ENGR 101 Project 2: Dealing with Radiation

Project Due **Wednesday**, February 12, 2020

In this project, you will use MATLAB to process images through vectorization, logical indexing, and the use of built-in functions, especially to do signal and image processing.

The autograded portion of the final submission is worth 100 points, and the style and comments grade is worth 10 points.

You may work alone or with a partner. Please see the syllabus for partnership rules.

Engineering intersections:
Nuclear Engineering, Biomedical Engineering, Climate and Space Sciences and Engineering, Electrical Engineering, Signal Processing, Computer Science

Project Roadmap

This is a big picture view of how to complete this project. Most of the pieces listed here also have a corresponding section later on in this document that goes into more detail.

1. **Read through the project specification (DO THIS FIRST)**
   Start with the introduction and project essentials on the next page, then sketch out a plan/list of steps/flowchart for how you want to write your program. Be ready to show this to staff during office hours so we can help you more efficiently!

2. **Implement and test the helper functions: `removeNoise`, `heatmap`, and `zones`**
   You can work on `heatmap` and `zones` before completing `removeNoise`, but the results won't look right if you test them on a noisy image! The implementations for `heatmap` and `zones` share a lot of elements. Make sure you refer back to the section that explains the HSV image format as needed.

3. **Implement and test the `detectTumor` function**
   You have the freedom to design the algorithm for this function on your own. This is both a challenge and a great opportunity to learn. We're happy to help you brainstorm ideas here!

4. **Implement and test the `WatchDisplay` driver program**
   You must use all your image-generating helper functions to create local versions of the heatmap and zones images.

5. **Add finishing touches for style and comments**

6. **Submit to the Autograder**
   Submit the following files, to the autograder.

`removeNoise.m`	`heatmap.m`	`zones.m`	`detectTumor.m`	`WatchDisplay.m`

Purpose

The primary purpose of this project is to reinforce your mastery of vectorized code in MATLAB for processing large amounts of data. In particular, element-by-element array operations, logical indexing, and the use of built-in vectorized functions are skills that will serve you well on this project. Another goal is to give you familiarity with the use of MATLAB for working with images.

Introduction

The construction of our newest dome has recently been completed, and we now have room for more people. Many people are en route from Earth and eager to join the settlement, but the work of taming the harsh environment is just beginning. One of the most pressing concerns is the danger of radiation exposure.

Because Proxima Centauri is a flare star, and also due to Proxima b's thin atmosphere and weak magnetic field, radiation exposure is a serious concern outside the dome. Flares are difficult to predict, and radiation levels change throughout the day and vary by location. Even low levels of chronic exposure are known to have adverse health effects, so we must be vigilant. As such, a radiation scanning system has been developed to detect radiation levels in the settlement area. You are working with a team of engineers to process the scanning data and keep the public informed.

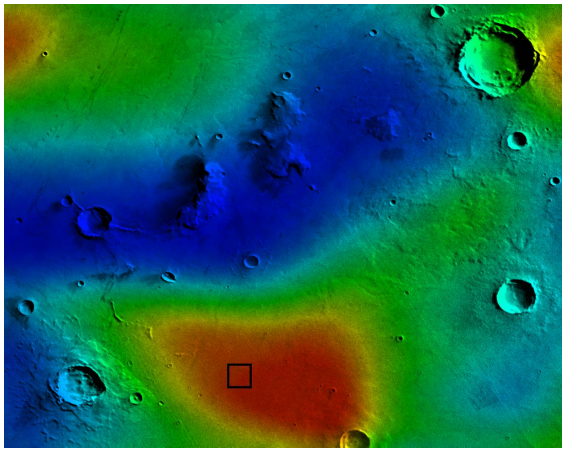


Figure 1. A severe radiation storm near the dome (dome is the black square).

In addition to tracking radiation storms, citizens who may be exposed to higher levels of radiation participate in routine health checkups involving MRI scans. Early detection of potential tumors can enable more effective treatments of radiation-linked cancers. In task 5, you

Your Job

In **Tasks 1-3**, you implement **functions** to process the radiation data and create generate images that show the current conditions.

In **Task 4**, you write a **driver** program that calls these functions to present the data.

In **Task 5**, you develop an algorithm to identify potential tumors in brain scan images.

Project Essentials (IMPORTANT)

This project utilizes some functions that are part of MATLAB's **Image Processing Toolbox**. Lab 0 instructed you to install this toolbox when you installed MATLAB. You will also need to download the set of starter files for the project from the Google Drive.

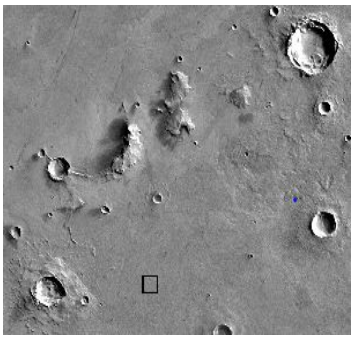
You may want to review the lecture material on working with images. In particular, the examples of working with individual channels of an HSV image are relevant to this project.

Download Files

Before writing your functions, go to Project 2 on the google drive and download the files: `scan_radiation.p`, `scan_brain.p`, `GPS_data.p`, `display_settings.p`, `dome_area.jpg`, and `brain.jpg`. You will use these in writing and testing your functions. Also download the starter files for the functions below.

