M-1-n}$. For M even, we have

$$H_{re}(\omega) = 2\sum_{n=0}^{M/2-1} h_n \sin\left(\omega\frac{M-1-2n}{2}\right)$$

and for odd M.

$$H_{re}(\omega) = 2\sum_{n=0}^{(M-3)/2} h_n \sin\left(\omega\frac{M-1-2n}{2}\right)$$

By narrowing our focus to $H_{re}(\omega)$ and separating out the linear-phase part, we can formulate design techniques that focus solely on this real-valued function, as we will see later with Parks-McClellen FIR design.

Example 1. We can use these results to reconsider our earlier result for the two-tap moving average filter for which $M = 2$ and $h_0 = 1/2$. Then,

$$H_{re}(\omega) = \cos(\omega/2)$$

with phase,

$$\exp(-j\omega(M-1)/2) = \exp(-j\omega/2)$$

which equals $\exp(-j\pi/4)$ when $\omega = \pi/2$ as we observed numerically earlier. As an exercise, you can plot these expressions for magnitude and phase and compare with Fig. 5.1.

5.7 The Story So Far

We began our work with FIR filters by considering the concepts of linear phase, symmetry, and causality. By defining FIR filter coefficients symmetrically, we were able to enforce both causality and linear phase. We introduced the continuous-frequency version of the Fourier Transform and the more general z-transform as tools to understand the role of zeros in filter design. All this led us to conditions on the filter coefficients that satisfy our practical requirements of linear phase (no phase-distortions across frequency) and causality (no future knowledge of inputs). Finally, we considered the mathematical properties of FIR filters that apply to *any* design.

Sadly, all this work is exactly backwards because all our examples so far started with a set of filter coefficients (h_n) and then analysed filter performance. In a real situation, we *start* with a desired filter performance, and *then* (by various means) come up with the corresponding filter coefficients. Our next section gets into this nitty-gritty.

5.8 Filter Design Using the Window Method

In this section, we start designing FIR filters using the window design method. This is the most straightforward design method and it illustrates the concepts we developed in the previous section. The window method of FIR filter design starts by constructing the ideal filter response, $H_d(\omega)$. For example, consider the ideal lowpass filter with cutoff frequency ω_c,

$$|H_d(\omega)| = 1 \forall \omega \in (-\omega_c, \omega_c)$$

and zero otherwise. The inverse Fourier Transform is the sequence,

$$h_n = \frac{\omega_c}{\pi} \frac{\sin(\omega_c n)}{\omega_c n}$$

if $n \neq 0$ and

$$h_{n=0} = \frac{\omega_c}{\pi}$$

This is obviously non-causal and infinitely long. We can shift the sequence by an arbitrary amount and then truncate to fix both problems, respectively.

Figure 5.4 shows the filter sequence h_n in the top plot. This has been shifted over to enforce causality. The middle plot shows $|H(\omega)|$ as a function of frequency. The vertical dashed lines show $\pm\omega_c$ limits and the horizontal dashed line shows the ideal response, $|H_d(\omega)|$. The bottom plot shows $20 \log_{10} |H(\omega)|$ as a function of frequency. The middle plot reveals rippling in the passband that increases towards the edge of the passband. This is known as the *Gibbs phenomenon* and we need to examine it carefully because it is a serious form of distortion. The bottom plot is 20 times the logarithm of the middle plot and shows the fine detail in the sidelobes on this scale.

Truncating the filter coefficients is the same as multiplying the infinitely-long desired filter sequence (h_d) by a rectangular window (r_n). This is equivalent to a convolution in the frequency domain between the desired response, $H_d(\omega)$, and $R(\omega)$, the Fourier transform of the rectangular window. Thus, the filter output is

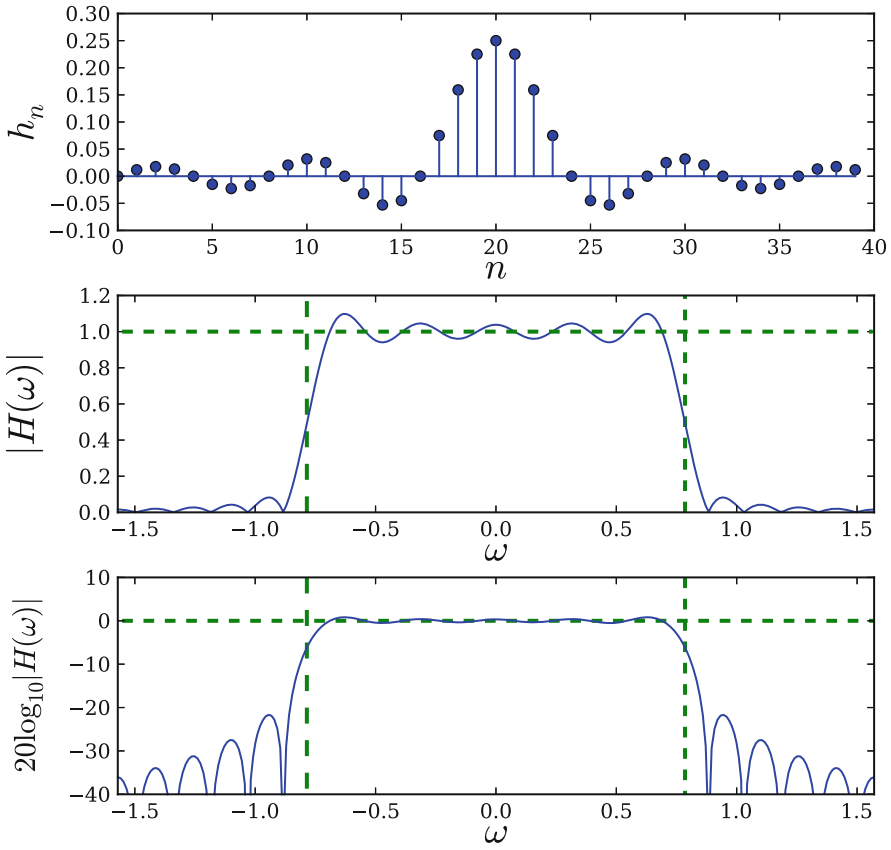


Fig. 5.4 Figure generated by Listing 5.4. The *top plot* shows the filter coefficients. The *middle plot* shows the magnitude response in linear units. The *bottom plot* shows the magnitude response is decibel units. The reason we favor decibel units is that it emphasizes the detail in the small values that the linear plot does not

$$Y(\omega) = \frac{1}{2\pi}\int_{-\pi}^{\pi} H_d(\xi)R(\omega-\xi)d\xi = \frac{1}{2\pi}\int_{-\omega_c}^{\omega_c} R(\omega-\xi)d\xi$$

where

$$R(\omega) = \sum_{n=0}^{N_s-1} \exp(-j\omega n)$$

Because everything is finite, we can combine these two equations to obtain the following:

$$Y(\omega) = \frac{1}{2\pi}\sum_{n=0}^{N_s-1}\int_{-\omega_c}^{\omega_c} \exp(-j(\omega-\xi)n)\,d\xi = \frac{1}{\pi}\sum_{n=0}^{N_s-1}\frac{\sin(n\omega_c)}{n}\exp(j\omega n)$$

