Before you turn this problem in, make sure everything runs as expected. First, **restart the kernel** (in the menubar, select Kernel→Restart) and then **run all cells** (in the menubar, select Cell→Run All).

Make sure you fill in any place that says `YOUR CODE HERE` or "YOUR ANSWER HERE", as well as your name and collaborators below:

In [1]:
```
NAME = ""
COLLABORATORS = ""
```

CSE 30 Spring 2022 - Homework 8

Instructions

Please disregard the YOUR NAME and COLLABORATORS above. They are put there atomatically by the grading tool. You can find instructions on how to work on a homework on Canvas. Here is a short summary:

Submitting your work

To submit your work:

- First, click on "Runtime > Restart and run all", and check that you get no errors. This enables you to catch any error you might have introduced, and not noticed, due to your running cells out of order.
- Second, download the notebook in .ipynb format (File > Download .ipynb) and upload the .ipynb file to this form. This homework is due on **Tuesday, May 03, at 11:59pm**.

You can submit multiple times; the last submission before the deadline is the one that counts.

Homework format

For each question in this notebook, there is:

- A text description of the problem.
- One or more places where you have to insert your solution. You need to complete every place marked:

  `# YOUR CODE HERE`

  and you should not modify any other place.

- One or more test cells. Each cell is worth some number of points, marked at the top. You should not modify these tests cells. The tests pass if no error is printed out: when there is a statement that says, for instance:

  ```
  assert x == 2
  ```

  then the test passes if `x` has value 2, and fails otherwise. You can insert a `print(x)` (for this case!) somewhere if you want to debug your work; it is up to you.

Notes:

- Your code will be tested both according to the tests you can see (the `assert` statements you can see), *and* additional tests. This prevents you from hard-coding the answer to the particular questions posed. Your code should solve the *general* intended case, not hard-code the particular answer for the values used in the tests.
- **Please do not delete or add cells!** The test is autograded, and if you modify the test by adding or deleting cells, even if you re-add cells you delete, you may not receive credit.
- **Please do not import modules that are not part of the standard library.** You do not need any, and they will likely not available in the grading environment, leading your code to fail.
- **If you are inactive too long, your notebook might get disconnected from the back-end.** Your work is never lost, but you have to re-run all the cells before you continue.
- You can write out print statements in your code, to help you test/debug it. But remember: the code is graded on the basis of what it outputs or returns, not on the basis of what it prints.
- **TAs and tutors have access to this notebook,** so if you let them know you need their help, they can look at your work and give you advice.

Grading

Each cell where there are tests is worth a certain number of points. You get the points allocated to a cell only if you pass *all* the tests in the cell.

The tests in a cell include both the tests you can see, and other, similar, tests that are used for grading only. Therefore, you cannot hard-code the solutions: you really have to solve the essence of the problem, to receive the points in a cell.

Code of Conduct

- Work on the test yourself, alone.

- You can search documentation on the web, on sites such as the Python documentation sites, Stackoverflow, and similar, and you can use the results.
- You cannot share your work with others or solicit their help.

About this homework

This homework notebook has many cells, as it is derived from the chapter, but there are only three questions. Each question is marked

```
### Question n:
```

for $n = 1, 2, 3$. The questions are:

- Implementing a sliding window averagerator
- Implementing a class to clean data streams
- Implementing a class to perform motion detection.

Suppose you have a series of numbers, and you need to compute their average and standard deviation. What is a good way for doing this? The obvious way is to use the numpy library, which offers a wealth of functions to operate on matrices, arrays, and much more. Numpy is one of the fundamental packages of Python, and you would be well advised to browse its documentation and familiarize yourself with what it can do. With numpy, we can compute average and standard deviation of a list of numbers very simply:

In [2]:
```
import numpy as np

s = [1., 2., 3., 3., 2., 4., 3.]
print("avg:", np.average(s))
print("std:", np.std(s))
```

```
avg: 2.5714285714285716
std: 0.9035079029052513
```

Stream statistics

Assume now that the numbers do not form a fixed length sequence, but rather, a stream of numbers, with new numbers always arriving. The numbers could represent real-time temperature measurements, or water pressure, or electricity usage, or percentages of utilized CPU cycles, and so forth. What do we do in order to compute their average and standard deviation?

There are various choices, and the way one does it depends on the application. It is certainly possible to accummulate all numbers, and then compute their overall average and standard deviation; this allows the computation of statistics that apply to the entire time range for which the data was available. More commonly, one is interested in knowing the *recent*

aveage and standard deviation, so that one can compare the most recent data with the average of the last day.

Stream Averagerators

One could implement the code that computes the average of a stream in the same portion of code where one reads the stream, as follows:

In [3]:
```
import random # We use random to simulate a stream.

def read_stream():
    """Reads and returns one number from the stream."""
    return random.random()

def use(x):
    """Code to do something with x"""
    pass

# Here we accummulate the sequence, so we can average it.
seq = []

while True:
    x = read_stream()

    # We add x to the average
    seq.append(x)
    print("avg:", np.average(seq))
    use(x)

    # This is an example, and I don't what the code to run forever.
    if len(seq) == 10:
        break
```

```
avg: 0.9181642831705019
avg: 0.786172656722353
avg: 0.7962596347125319
avg: 0.6963312313905462
avg: 0.7538428552786136
avg: 0.7366690659537761
avg: 0.6873151840413075
avg: 0.7197172172810551
avg: 0.6941266660739573
avg: 0.673728216142046
```

However, this approach is horrible in two different ways. One way is that the implementation is horribly inefficient; our sequence seq will have to hold all the data we read from the stream. This is bad, and we will fix it later.

The other way in which this is horrible is that the code to compute the average is intermingled with the code that reads the sequence and passes it to the code that uses it. It would be much better to separate out the code, for two related reasons.

Separation of concerns. Separating the code makes it easier both to read and to write, because we separate the concerns: when we write the code to compute the average, we can focus on that, disregarding the details of how the stream is read or used; when we write the code that processes the stream, we can focus on that, simply calling a method to compute the average, but disregarding how the average is computed. Separating the concerns, or dividing the overall coding task into smaller, independent units, is key. Each person, at any given time, can keep in mind only a fairly small set of facts; indeed, several studies on software verification point out to the fact that in order to write correct code, programmers usually use no more than a dozen facts about the previous code and input, reflecting what likely is an underlying limitation of our brains.
By focusing on one task at a time, we can apply our full mental powers to that particular task, making it much easier to write its code. The same goes for reading code: it is much easier to understand code that does one specific thing, than code that mixes multiple goals at a time.

Ease of modification. As we mentioned, there are various ways of computing a stream average: there are more and less efficient implementations, and we can consider the entirety of the data read from the stream, or only the most recent one. It will be easier to change the implementation if the code for computing the stream average is all in the same place, rather than sprinkled in multiple places that must be tracked and updated.

For these reasons, we introduce *averagerator* classes that comput running averages and standard deviations. The first we write, *FullAveragerator*, is for computing the statistics of complete sequences.

The class has one method, *add*, used to add data to it, and two properties, *avg* and *std*, which return the average and standard deviation so far.

In [4]:
```
class FullAveragerator(object):

    def __init__(self):
        self.seq = []

    def add(self, x):
        self.seq.append(x)

    @property
    def avg(self):
        return np.average(self.seq)

    @property
    def std(self):
        return np.std(self.seq)
```

The previous code can be rewritten like this:

In [5]:
```
averagerator = FullAveragerator()
```

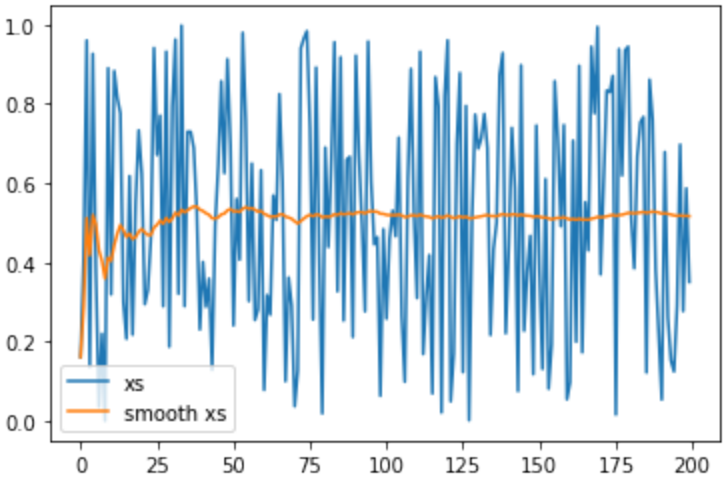
```
for _ in range(10):
    x = read_stream()

    # We add x to the average
    averagerator.add(x)
    print("avg:", averagerator.avg)
    use(x)
```

```
avg: 0.5689294650377672
avg: 0.34147910665997305
avg: 0.3050866558174737
avg: 0.35616678200317553
avg: 0.4440077192084759
avg: 0.44813369953271226
avg: 0.41100076031124994
avg: 0.4762151724632172
avg: 0.4659943545929512
avg: 0.455099595111633
```

The improvement seems minor, but this is only because our averagerator, as written is very simple. Very simple, and very inefficient, as remarked. Let us write it more efficiently.

A picture is worth a thousand words, so let's draw one.

In [6]:
```
import matplotlib.pyplot as plt

averagerator = FullAveragerator()

xs = []
smooth_xs = []
for _ in range(200):
    x = read_stream()
    xs.append(x)
    averagerator.add(x)
    smooth_xs.append(averagerator.avg)
plt.plot(xs, label="xs")
plt.plot(smooth_xs, label="smooth xs")
plt.legend()
plt.show()
```



Efficient Stream Averagerator

The idea in computing a more efficient implementation is to avoid storing the entire sequence, summarizing it instead by aggregate statistics. The average $E[X]$ of a sequence of n numbers $x_1, x_2, \ldots, x_n$ can be computed as S_n/n, where

$$S_n = \sum_{i=1}^{n} x_i \ .$$

When x_{n+1} arrives, all we need to do is compute $S_{n+1} = x_{n+1} + S_n$, and return the average $S_{n+1}/(n+1)$.

Thus, we do not need to store the complete sequence to compute the average: we need to store only the sequence length (n above), and the sequence sum (S_n above).

In [7]:
```
class EfficientFullAveragerator(object):

    def __init__(self):
        self.sum_x = 0.
        self.n = 0

    def add(self, x):
        self.sum_x += x
        self.n += 1

    @property
    def avg(self):
        return self.sum_x / self.n
```

This works, but what about the standard deviation?

