# Feedback for 1804170

## *Computer Vision Assignment*

### **Presentation:** 25/25

The appearance of your code was generally fine. The introductory comments at the top of your program were good. The comments interspersed with your code made a good job of explaining how the code works and what individual sections of it are meant to do. In fact, this is the best-commented program I have seen (so far) for the assignment — well done!

### **Algorithms:** 22/25

You identify the various targets by colour, which is the most sensible way. You use OpenCV's `inRange` function in HSV space, a sensible choice, with some per-pixel processing to identify object limits. This approach is fine here but wouldn't work if there were to be more than one object of a particular colour. Your code for determining the distance to the meteors looks sensible.

You were able to track the motion of objects across the frames, the trickiest part of the assignment. You are using the lengths of normalized motion vectors computed using the inner product. That is close to the approach I used in my own solution, the angles between vectors computed by the inner product. My view is that this is the most general approach with the information we have been given by that lowly technician.

### **Coding style:** 22/25

You have nicely separated some of the code out into separate routines, which makes the program as a whole easier to read, though in a few places I think the functionality could have been made more straightforward. Also nice to see some `doctests` and `asserts` in your code, something I use quite often. Incidentally, you can use

```
im = cv2.imread ("...")
if im is None:
```

to detect `imread`'s failure to read an image; note the use of `is`.

### **Results:** 21/25

Your program works, correctly distinguishing the alien spacecraft from the meteors — though you have identified meteors as alien spacecraft too. Fortunately, there is more than one missile to fire and you have saved the human race from what seemed like certain destruction. The entire 10 billion people on Earth breathe a collective sigh of relief and you are awarded the Galactic Order of Merit for your achievements `;-)`. I read with some interest your remarks about the relative danger of false positives and false negatives.

Your submission works well with the test harness. Your program also runs with my unseen test sequence, though again it identifies most of the objects as UFOs.

Although not an assessment criterion, I was impressed by how quickly your program ran.

