27. Means and Medians

Three Instructive Problems:

The Apportionment Problem
The Polygon Averaging Problem
The Median Filtering Problem
What?

The Apportionment Problem

How to fairly distribute 435 Congressional Districts among the 50 states.
<table>
<thead>
<tr>
<th>State</th>
<th>Pop</th>
<th>nDist</th>
<th>Pop/nDist</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alabama</td>
<td>4802982</td>
<td>7</td>
<td>686140</td>
</tr>
<tr>
<td>Alaska</td>
<td>721523</td>
<td>1</td>
<td>721523</td>
</tr>
<tr>
<td>Arizona</td>
<td>6412700</td>
<td>9</td>
<td>712522</td>
</tr>
<tr>
<td>Arkansas</td>
<td>2926229</td>
<td>4</td>
<td>731557</td>
</tr>
<tr>
<td>California</td>
<td>37541989</td>
<td>53</td>
<td>708339</td>
</tr>
<tr>
<td>Colorado</td>
<td>5044930</td>
<td>7</td>
<td>720704</td>
</tr>
<tr>
<td>Connecticut</td>
<td>3581628</td>
<td>5</td>
<td>716325</td>
</tr>
<tr>
<td>Delaware</td>
<td>900877</td>
<td>1</td>
<td>900877</td>
</tr>
<tr>
<td>Florida</td>
<td>18900773</td>
<td>27</td>
<td>700028</td>
</tr>
<tr>
<td>etc</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The Polygon Averaging Problem

Given a polygon, connect the midpoints of the sides. This gives a new polygon. Repeat many times.
A Random Pentagon
The Side Midpoints
Connect the Midpoints
The Polygon Untangles Itself and Heads Towards an Ellipse
What?

The Median Filtering Problem

Visit each pixel in a picture and replace its value by the median value of its "neighbors".
A Picture With Dirt Specks
After Median Filtering is Applied
Why?

Each Problem has a couple of Python “nuggets” to practice with.

Each problem has something to “say” about averaging.

Each problem has a high-level “message”

A Nice Way to Wrap Up
The Apportionment Problem
The Apportionment Problem

How do you distribute 435 Congressional seats among the 50 states so that the ratio of population to delegation size is roughly the same from state to state?

Quite possibly one of the greatest division problems of all time!
Notation

Number of states: \( n \)

State populations: \( p[0], \ldots, p[n-1] \)

State delegation size: \( d[0], \ldots, d[n-1] \)

Total Population: \( P \)

Total number of seats: \( D \)
Ideal: Equal Representation

Number of states: \( n \)
State populations: \( p[0], \ldots, p[n-1] \)
State delegation size: \( d[0], \ldots, d[n-1] \)
Total Population: \( P \)
Number of seats: \( D \)

\[
\frac{P}{D} = \frac{p[0]}{d[0]} = \ldots = \frac{p[49]}{d[49]} 
\]
i.e.,

\[
d[k] = \frac{p[k]}{P} D
\]

And so for NY in 2010..

\[
NY : \frac{19421055}{309239463} \cdot 435 = 27.13
\]

But delegation size must be a whole number!!!
More Realistic...

Number of states: \( n \)

State populations: \( p[0], \ldots, p[n-1] \)

State delegation size: \( d[0], \ldots, d[n-1] \)

Total Population: \( P \)

Number of seats: \( D \)

\[
\frac{P}{D} \approx \frac{p[0]}{d[0]} \approx \ldots \approx \frac{p[49]}{d[49]}
\]
An **Apportionment Method** determines delegation sizes $d[0], \ldots, d[49]$ that are whole numbers so that representation is approximately equal:

\[
\frac{p[0]}{d[0]} \approx \ldots \approx \frac{p[49]}{d[49]}
\]
How it Is Done

Think in terms of dealing cards.

You are the dealer.