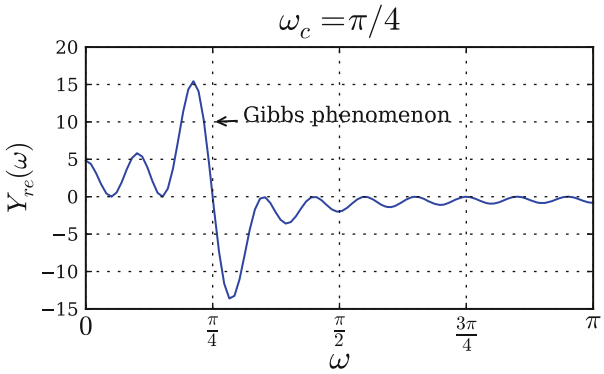


Fig. 5.5 Figure generated by Listing 5.5 that shows the Gibbs phenomenon at the edge of the filter's passband. The cutoff frequency is ω_c

We can expand the summation term to obtain,

$$Y(\omega) = \frac{1}{\pi} \sum_{n=0}^{N_s-1} \frac{\cos(n\omega)\sin(n\omega_c) + j\sin(n\omega)\sin(n\omega_c)}{n}$$

To understand the Gibbs phenomenon at the edges where $\omega = \omega_c$, we recall the following two trigonometric identities:

$$2\cos(n\omega)\sin(n\omega_c) = \sin(n\omega + n\omega_c) - \sin(n\omega - n\omega_c)$$

and likewise,

$$2\sin(n\omega)\sin(n\omega_c) = \cos(n\omega - n\omega_c) - \cos(n\omega + n\omega_c)$$

These two identities show that as $\omega \to \omega_c$, the surviving terms in $Y(\omega)$ summation oscillate at double the cut-off frequency (i.e. $2\omega_c$) which accounts for the intense rippling as ω moves from $\omega = 0$ to $\omega = \omega_c$. Consider the real part of $Y(\omega)$ when $\omega < \omega_c$,

$$Y_{re}(\omega) = \frac{1}{\pi} \sum_{n=0}^{N_s-1} \frac{\sin(n\omega + n\omega_c) - \sin(n\omega - n\omega_c)}{2n}$$

In this case, $\sin(n\omega - n\omega_c) < 0$ so it contributes positively (i.e. constructive interference) to the summation. When $\omega > \omega_c$, the sign reverses and destructive interference reduces the real part. This is the mechanism for the transition from passband to stopband. Figure 5.5 illustrates this effect.

Now that we have a grip on the Gibbs phenomenon, we need to control this form of distortion. Fortunately, we already have all of the tools in place from our prior discussion of window functions.

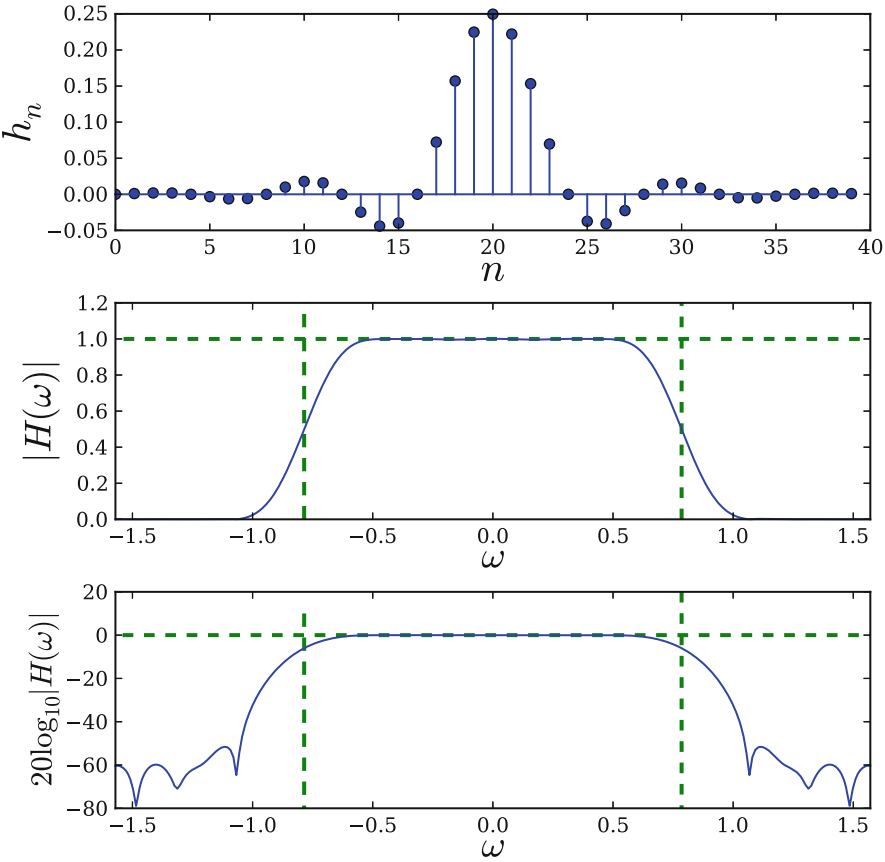


Fig. 5.6 Figure generated by Listing **5.6** that shows the filter coefficients in the *top plot*, the magnitude of the frequency response in the *middle plot* (linear units), and the magnitude of the frequency response in decibel units in the *bottom plot*. In this case, we use a Hamming window to control the Gibbs phenomenon at the edges of the passband. Note that the entire passband is much flatter than the previous case without the Hamming window shown in Fig. 5.4

5.8.1 Using Windows for FIR Filter Design

The root of the Gibbs phenomenon is the sudden truncation of the filter sequence by the rectangular window. We can mitigate this effect by smoothly guiding the filter coefficients to zero using a window function. Figure 5.6 is the same as Fig. 5.4 but now using a Hamming window to terminate the ideal filter sequence. Note that the Gibbs effect is significantly reduced, producing a much flatter but wider mainlobe. The low sidelobes of the Hamming window flatten the Gibbs phenomenon at the edge of the passband.