## **Overall mark: 90%**

Your program listing is on the following pages. Any line numbers mentioned above appear in its left-hand margin.

```
"""
---
**ce316ass.py**
---

---
**PURPOSE:**
---

Given:

* A number of frames to look at
    * Must be at least 1. Can't really look at less than 1 frame, y'know?
* Templates for filenames for left/right images
    * such as left-%03d.png and right-%03d.png
* Image files, which can be opened, with filenames that conform to the
  aforementioned templates
    * They must be of the same dimensions as each other.
        * If the dimensions differ, the program will terminate with an error
          message.
    * They must be openable.
        * If they cannot be opened, the program will terminate with an error
          message.

This will:

* Identify objects identified in the images
    * Assumptions
        * Assumes that the objects have these HSV values
            * Cyan
                * Hue 180
            * Red
                * Hue 0
                * Saturation of at least 1/256
            * White
                * Hue 0
                * Saturation 0
                * Value of at least 1/256 (and not in 'cyan')
            * Blue
                * Hue 240
                * Saturation of at least 1/256
                * Value of at least 1/256
            * Green
                * Hue 120
                * Saturation of at least 1/256
                * Value of at least 1/256
            * Yellow
                * Hue 60 (and not in 'orange')
                * Saturation of at least 1/256
            * Orange
                * Hue 40-58
        * Assumes there will be only one object of the given colour in the image
            * or 0 objects of that given colour :shrug:
        * Assumes that the objects are not overlapping
    * Limitations
        * If any object is at the boundary of the image, it will be rejected.
            * This is because, if it's at the boundary, the midpoint calculated
              is likely to be *very* inaccurate (because the object is likely to
              have been cut off somewhat), so, instead of dealing with that
              headache, it's just ignored.
        * If the x midpoint of an object in both images is identical, that
          object will be ignored because python doesn't like dividing by 0.
* For each of the objects identified in every frame
    * Print their distance (in terms of Z depth) from the cameras, in metres.
        * Along with their full X, Y, Z position from the cameras
    * Assuming
        * X baseline of 3500 metres
        * Y baseline of 0
        * Focal length of 12 metres
        * Pixel spacing of 10 microns
    * Limitations
        * If an object is not present in both images, it shall be ignored, due
          to not having full information about the x disparity for that frame.
        * If an object is at the edge of one of the images, it will also be
          ignored, due to the lack of accurate information about midpoints
* Work out the trajectories of each of the objects, printing a space-delimited
  list of the identifiers of the objects that are probably UFOs (not moving
  in a straight line)
    * Assuming
        * The object's position could be worked out for at least 3 frames.
            * If fewer than 3 positions are known, there won't be enough points
              in the line for the line to actually bend, so it'll treat it like
              an asteroid.
    * This is calculated via (ab)use of the dot products of unit vectors.
        * Gets a normalized version of the vector between the first position and
          the last position of the object in 3D space during these frames
        * Also gets a normalized version of the vector that describes the
          movement of the object between each 'frame'
            * If this normalized vector is (0, 0, 0), we omit it because it will
              completely mess up the maths.
        * Finds the dot product of the normalized current frame movement vector
```

```
          and the normalized total movement vector.
            * Puts it on a list with all the found dot products
        * Works out what 5% of the count of found dot products is
        * Two unit vectors are equal if their dot product = 1.0
            * Goes through that list of dot products, working out if that
              dot product isClose to 1.0 (using isClose (hey!), to 9dp)
                * We're using floats, and there's some inaccuracy with the
                  position measurement, so we know that we're not going to get
                  any dots that actually are going to be exactly 1.0
        * If at least ~5% of the dots are **not** isClose to 1.0
            * I am not ~95% sure that the line is straight, so, I'll accept the
              null hypothesis that the object in question is a UFO.


---

**USAGE**

---

* python3 ce316ass.py 50 left-%03d.png right-%03d.png
    * Assuming you have images called 'left-000.png' numbered to 'left-049.png',
      and 'right-000.png' numbered to 'left-049.png' in this directory,
      following all the assumptions/constraints in the 'PURPOSE' thing,
      this will run.

---

**AUTHOR**

--

Student 1804170

All the code written within this program is entirely my own work.

--

**RESULTS**

--

Given the sample data, this program produces an output of:
    UFO: cyan white blue yellow orange

I was expecting Cyan to be a UFO. However, I wasn't really expecting the others
to be UFOs.

This is using a 'straightLineMaxUncertainty' global variable, which is defined
just above the 'isThisAStraightLine' function. Basically, if that proportion of
unit vector versions of the movement of the object between the frames are not
close enough to the unit vector of the start position to end position movement
(worked out 'if the dot of current movement and total movement = 1', because yay
dot product abuse), we can't confidently say it's a straight line, therefore,
we'll assume it's an asteroid.

straightLineMaxUncertainty must be between 0.0 and 1.0 (inclusive). If there
are very few current movements, the minimum 'sus' threshold will be 1.

Here's some outputs of the program with different 'straightLineMaxUncertainty'
values (specifically, the lowest values where I noticed a change in the number
of UFOs that were output).

    * 0
        * UFO: cyan red white blue yellow orange
    * 0.05
        * UFO: cyan white blue yellow orange
    * 0.0625
        * UFO: cyan blue yellow orange
    * 0.075
        * UFO: cyan blue orange
    * 0.11
        * UFO: cyan blue
    * 0.125
        * UFO: cyan
    * 0.35
        * UFO:

Green was never detected as a UFO.

Then there was the problem of what threshold to use.

So we need to consider the context of the problem.

The problem is 'which of these things are aliens trying to attack earth and nick
our PDMS'. Which, to me, sounds like the sort of program where false negatives
are more dangerous than false positives.

Therefore, to minimize the chance of false positives, and also to satisfy the
self-declared stats person inside me, I am going to stick with the 95% 'is not
UFO' confidence-y threshold thing.
```

```

So basically, when you see 'UFO' on the printout, read '>5% not an asteroid'.
Because if I'm not 95% confident of it being an asteroid (by having more than 5%
of the dot products of normalized movements * overall normalized movement not be
close to 1), I'm going to accuse it of being a UFO.

Yes, I know, probably wasting shots with the thing that shoots the objects.

However, seeing as not all of the objects are listed as 'UFOs' when using this
threshold, I'm confident that I'm not getting *too* many false positives, so I'm
considering it to be good enough.


---
It runs pretty quickly though which is nice I guess.

6 seconds on the lab VMs!

"""


# and time for some stuff to be imported.

import sys
# we need the command line arguments, the ability to quit, and the error stream
from math import sqrt, isclose  # we need these for some of the maths.
from typing import Tuple, List, Dict, Union  # No excuse to not type annotate.

import cv2
# we're doing computer vision, so opencv is also pretty darn useful.
import numpy as np  # and it involves some numpy.ndarray objects!

"""
--DEBUGGING FUNCTIONS FOR SEEING HOW THINGS WORK--

You ever wanted to see what goes on under the hood with this program?
No?

Either way, here are some functions that can be used for debugging.

Some code later on does have inbuilt debugging functions.
"""

debugging: bool = False
"""
Set this to 'True' if you want to enable these debug functions.
Or just keep it as 'False' if you want to just run the thing.
"""


def debug(leftFilename: str, rightFilename: str,
          leftIm: np.ndarray, rightIm: np.ndarray, frameNum: int = 0) -> None:
    """
    The wrapper function for the debugging functions, called by the main program
    if 'debug' is set to True.

    :param leftFilename: filename of the left image
    :param rightFilename: filename of the right image
    :param leftIm: the left image, BGR format
    :param rightIm: the right image, BGR format
    :param frameNum: What frame this is (used for labelling the mask
     previews). If not defined, defaults to 0.
    :return: nothing.
    """

    # prints the filenames, to make sure they're correct
    print(leftFilename)
    print(rightFilename)

    # shows the left and right images
    cv2.imshow("left", leftIm)
    cv2.imshow("right", rightIm)
    handleShowingStuff()
    showMasksForDebugging(leftIm, rightIm, frameNum)
    # and then proceeds to show all the masks produced for each object in
    # the left and the right images.


# noinspection PyPep8Naming
def getObjectMasks(hsvIn: np.ndarray) -> Tuple[np.ndarray, np.ndarray,
                                               np.ndarray, np.ndarray,
                                               np.ndarray, np.ndarray,
                                               np.ndarray]:
    """
    Generates the object masks for a given HSV image.

    Yes, this uses some tuples that are defined as global variables in the
    'actually important code' section after this section of debugging code,
    because those tuples are part of the 'actually important code'.
    This has been done so, if this debugging code section is removed (along with
    the one branch in the main program that may potentially call this code),
```

```

    Either way, this function won't be run until after those declarations are
    reached, so no harm no foul or something along those lines.

    This code is explained better in the 'getObjectMidpoints' function,
    which has code that's basically identical to this.

    You're probably going to ask 'why have you duplicated the code?'

    Short answer: I don't want the production code to have a dependency on
    this debug code, and, if this debug code was to be omitted, I'd have a
    redundant dependency in the production code.

    :param hsvIn: the hsv image that contains the objects
    :return: masks for each of the 7 coloured objects that are in it.
    (cyan, red, white, blue, green, yellow, orange)
    """
    cyan_mask: np.ndarray = cv2.inRange(hsvIn, min_cyan, max_cyan)
    red_mask: np.ndarray = cv2.inRange(hsvIn, min_red, max_red)
    white_mask: np.ndarray = cv2.inRange(hsvIn, min_white, max_white)
    blue_mask: np.ndarray = cv2.inRange(hsvIn, min_blue, max_blue)
    green_mask: np.ndarray = cv2.inRange(hsvIn, min_green, max_green)
    yellow_mask: np.ndarray = cv2.inRange(hsvIn, min_yellow, max_yellow)
    orange_mask: np.ndarray = cv2.inRange(hsvIn, min_orange, max_orange)

    cyan_mask = cv2.morphologyEx(cyan_mask, cv2.MORPH_CLOSE, kernel33)

    white_mask = cv2.subtract(white_mask, cyan_mask)

    orange_mask = cv2.morphologyEx(orange_mask, cv2.MORPH_CLOSE, kernel33)

    yellow_mask = cv2.subtract(yellow_mask, orange_mask)

    return (cyan_mask, red_mask, white_mask, blue_mask,
            green_mask, yellow_mask, orange_mask)


# noinspection PyPep8Naming
def getStereoMasks(left_in: np.ndarray, right_in: np.ndarray) -> \
        List[Tuple[np.ndarray, np.ndarray]]:
    """
    Generates a list of masks for the left and right stereo images

    :param left_in: the left image.
    :param right_in: the right image.
    :return: a list of tuples containing the masks for each of the images.
     Tuples are in the form (left image mask, right image mask).
     The list itself is in the order
     cyan->red->white->blue->green->yellow->orange.
    """
    leftMasks: Tuple[np.ndarray, np.ndarray, np.ndarray, np.ndarray,
                     np.ndarray, np.ndarray, np.ndarray] =\
        getObjectMasks(cv2.cvtColor(left_in, cv2.COLOR_BGR2HSV))
    rightMasks: Tuple[np.ndarray, np.ndarray, np.ndarray, np.ndarray,
                      np.ndarray, np.ndarray, np.ndarray] =\
        getObjectMasks(cv2.cvtColor(right_in, cv2.COLOR_BGR2HSV))

    theMasks: List[Tuple[np.ndarray, np.ndarray]] = []
    for i in range(0, 7):
        theMasks.append((leftMasks[i], rightMasks[i]))

    return theMasks


# noinspection PyPep8Naming
def showMasksForDebugging(i_left: np.ndarray, i_right: np.ndarray, f: int) -> \
        None:
    """
    This is here mostly for debugging purposes, showing the masks of each image.
    The 'left' image will have some text on it with the colour name identifier
    of the object, as well as what number frame this is.
    Apologies in advance if the annotations overlap with the position of
    one of the objects in the set of images you are using to mark this.

    :param i_left: left image (BGR format)
    :param i_right: right image (BGR format)
    :param f: frame number
    :return: nothing.
    """
    # order in which the masks of objects are returned
    objectOrder: List[str] = \
        ["cyan", "red", "white", "blue", "green", "yellow", "orange"]

    i: int = 0
    debugMasks: List[Tuple[np.ndarray, np.ndarray]] =\
        getStereoMasks(i_left, i_right)  # the masks for each image.
    for m in debugMasks:
        showLeft: np.ndarray = np.ndarray.copy(m[0])  # copy of the left mask
        label: str = str(objectOrder[i]) + " " + str(f)
        print(label)
        # puts some text with object id and frame num on the left mask copy.
```

```python
        cv2.putText(showLeft, label, (50, 50),
                    cv2.FONT_HERSHEY_SIMPLEX, 1, 255, 2, cv2.LINE_AA)
        # and shows the left/right masks
        cv2.imshow("left", showLeft)
        cv2.imshow("right", m[1])
        handleShowingStuff()

        # finds midpoints of the object on the mask
        leftMid: Tuple[float, float] = getObjectMidpoint(m[0])
        rightMid: Tuple[float, float] = getObjectMidpoint(m[1])

        print(leftMid)
        print(rightMid)

        showLeft: np.ndarray = cv2.bitwise_and(i_left, i_left, mask=m[0])
        showRight: np.ndarray = cv2.bitwise_and(i_right, i_right, mask=m[1])
        # puts the annotation on the left copy again.
        cv2.putText(showLeft, label, (50, 50),
                    cv2.FONT_HERSHEY_SIMPLEX, 1, (255, 255, 255), 2,
                    cv2.LINE_AA)
        if leftMid != (-1, -1):
            # if the left midpoint is valid, it's put on the left copy as a
            # magenta dot at the int version of the midpoint.
            iLeftMid: Tuple[int, int] = (int(leftMid[0]), int(leftMid[1]))
            cv2.line(showLeft, iLeftMid, iLeftMid,  (255, 0, 255), 1)
        if rightMid != (-1, -1):
            # ditto for the right image.
            iRightMid: Tuple[int, int] = (int(rightMid[0]), int(rightMid[1]))
            cv2.line(showRight, iRightMid, iRightMid,  (255, 0, 255), 1)

        # and shows them.
        cv2.imshow("left", showLeft)
        cv2.imshow("right", showRight)
        handleShowingStuff()
        i += 1


# noinspection PyPep8Naming
def handleShowingStuff() -> None:
    """
    This is here to handle actually showing stuff when debugging the program.
    Prints something to console to let the user know that they need to press
    something to continue, handles doing the cv2.waitKey(0) call (thereby
    allowing any pending cv2.imshow()'d images to be shown), and,
    if q or escape are pressed, the program will close.

    :return: nothing is returned.
    """
    print("press something to continue (press q or escape to quit)")
    key: int = cv2.waitKey(0)
    # quit if escape (27) or q (113) are pressed
    if key == 27 or key == 113:
        cv2.destroyAllWindows()
        print("quitting!")
        sys.exit(0)


"""
-- THE ACTUALLY IMPORTANT CODE THAT ACTUALLY DOES STUFF --

Yep. Everything from here is actually of some use.
"""

"""
~~~~~~ A VECTOR3D CLASS (and also a function that uses it) ~~~~~~

