The scikit-fuzzy Documentation

Release 0.2

The scikit-image team

June 19, 2016

Contents

This SciKit is a fuzzy logic toolbox for SciPy.

CHAPTER 1

Sections

1.1 SciKit-Fuzzy

Scikit-Fuzzy is a collection of fuzzy logic algorithms intended for use in the SciPy Stack, written in the Python computing language.

This SciKit is developed by the SciPy community. Contributions are welcome! Please join us on the mailing list or our persistent chatroom on Gitter.IM.

1.1.1 Homepage and package documentation

http://pythonhosted.org/scikit-fuzzy/

1.1.2 Source, bugs, and development

http://github.com/scikit-fuzzy/scikit-fuzzy

1.1.3 Gitter.IM

https://gitter.im/scikit-fuzzy/scikit-fuzzy

1.1.4 Mailing List

http://groups.google.com/group/scikit-fuzzy

1.2 Pre-built installation

On systems that support setuptools, the package can be installed from the Python packaging index using

```
easy_install -U scikit-fuzzy
```

or

```
pip install -U scikit-fuzzy
```

1.3 Installation from source

Obtain the source from the git-repository at http://github.com/scikit-fuzzy/scikit-fuzzy by running:

```
git clone http://github.com/scikit-fuzzy/scikit-fuzzy.git
```

in a terminal (you will need to have git installed on your machine).

If you do not have git installed, you can also download a zipball from https://github.com/scikit-fuzzy/scikit-fuzzy/zipball/master.

The SciKit can be installed globally using:

```
pip install -e .
```

or locally using:

```
python setup.py install --prefix=${HOME}
```

If you prefer, you can use it without installing, by simply adding this path to your `PYTHONPATH` variable.

1.4 User Guide

1.4.1 Getting started

`scikit-fuzzy` is an fuzzy logic Python package that works with `numpy` arrays. The package is imported as `skfuzzy`:

```
>>> import skfuzzy
```

though the recommended import statement uses an alias:

```
>>> import skfuzzy as fuzz
```

Most functions of `skfuzzy` are brought into the base package namespace. You can introspect the functions available in `fuzz` when using IPython by:

```
[1] import skfuzzy as fuzz
[2] fuzz.
```

and pressing the **Tab** key.

1.4.2 Finding your way around

A list of submodules and functions is found on the API reference webpage.

Within `scikit-fuzzy`, universe variables and fuzzy membership functions are represented by `numpy` arrays. Generation of membership functions is as simple as:

```
>>> import numpy as np
>>> import skfuzzy as fuzz
>>> x = np.arange(11)
>>> mfx = fuzz.trimf(x, [0, 5, 10])
>>> x
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
>>> mfx
array([ 0. ,  0.2,  0.4,  0.6,  0.8,  1. ,  0.8,  0.6,  0.4,  0.2,  0. ])
```

While most functions are available in the base namespace, the package is factored with a logical grouping of functions in submodules. If the base namespace appears overwhelming, we recommend exploring them individually. These include

`fuzz.membership`
Fuzzy membership function generation

`fuzz.defuzzify`
Defuzzification algorithms to return crisp results from fuzzy sets

`fuzz.fuzzymath`
The core of `scikit-fuzzy`, containing the majority of the most common fuzzy logic operations.

`fuzz.intervals`
Interval mathematics. The restricted Dong, Shah, & Wong (DSW) methods for fuzzy set math live here.

`fuzz.image`
Limited fuzzy logic image processing operations.

`fuzz.cluster`
Fuzzy c-means clustering.

`fuzz.filters`
Fuzzy Inference Ruled by Else-action (FIRE) filters in 1D and 2D.

1.4.3 Fuzzy Control Primer

Overiveiw and Terminology

Fuzzy Logic is a methodology predicated on the idea that the "truthiness" of something can be expressed over a continuum. This is to say that something isn't *true* or *false* but instead *partially true* or *partially false*.

A **fuzzy variable** has a **crisp value** which takes on some number over a pre-defined domain (in fuzzy logic terms, called a **universe**). The crisp value is how we think of the variable using normal mathematics. For example, if my fuzzy variable was how much to tip someone, it's universe would be 0 to 25% and it might take on a crisp value of 15%.

A fuzzy variable also has several **terms** that are used to describe the variable. The terms taken together are the **fuzzy set** which can be used to describe the "fuzzy value" of a fuzzy variable. These terms are usually adjectives like "poor," "mediocre," and "good." Each term has a **membership function** that defines how a crisp value maps to the term on a scale of 0 to 1. In essence, it describes "how good" something is.

So, back to the tip example, a "good tip" might have a membership function which has non-zero values between 15% and 25%, with 25% being a "completely good tip" (ie, it's membership is 1.0) and 15% being a "barely good tip" (ie, its membership is 0.1).

A **fuzzy control system** links fuzzy variables using a set of **rules**. These rules are simply mappings that describe how one or more fuzzy variables relates to another. These are expressed in terms of an IF-THEN statement; the IF part is called the **antecedent** and the THEN part is the **consequent**. In the tiping example, one rule might be "IF the service was good THEN the tip will be good." The exact math related to how a rule is used to calcualte the value of the consequent based on the value of the antecedent is outside the scope of this primer.

The Tipping Problem

Taking the tipping example full circle, if we were to create a controller which estimates the tip we should give at a restaurant, we might structure it as such:

- **Antecednets (Inputs)**
  - ***service***
    - Universe (ie, crisp value range): How good was the service of the waitress, on a scale of 1 to 10?
    - Fuzzy set (ie, fuzzy value range): poor, acceptable, amazing
  - ***food quality***
    - Universe: How tasty was the food, on a scale of 1 to 10?
    - Fuzzy set: bad, decent, great
- **Consequents (Outputs)**
  - ***tip***
    - Universe: How much should we tip, on a scale of 0% to 25%
    - Fuzzy set: low, medium, high
- **Rules**
  - IF the *service* was good *or* the *food quality* was good, THEN the tip will be high.
  - IF the *service* was average, THEN the tip will be medium.
  - IF the *service* was poor *and* the *food quality* was poor THEN the tip will be low.
- **Usage**
  - **If I tell this controller that I rated:**
    - the service as 9.8, and
    - the quality as 6.5,
  - **it would recommend I leave:**
    - a 20.2% tip.

Example

To see a worked example of the tipping problem using the `scikit-fuzzy` library visit the Fuzzy Control Systems example.

1.5 How to contribute to `skfuzzy`

Developing Open Source is great fun! Join us on the scikit-fuzzy mailing list and tell us which challenges you'd like to solve.

- Guidance is available for those new to scientific programming in Python.
- If you're looking for something to implement, you can browse the open issues on GitHub or suggest a new, useful feature.
- The technical detail of the *development process* is summed up below. Refer to the `gitwash` for a step-by-step tutorial.

1.5.1 Development process

Here's the long and short of it:

1. If you are a first-time contributor:
   - Go to https://github.com/scikit-fuzzy/scikit-fuzzy and click the "fork" button to create your own copy of the project.
   - Clone the project to your local computer:

```
git clone git@github.com:your-username/scikit-fuzzy.git
```

   - Add the upstream repository:

```
git remote add upstream git@github.com:scikit-fuzzy/scikit-fuzzy.git
```

   - Now, you have remote repositories named:
     - `upstream`, which refers to the `scikit-fuzzy` repository
     - `origin`, which refers to your personal fork
2. Develop your contribution:
   - Pull the latest changes from upstream:

```
git checkout master
git pull upstream master
```

   - Create a branch for the feature you want to work on. Since the branch name will appear in the merge message, use a sensible name such as 'transform-speedups':

```
git checkout -b transform-speedups
```

   - Commit locally as you progress (`git add` and `git commit`)
3. To submit your contribution:
   - Push your changes back to your fork on GitHub:

```
git push origin transform-speedups
```

   - Go to GitHub. The new branch will show up with a green Pull Request button - click it.

- If you want, post on the mailing list to explain your changes or to ask for review.

For a more detailed discussion, read these `detailed documents` on how to use Git with `scikit-fuzzy` (../git-wash/index.html).

4. Review process:
   - Reviewers (the other developers and interested community members) will write inline and/or general comments on your Pull Request (PR) to help you improve its implementation, documentation and style. Every single developer working on the project has their code reviewed, and we've come to see it as friendly conversation from which we all learn and the overall code quality benefits. Therefore, please don't let the review discourage you from contributing: its only aim is to improve the quality of project, not to criticize (we are, after all, very grateful for the time you're donating!).
   - To update your pull request, make your changes on your local repository and commit. As soon as those changes are pushed up (to the same branch as before) the pull request will update automatically.
   - Travis-CI, a continuous integration service, is triggered after each Pull Request update to build the code, run unit tests, measure code coverage and check coding style (PEP8) of your branch. The Travis tests must pass before your PR can be merged. If Travis fails, you can find out why by clicking on the "failed" icon (red cross) and inspecting the build and test log.
5. Document changes

   Before merging your commits, you must add a description of your changes to the release notes of the upcoming version in `doc/release/release_dev.txt`.

Note: To reviewers: if it is not obvious, add a short explanation of what a branch did to the merge message and, if closing a bug, also add "Closes #123" where 123 is the issue number.

1.5.2 Divergence between `upstream master` and your feature branch

Do *not* ever merge the main branch into yours. If GitHub indicates that the branch of your Pull Request can no longer be merged automatically, rebase onto master:

```
git checkout master
git pull upstream master
git checkout transform-speedups
git rebase master
```

If any conflicts occur, fix the according files and continue:

```
git add conflict-file1 conflict-file2
git rebase --continue
```

However, you should only rebase your own branches and must generally not rebase any branch which you collaborate on with someone else.

Finally, you must push your rebased branch:

```
git push --force origin transform-speedups
```

(If you are curious, here's a further discussion on the dangers of rebasing. Also see this LWN article.)

1.5.3 Guidelines

- All code should have tests (see *test coverage* below for more details).

- All code should be documented, to the same standard as NumPy and SciPy.
- For new functionality, always add an example to the gallery.
- No changes are ever committed without review. Ask on the mailing list if you get no response to your pull request. **Never merge your own pull request.**
- Examples in the gallery should have a maximum figure width of 8 inches.

1.5.4 Stylistic Guidelines

- Set up your editor to remove trailing whitespace. Follow PEP08. Check code with pyflakes / flake8.
- Use numpy data types instead of strings, e.g., `np.uint8` instead of `"uint8"`.
- Use the following import conventions:

```
import numpy as np
import matplotlib.pyplot as plt

cimport numpy as cnp # in Cython code
```

- When documenting array parameters, use `image : (M, N) ndarray` and then refer to `M` and `N` in the docstring, if necessary.
- Functions should support all input image dtypes. Use utility functions such as `img_as_float` to help convert to an appropriate type. The output format can be whatever is most efficient. This allows us to string together several functions into a pipeline, e.g.:

```
hough(canny(my_image))
```

- Use `Py_ssize_t` as data type for all indexing, shape and size variables in C/C++ and Cython code.

