CE316: Computer vision
Assignment

Adrian F. Clark
alien@essex.ac.uk

Contents

1 The task

The year is 2500 and the Earth is under attack from aliens, intent on stripping the planet of its natural resource of polydimethylsiloxane. You have just experienced the worst solar flare in living memory, which has wiped out the Earth's automatic defence system... and that means that the buck stops with you as Chief Scientist of the Earth Defence Corps.

A lowly technician, one Adrian Clark, has just sent you a message. He says the long-range cameras on Nyx and Styx, two of the moons of Pluto, have detected a meteor shower coming from the Oort Cloud and he is concerned that an alien vessel is hiding amongst the meteors. The aliens have tried this kind of thing before — it is usually easy to spot alien craft as they have to manoeuvre to avoid the meteors, whereas the meteors themselves travel in straight lines. Surely it cannot be too difficult to knock together a program to identify any meteor exhibiting non-linear motion in 3D?

The trouble is that the solar flare knocked out all modern computers. Looking around, you realise with horror that the only computer that has survived the flare unscathed is one of your antiques, a 2020s-vintage PC running Linux. Booting it and logging in, you see that you have support for only Python, so you will have to program a solution using that. You also have copies of numpy and the OpenCV library from the same period, which you know will be able to handle the images from the long-range scanner.

You sigh deeply and reply to Adrian Clark: "Send me the imagery and I'll try to write some software to report anything that manoeuvres in the meteor shower."

"OK," he messages back. "We have only low-resolution imagery from two cameras; thankfully they are identical. The telescopes to which the cameras are attached have focal lengths of 12 m and are arranged 3.5 km apart with their optical axes exactly aligned. The cameras have a 10-micron pixel spacing. I have 50 frames captured simultaneously from the left and the right cameras, which I'll put into a zip-file and send to you. I reckon you have until just before noon on Thursday 18$\frac{\text{th}}{}$ March before the meteors reach Earth. You have to make your program work by then." Mentally, you roll your sleeves up before turning to the keyboard and starting up Emacs. If the situation wasn't so serious, this would be fun...

2 Operation and evaluation

From the above, you need to identify each visible object in each frame. From the positions of objects in the left and right images, you will be able to calculate their distance from the cameras. You then need to record the 3D position of each object and try to establish which are, and which are not, travelling in straight lines.

The functionality of your program will be checked using a *test harness*, a piece of software that runs it and analyses its output. You will lose marks if your submission doesn't work with the test harness. To integrate with it, your program must accept precisely three arguments from the command line:

- the number of frames to be processed;
- a template (see below) for generating the filename of a left-hand frame;
- a template for generating the filename of a right-hand frame.

A typical invocation to process the test data files from your current directory would be:

```
python3 ce316ass.py 50 left-%3.3d.png right-%3.3d.png
```

The following lines of Python use these templates to convert frame numbers into filenames in the right way:

```
nframes = int (sys.argv[1])
for frame in range (0, nframes):
  fn_left  = sys.argv[2] % frame
  fn_right = sys.argv[3] % frame
```

The output from your submitted program should be in exactly the format shown below:

```
frame  identity  distance
...
   12  red       1.71e+12
   12  yellow    1.62e+12
...
UFO: yellow blue
```

After the heading, the first thing output on each line is the frame number, the second is an identifier for an object (you are welcome to use any whitespace-free text string for this) and the third the distance you have calculated. Any further values on the line are ignored, so you can use them for debugging information such as the locations of objects. The very last line of output contains the text "UFO: " and is followed by a space-separated list of the identities of any objects whose motion suggests they are UFOs. There should be no other output.

Note that, in accordance with good testing in computer vision, the imagery I shall test your program with is different from that you are using to develop it. It does however have identical characteristics, so if your program works on the supplied imagery it should also work on my unseen test imagery.

3 Submission

Remember to identify your work with your registration number and not your name. FASER allows you to upload your work as often as you like, so do keep uploading your program as you develop it.

submission deadline	Thursday 18th March at 11:59:59 (just before midday)
what to submit	only your program (**not** the images!)
submission format	Python code or a zip archive
marks returned	start of the summer term
marking criteria	see Section 5

Please submit your code as Python (if everything is in one file) or as a zip-file. Please do not use other archive formats.

