## Contents

1 **Installation**  
2 **Documentation**  
3 **Roadmap**  
   3.1 Basic Concepts  
   3.2 Scale Creation  
   3.3 Metric Functions  
   3.4 Scale Analysis and Creation (Redux)  
   3.5 Tuning Tables  
   3.6 Number Theory  
   3.7 Utilities  
   3.8 Visualizations  
   3.9 Using PyTuning Interactively  
4 **Index**

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Installation</td>
<td>3</td>
</tr>
<tr>
<td>Documentation</td>
<td>5</td>
</tr>
<tr>
<td>Roadmap</td>
<td>7</td>
</tr>
<tr>
<td>3.1 Basic Concepts</td>
<td>7</td>
</tr>
<tr>
<td>3.2 Scale Creation</td>
<td>12</td>
</tr>
<tr>
<td>3.3 Metric Functions</td>
<td>18</td>
</tr>
<tr>
<td>3.4 Scale Analysis and Creation (Redux)</td>
<td>20</td>
</tr>
<tr>
<td>3.5 Tuning Tables</td>
<td>26</td>
</tr>
<tr>
<td>3.6 Number Theory</td>
<td>32</td>
</tr>
<tr>
<td>3.7 Utilities</td>
<td>34</td>
</tr>
<tr>
<td>3.8 Visualizations</td>
<td>38</td>
</tr>
<tr>
<td>3.9 Using PyTuning Interactively</td>
<td>42</td>
</tr>
<tr>
<td>Index</td>
<td>55</td>
</tr>
</tbody>
</table>
PyTuning is a Python library intended for the exploration of musical scales and microtonalities. It can be used by developers who need ways of calculating, analyzing, and manipulating musical scales, but it can also be used interactively.

It makes heavy use of the SymPy package, a pure-Python computer algebra system, which allows scales and scale degrees to be manipulated symbolically, with no loss of precision. There is also an optional dependency on Matplotlib (and Seaborn) for some visualizations that have been included in the package.

Some of the package’s features include:

• Creation of scales in a variety of ways (EDO, Euler-Fokker, Diatonic, Harmonic, from generator intervals, etc.)
• Ability to represent created scales in ways that are understood by external software (Scala, Timidity, Fluidsynth, Yoshimi, Zynaddsubfx).
• Some analysis functions (for example, PyTuning provides a framework for searching for scale modes based upon defined metric functions and combinatorial analysis). Also included are some number-theoretic functions, such as prime limits and odd limits.
• Some scale visualizations.
• Interactive use.

As a simple example, to create a 31-TET scale and then create a tuning table for the timidity soft-synth:

```python
scale = create_edo_scale(31)
tuning_table = create_timidity_tuning(scale, reference_note=69)
```

The design of PyTuning is purposefully simple so that non-computer professionals can use it without much difficulty (musicians, musicologist, interested people of all stripes).

In scope this project is similar to the Scala software package, with a few differences:

• Scala is a mature, full-featured package that includes many, many scales and functions for manipulating and analyzing those scales. This project is much newer and less mature; its scope is currently much less (but hopefully it will be easy to extend).
• PyTuning is written in Python and relies on modern, well maintained dependencies. Scala is written in Ada, and while this is an interesting choice, it probably limits the population of users who could change or extend it should a need arise.
• Scala is mainly an application. PyTuning is a development library, but with ways for non-programmers to use it interactively.
• This package does not interact with sound cards or audio drivers, so one can’t play a scale directly. There are, however, functions for exporting scales into other software packages so that music and sound can be produced.
CHAPTER 1

Installation

PyTuning runs under Python 2.7.X and 3.X.

The easiest way to install PyTuning is via the Python Package Index, with which Pytuning is registered:

```
pip install pytuning
```

There are two hard dependencies for PyTuning: SymPy and NumPy. SymPy is a pure Python library and pip will handle its installation nicely. NumPy is a more complicated package and if installed via pip may involve much compilation; it would probably behoove you to install the package manually via whatever mechanism your platform provides before pip installing the package.

If you are running the package interactively it is recommended that the Jupyter interactive shell be installed. This is discussed in the documentation under the notes on Interactive use.

The source-code is available on GitHub, where it can be cloned and installed.
CHAPTER 2

Documentation

Documentation for the package can be found on Read the Docs.
CHAPTER 3

Roadmap

More scales, more visualizations, more analysis functions. Pull requests are welcome!

Basic Concepts

PyTuning is purposefully designed to be as simple as possible, so that non-programmers (musicologists, musicians, etc.) can use it without too much difficulty: the data structures are relatively simple; there are currently no classes defined, instead opting for an imperative/procedural approach.

Regardless of how it is used (interactively or as a development library), the user should have a good understanding of some of the basic, foundational concepts the package uses.

Scales

A scale is, simply, a list of degrees. By convention the list is bookended by the unison and the octave, with each degree given as a frequency ratio relative to the root tone of the scale. The first degree is always 1, the ratio of the first degree to the first degree. Most commonly the last degree is 2, as the octave is usually twice the frequency of the root (although the package has some support for non-standard “octaves”).

As an example, this is the standard 12-tone equal temperament scale (sometimes referred to as 12-TET, or 12-EDO, for Equal Division of the Octave).

\[
[1, \sqrt[12]{2}, \sqrt[12]{2}, \sqrt[12]{2}, 2^{\frac{5}{12}}, \sqrt[12]{2}, 2^{\frac{7}{12}}, 2^{\frac{5}{12}}, 2^{\frac{5}{12}}, 2^{\frac{5}{12}}, 2^{\frac{5}{12}}, 2^{\frac{5}{12}}, 2^{\frac{5}{12}}]
\]

A few things to note:

- As mentioned previously, the scale includes the unison and octave.
- Each scale degree is a SymPy number, so it is represented symbolically. Note that algebraic simplifications are performed by default.
- Even though the length of this list of 13, it is considered a 12 note scale, because the unison and the octave is in actuality the same note. Many of the functions in this package ask one to choose the number of notes or degrees to be used in this function. For these you should follow this convention.
(For those who are curious: the generation of scales is documented in Scale Creation, but the above scale was generated with the following code:

```python
from pytuning.scales import create_edo_scale
edo_12_scale = create_edo_scale(12)
```

Simplified versions of most functions are provided in the interactive environment.)

## Degrees

Scale degrees (which are expressed as frequency ratios relative to the tonic of the scale) are expressed in SymPy values. In practical terms the `Integer` and `Rational` class will be used the most, but SymPy is a full-featured package, and you may benefit from having some familiarity with it.

An example of a few degrees:

```python
import sympy as sp

unison = sp.Integer(1)  # Normal unison
octave = sp.Integer(2)  # Normal octave
perfect_fifth = sp.Rational(3,2)  # As a rational number
minor_second_tet = sp.Integer(2) ** sp.Rational(1,12)  # the 12th root of 2
# could also by sp.root(2,12)
lucy_L = sp.root(2,2*sp.pi)  # Lucy scale Long step
```

Will yield the following:

\[
1 \\
2 \\
3 \\
\sqrt{2} \\
\frac{3}{2} \\
2^{\frac{\pi}{2}}
\]

SymPy manipulates all values analytically, but sometimes one needs a floating approximation to a degree (for example, tuning a synthesizer usually needs frequencies expressed as floating point numbers). For the `evalf()` member function can be used:

```python
print(unison.evalf())
1.0000000000000
print(octave.evalf())
2.0000000000000
print(perfect_fifth.evalf())
1.5000000000000
print(minor_second_tet.evalf())
1.0594630943593
print(lucy_L.evalf())
1.1166328800911
```

## Modes

A **mode** is a selection of notes from a scale, and is itself a list of degrees (and therefore is also a scale). A mode can be produced from a scale by applying a **mask** to the scale. Again, the functions involved are documented elsewhere,
but as example this is how we would produce the standard major scale (which in the context of this package would be referred to as a mode):

```python
major_mask = (0,2,4,5,7,9,11,12)
major_mode = mask_scale(edo_12_scale, major_mask)
```

which produces the following scale:

\[ \left[ 1, \sqrt[3]{2}, \sqrt[3]{2}, 2^{\frac{5}{12}}, 2^{\frac{7}{12}}, 2^{\frac{3}{4}}, 2^{\frac{11}{12}}, 2 \right] \]

**Mode Objects**

Some functions in this package return a **mode object**. For example, the `find_best_modes()` function will take a scale and find a mode (or modes), based upon some consonance metric function. Here is one such object, which is implemented as a Python dict.

```python
{'mask': (0, 2, 3, 5, 7, 9, 10, 12),
'metric_3': 22.1402597402597,
'original_scale': [1,
256/243,
9/8,
32/27,
81/64,
4/3,
1024/729,
3/2,
128/81,
27/16,
16/9,
243/128,
2],
'scale': [1, 9/8, 32/27, 4/3, 3/2, 27/16, 16/9, 2],
'steps': [2, 1, 2, 2, 2, 1, 2],
'sum_distinct_intervals': 12,
'sum_p_q': 161,
'sum_p_q_for_all_intervals': 4374,
'sum_q_for_all_intervals': 1822}
```

The meaning of these keys:

- **original_scale** is the original scale which was input into the function. In this example is was a Pythagorean scale.
- **scale** is the output scale of the function
- **mask** is the mask of the original scale that produces the output
- **steps** is similar to mask, but reported in a different format. Each entry in the steps list represents the number of degrees in the original scale between successive degrees in the returned scale. The standard major scale, for example, would be represented by \([2, 2, 1, 2, 2, 1, 2] \).

In this example there are also other keys included. **sum_distinct_intervals**, **sum_p_q**, **sum_p_q_for_all_intervals**, **sum_q_for_all_intervals**, and **metric_3** are the outputs of calculated metric functions. This particular mode, for example, has a rating of 161 by the **sum_p_q** metric.

Metric functions are describe briefly below, and in more detail in *Metric Functions*.
Tuning Tables

*Tuning Tables* are a representation of a scale, usually a string (which can be written to a file), which can be understood by an external software package. As an example, to take a standard Pythagorean scale and produce a representation understood by Scala:

```python
pythag_scale = create_pythagorean_scale()
scala_tuning_table = create_scala_tuning(pythag_scale, "Pythagorean Scale")
```

The variable `scala_tuning_table` now contains the following:

```
! Scale produced by pytuning. For tuning yoshimi or zynaddsubfx,
! only include the portion below the final '!
!
Pythagorean Scale
  12
  !
  256/243
  9/8
  32/27
  81/64
  4/3
  1024/729
  3/2
  128/81
  27/16
  16/9
  243/128
  2/1
```

For many tuning tables one has to pin the scale to some reference frequency. For this the convention of MIDI note number is employed. For example, in the MIDI standard the note 69 is A 440 Hz, so by specifying a reference of 69, the corresponding entry in the table would be 400 Hz, and this would represent the root or tonic degree of the scale.