1.5.5 Test coverage

Tests for a module should ideally cover all code in that module, i.e., statement coverage should be at 100%.

To measure the test coverage, install coverage.py (using `easy_install coverage`) and then run:

```
$ make coverage
```

This will print a report with one line for each file in *skfuzzy*, detailing the test coverage:

```
Name                                          Stmts   Miss  Cover   Missing
---------------------------------------------------------------------------
skfuzzy.cluster                                   2      0   100%
skfuzzy.defuzzify                                 2      0   100%
skfuzzy.filters                                   2      0   100%
...
```

1.5.6 Activate Travis-CI for your fork (optional)

Travis-CI checks all unittests in the project to prevent breakage.

Before sending a pull request, you may want to check that Travis-CI successfully passes all tests. To do so,

- Go to Travis-CI and follow the Sign In link at the top
- Go to your profile page and switch on your scikit-fuzzy fork

It corresponds to steps one and two in Travis-CI documentation (Step three is already done in scikit-fuzzy).

Thus, as soon as you push your code to your fork, it will trigger Travis-CI, and you will receive an email notification when the process is done.

Every time Travis is triggered, it also calls on Coveralls to inspect the current test overage.

1.5.7 Bugs

Please report bugs on GitHub.

1.6 License

```
Unless otherwise specified by LICENSE.txt files in subdirectories (or below),
all code is:

Copyright (c) 2012, the scikit-fuzzy team
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:

    1) Redistributions of source code must retain the above copyright notice,
       this list of conditions and the following disclaimer.
    2) Redistributions in binary form must reproduce the above copyright
       notice, this list of conditions and the following disclaimer in the
       documentation and/or other materials provided with the distribution.
    3) Neither the name of scikit-fuzzy (a.k.a. skfuzzy) nor the names of its
       contributors may be used to endorse or promote products derived from
       this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.


################################################################################
Overall package structure, setup.py, and .travis.yml are derived from
scikit-image. They are also covered by the 3-clause BSD license. The original
code for these elements are:

Copyright (C) 2011, the scikit-image team
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are
met:

 1. Redistributions of source code must retain the above copyright
```

```
    notice, this list of conditions and the following disclaimer.
 2. Redistributions in binary form must reproduce the above copyright
    notice, this list of conditions and the following disclaimer in
    the documentation and/or other materials provided with the
    distribution.
 3. Neither the name of skimage nor the names of its contributors may be
    used to endorse or promote products derived from this software without
    specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR ``AS IS'' AND ANY EXPRESS OR
IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
DISCLAIMED. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT,
INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
(INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING
IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
POSSIBILITY OF SUCH DAMAGE.
```

1.7 General examples

General-purpose and introductory examples for the scikit.

1.7.1 Fuzzy c-means clustering

Fuzzy logic principles can be used to cluster multidimensional data, assigning each point a *membership* in each cluster center from 0 to 100 percent. This can be very powerful compared to traditional hard-thresholded clustering where every point is assigned a crisp, exact label.

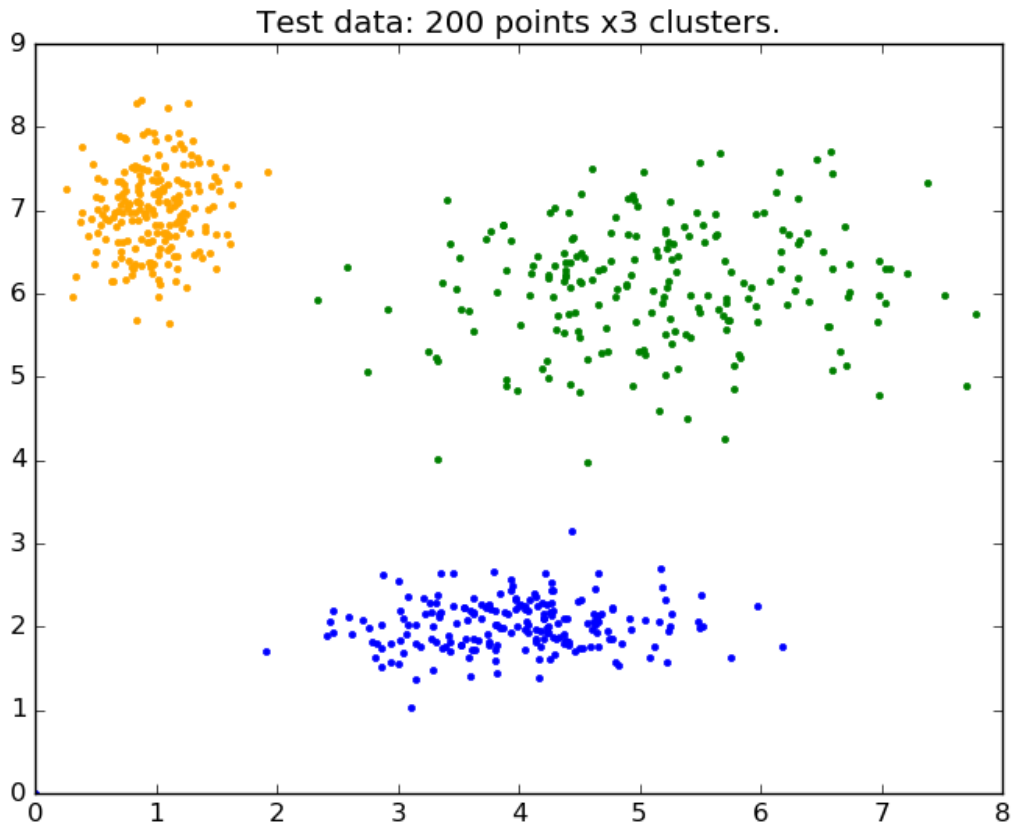
Fuzzy c-means clustering is accomplished via `skfuzzy.cmeans`, and the output from this function can be repurposed to classify new data according to the calculated clusters (also known as *prediction*) via `skfuzzy.cmeans_predict`

Data generation and setup

In this example we will first undertake necessary imports, then define some test data to work with.

```
from __future__ import division, print_function
import numpy as np
import matplotlib.pyplot as plt
import skfuzzy as fuzz

colors = ['b', 'orange', 'g', 'r', 'c', 'm', 'y', 'k', 'Brown', 'ForestGreen']

# Define three cluster centers
centers = [[4, 2],
           [1, 7],
           [5, 6]]

# Define three cluster sigmas in x and y, respectively
sigmas = [[0.8, 0.3],
          [0.3, 0.5],
```

```
          [1.1, 0.7]]

# Generate test data
np.random.seed(42)  # Set seed for reproducibility
xpts = np.zeros(1)
ypts = np.zeros(1)
labels = np.zeros(1)
for i, ((xmu, ymu), (xsigma, ysigma)) in enumerate(zip(centers, sigmas)):
    xpts = np.hstack((xpts, np.random.standard_normal(200) * xsigma + xmu))
    ypts = np.hstack((ypts, np.random.standard_normal(200) * ysigma + ymu))
    labels = np.hstack((labels, np.ones(200) * i))

# Visualize the test data
fig0, ax0 = plt.subplots()
for label in range(3):
    ax0.plot(xpts[labels == label], ypts[labels == label], '.',
             color=colors[label])
ax0.set_title('Test data: 200 points x3 clusters.')
```

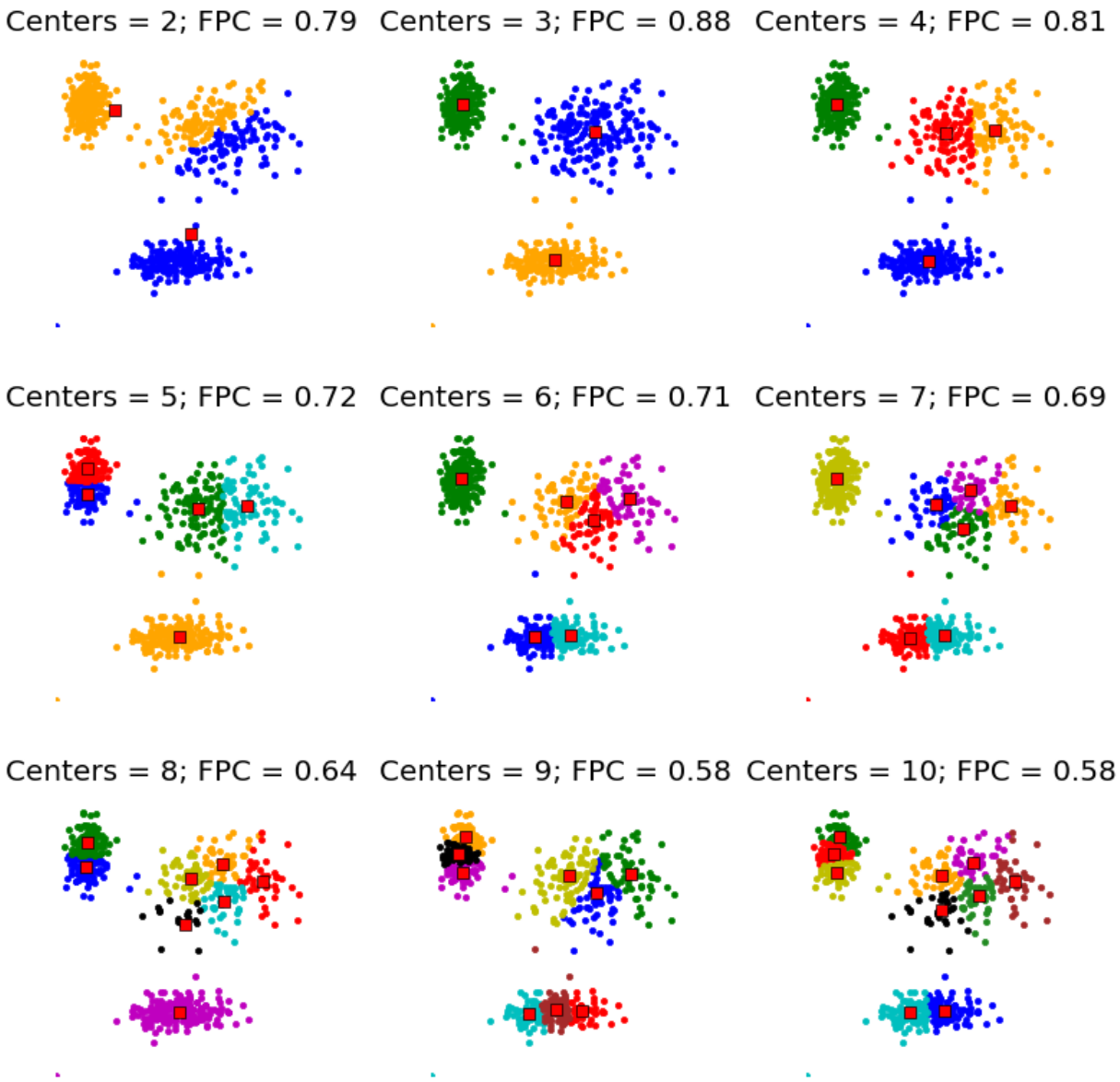


Clustering

Above is our test data. We see three distinct blobs. However, what would happen if we didn't know how many clusters we should expect? Perhaps if the data were not so clearly clustered?