This is used later on, represents points in 3D space
"""


class Vector3D:
    """
    A class to represent a vector in 3D space.
    This code was written by myself, but I fully acknowledge that somebody else
    has probably written a python implementation of a 3D vector before, so any
    relationship between this Vector3D and another implementation of Vector3D
    is entirely coincidental.

    This implementation just contains a normalize, subtract, isZero, dot product
    , and __str__ method (as well as a constructor ofc), because that's all
    the math stuff I needed for this particular use case.

    References for particular sources for math stuff (when used) have been given
    in the methods for each of the functions that use the math stuff.

    Now, you may ask yourself 'why is a class being used, when a tuple could
    do the same thing?'. Simple answer; encapsulation (so I have the methods all
    in the same place as each other). And also making sure I don't get confused
    between tuples of floats and 3D vectors. The mutability is also nice for the
    subtraction and the normalization stuff.
    """
```

```python
    def __init__(self, x: float, y: float, z: float):
        """
        Constructs a Vector3D with the given x, y, and z coordinates
        :param x: x coordinate
        :param y: y coordinate
        :param z: z coordinate
        """
        self.x: float = x
        self.y: float = y
        self.z: float = z

    def magnitude(self) -> float:
        """
        dist between 2 3D points:
        sqrt(((x2-x1)^2) + ((y2-y1)^2) + ((z2 - z1)^2))
        and we know that x1,y1,z1 = 0 already (because vector comes from origin
        0) and x2, y2, and z2 are the x, y, and z of this vector.

