Write-up for the Project 1: Search and Sample Return

Notebook Analysis

I have run the notebook as instructed and provided the following changes:

1. **Color thresholds.**

I have left the `color_thresh` method unchanged and will continue to use it for the determination of the navigational path with the default filter (160, 160, 160). In a real application we should also consider changing the name of the method to something that is more clear that it relates to the filtering of the navigational path. For this project I have left the name the same.

In addition I have defined two additional methods:

```
def obstacle_thresh(img, rgb_thresh=(160, 160, 160)):
    # Create an array of zeros same xy size as img, but single channel
    color_select = np.zeros_like(img[:,:,0])
    # Require that each pixel be above all three threshold values in RGB
    # above_thresh will now contain a boolean array with "True"
    # where threshold was met
    above_thresh = (img[:,:,0] < rgb_thresh[0]) \
                & (img[:,:,1] < rgb_thresh[1]) \
                & (img[:,:,2] < rgb_thresh[2])
    # Index the array of zeros with the boolean array and set to 1
    color_select[above_thresh] = 1
    # Return the binary image
    return color_select
```

This method returns the opposite of the `color_thresh` in the sense that the values in the return matrix are 1 for non-navigable areas, meaning obstacles. In principle this could also have been implemented by simply subtracting 1 from the result of the `color_thresh` (thus inverting the result), but by implementing it separately we have the advantage that we can experiment with different threshold values, distinct from the ones used in the `color_thresh`.

The second method is used to identify rocks:

```
def rock_thresh(img):
    # Create an array of zeros same xy size as img, but single channel
    color_select = np.zeros_like(img[:,:,0])
    #
    # threshold values:
    low = (120, 100, 0)
    high = (180, 160, 25)
    # Require that each pixel be above all three threshold values in RGB
    # above_thresh will now contain a boolean array with "True"
    # where threshold was met
    r = (img[:,:,0] >= low[0]) & (img[:,:,0] <= high[0])
    g = (img[:,:,1] >= low[1]) & (img[:,:,1] <= high[1])
    b = (img[:,:,2] >= low[2]) & (img[:,:,2] <= high[2])
    all = r & g & b
    # Index the array of zeros with the boolean array and set to 1
    color_select[all] = 1
    # Return the binary image
    return color_select
```

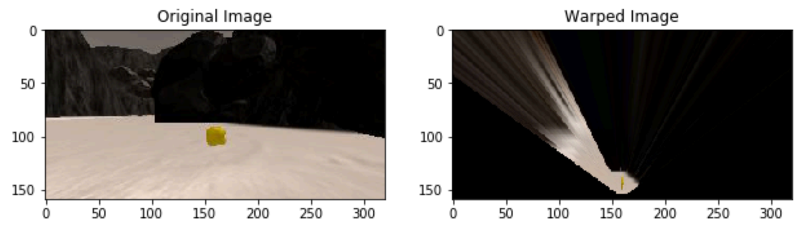
For this method we use two filters, one that will be used as low filter (will filter all pixels that have values above) and one that is used as a high filter (will filter all pixels that have values bellow). In this way we filter the pixels that are between these two ranges. To make things easier to read we do the filtering per channel (RGB) and then merge them.

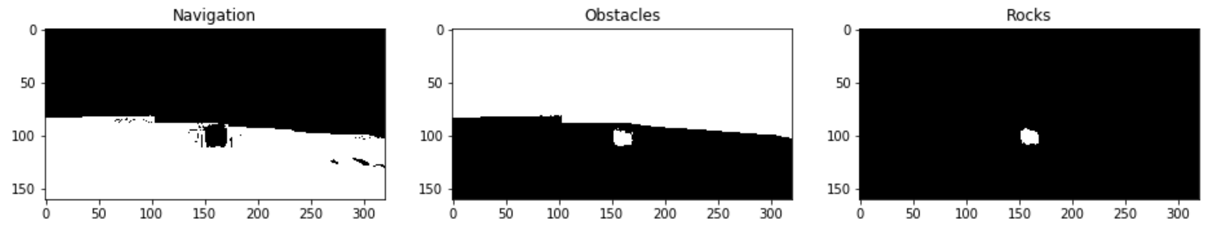
This approach is suitable for this case where the colour of the rocks are very distinct and we can do this type of filtering. In real life it is significantly more accurate to convert to HSV (Hue, Saturation, Value) and to filter in that space as real object have very large dynamic representations in RGB space in different lighting conditions (consider the difference in RGB values for a low illuminated object vs. a well lit one; it will make it almost impossible to do the filter in RGB as it will include too many combinations in that space that are not really representative for the image of that object). But in this case the filter seems to work fine in RGB space and it is faster.