Let's try clustering our data several times, with between 2 and 9 clusters.

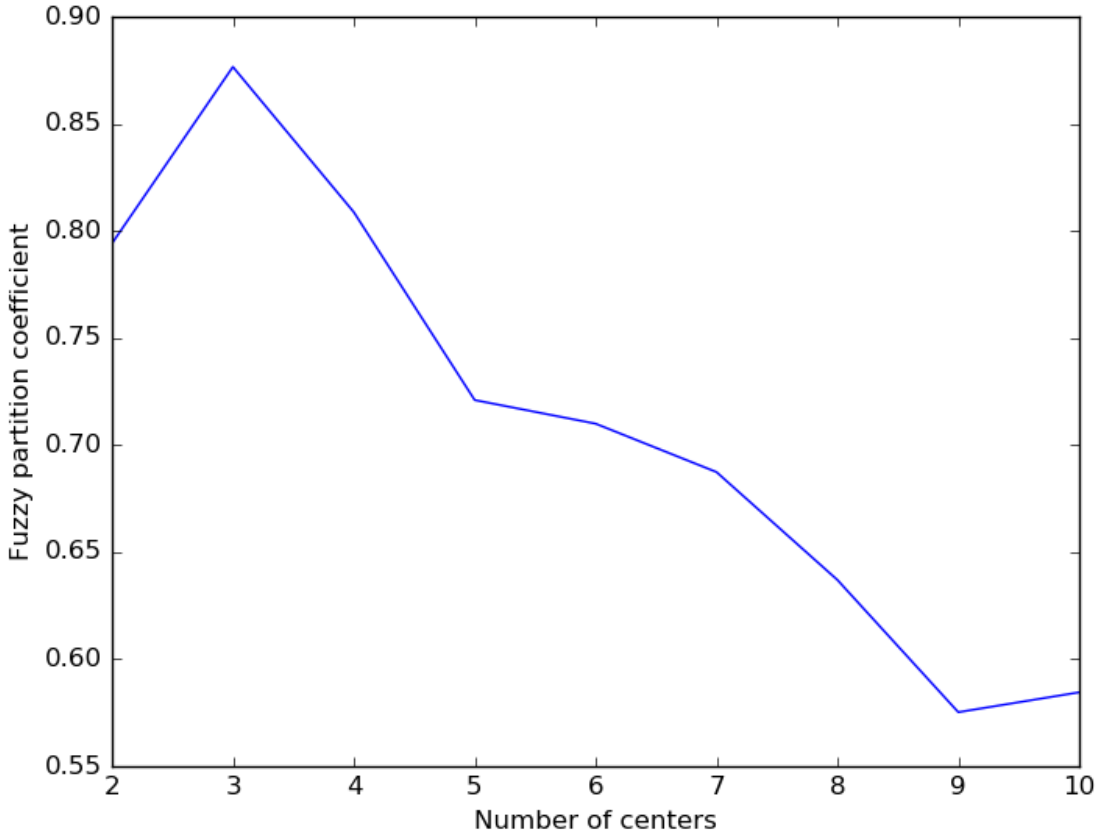
```
# Set up the loop and plot
fig1, axes1 = plt.subplots(3, 3, figsize=(8, 8))
alldata = np.vstack((xpts, ypts))
fpcs = []

for ncenters, ax in enumerate(axes1.reshape(-1), 2):
    cntr, u, u0, d, jm, p, fpc = fuzz.cluster.cmeans(
        alldata, ncenters, 2, error=0.005, maxiter=1000, init=None)

    # Store fpc values for later
    fpcs.append(fpc)

    # Plot assigned clusters, for each data point in training set
    cluster_membership = np.argmax(u, axis=0)
    for j in range(ncenters):
        ax.plot(xpts[cluster_membership == j],
                ypts[cluster_membership == j], '.', color=colors[j])

    # Mark the center of each fuzzy cluster
    for pt in cntr:
        ax.plot(pt[0], pt[1], 'rs')

    ax.set_title('Centers = {0}; FPC = {1:.2f}'.format(ncenters, fpc))
    ax.axis('off')

fig1.tight_layout()
```



The fuzzy partition coefficient (FPC)

The FPC is defined on the range from 0 to 1, with 1 being best. It is a metric which tells us how cleanly our data is described by a certain model. Next we will cluster our set of data - which we know has three clusters - several times, with between 2 and 9 clusters. We will then show the results of the clustering, and plot the fuzzy partition coefficient. When the FPC is maximized, our data is described best.

```
fig2, ax2 = plt.subplots()
ax2.plot(np.r_[2:11], fpcs)
ax2.set_xlabel("Number of centers")
ax2.set_ylabel("Fuzzy partition coefficient")
```



As we can see, the ideal number of centers is 3. This isn't news for our contrived example, but having the FPC available can be very useful when the structure of your data is unclear.

Note that we started with *two* centers, not one; clustering a dataset with only one cluster center is the trivial solution and will by definition return FPC == 1.

1.7.2 Classifying New Data

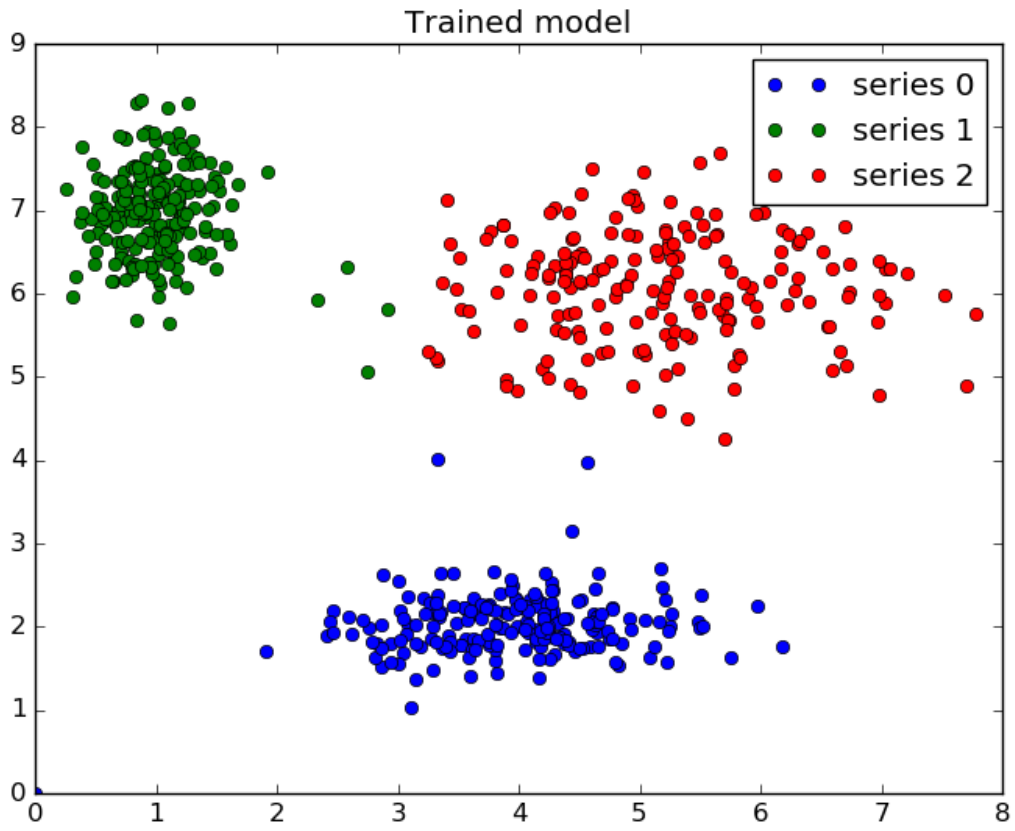
Now that we can cluster data, the next step is often fitting new points into an existing model. This is known as prediction. It requires both an existing model and new data to be classified.

Building the model

We know our best model has three cluster centers. We'll rebuild a 3-cluster model for use in prediction, generate new uniform data, and predict which cluster to which each new data point belongs.

```
# Regenerate fuzzy model with 3 cluster centers - note that center ordering
# is random in this clustering algorithm, so the centers may change places
cntr, u_orig, _, _, _, _, _ = fuzz.cluster.cmeans(
    alldata, 3, 2, error=0.005, maxiter=1000)

# Show 3-cluster model
fig2, ax2 = plt.subplots()
ax2.set_title('Trained model')
```

```
for j in range(3):
    ax2.plot(alldata[0, u_orig.argmax(axis=0) == j],
             alldata[1, u_orig.argmax(axis=0) == j], 'o',
             label='series ' + str(j))
ax2.legend()
```