        Got the maths from https://www.calculator.net/distance-calculator.html

        :return: the magnitude of this vector
        """
        return sqrt((self.x ** 2) + (self.y ** 2) + (self.z ** 2))

    def normalized(self) -> "Vector3D":
        """
        Normalizes this Vector3D (makes the magnitude 1 by dividing all the
         components of this Vector3D by its magnitude)

        :return: This Vector3D, but with a magnitude of 1 instead.
         if this already had a magnitude of 0, it'll return itself as-is.
        """
        mag: float = self.magnitude()
        if mag > 0:
            self.x = self.x / mag
            self.y = self.y / mag
            self.z = self.z / mag

        return self

    def subtract(self, other: "Vector3D") -> "Vector3D":
        """
        Subtracts the other Vector3D from this Vector3D, returning this
        modified Vector3D.

        Didn't need to get the maths from anywhere because subtraction is pretty
        darn simple and doesn't have any weirdness.

        :param other: the other Vector3D to subtract from this.
        :return: this Vector3D minus 'other'. Would have type-annotated the
         return type as Vector3D, but python didn't like that.
        """
        self.x -= other.x
        self.y -= other.y
        self.z -= other.z
        return self

    def dot(self, other: "Vector3D") -> float:
        """
        Returns the dot product of this Vector3D and the other Vector3D.

        Got the maths from https://www.quantumstudy.com/physics/vectors-2/

        :param other: the other Vector3D this is being dot product-ed against.
        :return: the dot product of this and the other vector3D
        """
        return (self.x * other.x) + (self.y * other.y) + (self.z * other.z)

    # noinspection PyPep8Naming
    def isZero(self) -> bool:
        """
        Check if this vector is (0,0,0)
        :return: Returns true if x, y and z are exactly equal to 0
        """
        return (self.x == 0) and (self.y == 0) and (self.z == 0)

    def __str__(self) -> str:
        """
        Outputs this as a string; as a tuple in the form (x, y, z)
        :return: a string of the tuple (self.x, self.y, self.z)

        """
        return str((self.x, self.y, self.z))


# noinspection PyPep8Naming
def normalizeVectorBetweenPoints(fromVec: Vector3D, toVec: Vector3D) ->\
        Vector3D:
    """
    Get lhs-rhs but normalized instead (leaving lhs and rhs untouched)

```

```
    >>> normalizeVectorBetweenPoints(Vector3D(0,0,0),Vector3D(1,1,1))
    (0.5773502691896258, 0.5773502691896258, 0.5773502691896258)

    >>> normalizeVectorBetweenPoints(Vector3D(0,0,0),Vector3D(2,2,2))
    (0.5773502691896258, 0.5773502691896258, 0.5773502691896258)

    >>> normalizeVectorBetweenPoints(Vector3D(0,0,0),Vector3D(1,1.5,2))
    (0.3713906763541037, 0.5570860145311556, 0.7427813527082074)

    >>> normalizeVectorBetweenPoints(Vector3D(1,1,1),Vector3D(1,1,1))
    (0, 0, 0)

    :param fromVec: going from this vector
    :param toVec: to this other vector
    :return: toVec - fromVec but normalized. Or in other words, the direction of
     movement from the position 'fromVec' to the position 'toVec'
    """
    return Vector3D(toVec.x, toVec.y, toVec.z)\
        .subtract(fromVec)\
        .normalized()


"""
~~~~~ READING IMAGES, FINDING OBJECTS, AND ALSO CALCULATING AND
 PRINTING THE POSITIONS OF SAID OBJECTS ~~~~~

These functions (and also globals) are responsible for finding and printing the
positions of the objects in 3D space.
"""


# HSV values to be used to extract the objects from the image.
min_cyan: Tuple[int, int, int] = (90, 0, 0)
"""minimum HSV threshold for the cyan object"""
max_cyan: Tuple[int, int, int] = (90, 255, 255)
"""maximum HSV threshold for the cyan object"""

min_red: Tuple[int, int, int] = (0, 1, 0)
"""minimum HSV threshold for the red object"""
max_red: Tuple[int, int, int] = (0, 255, 255)
"""maximum HSV threshold for the red object"""

min_white: Tuple[int, int, int] = (0, 0, 1)
"""minimum HSV threshold for the white object"""
max_white: Tuple[int, int, int] = (0, 0, 255)
"""maximum HSV threshold for the white object"""

min_blue: Tuple[int, int, int] = (120, 1, 1)
"""minimum HSV threshold for the blue object"""
max_blue: Tuple[int, int, int] = (120, 255, 255)
"""maximum HSV threshold for the blue object"""

min_green: Tuple[int, int, int] = (60, 1, 1)
"""minimum HSV threshold for the green object"""
max_green: Tuple[int, int, int] = (60, 255, 255)
"""maximum HSV threshold for the green object"""

min_yellow: Tuple[int, int, int] = (30, 1, 0)
"""minimum HSV threshold for the yellow object"""
max_yellow: Tuple[int, int, int] = (30, 255, 255)
"""maximum HSV threshold for the yellow object"""

min_orange: Tuple[int, int, int] = (20, 0, 0)
"""minimum HSV threshold for the orange object"""
max_orange: Tuple[int, int, int] = (29, 255, 255)
"""maximum HSV threshold for the orange object"""

kernel33: np.ndarray = np.ones((3, 3), np.uint8)
"""
A 3*3 numpy array of 1s, to be used when closing up holes in some of the masks
"""


# noinspection PyPep8Naming
def getObjectMidpoint(objectMask: np.ndarray) -> Tuple[float, float]:
    """
    Returns the midpoint of the region of 1s in the given binary image array

    If there is no region of 1s, we return (-1,-1). If there's a 1 on the
    boundary of the image, once again, we return (-1,-1), because we'll know
    that the midpoint we find will probably be inaccurate.

    This estimates the midpoint by finding the upper/lower X and Y bounds of
    that region of 1s in the image. Yes, it's a pretty naiive, brute force-y
    method. However, I tried several object/blob detection algorithms within
    OpenCV, however, none of them really worked as intended (not detecting the
    objects in the earlier images, not really being able to work out which
    keypoints correspond to each object, and refusing to detect the single
    object when I go through the effort of masking out everything else in the
    image), so I went 'fuck it, guess I'm doing it myself'