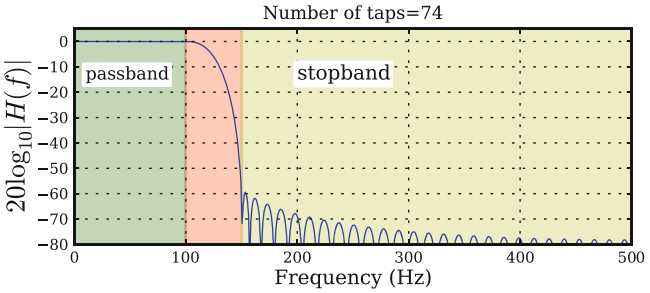


Fig. 5.7 Figure generated by Listing **5.8** showing the passband region, the transition region, and the stopband region from the filter specification. The specification called for 60 dB of attenuation in the stopband and that is clearly shown at the stopband cutoff frequency (150 Hz)

I invite you to download the IPython notebook corresponding to this section and try to generate Fig. 5.6 for different values of M. Note that drop from the passband to the sidelobes steepens with increasing M. As we have seen before with window functions, this is because the window's mainlobe narrows with increasing sequence length. Naturally, you can use other window functions besides Hamming and change the sidelobe level and mainlobe width. The most popular window used for this method of filter design is the Kaiser-Bessel window because it provides extra parameters for tuning the frequency response.

The `signal.fir_filter_design` module provides functions for filter design using the Kaiser-Bessel window (among other windows). For example, to design a lowpass filter using the Kaiser-Bessel window, we need the a subset of the following parameters: maximum passband ripple (δ), width of the transition region, Kaiser-Bessel parameter β, and the number of filter taps. The `fir_filter_design` provides tools to solve for some of these parameters given others. For example, the `kaiserord` function can determine the number of taps given the maximum allowable passband ripple and the width of the transition region.

Example 2. Consider the following low-pass filter specification in Listing **5.7**. In listing **5.8**, we find the number of taps and the Kaiser-Bessel β parameter using the `kaiserord` function. For the FIR window design method, the δ parameter is simultaneously the maximum allowable passband ripple and the desired attenuation in the stopband. The resulting filter is shown in Fig. 5.7.

Let's consider the performance of the filter with two equal-amplitude single-frequency tones, one in the passband and one in the stopband. Figure 5.8 shows the frequency domain input and the corresponding filtered output. Because second tone is in the stopband, and its relative energy is approximately 40 dB, it is completely extinguished by the filter shown in Fig. 5.7, whose attenuation in the stopband is 60 dB. I invite you to download the IPython Notebook corresponding to this section and try different amplitude values for the tone in the stopband.

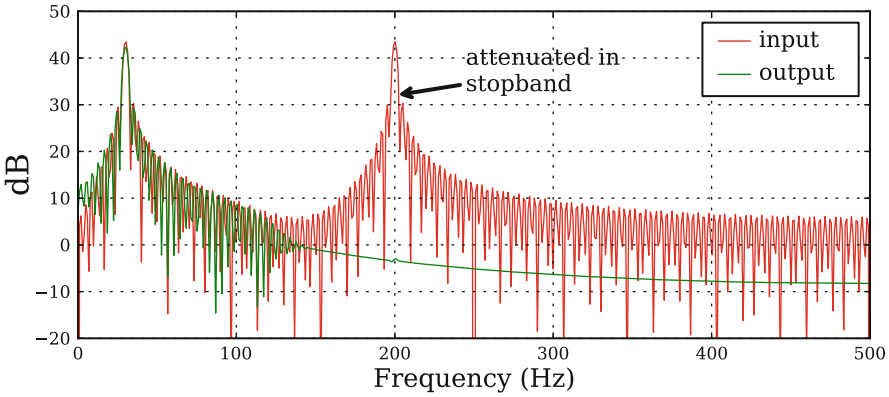


Fig. 5.8 Figure generated by Listing 5.9 showing the application of the filter on two equal-amplitude input tones where one tone is in the passband and the other in the stopband. The tone in the passband passes through to the output but the tone in the stopband is completely eliminated by the filter because the attenuation in the stopband is 60 dB and the tone is only 40 dB

5.9 The Story So Far

The window design method is the easiest FIR design method to understand and it draws upon what we already learned about window functions. The Kaiser-Bessel window is widely used in this method because it provides design flexibility and easy-to-understand filter specifications. Furthermore, there are many closed form approximations to the various derived terms (e.g. β, M) which means that iterative algorithms are not necessary and that engineering trade-offs appear explicitly in a formula. In the next section we consider Parks-McClellan FIR design that solves for the desired filter sequence using a powerful iterative exchange algorithm.

5.10 Filter Design Using the Parks-McClellan Method

In this section, we equate the filter design problem as the search for an optimal Chebyshev polynomial that has a minimal maximum deviation from the ideal desired response, $H_d(\omega)$. The Parks-McClellan algorithm generates filter coefficients by solving the *optimum equiripple Chebyshev approximation* to the ideal filter characterized by the specification. It turns out that solving for such polynomials employs the Remez exchange algorithm so the implemented function in `signal.fir_filter_design` is called `remez`, even though the application of this algorithm to FIR filter design is just part of the Parks-McClellan algorithm. Unfortunately, the algorithm itself is based on certain advanced theorems beyond our scope, so we will just use it to get acceptable filter coefficients.