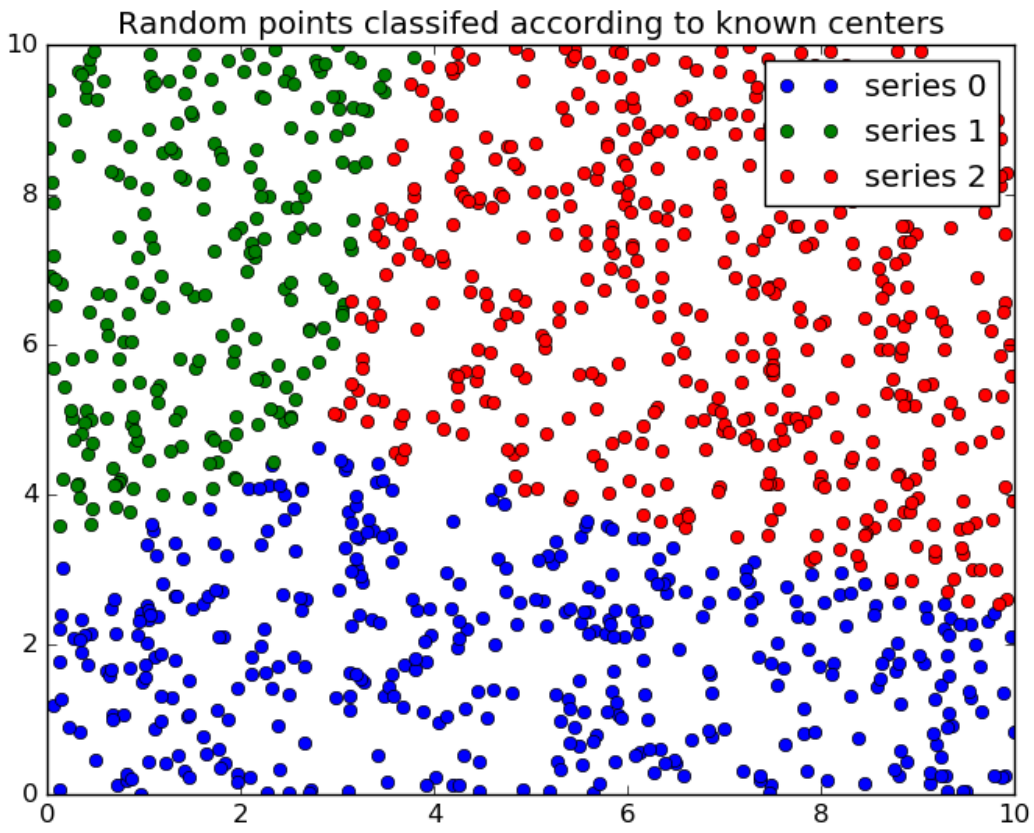


Prediction

Finally, we generate uniformly sampled data over this field and classify it via `cmeans_predict`, incorporating it into the pre-existing model.

```
# Generate uniformly sampled data spread across the range [0, 10] in x and y
newdata = np.random.uniform(0, 1, (1100, 2)) * 10

# Predict new cluster membership with `cmeans_predict` as well as
# `cntr` from the 3-cluster model
u, u0, d, jm, p, fpc = fuzz.cluster.cmeans_predict(
    newdata.T, cntr, 2, error=0.005, maxiter=1000)

# Plot the classified uniform data. Note for visualization the maximum
# membership value has been taken at each point (i.e. these are hardened,
# not fuzzy results visualized) but the full fuzzy result is the output
# from cmeans_predict.
cluster_membership = np.argmax(u, axis=0)  # Hardening for visualization
```

```
fig3, ax3 = plt.subplots()
ax3.set_title('Random points classifed according to known centers')
for j in range(3):
    ax3.plot(newdata[cluster_membership == j, 0],
             newdata[cluster_membership == j, 1], 'o',
             label='series ' + str(j))
ax3.legend()

plt.show()
```



Python source code: `download` (generated using `skimage` 0.2)

1.7.3 Fuzzy Control Systems: Advanced Example

The tipping problem is a classic, simple example. If you're new to this, start with the Fuzzy Control Primer and move on to the tipping problem.

This example assumes you're familiar with those topics. Go on. We'll wait.

Typical Fuzzy Control System

Many fuzzy control systems are tasked to keep a certain variable close to a specific value. For instance, the temperature for an industrial chemical process might need to be kept relatively constant. In order to do this, the system usually knows two things:

- The *error*, or deviation from the ideal value
- The way the error is changing. This is the mathematical first derivative; we'll call it *delta*

From these two values we can construct a system which will act appropriately.

Set up the Fuzzy Control System

We'll use the new control system API for this problem. It would be far too complicated to model manually!

```
import numpy as np
import skfuzzy.control as ctrl

# Sparse universe makes calculations faster, without sacrifice accuracy.
# Only the critical points are included here; making it higher resolution is
# unnecessary.
universe = np.linspace(-2, 2, 5)

# Create the three fuzzy variables - two inputs, one output
error = ctrl.Antecedent(universe, 'error')
delta = ctrl.Antecedent(universe, 'delta')
output = ctrl.Consequent(universe, 'output')

# Here we use the convenience `automf` to populate the fuzzy variables with
# terms. The optional kwarg `names=` lets us specify the names of our Terms.
names = ['nb', 'ns', 'ze', 'ps', 'pb']
error.automf(names=names)
delta.automf(names=names)
output.automf(names=names)
```

Define complex rules

This system has a complicated, fully connected set of rules defined below.

```
rule0 = ctrl.Rule(antecedent=((error['nb'] & delta['nb']) |
                              (error['ns'] & delta['nb']) |
                              (error['nb'] & delta['ns'])),
                  consequent=output['nb'], label='rule nb')

rule1 = ctrl.Rule(antecedent=((error['nb'] & delta['ze']) |
                              (error['nb'] & delta['ps']) |
                              (error['ns'] & delta['ns']) |
                              (error['ns'] & delta['ze']) |
                              (error['ze'] & delta['ns']) |
                              (error['ze'] & delta['nb']) |
                              (error['ps'] & delta['nb'])),
                  consequent=output['ns'], label='rule ns')

rule2 = ctrl.Rule(antecedent=((error['nb'] & delta['pb']) |
                              (error['ns'] & delta['ps']) |
                              (error['ze'] & delta['ze']) |
                              (error['ps'] & delta['ns']) |
                              (error['pb'] & delta['nb'])),
                  consequent=output['ze'], label='rule ze')

rule3 = ctrl.Rule(antecedent=((error['ns'] & delta['pb']) |
                              (error['ze'] & delta['pb']) |
```

```
                                (error['ze'] & delta['ps']) |
                                (error['ps'] & delta['ps']) |
                                (error['ps'] & delta['ze']) |
                                (error['pb'] & delta['ze']) |
                                (error['pb'] & delta['ns'])),
                    consequent=output['ps'], label='rule ps')

rule4 = ctrl.Rule(antecedent=((error['ps'] & delta['pb']) |
                                (error['pb'] & delta['pb']) |
                                (error['pb'] & delta['ps'])),
                    consequent=output['pb'], label='rule pb')
```

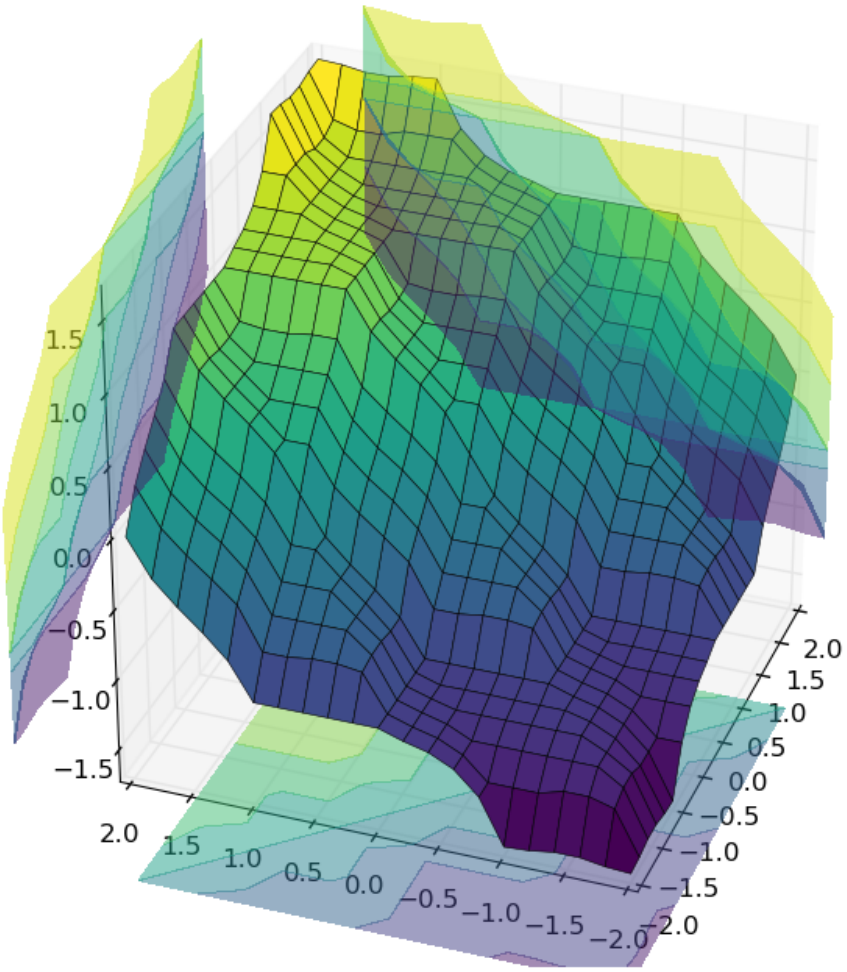
Despite the lengthy ruleset, the new fuzzy control system framework will execute in milliseconds. Next we add these rules to a new `ControlSystem` and define a `ControlSystemSimulation` to run it.

```
system = ctrl.ControlSystem(rules=[rule0, rule1, rule2, rule3, rule4])

# Later we intend to run this system with a 21*21 set of inputs, so we allow
# that many plus one unique runs before results are flushed.
# Subsequent runs would return in 1/8 the time!
sim = ctrl.ControlSystemSimulation(system, flush_after_run=21 * 21 + 1)
```

View the control space

With helpful use of Matplotlib and repeated simulations, we can observe what the entire control system surface looks like in three dimensions!

```
# We can simulate at higher resolution with full accuracy
upsampled = np.linspace(-2, 2, 21)
x, y = np.meshgrid(upsampled, upsampled)
z = np.zeros_like(x)

# Loop through the system 21*21 times to collect the control surface
for i in range(21):
    for j in range(21):
        sim.input['error'] = x[i, j]
        sim.input['delta'] = y[i, j]
        sim.compute()
        z[i, j] = sim.output['output']

# Plot the result in pretty 3D with alpha blending
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D  # Required for 3D plotting

fig = plt.figure(figsize=(8, 8))
ax = fig.add_subplot(111, projection='3d')

surf = ax.plot_surface(x, y, z, rstride=1, cstride=1, cmap='viridis',
                       linewidth=0.4, antialiased=True)

cset = ax.contourf(x, y, z, zdir='z', offset=-2.5, cmap='viridis', alpha=0.5)
cset = ax.contourf(x, y, z, zdir='x', offset=3, cmap='viridis', alpha=0.5)
cset = ax.contourf(x, y, z, zdir='y', offset=3, cmap='viridis', alpha=0.5)

ax.view_init(30, 200)
```



Final thoughts

This example used a number of new, advanced techniques which may be helpful in practical fuzzy system design:

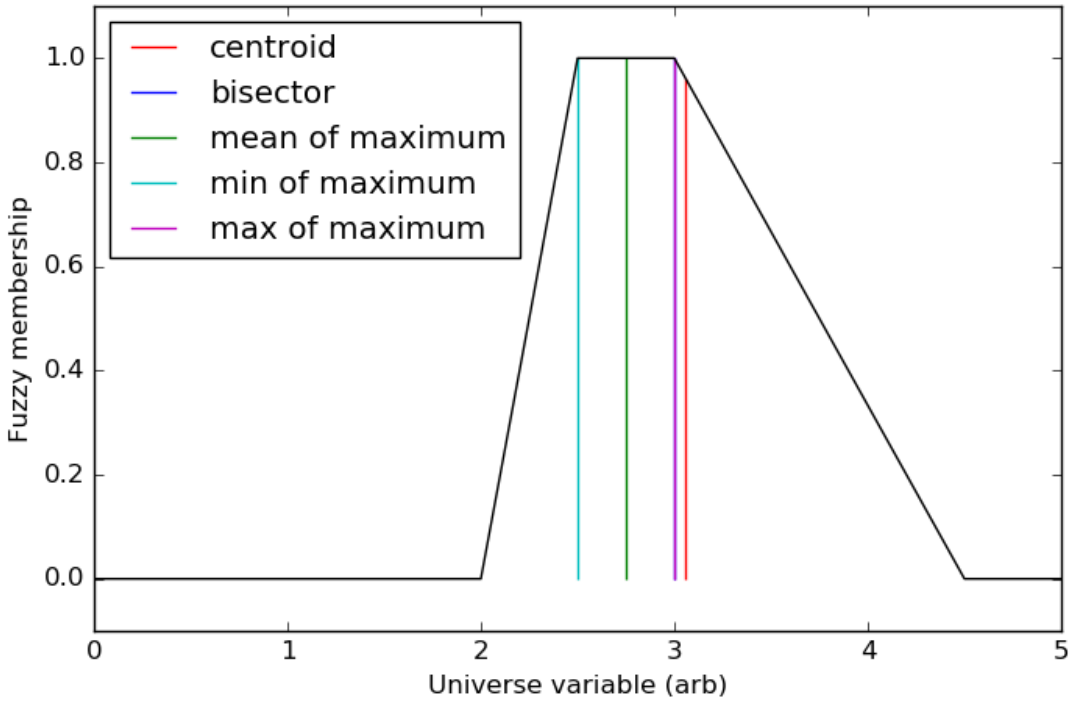
- A highly sparse (maximally sparse) system
- Controll of Term names generated by *automf*
- A long and logically complicated ruleset, with order-of-operations respected
- Control of the cache flushing on creation of a ControlSystemSimulation, which can be tuned as needed depending on memory constraints
- Repeated runs of a ControlSystemSimulation
- Creating and viewing a control surface in 3D.

