



Projects

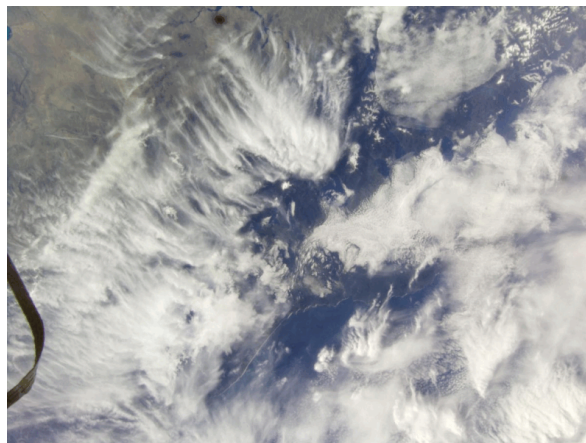
Calculate the speed of the ISS using photos

Calculate the speed of the ISS using image data



Step 1 Introduction

In this project, you will use photos taken by an Astro Pi Flight Unit on the International Space Station (ISS) to estimate the speed at which the ISS orbits the Earth.

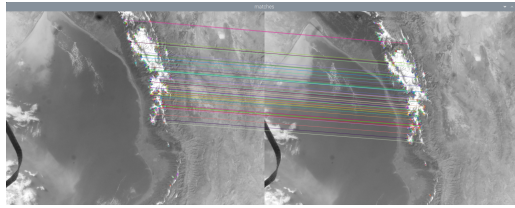


The European Astro Pi Challenge offers young people the amazing opportunity to conduct scientific investigations in space by writing computer programs that run on Raspberry Pi computers aboard the ISS.

You will:

- Extract EXIF data from images
- Use OpenCV to calculate distances between similar features in two images
- Calculate the speed of the ISS

The image below shows two photos taken from the ISS, with lines that connect similar features. By measuring the pixel distance between the features that have moved, you can calculate the speed that the camera was moving, and so work out how fast the ISS is travelling.



To complete this project, you will need:

Hardware

- A computer that can run Python or a web-browser and access to **repl.it** (<https://repl.it.com>).

Software

- Thonny – this project can be completed using the Thonny Python editor, which can be installed on a Linux, Windows, or Mac computer



Install Thonny

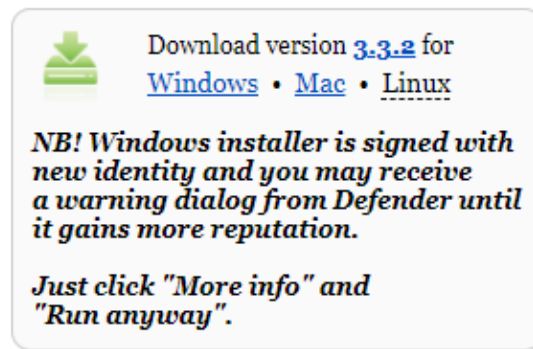
Install Thonny on a Raspberry Pi

- Thonny is already installed on Raspberry Pi OS, but may need to be updated to the latest version
- Open a terminal window, either by clicking the icon in the top left-hand corner of the screen or by pressing the Ctrl+Alt+T keys at the same time
- In the window, type the following to update your OS and Thonny

```
sudo apt update && sudo apt upgrade -y
```

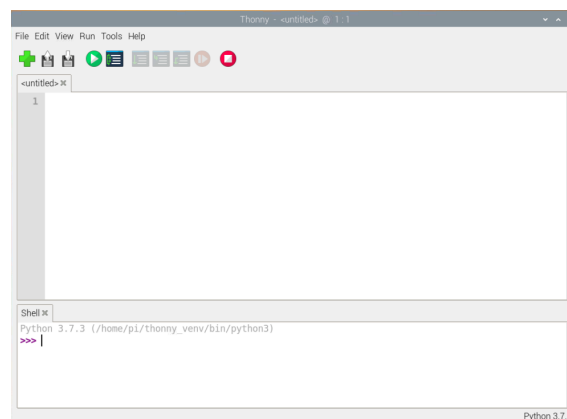
Install Thonny on other operating systems

- On Windows, macOS, and Linux, you can install the latest Thonny IDE or update an existing version
- In a web browser, navigate to **thonny.org** (<https://thonny.org/>)
- In the top right-hand corner of the browser window, you will see download links for Windows and macOS, and instructions for Linux
- Download the relevant files and run them to install Thonny



Opening Thonny

Open Thonny from your application launcher. It should look something like this:



You can use Thonny to write standard Python code. Type the following in the main window, and then click the **Run** button (you will be asked to save the file).

```
print('Hello World!')
```