Note: you will not be able to open the `.p` files. These are Matlab functions that you can call from your functions, but you can't see what is in the files. See below for examples of how to use these `.p` functions.

The `dome_area.jpg` file contains a satellite image of Proxima b. The black square shows the location of one of the domed settlements.



The `dome_area.jpg` file provides the starting image for the functions you will write in Tasks 2 & 3. You should read the dome image in either a driver program or a test script using the `imread` function and save it as a variable. This variable (which contains the numerical values corresponding to the image), can then be passed to individual functions as a parameter.

Notes

The individual functions you write for tasks 1-3 should NOT load the dome image directly using `imread()`.

The `brain.jpg` file is used by the `scan_brain.p` function. You should NOT directly read or interact with this file in your code. See How to Use `scan_brain.p` for details.

How to Use `scan_radiation.p`

The `scan_radiation` function provided to you allows access to the radiation scanning system's scans from the last 1000 hours in order to test your implementation before it is installed on the live system. To use the function, simply provide a number between 0 and 1000 indicating the time for which you want the data.

`scan_radiation` returns a 2D matrix with the same size as the `dome_area` image. Each element of the matrix represents the intensity of radiation at the corresponding location, measured in millisieverts (mSv) per hour. Radiation values range from 0 to 100 mSv/hr (inclusive).

Try typing the following into the Command Window:

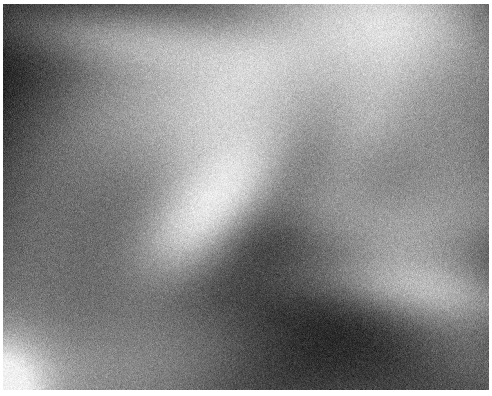
```
rad = scan_radiation(0);
```

You should see `rad` as a variable in the Workspace Window; this variable contains the 2D matrix of the intensity of radiation at the locations around the dome at hour 30. Check to make sure that the values are between 0 and 100 (inclusive).

In order to visualize the radiation matrix, we could treat it as a grayscale image and use `imshow`. However, `imshow` (and other MATLAB functions) expects intensity values to be between 0 and 1, so we first need to scale the radiation values down by 100. To try this, type the following into the Command Window:

```
imshow(rad ./ 100);
```

You should see the following picture pop up in a figure window (it may replace a picture in an existing figure window):



Here, white areas represent areas of high radiation. You may be able to tell from this image that something's not quite right. There are tiny spots with much different values than their surrounding area. Clearly this can't actually be the case. It seems that the readings from the scanning system are noisy -- factors other than the actual radiation levels are causing random fluctuations in the value measured. This is a limitation of the physical scanning system that your team will need to correct when processing the data.

How to Use `scan_brain.p`

To develop and test your tumor detection algorithm (see Task 5 for more details), a set of 100 simulated brain scans are available through the `scan_brain` function. Simply provide a number 1-100, and the function will return the scan as a `double` matrix representing a grayscale image (values between 0 and 1).

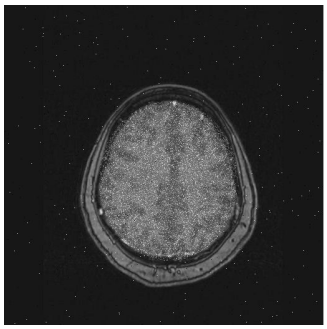
To see how the `scan_brain` function works, type the following into the Command Window:

```
brain1 = scan_brain(1);
```

This will create a variable, `brain1`, that contains a 2D matrix of numerical values representing a picture of a brain scan. To see the scan visually, type:

```
imshow(brain1);
```

You should see the following picture pop up in a figure window (it may replace a picture in an existing figure window):



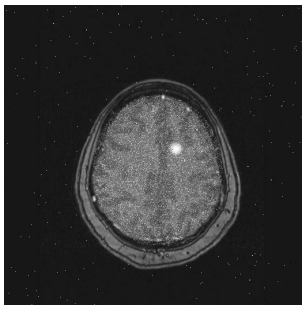
This picture shows no tumor in brain scan #1. However, if you type:

```
brain79 = scan_brain(79);
```

to load in a different brain scan, and type:

```
imshow(brain79);
```

to show this brain, scan #79, you will see the white area that indicates a tumor:



How to Use GPS_data.p

To develop and test your watch display driver program (see Task 4 for more details), sample GPS data is available through the GPS_data function. Simply call the function with no arguments, and the function will return three values:

1. Row number -- corresponds to the user's current row location in the heatmap or zones image
2. Column number -- corresponds to the user's current column location in the heatmap or zones image
3. Time -- corresponds to the user's current time, in hours; can be used as input to the scan_radiation function to get the correct radiation scan.

To see how the GPS_data function works, type the following into the Command Window:

```
[r,c,t] = GPS_data();
```

This will create three variables, r, c, and t, that each contain a single integer corresponding to the user's current row, column, and time. The GPS_data function returns random samples of data, so each time you call it, the values returned will change.