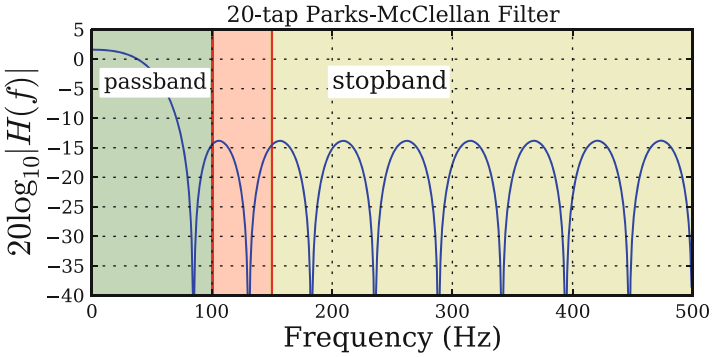


Fig. 5.9 Figure generated by Listing **5.10** showing the magnitude response of the filter we designed using the Parks-McClellan methods

The `remez` function takes the `numtaps` argument which is M in our notation, the `bands` argument is a sequence of passband/stopband edges in normalized frequency (i.e. scaled by $f_s/2$), the `desired` argument is a numpy array (or other array-like iterable) that is half the length of the `bands` argument and contains the desired gain in each of the specified bands. The next argument is the optional `weight` which is array-like, half the length of the `bands` argument, and provides the relative weighting for the passband/stopband. The unfortunately-named optional `Hz` argument (`default=1`) is the sampling frequency in the same units as the `bands`. The next optional argument is the type of filter response in each of the bands (i.e. bandpass, hilbert). The default is `bandpass` filter. The remaining arguments of the function have to do with the internal operation of the iterative algorithm. Now, let's see this approach in action.

Example 3.

Suppose we operate at a 1 kHz sample rate and we want a 20-tap lowpass filter with a passband up to 100 Hz and a stopband starting at 150 Hz. Figure 5.9 shows the frequency response for the FIR filter constructed in Listing **5.10**. The pink region is the transitional region between the pass and stop bands. As shown, the attenuation in the stopband only provides approximately 15 dB of attenuation. This means that a signal greater than 15 dB in the stopband will leak into the signals in the passband. One way to increase the attenuation in the stopband is to increase the filter order, M, as shown in the Fig. 5.10 below.

Figure 5.10 shows the frequency response when the filter order is doubled. Note that the attenuation in the stopband has improved but there is now a significant distortion due to the ripple in the passband. Given the same filter order, M, we can mitigate this with the `weight` argument as shown in Listing **5.12**. As the Fig. 5.12 shows, using the `weight` argument allows flattening of the passband at the expense of raising the stopband attenuation. This argument influences the iterative algorithm to penalize errors in the passband much more than in the stopband. I invite you to

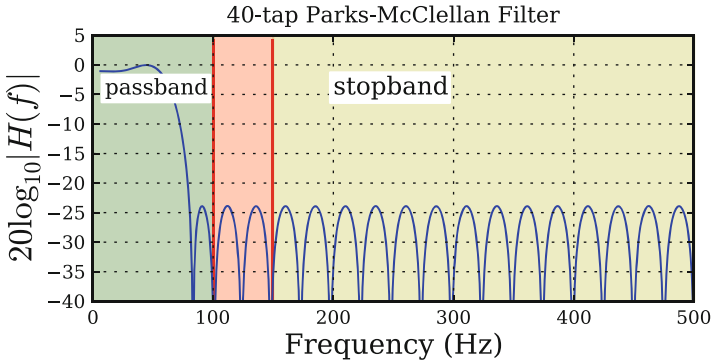


Fig. 5.10 Figure generated by Listing **5.11** showing the frequency response when the filter order is doubled. As compared to the previous case in Fig. 5.9, the attenuation in the stopband has improved, but now there is significant distortion in the passband

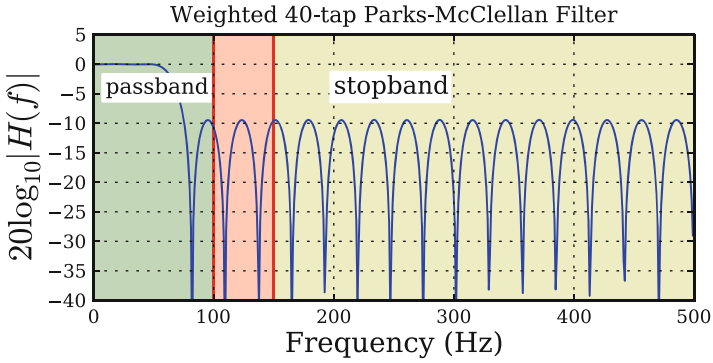


Fig. 5.11 Figure generated by Listing **5.12** showing the magnitude response of the filter and the embedded passband, transition band, and stopband

download the IPython Notebook corresponding to this section to try different filter orders and weights. Let's now consider the performance of this filter in practice (Fig. 5.11).

Figure 5.12 shows the input and output of the filter where the input consists of two equal-amplitude tones, one in the passband and one in the stopband. Because the filter's stopband provides approximately 10 dB of attenuation, the tone in the stopband is reduced by this amount. Note that the tone in the passband remains unchanged because the passband attenuation is negligible (as it was designed!). Let's consider the input and output signals in the time domain in Fig. 5.13.

Figure 5.13 breaks down the time-domain response of the filter. The top plot shows the two signals separately. The middle plot shows the sum of these two signals that is the filter's input. The bottom plot shows the filter's output compared to the input signal that was in the filter's passband. The hope for this filter was that the tone

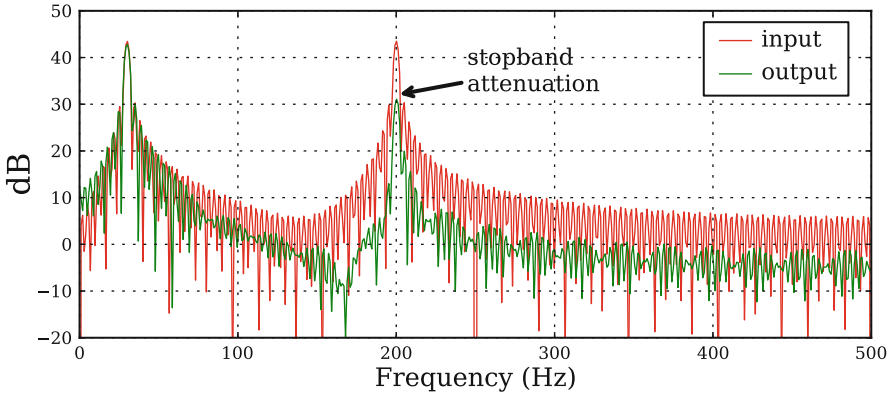


Fig. 5.12 Figure generated by Listing 5.13 showing the filter applied to two inputs tones, one in the passband and the other in the stopband. Because we preferentially weighted the passband over the stopband, the tone in the stopband still comes through because the attenuation there is only 10 dB and the signal peaks at approximately 40 dB. This leaves roughly 30 dB of signal *left* as shown