```

```
    DISCLAIMER: I produced this code on the 3rd of March, several days before
     that email was sent out suggesting that a procedure like this would be
     worth using for the assignment.

    :param objectMask: binary image, with 1s in the area where the object
     with the midpoint being looked for is, and 0s everywhere else.
    :return: A tuple holding the the midpoint of the object.
     If the object isn't present (mask all 0s), a value of (-1,-1) is returned.
     Additionally, if the object is at the edge of the image (a minimum is 0,
     or a maximum is at the maximum possible x/y), that heavily implies that
     the object is partially out-of-frame. Therefore, as that means the true
     bounds are likely to be out-of-frame, this midpoint detector will not find
     the true midpoint of the object, so it will give up and return -1s for that
     as well.
    """

    if objectMask.any():
        if objectMask[0].any() or objectMask[-1].any():
            # if there's anything in the topmost or bottommost row, that means
            # there's something on the image boundary, meaning that the midpoint
            # found will be inaccurate, so we're not going to bother finding it.
            return -1, -1
    else:
        # if nothing in the objectMask is a 1, we return -1s.
        return -1, -1

    # obtaining the shape of the actual mask image
    yx: Tuple[int, int] = objectMask.shape
    ny: int = yx[0]
    nx: int = yx[1]

    # and declaring some variables to hold the info we find out about
    # the shape of the object.
    minX: int = -1
    maxX: int = -1
    minY: int = -1
    maxY: int = -1
    notFoundFirst: bool = True  # set this to false when we find first pixel

    # now we just casually loop through the image pixels,
    # and find out about what sort of shape the object has
    for y in range(1, ny-1):
        # we already established that the topmost/bottommost rows are empty.
        if objectMask[y].any():
            # we only bother with this row if it contains 1s
            maxY = y
            # y wont get smaller.
            # and assignment has same complexity as checking a single
            # condition so I may as well just reassign y anyway.
            for x in range(0, nx):
                if objectMask[y][x] != 0:
                    # if it's not 0, we've found something!
                    if notFoundFirst:
                        # if we haven't found the first thing yet, we have now.
                        notFoundFirst = False
                        minX = maxX = x
                        minY = maxY = y
                    else:
                        if x < minX:
                            minX = x
                        elif x > maxX:
                            maxX = x

    # if it's at the x bounds of the image, the result definitely won't be
    # accurate, so we'll just return -1, -1.
    # (we already checked the y bounds earlier on)
    if (minX == 0) or (maxX == nx-1):
        return -1, -1

    # working out widths and heights
    w: int = (maxX - minX)
    h: int = (maxY - minY)

    # using that and the lower bounds for x and y to find the midpoints
    xMid: float = minX + (w / 2.0)
    yMid: float = minY + (h / 2.0)

    # and returning a tuple with those midpoints
    return xMid, yMid


# noinspection PyPep8Naming
def getObjectMidpoints(hsvIn: np.ndarray) -> Dict[str, Tuple[float, float]]:
    """
    Gets the midpoints for the objects that may be in the given HSV image.
    We generate masks that contain only the region of the HSV object that is
    occupied by the pixels that make up a particular object, do a bit of cleanup
    for the objects that have somewhat overlapping pixel values, and then
    put those masks into getObjectMidpoint to produce a dictionary holding the
    midpoints of each of the objects in the image.

```

```
    If an object is not present in the image, its entry in the dictionary will
    have the default value of (-1,-1) instead of an actual midpoint.

    You might be wondering 'why am I making all the masks at once and then
    finding the midpoints from them all at once instead of just making a mask,
    getting the midpoint, and moving on to the next mask?'

    Simple answer: Doing it this way allows me to do the cleanup stuff that
    needs to be done for cyan/white/orange/yellow more effectively,
    and it means I can declare the results dictionary as a literal and also
    immediately return it. yay efficiency.

    :param hsvIn: the hsv image that contains the objects
    :return: dict with midpoints for each of the 7 coloured objects
     that might be present in the given image. If not present, midpoint will
     be (-1,-1). Keys are (cyan, red, white, blue, green, yellow, orange)
    """
    # generating masks for each object.
    cyan_mask: np.ndarray = cv2.inRange(hsvIn, min_cyan, max_cyan)
    red_mask: np.ndarray = cv2.inRange(hsvIn, min_red, max_red)
    white_mask: np.ndarray = cv2.inRange(hsvIn, min_white, max_white)
    blue_mask: np.ndarray = cv2.inRange(hsvIn, min_blue, max_blue)
    green_mask: np.ndarray = cv2.inRange(hsvIn, min_green, max_green)
    yellow_mask: np.ndarray = cv2.inRange(hsvIn, min_yellow, max_yellow)
    orange_mask: np.ndarray = cv2.inRange(hsvIn, min_orange, max_orange)

    # and now, time for some cleanup

    # filling in the midpoint for the cyan mask (as that's actually white)
    cyan_mask = cv2.morphologyEx(cyan_mask, cv2.MORPH_CLOSE, kernel33)

    # removing the midpoint in the cyan mask from the white mask
    white_mask = cv2.subtract(white_mask, cyan_mask)

    # filling in the midpoint for the orange mask (as that's actually yellow)
    orange_mask = cv2.morphologyEx(orange_mask, cv2.MORPH_CLOSE, kernel33)

    # and removing the orange midpoint from some yellow
    yellow_mask = cv2.subtract(yellow_mask, orange_mask)

    # finally making + returning the dict with the midpoints of each object
    return {
        "cyan": getObjectMidpoint(cyan_mask),
        "red": getObjectMidpoint(red_mask),
        "white": getObjectMidpoint(white_mask),
        "blue": getObjectMidpoint(blue_mask),
        "green": getObjectMidpoint(green_mask),
        "yellow": getObjectMidpoint(yellow_mask),
        "orange": getObjectMidpoint(orange_mask)
    }


# noinspection PyPep8Naming
def getStereoPositions(left_in: np.ndarray, right_in: np.ndarray) -> \
        Dict[str,
             Tuple[Tuple[float, float],
                   Tuple[float, float]]]:
    """
    Gets the positions of objects in the left and right *coloured* images.
    This **will** assume that the dimensions of the left and right images
    are identical, and that the two images are Y-aligned already.

    :param left_in: the left input image *in colour*
    :param right_in: the right input image *in colour*
    :return: a dictionary with the names of the objects in both images as keys,
     and a tuple, containing the  x' and y' positions of that particular object
     in both images as the value

        1st tuple in the value tuple: (x',y') from left image.
        2nd tuple in the value tuple: (x',y') from right image.

        If an object is not present in **both** images, it will **not** be
        present in the returned dictionary.
    :raises: ValueError if the two images provided have different dimensions.
    """
    # gets the shape of the images
    yx: Tuple[int, int] = left_in.shape
    if yx != right_in.shape:
        # complains if the dimensions aren't identical.
        raise ValueError("Please provide images with identical dimensions.")

    lDict: Dict[str, Tuple[float, float]] = getObjectMidpoints(left_in)
    """dictionary with midpoints for every object in the left image"""

    rDict: Dict[str, Tuple[float, float]] = getObjectMidpoints(right_in)
    """dictionary with midpoints for every object in the right image"""

    # half of the x and y dimensions of the images
    halfY: float = yx[0] / 2
    halfX: float = yx[1] / 2