Exporting the above scale in a Csound compatible format:

```python
csound_tuning_table = create_csound_tuning(pythag_scale, reference_note=69)
```

yields the following:

```
f1 0 256 -2
  → 11.60156
  → 18.33333
  → 28.97119
  → 46.40625
  → 73.33333
  → 115.88477
  → 185.62500
  → 293.33333
  → 463.53909
    0.256  8.14815  8.70117  9.16667  9.65706  10.31250  10.86420
    → 12.22222
    → 19.31413
    → 20.62500
    → 30.93750
    → 32.59259
    → 48.88889
    → 52.20703
    → 77.25652
    → 123.75000
    → 130.37037
    → 195.55556
    → 208.82813
    → 309.02606
    → 495.00000
```

Chapter 3. Roadmap
This is a 128-entry table, mapping note number to absolute frequency. Csoud's table opcode can be used to index into the table and play the appropriate frequency, using something like the following:

```
inote init p4
iveloc init p5
ifreq table inote, 1
a1 oscil iveloc, ifreq, 2
outs a1, a1
```

(This assumes that p4 in the orchestra file contains MIDI note numbers, of course. If you use a different convention there are translation opcodes that can be used.)

## Metric Functions

**Metric Functions** are functions that takes a scale as an input and returns a numeric value calculated from that scale. It is used, for example, in `find_best_modes()` to evaluate the consonance of a scale (`find_best_modes()` uses a metric to evaluate the consonance of all possible modes of a scale and returns the evaluation of those modes as a `mode_object`).

The return value of a metric function should be a `dict` with a unique string identifier as the key and the metric as the value.

As an example, the following is one of the package-defined metrics:

```python
pytuning.metrics.sum_p_q_for_all_intervals(scale)
```

This metric is an estimate of scale consonance. It is formed by examining all unique intervals in the scale, and creating a numeric value based upon the summation of the numerators and denominators for all those intervals.

While the metric is numerically defined for ratios expressed as irrational or transcendental numbers, it is really only meaningful for scales with just degrees (ratios expressed as rational numbers).

Smaller values are more consonant.

As an example of use, the following:

```python
pythag = create_pythagorean_scale()
meter = sum_p_q_for_all_intervals(pythag)
```
yields the following:

```python
{'sum_p_q_for_all_intervals': 1092732}
```

## Scale Creation

There are several scale-creation functions in the package. They are found in `pytuning.scales` and can be imported into the program’s namespace with

```python
from pytuning.scales import *
```

(Note that for interactive use these are imported by default).

### The Harmonic Scale

We’ll start with the harmonic scale; it will illustrate many of the concepts used in scale creation.

There are two important concepts to understand:

- Normalization: If a scale is normalized (which is the default in all cases), then the intervals of the scale are normalized to fall within a single octave. This means scaling the interval either up or down the number of octaves needed to make the interval fall between the unison and the octave.
- The Octave: Normally an octave is defined as a doubling of frequency (2), but it is possible to define an octave by some other number. If this is the case the normalization will takes place over this new octave.

The function to create a harmonic scale is, `create_harmonic_scale`:

```python
pytuning.scales.create_harmonic_scale(first_harmonic, last_harmonic, normalize=True, octave=2)
```

Create a harmonic scale

**Parameters**

- `first_harmonic` – The first harmonic
- `last_harmonic` – The last harmonic
- `normalize` – If true, normalize the scale to an octave (2/1 by default, otherwise taken from `octave`)
- `octave` – The definition of the formal octave.

**Returns**

The scale

As an example of use, a normalized scale constructed from harmonics 3 to 20:

```python
scale = create_harmonic_scale(3, 20)
```

which yields:

```
[1, 13, 7, 5, 4, 17, 3, 19, 5, 11, 2]
```

To create a non-normalized scale:

```python
scale = create_harmonic_scale(3, 10, normalize=False)
```
which yields:

\[
\left[ 1, 4, 5, 2, 7, 8, 3, 10 \right]
\]

As an example, if we create a non-normalized harmonic scale of 10 harmonics:

```python
harmonic_scale = create_harmonic_scale(1, 10, normalize=False)
```

We have the following scale:

\[ [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] \]

If we normalize it each interval is scaled by a power of two to fall within 1 and 2. So, for example, the 9 becomes \(\frac{9}{8}\), because the nine must be scaled by three octaves to fall within that range:

\[ \frac{9}{8} = \frac{9}{2^3} \]

So the normalized scale is:

\[
\left[ 1, 9, 5, 3, 7, 2 \right]
\]

But if we change our octave definition to be 3, we normalize on powers of 3:

```python
harmonic_scale = create_harmonic_scale(1, 10, octave=3)
```

yields:

\[
\left[ 1, \frac{10}{9}, 4, 5, 2, \frac{7}{3}, \frac{8}{3}, 3 \right]
\]

**Equal Division of the Octave (Equal Temperament)**

Equal temperament scales can be created with the `create_edo_scale()` function. Note that this function does not accept a `normalize` argument, because EDO scales are normalized by definition. If does, however, allow you to change the definition of the formal octave.

```python
pytuning.scales.create_edo_scale(number_tones, octave=2)
```

Create an equal division of octave (EDO, ET) scale.

**Parameters**

- `number_tones` - The number of tones/divisions in the scale
- `octave` - The formal octave (frequency ratio)

**Example, 12T-ET:**

```python
edo_scale = create_edo_scale(12)
```

will yield the normal equal-tempered scale used in western music:

\[
\left[ 1, \sqrt[12]{2}, \sqrt[12]{2}, \sqrt[12]{2}, 2^{\frac{2}{12}}, \sqrt[12]{2}, 2^{\frac{4}{12}}, 2^{\frac{5}{12}}, 2^{\frac{6}{12}}, 2^{\frac{7}{12}} \right]
\]

Note that the length of the scale is 13, as both the unison and octave are included by convention.

It is also possible to have a non-2 formal octave. The code:
edo_scale = create_edo_scale(12,3)

will yield:

\[
\begin{bmatrix}
1, \sqrt[3]{3}, \sqrt[3]{3}, \sqrt[3]{3}, 3\frac{2}{3}, \sqrt[3]{3}, 3\frac{2}{3}, 3\frac{2}{3}, 3\frac{2}{3}, 3\frac{2}{3}, 3\frac{2}{3}, 3\frac{2}{3}
\end{bmatrix}
\]

Scales from a Generator Interval

The `create_equal_interval_scale()` function will generate a scale from a generator interval. This is the base function for several other scale types (for example, the Pythagorean scale is created with a generator interval of \(\frac{3}{2}\)).

In the creation of a scale, the generator interval can either be used directly (for, for example, making each successive tone a generator interval above the previous tone), or in an inverted sense (making each interval a generator down from the previous). This function starts from the unison and walks down the number specified, walking up for the rest of the intervals.

```python
pytuning.scales.create_equal_interval_scale(generator_interval, scale_size=12, number_down_intervals=6, epsilon=None, sort=True, octave=2, remove_duplicates=True, normalize=True)
```

Create a scale with equal-interval tuning

**Parameters**

- `generator_interval` – The interval to use for generation (sympy value)
- `scale_size` – The number of degrees in the scale
- `number_down_intervals` – The number of inverted intervals to use in scale construction.
- `epsilon` – Rounding parameter. If set to `None` no rounding is done. Otherwise the scale degrees are rounded to the nearest epsilon
- `sort` – If `True`, sort the output by degree size
- `octave` – The formal octave
- `remove_duplicates` – If `True` remove duplicate entries
- `normalize` – IF `True`, normalize the degrees to the octave

In general one should keep epsilon at `None` and perform and rounding outside the function.

This is a base function from which several other scales are derived, including:

- **The Pythagorean scale** A scale with a perfect fifth \((3/2)\) as the generating interval

  \[
  P_5 = \frac{3}{2}
  \]

- **The quarter-comma meantone scale** A scale in which the generating interval is a perfect fifth narrowed by one quarter of syntonic comma

  \[
  P_5 = \frac{3}{2} \sqrt[4]{\frac{81}{80}}
  \]
• EDO Scales EDO scales can be generated from an appropriate selection of the fifth. For example, the 12-TET scale would use the fifth:

\[ P_5 = \sqrt[12]{2} \]

The Pythagorean Scale

This is the standard Pythagorean scale. Note that we can choose the number of up and down intervals in the scale. The default yields the standard scale, with the fourth degree as a diminished fifth, as opposed to the augmented fourth.

```python
pytuning.scales.create_pythagorean_scale(scale_size=12, number_down_fifths=6, epsilon=None, sort=True, octave=2, remove_duplicates=True)
```

Create a Pythagorean scale

Parameters

- `scale_size` – The number of degrees in the scale
- `number_down_fifths` – The number of inverted fifths to use in scale construction.
- `epsilon` – Rounding parameter. If set to `None` no rounding is done. Otherwise the scale degrees are rounded to the nearest epsilon
- `sort` – If `True`, sort the output by degree size
- `octave` – The formal octave
- `remove_duplicates` – If `True` remove duplicate entries

The Pythagorean scale is an even-interval scale with the following generating interval:

\[ P_5 = \frac{3}{2} \]

So, for the standard scale we can use:

```python
scale = create_pythagorean_scale()
```

yielding:

\[
\begin{bmatrix}
1, & 256, & 9, & 32, & 81, & 4, & 1024, & 3, & 128, & 27, & 16, & 9, & 243, & 128, & 2
\end{bmatrix}
\]

If we wanted the augmented fourth:

```python
scale = create_pythagorean_scale(number_down_fifths=5)
```

yielding:

\[
\begin{bmatrix}
1, & 256, & 9, & 32, & 81, & 4, & 729, & 3, & 128, & 27, & 16, & 9, & 243, & 128, & 2
\end{bmatrix}
\]

The Quarter-Comma Meantone Scale

```python
pytuning.scales.create_quarter_comma_meantone_scale(scale_size=12, number_down_fifths=6, epsilon=None, sort=True, octave=2, remove_duplicates=True)
```

Create a quarter-comma meantone scale
Parameters

- `scale_size` – The number of degrees in the scale
- `number_down_fifths` – The number of inverted fifths to use in scale construction.
- `epsilon` – Rounding parameter. If set to `None` no rounding is done. Otherwise the scale degrees are rounded to the nearest epsilon
- `sort` – If `True`, sort the output by degree size
- `octave` – The formal octave
- `remove_duplicates` – If `True` remove duplicate entries

The quarter-comma meantone scale is an even-interval scale with the following generating interval:

\[ P_5 = \frac{\frac{3}{2}}{\sqrt[4]{\frac{81}{80}}} \]

which is a perfect fifth (in a Pythagorean sense) narrowed by one quarter of the syntonic comma.