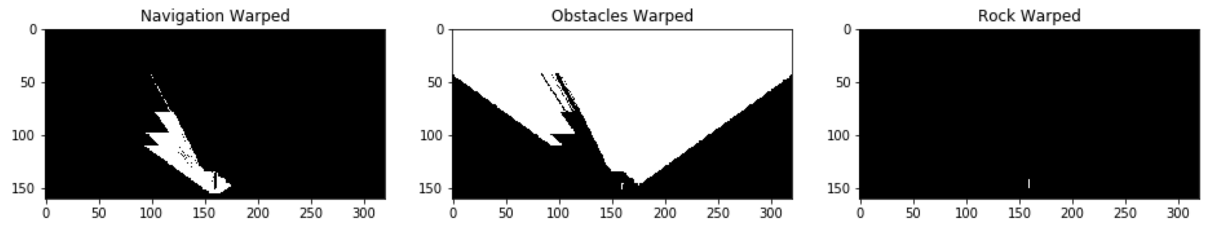
The filter I used is (RGB):

```
    low = (120, 100, 0)
    high = (180, 160, 25)
```

Here are some graphs that show how these 3 functions work:







One of the interesting thing to notice is that very close object on the warping image appear very small (due to perspective distortion) and it would be much more accurate for the rocks detection to first perform the filter, then the warping as the resulting image after warping will contain only a very small smear representing the rock and it might be possible for the filter to miss the colors.

2. process_image

The implementation of process_image is pretty straightforward and follows the steps suggested in the commentaries, except for swapping the filter and the warping of the images as motioned in the last paragraph.

I first to the transformation of the images calling the newly created methods to have images for navigation, obstacles and rocks:

```
    nav = perspect_transform(color_thresh(img), source, destination)
    obs = perspect_transform(obstacle_thresh(img), source, destination)
    rock = perspect_transform(rock_thresh(img), source, destination)
```

Convert to rover centric coordinates:

```
    nav_xpix, nav_ypix = rover_coords(nav)
    obs_xpix, obs_ypix = rover_coords(obs)
    rock_xpix, rock_ypix = rover_coords(rock)
```

Then convert to world coordinates:

```
    nav_x_world, nav_y_world = pix_to_world(nav_xpix, nav_ypix,
                                            data.xpos[data.count],
                                            data.ypos[data.count],
                                            data.yaw[data.count],
                                            200,
                                            10)

    obs_x_world, obs_y_world = pix_to_world(obs_xpix, obs_ypix,
                                            data.xpos[data.count],
                                            data.ypos[data.count],
                                            data.yaw[data.count],
                                            200,
                                            10)

    rock_x_world, rock_y_world = pix_to_world(rock_xpix, rock_ypix,
                                            data.xpos[data.count],
                                            data.ypos[data.count],
                                            data.yaw[data.count],
                                            200,
                                            10)
```

We update the world map colouring:

```
    data.worldmap[obs_y_world, obs_x_world, 0] += 1
    data.worldmap[rock_y_world, rock_x_world, 1] += 1
    data.worldmap[nav_y_world, nav_x_world, 2] += 1
```

I have not changed in the notebook the rest of the code that creates the mosaic image.

The resulting video is included with the submission.

Autonomous Navigation and Mapping

1. perception step

I first copied the two additional functions `obstacle_thresh` and `rock_thresh` from the Jupyter notebook to the perception.py file.

