METHODOLOGY FOR DESIGNING SIGNAL AND IMAGE PROCESSING ALGORITHMS

Jacques Bourg

September 9, 2024

ABSTRACT

I summarize the methodology that I use when designing ad hoc signal and image processing algorithms. I mention some technical points, and concentrate on project management and methodological aspects of the process of designing an algorithm: from interacting with the instrumentation engineers in order to validate the proof on concept, to the algorithm industrialization.

Keywords Signal processing, image processing, computer vision.

1 Introduction

When studying signal processing, one gets to know a certain number of tools and algorithms that can be used for different kinds of images, but in a very scholar way -scholar in the sense of canonical example. In industry, in particular in instrumentation, one needs to come up with simple solutions for particular use cases for which there is no ready-made solution on the Internet. In academia, tools and algorithms are presented as if there was a perfect solution for a given problem that could ideally be derived mathematically. From my limited experience, I found that for the class of problems I had to deal with, it had a part of craftsmanship. In here I expose the methodology of developing such algorithms. I worked for three years at a life science instrumentation company. There, I developed different algorithms such as bubble detection and tracking, canal detection, cross shape landmark detection, focal point estimation, grid detection and base line removal.

2 Basic toolkit

When designing algorithms one has to come up with a series of processes (often non-linear) that transform iteratively the signal. There is no restriction on the number of processes or on the nature of the processes that can be applied: the pipeline might be simply a feedforward chain, but it might as well have parallel pathways or skip connections. Usually one starts with a signal sampled at a certain frequency, and applies certain transforms. In computer vision, when doing segmentation for instance, there is often a binarization step at the end of the pipeline to perform the last steps. In the design, this part has to be as delayed as possible, since there is a notable loss of information when binarizing an image.

The usual basic toolkit one uses for designing algorithms is not very big, and comprises some tools that we will mention in what follows (which by no means an exhaustive list).

- For analog signals: min, max, median, percentiles, Fourier transform, normalisation, running means, running medians, convolution, pointwise multiplication, integration, differentiation, polynomial fits, clustering.

- For binary images: thresholding, rounding, mathematical morphology, logical operators, convex hulls, measure of region properties.

Think of these as building blocks that combined in the appropriate order to design pretty complex algorithms. With experience, one start to learn ’tricks’, or good combination of these simple operations that work well. Most of them are non-linear operations. One of the most valuable linear operation is the normalisation operator, since it may be applied before and after different several non linear analogic processings. The median operator is another very simple but extremely powerful non-linear processing, to get read of outliers. To use and reuse these building blocks, they must be written in separate classes with clean and documented code.

3 Project definition

When assigned to a new project, you will be given a limited number of images or time series to develop your algorithm. Limited, since, it does not cover all the instrument configurations (the illumination, the object positions) in which you will find the shape to detect, and limited in number since the experimenter time is also precious.

The best thing to do when receive new data with which you have to work, is to take the time to discuss with the person that requests the algorithm, understand the context in which this data will be produced. For instance, at which step of the experiment, which variables are available, what are the time constraints of the project and of the algorithm. It is a very good time to come up with many questions about what could go wrong. It is very important to have given a first look at the data: Be sure that the data is what it should be, and come to the meeting with naive questions: what is this part of the signal, what is the noise, why is the signal this way... It often happens that there are experimental flaws in the signals, since often the engineers are doing the measurements for the first time. Giving a quick feedback might save weeks of work.

4 Development: Fast iterations of trial and error

During the initial phase, forget all the optimization constraints and forget about writing nice code or all sort of aesthetic things. The first aim is to come with a series of steps of how to solve you problem, and this implies trying several steps at a fast step. Using scripting programming languages such as Python, Matlab or Julia, and image processing libraries such as openCV or Skimage allow to iterate quickly over different alternatives.

Having a good idea when developing a signal or image processing algorithm is never sufficient. One usually has ideas about the kinds of algorithms we might apply to a kind of image. An underestimated difficulty is how to come with the right preprocessing steps that adapt your image to the input of those algorithms. Moreover, it happens frequently that you try things that seem perfectly reasonable and they don’t work as you would like in your images.