An example of use:

```python
close = create_quarter_comma_meantone_scale()
```
yields:

\[
\begin{align*}
1, & \frac{8}{25}, \frac{\sqrt{5}}{2}, \frac{4\sqrt{5}}{5}, \frac{5}{4}, \frac{2\sqrt{5}}{25}, \frac{16\sqrt{5}}{15}, \frac{\sqrt{5}}{2}, \frac{4\sqrt{5}}{5}, \frac{5\sqrt{5}}{4}, 2
\end{align*}
\]

Euler-Fokker Genera

`pytuning.scales.create_euler_fokker_scale`(*intervals*, *multiplicities*, *octave=2*, *normalize=True*)

Create a scale in the Euler-Fokker Genera

Parameters

- `intervals` – The factors to use for the construction (usually prime numbers)
- `multiplicities` – The multiplicities of the factors (see below)
- `octave` – The formal octave
- `normalize` – If `True`, normalize the intervals to the octave.

`intervals` and `multiplicities` should both be lists of equal length. The entries in `multiplicities` give the number of each factor to use. Therefore the following:

```python
intervals = [3,5,7]
multiplicities = [1,1,1]
scale = create_euler_fokker_scale(intervals, multiplicities)
```

Will create a scale with one 3, one 5, and one 7 as generators.

The above will produce the following scale:

\[
\begin{bmatrix}
1, & 35, & 5, & 21, & 3, & 105, & 7, & 15, & 2
\end{bmatrix}
\]

Also note that the two statements will generate the same output:
intervals = [3, 5, 7]
multiplicities = [2, 2, 1]
scale1 = create_euler_fokker_scale(intervals, multiplicities)

intervals = [3, 3, 5, 5, 7]
multiplicities = [1, 1, 1, 1, 1]
scale2 = create_euler_fokker_scale(intervals, multiplicities)
scale1 == scale2
True

Diatonic Scales

pytuning.scales.create_diatonic_scale(generators, specification)
Create a diatonic scale.

Parameters

• generators – The generator intervals (see below)
• specification – The scale specification. This is a list of chars that correspond to entries in the generators. Note that if all the character representations are a single character, you can pass the specification in as a string for convenience.

Returns The specified scale

generators is a list of tuples, the first member of which is an interval specification, the second of which is a character representation. The entries in specification should correspond to this value.

As an example, we can create the 12 EDO generators thus:

edo12_constructors = [
    (sp.power.Pow(2, sp.Rational(2, 12)), "T"),
    (sp.power.Pow(2, sp.Rational(1, 12)), "s"),
]

We can then create the standard major mode with:

create_diatonic_scale(edo12_constructors, ["T","T","s","T","T","T","s"])

which will yield:

\[
\begin{bmatrix}
1, & \sqrt{2}, & \sqrt{2}, & 2^{\frac{1}{3}}, & 2^{\frac{5}{9}}, & 2^{\frac{1}{3}}, & 2^{\frac{7}{12}}, & 2
\end{bmatrix}
\]

As another example of creating a diatonic scale, we can use the five-limit constructors (which are defined in pytuning.constants):

five_limit_constructors = [
    (sp.Rational(16, 15), "s"),
    (sp.Rational(10, 9), "t"),
    (sp.Rational(9, 8), "T"),
]

to create Ptolemy’s Intense Diatonic Scale:
from pytuning.constants import five_limit_constructors
from pytuning.scales import create_diatonic_scale

scale = create_diatonic_scale(five_limit_constructors, ['T', 't', 's', 'T', 't', 'T', 's'])

which gives us:

\[
\begin{bmatrix}
1 & 9 \\
5 & 4 \\
8 & 3 \\
3 & 2 \\
5 & 3 \\
15 & 8 \\
2 & \\
\end{bmatrix}
\]

Note that if every identifier is a single-character string, specification can also be passed in as a string. So this is equivalent:

from pytuning.constants import five_limit_constructors
from pytuning.scales import create_diatonic_scale

scale = create_diatonic_scale(five_limit_constructors, "TtTsTsT")

## Metric Functions

A metric function is a function that takes a scale as input and returns a calculated value. As mentioned in Basic Concepts, it returns a Python dict with the metric name as the key, and the metric value as the value.

The currently defined metrics all estimate the consonance or dissonance of a scale.

**sum_p_q()**

pytuning.metrics.sum_p_q(scale)

Calculate a metric for a scale

**Parameters**

scale – The scale.

**Returns**

A dict with the metric value.

This is an estimate of scale consonance. It is derived from summing the numerators and denominators of the scale degrees.

Smaller values are more consonant.

Note that this metric looks at the degrees of the scale, so it is somewhat tonic-focused. The similar metric sum_p_q_for_all_intervals() is similar, but it sums the numerator and denominator values for all distinct intervals within the scale.

While the metric is numerically defined for ratios expressed as irrational or transcendental numbers, it is really only meaningful for scales with just degrees (ratios expressed as rational numbers).

sum_p_q(create_pythagorean_scale())

yields:

\[
\{'sum_p_q': 3138\}
\]
**sum_distinct_intervals()**

`pytuning.metrics.sum_distinct_intervals(scale)`  
Calculate a metric for a scale

**Parameters**  
`scale` – The scale.

**Returns**  
A `dict` with the metric value.

This metric is an estimate of scale consonance. Numerically it is the number of distinct intervals within the scale (including all ratios and their inversions).

Smaller values are more consonant.

```python
gives:

{'sum_distinct_intervals': 22}
```

**metric_3()**

`pytuning.metrics.metric_3(scale)`  
Calculate a metric for a scale

**Parameters**  
`scale` – The scale.

**Returns**  
A `dict` with the metric value.

Metric 3 is an estimate of scale consonance. Given a ratio p/q, it is a heuristic given by the following:

\[
m_3 = \sum \frac{1}{p/q} = \sum \frac{q}{p - q}
\]

(3.1)

Smaller values are more consonant.

The summation takes place over all of the intervals in the scale. It does not form a set of distinct intervals.

**sum_p_q_for_all_intervals()**

`pytuning.metrics.sum_p_q_for_all_intervals(scale)`  
Calculate a metric for a scale

**Parameters**  
`scale` – The scale (i.e., a list of `sympy.Rational` values)

**Returns**  
The metric.

This metric is an estimate of scale consonance. It is formed by examining all unique intervals in the scale, and creating a numeric value based upon the summation of the numerators and denominators for all those intervals.

While the metric is numerically defined for ratios expressed as irrational or transcendental numbers, it is really only meaningful for scales with just degrees (ratios expressed as rational numbers).

Smaller values are more consonant.
**sum_q_for_all_intervals()**

```python
pytuning.metrics.sum_q_for_all_intervals(scale)
```

Calculate a metric for a scale.

**Parameters**
- `scale`: The scale (i.e., a list of `Rational`s)

**Returns**
The metric.

Metric 5 is an estimate of scale consonance. It is summation of the denominators of the normalized distinct ratios of the scale.

Smaller values are more consonant.

**All Metrics**

There is also a function that calculates all defined metrics for a scale.

```python
pytuning.metrics.all_metrics(scale)
```

Calculate all metrics for the scale.

**Parameters**
- `scale`: The scale (i.e., a list of `Rational`s)

**Returns**
A `dict` containing all metrics.

As an example:

```python
pythag = create_pythagorean_scale()
metrics = all_metrics(pythag)
```

will (currently) produce:

```python
{
    'metric_3': 49.9049074891784,
    'sum_distinct_intervals': 22,
    'sum_p_q': 3138,
    'sum_p_q_for_all_intervals': 1092732,
    'sum_q_for_all_intervals': 452817
}
```

If new metrics are coded they should be added to the `__all__` data member for inclusion here.

**Scale Analysis and Creation (Redux)**

*Scale Creation* describes the way that many standard scales are generated from within the package. But there are other ways to create scales.

**Mode Selection**

When one creates a scale – for example, the Pythagorean scale or a scale of the Euler-Fokker Genera – one can looks at the various modes that can be created for that scale and evaluate them by certain criteria.

The `find_best_modes()` function can be used for this. This function accepts and input scale, the number of tones for the mode, and the optimization functions that should be used for evaluating the scale.
As an example, one scale that I've used in compositions is created from choosing a seven-note mode from a harmonic scale, optimized over the metric \texttt{sum_p_q_for_all_intervals()}. This particular scale is based upon the harmonic series referenced to the fourth harmonic.

The following code:

```python
def harmonic_scale = create_harmonic_scale(4, 30)
result = find_best_modes(harmonic_scale,
                              num_tones=7,
                              sort_order=['sum_p_q_for_all_intervals'],
                              num_scales=1,
                              metric_function = sum_p_q_for_all_intervals)
```

yields the following object:

```
\[
\begin{array}{c}
\text{mask}: (0, 2, 4, 5, 8, 12, 14, 15), \\
\text{original_scale}: [1, 17/16, 9/8, 19/16, 5/4, 21/16, 11/8, 23/16, 3/2, 25/16, 13/8, 27/16, 7/4, 29/16, 15/8, 2], \\
\text{scale}: [1, 9/8, 5/4, 21/16, 3/2, 7/4, 15/8, 2], \\
\text{steps}: [2, 2, 1, 3, 4, 2, 1], \\
\text{sum_p_q_for_all_intervals} = 572
\end{array}
\]
```

The returned scale:

\[
\begin{bmatrix}
1 & 9/8 & 5/4 & 21/16 & 3/2 & 7/4 & 15/8 & 2
\end{bmatrix}
\]

minimizes the metric for all possible combinations of 7 notes chosen from the original harmonic scale.

\texttt{pytuning.scale_creation.find_best_modes(scale, num_tones, sort_order=['sum_p_q_for_all_intervals', 'sum_p_q', 'sum_distinct_intervals'], num_scales=1, metric_function=None)}

Find the best modes for a scale, as defined by the specified metrics.

**Parameters**

- **scale** – The scale to analyze
- **num_tones** – The number of degrees in the mode
- **sort_order** – How the return should be sorted, referenced to the metrics calculated
- **num_scales** – The number of scales to return. If None all scales will be returned
- **metric_function** – The metric function to use. If None then all_metrics will be used.
Returns A sorted list of mode objects.