You have 435 cards to deal to 50 people.
At the Start

Everybody gets one card...

\begin{verbatim}
N = 435
d = []
for k in range(50):
    d.append(1)
N = N-1
\end{verbatim}

Every state has at least one congressional district
Dealing out the Rest...

while N > 0:
    Let k be the index of that state which is most deserving of an additional district.
    # Increase that state's delegation
    d[k] += 1
    # Decrease what's left to deal
    N = N-1

Several reasonable definitions of "most deserving."
The Method of Small Divisors

At this point in the “card game” deal a district to the state having the largest quotient

\[
\frac{p[k]}{d[k]}
\]

Tends to favor big states
def smallDivisor(p,d):
    """ returns an int j with the property that p[j]/d[j] is max. """

    PreC:p and d are length-50 arrays of ints and the d-entries are pos. """

    m = 0
    for k in range(50):
        if p[k]/d[k] >= m
            m = p[k]/d[k]
            j = k
    return j

This is the old “Look for a max” problem
Dealing out the Rest…

```python
while N > 0:
    k = smallDivisors(p,d)
    # Increase that state’s delegation
    d[k] += 1
    # Decrease what’s left to deal
    N = N-1
```

Several reasonable definitions of “most deserving.”
The Method of Large Divisors

At this point in the “card game” deal a district to the state having the largest quotient

\[
\frac{p(k)}{d(k) + 1}
\]

Tends to favor small states
while N > 0:
    k = largeDivisors(p,d)
    # Increase that state’s delegation
    d[k] += 1
    # Decrease what’s left to deal
    N = N-1
The Method of Major Fractions

At this point in the “card game” deal a district to the state having the largest value of

\[ \frac{1}{2} \left( \frac{p(k)}{d(k)} + \frac{p(k)}{d(k) + 1} \right) \]

Several reasonable definitions of “most deserving.”
Dealing out the Rest...

```python
while N > 0:
    k = majorFractions(p,d)
    # Increase that state's delegation
    d[k] += 1
    # Decrease what's left to deal
    N = N-1
```
The Method of Equal Proportions

At this point in the “card game” deal a district to the state having the largest value of

\[ \sqrt{\frac{p(k) \times p(k)}{d(k) \times d(k) + 1}} \]

This method is in use today.

Compromise via the Geometric Mean
while N > 0:
    k = equalProportions(p,d)
    # Increase that state's delegation
    d[k] += 1
    # Decrease what's left to deal
    N = N-1
Four Different Ways to Compute “Most Deserving”

\[
\frac{p(k)}{d(k)} + \frac{p(k)}{d(k) + 1}
\]

\[
\frac{p(k)}{d(k) + 1}
\]

\[
\frac{1}{2} \left( \frac{p(k)}{d(k)} + \frac{p(k)}{d(k) + 1} \right)
\]

\[
\sqrt{\frac{p(k)}{d(k)} \ast \frac{p(k)}{d(k) + 1}}
\]
Takeaway: There is a Subjective Component to Math+Computing

One can design more equitable methods for apportionment, but they are complicated and cannot be “sold” to the lay public.
Another Division Problem

Gerrymandering: The Art of drawing district boundaries so as to favor incumbents
Polygon Averaging
Connect the Midpoints
class polygon:
    def __init__(self,x,y):
        self.x = x
        self.y = y

x and y are numpy arrays that name the vertices of the polygon:

(x[0],y[0]),..., (x[n-1],y[n-1])
def newPoly(self):
    n = len(self.x); x = zeros(n); y = zeros(n)
    for k in range(n):
        # Get the next midpoint.
        j = (k+1)%n
        x[k] = (self.x[k]+self.x[j])/2
        y[k] = (self.y[k]+self.y[j])/2
    self.x = x
    self.y = y
Order From Chaos

1. Pick n, say n= 30
2. Generate random lists of floats x and y
3. P = polygon(x,y)
4. Then repeatedly replace P by with a new polygon obtained by connecting midpoints:

```python
for i in range(200):
    P.newPolygon()
    P.plotPoly()
```
The Polygon Untangles Itself and Heads Towards an Ellipse

At the Start  After 40 iterations  After 200 iterations
It’s About Repeated Averaging

A midpoint is the average of the endpoints.

\[(a, b) \quad \text{to} \quad (c, d)\]

Midpoint:

\[
\left(\frac{a+c}{2}, \frac{b+d}{2}\right)
\]
Median Filtering
Pictures as Arrays

A black and white picture can be encoded as a 2-dimensional array of numbers.