The `perception_step()` function is implemented similar to the one in the notebook, adjusting for the different access to the rover data and also keeping the threshold mappings in some interim variables with the purpose of including them in the `vision_image`.

First, we define the perspective transform parameters:

```
    source = np.float32([[14, 140], [301 ,140],[200, 96], [118, 96]])
    imgshape = Rover.img.shape
    dst_size = 5
    bottom_offset = 6
   destination = np.float32([[imgshape[1]/2 - dst_size, imgshape[0] - bottom_offset], \
                            [imgshape[1]/2 + dst_size, imgshape[0] - bottom_offset], \
                            [imgshape[1]/2 + dst_size, imgshape[0] - 2*dst_size - bottom_offset], \
                            [imgshape[1]/2 - dst_size, imgshape[0] - 2*dst_size - bottom_offset], \
                            ])
```

And apply the functions created earlier to determine the filters images and their warped equvaluents:

```
    nav_th = color_thresh(Rover.img)
    obs_th = obstacle_thresh(Rover.img)
    rock_th = rock_thresh(Rover.img)

    nav = perspect_transform(nav_th, source, destination)
    obs = perspect_transform(obs_th, source, destination)
    rock = perspect_transform(rock_th, source, destination)
```

The filters are added to the vision_image. Rocks will appear a little funny: if the background is an obstacle (blue) they will be violet (blue + green) , if the background is the navigation path (red) they will be yellow (red + green).

```
    Rover.vision_image[:,:,0] = obs_th * 255
    Rover.vision_image[:,:,1] = rock_th * 255
    Rover.vision_image[:,:,2] = (nav_th * 255
```

Similar to the processing in the notebook we convert to rover-centric coordinates:

```
    nav_xpix, nav_ypix = rover_coords(nav)
    obs_xpix, obs_ypix = rover_coords(obs)
    rock_xpix, rock_ypix = rover_coords(rock)
```

And then to the world map coordinates:

```
    nav_x_world, nav_y_world = pix_to_world(nav_xpix, nav_ypix,
                                       Rover.pos[0],
                                       Rover.pos[1],
                                       Rover.yaw,
                                       200,
                                       10)

    obs_x_world, obs_y_world = pix_to_world(obs_xpix, obs_ypix,
                                       Rover.pos[0],
                                       Rover.pos[1],
                                       Rover.yaw,
                                       200,
                                       10)

    rock_x_world, rock_y_world = pix_to_world(rock_xpix, rock_ypix,
                                       Rover.pos[0],
                                       Rover.pos[1],
                                       Rover.yaw,
                                       200,
                                       10)
```

Finally we update the worldmap with the information we have detected (navigation, obstacles, rocks). In order to keep the accuracy high, as per the tips provided we only update the worldmap if both the roll and pitch angle of the rover are within a low tolerance (ex. 0.5 degrees):

```
    if abs(angleto180(Rover.pitch) < 0.5) and abs(angleto180(Rover.roll) < 0.5)
                                          and abs(Rover.steer) < 7.5:
          Rover.worldmap[obs_y_world, obs_x_world, 0] += 1
          Rover.worldmap[rock_y_world, rock_x_world, 1] += 1
          Rover.worldmap[nav_y_world, nav_x_world, 2] += 1
```

Where angleto180 is a function that returns an angle in the -180 to 180 range instead of the 0 to 360 range (so that we can easily compare with the tolerance threshold:

```
def angleto180(angle):
      if angle > 180.0:
            return angle - 360.0
      else:
            return angle
```

And determine the polar-coordinates equivalent of the navigational coordinates to be used later in the navigation:

```
    Rover.nav_dists, Rover.nav_angles = to_polar_coords(nav_xpix, nav_ypix)
```

With these changes in place the rover seems to be able to do some mediocre mapping and identification of rocks, but no pickup and also I have had situations where the rover starts moving in circles…

Time to move to the `decision_step()`.