The sort order is a list of keys that the metric function should return, applied in order, with an assumption that the lower the metric the more consonant (and “better”) the scale. As an example, the default sort order:

```python
['sum_p_q_for_all_intervals','sum_p_q','sum_distinct_intervals']
```

Will order the scales by increasing **sum_p_q_for_all_intervals**. If two scales have the same **sum_p_q** value they will be secondarily sorted on **sum_p_q**. If scales have the same **sum_p_q_for_all_intervals** and **sum_p_q** then **sum_distinct_intervals** will be used.

If no metric function is specified the default **all_metrics** will be used. However, for efficiency one may not want to calculate all metrics if they are not being used. For example, if one is just interested in one metric, you can pass the metric directly:

```python
from pytuning import create_pythagorean_scale
from pytuning.metrics import sum_p_q_for_all_intervals
pythag = create_pythagorean_scale()
my_metric = lambda scale: dict(sum_p_q(scale), **sum_p_q_for_all_intervals(scale))
best_modes = find_best_modes(pythag, 7, sort_order=['sum_p_q_for_all_intervals'],
                             num_scales=1, metric_function=sum_p_q_for_all_intervals)
```

which would yield:

```python
[{'mask': (0, 1, 3, 5, 6, 8, 10, 12),
 'original_scale': [1,
 256/243,
 9/8,
 32/27,
 81/64,
 4/3,
 1024/729,
 3/2,
 128/81,
 27/16,
 16/9,
 243/128,
 2],
 'scale': [1, 256/243, 32/27, 4/3, 1024/729, 128/81, 16/9, 2],
 'steps': [1, 2, 2, 1, 2, 2, 2],
 'sum_p_q_for_all_intervals': 4374}]
```

If one is interested in two of the metrics, you could, for example:

```python
from pytuning import create_pythagorean_scale
from pytuning.metrics import sum_p_q, sum_p_q_for_all_intervals
pythag = create_pythagorean_scale()
my_metric = lambda scale: dict(sum_p_q(scale), **sum_p_q_for_all_intervals(scale))
best_modes = find_best_modes(pythag, 7, sort_order=['sum_p_q','sum_p_q_for_all_intervals'],
                             num_scales=1, metric_function=my_metric)
```

which would yield:
Factoring an Interval

Sometimes it is interesting to take an interval and find an expression for that interval over some set of generator intervals. For this the function `find_factors()` is provided.

One has to specify the generator intervals. This is done by passing the function a list of tuples. Each tuple has two members: The generator interval, and a character representation of the generator interval. Usually these are a single, unique character (such as \( X \)), but it can also be in the form \( 1/X \). If it is in this form the generator interval should be the reciprocal of the interval designated by \( X \).

As an example, we could create a generator interval that represents the tone and semi-tone of a 31-EDO scale:

```python
edo31_constructors = [
    (sp.power.Pow(2,sp.Rational(2,31)), "T"),  # a tone
    (sp.power.Pow(2,sp.Rational(1,31)), "s"),   # a semitone
]
```

(Note that the tone is just twice the semitone, so we could probably get by with just defining the semitone).

Now we can define an interval, say, one of the intervals of the Pythagorean scale:

\[
\begin{align*}
\frac{21}{16}
\end{align*}
\]

and see what factoring yields an interval closest to the original.

```python
results = find_factors(interval, edo31_constructors)
results
```

`results` now contains the factoring, the factoring in symbolic terms, and the resultant interval.

\[
\begin{align*}
\{'T', 'T', 'T', 'T', 'T', 'T'\},
2**(12/31)]
\end{align*}
\]

The last entry is the returned interval:

\[
\begin{align*}
2^{12}
\end{align*}
\]
If one is interested in seeing how closely the factored interval matches the original interval, the `ratio_to_cents()` function in `pytuning.utilities` can be used.

```python
from pytuning.utilities import ratio_to_cents
print(ratio_to_cents(results[2] / interval))
```

yields:

```
-6.26477830225428
```

In other words, the derived interval is flat about 6.3 cents from the target interval.

**pytuning.utilities.ratio_to_cents(ratio)**

Convert a scale degree to a cent value

- **Parameters**
  - `ratio` – The scale degree (sympy value)

- **Returns**
  - The scale degree in cents

Calculates:

\[
2 \left[ \frac{\text{ratio}}{\text{degree}} \right] \sqrt[2]{\text{degree}}
\]

Note that this function returns a floating point number, not a sympy ratio.

**Approximating a Scale with Another Scale**

The above factoring of an interval over a set of generators can be extended: a scale can be factored too.

To do this the `create_scale_from_scale()` function is used.

The first step in using this function is to create an interval function. It is similar to `find_factors()` in that it accepts an interval and a max factor, and it returns the factor. But the actual generator intervals are bound to this function.

The easiest way of creating this function is to take the generator intervals that you’re interested in and to bind them to `find_factors()` via a partial function application. As an example, we can take the five-limit constructors:

```python
five_limit_constructors = [
    (sp.Rational(16,15), "s"),
    (sp.Rational(10,9), "t"),
    (sp.Rational(9,8), "T"),
]
```

And use them to approximate the Pythagorean scale:

```python
from pytuning.scales import create_pythagorean_scale
from pytuning.scale_creation import create_scale_from_scale, find_factors
from pytuning.constants import five_limit_constructors
from functools import partial

interval_function = partial(find_factors, constructors=five_limit_constructors)
pythag = create_pythagorean_scale()
results = create_scale_from_scale(pythag, interval_function)
```

The return value is a tuple, the first element of which is derived scale, the second of which is the symbolic factoring. The scale which was found was:

\[
\begin{bmatrix}
1, & 16, & 9, & 32, & 81, & 4, & 1024, & 3, & 128, & 27, & 16, & 243, & 128, & 2
\end{bmatrix}
\]
If you look at the Pythagorean scale:

\[
\begin{bmatrix}
1, & 256 & 9 & 81 & 32 & 4 & 1024 & 2^2 & 3 & 128 & 27 & 16 & 9 & 243 & 128 & 2
\end{bmatrix}
\]

you can see that they only differ in the second degree (which if we look at the first member of the return we can see is factored as ['s']). Looking at how much they differ:

```python
ratio = results[0][1] / pythag[1]
print(ratio)
81/80
print(ratio_to_name(ratio))
Syntonic Comma
delta = ratio_to_cents(ratio)
print(delta)
21.5062895967149
```

we see that the difference is $\frac{81}{80}$, which is the syntonic comma (about 21.5 cents).

`create_scale_from_scale()` can also accept a `tone_table` which is a list of the potential breakdowns that can be used in the analysis.

```python
pytuning.scale_creation.create_scale_from_scale(scale, interval_function, max_terms=8, tone_table=None)
```

Given a target scale, calculate the closest matching N-rank linear temperament scale over the provide basis functions.

**Parameters**

- `scale` – The target scale (list or ratios)
- `interval_function` – The interval function (see below)
- `max_terms` – The maximum number of terms for the factoring along the basis intervals
- `tone_table` – A constrained tone table (see below)

**Returns** A tuple, the first member of which is a list of the derived scale values, and the second of which is a symbolic representation of the factoring found.

The interval function is a function that accepts a degree and `max_terms` and returns the breakdown in the same format as `find_factors`. In general the easiest way to create this function is thought a partial function application of `find_factors` and the specific constructors wanted. As an example, to create a function that will approximate a scale with the five-limit constructors:

```python
from pytuning.scale_creation import find_factors, create_scale_from_scale
from pytuning.constants import five_limit_constructors
import functools

find_five_limit_interval = functools.partial(find_factors, constructors=five_limit_constructors, max_terms=15)
create_five_limit_scale_from_scale = functools.partial(create_scale_from_scale, interval_function=find_five_limit_interval)
```

if `tone_table` is None, the code will perform the factoring with up to `max_terms`. If the tone table is defined only the intervals defined in this table will be used.

The tone table is formatted as a list of tuples, where the members of each tuple are the degree name (String), the interval composition (a list of characters taken from the symbolic portion of the constructors), and the value of that factoring. As an example, the tone table that matches published values for the Lucy-tuned scale begins:
(The last member of the tuple is a `sympy` symbolic value.)

With the tone table, the code will return the defined tone which most closely matches the target degree.

Extending the above example, if we were to try to match a Pythagorean scale with an unconstrained factoring of the Five-limit intervals (i.e., with no tone table):

```python
pythag = create_pythagorean_scale()
lp = create_five_limit_scale_from_scale(pythag)
```

yields

```
[(1,
  16/15,
  9/8,
  32/27,
  81/64,
  4/3,
  1024/729,
  3/2,
  128/81,
  27/16,
  16/9,
  243/128,
  2],
[[],
 ['s'],
 ['T'],
 ['s', 't'],
 ['T', 'T'],
 ['s', 's', 't'],
 ['T', 's', 's', 't'],
 ['T', 'T', 's', 't'],
 ['T', 'T', 'T', 's', 't'],
 ['T', 'T', 'T', 's', 's', 't'],
 ['T', 'T', 'T', 'T', 's', 't'],
 ['T', 'T', 'T', 'T', 's', 's', 't', 't'])
```

Note that the first entry of the factors is always for the ratio 1, and is returned as an empty list (as there really are no factors in this sense).

## Tuning Tables

A **tuning table** is a text representation of a scale that can be interpreted by an external program. This is usually used to tune a synthesizer or other sound source so that the scale can be used in a musical composition.

Some tuning tables – such as that used by Scala – describe the scale in absolute terms, but most need to have a reference defined so that the scale degrees can be mapped to a frequency. For this purpose the PyTuning package has adopted
the MIDI standard for note numbers and frequencies.