How to Use display_settings.p

To develop and test your watch display driver program (see Task 4 for more details), sample watch display settings are available through the display_settings function. Simply call the function with no arguments, and the function will return one value: the zoom offset for the local views of the heatmap and zones images.

To see how the display_settings function works, type the following into the Command Window:

```
z = display_settings();
```

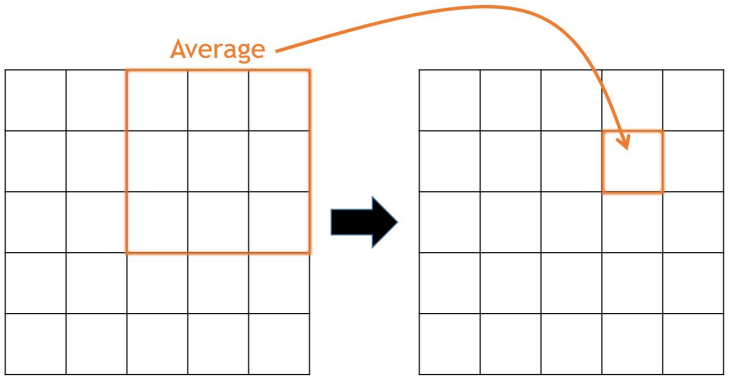
This will create one variable, z, that contains a single integer corresponding to the zoom offset needed to create the local views of the heatmap and zones images. The display_settings function returns random samples of data, so each time you call it, the values returned will change.

Note

You may assume that the combination of row, column, and zoom offset returned by the GPS_data and display_settings functions will not cause an out-of-bounds array indexing error in MATLAB.

Task 1: Noise Reduction (`removeNoise`)

The noise appears to be Gaussian, which means a basic mean (average) filter will work well to smooth out the noise. In general, the idea is that we'll create a new matrix where each element from the original matrix of radiation values is replaced by an average of itself and its neighbors. In signal processing, this would be called a "low-pass filter" because we're not allowing high frequency noise to pass through.



Computing a new element based on a 3x3 mean filter.

1/9	1/9	1/9
1/9	1/9	1/9
1/9	1/9	1/9

The matrix representation of a 3x3 mean filter.

MATLAB supports image filtering with the `imfilter` function. It takes the original matrix and a matrix representing the filter as parameters and returns a new, filtered matrix. The filter matrix specifies how much of each neighbor is used in the calculation of the corrected value of each element. For example, for a 3x3 mean filter, we want to take $1/3^2$ of each of the 9 neighboring elements to compute our new value (i.e. the average). Therefore, the matrix representing the filter is a 3 x 3 matrix where each element is 1/9 (because $1/3^2 = 1/9$). `imfilter` also takes an optional third parameter to specify how to handle edges: **use `'replicate'` for this project**.

In your `removeNoise` function, you will use `imfilter` to apply a mean filter with a given size to the radiation data **three times**. Each application reduces the noise incrementally and prepares the image for use in creating both the heatmap and the regions images. If you want to visually check the results, you can use the `imshow` function to visualize the filtered results.

Original

One Application of 15x15 Mean Filter

Three Applications of 15x15 Mean Filter

Function definition and description

```
function [ rad ] = removeNoise( rad, n )
%removeNoise Removes noise from a matrix of radiation values by
%  applying an nxn mean filter three times.
%       n: The size of the filter (e.g. if n=3, use a 3x3 filter)
%     rad: a matrix of numbers representing the radiation
%          measurements from the scanner.
%          NOTE: A matrix obtained from a call to the scan_radiation()
%          may be used as an input argument when calling this function,
%          but you should NOT call scan_radiation() inside of this
%          function!
```

Testing

To test your `removeNoise` function, first get a set of radiation data by typing this into the Command Window:

```
rad = scan_radiation(30);
```

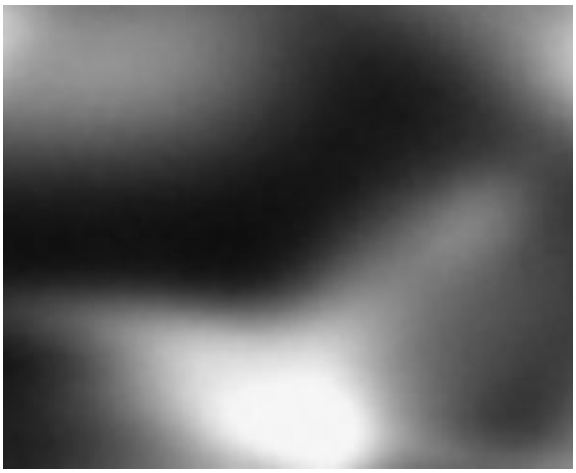
Then, call your `removeNoise` function and save the smoothed data back into `rad`:

```
rad = removeNoise(rad, 15);
```

Finally, show the smoothed radiation data as a picture:

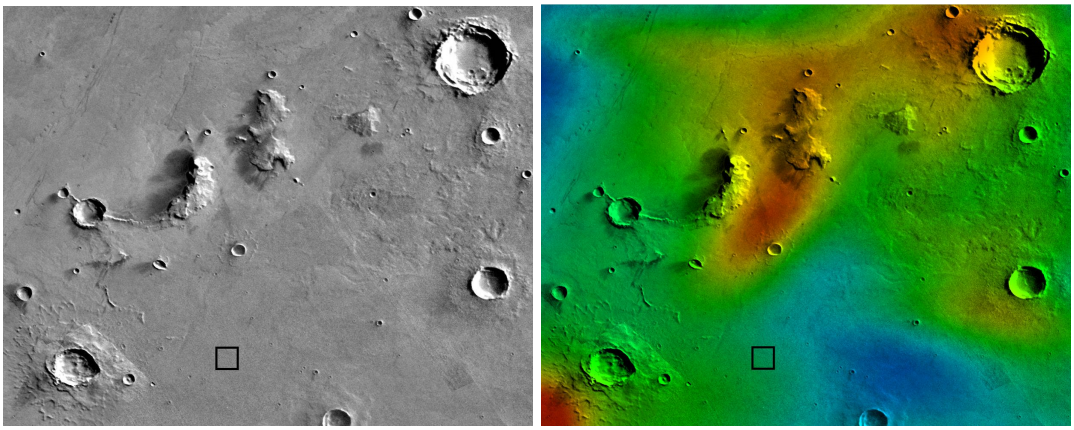
```
imshow(rad ./ 100);
```

If your `removeNoise` function is working correctly, you should see a picture that looks like this:



Task 2: Creating a Radiation Heatmap (`heatmap`)

Once the radiation matrix has been processed to remove noise, we would like to visualize radiation levels near the dome and settlement area. One format for presenting this is a heatmap image, which uses color to encode a particular quantity (in this case, the level of radiation). You may be familiar with heatmaps used in weather forecasting, where red areas on a map indicate higher levels of precipitation or inclement weather. In our case, we will start with a grayscale image of the settlement area and adjust the color of each pixel based on the corresponding value in the radiation matrix (see figures below).



Left: Grayscale image of the area around the proposed settlement.
Right: A heatmap of radiation levels at hour 0; red indicates areas of high radiation, blue represents areas of low radiation.

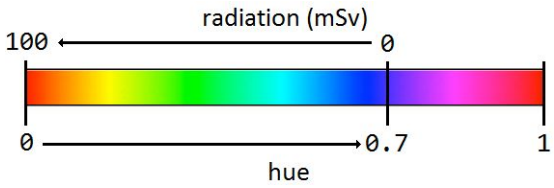
Write the `heatmap` function to generate these images. It should take an RGB image of the dome area and a radiation matrix as parameters and return the heatmap as an RGB image. Because we would like to manipulate the color of the image without changing anything else, the HSV image representation (see above) is quite convenient. Before working with the image, you should first convert it to HSV using the `rgb2hsv` function.

Recall from lecture...

A 3-dimensional matrix represents images. The rows and columns correspond to the pixels in the image (i.e. the rows and columns). The third dimension represents the different "channels" of the image (i.e. red-green-blue or hue-saturation-value).

Once the image is in HSV format, we can consider each channel separately:

- Channel 1: "Hue".
  This channel controls *which* color is used at each pixel, so we want to set its elements according to the corresponding values in the radiation matrix. To convert from a radiation level to the appropriate hue value, use the formula at the right.



$$hue = 0.7 - 0.7 .* rad ./ 100.0$$

- Channel 2: "Saturation".
  This channel controls *how strong* is the color of each pixel. Set all values in this channel to 1, since we want the colors in the heatmap to be as vibrant as possible.

- Channel 3: "Value".
  This channel controls how bright are the pixels in the image and thus contains the grayscale picture of the settlement area. Thus, you should just leave this channel as is.

Once you have made the appropriate changes to the HSV image, convert it back to an RGB image with the `hsv2rgb` function before returning from the `heatmap` function.

Function definition and description

```
function [ img ] = heatmap( img, rad )
%heatmap Generates a heatmap image by using values from rad to set
%  values in the hue channel for img. Hue values vary smoothly
%  depending on the corresponding radiation level.
%     img: a 3-dimensional matrix of numbers representing an image in
%          RGB (red-green-blue) format, which forms the background for
%          to which the heatmap colors are applied.
%     rad: a matrix of numbers representing the radiation
%          measurements, between 0 and 100 millisieverts.
%          It is has the same width and height as the img parameter.
```

Testing

To test your `heatmap` function, first read the dome picture into Matlab by typing this into the Command Window:

```
img = imread('dome_area.jpg');
```

Then get a set of radiation data:

```
rad = scan_radiation(30);
```

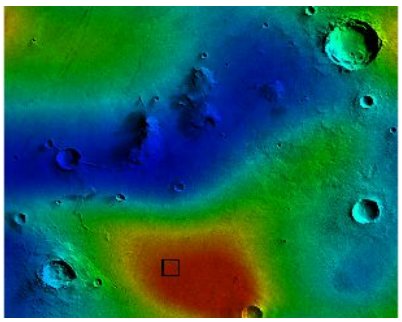
Next, call your `removeNoise` function and save the smoothed data back into `rad`:

```
rad = removeNoise(rad, 15);
```

Finally, call `heatmap` and show the result as an image:

```
imshow(heatmap(img, rad));
```