2. decision_step()

The `decision_step()` Is basically a state-transition graph that moves, depending on the environment, between the following "modes":

- ‘start’ – it is the original “mode”; used only to store the original position (so that we can return) then transitions to “explore”
- ‘explore’ – the main mode that the rover will be most of the time; it tries to move through the environment and might transition to ‘collect’, ‘return’ or ‘stop’ mode
- ‘stop’ – is the mode where the rover needs to reorient due to lack of clear path ahead; will normally transition to ‘explore’
- ‘collect’ is the mode that is employed when a sample is detected, and we want to collect it; it will get closer to the sample and then initiate the ‘pick’
- ‘pick’ mode is only created due to some quirks in how the simulator / rover interaction operates; basically this mode simply waits for 10 seconds without doing anything – letting the pick command sent to the rover do its work; triggers ‘done_picking’ when finished
- ‘done_picking’ is used to re-orient the robot to the same heading as before the ‘collect’ mode started so that the exploration can continue roughly in the same direction as before the collection; this way we avoid being positioned after pick in a way that will stop us from exploring a part of the map;
- ‘return’ is employed when all the conditions for the tasks are complete and the rover will try to navigate without any other interruptions until is close enough to the original starting point when the ‘finish’ mode is employed
- ‘finish’ is the last mode of the robot; practically all the work is done and we’re back close to where we stated so we’re just putting the brakes and call it a day.

For states here is the main `decision_step()`:

```
def decision_step(Rover):

      # Implement conditionals to decide what to do given perception data
      # Here you're all set up with some basic functionality but you'll need to
      # improve on this decision tree to do a good job of navigating autonomously!

      if Rover.nav_angles is None:
            Rover.throttle = Rover.throttle_set
            Rover.steer = 0
            Rover.brake = 0

      else:
            # Check for Rover.mode status
            if Rover.mode == 'start':
                  # save start position so we can return
                  Rover.return_pos = Rover.pos
                  Rover.mode = 'explore'

            elif Rover.mode == 'explore':
                  Rover = explore_mode(Rover)

            elif Rover.mode == 'stop':
                  Rover = stop_mode(Rover)

            elif Rover.mode == 'collect':
                  Rover = collect_mode(Rover)

            elif Rover.mode == 'pick':
                  sleep(10)   # we need this to deal with some dalays in operation
                  Rover.mode = 'done_pick' # finished picking
                  # otherwise just wait for the pickup to complete
            elif Rover.mode == 'done_pick':
                  Rover = done_pick_mode(Rover)

            elif Rover.mode == 'return':
                  Rover = return_mode(Rover)

            elif Rover.mode == 'finish':
                  # we're done; stop the rover
                  Rover.brake = 5
                  Rover.thorottle = 0


      # If in a state where want to pickup a rock send pickup command
      #if Rover.near_sample and Rover.vel == 0 and not Rover.picking_up:
      #      Rover.send_pickup = True

      return Rover      # Implement conditionals to decide what to do given
perception data
```

The `explore_mode()` is responsible for the main navigation of the rover. First, if there is a sample in sight (determined by the `Rover.samp_angles` that are updated by the perception step) then it transitions to the collect mode and slightly breaks the rover. It also saves the current yaw of the rover so that later in the done_picking we can reorient the rover to roughly the same orientation before continuing the exploration.

If in explore mode we have more than 6 samples collected, and more than 95% of the mapped then we transition to the ‘return’ mode.