<table>
<thead>
<tr>
<th>Octave</th>
<th>C</th>
<th>C#</th>
<th>D</th>
<th>D#</th>
<th>E</th>
<th>F</th>
<th>F#</th>
<th>G</th>
<th>G#</th>
<th>A</th>
<th>A#</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>-2</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>-1</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
</tr>
<tr>
<td>0</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
<td>32</td>
<td>33</td>
<td>34</td>
<td>35</td>
</tr>
<tr>
<td>1</td>
<td>36</td>
<td>37</td>
<td>38</td>
<td>39</td>
<td>40</td>
<td>41</td>
<td>42</td>
<td>43</td>
<td>44</td>
<td>45</td>
<td>46</td>
<td>47</td>
</tr>
<tr>
<td>2</td>
<td>48</td>
<td>49</td>
<td>50</td>
<td>51</td>
<td>52</td>
<td>53</td>
<td>54</td>
<td>55</td>
<td>56</td>
<td>57</td>
<td>58</td>
<td>59</td>
</tr>
<tr>
<td>3</td>
<td>60</td>
<td>61</td>
<td>62</td>
<td>63</td>
<td>64</td>
<td>65</td>
<td>66</td>
<td>67</td>
<td>68</td>
<td>69</td>
<td>70</td>
<td>71</td>
</tr>
<tr>
<td>4</td>
<td>72</td>
<td>73</td>
<td>74</td>
<td>75</td>
<td>76</td>
<td>77</td>
<td>78</td>
<td>79</td>
<td>80</td>
<td>81</td>
<td>82</td>
<td>83</td>
</tr>
<tr>
<td>5</td>
<td>84</td>
<td>85</td>
<td>86</td>
<td>87</td>
<td>88</td>
<td>89</td>
<td>90</td>
<td>91</td>
<td>92</td>
<td>93</td>
<td>94</td>
<td>95</td>
</tr>
<tr>
<td>6</td>
<td>96</td>
<td>97</td>
<td>98</td>
<td>99</td>
<td>100</td>
<td>101</td>
<td>102</td>
<td>103</td>
<td>104</td>
<td>105</td>
<td>106</td>
<td>107</td>
</tr>
<tr>
<td>7</td>
<td>108</td>
<td>109</td>
<td>110</td>
<td>111</td>
<td>112</td>
<td>113</td>
<td>114</td>
<td>115</td>
<td>116</td>
<td>117</td>
<td>118</td>
<td>119</td>
</tr>
<tr>
<td>8</td>
<td>120</td>
<td>121</td>
<td>122</td>
<td>123</td>
<td>124</td>
<td>125</td>
<td>126</td>
<td>127</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

So, for example, note number 69 corresponds to middle A, whereas 60 corresponds to middle C. The frequency standard is 12-EDO, and the note 69 is pegged to 440 Hz. Thus if you passed 69 as the reference note, the 69th entry in the table would be 440 Hz and this would correspond to the first degree of the scale. 60 would cause the first degree of the scale to be assigned to that note number (with a corresponding frequency of about 261.6 Hz). The timidity soft synth is an example of a synthesizer that needs this reference note and frequency.

In general the table will need to be output to disk so that it can read by the program. This can be done with something like:

```python
from pytuning.tuning_tables import create_timidity_tuning
from pytuning.scales import create_euler_fokker_scale

reference_note = 60
scale = create_euler_fokker_scale([3,5],[3,1])

tuning = create_timidity_tuning(scale, reference_note=reference_note)

with open("timidity.table", "w") as table:
    table.write(tuning)
```

This will cause the generated scale:

\[
\begin{bmatrix}
1, & 135, & 9, & 5, & 45, & 3, & 27, & 15, & 2
\end{bmatrix}
\]

to be written to a disk file, `timidity.table`, which can be understood by timidity:

```
timidity -2 timidity.table score.mid
```

**Timidity**

`pytuning.tuning_tables.create_timidity_tuning` creates a Timidity++ tuning table.

```python
pytuning.tuning_tables.create_timidity_tuning(scale, reference_note=60, reference_frequency=None)
```

Create a Timidity++ tuning table

**Parameters**

- `scale` – The scale to model (list of frequency ratios)
- `reference_note` – The MIDI number of the absolute frequency reference
- `reference_frequency` – The frequency of the reference note. If `None` (the default) the frequency will be taken from the standard MIDI 12-EDO tuning

3.5. Tuning Tables 27
Returns A Timidity tuning table as a String

The default value of reference_note pegs the scale to the standard concert tuning of middle C (A = 440Hz).

The Timidity table is basically a list of integers for all defined MIDI note numbers, with each entry as 1000 times the note frequency.

As a somewhat detailed example, let’s say that the user had a 12-EDO scale constructed, and wanted to pin the tonic note to the standard A440. The following will do this:

```python
from pytuning.scales import create_edo_scale
from pytuning.tuning_tables import create_timidity_tuning

scale = create_edo_scale(12)
tuning_table = create_timidity_tuning(scale, reference_note=69)
```

with the first part of the table given by:

```
# Timidity tuning table created by pytuning,
# call timidity with the -Z option to enable.
# Note reference: 69; Freq reference: 440.000000 Hz
8176
8662
9177
9723
10301
10913
11562
12250
12978
```

To use the table, one starts Timidity with the -Z option, i.e:

```
timidity -Z table.name -iA
```

Scala

The Scala tuning table can be used with the Scala package, but it can also be used to tune the soft synth Zynaddsubfx, as well as its derivative Youshimi. With the soft synths you will need to explicitly set the reference note and frequency in the scale GUI.

`pytuning.tuning_tables.create_scala_tuning(scale, name)`

Create a Scala scale file

Parameters

- `scale` – The scale (list of frequency ratios)
- `name` – The name of the scale

Returns A Scala file as a String

The Scala file can be used to tune various things, most germane being Yoshimi. However, keep in mind that the Scala file does not include a base note or frequency, so for tuning purposes those data will need to be captured or input in some other way.

As an example of use, the Scala file for the default Pythagorean tuning can be calculated thus:
from pytuning.scales.pythagorean import create_pythagorean_scale
from pytuning.tuning_tables import create_scala_tuning

scale = create_pythagorean_scale()
table = create_scala_tuning(scale, "Pythagorean Tuning")

which yields:

! Scale produced by pytuning. For tuning yoshimi or zynaddsubfx,
! only include the portion below the final '!

Pythagorean Tuning
12
!
256/243
9/8
32/27
81/64
4/3
1024/729
3/2
128/81
27/16
16/9
243/128
2/1

Note that the Scala file uses exact ratios where possible, otherwise it will convert to a cent value. Thus the code:

from pytuning.scales import create_edo_scale
from pytuning.tuning_tables import create_scala_tuning

scale = create_edo_scale(12)
table = create_scala_tuning(scale, "12-TET Tuning")

will produce:

! Scale produced by pytuning. For tuning yoshimi or zynaddsubfx,
! only include the portion below the final '!

12-TET Tuning
12
!
100.00000
200.00000
300.00000
400.00000
500.00000
600.00000
700.00000
800.00000
900.00000
1000.00000
1100.00000
2/1
**Fluidsynth**

**pytuning.tuning_tables.create_fluidsynth_tuning** *(scale, reference_note=60, chan=[0], bank=0, prog=[0], reference_frequency=None)*

Create a Fluidsynth tuning table

**Parameters**

- **scale** – The scale to model (list of frequency ratios)
- **reference_note** – The MIDI number of the absolute frequency reference
- **reference_frequency** – The frequency of the reference note. If None (the default) the frequency will be taken from the standard MIDI 12-EDO tuning
- **chan** – A list of channels for which to create the table
- **bank** – The bank for the tuning table
- **prog** – A list of program numbers for the tuning table

**Returns** A Fluidsynth tuning table as a String

The default value of reference_note pegs the scale to the standard concert tuning of middle C (A = 440Hz).

The Fluidsynth tuning model allows each channel, bank, and program to have a different tuning. Thus, if one, say, wants all programs to be tuned to the scale, the tuning table can get quite large.

As a somewhat detailed example, let’s say that the user had a 12-EDO scale constructed, and wanted to pin the tonic note to the standard A440. The following will do this:

```python
from pytuning.scales import create_edo_scale
from pytuning.tuning_tables import create_fluidsynth_tuning

scale = create_edo_scale(12)
tuning_table = create_fluidsynth_tuning(scale, prog=range(128), reference_note=69)
```

with the first part of the table given by:

```plaintext
# Fluidsynth Tuning Table created by pytuning
# Note reference: 69; Freq reference: 440.000000 Hz
tuning000 0 0
  tune 0 0 0 0.000000
  tune 0 0 1 100.000000
  tune 0 0 2 200.000000
  tune 0 0 3 300.000000
  tune 0 0 4 400.000000
  tune 0 0 5 500.000000
  tune 0 0 6 600.000000
  tune 0 0 7 700.000000
  tune 0 0 8 800.000000
  tune 0 0 9 900.000000
```

To use the table, one should start fluidsynth with the -f option:

```bash
fluidsynth -f table.name
```
**Csound**

For use in Csoud PyTuning will generate a table of frequencies that can be used as a table lookup, mapped to MIDI note number. As mentioned in the basic concepts, the easiest way to use this is via the table opcode:

<table>
<thead>
<tr>
<th>inote</th>
<th>init</th>
<th>p4</th>
</tr>
</thead>
<tbody>
<tr>
<td>iveloc</td>
<td>init</td>
<td>p5</td>
</tr>
<tr>
<td>ifreq</td>
<td>table</td>
<td>inote, l</td>
</tr>
<tr>
<td>al</td>
<td>oscil</td>
<td>iveloc, ifreq, 2</td>
</tr>
<tr>
<td>outs</td>
<td></td>
<td>al, al</td>
</tr>
</tbody>
</table>

```python
pytuning.tuning_tables.create_csound_tuning(scale, reference_note=60, reference_frequency=None, table_num=1)
```

Create a CSound tuning table

**Parameters**

- `scale` – The scale (list of frequency ratios)
- `reference_note` – The MIDI number of the absolute frequency reference
- `reference_frequency` – The frequency of the reference note. If `None` (the default) the frequency will be taken from the standard MIDI 12-EDO tuning
- `table_num` – The table number to use

CSound has many ways of generating microtonalities. For pytuning a table lookup keyed on MIDI note number is used.