Typical:

\[ 0 \leq A[i,j] \leq 255 \]

(black) \hspace{2cm} (white)

Values in between correspond to different levels of grayness.
Just a Bunch of Numbers

1458-by-2084

150 149 152 153 152 155
151 150 153 154 153 156
153 151 155 156 155 158
154 153 156 157 156 159
156 154 158 159 158 161
157 156 159 160 159 162
Dirt!

1458-by-2084

Note how the “dirty pixels” look out of place.
Can We Filter Out the “Noise”?
Idea

1458-by-2084

Assign “typical” neighborhood gray values to “dirty pixels”
Getting Precise

“Typical neighborhood gray values”

Could use Median Or Mean

radius 1

radius 3

We’ll look at “Median Filtering” first...
Median Filtering

Visit each pixel.

Replace its gray value by the median of the gray values in the "neighborhood".
Using a radius 1 “Neighborhood”

Before

<table>
<thead>
<tr>
<th>7</th>
<th>7</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

After

<table>
<thead>
<tr>
<th>7</th>
<th>7</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>
How to Visit Every Pixel

for i in range(m):
    for j in range(n):
        Compute new gray value for pixel (i,j).
Replace ⬜ with the median of the values under the window.
Replace \( \times \) with the median of the values under the window.
Replace × with the median of the values under the window.
Original:

\[ i = 0 \]

\[ j = n-1 \]

Filtered:

Replace \( \Box \) with the median of the values under the window.
Original:

\[
i = 1
\]

\[
j = 0
\]

Filtered:

Replace \( \boxed{\times} \) with the median of the values under the window.
Replace with the median of the values under the window.
Original:

\[ i = m - 1 \]
\[ j = n - 1 \]

Filtered:

Replace \( \bullet \) with the median of the values under the window.
What We Need...

(1) A function that computes the median value in a 2-dimensional array $C$:

$$ m = \text{medVal}(C) $$

(2) A function that builds the filtered image by using median values of radius $r$ neighborhoods:

$$ B = \text{medFilter}(A,r) $$
Medians vs Means

A =

150  151  158  159  156
153  151  156  155  151
150  155  152  154  159
156  154  152  158  152
152  158  157  150  157

Median = 154  Mean = 154.2
**Medians vs Means**

\[ A = \]

\[
\begin{array}{cccccc}
150 & 151 & 158 & 159 & 156 \\
153 & 151 & 156 & 155 & 151 \\
150 & 155 & 0 & 154 & 159 \\
156 & 154 & 152 & 158 & 152 \\
152 & 158 & 157 & 150 & 157 \\
\end{array}
\]

Median = **154**  
Mean = **148.2**
for i in range(m):
    for j in range(n):
        Compute new gray value for pixel (i,j).
    end
end
B = medFilter(A)
What About Using the Mean instead of the Median?

Replace each gray value with the average gray value in the radius $r$ neighborhood.
Mean Filter with $r = 3$
Mean Filter with $r = 10$
Why it Fails

The mean does not capture representative values.
And Median Filters Leave Edges (Pretty Much) Alone

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>200</td>
<td>200</td>
<td>200</td>
<td>200</td>
<td>200</td>
<td>200</td>
</tr>
<tr>
<td>200</td>
<td>200</td>
<td>200</td>
<td>200</td>
<td>200</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>200</td>
<td>200</td>
<td>200</td>
<td>200</td>
<td>100</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>200</td>
<td>200</td>
<td>200</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>200</td>
<td>200</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>200</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td></td>
</tr>
</tbody>
</table>

Inside the box, the 200’s stay at 200 and the 100’s stay at 100.
Takeaways

Image processing is all about operations on 2-dimensional arrays.

Simple operations on small patches are typically repeated again and again.

There is a profound difference between the median and the mean when filtering noise.