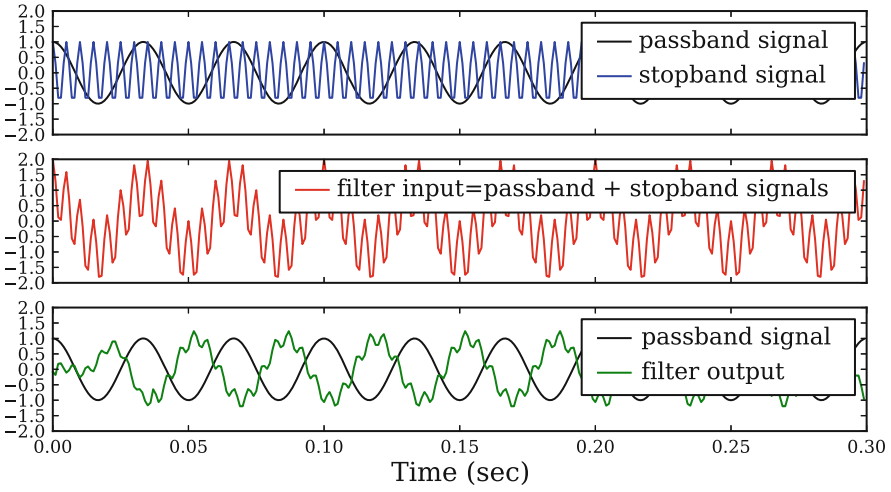


Fig. 5.13 Figure generated by Listing 5.14. The *top plot* shows the two input tones separately. The *middle plot* is the sum of the two tones in the *top plot*. The *bottom plot* shows the passband signal and the filter's output. The shift of the output comes from the filter's phase. Note that the time-domain output still has the wiggle in it from the tone in the stopband that was not sufficiently attenuated

in the passband would be the only signal to survive at the output. In the bottom plot in Fig. 5.13, observe that the output signal is shifted compared to the input. This is due the phase response of the filter. Because this is a FIR filter, this phase is linear. Also notice that the stopband signal that caused the ripple at the filter's input is still

obvious in the filter's output. This is a result of the relatively poor 10 dB attenuation in the stopband that we noted earlier. If we want to eliminate this rippling effect entirely, we have to design a filter with much greater attenuation in the stopband.

I invite you to download the IPython Notebook corresponding to this section and change the amplitude of the signal in the stopband and see how it affects the filter's output.

5.11 Summary

In this section, we covered the Parks-McClellan algorithm that generates FIR filter coefficients by solving the optimum equiripple Chebyshev approximation to the ideal filter provided in the specification. This design technique as implemented in the `signal.remez` function allows the designer to specify the passband and stopband frequencies, the idealized response in the passband, and optional relative weighting of the desired response in the passband as opposed to the stopband. All of these filter design techniques are subject to the *waterbed effect* which means that pushing the filter's response down in one area (i.e. increasing attenuation in the stopband) just pushes it up elsewhere in the frequency domain, potentially causing passband distortion as we illustrated. It is the designer's responsibility to reconcile these competing demands for filter performance in terms of the overarching cost of complexity (i.e. filter length, numerical precision of coefficients) that may dominate the filter's implementation in hardware.

One thing we did not discuss is how to pick the filter order given a desired passband/stopband specification. Unfortunately, this is left to trial-and error and the intuition of the filter designer, because unlike the windowing method with Kaiser-Bessel windows, there are no closed-form approximations for these parameters. Note that there are many other ways to design FIR filters, each in interpreting the FIR design problem differently.

Appendix

```
from scipy import signal

Ns=30 # length of input sequence
n= arange(Ns) # sample index
x = cos(arange(Ns)*pi/2.)
y= signal.lfilter([1/2.,1/2.],1,x)

fig,ax = subplots(1,1)
fig.set_size_inches(12,5)

ax.stem(n,x,label='input',basefmt='b-')
ax.plot(n,x,':')
ax.stem(n[1:],y[:-1],markerfmt='ro',linefmt='r-',label='output')
ax.plot(n[1:],y[:-1],'r:')
ax.set_ylim(ymin=-1.1,ymax=1.1)
ax.set_xlabel("n",fontsize=22)
ax.legend(loc=0,fontsize=18)
ax.set_xticks(n)
ax.set_xlim(xmin=-1.1,xmax=20)
ax.set_ylabel("Amplitude",fontsize=22);
```

Listing 5.1: Listing for Fig. 5.2. The `signal.filter` function implements Eq. 5.1.

```
from matplotlib import gridspec

fig=figure()
#fig.set_size_inches((8,5))

gs = gridspec.GridSpec(2,2)
# add vertical and horizontal space
gs.update(wspace=0.5, hspace=0.5)

ax = fig.add_subplot(subplot(gs[0,0]))

ma_length = 8 # moving average filter length
w,h=signal.freqz(ones(ma_length)/ma_length,1)
ax.plot(w,20*log10(abs(h)))
ax.set_ylabel(r"$ 20 \log_{10}|H(\omega)| $",fontsize=18)
ax.set_xlabel(r"$\omega$",fontsize=18)
ax.vlines(pi/3,-25,0,linestyles=':',color='r',lw=3.)
ax.set_ylim(ymin=-25)

ax = fig.add_subplot(subplot(gs[0,1]))
ax.plot(w,angle(h,deg=True))
ax.set_xlabel(r'$\omega$',fontsize=18)
ax.set_ylabel(r"$\phi $ (deg)",fontsize=16)
ax.set_xlim(xmax = pi)
ax.set_ylim(ymin=-180,ymax=180)
ax.vlines(pi/3,-180,180,linestyles=':',color='r',lw=3.)
ax = fig.add_subplot(subplot(gs[1,:]))
Ns=30
n= arange(Ns)
x = cos(arange(Ns)*pi/3.)
y= signal.lfilter(ones(ma_length)/ma_length,1,x)

ax.stem(n,x,label='input',basefmt='b-')
ax.plot(n,x,':')
ax.stem(n[ma_length-1:],y[:-ma_length+1],
        markerfmt='ro',
        linefmt='r-',
        label='output')
ax.plot(n[ma_length-1:],y[:-ma_length+1],'r:')
ax.set_xlim(xmin=-1.1)
ax.set_ylim(ymin=-1.1,ymax=1.1)
ax.set_xlabel("n",fontsize=18)
ax.set_xticks(n)
ax.legend(loc=0)
ax.set_ylabel("Amplitude",fontsize=18);
```