Python source code: `download` (generated using `skimage` 0.2)

1.7.4 Defuzzification

Fuzzy logic calculations are excellent tools, but to use them the fuzzy result must be converted back into a single number. This is known as defuzzification.

There are several possible methods for defuzzification, exposed via *skfuzzy.defuzz*.




```
import numpy as np
import matplotlib.pyplot as plt
import skfuzzy as fuzz


# Generate trapezoidal membership function on range [0, 1]
x = np.arange(0, 5.05, 0.1)
mfx = fuzz.trapmf(x, [2, 2.5, 3, 4.5])

# Defuzzify this membership function five ways
defuzz_centroid = fuzz.defuzz(x, mfx, 'centroid')  # Same as skfuzzy.centroid
defuzz_bisector = fuzz.defuzz(x, mfx, 'bisector')
defuzz_mom = fuzz.defuzz(x, mfx, 'mom')
defuzz_som = fuzz.defuzz(x, mfx, 'som')
defuzz_lom = fuzz.defuzz(x, mfx, 'lom')

# Collect info for vertical lines
labels = ['centroid', 'bisector', 'mean of maximum', 'min of maximum',
          'max of maximum']
xvals = [defuzz_centroid,
         defuzz_bisector,
```

```
         defuzz_mom,
         defuzz_som,
         defuzz_lom]
colors = ['r', 'b', 'g', 'c', 'm']
ymax = [fuzz.interp_membership(x, mfx, i) for i in xvals]

# Display and compare defuzzification results against membership function
plt.figure(figsize=(8, 5))

plt.plot(x, mfx, 'k')
for xv, y, label, color in zip(xvals, ymax, labels, colors):
    plt.vlines(xv, 0, y, label=label, color=color)
plt.ylabel('Fuzzy membership')
plt.xlabel('Universe variable (arb)')
plt.ylim(-0.1, 1.1)
plt.legend(loc=2)

plt.show()
```

Python source code: `download` (generated using `skimage` 0.2)

1.7.5 The Tipping Problem - The Hard Way

> Note: This method computes everything by hand, step by step. For most people, the new API for fuzzy systems will be preferable. The same problem is solved with the new API in this example.

The 'tipping problem' is commonly used to illustrate the power of fuzzy logic principles to generate complex behavior from a compact, intuitive set of expert rules.

Input variables

A number of variables play into the decision about how much to tip while dining. Consider two of them:

- `quality` : Quality of the food
- `service` : Quality of the service

Output variable

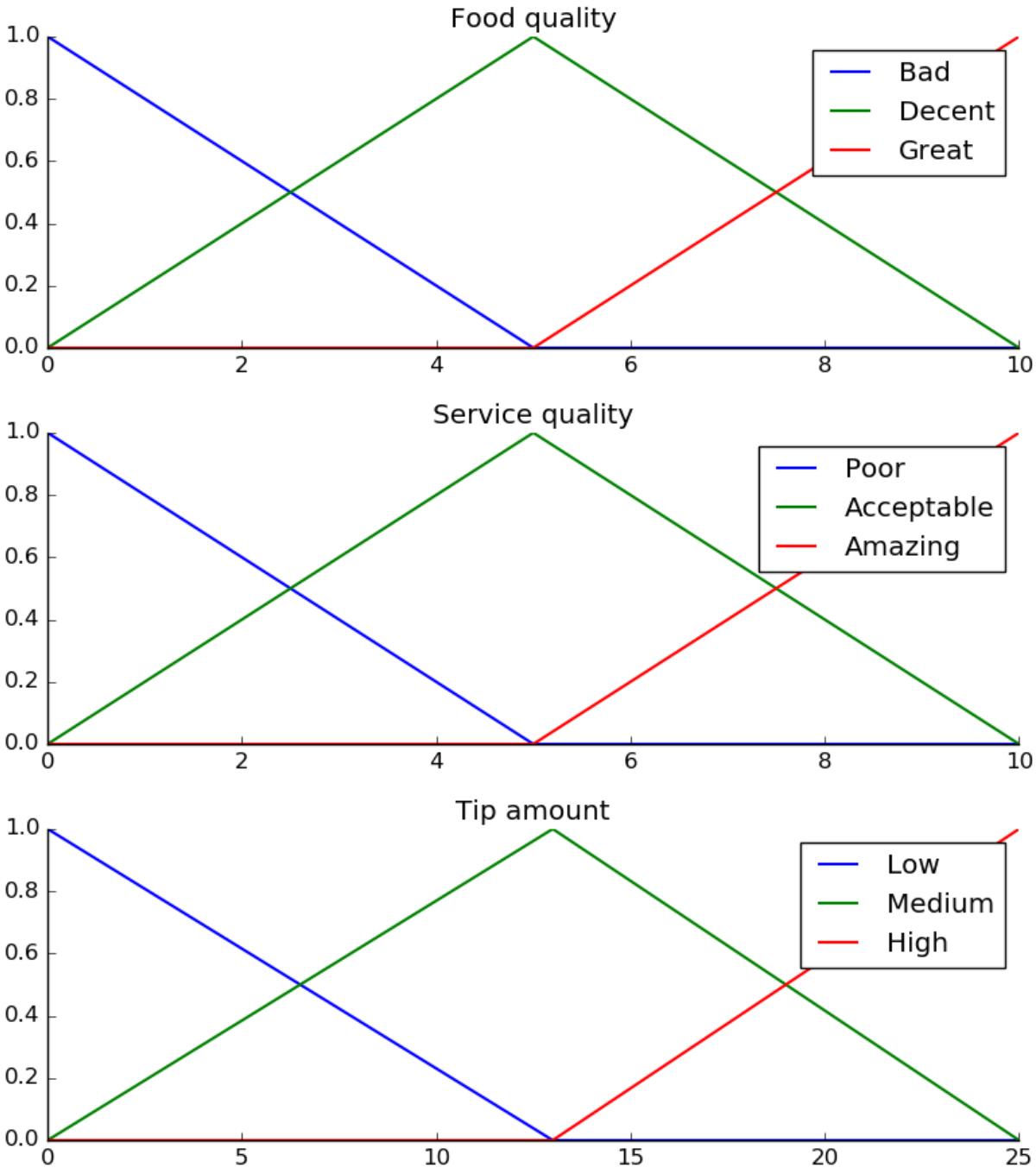
The output variable is simply the tip amount, in percentage points:

- `tip` : Percent of bill to add as tip

For the purposes of discussion, let's say we need 'high', 'medium', and 'low' membership functions for both input variables and our output variable. These are defined in scikit-fuzzy as follows

```
import numpy as np
import skfuzzy as fuzz
import matplotlib.pyplot as plt

# Generate universe variables
#   * Quality and service on subjective ranges [0, 10]
#   * Tip has a range of [0, 25] in units of percentage points
x_qual = np.arange(0, 11, 1)
x_serv = np.arange(0, 11, 1)
x_tip  = np.arange(0, 26, 1)
```

```
# Generate fuzzy membership functions
qual_lo = fuzz.trimf(x_qual, [0, 0, 5])
qual_md = fuzz.trimf(x_qual, [0, 5, 10])
qual_hi = fuzz.trimf(x_qual, [5, 10, 10])
serv_lo = fuzz.trimf(x_serv, [0, 0, 5])
serv_md = fuzz.trimf(x_serv, [0, 5, 10])
serv_hi = fuzz.trimf(x_serv, [5, 10, 10])
tip_lo = fuzz.trimf(x_tip, [0, 0, 13])
tip_md = fuzz.trimf(x_tip, [0, 13, 25])
tip_hi = fuzz.trimf(x_tip, [13, 25, 25])

# Visualize these universes and membership functions
fig, (ax0, ax1, ax2) = plt.subplots(nrows=3, figsize=(8, 9))

ax0.plot(x_qual, qual_lo, 'b', linewidth=1.5, label='Bad')
ax0.plot(x_qual, qual_md, 'g', linewidth=1.5, label='Decent')
ax0.plot(x_qual, qual_hi, 'r', linewidth=1.5, label='Great')
ax0.set_title('Food quality')
ax0.legend()

ax1.plot(x_serv, serv_lo, 'b', linewidth=1.5, label='Poor')
ax1.plot(x_serv, serv_md, 'g', linewidth=1.5, label='Acceptable')
ax1.plot(x_serv, serv_hi, 'r', linewidth=1.5, label='Amazing')
ax1.set_title('Service quality')
ax1.legend()

ax2.plot(x_tip, tip_lo, 'b', linewidth=1.5, label='Low')
ax2.plot(x_tip, tip_md, 'g', linewidth=1.5, label='Medium')
ax2.plot(x_tip, tip_hi, 'r', linewidth=1.5, label='High')
ax2.set_title('Tip amount')
ax2.legend()

# Turn off top/right axes
for ax in (ax0, ax1, ax2):
    ax.spines['top'].set_visible(False)
    ax.spines['right'].set_visible(False)
    ax.get_xaxis().tick_bottom()
    ax.get_yaxis().tick_left()

plt.tight_layout()
```



Fuzzy rules

Now, to make these triangles useful, we define the *fuzzy relationship* between input and output variables. For the purposes of our example, consider three simple rules:

1. If the food is bad OR the service is poor, then the tip will be low
2. If the service is acceptable, then the tip will be medium
3. If the food is great OR the service is amazing, then the tip will be high.