Otherwise we deal with normal navigation: if we don’t have enough navigation path ahead we transition to the ‘stop’ mode that will take us from there. If there is sufficient path ahead we will move along that path <u>but using a bias to move towards left side</u>. This way the rover will naturally follow the landscape edge preferring to turn left when other things are equal. Given the structure of the map this is a very simple way of ensuring that we navigate the whole environment. That approach is though simple and sometimes, given the environment situation the rover can end up going in circles (happens very rarely if spawned in the middle of the map and all path is clear to create a circle) or sometimes might miss to navigate to a tight corner (this is even more rare). The way the left bias is implemented is by adding to the median navigation steer a weighted component produced from the standard deviation of the available navigational angles. When navigation is narrow the left bias will be less pronounced. When the standard deviation is large (meaning we have wide open road ahead) the bias will be more pronounced, and the rover will ‘keep left’ in that wide open road. The expectation is that we will return on the same road and we’ll scan the other side on that time. The weighted parameter for the left bias has been determined by trial-and-error to provide a balance between aggressive left bias that might drive the rover too close to the edge of the path and too low bias that might make the rover to miss certain exploration of areas. Before passing the steer we also use another multiplication parameter that is acting as a P parameter in a PID controller. Without this parameter the rover tends to wobble left to right in the path and has a negative impact on the accuracy of mapping too. The parameter was also determined via trial and error. Here is the complete listing of the method:

```
#
# Does the work for the 'explore' mode
# As long as there is a path ahead it will try to move on that path
# with a slight bias towards the left side; this is a simple way of
# avoiding going in circles and exploring the whole map
# if too litle navigation path is available it will go in 'stop' mode
# if a rock is detected it will stitch to 'collect' mode
#
def explore_mode(Rover):
      # check if there are any samples to pick and they are on the left side
      # if they are on the right side they will be picked up when we come back
      if len(Rover.samp_angles) > 3:
            Rover.throttle = 0
            Rover.brake = 1   # slight brake
            # we need to store the positon before pickup otherwise there is a
            # very high risk that after pickup we will not be able to continue on
            # the same path
            Rover.save_yaw = Rover.yaw
            Rover.steer = np.clip(np.mean(Rover.samp_angles) * 180/np.pi, -15, 15)
            Rover.mode = 'collect'

      elif Rover.samples_collected > 5 and Rover.perc_mapped > 0.95:
            Rover.mode = 'return'

      # Check the extent of navigable terrain
      elif len(Rover.nav_angles) >= Rover.stop_forward:
            # If mode is forward, navigable terrain looks good
            # and velocity is below max, then throttle
            if Rover.vel < Rover.max_vel:
                  Rover.throttle = Rover.throttle_set
            else: # Else coast
                  Rover.throttle = 0
            Rover.brake = 0
            # Set steering to average angle clipped to the range +/- 15
            nav_mean = np.mean(Rover.nav_angles)
            nav_std = np.std(Rover.nav_angles)
            steer = nav_mean + nav_std / 2.25 # slight bias to drive on the left
            steer = steer * 0.6 # acts like a P factor (PID); reduces overshooting
            Rover.steer = np.clip(steer * 180/np.pi, -15, 15)
      # If there's a lack of navigable terrain pixels then go to 'stop' mode
      elif len(Rover.nav_angles) < Rover.stop_forward:
                  # Set mode to "stop" and hit the brakes!
                  Rover.throttle = 0
                  # Set brake to stored brake value
                  Rover.brake = Rover.brake_set
                  Rover.steer = 0
                  Rover.mode = 'stop'
      # make sure we return the updated Rover
      return Rover
```

The stop_mode() is responsible to getting the rover out from a dead-end. It will stop the rover and then try to turn it until there is enough navigational path to continue the exploration. What is important here is that the steer angle has to be towards right so that the exploration continues along the same route. If the steer would have been to the left then there would have been a very high chance that we will emerge from a dead-end positioned so that when applying the left drive bias we might not have enough room to continue towards the same direction before entering the dead-end but return on the path we came and miss entire regions from the map. The method is not working very well for obstacles that are not fully on the ground – for instance some rocks are really tricky to deal with and frankly the best strategy is to stay close to the edge.

```
def stop_mode(Rover):
	# If we're in stop mode but still moving keep braking
	if Rover.vel > 0.2:
		Rover.throttle = 0
		Rover.brake = Rover.brake_set
		Rover.steer = 0
	# If we're not moving (vel < 0.2) then do something else
	elif Rover.vel <= 0.2:
		# Now we're stopped and we have vision data to see if there's a path 
forward
		if len(Rover.nav_angles) < Rover.go_forward:
			Rover.throttle = 0
			Rover.brake = 0      # Release the brake to allow turning
			Rover.steer = -15    # we trun to the left to keep aligned with
			                     # the fact that we drive on the left side
		# If we're stopped but see sufficient navigable terrain in front then go!
		if len(Rover.nav_angles) >= Rover.go_forward:
			Rover.throttle = Rover.throttle_set
			Rover.brake = 0
			# Set steer to mean angle
			Rover.steer = np.clip(np.mean(Rover.nav_angles * 180/np.pi), -15, 
15)
			Rover.mode = 'explore'
	# make sure we return the updated Rover
	return Rover
```