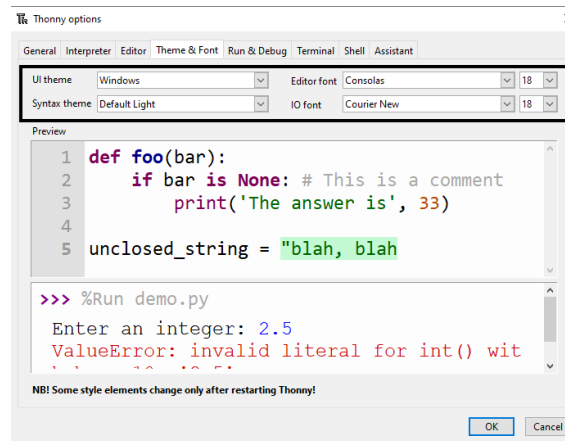


Change the theme and font in Thonny

Thonny allows you to change the theme and font of the software. This feature means that you can increase the font size and change the background and text colours to suit your needs.

To change the theme and font:

- Click on Tools -> Options.
- Click on the 'Theme & Font' tab.
- Click on the drop down boxes for each option until you find the settings that best suit your needs.



- Press OK when you are finished.

Warning: Stick to simple, clean fonts. If you use a handwriting style font then it can make it difficult to read and debug.

Step 2 Use Exif data to find a time difference

An **exchangeable image file format (Exif)** is a standard that sets formats for image and sound files, and various tags that can be stored within the file. These tags can include the time the photo was taken, the location it was taken at, and the type of camera that was used.

To collect Exif data from a photograph, you need a Python library called **exif**.

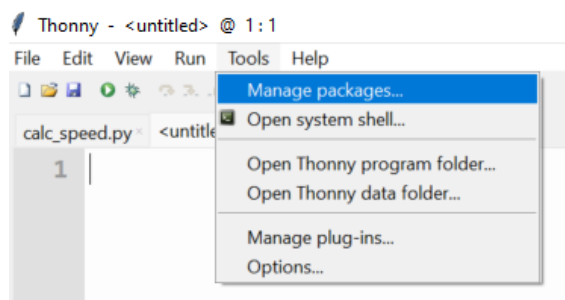
Open the Thonny Python IDE, and click on **Tools > Manage packages**, then search for and install the **exif** library.



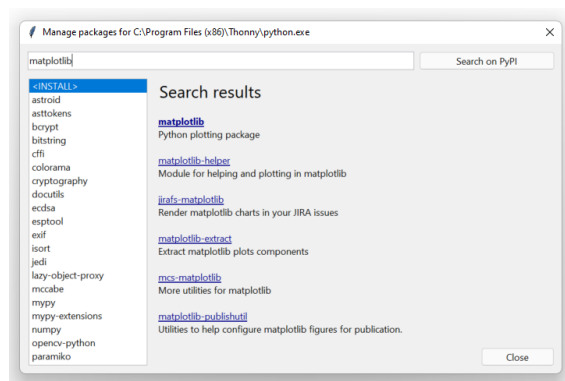
Installing Python packages with Thonny

Installing Python packages with Thonny

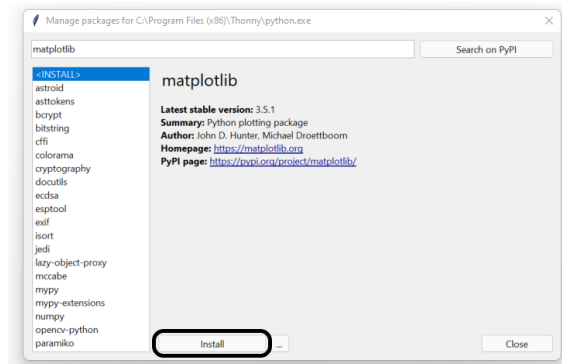
- Open Thonny from your application menu
- Click on **Tools** in the menu bar, and then select **Manage packages**



- Use the search box to find the package you are looking for.



- Select the package you want to install, then click on the **Install** button.



- A window will open showing the progress of your package install. Once completed the package will be available to use in your programs.



In the main code editor, add the following two lines of code. ✓

iss_speed.py

```
1 | from exif import Image
2 | from datetime import datetime
```


Save your file as `iss_speed.py`. ✓

You will need some images taken from the Astro Pi unit on the ISS. You can download a zip file of the photos by clicking on **this link** (<https://rpf.io/p/en/astro-pi-iss-speed-go>). Once the photos have been downloaded, you can right-click on the folder in your **Downloads** and unzip the folder. **Then move the photos to the same place that you have saved your python script.** ✓

Beneath your `import` lines, create a function to find the time that a photo was taken. It will take one argument, which will be the photo's file name. ✓

iss_speed.py

```
1 | from exif import Image
2 | from datetime import datetime
3 |
4 |
5 | def get_time(image):
```

The image needs to be opened and then converted to an `Image` object, which is part of the `exif` package. 