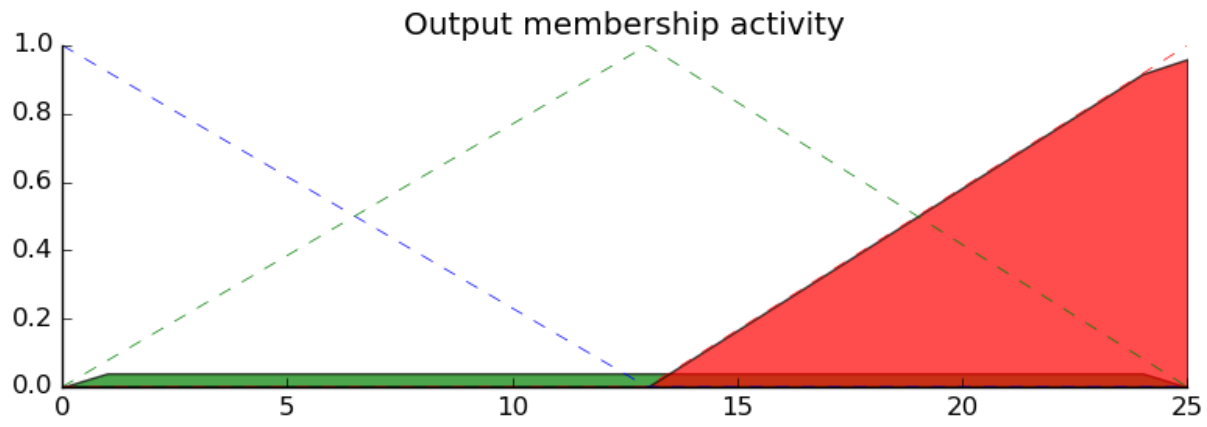
Most people would agree on these rules, but the rules are fuzzy. Mapping the imprecise rules into a defined, actionable tip is a challenge. This is the kind of task at which fuzzy logic excels.

Rule application

What would the tip be in the following circumstance:

- Food *quality* was **6.5**
- *Service* was **9.8**

```
# We need the activation of our fuzzy membership functions at these values.
# The exact values 6.5 and 9.8 do not exist on our universes...
# This is what fuzz.interp_membership exists for!
qual_level_lo = fuzz.interp_membership(x_qual, qual_lo, 6.5)
qual_level_md = fuzz.interp_membership(x_qual, qual_md, 6.5)
qual_level_hi = fuzz.interp_membership(x_qual, qual_hi, 6.5)

serv_level_lo = fuzz.interp_membership(x_serv, serv_lo, 9.8)
serv_level_md = fuzz.interp_membership(x_serv, serv_md, 9.8)
serv_level_hi = fuzz.interp_membership(x_serv, serv_hi, 9.8)

# Now we take our rules and apply them. Rule 1 concerns bad food OR service.
# The OR operator means we take the maximum of these two.
active_rule1 = np.fmax(qual_level_lo, serv_level_lo)

# Now we apply this by clipping the top off the corresponding output
# membership function with `np.fmin`
tip_activation_lo = np.fmin(active_rule1, tip_lo)  # removed entirely to 0

# For rule 2 we connect acceptable service to medium tipping
tip_activation_md = np.fmin(serv_level_md, tip_md)

# For rule 3 we connect high service OR high food with high tipping
active_rule3 = np.fmax(qual_level_hi, serv_level_hi)
tip_activation_hi = np.fmin(active_rule3, tip_hi)
tip0 = np.zeros_like(x_tip)

# Visualize this
fig, ax0 = plt.subplots(figsize=(8, 3))

ax0.fill_between(x_tip, tip0, tip_activation_lo, facecolor='b', alpha=0.7)
ax0.plot(x_tip, tip_lo, 'b', linewidth=0.5, linestyle='--', )
ax0.fill_between(x_tip, tip0, tip_activation_md, facecolor='g', alpha=0.7)
ax0.plot(x_tip, tip_md, 'g', linewidth=0.5, linestyle='--')
ax0.fill_between(x_tip, tip0, tip_activation_hi, facecolor='r', alpha=0.7)
ax0.plot(x_tip, tip_hi, 'r', linewidth=0.5, linestyle='--')
ax0.set_title('Output membership activity')

# Turn off top/right axes
for ax in (ax0,):
    ax.spines['top'].set_visible(False)
    ax.spines['right'].set_visible(False)
    ax.get_xaxis().tick_bottom()
    ax.get_yaxis().tick_left()

plt.tight_layout()
```



Rule aggregation

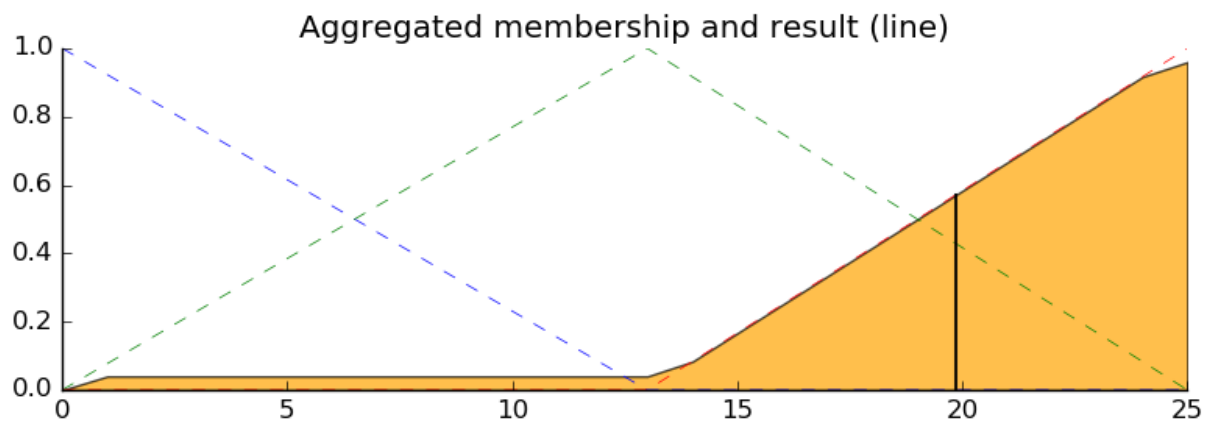
With the *activity* of each output membership function known, all output membership functions must be combined. This is typically done using a maximum operator. This step is also known as *aggregation*.

Defuzzification

Finally, to get a real world answer, we return to *crisp* logic from the world of fuzzy membership functions. For the purposes of this example the centroid method will be used.

The result is a tip of 20.2%.

```
# Aggregate all three output membership functions together
aggregated = np.fmax(tip_activation_lo,
                     np.fmax(tip_activation_md, tip_activation_hi))

# Calculate defuzzified result
tip = fuzz.defuzz(x_tip, aggregated, 'centroid')
tip_activation = fuzz.interp_membership(x_tip, aggregated, tip)  # for plot

# Visualize this
fig, ax0 = plt.subplots(figsize=(8, 3))

ax0.plot(x_tip, tip_lo, 'b', linewidth=0.5, linestyle='--', )
ax0.plot(x_tip, tip_md, 'g', linewidth=0.5, linestyle='--')
ax0.plot(x_tip, tip_hi, 'r', linewidth=0.5, linestyle='--')
ax0.fill_between(x_tip, tip0, aggregated, facecolor='Orange', alpha=0.7)
ax0.plot([tip, tip], [0, tip_activation], 'k', linewidth=1.5, alpha=0.9)
ax0.set_title('Aggregated membership and result (line)')

# Turn off top/right axes
for ax in (ax0,):
    ax.spines['top'].set_visible(False)
    ax.spines['right'].set_visible(False)
    ax.get_xaxis().tick_bottom()
    ax.get_yaxis().tick_left()

plt.tight_layout()
```



Final thoughts

The power of fuzzy systems is allowing complicated, intuitive behavior based on a sparse system of rules with minimal overhead. Note our membership function universes were coarse, only defined at the integers, but `fuzz.interp_membership` allowed the effective resolution to increase on demand. This system can respond to arbitrarily small changes in inputs, and the processing burden is minimal.

Python source code: `download` (generated using `skimage` 0.2)

1.7.6 Fuzzy Control Systems: The Tipping Problem

The 'tipping problem' is commonly used to illustrate the power of fuzzy logic principles to generate complex behavior from a compact, intuitive set of expert rules.

If you're new to the world of fuzzy control systems, you might want to check out the Fuzzy Control Primer before reading through this worked example.

The Tipping Problem

Let's create a fuzzy control system which models how you might choose to tip at a restaurant. When tipping, you consider the service and food quality, rated between 0 and 10. You use this to leave a tip of between 0 and 25%.

We would formulate this problem as:

- **Antecednets (Inputs)**
  - ***service***
    - Universe (ie, crisp value range): How good was the service of the wait staff, on a scale of 0 to 10?
    - Fuzzy set (ie, fuzzy value range): poor, acceptable, amazing
  - ***food quality***
    - Universe: How tasty was the food, on a scale of 0 to 10?
    - Fuzzy set: bad, decent, great

- **Consequents (Outputs)**
  - ***tip***
    - Universe: How much should we tip, on a scale of 0% to 25%
    - Fuzzy set: low, medium, high
- **Rules**
  - IF the *service* was good *or* the *food quality* was good, THEN the tip will be high.
  - IF the *service* was average, THEN the tip will be medium.
  - IF the *service* was poor *and* the *food quality* was poor THEN the tip will be low.
- **Usage**
  - **If I tell this controller that I rated:**
    - the service as 9.8, and
    - the quality as 6.5,
  - **it would recommend I leave:**
    - a 20.2% tip.