The standard deviation is the square root of the variance, and for a series $X = x_1, x_2, \ldots, x_n$ of numbers, the variance is

$$E[(X - E[X])^2] = E[(X - \mu)^2]$$

where $\mu = E[X]$ is the average. This equation does not tell us directly what to store. We cannot compute the average of $x_1 - \mu, x_2 - \mu, \ldots, x_n - \mu$, because μ is not known when x_1 arrives: μ depends on the *entire* sequence, so when its first element x_1 arrives, μ is not known yet!

To obtain a form that we can compute on the fly, we need to develop the above equation.

$$E[(X - \mu)^2] = E[X^2 - 2\mu X + \mu^2] = E[X^2] - 2\mu E[X] + \mu^2 = E[X^2] - \mu^2 \ ,$$

where we have used that $E[X] = \mu$. The relation

$$E[(X - \mu)^2] = E[X^2] - \mu^2 \ ,$$

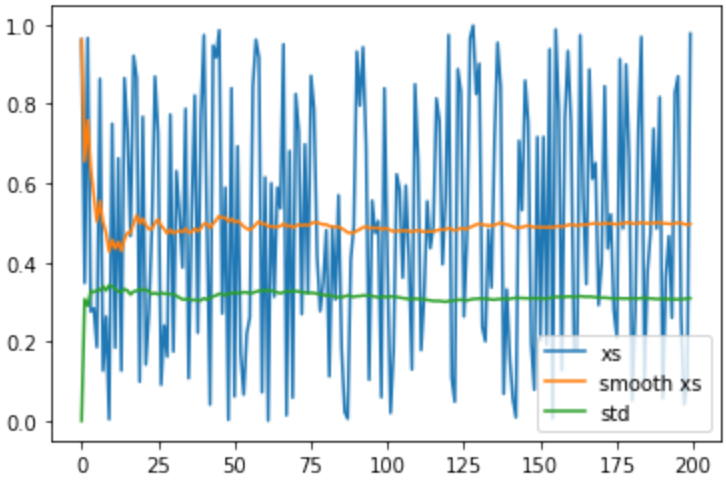
is suitable to be computed on the fly. It is just the average of the sequence of squares $x_1^2, x_2^2, \ldots$ (and we already know how to compute sequence averages), minus μ^2, which we also already know how to compute.

Using these ideas, here is the complete implementation of our efficient averagerator class.

In [8]:
```
class EfficientFullAveragerator(object):

    def __init__(self):
        self.sum_x = 0.
        self.sum_x_sq = 0.
        self.n = 0

    def add(self, x):
        # We compute the sum of the x, to compute their average.
        self.sum_x += x
        # Sum of the x^2, so we can later compute the average of the x^2.
        self.sum_x_sq += x * x
        self.n += 1

    @property
    def avg(self):
        return self.sum_x / self.n

    @property
    def std(self):
        mu = self.avg # To avoid calling self.avg twice.
        return np.sqrt(self.sum_x_sq / self.n - mu * mu)
```

Let us play with this implementation.

In [9]:
```
averagerator = EfficientFullAveragerator()

xs = []
smooth_xs = []
stdevs = []
for _ in range(200):
    x = read_stream()
    xs.append(x)
    averagerator.add(x)
    smooth_xs.append(averagerator.avg)
    stdevs.append(averagerator.std)
plt.plot(xs, label="xs")
plt.plot(smooth_xs, label="smooth xs")
plt.plot(stdevs, label="std")
plt.legend()
plt.show()
```



This works, and we can see that the average tends to 0.5, and the standard deviation tends to $1/\sqrt{12}$. This because the input numbers are uniformy distributed between 0 and 1, and:

$$\int_0^1 x\,dx = \frac{1}{2}\,, \qquad \int_0^1 \left(x - \frac{1}{2}\right)^2 dx = \frac{1}{12}\,.$$

Sliding Windows Averagerators

Often, we are more interested in the recent average than in the average since the start of a stream. This is especially true if we plan to use the average and standard deviation to identify outliers (possibly incorrect data) in a data stream.

Consider, for instance, a temperature sensor giving us readings of outside air temperature once per minute. If one considers statistics that span more than one year, a location might have an average temperature of 15 (Celsius; all temperatures in the following are in Celsius), and a standard deviation of 15, accommodating Winter temperatures slightly above freezing and Summer ones around 30C.

Yet, if we saw input data:

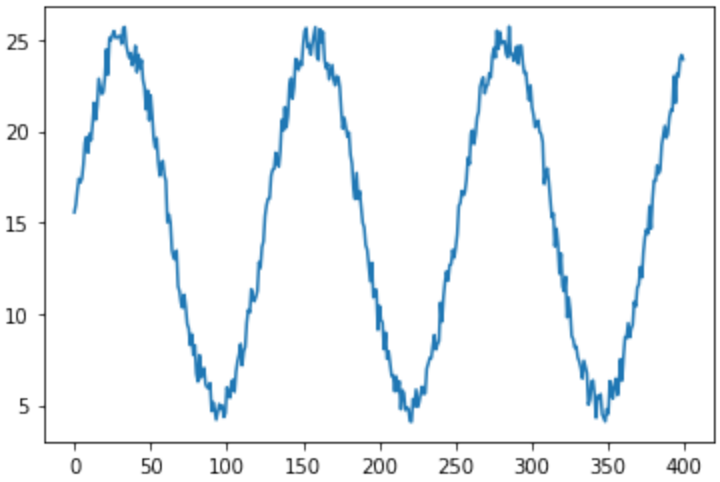
$$12.3, 12.3, 12.4, 12.3, 12.4, 12.4, 23.3, 17.5, 12.4, 12.5, 12.6$$

we should be suspicious: outside air temperature does not change by more than 10C in a minute. We do not know what happened --- someone touched the temperature sensor with a finger, perhaps --- but we know that the data is probably not reflective of air temperature.

As a concrete example, let us simulate a sensor that senses air temperature. Air temperature varies between 5C and 25C, as a sine wave (just to make it easy to draw); the sensor has a noise of $\pm 1C$ on each measurement.

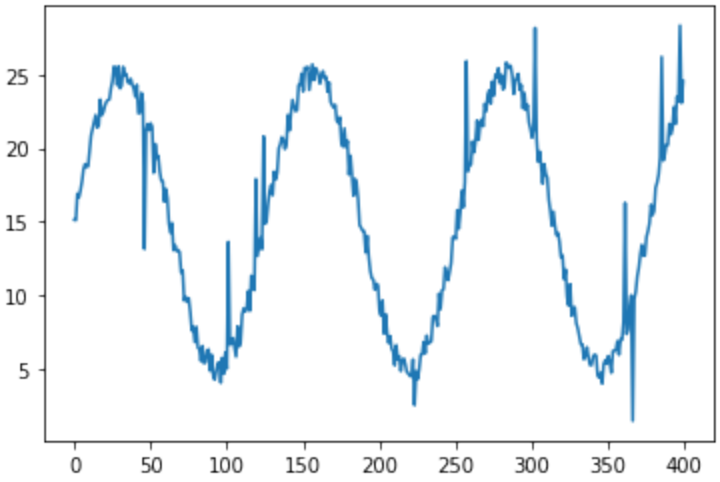
In [10]:
```
def noisy_temp(noise=1., d=0.05):
    t = -d # time
    while True:
        t += d # We increment time.
```

```
        yield 15. + 10. * np.sin(t) + noise * 2.  * (random.random() - 0.5)

# Let's show how this looks.
xs = []
for x in noisy_temp():
    xs.append(x)
    if len(xs) == 400:
        break
plt.plot(xs)
plt.show()
```



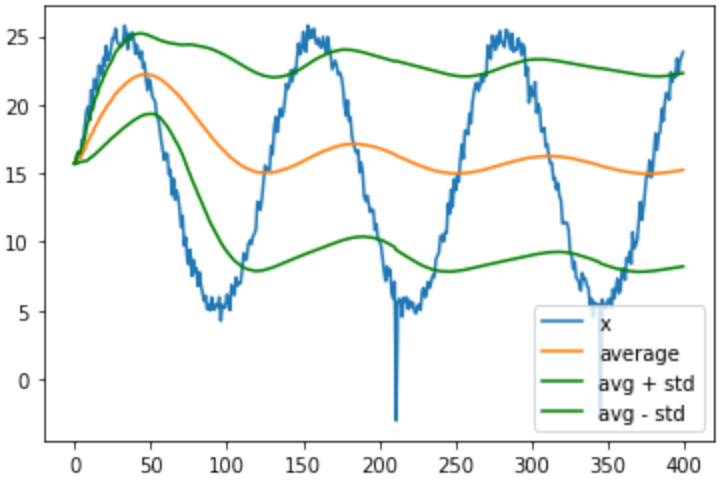
Let us construct a version of this signal with outliers, where once in 50 readings, about, the sensor has an error that can be up to 10C.

In [11]:
```
def noisy_temp_with_outliers(noise=1., d=0.05, outlier_prob=0.02, outlier_size=10.)
    t = -d # time
    while True:
        t += d # We increment time.
        x = 15. + 10. * np.sin(t) + noise * 2.  * (random.random() - 0.5)
        # Adds the outlier, with a certain probability.
        if random.random() < outlier_prob:
            x += outlier_size * 2. * (random.random() - 0.5)
        yield x

# Let's show how this looks.
xs = []
for x in noisy_temp_with_outliers():
    xs.append(x)
    if len(xs) == 400:
        break
plt.plot(xs)
plt.show()
```



Let us compare these outliers, with the average and standard deviation of the whole series.

In [12]:
```
xs = []
avgs = []
stds = []
a = EfficientFullAveragerator()
for x in noisy_temp_with_outliers():
    xs.append(x)
    a.add(x)
    avgs.append(a.avg)
    stds.append(a.std)
    if len(xs) == 400:
        break
plt.plot(xs, label='x')
plt.plot(avgs, label='average')
# Let's move to numpy to compute average plus and minus standard deviation.
a_avg = np.array(avgs)
a_std = np.array(stds)
plt.plot(a_avg + a_std, label='avg + std', color='g')
plt.plot(a_avg - a_std, label='avg - std', color='g')
plt.legend()
plt.show()
```



As we see:

- The temperature, even when read without outliers, often differs from the average by more than the overall standard deviation, and this simply due to the daily temperature variations.
- The outlier themselves, even though quite visible to our eye, often differ from the average by *less* than the standard deviation, just because the standard deviation is really rather large, as it is influenced not only by sensor noise, but also by the daily temperature cycle.