As an example of use, let’s say that we want to use the 12-EDO scale with the tonic at A440:

```python
from pytuning.scales import create_edo_scale
from pytuning.tuning_tables import create_csound_tuning

scale = create_edo_scale(12)
table = create_csound_tuning(scale, reference_note=69)
```

This will produce the following output, which can be included in the CSound score file:

<table>
<thead>
<tr>
<th>f1</th>
<th>0 256</th>
<th>-2</th>
<th>8.17580</th>
<th>8.66196</th>
<th>9.17702</th>
<th>9.72272</th>
<th>10.30086</th>
<th>10.91338</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>29.13524</td>
<td>30.86771</td>
<td>32.70320</td>
<td>34.64783</td>
<td>36.70810</td>
<td>38.89087</td>
<td>41.20344</td>
<td>43.67529</td>
</tr>
<tr>
<td></td>
<td>46.24930</td>
<td>48.99943</td>
<td>51.91309</td>
<td>55.00000</td>
<td>58.27047</td>
<td>61.73541</td>
<td>65.40639</td>
<td>69.30681</td>
</tr>
<tr>
<td></td>
<td>73.41619</td>
<td>77.78175</td>
<td>82.40689</td>
<td>87.30706</td>
<td>92.49861</td>
<td>97.99886</td>
<td>103.82617</td>
<td>110.0000</td>
</tr>
<tr>
<td></td>
<td>116.54094</td>
<td>123.47083</td>
<td>130.81278</td>
<td>138.59132</td>
<td>146.83238</td>
<td>155.56349</td>
<td>164.81378</td>
<td>174.18263</td>
</tr>
<tr>
<td></td>
<td>184.99721</td>
<td>195.99772</td>
<td>207.65235</td>
<td>220.00000</td>
<td>233.08188</td>
<td>246.94165</td>
<td>261.62557</td>
<td>277.88888</td>
</tr>
<tr>
<td></td>
<td>293.66477</td>
<td>311.12698</td>
<td>329.62756</td>
<td>349.22823</td>
<td>369.99442</td>
<td>391.99544</td>
<td>415.30470</td>
<td>440.0000</td>
</tr>
<tr>
<td></td>
<td>466.16376</td>
<td>493.88330</td>
<td>523.25113</td>
<td>554.36526</td>
<td>587.32954</td>
<td>622.25397</td>
<td>659.25511</td>
<td>698.45646</td>
</tr>
<tr>
<td></td>
<td>739.98885</td>
<td>783.99087</td>
<td>830.60940</td>
<td>880.00000</td>
<td>932.32752</td>
<td>987.76660</td>
<td>1046.50226</td>
<td>1108.73052</td>
</tr>
</tbody>
</table>

3.5. Tuning Tables
Number Theory

While this part of the package isn’t particularly fleshed out yet, there are a few number-theoretic functions for the analysis of scales.

Odd Limits

PyTuning contains functions for finding the odd Limit for both intervals and scales.

We can define and interval – say, \( \frac{45}{32} \), and find its odd-limit with the following:

```python
from pytuning.number_theory import odd_limit
interval = sp.Rational(45, 32)
limit = odd_limit(interval)
```

which yields an answer of 45.

One can also find the odd limit of an entire scale with the `find_odd_limit_for_scale()` function:

```python
from pytuning.scales import create_euler_fokker_scale
from pytuning.number_theory import find_odd_limit_for_scale

scale = create_euler_fokker_scale([3,5],[3,1])
limit = find_odd_limit_for_scale(scale)
```

which yields 135. (Examining the scale:

\[
\begin{bmatrix}
1, & 135, & 9, & 5, & 45, & 3, & 27, & 15, & 2
\end{bmatrix}
\]

you will see that this is the largest odd number, and is found in the second degree.)

pytuning.number_theory.odd_limit(i)

Find the odd-limit of an interval.

**Parameters**

- **i** – The interval (a `sympy.Rational`)

**Returns**

The odd-limit for the interval.

pytuning.number_theory.find_odd_limit_for_scale(s)

Find the odd-limit of an interval.

**Parameters**

- **s** – The scale (a list of `sympy.Rational` values)

**Returns**

The odd-limit for the scale.
Prime Limits

One can also compute prime limits for both scales and intervals. Extending the above example, one would assume that the Euler-Fokker scale would have a prime-limit of 5, since that’s the highest prime used in the generation, and in fact:

```python
from pytuning.scales import create_euler_fokker_scale
from pytuning.number_theory import find_prime_limit_for_scale

scale = create_euler_fokker_scale([3, 5], [3, 1])
limit = find_prime_limit_for_scale(scale)
```

will return 5 as the limit.

```python
pytuning.number_theory.prime_limit(i)
    Find the prime-limit of an interval.
    Parameters i – The interval (a sympy.Rational)
    Returns The prime-limit for the interval.
```

```python
pytuning.number_theory.find_prime_limit_for_scale(s)
    Find the prime-limit of an interval.
    Parameters s – The scale (a list of sympy.Rational values)
    Returns The prime-limit for the scale.
```

Prime Factorization of Degrees

PyTuning has functions for breaking a ratio down into prime factors, and the inverse of reassembling them.

```python
pytuning.number_theory.prime_factor_ratio(r, return_as_vector=False)
    Decompose a ratio (degree) to prime factors
    Parameters
    • r – The degree to factor (sympy.Rational)
    • return_as_vector – If True, return the factors as a vector (see below)
    Returns The factoring of the ratio
```

By default this function will return a dict, with each key being a prime, and each value being the exponent of the factorization.

As an example, the syntonic comma, $\frac{81}{80}$, can be factored:

```python
r = syntonic_comma
q = prime_factor_ratio(r)
print(q)
{2: -4, 3: 4, 5: -1}
```

which can be interpreted as:

$$\frac{3^4}{2^4 \cdot 5^1}$$

If return_as_vector is True, the function will return a tuple, with each position a successive prime. The above example yields:
```python
r = syntonic_comma
q = prime_factor_ratio(r, return_as_vector=True)
print(q)
(-4, 4, -1)
```

Note that zeros will be included in the vector, thus:

```python
r = sp.Rational(243, 224)
q = prime_factor_ratio(r, return_as_vector=True)
print(q)
(-5, 5, 0, -1)
q = prime_factor_ratio(r)
p = sympy.Rational(243, 224)
print(p)
{2: -5, 3: 5, 7: -1}
```

pytuning.number_theory.create_ratio_from_primes(factors)

Given a prime factorization of a ratio, reconstruct the ratio. This function is the inverse of prime_factor_ratio().

**Parameters**

- **factors** – The factors. Can be a dict or a tuple (see prime_factor_ratio for a discussion).

**Returns**
The ratio (sympy value).

### Utilities

The PyTuning package contains some utilities which may be useful. In general these tend to be smaller utilities and tasks that are useful in the analysis of musical scales, but they are not full-featured “things” in and of themselves.

### Interval Normalization

pytuning.utilities.normalize_interval(interval, octave=2)

Normalize a musical interval

**Parameters**

- **interval** – The interval to normalize. Should be a frequency ratio, most usefully expressed as a sympy.Rational or related data item
- **octave** – The formal octave. Defaults to 2

**Returns**
The interval, normalized

Note that any formal octave can be used. In normal usage a 2 will be used (i.e., a doubling of frequency is an octave).

Normalization works by finding the smallest power of two (or octave) that when multiplied by the interval (in the case of an interval less than 1) or divided into the interval (for intervals greater than 2) will bring the interval into the target range of $1 \leq i \leq 2$.

As an example, the interval $9 \frac{3}{8}$ would normalize to $\frac{9}{2^3}$, because $9$ needs to be scaled down by three octaves to fall within the limit of $1$ and $2$:

$$\frac{9}{8} = \frac{9}{2^3}$$
ni = normalize_interval(sp.Integer(9))
print(ni)
9/8

One can also normalize on a non-standard interval, for example, 3:

ni = normalize_interval(sp.Integer(34), octave=3)
print(ni)
34/27

Distinct Intervals

pytuning.utilities.distinct_intervals(scale)

Find the distinct intervals in a scale, including inversions

Parameters scale – The scale to analyze

Returns A list of distinct intervals

The scale should be specified as a list of sympy numerical values (Rational or Integer). Note that the convention adopted in this code is that scale[0] is a unison and scale[-1] is the formal octave (often 2).

As an example of a valid scale, a standardized Pythagorean tuning could be passed into the function:

\[
\begin{bmatrix}
1 & 256 & 9 & 32 & 81 & 4 & 1024 & 3 & 128 & 27 & 16 & 243 \\
\frac{1}{243} & \frac{8}{27} & \frac{64}{3} & \frac{729}{2} & \frac{81}{16} & \frac{9}{2} & \frac{128}{27} & \frac{256}{9} & \frac{9}{8} & \frac{32}{27} & \frac{81}{64} & \frac{4}{3}
\end{bmatrix}
\]

If one were hand-crafting this scale, it would look something like:

```python
import sympy as sp
scale = [sp.Integer(1), sp.Rational(256,243), sp.Rational(9,8), ...]
```

The function returns a list in rational/symbolic terms. If numerical values are needed, one can, for example, map ratio_to_cents to obtain it:

```python
di = distinct_intervals(scale)
di_in_cents = [ratio_to_cents(x) for x in di]
```

distinct_intervals() returns all the distinct intervals within a musical scale. Note, though, that it does not include the unison (or the octave) in the results, as all scales contain those intervals by definitions.

As an example, if we were to take a Pythagorean scale and find the intervals that exist within it:

```python
pythag = create_pythagorean_scale()
di = distinct_intervals(pythag)
```

we end up with:

\[
\begin{bmatrix}
2187 & 256 & 8192 & 262144 & 4096 & 3 & 243 & 1024 & 3 & 19683 & 729 & 6561 & 65536 & 177147 & 59049 & 81 \\
2048 & 243 & 6561 & 177147 & 2187 & 2 & 128 & 729 & 16384 & 512 & 4096 & 59049 & 131072 & 32768 & 64
\end{bmatrix}
\]

Converting a Ratio to a Cent Value

pytuning.utilities.ratio_to_cents(ratio)

Convert a scale degree to a cent value

Parameters ratio – The scale degree (sympy value)
Returns  The scale degree in cents

Calculates:

\[ \left( \frac{12}{1} \right)^{\text{degree}} \]

Note that this function returns a floating point number, not a sympy ratio.

This function is useful if you have a symbolic value (a rational or transcendental, for example) and you want to see its value in cents (a logarithmic scale in which there are 1200 steps in a factor of two). For example:

```python
interval = sp.Rational(3,2)  # A perfect fifth
cents = ratio_to_cents(interval)
print(cents)
701.955000865387
```

Converting a Cent Value to a Ratio

```python
pytuning.utilities.cents_to_ratio(cents)
```

Convert a cent value to a ratio

**Parameters**  
- **cents** – The degree value in cents

**Returns**  
The frequency ratio

This function takes a cent value and returns it as a frequency ratio (a sympy floating point number).

```python
print(cents_to_ratio(700.0))
1.49830707687668
```

(In other words, the 12-EDO fifth (700 cents) is very close to that of the Pythagorean fifth (\(\frac{3}{2}\), or 1.5).)

Converting a Note Number to a Frequency

```python
pytuning.utilities.note_number_to_freq(note, scale=None, reference_note=69, reference_frequency=440.0)
```

Convert a note number (MIDI) to a frequency (Hz).

**Parameters**

- **note** *(reference)* – The note number (0<=note<=127)
- **scale** – The scale. If none it assume EDO 12.
- **note** – The conversions reference note
- **reference_frequency** – The frequency of the reference note

**Returns**  
The frequency of the note in Hertz

The default values for **reference_note** and **reference_frequency** correspond to standard orchestral tuning, a4 = 440 Hz.