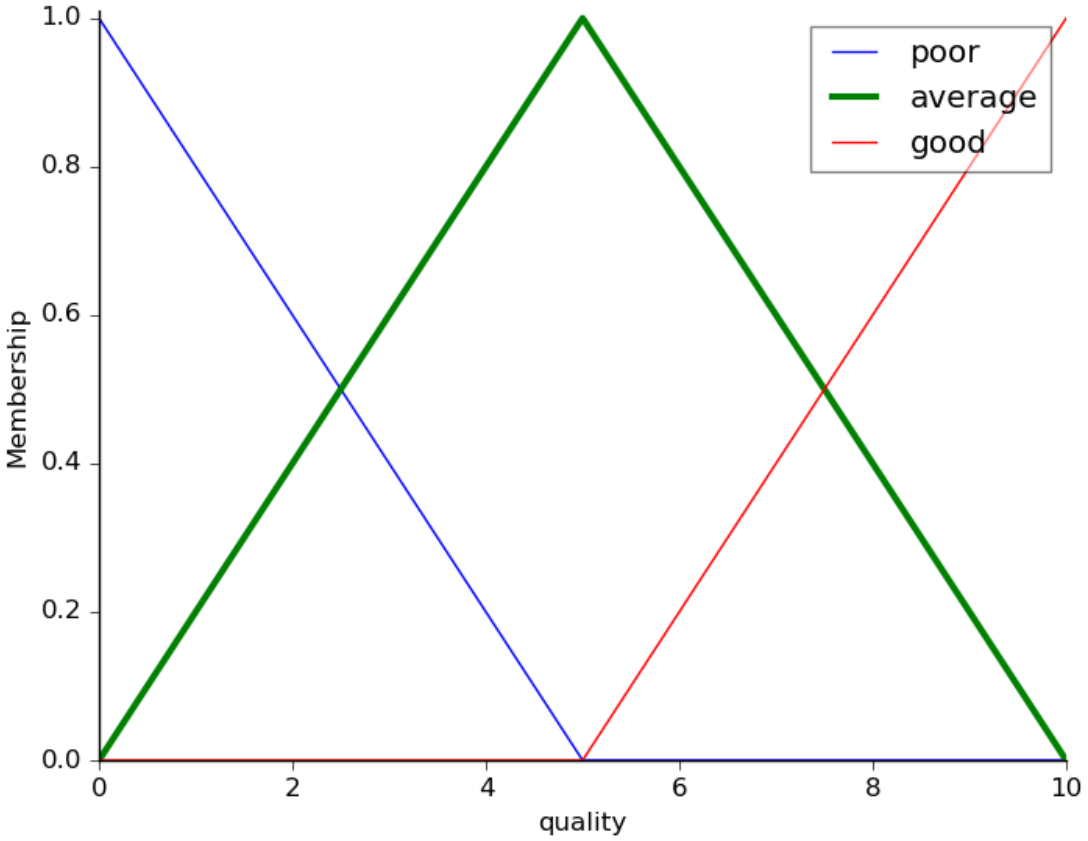
Creating the Tipping Controller Using the skfuzzy control API

We can use the *skfuzzy* control system API to model this. First, let's define fuzzy variables

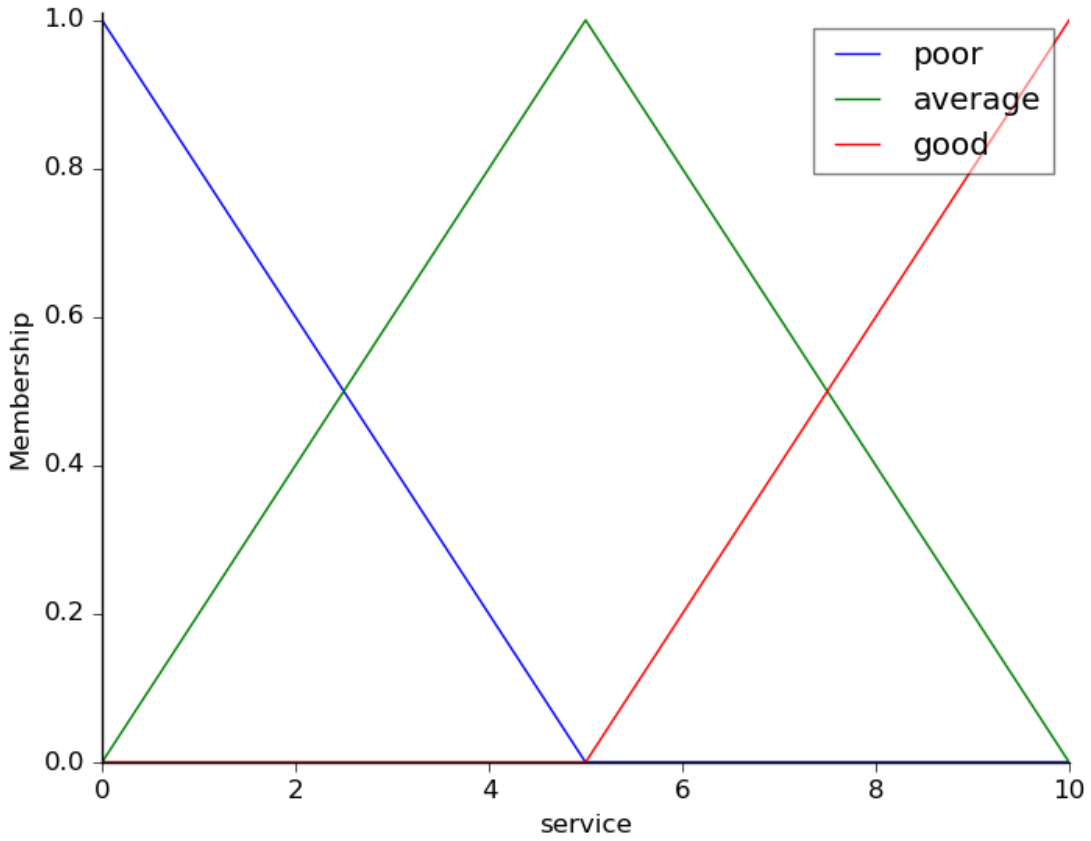
```
import numpy as np
import skfuzzy as fuzz
from skfuzzy import control as ctrl

# New Antecedent/Consequent objects hold universe variables and membership
# functions
quality = ctrl.Antecedent(np.arange(0, 11, 1), 'quality')
service = ctrl.Antecedent(np.arange(0, 11, 1), 'service')
tip = ctrl.Consequent(np.arange(0, 26, 1), 'tip')

# Auto-membership function population is possible with .automf(3, 5, or 7)
quality.automf(3)
service.automf(3)

# Custom membership functions can be built interactively with a familiar,
# Pythonic API
tip['low'] = fuzz.trimf(tip.universe, [0, 0, 13])
tip['medium'] = fuzz.trimf(tip.universe, [0, 13, 25])
tip['high'] = fuzz.trimf(tip.universe, [13, 25, 25])
```

To help understand what the membership looks like, use the `view` methods.

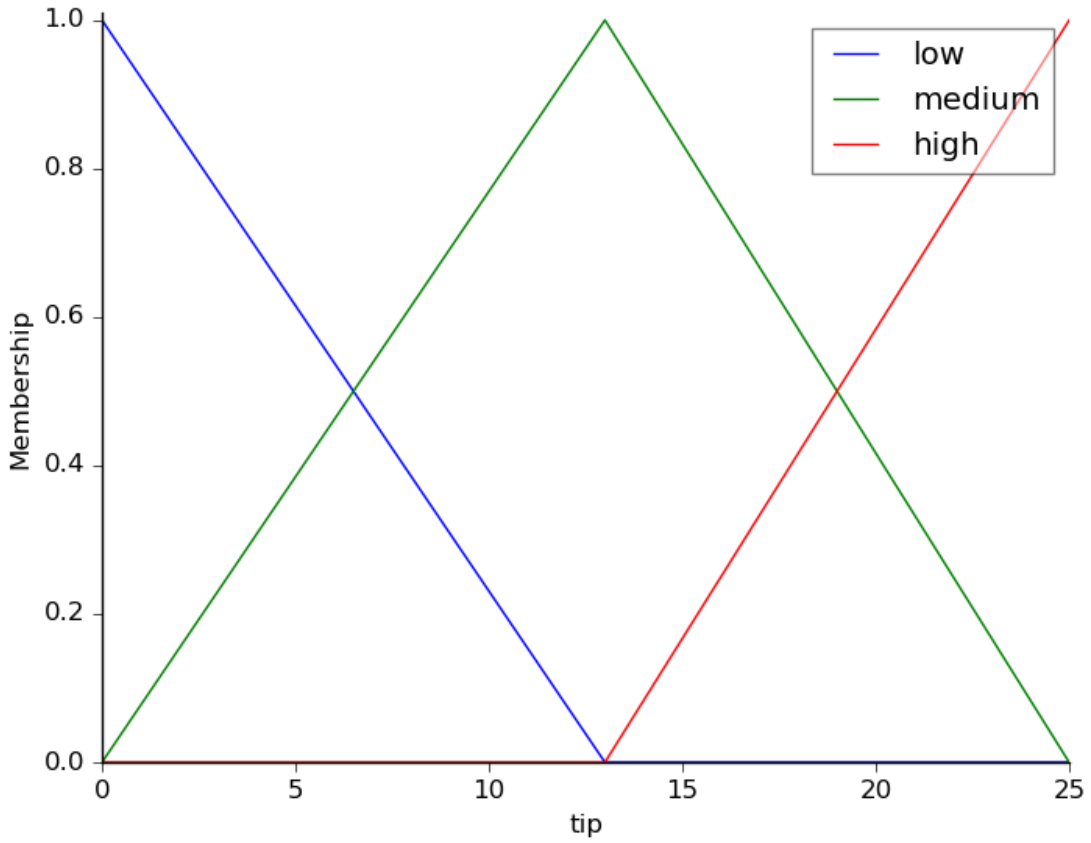
```
# You can see how these look with .view()
quality['average'].view()
```




```
service.view()
```


1.0
0.8
0.6
0.4
0.2
0.0
Membership
0
2
4
6
8
10
service
poor
average
good

```
tip.view()
```



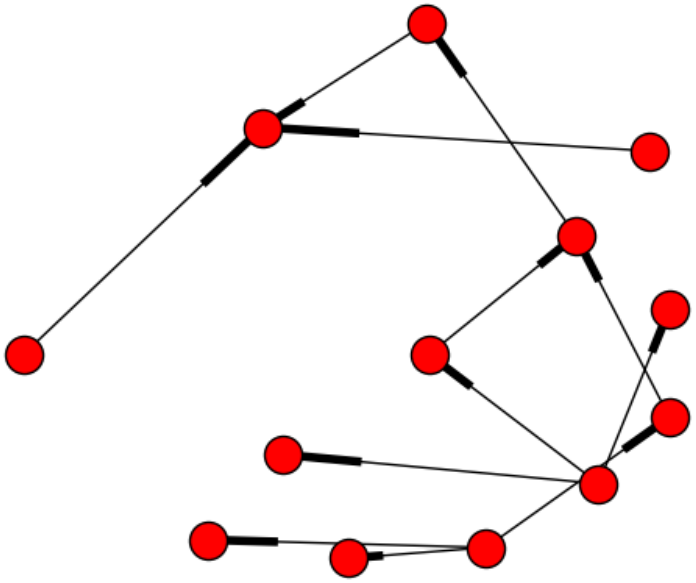
Fuzzy rules

Now, to make these triangles useful, we define the *fuzzy relationship* between input and output variables. For the purposes of our example, consider three simple rules:

1. If the food is poor OR the service is poor, then the tip will be low
2. If the service is average, then the tip will be medium
3. If the food is good OR the service is good, then the tip will be high.