iss_speed.py

```
1 from exif import Image
2 from datetime import datetime
3
4
5 def get_time(image):
6     with open(image, 'rb') as image_file:
7         img = Image(image_file)
```

You can have a look at all the Exif data that is saved in the image file. 

iss_speed.py

```
1 from exif import Image
2 from datetime import datetime
3
4
5 def get_time(image):
6     with open(image, 'rb') as image_file:
7         img = Image(image_file)
8         for data in img.list_all():
9             print(data)
```

You can test out your function using one of the image names that you have downloaded. This needs to be a string. 

iss_speed.py

```
1 from exif import Image
2 from datetime import datetime
3
4
5 def get_time(image):
6     with open(image, 'rb') as image_file:
7         img = Image(image_file)
8         for data in img.list_all():
9             print(data)
10
11
12 get_time('photo_0683.jpg')
```

Run your code, and you should see some output that looks like this:



```
image_width
image_height
make
model
x_resolution
y_resolution
resolution_unit
datetime
y_and_c_positioning
_exif_ifd_pointer
_gps_ifd_pointer
compression
jpeg_interchange_format
jpeg_interchange_format_length
exposure_time
exposure_program
photographic_sensitivity
exif_version
datetime_original
datetime_digitized
components_configuration
shutter_speed_value
brightness_value
metering_mode
flash
maker_note
flashpix_version
color_space
pixel_x_dimension
pixel_y_dimension
_interoperability_ifd_pointer
exposure_mode
white_balance
gps_latitude_ref
gps_latitude
gps_longitude_ref
gps_longitude
```

The data that is needed for this project is `datetime_original`. This can be saved as a string, and then it needs to be converted to a `datetime` object so that calculations can be performed on it.



iss_speed.py

```
1 from exif import Image
2 from datetime import datetime
3
4
5 def get_time(image):
6     with open(image, 'rb') as image_file:
7         img = Image(image_file)
8         time_str = img.get("datetime_original")
9         time = datetime.strptime(time_str, '%Y:%m:%d %H:%M:%S')
10    return time
11
12
13 print(get_time('photo_0683.jpg'))
```

When you run this code, you should see output that looks like this:

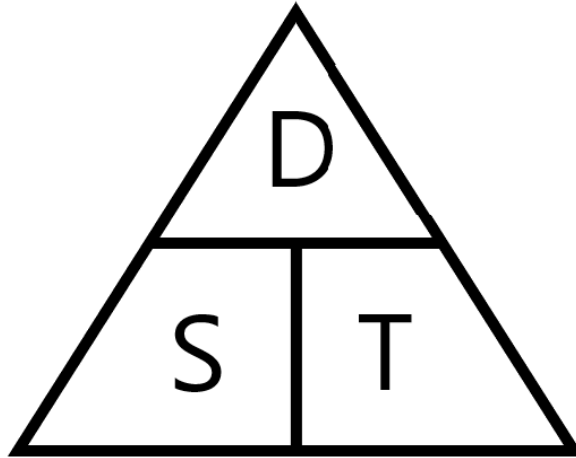
```
2023-05-08 15:31:57
```




Save your project

Step 3 Finding the time difference for two photos

You can calculate the speed an object is traveling at by dividing the distance it moves by the time it takes to move.



So, to calculate the speed of the ISS from photos, you need to know how much time has passed between when the photos were taken.

Remove the call to `print` the results of your `get_time` function.



iss_speed.py

```
1 | from exif import Image
2 | from datetime import datetime
3 |
4 |
5 | def get_time(image):
6 |     with open(image, 'rb') as image_file:
7 |         img = Image(image_file)
8 |         time_str = img.get("datetime_original")
9 |         time = datetime.strptime(time_str, '%Y:%m:%d %H:%M:%S')
10 |     return time
```

Create a new function called `get_time_difference`. It will take two arguments, which will be the file names of the two images.



iss_speed.py

```
13 | def get_time_difference(image_1, image_2):
```

Use your `get_time` function to get the times from the Exif data from each of the two images.



iss_speed.py

```
13 def get_time_difference(image_1, image_2):
14     time_1 = get_time(image_1)
15     time_2 = get_time(image_2)
```

Subtract the two times from each other, and test it by printing.



issc_speed.py

```
13 def get_time_difference(image_1, image_2):
14     time_1 = get_time(image_1)
15     time_2 = get_time(image_2)
16     time_difference = time_2 - time_1
17     print(time_difference)
```

You can run your function by calling it with two different image names.



iss_speed.py

```
13 def get_time_difference(image_1, image_2):
14     time_1 = get_time(image_1)
15     time_2 = get_time(image_2)
16     time_difference = time_2 - time_1
17     print(time_difference)
18
19
20 get_time_difference('photo_0683.jpg', 'photo_0684.jpg')
```