I have a fuzzy approach when I attack complicated problems, I allow myself to try many different approaches on Jupyter notebooks (that are not very clean), and start combining the blocks that make sense. It is totally fine to try things that you don’t understand fully: in skimage for instance there are several image processing algorithms that are already implemented. Nothing should stop you from trying new ones: look at the input, the output and see if it is what you are looking for. If an algorithm it is not making what you want, it might be that it is not adapted to your purposes, or that because your data is not in the best format for the algorithm to handle it. Then you can have a funneling approach to an algorithm: read the wikipedia or watch a youtube video, try to grasp the mathematical essence and see if it makes sense with regard to the processings that precede it. I would recommend to do not spend too much time on each algorithm that you try: understand what it does, why it is suited or not for your problem, and explain it to yourself in simple words to see if you really understood it. There is of course a tradeoff in your heuristic process to find an ad hoc algorithm: on one extreme you could "try things" in a an purely experimental way (without fully understanding the algorithms): this has the advantage of being fast. The opposite approach is to find the exact class of problem you are in and read the literature and so on. If you are a beginner, the first approach could make you go in circles as the combinatorics of the processings is huge, and could also be really frustrating. The other approach could just take too much time, since often there is no predefined algorithm for your problem, and be also very frustrating. Therefore, to really advance towards the solution one has to find an idea and test it in a reasonable amount of time. There are several ways to find ideas: by thinking hard, to browse on the Internet, to read the documentation (for instance in skimage and in Matlab, the documentation is very well done), or to read papers. Reading research paper might give you ideas, but you might get lost trying to understand all the details. It is therefore essential, instead of reading linearly a paper, to interrogate it to see what problem it is solving, what are the methods applied and to see if they could be used for your problem.

It is therefore interesting to try to cut your problem in pieces, and evaluate your algorithms "from left to right", meaning that you develop and evaluate your building blocks starting from the input up to a certain point and including all the steps. This gives you a sense of progression, and allows to concentrate on subproblems.

5 Robustness versus simplicity

There is another consideration when developing an algorithm is the tradeoff between robustness and simplicity: data comes with different levels of noise and if you want to make a jump detector just by differentiating your signal, it will never work with a signal coming from a sensor. The notion of simplicity might be related in analogy to the notion of over-fitting in machine learning: an algorithm that is too specific for a subspace of the parameters will not "generalize" properly. The robustness that you add to the algorithms is intended to make processings that are not very sensitive to specific set of parameters, and also that allow for variability in the range of accepted data.

Robustness might also be a problem: I recently designed signal quality controls that were working fine for a type of data, but some months later after the development of the machine went on, the quality control tests that I implemented inside the algorithms did not pass anymore. I lowered my quality control constraints and the tests passed, but the experimentalists had already lost some time. After reanalyzing the images, it turned out that the signal quality was bad overall: some week after, the engineers found that there was a prism that was not working fine since it was not glued correctly, and the light path was disturbed. During development, where everybody tweaks the machine in different ways, the best practice is to be able to see the data, and to see the effect of the processing at each step, with a particular emphasis on the quality control steps. As development gets more mature, one can stop visualizing the signal.

Therefore, the design philosophy is to make algorithms as simple as possible, putting a small level of robustness to cope with the observed noise, but you have to trust the algorithm and wait for the feedback of the experimenters, to eventually add more layers of robustness.

6 Towards a first algorithm version

I only start cleaning the code (encapsulating it in well written functions) when either I see that one part of the code comes back over and over, but also when I successfully completed a first series of coherent operations that give a promising result. I also use this time spent cleaning the code, to think back about I just did makes sense. In fact, encapsulating the code is dangerous in the early phase when you have to experiment to find the simplest and more effective processings on your data, since you tend to forget what are the operations done. A good approach to find a nice algorithm is to repeat the hot-cold process of trying things and them taking time to evaluate if the results are satisfactory or whether one can find simpler and cleaner ways to do it. If you have several projects at once, a good idea is to get a first proof of concept, to leave it aside and then come back a week later or even a month later and evaluate critically the algorithm. When evaluating your algorithm, you will de-encapsulate it again, to look again what each processing is doing to your data. Jupyter notebooks are half adapted for this: in a positive way since you can execute individual snippets of code, and plot right away. Negatively, since you would like to put breakpoints in your code.