With this function we can calculate the frequency of any note number. If defaults to the MIDI standard, which pegs note number 69 to 440 Hz and uses a 12-EDO scale.

As an example, MIDI note 60 (Middle-C):

```python
print(note_number_to_freq(60))
261.625565300599
```
But if, for example, we wanted to use a different pitch standard, we could peg A to 444 Hz.

```python
print(note_number_to_freq(60, reference_frequency=444.0))
```

264.003979530604

You can also pass in a non-EDO tuning if you’re converting a different kind of scale to frequencies. This is used often in the code associated with the tuning tables.

**Naming A Ratio**

`pytuning.utilities.ratio_to_name(ratio)`  
Convert a scale degree to a name

- **Parameters**  
  - `ratio` – The input scale degree (a `sympy` value)

- **Returns**  
  - The degree name if found, `None` otherwise

This function will look up the name of a ratio and return it (returning `None`) if it is not found.

As an example:

```python
pythag = create_pythagorean_scale()
names = [ratio_to_name(x) for x in pythag]
```

names now contains:

- 'Unison'
- 'Pythagorean Minor Second'
- 'Pythagorean Major Second'
- 'Pythagorean Minor Third'
- 'Pythagorean Major Third'
- 'Perfect Fourth'
- 'Pythagorean Diminished Fifth'
- 'Perfect Fifth'
- 'Pythagorean Minor Sixth'
- 'Pythagorean Major Sixth'
- 'Pythagorean Minor Seventh'
- 'Pythagorean Major Seventh'
- 'Octave'

There are currently about 260 intervals in the internal catalog, so while not complete, the database is fairly extensive.

**Comparing Two Scales**

`pytuning.utilities.compare_two_scales(scale1, scale2, reference_freq=220.0, title=['Scale1', 'Scale2'])`

Compare two scales

- **param scale1** – The first scale (list of `sympy` values)
- **param scale2** – The second scale (list of `sympy` values)
- **param reference_freq** – The frequency (Hz) of the first degree
- **param title** – The scale names (list of strings with len = 2)

**returns** – `None`, (ie nothing)
This function will produce a simple textual representation of the difference between two scales. As an example, comparing the 12-EDO and Pythagorean scales:

```python
from pytuning.scales import create_edo_scale, create_pythagorean_scale
from pytuning.utilities import compare_two_scales

scale_1 = create_edo_scale(12)
scale_2 = create_pythagorean_scale()

compare_two_scales(scale_1, scale_2, title=['12-TET', 'Pythagorean'])
```

produces:

<table>
<thead>
<tr>
<th></th>
<th>12-TET</th>
<th></th>
<th>Pythagorean</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Cents</td>
<td>Freq</td>
<td>Cents</td>
<td>Freq</td>
</tr>
<tr>
<td>-------</td>
<td>--------</td>
<td>------------------</td>
<td>-------------</td>
<td>------------------</td>
</tr>
<tr>
<td>0.0000</td>
<td>220.0000</td>
<td>0.0000</td>
<td>220.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>100.0000</td>
<td>233.0819</td>
<td>90.2250</td>
<td>231.7695</td>
<td>9.7750</td>
</tr>
<tr>
<td>200.0000</td>
<td>246.9417</td>
<td>203.9100</td>
<td>247.5000</td>
<td>-3.9100</td>
</tr>
<tr>
<td>300.0000</td>
<td>261.6256</td>
<td>294.1350</td>
<td>260.7407</td>
<td>5.8650</td>
</tr>
<tr>
<td>400.0000</td>
<td>277.1826</td>
<td>407.8200</td>
<td>278.4375</td>
<td>-7.8200</td>
</tr>
<tr>
<td>500.0000</td>
<td>293.6648</td>
<td>498.0450</td>
<td>293.3333</td>
<td>1.9550</td>
</tr>
<tr>
<td>600.0000</td>
<td>311.1270</td>
<td>588.2700</td>
<td>309.0261</td>
<td>11.7300</td>
</tr>
<tr>
<td>700.0000</td>
<td>329.6276</td>
<td>701.9550</td>
<td>330.0000</td>
<td>-1.9550</td>
</tr>
<tr>
<td>800.0000</td>
<td>349.2282</td>
<td>792.1800</td>
<td>347.6543</td>
<td>7.8200</td>
</tr>
<tr>
<td>900.0000</td>
<td>369.9944</td>
<td>905.8650</td>
<td>371.2500</td>
<td>-5.8650</td>
</tr>
<tr>
<td>1000.0000</td>
<td>391.9954</td>
<td>996.0900</td>
<td>391.1111</td>
<td>3.9100</td>
</tr>
<tr>
<td>1100.0000</td>
<td>415.3047</td>
<td>1109.7750</td>
<td>417.6562</td>
<td>-9.7750</td>
</tr>
<tr>
<td>1200.0000</td>
<td>440.0000</td>
<td>1200.0000</td>
<td>440.0000</td>
<td>0.0000</td>
</tr>
</tbody>
</table>

Visualizations

PyTuning includes some visualizations for the analysis of scales, but this is not yet fleshed out.

The graphics are via Matplotlib, which is the de facto implementation of analytical graphics for python. There is also an optional dependency on Seaborn. Seaborn is good to include if you can – it makes the graphs better – but it’s a large package, owning to its dependencies (which include SciPy and Pandas; great packages, but extensive and large). If you have disk space to spare you may want to install it; otherwise you can get by with Matplotlib alone. On my system SciPy and Pandas weigh in at about 200 megabytes, not including any dependencies that they require. Matplotlib, on the other hand, is about 26 megabytes.

The Consonance Matrix

It’s nice to be able to get an intuitive feeling for the consonance or dissonance of a scale. For this, the consonance matrix is provided.

The consonance matrix forms an interval between every degree in the scale and applies a metric function to it. The default metric function just measures the denominator of the interval after any simplification it can undergo (and is really only meaningful for degrees which are expressed as integer ratios).

As an example, we can create a scale of the Euler-Fokker type:

```python
scale = create_euler_fokker_scale([3, 5], [2, 1])
```
PyTuning Documentation, Release 0.7.1

which is

\[
\begin{pmatrix}
1, & \frac{9}{8}, & \frac{5}{4}, & \frac{45}{32}, & \frac{3}{2}, & \frac{15}{8}, & 2
\end{pmatrix}
\]

Now, to create a consonance matrix:

```python
from pytuning.visualizations import consonance_matrix
matrix = consonance_matrix(scale)
```

`matrix` now contains a matplotlib `Figure`. To write it as PNG:

```python
matrix.savefig('consonance.png')
```

Which yields:

![Consonance Matrix](image)

Because this function accepts an arbitrary metric, it can be used for any analysis performed on the intervals of a scale; it does not need to really be a measurement or estimate of consonance. As an example, let’s say that (for some reason) you’re interested in how far from unity each interval is:

```python
def metric_unity(degree):
    normalized_degree = normalize_interval(degree)
    y = abs(1.0 - normalized_degree.evalf())
    return y
```

---

3.8. Visualizations 39
scale = create_euler_fokker_scale([3,5],[2,1])
matrix = consonance_matrix(scale, metric_function=metric_unity, title="Distance from Unity")

Our graph now looks like:

![Distance from Unity](image)

We could even do something like look for perfect fifths within our scale:

```python
def metric_fifth(degree):
    p5 = sp.Rational(3,2)
    normalized_degree = normalize_interval(degree)
    y = normalize_interval(p5 / normalized_degree).evalf()
    y = y if y == 1 else 0
    return y

scale = create_euler_fokker_scale([3,5],[2,1])
matrix = consonance_matrix(scale, metric_function=metric_fifth,
                           title="Relation to Perfect Fifth", annot=False)
```

which gives us:
where the highlighted cells denote perfect fifths.

You’ll note that the entry for $\frac{45}{32}$ and $\frac{15}{8}$ is 1 (a perfect fifth). We can verify this:

```python
print(normalize_interval(sp.Rational(45,32) / sp.Rational(15,8)))
```

$$\frac{3}{2}$$

**pytuning.visualizations.consonance_matrix**

```
pytuning.visualizations.consonance_matrix(scale, metric_function=None, figsize=(10, 8),
title='Consonance Matrix', annot=True, cmap=None, fig=None, vmin=None, vmax=None)
```

Display a consonance matrix for a scale

**Parameters**

- **scale** – The scale to analyze (list of frequency ratios)
- **metric_function** – The metric function (see below)
- **figsize** – Size of the figure (tuple, in inches)
- **title** – the graph title
- **annot** – If True, display the value of the metric in the grid cell
- **cmap** – A custom matplotlib colormap, if desired
• **fig** – a matplotlib.figure.Figure or None. If None a figure will be created
• **vmin** – If specified, the lowest value of the range to plot. Only works if Seaborn is included.
• **vmax** – If specified, the largest value of the range to plot. Only works if Seaborn is included.

**Returns** a matplotlib.Figure for display

The consonance matrix is created by taking each scale degree along the bottom and left edges of a matrix, forming a frequency ratio between the left and bottom value, and applying the metric function to that value.

The default metric function is the denominator of the normalized ratio – the thought being that the smaller this number is the more consonant the interval.

If a user-specified metric function is used, that function should accept one parameter (the ratio) and return one value. As an example, the default metric function:

```python
def metric_denom(degree):
    normalized_degree = normalize_interval(degree)
    return sympy.fraction(normalized_degree)[1]
```

Note that, as with most parts of this code, the ratios should be expressed in terms of sympy values, usually sympy.Rational's

A note of figures: With the default of fig=None, this function will just produce a figure and return it for display. However, if one wants to plot multiple figures and can pre-allocate it. This is useful, for example, if one wants to plot multiple figures on the same chart. As an example, the following will plot two separate consonance matrices, side-by-side:

```python
import matplotlib.pyplot as plt
fig = plt.figure(figsize=(15,6))
plt.subplot(plt.GridSpec(1, 2)[0,0])
fig = consonance_matrix(scale,title="Full Scale", fig=fig)
plt.subplot(plt.GridSpec(1, 2)[0,1])
fig = consonance_matrix(mode_scale,title="Mode Scale", fig=fig)
fig
```

**vmin** and **vmax** are useful if you’re plotting multiple graphs in the same figure; they can be used to ensure that all the component graphs use the same scale, so that visually the graphs are related. Otherwise each graph will have its own scale.

---

**Using PyTuning Interactively**

With a little knowledge of Python, one can use PyTuning in an interactive environment.