To detect these outliers, and change in conditions, it is far more useful to have the average and standard deviation of *recent* data only, rather than computed over the whole series. A *sliding window averagerator* considers, in the computation of averages and standard deviations, only the most recent N data values, for a specified N.

Question 1: Sliding Window Averagerator

Complete the code below, defining a sliding window averagerator. The class should have methods:

- `__init__(self, window_size)` to initialize the method and specify the window size;
- `add(self, x)`, to add a value

as well as properties `avg` and `std`, as in the previous averagerator classes.

In [13]:
```
### Question 1: Implement a `SlidingWindowAveragerator`

class SlidingWindowAveragerator(object):

    # YOUR CODE HERE
    def __init__(self, window_size):
        self.sum = 0
        self.n = 0
        self.wS = window_size
        self.q = []
        self.tq = []

    def add(self, x):
        if len(self.q) == self.wS:
            self.q.pop(0)
        self.q.append(x)
        self.sum = sum(self.q)
        self.n = len(self.q)
        self.tq = []

    @property
    def avg(self):
        return self.sum / self.n

    @property
    def std(self):
```

```
        avg = self.avg
        for x in self.q:
            self.tq.append((x - avg) ** 2)
        return np.sqrt(sum(self.tq) / self.n)
```

In [14]:
```
### 5 points: Tests for `SlidingWindowAveragerator`

# First some simple cases.
sa = SlidingWindowAveragerator(20)
for _ in range(10):
    sa.add(10)
    assert sa.avg == 10
    assert sa.std == 0

sa = SlidingWindowAveragerator(10)
for _ in range(10):
    sa.add(4)
assert sa.avg == 4
for _ in range(10):
    sa.add(8)
assert sa.avg == 8
assert sa.std == 0
```

In [15]:
```
# 10 points. Then, for more complex cases.

sa = SlidingWindowAveragerator(10)
for _ in range(10):
    sa.add(1)
    assert sa.avg == 1
    assert sa.std == 0

sa = SlidingWindowAveragerator(4)
for _ in range(4):
    sa.add(3)
assert sa.avg == 3
for _ in range(4):
    sa.add(2)
assert sa.avg == 2
assert sa.std == 0
for _ in range(4):
    sa.add(1)
assert sa.avg == 1
assert sa.std == 0

sa = SlidingWindowAveragerator(4)
for x in range(10):
    sa.add(x)
    if x < 4:
        assert sa.avg == x / 2
    else:
        assert sa.avg == (2 * x - 3) / 2

sa = SlidingWindowAveragerator(5)
for x in range(10):
    sa.add(x)
```

```
    if x < 5:
        assert sa.avg == x / 2
    else:
        assert sa.avg == (2 * x - 4) / 2
        assert abs(sa.std - np.sqrt(2)) < 0.001
```

In [16]:
```
### 10 points: Now for even more complex tests.

sa = SlidingWindowAveragerator(10)
for i in range(10):
    sa.add(i)
assert sa.avg == 4.5
assert abs(sa.std - 2.87) < 0.1
for i in range(10):
    sa.add(i)
assert sa.avg == 4.5
assert abs(sa.std - 2.87) < 0.1
for _ in range(10):
    sa.add(1)
assert sa.avg == 1
assert sa.std == 0
```

Duck Typing

You may wonder: should we have not defined an abstract *Averagerator* class, and make all these classes, such as *FullAveragaerator*, *EfficientFullAveragerator*, *SlidingWindowAveragerator*, subclasses of the superclass?

If we were in a strongly typed language, such as Java, the answer would be a resounding Yes. In Java, a superclass serves as the common type of all objects belonging to the more specialized classes. One can then define a method accepting a superclass, say, an *Averagerator*, and then pass to it objects of any of its subclasses.

In Python, objects are rarely tested for the class to which they belong. The more common pattern in Python is simply the one of calling methods of objects, assuming the methods do the proper thing. This approach to type checking (or the lack of it) is sometimes called *duck typing:* if it quacks like a duck, and it waddles like a duck, it is a duck --- meaning, if the object's methods do the right thing, that suffices for us.

Thus, in Python, except in special cases, the subclass relationship is useful especially if there is non-trivial shared code between a subclass and its superclass. This not being the case for the Averagerator classes we have defined so far, we have preferred the simpler approach of defining each class individually, which has the advantage of keeping all the class code in the same place.

Discounting Averagerators

A sliding window abruptly truncates the past: the stream values go from being considered fully considered as part of the average, to being disregarded, in one step. A consequence of this is that to implement a sliding window average of size N, we actually need to store N values: otherwise, we would not know how to remove a value from the sliding window when the value "falls off" the window. Can we do better? Can we obtain something similar to a sliding window average, but that forgets past values in a smoother way, rather than with an abrupt threshold, and such that the amount of data to remember is independent on window size?

The answer is Yes. Given a data stream $x_0, x_1, \ldots, x_n$, the idea is to give to the most recent value x_n a weight of 1, to x_{n-1} a weight of α for $\alpha < 1$, to x_{n-2} weight α^2, and so forth: a value that ocurred k "times" ago has weight α^k. This approach is known as *discounting:* it is as if the value of the past accumulated experience decreased by a factor of α upon the arrival of a new data value.

Aside from being a smoother way to average (the effect of past values slowly decays, rather than abruptly dropping out of a fixed size window), discounted averages are also far more efficient to implement. The idea, for the average, consists in keeping the running sum of values S, and the running sum of weights W.
The average is simply the sum divided by the total weight, or S/W. When a new value x arrives, S and W are updated by first discounting their current values by α, and then adding the contribution of the last value:

$$S := \alpha S + x \qquad W := \alpha W + 1 \ .$$

For the computation of the variance, we proceed in similar fashion. The implementation is below.

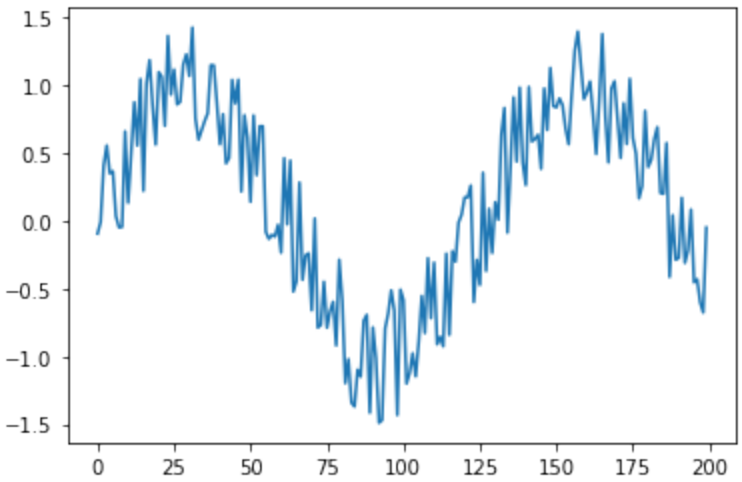
In [17]:
```
class DiscountedAveragerator:

    def __init__(self, alpha):
        """Creates an averagerator with a specified discounting factor alpha."""
        self.alpha = alpha
        self.w = 0.
        self.sum_x = 0.
        self.sum_x_sq = 0.

    def add(self, x):
        self.w = self.alpha * self.w + 1.
        self.sum_x = self.alpha * self.sum_x + x
        self.sum_x_sq = self.alpha * self.sum_x_sq + x * x

    @property
    def avg(self):
        return self.sum_x / self.w

    @property
    def std(self):
        mu = self.avg
```

```
        # The np.maximum is purely for safety.
        return np.sqrt(np.maximum(0., self.sum_x_sq / self.w - mu * mu))
```

Data Smoothing

Running averages can be used for smoothing data. If you have some background in digital signal theory, a discounted average is a digital filtering operation, whose behavior in the frequency domain can be modeled also with the help of its *z-transform* $1/(1-\alpha/z)$. We will be content here with watching it at work. Let's build a stream where there is a sinusoidal signal, with superimposed noise. Here, an iterator comes in handy.

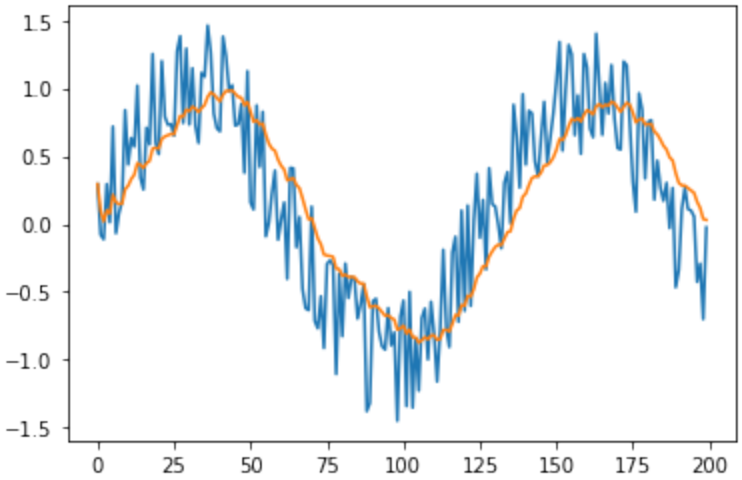
In [18]:
```
def noisy_sin(noise=1.):
    d = 0.05 # Time increment.
    t = -d # time
    while True:
        t += d # We increment time.
        yield np.sin(t) + noise * (random.random() - 0.5)


# Let's display it.
xs = []
for x in noisy_sin():
    xs.append(x)
    if len(xs) == 200:
        break
import matplotlib.pyplot as plt
plt.plot(xs)
plt.show()
```



Let's apply now our smoothing average with $\alpha = 0.9$, and compare raw and smoothed data.