Listing 5.2: Listing for Fig. 5.3. The `signal.freqz` function computes the filter's magnitude ($|H(\omega)|$) and phase response given the filter coefficents.

```
from scipy import signal

fig, axs = subplots(2,1,sharex=True)
subplots_adjust(hspace = .2)
fig.set_size_inches((5,5))

ax=axs[0]
w,h=signal.freqz([1/2., 1/2.],1) # Compute impulse response
ax.plot(w,20*log10(abs(h)))
ax.set_ylabel(r"$20 \log_{10} |H(\omega)| $",fontsize=18)
ax.grid()

ax=axs[1]
ax.plot(w,angle(h,deg=True))
ax.set_xlabel(r'$\omega$',fontsize=18)
ax.set_ylabel(r"$\phi $ (deg)",fontsize=18)
ax.set_xlim(xmax = pi)
ax.grid()
```

Listing 5.3: Listing corresponding to Fig. 5.1. The `angle` function returns the angle of the complex number. The `deg=True` option returns degrees instead of radians.

```
from scipy import signal
from numpy import fft

wc = pi/4
M=20
N = 512 # DFT size
n = arange(-M,M)
h = wc/pi * sinc(wc*(n)/pi)  # see definition of np.sinc()

w,Hh = signal.freqz(h,1,whole=True, worN=N) # get entire frequency domain
wx = fft.fftfreq(len(w)) # shift to center for plotting

fig,axs = subplots(3,1)
fig.set_size_inches((8,8))
subplots_adjust(hspace=0.3)
ax=axs[0]
ax.stem(n+M,h,basefmt='b-')
ax.set_xlabel("$n$",fontsize=22)
ax.set_ylabel("$h_n$",fontsize=22)

ax=axs[1]
ax.plot(w-pi,abs(fft.fftshift(Hh)))
ax.axis(xmax=pi/2,xmin=-pi/2)
ax.vlines([-wc,wc],0,1.2,color='g',lw=2.,linestyle='--',)
ax.hlines(1,-pi,pi,color='g',lw=2.,linestyle='--',)
ax.set_xlabel(r"$\omega$",fontsize=22)
ax.set_ylabel(r"$|H(\omega)|$",fontsize=22)

ax=axs[2]
ax.plot(w-pi,20*log10(abs(fft.fftshift(Hh))))
ax.axis(ymin=-40,xmax=pi/2,xmin=-pi/2)
ax.vlines([-wc,wc],10,-40,color='g',lw=2.,linestyle='--',)
ax.hlines(0,-pi,pi,color='g',lw=2.,linestyle='--',)
ax.set_xlabel(r"$\omega$",fontsize=22)
ax.set_ylabel(r"$20\log_{10}|H(\omega)|$",fontsize=18)
```

Listing 5.4: Listing corresponding to Fig. 5.4. The `fftshift` function reorganizes the DFT so that the frequencies are centered on zero ($f \in [-f_s/2, f_s/2]$) instead of on $f_s/2$ ($f \in [0, f_s]$).

```
fig,ax = subplots()
fig.set_size_inches(6,3)

k=arange(M)
omega = linspace(0,pi,100)

ax.plot(omega,(sin(k*omega[:,None]+k*wc)
                -sin(k*omega[:,None]-k*wc)).sum(axis=1))
ax.set_ylabel(r"$Y_{re}(\omega)$",fontsize=18)
ax.grid()
t=ax.set_title(r"$\omega_c = \pi/4$",fontsize=22)
t.set_y(1.03) # scoot title up a bit
ax.set_xlabel(r"$\omega$",fontsize=22)
# setup xticks and labels for LaTeX
ax.set_xticks([0, pi/4,pi/2.,3*pi/4, pi,])
ax.set_xticklabels(['$0$',r'$\frac{\pi}{4}$',r'$\frac{\pi}{2}$',
                    r'$\frac{3\pi}{4}$', r'$\pi$'],fontsize=18)
ax.set_xlim(xmax=pi)
ax.annotate("Gibbs phenomenon",xy=(pi/4,10),fontsize=14,
            xytext=(20,0),
            textcoords='offset points',
            arrowprops={'facecolor':'b','arrowstyle':'->'})
```

Listing 5.5: Listing corresponding to Fig. 5.5 that shows the Gibbs phenomenon at the edge of the filter's passband. The `set_y` function moves the title a bit up so the fonts can be easily seen.

```
wc = pi/4

M=20

N = 512 # DFT size
n = arange(-M,M)
win = signal.hamming(len(n))
h = wc/pi * sinc(wc*(n)/pi)*win  # see definition of np.sinc()

w,Hh = signal.freqz(h,1,whole=True, worN=N) # get entire frequency domain
wx = fft.fftfreq(len(w)) # shift to center for plotting

fig,axs = subplots(3,1)
fig.set_size_inches((8,8))
subplots_adjust(hspace=0.3)

ax=axs[0]
ax.stem(n+M,h,basefmt='b-')
ax.set_xlabel("$n$",fontsize=24)
ax.set_ylabel("$h_n$",fontsize=24)
ax=axs[1]
ax.plot(w-pi,abs(fft.fftshift(Hh)))
ax.axis(xmax=pi/2,xmin=-pi/2)
ax.vlines([-wc,wc],0,1.2,color='g',lw=2.,linestyle='--',)
ax.hlines(1,-pi,pi,color='g',lw=2.,linestyle='--',)
ax.set_xlabel(r"$\omega$",fontsize=22)
ax.set_ylabel(r"$|H(\omega)| $",fontsize=22)

ax=axs[2]
ax.plot(w-pi,20*log10(abs(fft.fftshift(Hh))))
ax.axis(ymin=-80,xmax=pi/2,xmin=-pi/2)
ax.vlines([-wc,wc],10,-80,color='g',lw=2.,linestyle='--',)
ax.hlines(0,-pi,pi,color='g',lw=2.,linestyle='--',)
ax.set_xlabel(r"$\omega$",fontsize=22)
ax.set_ylabel(r"$20\log_{10}|H(\omega)| $",fontsize=18)
```