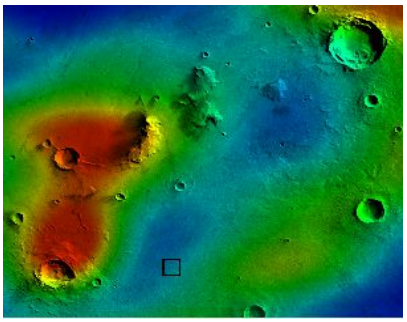
If your `removeNoise` function and `heatmap` functions are working correctly, you should see a picture that looks like this:



Also test your functions using different radiation scans like this:

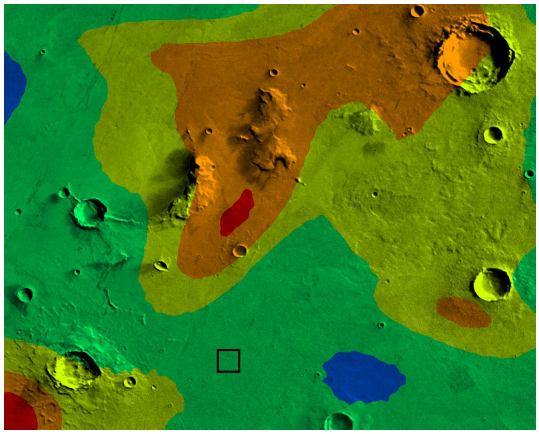
```
imshow(heatmap(img, removeNoise(scan_radiation(100),15)));
```

Which should result in this picture:



Task 3: Identifying Radiation Threat Zones (zones)

While the heatmap described above is useful for a number of applications, it is not clear from the image exactly what level of radiation corresponds to a particular color. One problem is that the hue space that is displayed as green is significantly larger than other colors. There are many strategies for coping with this (e.g. changing to a nonlinear mapping to hue values), but the one we will use is to select a limited number of colors to represent different "radiation threat zones" in order to convey clearly the severity of the radiation.



Radiation threat zones at hour 0.

All standard attire for external settlement workers contains a protective layer of radiation absorbing fabric, but in some cases more significant protection may be needed. A few members of your team have experience in nuclear and biomedical engineering and have developed a table that describes the correspondence between safety and different levels of radiation. This table must be used to identify radiation threat zones and to relate hue("radiation threat zone") with safety.

Threat Zone	Radiation Levels	Hue		Description
1	[0,20) mSv/hr	0.6		Safe with standard issue attire.
2	[20,50) mSv/hr	0.4		Safe for brief exposure with standard issue attire, or prolonged exposure with hazmat suit.
3	[50,70) mSv/hr	0.2		Safe for brief exposure with hazmat suit.
4	[70,90) mSv/hr	0.1		Unsafe for human exposure. Radiation shielding required on all autonomous equipment.
5	90+ mSv/hr	0		Bad news bears.

The notation [x,y) indicates a range where the lower bound, x, is included, but the upper bound, y, is not.

Write a function called zones that produces an image of the radiation threat zones near the settlement area. For your implementation, use the same procedure as for the heatmap to overlay colors on the image of the settlement area, except this time use the table above to determine the appropriate hue. That is, take an RGB image of the area and a radiation matrix as parameters, convert the image to HSV, set all saturation levels to 1, modify the hue channel according to the table, and convert it back to RGB to be returned.

Function definition and description

```
function [ img ] = zones( img, rad )
%zones Generates an image colored according to radiation threat
%  zones. Values from rad are used to determine the zone, and the hue
%  value in img is set accordingly.
%     img: a 3-dimensional matrix of numbers representing an image in
%          RGB (red-green-blue) format, which forms the background for
%          to which the heatmap colors are applied.
%     rad: a matrix of numbers representing the radiation
%          measurements, between 0 and 100 millisieverts.
%          It is has the same width and height as the img parameter.
```

Testing

To test your `zones` function, first read the dome picture into Matlab by typing this into the Command Window:

```
img = imread('dome_area.jpg');
```

Then get a set of radiation data:

```
rad = scan_radiation(30);
```

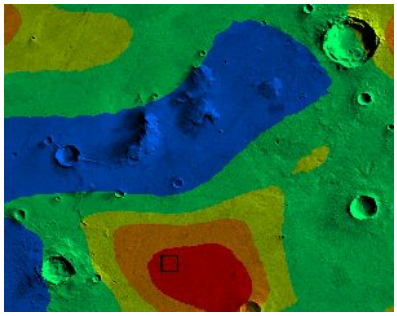
Next, call your `removeNoise` function and save the smoothed data back into `rad`:

```
rad = removeNoise(rad, 15);
```

Finally, call `zones` and show the result as an image:

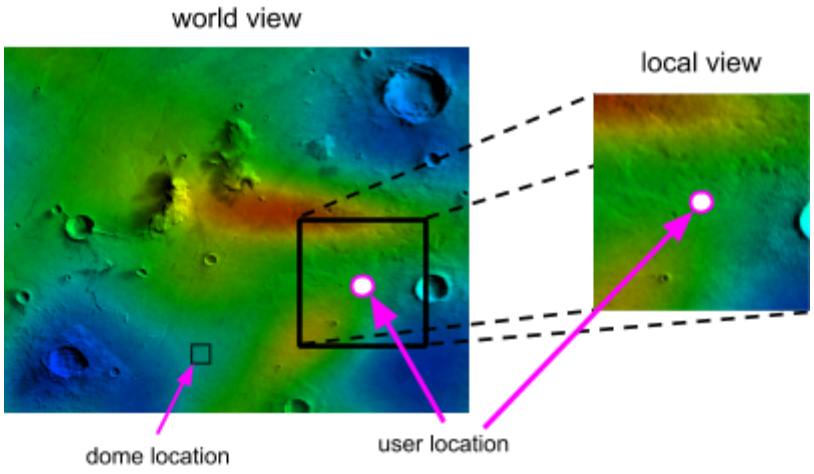
```
imshow(zones(img, rad));
```

If your `removeNoise` function and `zones` functions are working correctly, you should see a picture that looks like this:

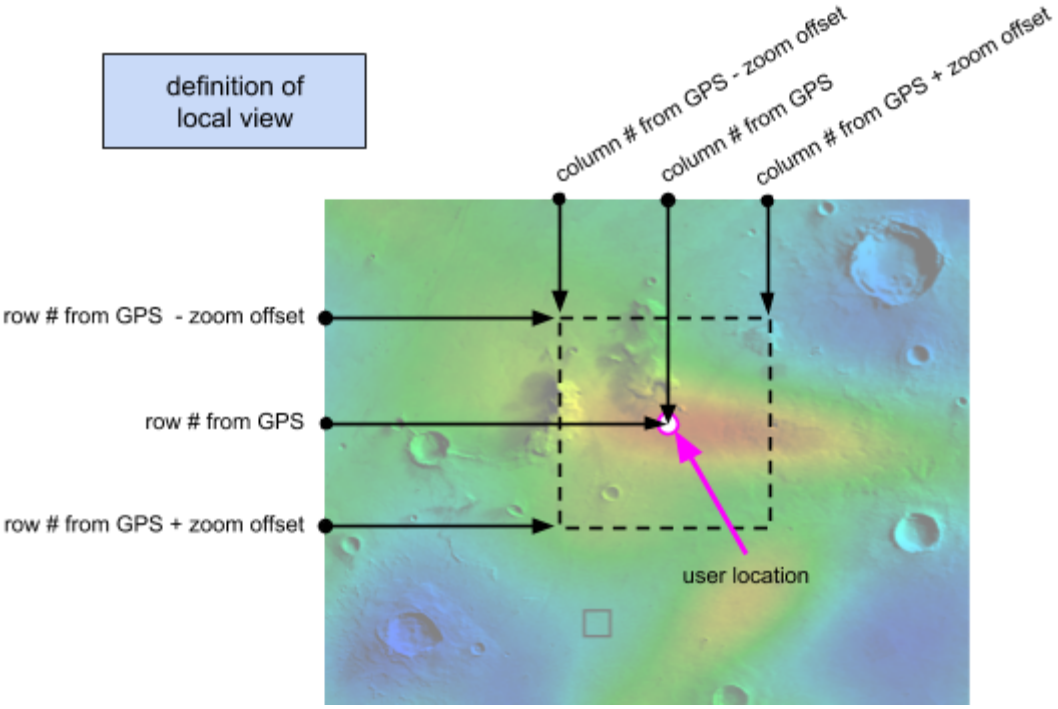


Task 4: Smartwatch Driver Program (`WatchDisplay.m`)

All Proximabians are equipped with a standard issue, radiation-protected smartwatch. The smartwatch has a satellite link so it can pull GPS data and communicate to the sensor array that monitors radiation. When the user brings up the radiation monitoring app, the app needs to update its local heatmap and local zones images based on where the user is located (using GPS data) and the current display settings on the watch (for example, the current zoom setting).



The amount of area shown in the local view depends on the zoom offset in the watch. A schematic showing what part of the world view is kept for the local view is shown below:



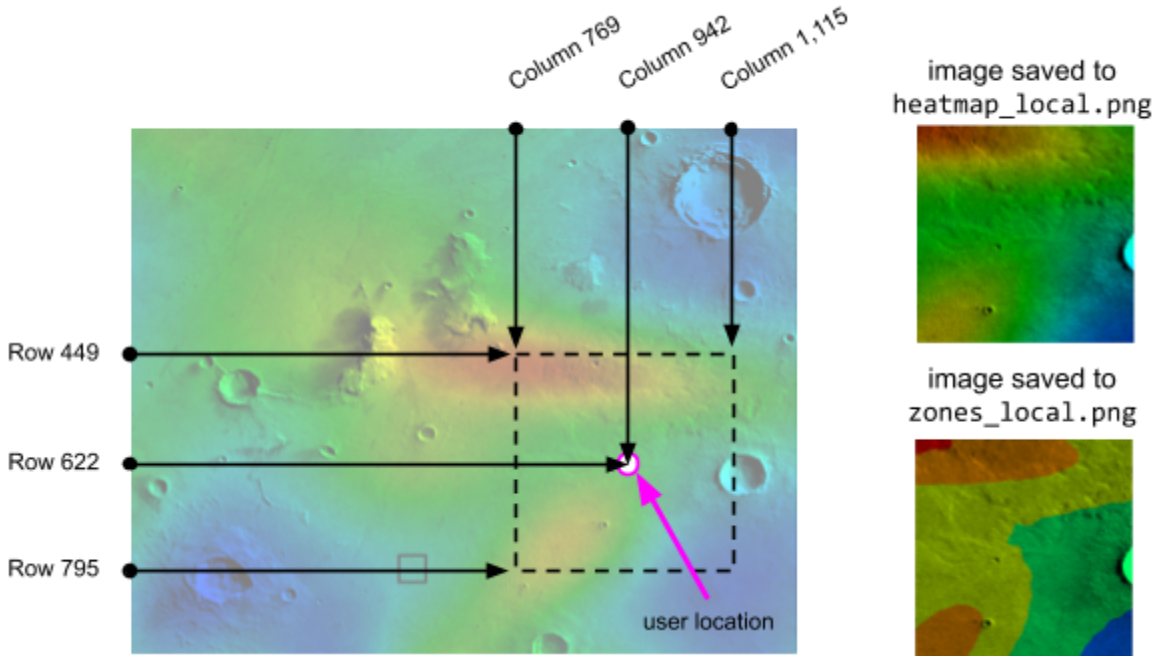
NOTE: The width and height of the local view image are both equal to 1 + 2 * (zoom offset).