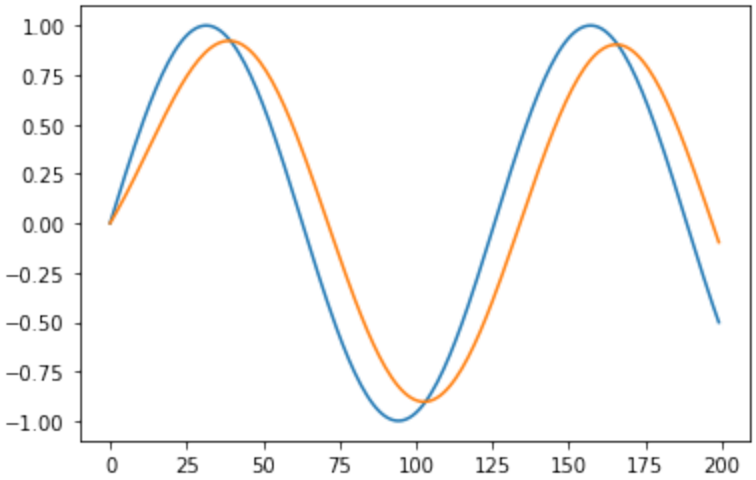
In [19]:
```
xs = []
smooth_xs = []
a = DiscountedAveragerator(0.9)
for x in noisy_sin():
    xs.append(x)
    a.add(x)
    smooth_xs.append(a.avg)
```

```
    if len(xs) == 200:
        break
import matplotlib.pyplot as plt
plt.plot(xs)
plt.plot(smooth_xs)
plt.show()
```



We see that the output is a smoother, time-delayed, and somewhat contracted (multiplied by a factor smaller than 1) verson of the input. The time delay and contraction are due to the fact that the average mixes present with past of the sine wave, and would be present even in absence of noise.

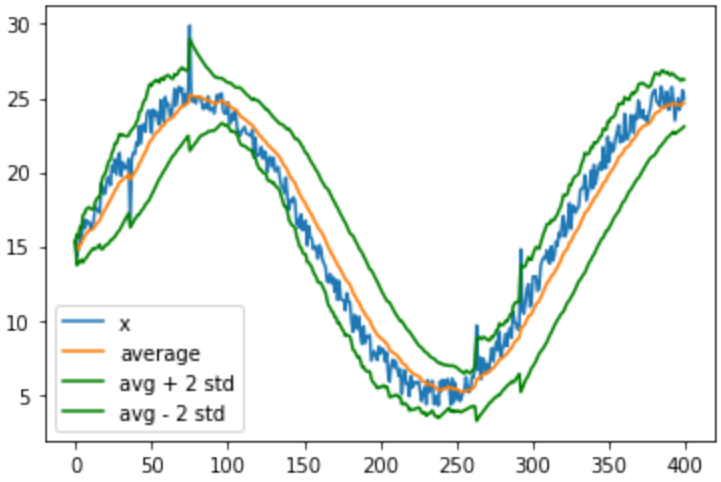
In [20]:
```
xs = []
smooth_xs = []
a = DiscountedAverageator(0.9)
for x in noisy_sin(noise=0.):
    xs.append(x)
    a.add(x)
    smooth_xs.append(a.avg)
    if len(xs) == 200:
        break
import matplotlib.pyplot as plt
plt.plot(xs)
plt.plot(smooth_xs)
plt.show()
```



Outlier Detection

Let us return to our noisy temperature sensor. How can we detect the outlier errors? One simple idea consists in calling an outlier any point that differs from the average by more than, say, two standard deviations. Let us see how this approach would work.

In [21]:
```
a = DiscountedAverageratorator(0.9)

xs = []
avgs = []
stds = []
for x in noisy_temp_with_outliers(d=0.02):
    xs.append(x)
    a.add(x)
    avgs.append(a.avg)
    stds.append(a.std)
    if len(xs) == 400:
        break
plt.plot(xs, label='x')
plt.plot(avgs, label='average')
# Let's move to numpy to compute average plus and minus standard deviation.
a_avg = np.array(avgs)
a_std = np.array(stds)
plt.plot(a_avg + 2. * a_std, label='avg + 2 std', color='g')
plt.plot(a_avg - 2. * a_std, label='avg - 2 std', color='g')
plt.legend()
plt.show()
```



Indeed, this approach would be able to detect most of the large outliers. We can use this idea to define a cleaned version of the data: when a reading is further away than two standard deviations from the average, we replace the reading. Let us define a *CleanData* class that does it for us.

Exercise: Use the averagerator to write a class that counts how many spikes there are in the last `n` time units, where `n` is a parameter.

Question 2: Implementing a CleanData class

Complete the following implementation, in which an averagerator is used in order to replace values that are more than num_stds away from the average, with the average itself.

The `CleanData` class is initialized by passing a discount factor for its averagerator. Every piece `x` of data is then filtered via a call to `filter(x, num_stdevs)`; this call returns:

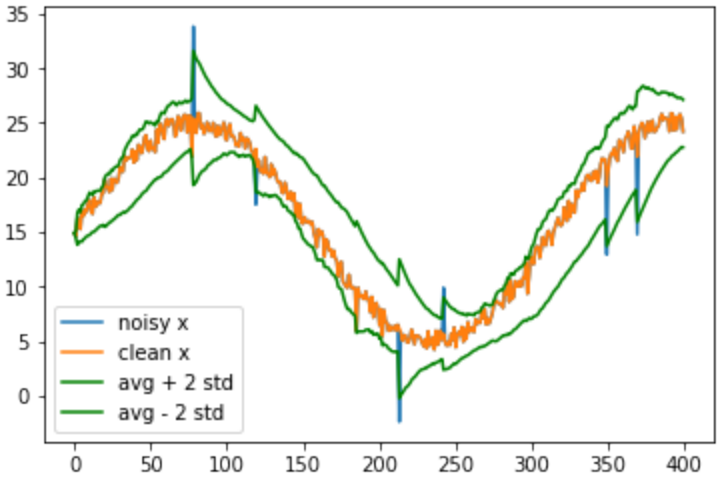
- `x` if the value of `x` is closer than `num_stdevs` standard deviations from the running average,
- the running average if the value of `x` differs from the running average by more than `num_stdevs` standard deviations.

In [22]:
```
### Question 2: Implement the `CleanData` class

class CleanData(object):

    def __init__(self, discount_factor):
        """
        @param discount_factor: discount factor for the averagerator.
        """
        # YOUR CODE HERE
        self.d = discount_factor
        self.davg = DiscountedAveragerator(self.d)
```

```
    def filter(self, x, num_stdevs=2.):
        """Returns a filtered value for x.
        @param x: the value to be filtered.
        @param num_stdevs: number of standard deviations from the average
           beyond which data is rejected.
        It can be done in about 5 lines of code; no worries if your solution is a b
        """
        # YOUR CODE HERE
        self.davg.add(x)
        if x > self.davg.avg - self.davg.std * 2 and x < self.davg.avg + self.davg.
            return x
        else:
            return self.davg.avg
```

Let us see how it works, visually:

In [23]:
```
a = DiscountedAveragerator(0.9)
xs = []
clean_xs = []
avgs = []
stds = []
cleaner = CleanData(0.9)
for x in noisy_temp_with_outliers(d=0.02):
    xs.append(x)
    a.add(x)
    avgs.append(a.avg)
    stds.append(a.std)
    clean_xs.append(cleaner.filter(x, num_stdevs=2))
    if len(xs) == 400:
        break
plt.plot(xs, label='noisy x')
plt.plot(clean_xs, label='clean x')
# Let's move to numpy to compute average plus and minus standard deviation.
a_avg = np.array(avgs)
a_std = np.array(stds)
plt.plot(a_avg + 2. * a_std, label='avg + 2 std', color='g')
plt.plot(a_avg - 2. * a_std, label='avg - 2 std', color='g')
plt.legend()
plt.show()
```



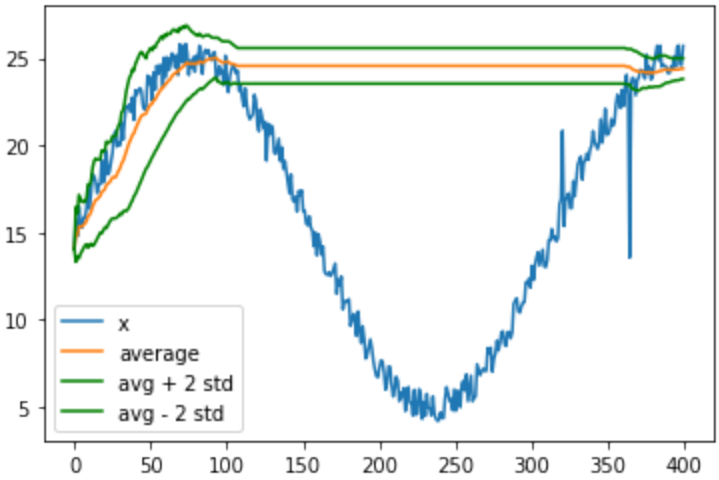
And let us put it through some tests.

In [24]:
```
### 10 points: Tests for `CleanData`
import numpy as np

a = np.zeros(10)
a[3] = 1
a[8] = 10
c = CleanData(0.9)
aa = [c.filter(x) for x in a]
assert max(aa) < 2.

a = np.zeros(100)
a[13] = 10
a[14] = 10
a[18] = 10
a[50] = 10
c = CleanData(0.95)
aa = [c.filter(x, num_stdevs=2) for x in a]
assert aa[13] < 10
assert 1 < aa[14] < 2
assert aa[18] == 10
assert aa[50] < 10
```

An alternative idea that seems promising at first thought is to include in the computation of the running average and standard deviation only points that are not outliers. Let us play with the approach.

In [25]:
```
a = DiscountedAveragerator(0.9)

xs = []
avgs = []
stds = []
for x in noisy_temp_with_outliers(d=0.02):
    xs.append(x)
    if len(xs) >= 20:
        # We need enough data to be able to rely on the statistics.
        a_avg, a_std = a.avg, a.std
        x_min, x_max = a_avg - 2 * a_std, a_avg + 2. * a_std
        if x_min < x < x_max:
            # The data is good.
            a.add(x)
    else:
        # We add all data until we have reliable statistics.
        a.add(x)
    avgs.append(a.avg)
    stds.append(a.std)
    if len(xs) == 400:
        break

plt.plot(xs, label='x')
plt.plot(avgs, label='average')
# Let's move to numpy to compute average plus and minus standard deviation.
a_avg = np.array(avgs)
```

```
a_std = np.array(stds)
plt.plot(a_avg + 2. * a_std, label='avg + 2 std', color='g')
plt.plot(a_avg - 2. * a_std, label='avg - 2 std', color='g')
plt.legend()
plt.show()
```



We see that the problem with this approach is that, should the signal change behavior or become more noisy, or drift from its previous range, we risk disregarding all future data. Better play it safe and include in the statistics all input, in case what we think of as an outlier is really the first in a series of data with higher noise or drift in them.

This is one more case in point supporting the author's motto: *when in doubt, be stupid*.

It is often better to take a simpler, more robust approach (in this case, averaging all data) than to try to be smart without understanding all the facets of a problem (in this case, trusting our simple outlier detection to the point of letting it screen even the data we feed to it).