Listing 5.6: Listing corresponding to Fig. 5.6. The `fftfreq` function generates the DFT sample frequencies.

```
Ns =300 # number of samples
N = 1024 # DFT size

fs = 1e3 # sample rate in Hz
fpass = 100 # in Hz
fstop = 150 # in Hz
delta = 60 # in dB, desired attenuation in stopband
```

Listing 5.7: Listing showing the filter specification for our example.

```
from matplotlib.patches import Rectangle

M,beta= signal.fir_filter_design.kaiserord(delta, (fstop-fpass)/(fs/2.))

hn = signal.firwin(M,(fstop+fpass)/2.,window=('kaiser',beta),nyq=fs/2.)
w,H = signal.freqz(hn) # frequency response

fig,ax = subplots()
fig.set_size_inches((8,3))

ax.plot(w/pi*fs/2.,20*log10(abs(H)))
ax.set_xlabel("Frequency (Hz)",fontsize=16)
ax.set_ylabel(r"$20\log_{10} |H(f)| $",fontsize=22)
ymin,ymax = -80,5
ax.axis(ymin = ymin,ymax=ymax)
ax.add_patch(Rectangle((0,ymin),width=fpass,
             height=ymax-ymin,
             color='g',alpha=0.3))
ax.add_patch(Rectangle((fpass,ymin),width=fstop-fpass,
             height=ymax-ymin,
             color='r',alpha=0.3))
ax.add_patch(Rectangle((fstop,ymin),width=fs/2-fstop,
             height=ymax-ymin,
             color='y',alpha=0.3))
ax.set_title("Number of taps=%d"%M)
ax.text(10,-15,'passband',fontsize=14,bbox=dict(color='white'))
ax.text(200,-15,'stopband',fontsize=16,bbox=dict(color='white'))
ax.grid()
```

Listing 5.8: Listing corresponding to Fig. 5.7. The `bbox` puts the text in the foreground of a colored box.

```
from numpy import fft

t = arange(0,Ns)/fs
x = cos(2*pi*30*t)+cos(2*pi*200*t)
X = fft.fft(x,N)

y=signal.lfilter(hn,1,x)
Y = fft.fft(y,N)

fig,ax = subplots()
fig.set_size_inches((10,4))
ax.plot(arange(N)/N*fs,20*log10(abs(X)),'r-',label='input')
ax.plot(arange(N)/N*fs,20*log10(abs(Y)),'g-',label='output')
ax.set_xlim(xmax = fs/2)
ax.set_ylim(ymin=-20)
ax.set_ylabel(r'dB',fontsize=22)
ax.set_xlabel("Frequency (Hz)",fontsize=18)
ax.grid()
ax.annotate('attenuated in\nstopband',fontsize=16,xy=(200,32),
             xytext=(50,3),textcoords='offset points',
             arrowprops=dict(arrowstyle='->',lw=3),
            )
ax.legend(loc=0,fontsize=16);
```

Listing 5.9: Listing corresponding to Fig. 5.8.

```
from matplotlib.patches import Rectangle
from scipy import signal

fs = 1e3 # sample rate in Hz
M = 20
fpass = 100 # in Hz
fstop = 150 # in Hz

hn = signal.remez(M,
                  array([0, fpass, fstop, fs])/2., # scaled passband, and stop
                  band [1,0],  # low pass filter
                  Hz = fs, # sampling frequency
                  )

w,H=signal.freqz(hn,1) # frequency response

def apply_plot_overlay():
    'convenience function to illustrate stop/passband in frequency response
    plot'
    ax.plot(w/pi*(fs/2),20*log10(abs(H)),label='Filter response')
    ax.set_ylim(ymax=5)
    ax.vlines(100,*ax.get_ylim(),color='r')
    ax.vlines(150,*ax.get_ylim(),color='r')
    ax.set_ylim(ymin=-40)
    ax.set_xlabel("Frequency (Hz)",fontsize=18)
    ax.set_ylabel(r"$20\log_{10}|H(f)|$",fontsize=22)
    ax.add_patch(Rectangle((0,-40),width=fpass,height=45,color='g',alpha=0.3))
    ax.add_patch(Rectangle((fpass,-40),width=fstop-fpass,height=45,color='r',
    alpha=0.3))
    ax.add_patch(Rectangle((fstop,-40),width=fs/2-fstop,height=45,color='y',
    alpha=0.3))
    ax.text(10,-5,'passband',fontsize=14,bbox=dict(color='white'))
    ax.text(200,-5,'stopband',fontsize=16,bbox=dict(color='white'))
    ax.grid()

fig,ax = subplots()
fig.set_size_inches((7,3))
apply_plot_overlay()
ax.set_title('%d-tap Parks-McClellan Filter'%M)
```

Listing 5.10: Listing corresponding to Fig. 5.9. The `remez` function computes the optimal filter coefficients for the Parks-McClellan method. The `numtaps` argument which is M in our notation, the `bands` argument is a sequence of passband/stopband edges in normalized frequency (i.e. scaled by $f_s/2$), the `desired` argument is a Numpy array (or other array-like iterable) that is half the length of the `bands` argument and contains the desired gain in each of the specified bands. The next argument is the optional `weight` which is array-like, half the length of the `bands` argument, and provides the relative weighting for the passband/stopband. The `Hz` argument (`default=1`) is the sampling frequency in the same units as the `bands`. The next optional argument is the type of filter response in each of the bands (i.e. bandpass, hilbert). The default is `bandpass` filter. The remaining arguments of the function call have to do with the internal operation of the iterative algorithm.

```
M = 40 # double filter length
hn = signal.remez(M,
                  array([0, fpass, fstop, fs])/2., # scaled passband, and stop
                  band [1,0],  # low pass filter
                  Hz = fs, # sampling frequency
                  )

w,H=signal.freqz(hn,1) # frequency response
fig,ax = subplots()
fig.set_size_inches((7,3))
apply_plot_overlay()
ax.set_title('%d-tap Parks-McClellan Filter'%M)
```