4 Incorporating others' work

Almost everyone looks for inspiration when writing software: a web search is very quick and easy to do and can often help with something that has you stumped. This is absolutely fine and you are encouraged to do it — but do be careful how you use the information you find to avoid being accused of cheating. This is perhaps best illustrated by an example.

Let's say you cannot remember how to work out the roots of the quadratic $ax^2 + bx + c = 0$ and you do a web search for:

```
python roots of a quadratic equation
```

One of the top links gives you the two lines of Python you need. If you incorporate this directly into your code, *you must acknowledge the source* as shown below:

```
# The following two lines are from
# https://www.w3resource.com/python-exercises/math/python-math-exercise-30.php
x1 = (((-b) + sqrt(r))/(2*a))
x2 = (((-b) - sqrt(r))/(2*a))
```

Studying the code you might see that, at least in Python 2, this could give the wrong answer if `a`, `b` and `c` were all integers; and there are far too many parentheses. If you take that code but correct it and change it to fit into your algorithm, you should write in your program:

```
# Adapted fromm
# https://www.w3resource.com/python-exercises/math/python-math-exercise-30.php
x1 = (-b + math.sqrt(r)) / 2.0 / a
x2 = (-b + math.sqrt(r)) / 2.0 / a
```

You don't lose any credit for doing so.

Another way to solve problems is to talk about them with your friends. Feel free to do this while talking about algorithms and approaches…but stop when you reach the point of talking about code. To continue the above example, you might say to your friend that you need to intersect a line with a circle in your program. Your friend may then explain how the problem turns out to be a quadratic equation, a term you remember vaguely only from school. This is fine…but if she gives you code to put into your program, you are both guilty of cheating. (Incidentally, this equation is a remarkably poor numerical solution to a quadratic equation.)

Be warned that I am pretty rigorous in checking for both kinds of cheating, using a combination of tools such as `turnitin` (plus some of my own) *and* identifying chunks of code in submissions that look similar as I work through them. In general, you lose no marks by incorporating modest amounts of external code (when done correctly), so cheating is a really, really stupid thing to do — and especially in the last year of study.

5 Marking criteria

I shall assess your submissions to the assignment under four broad headings, *algorithms*, *style*, *result* and *presentation*. Each of these is marked out of 25, so they add up to give an overall percentage mark. What I look for under each of these headings is explained below. After that, the feedback from a pair of programs achieving substantially different marks is presented to help you gauge how to prepare your own submission.

Algorithms. Underlying every program is an algorithm or, more commonly, several algorithms. An algorithm is a series of steps that transform some input to an output. The questions I ask when marking submissions are:

- When there are several alternative algorithms, has the most suitable one been chosen? Are there comments that justify the choice?
- When an algorithm has had to be developed from scratch, does it do the job it is intended to do? Has it been implemented correctly?
- A program that runs in one second is better than an equivalent program that takes ten seconds to run, and that is rewarded under this heading. Conversely, attempting to squeeze every nanosecond of performance from a single line of code that is executed only once and takes little time to run is pointless and will be penalised. It is important to know *when* to improve performance as well as *how* and *where* to improve it.

Coding style. Although an algorithm is the core of a piece of code, the way in which it is implemented is important. When reviewing submissions, I ask the following questions:

- Is your code easy for someone to read? You might know that, say, a particular numpy (numerical Python) expression can be made to execute very quickly; but if it is hard to understand and not in the most time-critical part of a program, it is probably better written to be easier to understand, or at least well-explained in a comment.
- Does the way the program is structured help its maintenance? In the Real World[1], maintaining and extending existing code is much more common than writing code from scratch, so a good programmer will make sure there is enough explanation for someone coming to the program can maintain it.

[1] This is where people work for a living and time costs money.