The watch has two helper functions you can use: `GPS_data` and `display_settings`. See How to Use `GPS_data.p` and How to Use `display_settings.p` for examples on calling these functions. For this project, you need to write a driver program that will:

1. Clear any existing data in MATLAB
2. Load the image of the dome
3. Get GPS data from the watch using the `GPS_data` helper function
4. Get radiation data from the scanner using the `scan_radiation` helper function and the time returned by the `GPS_data` helper function
5. Remove noise in the radiation data using the `removeNoise` helper function (using a filter size of 15)
6. Get the zoom offset using the `display_settings` helper function
7. Create the local version of the heatmap and the zones image according to the GPS' row and column and the display's zoom offset
8. Save the local version of the heatmap as `heatmap_local.png`
9. Save the local version of the zones image as `zones_local.png`

Note
The `WatchDisplay` driver program should not display any images within MATLAB. If you use `imshow()` from the driver program or from any function, the Autograder will crash.

Below is an example of the local heatmap and zones images for a particular configuration:



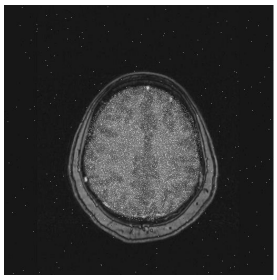
GPS Row **622**, GPS Col **942**, GPS Time **935**, Zoom Offset **173**.
Size of local images: 347 rows and 347 columns.

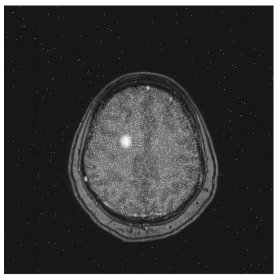
Task 5: Detecting Potential Tumors in Medical Scans (`detectTumor`)

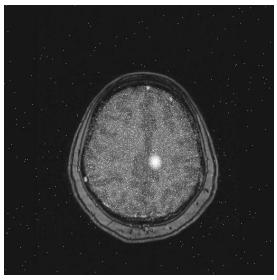
Although the protective equipment used by settlement workers and the precautionary radiation forecasting developed by your team has helped to prevent instances of acute radiation sickness, long-term exposure to low level radiation is still a concern. On Earth, even miniscule increases in radiation exposure have been linked to increased risk of certain cancers. It is unavoidable that natural exposure on Proxima b will be significantly higher, so you are also working with a team of biomedical engineers to develop a system for detecting potential tumors in medical scans.

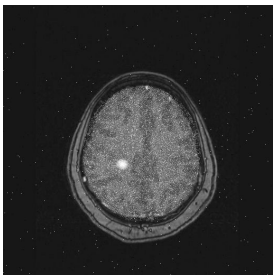
Because the risk from long-term exposure is so high and early detection significantly improves prognosis, all members of the settlement participate in regular medical imaging. However, this generates far too much data to be processed by hand, and so the system you design will serve as an automated mechanism for detecting likely tumors to be reviewed by doctors for confirmation. The tradeoff implicit in your task is to ensure no possible tumors are missed, but to also minimize false positive reports that waste time later on in the review process.

For this project, you will create a specific algorithm to detect the presence of tumors in brain scans. To develop and test your algorithm, a set of 100 simulated brain scans are available through the `scan_brain` function - simply provide a number 1-100 and the function will return the scan as a `double` matrix representing a grayscale image (values between 0 and 1).


Scan 1


Scan 4


Scan 24


Scan 46

As can be seen from these sample images, the tumors are quite easy to spot with the human eye. This is because patients are given a special non-toxic dye to drink that causes a strong response to the medical imaging system. However, your job is to automate the process by designing an algorithm that can detect the presence of tumors - this requires breaking down the problem into concrete operations that can be performed in MATLAB.

For this project, implement a function called `detectTumor` that takes in a grayscale image as a matrix of doubles (i.e. one of the brain scans) and returns a logical value (i.e. 1 for true or 0 for false) to indicate whether a tumor is detected. You are free to design the algorithm for this function however you want -- this is an open-ended problem! We encourage you to use the tools and functions you have already worked with to write your algorithm. If you're stuck, or you have some ideas and just want a chance to talk through them, don't hesitate to come find us in office hours!