Motion Detection

We can use our averagerators to perform motion detection in a sequence of images captured by a webcam. The idea is this. Each image will be represented as a H x W x 3 3-d array; H and W are the image height and width, respectively, and 3 is the number of color channels of a RGB image.

We will compute the discounted average and standard deviations of *every single color pixel* in the image. If a pixel has a value that is outside the interval $[\mu - k\sigma, \mu + k\sigma]$, where μ is the pixel average σ is the pixel standard deviation, we detect motion. Here, k is a sensitivity threshold that specifies how many standard deviations must separate the value of a pixel from its average for us to detect motion.

Computing mean and standard deviation of every pixel sounds like a crazy idea, until we realize that our DiscountedAveragerator essentially does it for us already.

So far, we have used the DiscountedAveragerator by passing to it a scalar, that is, a floating point number. If we pass to it a value of *x* which is a Numpy array, everything works: Numpy will re-interpret our $+$, $-$, $*$ operators as operators betwen arrays, and compute mean and standard deviation as *arrays*, one entry per color pixel, rather than scalars.

To experiment with motion detection, let us get a series of images captured by a webcam, and convert each image to a numpy matrix.
We will obtain a list of numpy matrices.

In [26]:
```
from PIL import Image
import requests
from zipfile import ZipFile
from io import BytesIO
```

In [27]:
```
# Gets the zip file.
ZIP_URL = "https://storage.googleapis.com/lucadealfaro-share/GardenSequence.zip"
r = requests.get(ZIP_URL)
# List of images, represented as numpy arrays.
images_as_arrays = []
# Makes a file object of the result.
with ZipFile(BytesIO(r.content)) as myzip:
    for fn in myzip.namelist():
        with myzip.open(fn) as my_image_file:
            img = Image.open(my_image_file)
            # Converts the image to a numpy matrix, and adds it to the list.
            images_as_arrays.append(np.array(img).astype(np.float32))
```

Each numpy 3-d array has shape (Y, X, 3), where Y and X are the dimensions of the image (480 x 640 in our case), and 3 correspons to the three color channels.

In [28]:
```
print(images_as_arrays[0].shape)
```

```
(480, 640, 3)
```

In [29]:
```
print(images_as_arrays[0][10, 20, 2])
```

```
255.0
```

We can then construct a `MotionDetection` class. Internally, it will initialize a discounted averagerator.

We will feed images to `MotionDetection`, one by one; the images will be of size $h \times w \times c$, where h is the height, w the width, and c the color depth: in our case, $480 \times 640 \times 3$ (but please, write you class without hardcoding h and w).

As we feed each image, `MotionDetection` computes which pixels of the image have one of the 3 color channels that are outside the $\mu \pm \kappa\sigma$ interval, where μ is the average, σ is the standard deviation, and κ is a parameter; we will use $\kappa = 4$ in our experiments, thus detecting motion if values deviate from the average by more than 4 standard deviations. The result is a $h \times w \times c$ boolean matrix filled with True/False values.

To perform the above check, you can use a trick: if `a` and `b` are Numpy arrays of the same size, then `a > b` returns an array of the same size, filled with True and False:

In [30]:
```
a = np.random.random((4, 5, 3))
b = np.random.random((4, 5, 3))
print("a:", a)
print("b:", b)
print("a > b:", a > b)
```

```
a: [[[5.68662928e-01 4.33606309e-01 5.33160693e-01]
  [8.58903588e-01 9.73121491e-02 7.73924516e-02]
  [8.62070982e-01 4.39370528e-01 2.66555284e-01]
  [3.21646260e-01 8.29972257e-01 7.14757725e-01]
  [4.71533314e-01 9.41650160e-02 1.63568796e-02]]

 [[4.70384213e-01 1.31572220e-02 1.59188902e-01]
  [9.61293316e-01 3.30628207e-01 7.45465509e-02]
  [4.63288536e-01 3.89458201e-01 2.64292415e-01]
  [6.20669853e-01 2.31803215e-01 3.09444528e-02]
  [4.80637532e-01 5.37748537e-01 2.91150449e-01]]

 [[3.61717240e-01 3.78939065e-01 8.79000212e-02]
  [9.80010126e-02 4.50221875e-01 2.72479619e-01]
  [5.16359227e-01 2.79954715e-01 2.82315981e-01]
  [2.95989795e-01 3.04971356e-01 8.07180794e-03]
  [4.67092330e-01 8.72490037e-01 4.37160189e-01]]

 [[6.84913321e-01 4.17080429e-01 2.45450775e-01]
  [9.38917730e-01 7.07355951e-01 8.57505832e-01]
  [9.51793159e-01 2.91353417e-02 7.94541049e-01]
  [7.87694664e-01 6.79447428e-01 2.74782069e-03]
  [9.09077272e-01 4.84405685e-01 7.11951372e-04]]]
b: [[[0.342244   0.4072814  0.8889835 ]
  [0.1777604  0.34607805 0.36156972]
  [0.87166505 0.06635518 0.31805425]
  [0.98765432 0.92861599 0.53105421]
  [0.35467153 0.17267953 0.1818895 ]]

 [[0.54846653 0.62677822 0.61474537]
  [0.60114367 0.84968289 0.82008775]
  [0.23795863 0.43759472 0.16888499]
  [0.98442816 0.37971328 0.92640669]
  [0.56802578 0.76907544 0.03241006]]

 [[0.96776551 0.75775362 0.77347261]
  [0.77268534 0.99860747 0.58047339]
  [0.96090701 0.49580211 0.95644682]
  [0.64170371 0.64716715 0.83787259]
  [0.19716642 0.16295338 0.23295701]]

 [[0.46198072 0.44332072 0.57632591]
  [0.42027088 0.30443692 0.16522991]
  [0.59671183 0.94386564 0.06831712]
  [0.79603433 0.2867405  0.54534295]
  [0.41554709 0.09161967 0.20157826]]]
a > b: [[[ True  True False]
  [ True False False]
  [False  True False]
  [False False  True]
  [ True False False]]

 [[False False False]
  [ True False False]
  [ True False  True]
  [False False False]
```

```
 [False False  True]]

[[False False False]
 [False False False]
 [False False False]
 [False False False]
 [ True  True  True]]

[[ True False False]
 [ True  True  True]
 [ True False  True]
 [False  True False]
 [ True  True False]]]
```

Moreover, if you have two arrays of the same size, you can compute their *or* via `np.logical_or`:

In [31]:
```
a = np.random.random((4, 5)) > 0.7
b = np.random.random((4, 5)) > 0.7
np.logical_or(a, b)
```

Out[31]:
```
array([[ True, False,  True, False, False],
       [ True, False,  True,  True, False],
       [False, False,  True, False,  True],
       [ True, False,  True, False,  True]])
```

Finally, we take the union of the motion detections over the three color channels, obtaining a boolean 2-d array of size $h \times w$. This array will contain the motion detection for each image. To take the union, you can use `np.max`, specifying the max to be taken over axis 2, which is the one for color:

In [32]:
```
a = np.random.random((4, 5, 3))
aa = a > 0.8
print("aa shape:", aa.shape)
b = np.max(aa, axis=2)
print("b shape:", b.shape)
print("b:", b)
```

```
aa shape: (4, 5, 3)
b shape: (4, 5)
b: [[ True  True  True False  True]
 [False False False False  True]
 [False  True False False  True]
 [ True  True  True False False]]
```

Question 3: Build the class `MotionDetection`.

In [33]:
```
from re import A
### Question 3: Implement the `MotionDetection` class

class MotionDetection(object):

    def __init__(self, num_sigmas=4., discount=0.96):
```

```
        """Motion detection implemented via averagerator.
        @param num_sigmas: by how many standard deviations should a pixel
            differ from the average for motion to be detected.  This is
            the \kappa of the above explanation.
        @param discount: discount factor for the averagerator.
        """
        # YOUR CODE HERE
        self.sigma = num_sigmas
        self.davg = DiscountedAveragerator(discount)

    def detect_motion(self, img):
        """Detects motion.
        @param img: an h x w x 3 image.
        @returns: an h x w boolean matrix, indicating where motion occurred.
        A pixel is considered a motion pixel if one of its color bands deviates
        by more than num_sigmas standard deviations from the average."""
        # YOUR CODE HERE
        d = self.davg
        d.add(img)
        bound = abs(img - d.avg)
        deviates = bound > d.std * self.sigma

        return np.max(deviates, axis=2)
```

Let's write a `detect_motion` function to facilitate our experiments. It will take a list of images, and compute the motion detection of each. If the motion detection contains more than 500 motion pixels, it puts the detection, and the index of the image, into a list of results.

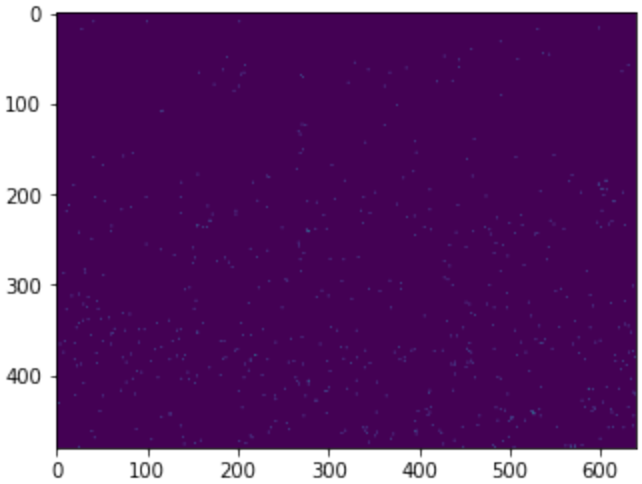
In [34]:
```
def detect_motion(image_list, num_sigmas=4., discount=0.96):
    """Takes as input:
    @param image_list: a list of images, all of the same size.
    @param num_sigmas: a parameter specifying how many standard deviations a
        pixel should be to count as detected motion.
    @param discount: the discount factor for the averagerator.
    """
    detector = MotionDetection(num_sigmas=num_sigmas, discount=discount)
    detected_motion = []
    for i, img in enumerate(image_list):
        motion = detector.detect_motion(img)
        if np.sum(motion) > 500:
            detected_motion.append((i, motion))
    return detected_motion
```

In [35]:
```
# Compute the motion detections.
motions = detect_motion(images_as_arrays[:60])
motion_idxs = [i for i, _ in motions]
assert motion_idxs == [1, 10, 47, 48, 49, 57, 58, 59]

assert np.sum(motions[7][1]) == 896

motions = detect_motion(images_as_arrays)
motion_idxs = [i for i, _ in motions]
assert motion_idxs[:13] == [1, 10, 47, 48, 49, 57, 58, 59, 66, 67, 68, 85, 96]
```

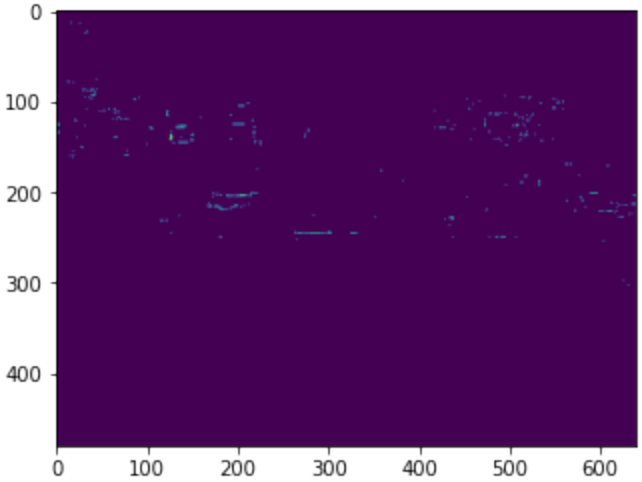
We can finally visualize the detected motions.