Listing 5.11: Listing corresponding to Fig. 5.10.

```
hn = signal.remez(M,
                  array([0, fpass, fstop, fs])/2., # scaled passband, and stop
                  band [1,0],  # low pass filter
                  weight=[100,1], # passband 100 times more important than
                  stopband Hz = fs, # sampling frequency
                  )

w,H=signal.freqz(hn,1) # frequency response
fig,ax = subplots()
fig.set_size_inches((7,3))
apply_plot_overlay()
ax.set_title('Weighted %d-tap Parks-McClellan Filter'%M)
```

Listing 5.12: Listing corresponding to Fig. 5.11. By using the `weight` option in the `remez` function, we influenced the algorithm to penalize errors in the passband more than errors in the stopband.

```
Ns =300 # number of samples
N = 1024 # DFT size
t = arange(0,Ns)/fs

x = cos(2*pi*30*t)+cos(2*pi*200*t)
#x = x*signal.hamming(Ns) # try windowing also!
X = fft.fft(x,N)

y=signal.lfilter(hn,1,x)
Y = fft.fft(y,N)

fig,ax = subplots()
fig.set_size_inches((10,4))
apply_plot_overlay()
ax.set_ylim(ymin=-30,ymax=7)
ax.legend(loc='upper left',fontsize=16)


ax2 = ax.twinx()
ax2.plot(arange(N)/N*fs,20*log10(abs(X)),'r-',label='filter input')
ax2.plot(arange(N)/N*fs,20*log10(abs(Y)),'g-',label='filter output')
#ax2.plot(arange(N)/N*fs,20*log10(abs(X)*abs(H)),'g:',lw=2.,label='YY')
ax2.set_xlim(xmax = fs/2)
ax2.set_ylim(ymin=-20)
ax2.set_ylabel(r'$20\log|Y(f)|$',fontsize=22)
ax2.legend(loc=0,fontsize=16);

fig,ax = subplots()
fig.set_size_inches((10,4))
ax.plot(arange(N)/N*fs,20*log10(abs(X)),'r-',label='input')
ax.plot(arange(N)/N*fs,20*log10(abs(Y)),'g-',label='output')
ax.set_xlim(xmax = fs/2)
ax.set_ylim(ymin=-20)
ax.set_ylabel('dB',fontsize=22)
ax.set_xlabel("Frequency (Hz)",fontsize=18)
ax.grid()
ax.annotate('stopband\nattenuation',fontsize=16,xy=(200,32),
             xytext=(50,3),textcoords='offset points',
             arrowprops=dict(arrowstyle='->',lw=3),
            )
ax.legend(loc=0,fontsize=16);
```

Listing 5.13: Listing corresponding to Fig. 5.12.

```
x_pass = cos(2*pi*30*t) # passband signal
x_stop = cos(2*pi*200*t) # stopband signal
x = x_pass + x_stop
y=signal.lfilter(hn,1,x)

fig,axs = subplots(3,1,sharey=True,sharex=True)
fig.set_size_inches((10,5))

ax=axs[0]
ax.plot(t,x_pass,label='passband signal',color='k')
ax.plot(t,x_stop,label='stopband signal')
ax.legend(loc=0,fontsize=16)

ax=axs[1]
ax.plot(t,x,label='filter input=passband + stopband signals',color='r')
ax.legend(loc=0,fontsize=16)

ax=axs[2]
ax.plot(t,x_pass,label='passband signal',color='k')
ax.plot(t,y,label='filter output',color='g')
ax.set_xlabel("Time (sec)",fontsize=18)
ax.legend(loc=0,fontsize=16);
```

Listing 5.14: Listing corresponding to Fig. 5.13.

References

[CBB+05] H. Childs, E.S. Brugger, K.S. Bonnell, J.S. Meredith, M. Miller, B.J. Whitlock, N. Max, A contract-based system for large data visualization, in *Proceedings of IEEE Visualization 2005*, Minneapolis, 2005, pp. 190–198

[Haran] F.J. Harris, On the use of windows for harmonic analysis with the discrete fourier transform. Proc. IEEE **66**(1), 51–83 (1978)

[Laf90] P. Lafrance, *Fundamental Concepts in Communication* (Prentice-Hall, Inc., Englewood, 1990)

[Lan09] H.P. Langtangen, *Python Scripting for Computational Science*, vol. 3 (Springer, Berlin, 2009)

[MP73] J. McClellan, T. Parks, A united approach to the design of optimum fir linear-phase digital filters. IEEE Trans. Circuit Theory **20**(6), 697–701 (1973)

[MPR73] J. McClellan, T.W. Parks, L. Rabiner, A computer program for designing optimum fir linear phase digital filters. IEEE Trans. Audio Electroacoust. **21**(6), 506–526 (1973)

[Oli06] T.E. Oliphant, *A Guide to Numpy*, vol. 1 (Trelgol Publishing, USA, 2006)

[OW96] A.V. Oppenheim, A.S. Willsky, *Signals and Systems* (Prentice-Hall, Upper Saddle River, 1996)

[OSB+89] A.V. Oppenheim, R.W. Schafer, J.R. Buck et al., *Discrete-Time Signal Processing*, vol. 2 (Prentice Hall, Englewood Cliffs, 1989)

[Orf95] S.J. Orfanidis, *Introduction to Signal Processing* (Prentice-Hall, Englewood Cliffs, 1995)

[Pap77] A. Papoulis, *Signal Analysis* (McGraw-Hill, New York, 1977)

[Pro01] J.G. Proakis, *Digital Signal Processing: Principles Algorithms and Applications* (Pearson Education, New Delhi, 2001)

[Sle78] D. Slepian, Prolate spheroidal wave functions, Fourier analysis, and uncertainty. ATT Tech. J. **57**, 1371–1430 (1978)

J. Unpingco, *Python for Signal Processing*, DOI 10.1007/978-3-319-01342-8,

Symbols

$\forall$	For all
$\in$	Element of
$\mathbb{C}$	Set of complex numbers
$\mathbb{E}$	Expectation
$\mathbb{R}$	Set of real numbers
$\mathbb{R}^n$	n-dimensional vector of real numbers
$\mathbb{Z}$	Set of integers
$\odot$	Hadamard product

J. Unpingco, *Python for Signal Processing*, DOI 10.1007/978-3-319-01342-8,

Index

J. Unpingco, *Python for Signal Processing*, DOI 10.1007/978-3-319-01342-8,