```

```
    posDict: Dict[str, Tuple[Tuple[float, float], Tuple[float, float]]] = {}
    """a dictionary for all the calculated (X’,Y’) positions for each image"""

    # obtains the keys from lDict but as a list so it can be foreach’d,
    # and also foreaches through them
    for key in [*lDict.keys()]:
        if lDict[key] != (-1, -1):
            if rDict[key] != (-1, -1):
                # if both dictionaries have an actual value for the key

                # just getting a copy of those raw values real quick
                rawL: Tuple[float, float] = lDict[key]
                rawR: Tuple[float, float] = rDict[key]

                # work out the X’ and the Y’ stuff for left and right
                # and put it into the posDict.
                #   X‘ = x - halfX
                #   Y’ = halfY - y
                posDict[key] = (
                    (rawL[0] - halfX, halfY - rawL[1]),
                    (rawR[0] - halfX, halfY - rawR[1])
                )

    # and now return the posDict
    return posDict


placeholderOutString: str = "{:5}  {:8}  {:8.2e}  {}"
"""
This is a placeholder string to be used when formatting the frame-by-frame
printout of object data. Double space between each thing of data.
1st value: will be the frame number. 5 width, right-aligned.
2nd value: object identifier. 8 width, also left-aligned
3rd value: object distance (Z pos, in metres). 8 width, in the form 1.23e+45

4th value: just the raw (X,Y,Z) position of the object in 3D space (in metres),
for sake of curiosity.
"""
focalLength: float = 12
"Focal length of camera is 12m"
baseline: float = 3500
"baseline between cameras is 3.5km -> 3500m"
pixelSize: float = float(1e-5)
"pixel spacing: 10 microns -> 1e-5 metres"


# noinspection PyPep8Naming
def calculateAndPrintPositionsOfObjects(leftIm: np.ndarray,
                                        rightIm: np.ndarray,
                                        frameNum: int = 0) -> \
        Dict[str, Vector3D]:
    """
    Given a left image (BGR), a right image (BGR), and a frame number (optional)
    , this method will print the details about the identifiers and the depths
    (Z axis positions) of the objects in the image (formatted as per the
    assignment brief, using the global placeholderOutString), and will return a
    dictionary with vector3 positions of the objects in the images.

    Objects in only one image will be omitted. Distances will be in metres.

    leftIm and rightIm must be in BGR colour, and have identical dimensions.

    This will use the focalLength, baseline, and pixelSize global variables to
    calculate the positions of the objects.

    placeholderOutString, focalLength, baseline, and pixelSize are present
    just above this function.

    :param leftIm: The left stereo image to look at (BGR colour)
    :param rightIm: The right stereo image to look at (BGR colour)
    :param frameNum: The frame number (optional). Will only be used to prefix
     the printout. If not supplied, 0 will be used.
    :return: A dictionary with the identifiers of the objects identified, along
     with their vector3 positions, in metres, relative to the midpoint between
     the cameras.
    """

    imgPositions: \
        Dict[str, Tuple[Tuple[float, float], Tuple[float, float]]] = \
        getStereoPositions(cv2.cvtColor(leftIm, cv2.COLOR_BGR2HSV),
                           cv2.cvtColor(rightIm, cv2.COLOR_BGR2HSV))
    """
    These are the left and right image X’Y’ coords of every object
    in both the left and the right images.
    """

    # noinspection PyShadowingNames
    posXYZ: Dict[str, Vector3D] = {}
    """
    this will hold the X, Y, Z coords of the objects in 3D space,
    using the midpoint between the cameras as the origin,
```

```
    measured in metres.
    """

    # unpacking the keys/object names as a list so we can iterate through them.
    # Why do I need to do this? Because Dict.keys() returns a KeysView object,
    # which isn't iterable, and is generally awkward to work with. However,
    # putting a KeysView kv into [*kv] basically unpacks it into a list, which
    # we can iterate through. So that's what happens here.
    for key in [*imgPositions.keys()]:

        currentPos: Tuple[Tuple[float, float], Tuple[float, float]] = \
            imgPositions[key]
        "We obtain info about current object's 2d pos from imagePositions"

        xDisparity: float = currentPos[0][0] - currentPos[1][0]
        "x disparity = xL - xR"

        if xDisparity == 0.0:
            # undefined behaviour (aka runtime error) if the x disparity is 0.
            # so we'll just forget it ever happened
            continue

        rawZ: float = (focalLength * baseline) / (xDisparity * pixelSize)
        """
        Z = (f * b) / (xl - xr)
        which is how we work out what dist is.
        """

        rawX: float = ((-currentPos[1][0] * pixelSize) / focalLength) * rawZ
        """
        (-xr/f) = (X/Z), therefore (-xr/f) * Z = X
        So I used that equation to find out what the actual X position
        of the object is in 3D space.
        """

        yMid: float = (currentPos[0][1] + currentPos[1][1]) / 2
        """
        This is the y midpoint of the object. I'm getting the average of the y
        position for the two images, just in case they differ a bit (and, if
        they're actually identical, the yMid will just be the same as them)
        """

        rawY: float = -((yMid * pixelSize) / focalLength) * rawZ
        """
        (xl / f) = (B-X)/Z
        so, substituting the x for y
        (y / f) = (B-Y)/Z = -Y/Z
        and if we rearrange that so -Y is the result
        (y/f) * Z = -Y
        and negating that to get positive Y as the result
        -(y/f) * Z = Y
        """

        # we put the raw XYZ into posXYZ
        posXYZ[key] = Vector3D(rawX, rawY, rawZ)

        # Now, we just print the required info, as per the specification.
        print(placeholderOutString.format(
            frameNum,  # what frame number this is
            key,  # identifier of this object
            rawZ,  # we print the Z depth
            posXYZ[key]  # the XYZ pos, printed for debug reasons.
        ))

    # we finish by returning the posXYZ dictionary.
    return posXYZ


"""
~~~~~~ WORKING OUT WHAT IS/IS NOT A UFO ~~~~~~