If you plan on doing this, I recommend using the Jupyter QtConsole. Jupyter is a full-featured interactive environment for several programming languages (the project started as IPython, but has expanded to cover many languages).

Included in the distribution is a script, interactive.py, which is useful for setting up your environment.

**Installing Jupyter**

(Note: I have experience with Linux and MacOS, so these instructions are focused on these platforms. Jupyter runs under Windows, but I have no experience on that platform.)

On Linux, good ways of installing Jupyter include using your native package manager or installing it via a third-party distribution.
Native Packages

On Ubuntu Jupyter is still referred to as IPython. On Xenial, for example, there are packages for both Python 2.7 and Python 3:

```
vagrant@ubuntu-xenial:~$ aptitude search qtconsole
p  ipython-qtconsole  - enhanced interactive Python shell - Qt
   console
p  ipython3-qtconsole - enhanced interactive Python 3 shell - Qt
   console
vagrant@ubuntu-xenial:~$
```

On Arch Linux:

```
mark@lucid:~$ pacman -Ss qtconsole
community/python-qtconsole 4.2.1-1
   Qt-based console for Jupyter with support for rich media output
community/python2-qtconsole 4.2.1-1
   Qt-based console for Jupyter with support for rich media output
mark@lucid:~$
```

(PyTuning will run under either Python 2.7 or Python 3.X, so the version you install is up to you.)

I would also suggest installing Matplotlib so that graphics can be used within the console, i.e:

```
vagrant@ubuntu-xenial:~$ aptitude search matplotlib
p  python-matplotlib - Python based plotting system in a
   style similar to
p  python-matplotlib:.i386 - Python based plotting system in a
   style similar to
p  python-matplotlib-data - Python based plotting system (data
   package)
. . .
p  python3-matplotlib - Python 3 plotting area-proportional two-
   and three-
p  python3-matplotlib:dbg - Python based plotting system (debug
   extension)
. . .
vagrant@ubuntu-xenial:~$
```

And optionally Seaborn:

```
vagrant@ubuntu-xenial:~$ aptitude search seaborn
p  python-seaborn - statistical visualization library
p  python3-seaborn - statistical visualization library
vagrant@ubuntu-xenial:~$
```
(But note that Seaborn is a bit large. See the discussion in *Visualizations.*)

**Third Party Packages**

Jupyter can also be installed via third-party Python distributions. This is my preferred way of doing it, and on MacOS it is (in my opinion) the only viable option. I imagine that a third-party distribution would be the easiest way to do this on Windows.

One good distribution is Continuum Analytics Miniconda. Once miniconda is installed, the conda tool can be used to install the necessary packages:

```bash
vagrant@ubuntu-xenial:~$ conda install jupyter qtconsole matplotlib seaborn sympy numpy
```

**Setting the Environment**

The PyTuning distribution contains a script, `interactive.py`, that can be used to import the package into your namespace, as well as setting up some convenience functions. Where that script lives on your computer can vary by platform as well as python distribution. If you’re on Linux and installed PyTuning with your system python, there’s a good chance it’s in `/usr/bin`. If you installed into the Miniconda distribution, then it will probably be somewhere like `~/miniconda/bin`.

Once you’ve launched the console, this script should be loaded into the environment with the `%load` command. This will load it into the console, but you’ll need to execute it. This is normally done with `Shift-enter`, although `Control-enter` may be used on some platforms/versions.
This will bring load the script into the console, at which point a [Shit-Enter] will execute it.
A (Very) Brief Introduction to Jupyter

There are a few things about Jupyter which are useful for interacting with the Python interpreter.

Tab Completion

Jupyter has a very good tab completion system which can save a lot of typing.

As a first example, the `%load` command (above) can use completion to navigate to the file. One need only type enough to disambiguate each directory in the path and the tab will complete it, much in the same way that the bash shell will do so.

Tab completion can also be used to find functions that are defined in your namespace. As an example, by typing `create_` into the console and hitting tab you will see all objects and functions that begin with that string, and by hitting the tab a second time a selector will be brought up that allows you to select the function you’re after:
Tool Tips

Jupyter also has a nice tool-tip function that will show a function’s documentation once the opening \() is typed:
History

Jupyter also has a nice history function. If, for example, at some point in your session you entered the following:

```
In [3]: scale = create_harmonic_scale(2,6)
In [4]:
```

Then, later on in the session if you type `scale =`, each time you hit the up arrow it will search through your history and bring up lines beginning with that character string. You can then edit that line and make changes.

```
In [3]: scale = create_harmonic_scale(2,6)
In [4]: scale2 = create_harmonic_scale(2,7)
In [5]:
```

Rich Display

By default scales and degrees will be displayed symbolically. If you want text display you can use the `print()` function.
Graphics

With matplotlib installed one also has access to graphics. A graph can be displayed within the console, or it can be displayed in a simple viewer that allows some editing and the saving of the graphic file in a few different formats (JPEG, PNG, SVG). The viewer comes up automatically. If you close it at want to bring it back up later, you can use the `show()` function (i.e., `matrix.show()`).
Note that by re-sizing the window, you re-size the graphic.

You can also save the figure directly from the console:
In [16]: matrix
Out[16]:

Consonance Matrix

In [17]: matrix.savefig('test.png')

Signature: matrix.savefig(*args, **kwargs)
Docstring:
Save the current figure.

Call signature:

    savefig(fname=None, dpi=100, facecolor='w', edgecolor='w',
             orientation='portrait', papertype=None, format=None,
             transparent=False, bbox_inches=None, pad_inches=0.1,
             frameon=None)

The output formats available depend on the backend being used.

Arguments:

    *fname*:
    A string containing a path to a filename, or a Python file-like object, or possibly some backend-dependent object such as :class:`matplotlib.backends.backend_pdf.PdfPages`.

[Documentation continues...]
A Sample Session

On my personal website I discuss a scale that I’ve been working with recently for music composition. It’s a mode of the harmonic scale which minimizes dissonance by one of the metrics included in the distribution. In the following session I create the scale and create two tuning tables (a timidity and scala table) for use in music composition.

```
In [17]: scale = create_harmonic_scale(4, 30)
In [18]: modes = find_best_modes(scale, 7, ['sum_p_q_for_all_intervals'], 1, sum_p_q_for_all_intervals)
In [19]: my_scale = modes[0]['scale']
In [20]: my_scale
Out[20]:
[1, 9/8, 5/4, 21/16, 3/2, 7/4, 15/8, 2]

In [21]: with open('scale.timidity', 'w') as tim_file:
    ...:     tim_file.write(create_timidity_tuning(my_scale, reference_note=69))
    ...

In [22]: with open('scale.scala', 'w') as scala_file:
    ...:     scala_file.write(createScala_tuning(my_scale, "Scale 1"))
    ...
```

Helper Functions

interactive.py also creates a few helper functions for the creation of scales. They wrap the base functions in an interactive prompt and define a global variable, `scales` into which the calculated scale is placed.

As an example, to create a harmonic scale:
Only a few functions have yet been written, but more will be included in future releases.

```python
pytuning.interactive.harmonic_scale()
Create a harmonic scale in an interactive environment.

This function will prompt for the first and last harmonic, as well as the user's desire for normalization. It assumes the standard octave of 2.

The output is put into the global `scale`.

pytuning.interactive.edo_scale()
Create an EDO scale in an interactive environment.

This function will prompt for the number of divisions (tones) and the formal octave.

The output is put into the global `scale`.

pytuning.interactive.euler_fokker()
Create a scale of the Euler Fokker Genus in an interactive environment.

This function will prompt for generator primes and the formal octave. Note that multiplicities is not prompted for, so if there are repetitions they will need to be spelled out separately.

The output is put into the global `scale`.

### Running in a Persistant Way

Jupyter also offers an interactive notebook, similar to a Matlab notebook. For more complicated analysis it is my preferred way of interacting with the PyTuning library. Documentation, graphics, equations, code, and output calculations can all be included. It can be installed in a way that is similar to the console (and in fact may be installed along with it, depending on how the packages maintainers on your platform have chosen to break things up).

The Github repository for this project has a directory which contains a rendered notebook that shows an exploration of pentatonic scales in the Pythagorean tuning. Github renders notebooks well, so you can see what’s possible to
decide if you want to install the software. If you’re going to be doing anything really complicated in an interactive environment, I would recommend installing and using this.
CHAPTER 4

Index

- genindex
Index

A
all_metrics() (in module pytuning.metrics), 20

C
cents_to_ratio() (in module pytuning.utilities), 36
compare_two_scales() (in module pytuning.utilities), 37
consonance_matrix() (in module pytuning.visualizations), 41
create_csound_tuning() (in module pytuning.tuning_tables), 31
create_diatonic_scale() (in module pytuning.scales), 17
create_edo_scale() (in module pytuning.scales), 13
create_equal_interval_scale() (in module pytuning.scales), 14
create_euler_fokker_scale() (in module pytuning.scales), 16
create_fluidsynth_tuning() (in module pytuning.tuning_tables), 30
create_harmonic_scale() (in module pytuning.interactive), 53
create_pythagorean_scale() (in module pytuning.scales), 15
create_quarter_comma_meantone_scale() (in module pytuning.scales), 15
create_ratio_from_primes() (in module pytuning.number_theory), 34
create_scale_tuning() (in module pytuning.tuning_tables), 28
create_scale_from_scale() (in module pytuning.scale_creation), 25
create_timidity_tuning() (in module pytuning.tuning_tables), 27

D
distinct_intervals() (in module pytuning.utilities), 35

E
edo_scale() (in module pytuning.interactive), 53
euler_fokker() (in module pytuning.interactive), 53

F
find_best_modes() (in module pytuning.scale_creation), 21
find_odd_limit_for_scale() (in module pytuning.number_theory), 32
find_prime_limit_for_scale() (in module pytuning.number_theory), 33

H
harmonic_scale() (in module pytuning.interactive), 53

M
metric_3() (in module pytuning.metrics), 19

N
normalize_interval() (in module pytuning.utilities), 34
note_number_to_freq() (in module pytuning.utilities), 36

O
odd_limit() (in module pytuning.number_theory), 32

P
prime_factor_ratio() (in module pytuning.number_theory), 33
prime_limit() (in module pytuning.number_theory), 33

R
ratio_to_cents() (in module pytuning.utilities), 35
ratio_to_name() (in module pytuning.utilities), 37

S
sum_distinct_intervals() (in module pytuning.metrics), 19
sum_p_q() (in module pytuning.metrics), 18
sum_p_q_for_all_intervals() (in module pytuning.metrics), 19
sum_q_for_all_intervals() (in module pytuning.metrics), 20