7 Algorithm validation: presenting the algorithm and obtain the right data

Then the next thing once you have a first working algorithm is to present it to the engineers you work with :

I usually present my algorithm in a single slide in which there will be see the functional blocks linked with arrows and even -when necessary- signals at the input and at the output, so that each person can have an idea of how an important step transforms the signal. I draw those functional diagrams in Inkscape. I choose a standard size and color for the boxes and for the police: for instance all the boxes that represent data (or an intermediary output that is particularly relevant), are drawn in green. I choose blue for the algorithms (or processing steps), I draw a black box for the final output of the algorithm, and I do use syntactic elements like dotted rectangles for the loops (In some kind of LabView style). I do not recommend to write presentations in a pure Latex / Beamer format in which there are only equations and in which the algorithms are written formally as you would do for a research paper since you might loose part of your audience, either because they don't have the same background or simply because you didn't do a good job in explaining something that in fact is not that difficult if you explain it in simple terms. For this kind of algorithms, there are often several technicalities, but it doesn't matter, you should leave them aside. What is really important is to give to your audience and overview of what the algorithm is doing, ideally stem by step, and to show them several examples of where it works and where it doesn't, so that they can come with insights, or that they get convinced that your approach is promising, and that it is urgent to do more experiments to test the algorithm.

It might happen that in the development cycle, the algorithms you developed "don't work" and people complain. At this point you have to analyze all the possibilities:

1- Your algorithm is not good enough. You developed an idea based on a restricted data set, recorded with a fixed set of parameters. When you get more data, this data might be noisy in ways that you didn't expected, and sometimes this is fatal to the algorithm, since it is based on certain assumptions. I developed once an algorithm on a dataset that had a very good baseline, and I used some property of the autocorrelation to discard bad signals. It turns out that when I got new data, the baseline was slowly varying in unpredictable ways, and that I could not really remove it: the autocorrelation was not a good tool to measure the signal quality.

2- The data that is feed in the algorithm is not good enough since the machine is being run with a different parametrization. Ideally engineers should keep track of the changes that were done to the machine and of the parameters of each experiment, but an unexpected drop in the production quality may happens and go unnoticed. Instrumentation engineers use the machine for other purposes and then the parameters might vary depending on the usages.

Algorithm validation might also require to design, hands in hands with the experimenter, a testing protocol to validate the algorithm, such as the test benches in mechanical engineering. In there, different experimental conditions will be systematically tested, and we will asses the performance of the algorithm in all of them. This procedure might be particularly difficult and time consuming for the experimenters.

8 Industrialisation

Industrializing an algorithm is a hole subject in its own right. Nonetheless, we will mention some key aspects one should keep in mind when entering this phase.

As we already mentioned, the code has to be of very high quality, modular, documented and readable by any developer, this means applying good practices like classes, code linting, commenting, variable typing, docstrings, using the principle of that a function does only one thing. To document a code, in addition to the comments the docstrings, there are tools like Sphinx or Mkdocs that generate a documentation. Also, in order to start evaluating the processing time and the computational ressources, there are tools like Snakeviz, to profile the code in order to concentrate on the functions that consume more time and to optimize them. One can choose to optimize certain key parts in Cython (if coding in python) or directly recoding the functions in C. In numpy, scipy and skimage, several functions are optimized already in C, so intead of optimizing particular functions, I often find myself working on the algorithmics to speed up the code. This part can also be done by software engineers, provided that the code is clear enough and that the matematical aspects are well explained. One alternative to Python is to develop directly in Julia language. It has the advantage of having a very simple scripting synthax, and it can run as fast as C++, but has a way smaller community than Python.

9 Conclusion

I covered here some relevant aspects for the development of signal and image processing algorithms, and how to structure the development in time, to solve some aspects which seem incompatible at the first sight like algorithmic robustness versus simplicity, algorithmic deep understanding versus trial and error and code quality versus pace of ideas. I also mentioned some aspects about the project flow, including the interaction with the instrumentation engineers, and the process of being sure that the data is good and that the algorithmic solution is also good. I hope these remarks might be helpful for people entering in this field.