These methods (and globals) are used to work out what is/isn't a UFO from a
dictionary of UFO identifiers and their frame-by-frame positions as lists of
Vector3D objects.
"""

debuggingLineStuff: bool = False
"""
Set this to true if you want to enable the debug printouts for the
isThisAStraightLine function (immediately below this)
"""

straightLineMaxUncertainty: float = 0.05
"""
How uncertain we are allowing ourselves to be about whether a line is straight
or not. If there is more than this amount of uncertainty (0.05 = 5%), we won't
consider it to be a straight line; instead, we'll consider it to be a UFO.

MUST BE BETWEEN 0 AND 1.0!

Outputs at different thresholds of this value:
```

```
    * 0
        * UFO: cyan red white blue yellow orange
    * 0.05
        * UFO: cyan white blue yellow orange
    * 0.0625
        * UFO: cyan blue yellow orange
    * 0.075
        * UFO: cyan blue orange
    * 0.11
        * UFO: cyan blue
    * 0.125
        * UFO: cyan
    * 0.35
        * UFO:

"""

assert (0.0 <= straightLineMaxUncertainty <= 1.0)


# noinspection PyPep8Naming
def isThisAStraightLine(line: List[Vector3D]) -> bool:
    """
    Returns whether or not a sequence of 3D points is a straight line,
    using the getNormDifferenceBetweenPoints function, and dot product abuse.

    If there's 2 or fewer points, it certainly ain't bent, so it will return
    true.

    Due to the inherent uncertainty with how the points are calculated, I am
    giving some leeway in the calculations.

    And basically I'm working out if it's straight or not by seeing if at least
    ~95% of the dot products for the normalized vectors between each vector of
    the line and the normalized vector between the start and the end of the line
    are ~1.0 (tl;dr the dot product of two identical unit vectors is 1, but, if
    they aren't identical, it'll be less than 1).

    :param line: the sequence of 3D points
    :return: true if they're a straight enough line, false otherwise.
    """

    if len(line) < 3:
        # 3 short 5 bend
        return True

    startEndDiff: Vector3D = \
        normalizeVectorBetweenPoints(line[0], line[-1])
    """
    This is a normalized vector between the starting point and the ending
    point of the object. Every single vector between each pair of consecutive
    points will be checked for similarity to this via their dot products
    """

    if debuggingLineStuff:
        print(startEndDiff)

    dots: List[float] = []
    """
    A list to hold the dot product(s) of startEndDiff and the normalized
    versions of the vectors between each pair of consecutive vectors in line
    """

    thisIndex: int = 0
    """
    A cursor to the index of the line used for this iteration. This starts at 0,
    so, when the first iteration increments it to 1, the first iteration will
    look at indexes [0] and [1] (getting the first movement vector).
    """

    while True:
        thisIndex += 1  # we move to the next index of the list
        if thisIndex >= len(line):
            # we're basically emulating a do/while loop here
            # with a while condition of thisIndex < len(line)
            # so when we get to the end of the list, we stop looping.
            break

        thisDiff: Vector3D = \
            normalizeVectorBetweenPoints(line[thisIndex-1], line[thisIndex])
        """
        We find the vector between the position at the index  thisIndex and the
        point on the line behind it, but normalized instead, so we can compare
        it to the normalized startEndDiff.
        """

        if debuggingLineStuff:
            print(thisDiff)

        if thisDiff.isZero():
            # if it's a 0 vector, that will mess up our calculations, so
```

```
            # we'll just ignore it and move on to the next pair of vectors.
            continue

        thisDot: float = startEndDiff.dot(thisDiff)
        """
        TIME FOR SOME ILLEGAL MATHS!!!

        Funnily enough, you can actually use the dot product of two unit vectors
        to compare the unit vectors for similarity.

        Unit vectors have a magnitude of 1. And the dot product of two vectors
        is basically working out how far a vector projects onto another. Forgot
        the technical terms.

        But, the important thing is that if you have two unit vectors, and the
        two unit vectors are identical (same x, y, z; same direction), the dot
        product of those two vectors will be 1.

        For a more practical example of this illegal maths in action,
        there's a rather nice demo of the dot products of vectors (but in two
        dimensions) here, where you can try messing around with unit vectors:
    https://www.youphysics.education/scalar-and-vector-quantities/dot-product/
        """

        if debuggingLineStuff:
            print(thisDot)

        dots.append(thisDot)  # and we append the current dot product to dots.

    if len(dots) == 0:
        # if all the differences between positions were (0,0,0), this ain't bent
        # so it'll return True.
        return True

    maxSusFloat: float = len(dots) * straightLineMaxUncertainty

    maxSus: int = int(maxSusFloat)
    """
    This is how many of the dot products have to be not roughly equal to 1 for
    the object to be labelled as a UFO. It's currently set up so, if ~5% of the
    dots are not equal to 1, that's sus enough for us to label it as a UFO, with
    95% certainty of this being the case.