The `collect_mode()` is called when the rover has a rock in sight and will like to collect it. The implementation is pretty simple and simply tries to get close to the rock on a straight path with a reduced speed until the `near_sample` indicator in the rover becomes True. This works fine in almost all cases, but those where the rock is quite far and there is some difficult terrain between the rover and the rock. In that case the straight path to rock approach might not be the best and the rover might struggle or even get stuck trying to reach the sample. That happens though very rarely. Once the `near_sample` indicator is detected, it sends the pickup signal to the rover and then transitions to the 'pick' mode where it simply waits for 10s for the pickup to complete. The reason we have to do this is because once the `send_pickup` message is set in the Rover we need to pass back the control to the simulator to process that signal and we will get back in the `decision_step` soon after that, but we will have to be in another mode where we can simply just wait 10s and then finish the pickup sequence. The code for this method is:

```
def collect_mode(Rover):
	# near sample?
	if Rover.near_sample > 0:
		Rover.brake = 5
		Rover.throttle = 0
		Rover.send_pickup = True
		Rover.mode = 'pick'

	# not yet; move closer
	else:
		Rover.brake = 0
		if Rover.vel < Rover.max_vel / 2.0:             # go slower
			Rover.throttle = Rover.throttle_set
		else: # Else coast
			Rover.throttle = 0
		if len(Rover.samp_angles) > 0:
			mean_dir = np.mean(Rover.samp_angles)
			steer_dir = mean_dir * 180 / np.pi
```

```
                Rover.steer = np.clip(steer_dir, -15, 15)
        # make sure we return the updated Rover
        return Rover
```

The `done_pickup_mode()` is triggered after waiting for 10s in 'pick' mode and the purpose of this status is to allow the rover to return to the original heading so that the exploration continues in the same direction as before the collection happened. This method uses the `save_yaw` member of the Rover to reorient and then reengages the 'explore' mode:

```
def done_pick_mode(Rover):
        # we finished picking and we need to reorient the rover in the direction
        # we were initially
        Rover.throttle = 0
        Rover.brake = 0
        #print('save: %4.2f; actual: %4.2f; comm: %4.2f' % (Rover.save_yaw, Rover.yaw, 
Rover.save_yaw - Rover.yaw))
        if abs(Rover.save_yaw - Rover.yaw) > 2.0 :
              Rover.steer = np.clip(Rover.save_yaw - Rover.yaw, -15, 15)
        else:
              Rover.steer = 0
              Rover.mode = 'explore'
    # make sure we return the updated Rover
        return Rover
```

Finally the `return_mode()` is slightly similar to `explore_mode()` except that this time the left bias is no longer necessary – given the configuration of the map we would either go forward and reach the centre or reach a dead-end, use the 'stop' to turn around, briefly transition to 'explore' (stop only transitions to 'explore') before transitioning to 'return' to move straight towards the centre. Different from `explore_mode()` this method is calculating the distance between the current rover's position and the spawn position saved at the beginning in 'start' mode. If this distance is smaller than a number, we deem that we are close to the initial position and transition to 'finish' where process ends.

```
def return_mode(Rover):
        # calculates distance from start
        dx = Rover.pos[0] - Rover.return_pos[0]
        dy = Rover.pos[1] - Rover.return_pos[1]
        dist = np.sqrt(dx**2 + dy**2)
        if dist < 10:
              # we are close enough
              Rover.mode = 'finish'

        elif len(Rover.nav_angles) >= Rover.stop_forward:
              # navigate forward - will get them eventually
              if Rover.vel < Rover.max_vel:
                    Rover.throttle = Rover.throttle_set
              else: # Else coast
                    Rover.throttle = 0
              Rover.brake = 0
              # Set steering to average angle clipped to the range +/- 15
              nav_mean = np.mean(Rover.nav_angles)
              steer = nav_mean * 0.6 # acts like a P factor (PID); reduces overshooting
              Rover.steer = np.clip(steer * 180/np.pi, -15, 15)
        # If there's a lack of navigable terrain pixels then go to 'stop' mode
        elif len(Rover.nav_angles) < Rover.stop_forward:
                    # Set mode to "stop" and hit the brakes!
```

```
                Rover.throttle = 0
                # Set brake to stored brake value
                Rover.brake = Rover.brake_set
                Rover.steer = 0
                Rover.mode = 'stop'
    # make sure we return the updated Rover
    return Rover
```