Most people would agree on these rules, but the rules are fuzzy. Mapping the imprecise rules into a defined, actionable tip is a challenge. This is the kind of task at which fuzzy logic excels.

```
rule1 = ctrl.Rule(quality['poor'] | service['poor'], tip['low'])
rule2 = ctrl.Rule(service['average'], tip['medium'])
rule3 = ctrl.Rule(service['good'] | quality['good'], tip['high'])

rule1.view()
```



Control System Creation and Simulation

Now that we have our rules defined, we can simply create a control system via:

```
tipping_ctrl = ctrl.ControlSystem([rule1, rule2, rule3])
```

In order to simulate this control system, we will create a `ControlSystemSimulation`. Think of this object representing our controller applied to a specific set of cirucmstances. For tipping, this might be tipping Sharon at the local brew-pub. We would create another `ControlSystemSimulation` when we're trying to apply our `tipping_ctrl` for Travis at the cafe because the inputs would be different.

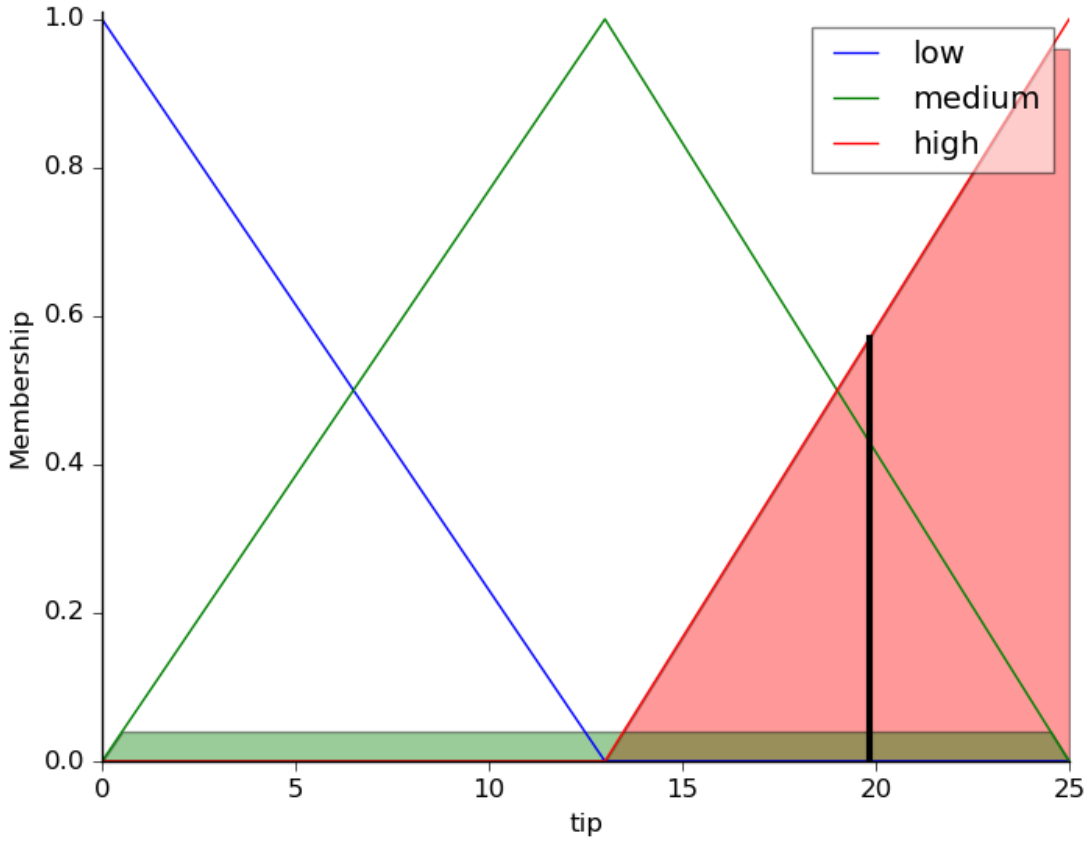
```
tipping = ctrl.ControlSystemSimulation(tipping_ctrl)
```

We can now simulate our control system by simply specifying the inputs and calling the `compute` method. Suppose we rated the quality 6.5 out of 10 and the service 9.8 of 10.

```
# Pass inputs to the ControlSystem using Antecedent labels with Pythonic API
# Note: if you like passing many inputs all at once, use .inputs(dict_of_data)
tipping.input['quality'] = 6.5
tipping.input['service'] = 9.8

# Crunch the numbers
tipping.compute()
```

Once computed, we can view the result as well as visualize it.

```
print tipping.output['tip']
tip.view(sim=tipping)
```



The resulting suggested tip is **20.24%**.

Final thoughts

The power of fuzzy systems is allowing complicated, intuitive behavior based on a sparse system of rules with minimal overhead. Note our membership function universes were coarse, only defined at the integers, but `fuzz.interp_membership` allowed the effective resolution to increase on demand. This system can respond to arbitrarily small changes in inputs, and the processing burden is minimal.

Python source code: `download` (generated using `skimage` 0.2)

Overview Introduction to skfuzzy.	API Reference Documentation for the functions included in skfuzzy.
Installation Steps How to install skfuzzy.	User Guide Usage guidelines.
Contribute Take part in development.	License Info Conditions on the use and redistribution of this package.
Examples Introductory examples	

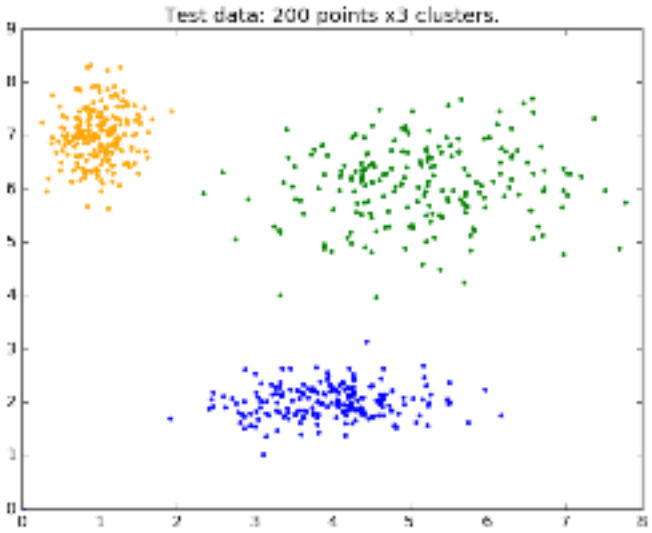


Fig. 1.1: *Fuzzy c-means clustering*

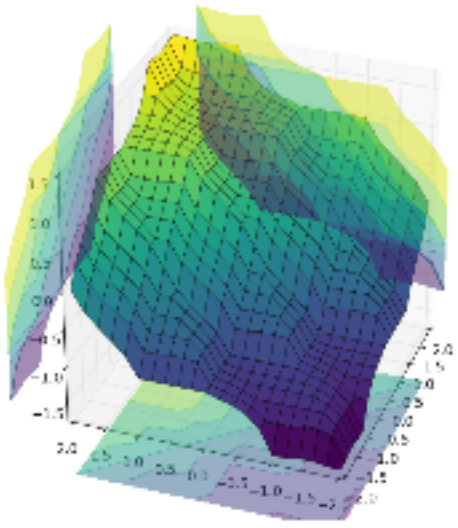


Fig. 1.2: *Fuzzy Control Systems: Advanced Example*

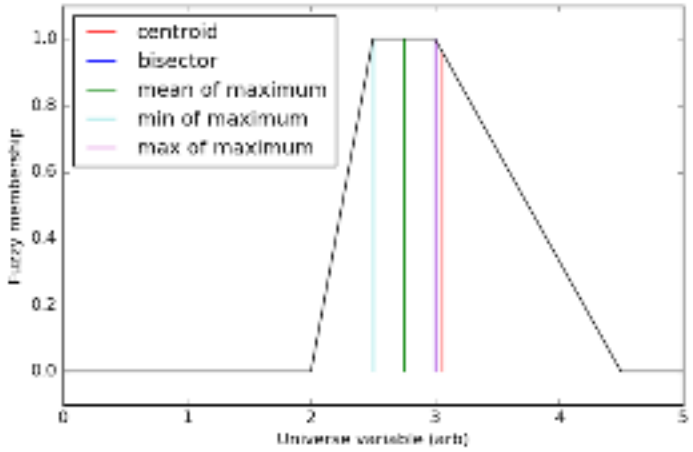


Fig. 1.3: *Defuzzification*

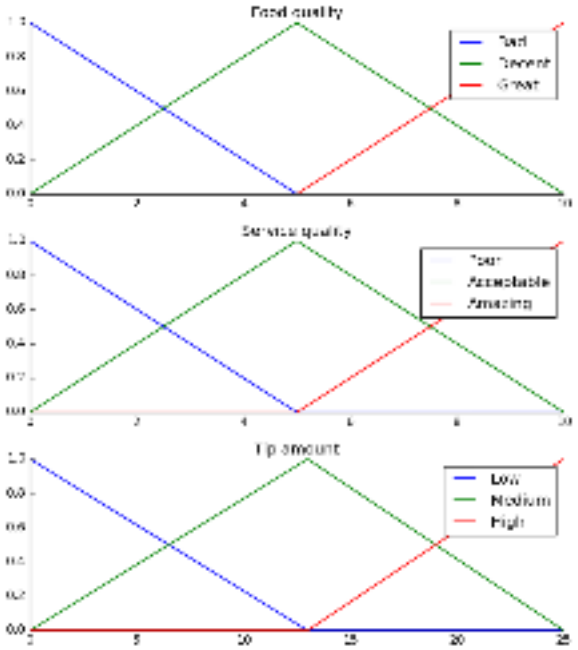


Fig. 1.4: *The Tipping Problem - The Hard Way*

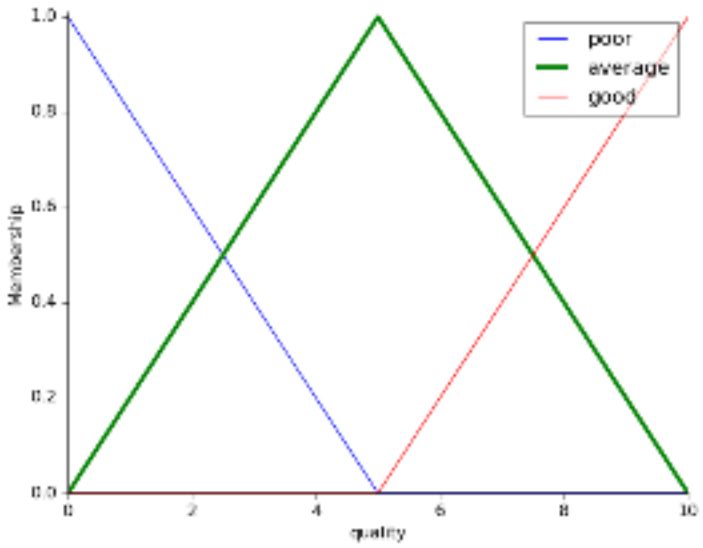


Fig. 1.5: *Fuzzy Control Systems: The Tipping Problem*

CHAPTER 2

Indices and tables

Table of Contents
Lists all sections and subsections.

search
Search this documentation.

genindex
All functions, classes, terms.