In [36]:
```
import matplotlib.pyplot as plt
for i, m in motions:
    # We only print images where there are at least 500 pixels of motion.
    if np.sum(m) > 500:
        print("Motion at image", i, ":", np.sum(m), "----------------------------------
        # We first show the image, for reference.
        plt.imshow(images_as_arrays[i] / 255)
        plt.show()
        # And then the motion detection.
        plt.imshow(m)
        plt.show()
```

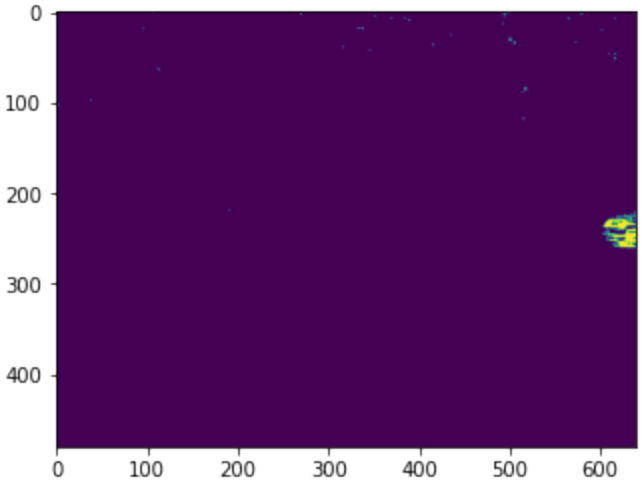
```
Motion at image 1 : 548 ------------------------------------
```



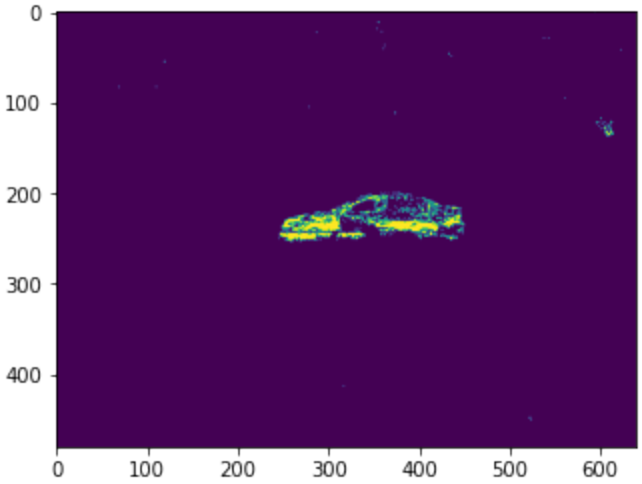

```
Motion at image 10 : 550 -----------------------------------
```



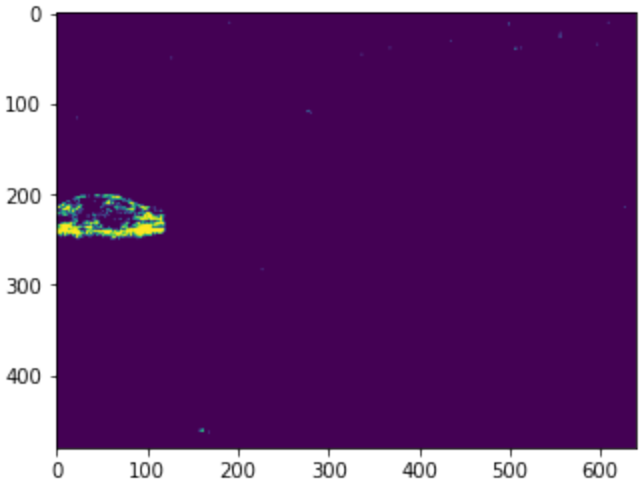

```
Motion at image 47 : 767 -----------------------------------
```



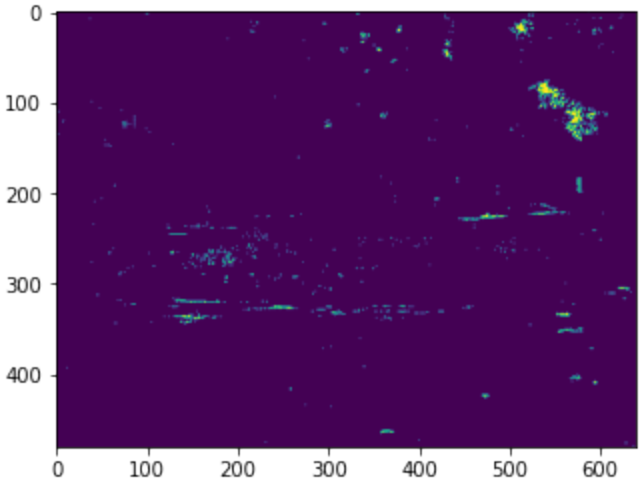

```
Motion at image 48 : 3288 -----------------------------------
```



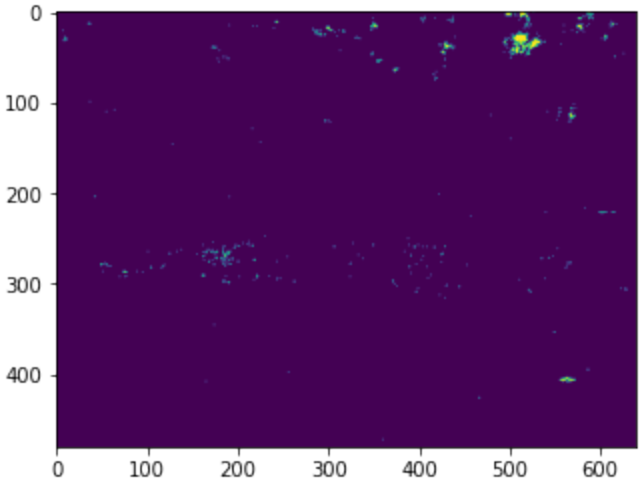

```
Motion at image 49 : 2043 -----------------------------------
```



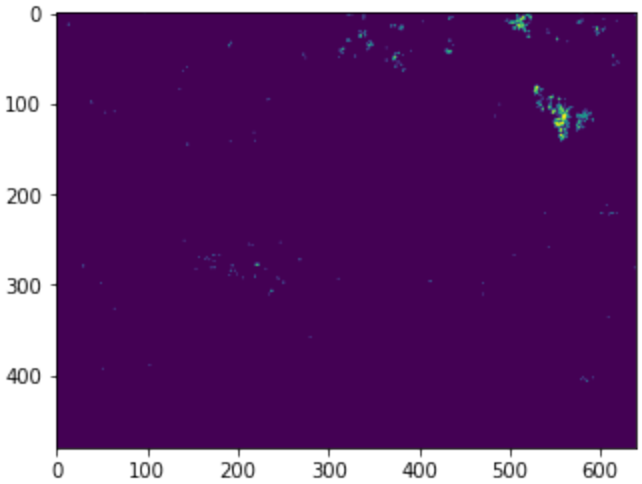

```
Motion at image 57 : 2651 -----------------------------------
```



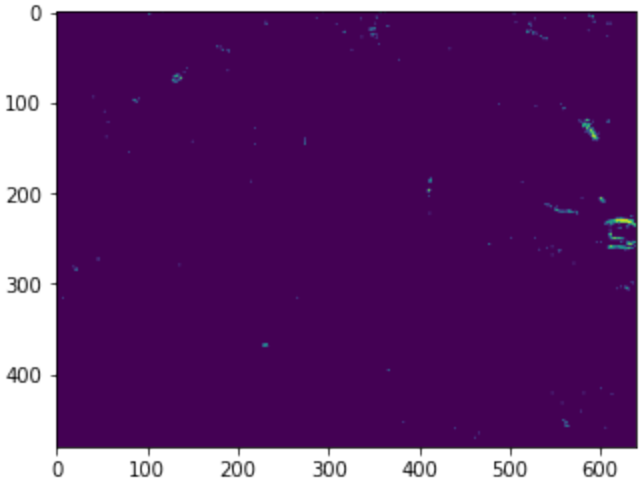

```
Motion at image 58 : 1199 -----------------------------------
```



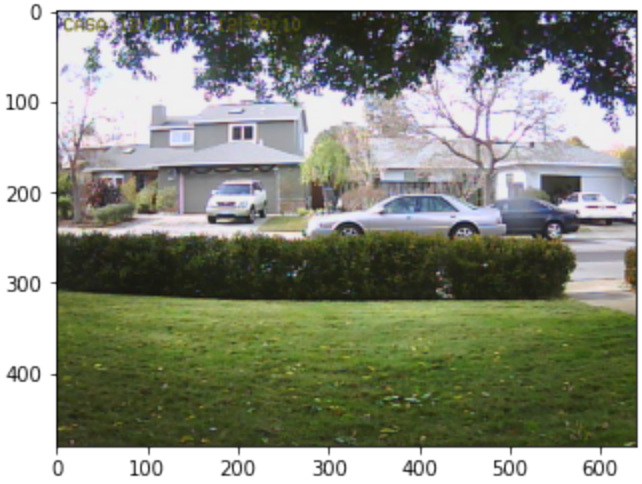

```
Motion at image 59 : 896 ------------------------------------
```