Run your code, and if you have used the two images shown above, you should see output like this:



```
>>> 0:00:09
```

The function needs to return the time in seconds, as an integer. The `datetime` package provides an easy conversion for this.



iss_speed.py

```
13 def get_time_difference(image_1, image_2):
14     time_1 = get_time(image_1)
15     time_2 = get_time(image_2)
16     time_difference = time_2 - time_1
17     return time_difference.seconds
```

To test your code, you can `print` the output of the new function.



iss_speed.py

```
13 def get_time_difference(image_1, image_2):
14     time_1 = get_time(image_1)
15     time_2 = get_time(image_2)
16     time_difference = time_2 - time_1
17     return time_difference.seconds
18
19
20 print(get_time_difference('photo_1754.jpg', 'photo_1755.jpg'))
```

Your output should look something like this, depending on the photos you have chosen.

```
>>> 9
```



Save your project

Step 4 Find matching features

The next step is to find matching features on the two images. There are algorithms to do this in OpenCV.

Delete the `print` statement from your code.



iss_speed.py

```
13 def get_time_difference(image_1, image_2):
14     time_1 = get_time(image_1)
15     time_2 = get_time(image_2)
16     time_difference = time_2 - time_1
17     return time_difference.seconds
```

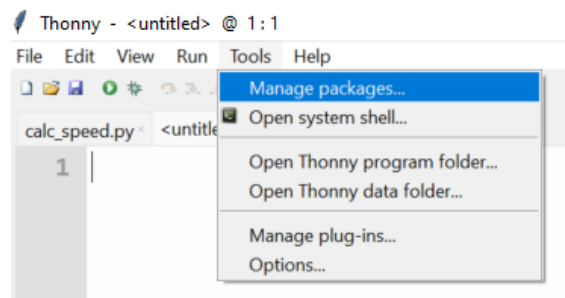
Install the `opencv-python` package in Thonny.



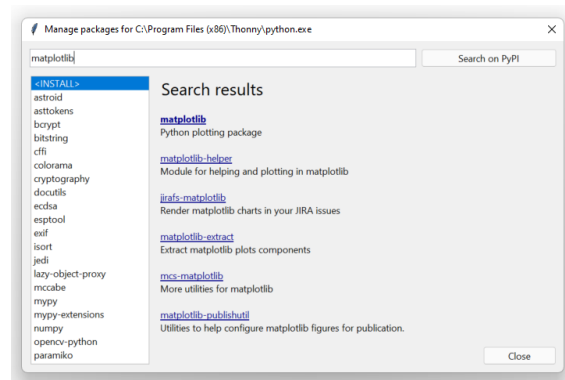
Installing Python packages with Thonny

Installing Python packages with Thonny

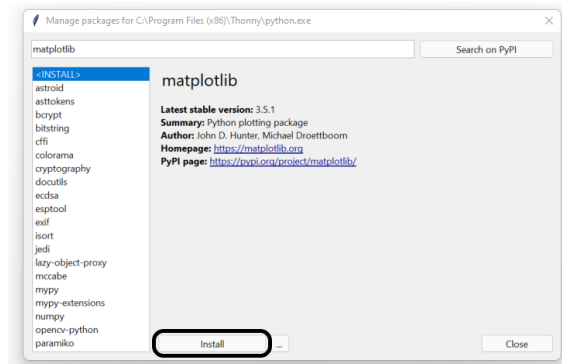
- Open Thonny from your application menu
- Click on **Tools** in the menu bar, and then select **Manage packages**



- Use the search box to find the package you are looking for.



- Select the package you want to install, then click on the **Install** button.



- A window will open showing the progress of your package install. Once completed the package will be available to use in your programs.



Import the `cv2` package and the in-built `math` package at the top of your script.



iss_speed.py

```
1 | from exif import Image
2 | from datetime import datetime
3 | import cv2
4 | import math
```

Delete your call to `print(get_time_difference('photo_07464.jpg', 'photo_07465.jpg'))` on line 22.




Images need to be converted to OpenCV objects so they can be processed, so add a function that takes the two images as arguments and then returns those objects.



iss_speed.py

```
22 | def convert_to_cv(image_1, image_2):
23 |     image_1_cv = cv2.imread(image_1, 0)
24 |     image_2_cv = cv2.imread(image_2, 0)
25 |     return image_1_cv, image_2_cv
```

The OpenCV objects that have been returned can now be used by other classes and methods in the OpenCV package. For this project, the Oriented FAST and Rotated BRIEF (ORB) algorithm can be used. This algorithm will detect **keypoints** in an image or in several images. If the images are similar, then the same keypoints in each image should be detected, even if some features have moved or changed. ORB can also assign **Descriptors** to the keypoints. These will contain information about the keypoint, such as its position, size, rotation, and brightness. By comparing the descriptors between keypoints, the changes from one image to the other can be calculated.