Function definition and description

```
function [ hasTumor ] = detectTumor( brain )
%detectTumor Returns whether or not a tumor was found in the image.
%     brain: a matrix of numbers representing a grayscale image of a
%            brain scan.  Bright areas may be tumors and need to
%            be flagged for further testing.  Get the numbers for
%            this matrix by first calling the scan_brain() function
%            provided and then passing in the matrix to this
%            function.
```

Testing

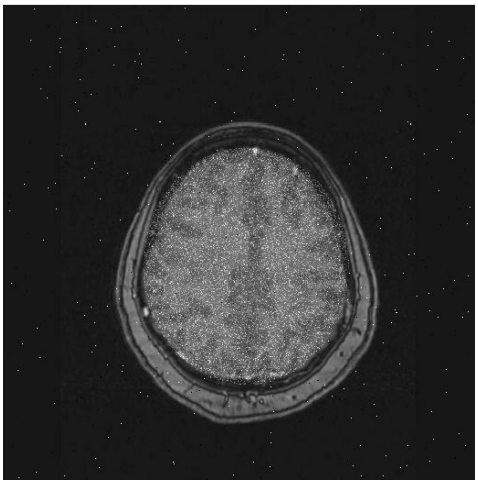
To test your `detectTumor` function, first read a brain scan into Matlab by typing this into the Command Window (this picks brain scan #1):

```
brain1 = scan_brain(1);
```

Show the scan to see if it has a tumor or not:

```
imshow(brain1);
```

Brain scan #1 has no tumor:



Therefore, if you call your `detectTumor` function like this (don't include a semicolon so it displays what it returns):

```
detectTumor(brain1)
```

Matlab should display:

```
ans =

     0
```

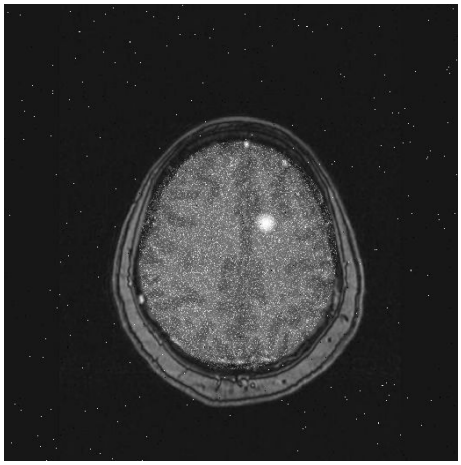
Similarly, if you read in brain scan #79):

```
brain79 = scan_brain(79);
```

Show the scan to see if it has a tumor or not:

```
imshow(brain79);
```

Brain scan #79 has a tumor:



Therefore, if you call your `detectTumor` function like this (don't include a semicolon so it displays what it returns):

```
detectTumor(brain79)
```

Matlab should display:

```
ans =

     1
```

Make sure to test several more brain scans to see if your `detectTumor` function is working properly!

Tips and Tricks

Here are some tips to (hopefully) reduce your frustration on this project:

- It's annoying when MATLAB prints out a giant matrix in the command window - make sure to terminate statements with `;` to suppress output! (If you forget, hit Ctrl-C to make it stop.)
- Make use of the `imshow` function often to check the contents of matrices, but don't do this as part of the functions themselves!
- Make sure to have an `end` statement as the last line of your function.
- If you get tired of typing statements into the Command Window to test your functions, then make a test script! But beware that some images might replace previously generated images in figure windows.

Submission and Grading

This project has five deliverables:

removeNoise.m	heatmap.m	zones.m	detectTumor.m	WatchDisplay.m

Submit the five `.m` files to the Autograder (autograder.io) for grading. The autograder provides you with a final score (out of 100 points) as well as information on *a subset of its test cases*. We provide this autograder to give you a sense of your current progress on the project.

You are limited to 5 submissions on the Autograder per day. After the 5th submission, you are still able to submit, but all feedback will be hidden other than confirmation that your code was submitted. The autograder will only report a subset of the tests it runs. It is up to you to develop tests to find scenarios where your code might not produce the correct results.

You will receive a score for the autograded portion equal to the score of your best submission. Your latest submission with the best score will be the code that is style graded. For example, let's assume that you turned in the following:

Submission #	1	2	3	4
Score	50	100	100	50

Then your grade for the autograded portion would be 100 points (the best score of all submissions) and Submission #3 would be style graded since it is the latest submission with the best score.

Please refer to the syllabus on more information regarding partner groups and general information about the Autograder.

After the due date, this project will be graded in two parts. First, the Autograder will be responsible for evaluating your submission. Second, one of our graders will evaluate your

submission for style and commenting and will provide a maximum score of 10 points. Thus the maximum total number of points on each project is 110 points. Each test case on the Autograder is worth the same amount. Some test cases are public (meaning you can see your output compared to the solution as well as the inputs used to test your submission) while many are private (you are simply are told pass/fail and the reason). The breakdown of points for style and commenting is described in the next section.

Style and Commenting

Maximum Score: 10 points

2 pts - Each submitted file has Name, Partner Uniqname (or "none"), Lab Section Number, and Date Submitted included in a comment at the top
2 pts - Comments are used appropriately to describe your code (e.g. major steps are explained)
2 pts - Indenting and white space are appropriate (including functions are properly formatted)
2 pts - Variables are named descriptively
2 pts - Other factors (Variable names aren't all caps, etc...)