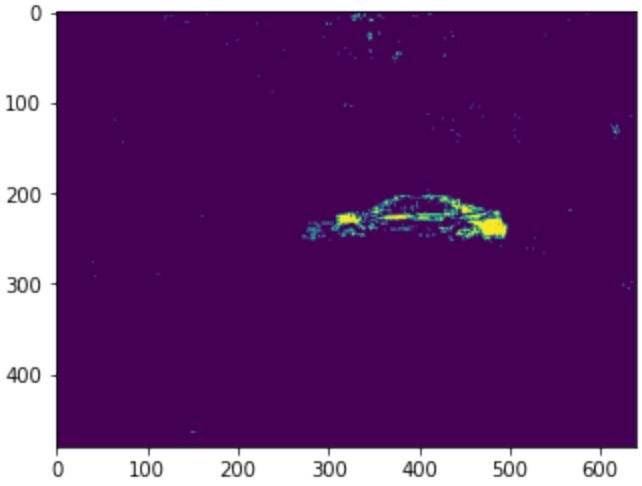



```
Motion at image 66 : 633 -----------------------------------
```

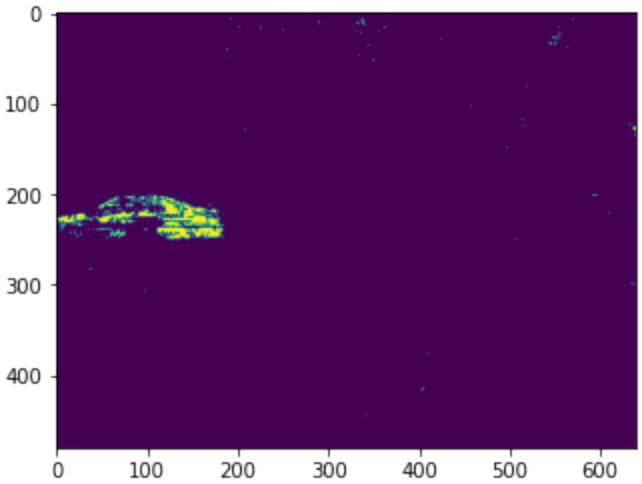



```
Motion at image 67 : 2645 -----------------------------------
```

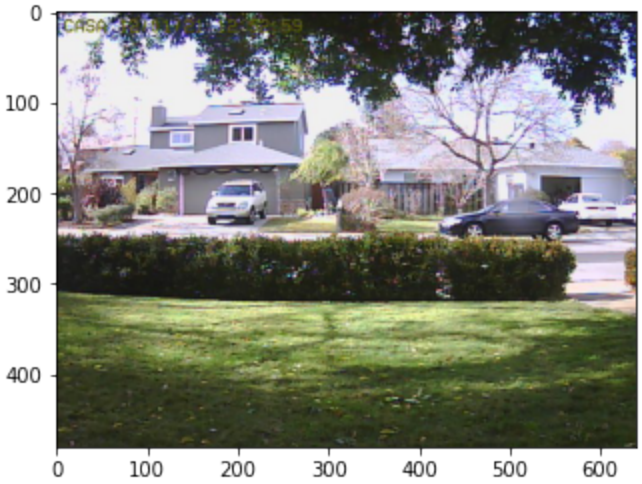


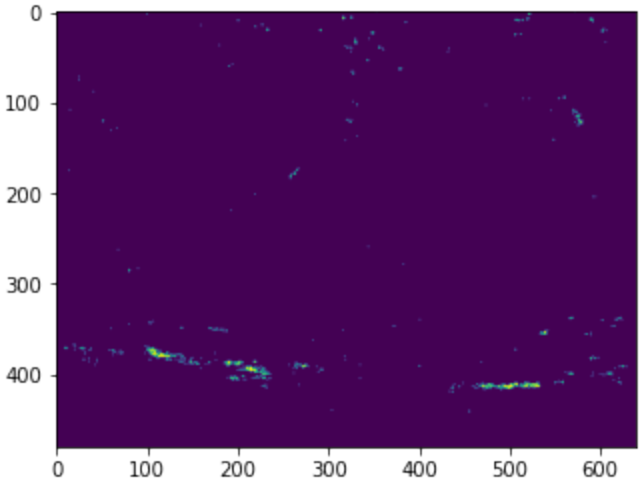



```
Motion at image 68 : 2802 -----------------------------------
```

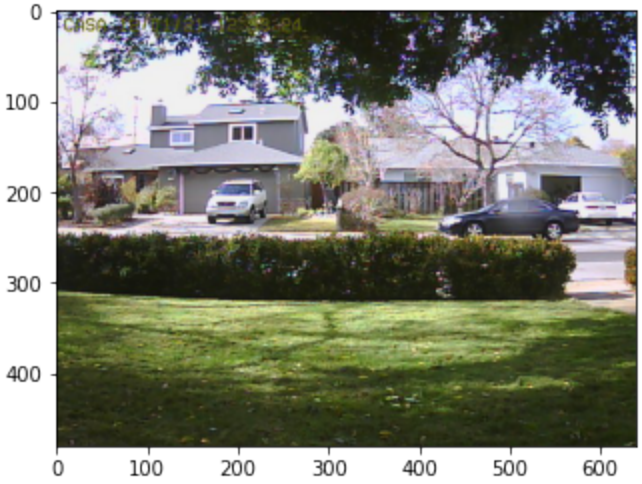


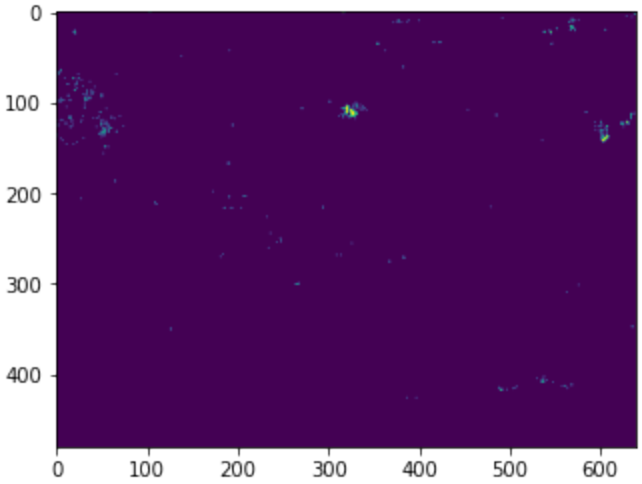

```
Motion at image 85 : 1086 -----------------------------------
```



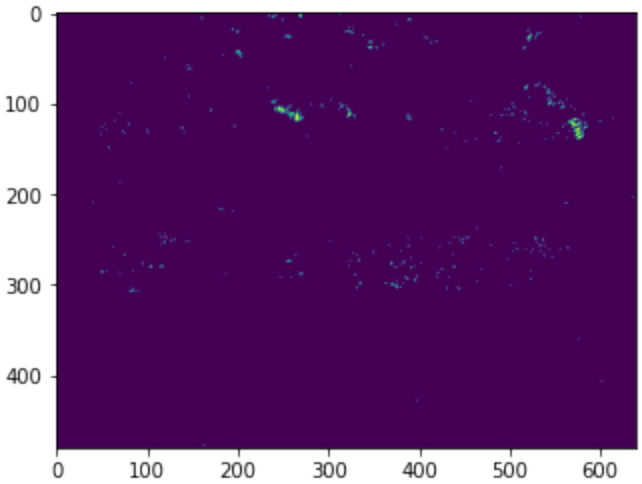



```
Motion at image 96 : 540 -----------------------------------
```

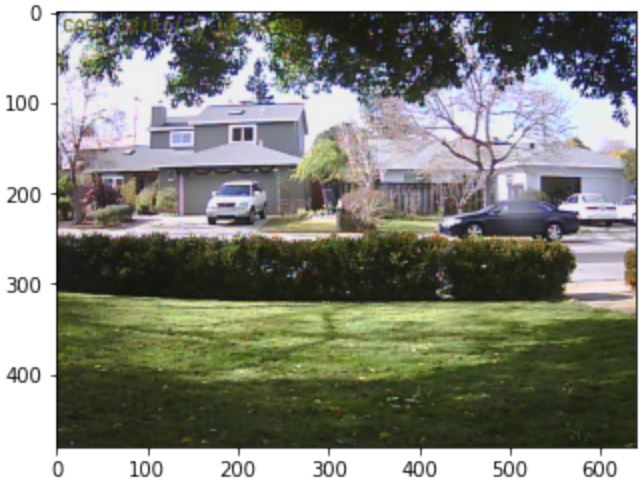


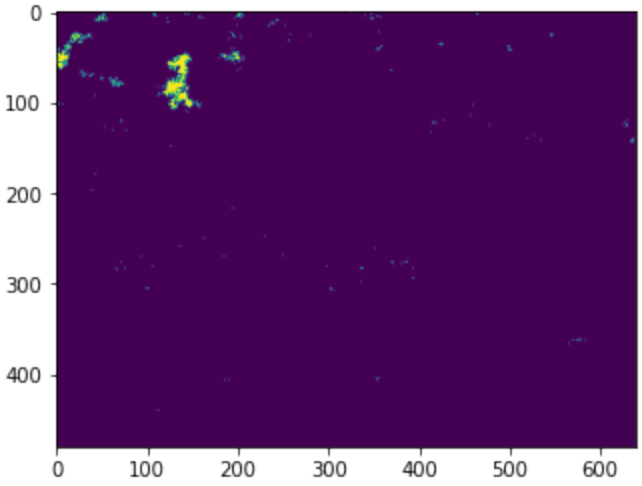



```
Motion at image 99 : 829 -----------------------------------
```