Write a function to find the keypoints and descriptors for the two images. It will take three arguments: the first two are the OpenCV image objects, and the last is the maximum number of features you want to search for. 

iss_speed.py

```
28 def calculate_features(image_1, image_2, feature_number):
29     orb = cv2.ORB_create(nfeatures = feature_number)
30     keypoints_1, descriptors_1 = orb.detectAndCompute(image_1_cv, None)
31     keypoints_2, descriptors_2 = orb.detectAndCompute(image_2_cv, None)
32     return keypoints_1, keypoints_2, descriptors_1, descriptors_2
```

Now you have the keypoints and the descriptors of the keypoints, they need to be matched between the two images. This will tell you whether a keypoint in the first image is the same keypoint in the second image. The simplest way to do this is to use brute force.

A **brute force** algorithm means the computer will try every possible combination. It's like trying to unlock a PIN-protected phone by starting with the PIN **0000**, then moving on to **0001** and keep going until it unlocks or you get to **9999**.


Brute force, in this context, means that you take a descriptor from the first image and try to match it against **all** the descriptors in the second image. A match will either be found or not. Then you take the second descriptor from the first image and repeat the process, and then you keep repeating this process until you have compared every descriptor in the first image to the ones from the second image.

Write a function that takes the two sets of descriptors and tries to find matches by brute force. 

iss_speed.py

```
35 def calculate_matches(descriptors_1, descriptors_2):
36     brute_force = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)
37     matches = brute_force.match(descriptors_1, descriptors_2)
38     matches = sorted(matches, key=lambda x: x.distance)
39     return matches
```

Now that you have your matches, you can run all your functions and have a look at the output.

Assign the two images you want to use, and add function calls to the end of your script to run your functions and print out the matches. 

iss_speed.py

```
42 image_1 = 'photo_0683.jpg'
43 image_2 = 'photo_0684.jpg'
44
45
46 time_difference = get_time_difference(image_1, image_2) # Get time difference between images
47 image_1_cv, image_2_cv = convert_to_cv(image_1, image_2) # Create OpenCV image objects
48 keypoints_1, keypoints_2, descriptors_1, descriptors_2 = calculate_features(image_1_cv, image_2_cv,
49 1000) # Get keypoints and descriptors
50 matches = calculate_matches(descriptors_1, descriptors_2) # Match descriptors
    print(matches)
```

The result should look something like this:

```
[< cv2.DMatch 0x11b34cb30>, < cv2.DMatch 0x11b2db8b0>, < cv2.DMatch 0x11b2dbef0>...
```

This is a list of matches along with the keypoint. It's not very helpful though, so in the next step you can plot the matches on the images and view them.



Save your project

Step 5 Display matching features

Matches can be displayed on both images, with lines linking each of the matched keypoints.

Delete the final `print` call at the end of your script.



iss_speed.py

```
46 | time_difference = get_time_difference(image_1, image_2) # Get time difference between images
47 | image_1_cv, image_2_cv = convert_to_cv(image_1, image_2) # Create OpenCV image objects
48 | keypoints_1, keypoints_2, descriptors_1, descriptors_2 = calculate_features(image_1_cv, image_2_cv,
49 | 1000) # Get keypoints and descriptors
    | matches = calculate_matches(descriptors_1, descriptors_2) # Match descriptors
```

Create a function, below your other functions, that takes the two OpenCV image objects, the keypoints, and the matches as arguments.



iss_speed.py

```
46 | def display_matches(image_1_cv, keypoints_1, image_2_cv, keypoints_2, matches):
47 |
48 |
49 | time_difference = get_time_difference(image_1, image_2) # Get time difference between images
50 | image_1_cv, image_2_cv = convert_to_cv(image_1, image_2) # Create OpenCV image objects
51 | keypoints_1, keypoints_2, descriptors_1, descriptors_2 = calculate_features(image_1_cv, image_2_cv,
52 | 1000) # Get keypoints and descriptors
    | matches = calculate_matches(descriptors_1, descriptors_2) # Match descriptors
```

Next, draw lines between the keypoints where the descriptors match.



iss_speed.py

```
46 | def display_matches(image_1_cv, keypoints_1, image_2_cv, keypoints_2, matches):
47 |     match_img = cv2.drawMatches(image_1_cv, keypoints_1, image_2_cv, keypoints_2, matches[:100], None)
```

The images can now be resized and shown, side by side on your screen, with the lines drawn between the matches.



iss_speed.py

```
46 | def display_matches(image_1_cv, keypoints_1, image_2_cv, keypoints_2, matches):
47 |     match_img = cv2.drawMatches(image_1_cv, keypoints_1, image_2_cv, keypoints_2, matches[:100], None)
48 |     resize = cv2.resize(match_img, (1600, 600), interpolation = cv2.INTER_AREA)
49 |     cv2.imshow('matches', resize)
```

To finish off the function, the script needs to wait until a key is pressed, and then close the image.



iss_speed.py

```
46 def display_matches(image_1_cv, keypoints_1, image_2_cv, keypoints_2, matches):
47     match_img = cv2.drawMatches(image_1_cv, keypoints_1, image_2_cv, keypoints_2, matches[:100], None)
48     resize = cv2.resize(match_img, (1600, 600), interpolation = cv2.INTER_AREA)
49     cv2.imshow('matches', resize)
50     cv2.waitKey(0)
51     cv2.destroyAllWindows('matches')
```

All these functions now need to be called in order, so that you can see the output.