- Does your program structure aid re-use of parts of the code? This could be by creating useful classes, modules or procedures that are general enough to be used elsewhere, for example. Similarly, do you use standard procedures rather than writing your own?
- Are you careful in your use of resources, especially imaged and large arrays? Programs that are wasteful in terms of memory usage are difficult to port to small computers or embedded platforms.
- Code that carefully arranges data to be organized so that a series of processing steps can be performed quickly and easily gains more credit than brute-force processing. It is not always straightforward to make code elegant in this way but it is easy to recognize it when you see it.

Clearly, there can be trade-offs between computing things several times and storing the results of calculations for subsequent use, and a good programmer will discuss this topic briefly in the code.

Results. This section catches all the other aspects of the program — principally whether it actually is a bug-free solution to the assignment, but also other factors such as whether or not the code provides useful output when it encounters problems. When marking submissions, I look for answers to the following questions:

- Is the program invoked as per the specification? Failure of a program to meet its specification is a problem in the Real World as it will often end up as a component of a larger system. By the same token, writing a program that does what you want rather than what was specified will yield a substantially lower mark than one that does meet the specification.
- Is the program buggy? All bugs what you can find should be exterminated before submitting your program.
- Does the program describe what should be produced from a specific input? How would a person receiving your program know that it works? The usual way is to include example inputs and the corresponding outputs in a block of comments at the top of the program. Python provides really cute doctest functionality, for example.
- Does the program produce helpful output if invoked incorrectly? If a program requires an argument on the command line, it should not crash but should instead output (on the *standard error* stream) a message that explains how the program should be used. Similarly, if the program reads files with specific names as part of its operation, you need to produce a useful message if they are not present.
- Are the results as expected? Does the program produce results that are consistent with what you expect? If not, *say so* in your program's comments as I can award you some credit for having spotted that your program is not giving the correct answers.

You might be interested to learn that some of my former students who work in the software development industry have to write a series of tests, with expected outputs, *before* writing the actual code.

Presentation. In the Real World, the software industry you may well encounter after you graduate, the presentation of code is more important than you might think. Many software companies establish guidelines for the presentation of source code, often called a *house style*, that developers and maintainers are expected to use. Maintenance tends to be a more expensive activity than development, so it is important that people reading someone else's code are able to understand quickly what it is doing and how. When marking submissions, questions I ask myself are:

- Is the code well commented? This means that the comments explain what the program is trying to do rather than interpreting individual lines of code in English. Well-commented programs usually have blocks of comments interspersed between sections of code, not a comment for each individual line.
- Is there a block of comments at the top of the program that describes its purpose and shows how it is to be used?

- Does the indentation of the program indicate its structure? Note that the indentation seen in an editor or IDE does not necessarily match what will be seen outside that editor.
- Does the program source code conform to the house style? For this assignment, you need to ensure that *no line of code is longer than 80 characters*. The underlying reason for this is that humans find lines containing more than 60–80 characters difficult to read. If your statement is longer than 80 characters in length, use the feature within the programming language for splitting long lines; there will be one.
- Are there blank lines between sections of code to aid the reader's understanding?
- Is there white space within lines to make them easier to read?

A good way to check that the code is OK is to open a conventional terminal window on your friendly Linux box or Mac and type the incantation

```
enscript -E --fancy-header=emacs -j -MA4 mysubmission.py -o t.ps
```

where `mysubmission.py` is the name of the program you want to submit. Then look at `t.ps`: if lines are truncated or wrapped, or the indentation is wrong, you need to fix it.

Examples To give you an idea of how programs are actually marked, here are two fictitious examples. The first is a bare pass: it works and produces correct results but omits all the niceties of good programming. The second is as close to a perfect submission as I think is possible to expect: it *does* include all the information a maintainer might reasonably want when seeing the code for the first time (or an academic when marking a submission). It is easy to read and coded in a way that encourages re-use.

- a program receiving 50%
- a program receiving 100%

In both cases, the document starts with cover sheet much like the one that you will receive for your own submission. That is followed by all the source code files of your submission that I marked, partly because I may refer to specific line numbers in my feedback and partly so that you can check there were no problems with FASER (which has happened in the past).