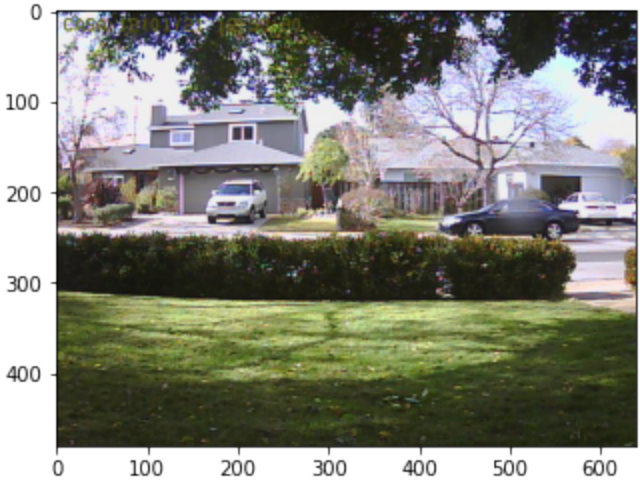


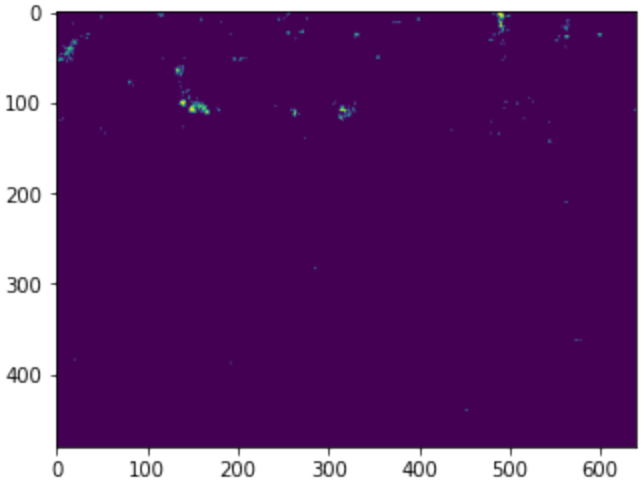

```
Motion at image 100 : 1574 -----------------------------------
```



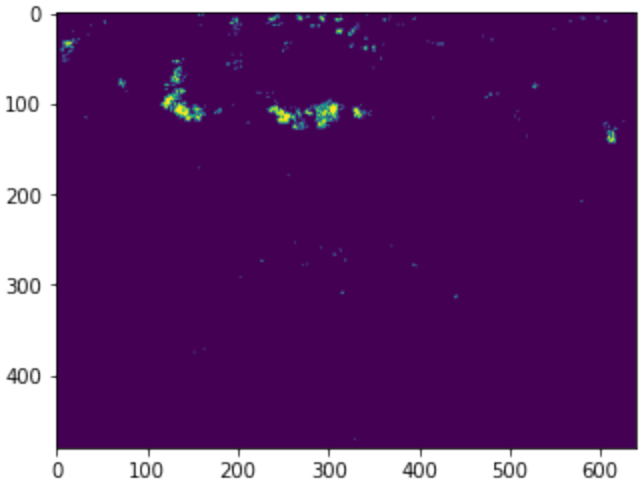



```
Motion at image 101 : 620 -----------------------------------
```

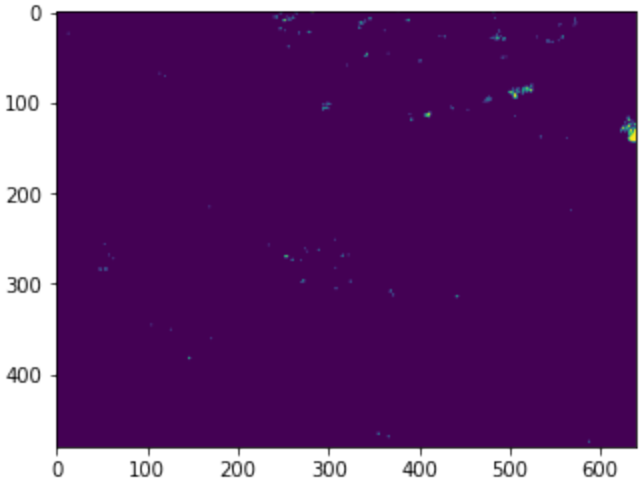




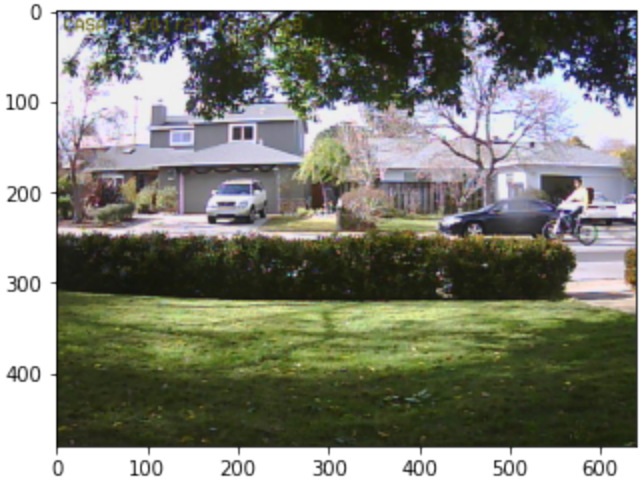

```
Motion at image 110 : 2016 -----------------------------------
```

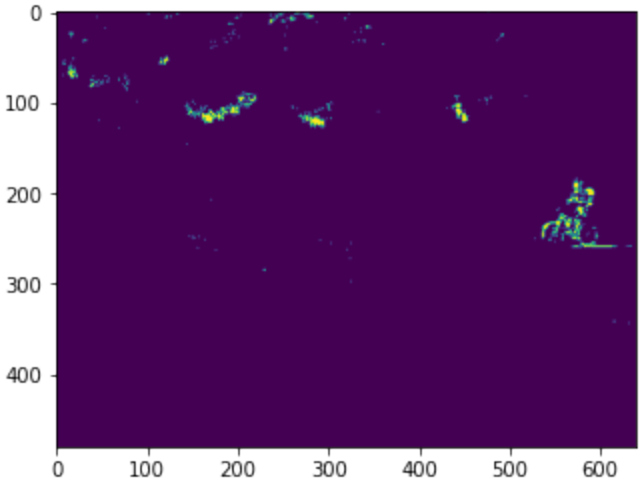



```
Motion at image 111 : 512 -----------------------------------
```

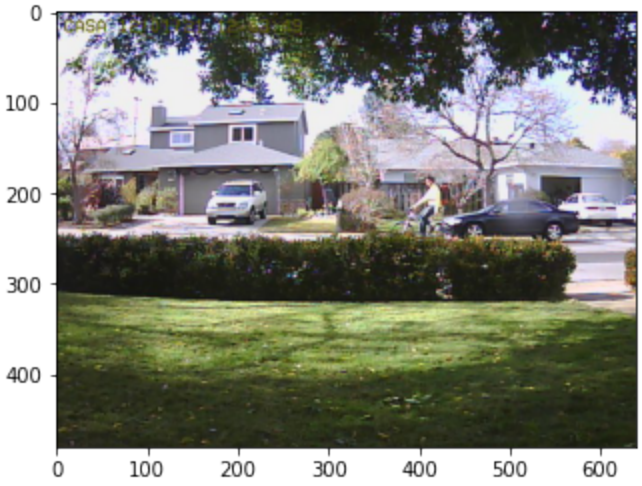


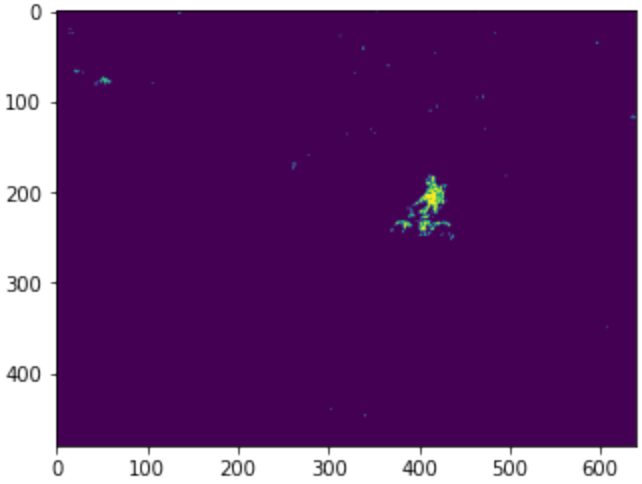

```
Motion at image 113 : 2129 -----------------------------------
```



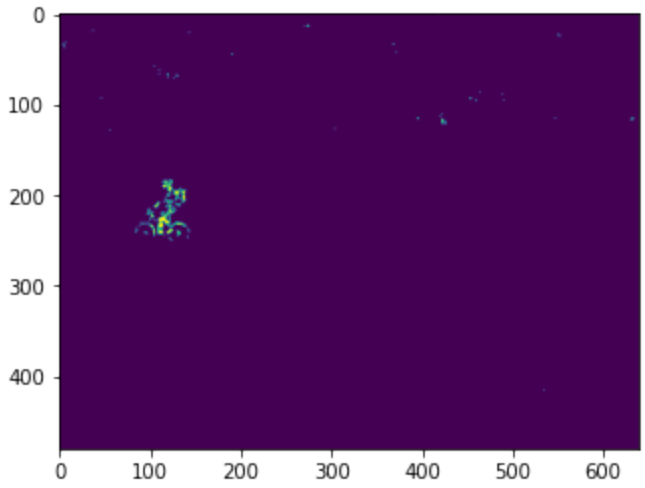



```
Motion at image 114 : 903 -----------------------------------
```






```
Motion at image 115 : 657 -----------------------------------
```



Here are some final tests.

In [37]:
```
### 20 points: Tests for motion detection

motions = detect_motion(images_as_arrays[:60])
motion_idxs = [i for i, _ in motions]
assert motion_idxs == [1, 47, 48, 49, 57, 58, 59]

assert np.sum(motions[6][1]) == 886
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-37-30b717b818bc> in <module>()
      3 motions = detect_motion(images_as_arrays[:60])
      4 motion_idxs = [i for i, _ in motions]
----> 5 assert motion_idxs == [1, 47, 48, 49, 57, 58, 59]
      6
      7 assert np.sum(motions[6][1]) == 886

AssertionError:
```

In [ ]:
```
# 10 points: some more tests.

motions = detect_motion(images_as_arrays)
motion_idxs = [i for i, _ in motions]
assert motion_idxs[:12] == [1, 47, 48, 49, 57, 58, 59, 66, 67, 68, 85, 96]
```

We can see that the motion detection does a very reasonable job of detecting the passing cars and bycicle, while almost entirely suppressing the tree that is shaking in the wind. As the tree shaking is constant, its pixels have very high standard deviation. If we plot the standard deviation of each pixel, we get a very good impression of how much noise or regular motion took place at that pixel.

In [ ]:
```
a = DiscountedAveragerator(0.96)
for i, img in enumerate(images_as_arrays):
    a.add(img)
# We display the final sigma.
sigma = np.max(a.std, axis=2)
```

```
plt.imshow(sigma, cmap='gnuplot')
plt.colorbar()
plt.show()
# Let's compare with the last image.
plt.imshow(images_as_arrays[-1] / 255)
plt.show()
```

We see how the constantly-shaking trees give rise by far to the greatest pixel standard deviation.