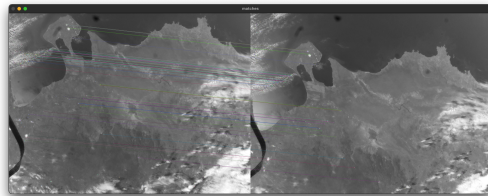


At the bottom of your script, add the following lines:

iss_speed.py

```
time_difference = get_time_difference(image_1, image_2) # Get time difference between images
image_1_cv, image_2_cv = convert_to_cv(image_1, image_2) # Create OpenCV image objects
keypoints_1, keypoints_2, descriptors_1, descriptors_2 = calculate_features(image_1_cv, image_2_cv, 1000)
# Get keypoints and descriptors
matches = calculate_matches(descriptors_1, descriptors_2) # Match descriptors
display_matches(image_1_cv, keypoints_1, image_2_cv, keypoints_2, matches) # Display matches
```


Run your code and you should see an image like the one below. Click in the window and press any key to exit the image view.



Save your project


Step 6 Find matching coordinates

Now that the features that are the same on each image have been matched, the coordinates of those features need to be fetched.

Create a new function that takes the two sets of keypoints and the list of matches as arguments. 

iss_speed.py

```
54 | def find_matching_coordinates(keypoints_1, keypoints_2, matches):
```

Create two empty lists to store the coordinates of each matching feature in each of the images. 

iss_speed.py

```
54 | def find_matching_coordinates(keypoints_1, keypoints_2, matches):  
55 |     coordinates_1 = []  
56 |     coordinates_2 = []
```

The list of matches contains many OpenCV `match` objects. You can iterate through the list to find the coordinates of each match on each image.

Add a `for` loop to fetch the coordinates (`x1, y1, x2, y2`) of each match. 

iss_speed.py

```
54 | def find_matching_coordinates(keypoints_1, keypoints_2, matches):  
55 |     coordinates_1 = []  
56 |     coordinates_2 = []  
57 |     for match in matches:  
58 |         image_1_idx = match.queryIdx  
59 |         image_2_idx = match.trainIdx  
60 |         (x1,y1) = keypoints_1[image_1_idx].pt  
61 |         (x2,y2) = keypoints_2[image_2_idx].pt
```

Next, those coordinates can be added to the two coordinates lists, and the two lists can be returned.



iss_speed.py

```

54 def find_matching_coordinates(keypoints_1, keypoints_2, matches):
55     coordinates_1 = []
56     coordinates_2 = []
57     for match in matches:
58         image_1_idx = match.queryIdx
59         image_2_idx = match.trainIdx
60         (x1,y1) = keypoints_1[image_1_idx].pt
61         (x2,y2) = keypoints_2[image_2_idx].pt
62         coordinates_1.append((x1,y1))
63         coordinates_2.append((x2,y2))
64     return coordinates_1, coordinates_2

```

Add a function call to the bottom of your script to store the outputs of the function. Add a line to print the first pair of coordinates from each list, and then run your program.



iss_speed.py

```

67 time_difference = get_time_difference(image_1, image_2) # Get time difference between images
68 image_1_cv, image_2_cv = convert_to_cv(image_1, image_2) # Create OpenCV image objects
69 keypoints_1, keypoints_2, descriptors_1, descriptors_2 = calculate_features(image_1_cv, image_2_cv,
70 1000) # Get keypoints and descriptors
71 matches = calculate_matches(descriptors_1, descriptors_2) # Match descriptors
72 display_matches(image_1_cv, keypoints_1, image_2_cv, keypoints_2, matches) # Display matches
73 coordinates_1, coordinates_2 = find_matching_coordinates(keypoints_1, keypoints_2, matches)
   print(coordinates_1[0], coordinates_2[0])

```

Your result should look something like this:


```
(666.8699340820312, 629.5451049804688) (661.8932495117188, 1062.512939453125)
```



Save your project

Step 7 Calculate feature distance


With the coordinates of matching features stored, the distance between the coordinates of the matching features can be calculated. This will be the distance on the images though, so it will need to be converted to the equivalent kilometer distance on Earth, and then divided by the time difference between the photos, to calculate the speed.