3. Other updates

To support the current design I have made a few small adjustments to code in `supporting_functions.py` and `drive_rover.py`. Specifically:

In `create_output_images(Rover)` I have added some lines of code and show the current mode of the rover in the display. This is very helpful to keep track of the rover's operation:

```
    cv2.putText(map_add,"Mode: "+str(Rover.mode), (0, 160),
                cv2.FONT_HERSHEY_PLAIN, 0.8, (255, 255, 255), 1)
```

In the defition of the RoverState class I have updated some of the constants (I seemed to have better results with these numbers) and added a few new members. The changes are in yellow bellow.

```
# Define RoverState() class to retain rover state parameters
class RoverState():
    def __init__(self):
        self.start_time = None # To record the start time of navigation
        self.total_time = None # To record total duration of naviagation
        self.img = None # Current camera image
        self.pos = None # Current position (x, y)
        self.yaw = None # Current yaw angle
        self.pitch = None # Current pitch angle
        self.roll = None # Current roll angle
        self.vel = None # Current velocity
        self.steer = 0 # Current steering angle
        self.throttle = 0 # Current throttle value
        self.brake = 0 # Current brake value
        self.nav_angles = None # Angles of navigable terrain pixels
        self.nav_dists = None # Distances of navigable terrain pixels
        self.ground_truth = ground_truth_3d # Ground truth worldmap
        self.mode = 'start' # Current mode (can be forward or stop)
        self.throttle_set = 0.4     # Throttle setting when accelerating
                                    # Original 0.2; sotimes it cannot get unstuck
        self.brake_set = 5          # Brake setting when braking
                                    # Original 10; way to aggressive
        # The stop_forward and go_forward fields below represent total count
        # of navigable terrain pixels.  This is a very crude form of knowing
        # when you can keep going and when you should stop.  Feel free to
        # get creative in adding new fields or modifying these!
        self.stop_forward = 200   # Threshold to initiate stopping
                                 # original was 50; sometimes it gets stuck
        self.go_forward = 1500 # Threshold to go forward again; original 500
        self.max_vel = 1.0 # Maximum velocity (meters/second) original 2.0
        # Image output from perception step
        # Update this image to display your intermediate analysis steps
        # on screen in autonomous mode
        self.vision_image = np.zeros((160, 320, 3), dtype=np.float)
        # Worldmap
        # Update this image with the positions of navigable terrain
        # obstacles and rock samples
        self.worldmap = np.zeros((200, 200, 3), dtype=np.float)
        self.samples_pos = None # To store the actual sample positions
        self.samples_to_find = 0 # To store the initial count of samples
        self.samples_located = 0 # To store number of samples located on map
        self.samples_collected = 0 # To count the number of samples collected
        self.near_sample = 0 # Will be set to telemetry value data["near_sample"]
        self.picking_up = 0 # Will be set to telemetry value data["picking_up"]
        self.send_pickup = False # Set to True to trigger rock pickup

        self.samp_angles = None # Angles of navigable sample pixels
        self.samp_dists = None  # Distances of navigable sample pixels
        self.return_pos = None # we'll keep here the start position to return to
        self.save_yaw = None    # for when collecting rocks

        self.perc_mapped = 0  # to keep the percentage mapped so that we can stop
```

Example recording

A sample recording of the rover operation is available at:

https://youtu.be/cqfI_YCrk2A