    This is because I'm working on the hypothesis that 'This line is straight',
    and I'm going only going to accept this hypothesis with a certainty of at
    least 95% (it's good enough for geography, and there's not enough data, at
    least in the sample dataset, for me to really be able to test for 99%
    certainty)

    So, if ~5% of the vectors indicate that this is not travelling in a straight
    line, we have a certainty of less than 95% that this is travelling in a
    straight line, therefore, we will reject the hypothesis that 'this object
    is travelling in a straight line', and accept the null hypothesis (of 'this
    object is not travelling in a straight line') instead.
    """

    # and if there's a maximum sus level that's below 0, we set it to 1.
    if maxSus < 1:
        maxSus = 1

    if debuggingLineStuff:
        print("ufo is " + str(maxSus) + " sus!")  # WHEN THE UFO IS SUS!

    susCount: int = 0
    "The count of how many times the dot product has been not equal to 1."

    debugCount: int = 0
    "just here as a printout for debug purposes."

    for d in dots:
        if not isclose(1, d, rel_tol=1e-09):
            """
            We're using floating-point numbers here, so, because it's nigh
            impossible to get a dot of 1.0 (mostly due to the slight inaccuracy
            inherent due to resolution and pixel values and stuff like that),
            we're using the 'isclose' method to check if the dots are within
            1e-09 of 1 (at least 0.999999999).

            If it isn't close enough to 1, the thing isn't going in
            a straight line. So it's sus.

            And if it's sus maxSus times, this is clearly not a straight line.
            """
            susCount += 1
            if debuggingLineStuff:
                print("diff " + str(debugCount) + " is " +
                      str(susCount) + " sus")
            if susCount >= maxSus:  # WHEN THE UFO IS SUS
                return False  # amogus
        debugCount += 1
```

```
    # if it hasn't been thrown out as sus yet, it's probably a straight line.
    return True


# noinspection PyPep8Naming
def makeUfoString(objPositions: Dict[str, List[Vector3D]]) -> str:
    """
    Given the dictionary of object identifiers + lists with all of
    their Vector3D positions, create the space delimited string with the
    list of all the identifiers of objects that are UFOs
    :param objPositions: dictionary of object identifiers + Vector3D
     positions for all the objects that may or may not be UFOs
    :return: string with the identifiers of what is and isn't a UFO
    """
    ufoString: str = ""
    """
    The space-delimited string of UFO identifiers.
    """

    for key in [*objPositions.keys()]:
        if not isThisAStraightLine(objPositions[key]):
            # we check if the list of points is a straight line.
            # If they aren't a straight line, we know this is a UFO, so
            # it's appended to the ufoString.
            ufoString = ufoString + " " + key

    return ufoString


"""
~~~~~~ CHECKING WHETHER OR NOT AN IMAGE OPENED ~~~~~

yeah this is just here to stop a big error from happening
"""


# noinspection PyPep8Naming
def checkIfImageWasOpened(filename: str, img: Union[np.ndarray, None]) -> None:
    """
    This will check if the image with the given filename could be opened

    :param filename: the name of the file
    :param img: the numpy.ndarray (or lack thereof) that opencv could open
     using that given filename
    :return: nothing. But, if the file couldn't be opened (causing img to be
     None instead of a np.ndarray), the program complains and promptly closes.
    """
    if isinstance(img, type(None)):
        """
        If the image is actually NoneType, the program complains and closes.

        And you may ask yourself 'why am I doing this in such a convoluted way?'
        Simple answer: There is no simpler way to do it.
        Opencv doesn't throw an exception if the image couldn't be read, it just
        returns None.
        So, if it does return None, I could just detect it with 'if im == None',
        right? WRONG!
        Thing is, if it doesn't return None, it returns a numpy.ndarray. And if
        you attempt to compare one of those against None, guess what? You get
        a ValueError and a snarky comment saying 'oh no the truth value of this
        is ambiguous pls use .any() or .all() instead'
        But if I try to use those methods to check if the image exists, and the
        image doesn't exist, guess what? You get an AttributeError.

        Now, do I want to bother with throwing and catching exceptions manually?
        No. cba to deal with that overhead.

        Would I have preferred it if OpenCV could have just thrown an exception
        or just returned an empty array instead of returning a mcfucking None?

        Yes.

        But, alas, we live in a society. Rant over.
        """
        print("ERROR: Could not open the file called " + filename)
        sys.exit(1)


"""
~~~~~~ THE MAIN PROGRAM ~~~~~~

Everything from here is the stuff that runs when you start running this.
"""


if len(sys.argv) < 4:
    # If you don't give 3 command line arguments, the program will complain
    print("Usage:", sys.argv[0],
          "<frame count> ",
          "<left frame filename template, such as left-%03d.png> ",
          "<right frame filename template, such as right-%03d.png>",
```

```
            file=sys.stderr)
    # and promptly quit
    sys.exit(1)


# this line was adapted from the assignment brief.
nframes: int = int(sys.argv[1])
"""
Reads the 1st (well, technically 2nd) command line argument as the number of
frames to look at.
"""

if nframes < 1:
    # complains if it's asked to look at less than 1 frame (and gives up)
    print("How the hell am I supposed to look at less than 1 frame!?")
    sys.exit(1)


objectPositions: Dict[str, List[Vector3D]] = {
    "cyan": [],
    "red": [],
    "white": [],
    "blue": [],
    "green": [],
    "yellow": [],
    "orange": []
}
"""
This is a dictionary which will hold the positions of the objects for every
frame.
"""

print("frame  identity  distance")  # header for the required frame data info.

# the following 10 lines were adapted from the assignment brief.
for frame in range(0, nframes):
    # we work out the filenames for the left and right images for this frame,
    # and then we open those images using opencv.
    # (and also check to see if the images could actually be opened.)
    fn_left: str = sys.argv[2] % frame
    im_left: np.ndarray = cv2.imread(fn_left)  # left image (BGR)
    checkIfImageWasOpened(fn_left, im_left)

    fn_right: str = sys.argv[3] % frame
    im_right: np.ndarray = cv2.imread(fn_right) # Right image (BGR)
    checkIfImageWasOpened(fn_right, im_right)

    if debugging:
        """
        You remember those debugging functions from earlier, right?
        Well, this is where they get used. If you enabled 'debugging' ofc.
        """
        debug(fn_left, fn_right, im_left, im_right, frame)
        # END OF DEBUGGING CODE

    posXYZ: Dict[str, Vector3D] = \
        calculateAndPrintPositionsOfObjects(im_left, im_right, frame)
    """
    We obtain the identifiers and XYZ positions of all the objects that are
    present within both of the stereo frames.
    """

    for o in [*posXYZ.keys()]:
        objectPositions[o].append(posXYZ[o])
        # and we append them to the list of all positions for that object.

# Finally, we print out what is/isn't a UFO.
print("UFO:{}".format(makeUfoString(objectPositions)))

"""
That's all, folks!
"""
```