Create a function to calculate the average distance between matching coordinates. Call it `calculate_mean_distance`. It takes two arguments, which will be the two coordinates lists. 

iss_speed.py

```
67 | def calculate_mean_distance(coordinates_1, coordinates_2):
```

The Python `zip` function will take items from two lists and join them together. So the 0th item from the first list and the 0th item from the second list are combined together. Then the 1st items from each of the lists are combined together. The zipped list object can easily be converted back to a single simple list.

Start by creating a variable to store the sum of all the distances between coordinates and call it `all_distances`. Next, you can `zip` the two lists, and then convert the zipped object back to a list. 

iss_speed.py

```
67 | def calculate_mean_distance(coordinates_1, coordinates_2):  
68 |     all_distances = 0  
69 |     merged_coordinates = list(zip(coordinates_1, coordinates_2))
```

To see what has happened here, you can add some `print` calls to see the details of the lists.

Add three `print` calls to see an item in `coordinates_1`, `coordinates_2`, and `merged_coordinates`. 

iss_speed.py

```
67 | def calculate_mean_distance(coordinates_1, coordinates_2):  
68 |     all_distances = 0  
69 |     merged_coordinates = list(zip(coordinates_1, coordinates_2))  
70 |     print(coordinates_1[0])  
71 |     print(coordinates_2[0])  
72 |     print(merged_coordinates[0])
```

Add a call to your function at the bottom of your script.



```
81 | average_feature_distance = calculate_mean_distance(coordinates_1, coordinates_2)
```

When you run the code, you should see output that looks like this:

```
(666.8699340820312, 629.5451049804688)
(661.8932495117188, 1062.512939453125)
((666.8699340820312, 629.5451049804688), (661.8932495117188, 1062.512939453125))
```

Hopefully you can see that the **x** and **y** coordinate from each feature, from each image, have been combined. This will allow for easy iteration over the new list.

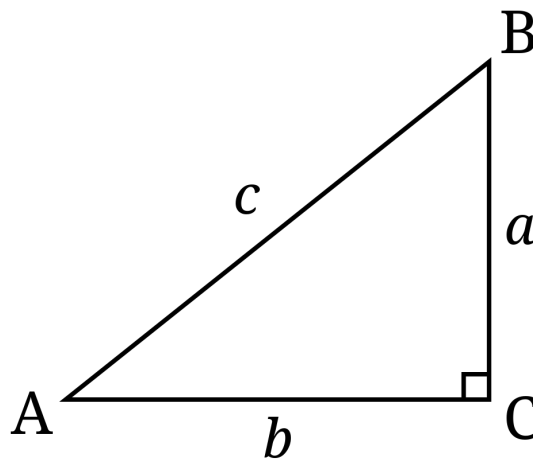
Delete the **print** calls and add a **for** loop to iterate over the **merged_coordinates** and calculate the differences between the **x** and **y** coordinates in each image.



iss_speed.py

```
67 | def calculate_mean_distance(coordinates_1, coordinates_2):
68 |     all_distances = 0
69 |     merged_coordinates = list(zip(coordinates_1, coordinates_2))
70 |     for coordinate in merged_coordinates:
71 |         x_difference = coordinate[0][0] - coordinate[1][0]
72 |         y_difference = coordinate[0][1] - coordinate[1][1]
```

Look at the following image:



The distance between points **A** and **B** is the length of the line **c**. This is called the hypotenuse. Using the **math** package, the hypotenuse (**hypot**) can be calculated.

Calculate the distance between the two points and add them to the `all_distances` variable.



iss_speed.py

```
67 def calculate_mean_distance(coordinates_1, coordinates_2):
68     all_distances = 0
69     merged_coordinates = list(zip(coordinates_1, coordinates_2))
70     for coordinate in merged_coordinates:
71         x_difference = coordinate[0][0] - coordinate[1][0]
72         y_difference = coordinate[0][1] - coordinate[1][1]
73         distance = math.hypot(x_difference, y_difference)
74         all_distances = all_distances + distance
```

Return the average distance between the features by dividing `all_distances` by the number of feature matches, which is the length of the `merged_coordinates` list.



iss_speed.py

```
67 def calculate_mean_distance(coordinates_1, coordinates_2):
68     all_distances = 0
69     merged_coordinates = list(zip(coordinates_1, coordinates_2))
70     for coordinate in merged_coordinates:
71         x_difference = coordinate[0][0] - coordinate[1][0]
72         y_difference = coordinate[0][1] - coordinate[1][1]
73         distance = math.hypot(x_difference, y_difference)
74         all_distances = all_distances + distance
75     return all_distances / len(merged_coordinates)
```

Add a function call at the bottom of your code to calculate the average distance, and then print the result.



iss_speed.py

```
time_difference = get_time_difference(image_1, image_2) # Get time difference between images
image_1_cv, image_2_cv = convert_to_cv(image_1, image_2) # Create OpenCV image objects
keypoints_1, keypoints_2, descriptors_1, descriptors_2 = calculate_features(image_1_cv, image_2_cv, 1000)
# Get keypoints and descriptors
matches = calculate_matches(descriptors_1, descriptors_2) # Match descriptors
display_matches(image_1_cv, keypoints_1, image_2_cv, keypoints_2, matches) # Display matches
coordinates_1, coordinates_2 = find_matching_coordinates(keypoints_1, keypoints_2, matches)
average_feature_distance = calculate_mean_distance(coordinates_1, coordinates_2)
print(average_feature_distance)
```

When you run your code, you should get an answer like `504.08973470622516`.



Save your project

Step 8 Calculate average speed

Now that the feature distances have been calculated for the two photographs, this needs converting into how far apart the features actually are on Earth. Once that has been calculated, the time difference between the two photos can be used to calculate the speed of the camera, and therefore the speed of the ISS.

Delete your `print` call at the bottom of your script.



Create a new function to calculate the speed of the ISS. It should take the `feature_distance`, a `GSD` factor, and the `time_difference` as arguments.



iss_speed.py

```
78 | def calculate_speed_in_kmps(feature_distance, GSD, time_difference):
```

You can use **this website** (<https://www.3dflow.net/ground-sampling-distance-calculator/>) to calculate the scaling factor between distance in pixels and distance on Earth. The Ground Sample Distance (GSD) is given in centimeters/pixels. You need the distance in kilometers though, and there are 100,000 centimeters in a kilometer. If you are using photos of a different resolution from the examples here, or photos taken with a different camera, you will need to recalculate the GSD.

Calculate the distance by multiplying the feature distance in pixels by the `GSD` and then divide it all by 100,000.



iss_speed.py

```
78 | def calculate_speed_in_kmps(feature_distance, GSD, time_difference):  
79 |     distance = feature_distance * GSD / 100000
```

The speed can then be calculated by dividing by the `time_difference` between the two images, and the speed returned.



iss_speed.py

```
78 | def calculate_speed_in_kmps(feature_distance, GSD, time_difference):  
79 |     distance = feature_distance * GSD / 100000  
80 |     speed = distance / time_difference  
81 |     return speed
```

The Ground Sampling Distance (<https://www.3dflow.net/ground-sampling-distance-calculator>) site gives a GSD of 12648 for the High Quality Camera on the ISS taking photos of the resolution of these examples (4056 x 3040). If you use different photos, you will need to recalculate the GSD if they have a different resolution or were taken with a different camera.

Call your function and then print the result at the end of your program.



iss_speed.py

```
84 | time_difference = get_time_difference(image_1, image_2) # Get time difference between images
85 | image_1_cv, image_2_cv = convert_to_cv(image_1, image_2) # Create OpenCV image objects
86 | keypoints_1, keypoints_2, descriptors_1, descriptors_2 = calculate_features(image_1_cv, image_2_cv,
87 | 1000) # Get keypoints and descriptors
88 | matches = calculate_matches(descriptors_1, descriptors_2) # Match descriptors
89 | display_matches(image_1_cv, keypoints_1, image_2_cv, keypoints_2, matches) # Display matches
90 | coordinates_1, coordinates_2 = find_matching_coordinates(keypoints_1, keypoints_2, matches)
91 | average_feature_distance = calculate_mean_distance(coordinates_1, coordinates_2)
92 | speed = calculate_speed_in_kmps(average_feature_distance, 12648, time_difference)
    | print(speed)
```

With the two images used in this project, a value of **7.084** is returned, which is not far from the 7.66kmps speed that the ISS actually travels at.



Save your project

Step 9 Improving your program

To refine your code and perhaps calculate a more accurate estimation of the speed of the ISS, you could try and investigate any of the following:

- Use a different feature detection **algorithm in OpenCV** (https://docs.opencv.org/3.4/db/d27/tutorial_py_table_of_contents_feature2d.html)
- Choose a different number of feature matches to use
- Use more than two photos
- Check how long a photo takes to be written to disk to get a more accurate value for the time between photos
- Does the curvature of the Earth have an effect on the actual distance values travelled?
- Does the height of the identified feature also have an effect?
- Does the angle of motion (diagonally across the frame) have a impact that needs to be corrected for?
 - If your matched features are clouds, can you compensate for the fact that they may be moving too?

Step 10 What next?

Published by Raspberry Pi Foundation (<https://www.raspberrypi.org>) under a **Creative Commons** license (<https://creativecommons.org/licenses/by-sa/4.0/>).

View project & license on GitHub (<https://github.com/RaspberryPiLearning/astropi-iss-speed>)