



# School of Physics

## PHY2004 Lecture 0

Comments and questions to [John Rowe](#).

*This lecture acts as a preface to the more-formal lectures which start next week. The aim is to introduce the main concepts we will encounter in a less formal and relatively "non-computer" way. Don't expect to have any **answers** to the issues we raise here, these will start to come over the rest of the course. Rather, the aim is that if you think about these issues now it will be easier to understand them when we address them later on.*

*This week's exercise in the computing lab will enable us to explore these in a fairly fun way before the real hard work begins in the next lecture.*

### Warm up discussion

Consider the following set of instructions:

1. Turn left out of the Physics building.
2. Take the first turning on the right.
3. At the T-junction, turn left.
4. At the T-junction, turn right.
5. At the T-junction, turn left.
6. Take the first turning on the right.
7. Turn right at the T-junction.
8. Take the first exit at the round-about.
9. The entrance to your destination is the first turning on your right.

(For simplicity assume there is no ambiguity about what constitutes a turning and we don't need to worry about roads being closed.)

Turn to the person next to you, or form a three if necessary, and discuss:

- What's good about these instructions?
- What's not so good?
- How could they be improved, particularly if we needed to extend them to a much longer route, for example to a particular house in a city hundreds of miles away?
- Do you have any other thoughts about them?

If you are reading these notes on your own it is still well-worth thinking about these questions, even if you are just revising the lecture.

### "Law of averages" errors

Imagine we were given a considerably longer set of directions in the style of the ones above without even the address of our destination, perhaps just a

I have a bad memory of a driver asking me for

house number. We would probably feel rather nervous about our chances of arriving at the right place and would certainly make sure we had the mobile-phone number of the person who gave them to us. Realistically, given a set of twenty or thirty instructions like this, one of the two humans involved, either the person who gave the directions or the person following them, will almost certainly make a mistake.

When dealing with people the best instructions to give, and the ones we most like to receive, are those that have confirming information such as "you'll see the Red Lion pub on your left" or "take the second turning on the right onto the A371".

**Simply writing a long list of instructions and looking at the final result is unlikely to be successful.** We are too likely to have made a mistake somewhere along the line and being faced with a list of hundreds or even thousands of instructions knowing that (at least) one contains a mistake but not knowing which one is not a pleasant thought.

Instead we should **test the results of obeying our instructions** along the way to see if they are correct and only proceed with writing the next ones once we are confident of the ones so far. This is easily done with directions, first because we know where we are going and what the roads and landmarks are along the way, and second because the person following the instructions is an intelligent agent who can decide whether the Red Lion is indeed on their left and if not take appropriate action.

directions which I gave starting with something like "Turn left at the first roundabout" and then remembering just after he drove off that I should have said the **second** roundabout...

## Would you drive into a river if your satnav told you to?



*A taxi driver following his satellite navigating instructions ended up stranded in a river after taking a wrong turning at a ford in the road.*

*The red-faced driver - working for a taxi company called Streamline - drove 200 yards up the River Nar in Norfolk before his eight-seater minibus ground to a halt on the muddy river bed.*

*[ [Daily Mail](#), photo [Albanpix.com](#) ]*



*A woman wrote off her £96,000 Mercedes-Benz and nearly drowned when she followed the instructions of her onboard satellite navigation device into a river... The woman, who identified herself as Hayley, 28, from London, drove her SL500 into the river where it was swept 200m downstream, bouncing from bank to bank.*

*[ [The Times](#), photo Tony Swift published in [The Evening Standard](#) ]*

The discussion in the previous section reminds us of a rather more subtle

problem: we don't expect people to follow our instructions literally. Instead we tell them what we are trying to achieve and (approximately) how to do it. We trust in their common sense to fill in any gaps and if necessary to modify or even disobey our instructions when it's clear that there's a difference between the literal interpretation of the instructions and what we are trying to achieve. And usually they do...

This more or less covers the situation we have been dealing with for most of our lives: giving instructions to humans. In this course we are learning to do something superficially similar but actually quite different:

**Measures put in place to  
stop sat-nav cliff  
diversion disaster.**  
*Darlington and Stockton  
Times*

## Giving instructions to computers

Our experience with computers so far has probably given us a false sense of security as we have interacted with them via applications (also known as "programs") written by other people. Our input consists of high-level commands such as "send this email" which are interpreted by the application which in turn sends a sequence of low-level instructions to the computer's processor to perform the task.

These programs are remarkably tolerant of our tendency to make mistakes and at working out what we want them to do even when that isn't what we actually tell them to do, warning us if we have mis-spelled a word or quit with unsaved changes to documents. They're nowhere near as good at this as a fellow human being but then again, they are much better spellers.

Now we are learning to write our own programs we will no longer have this luxury: we are giving instructions directly to the computer and it will do exactly what we say. Which is both good and bad.

## Algorithms

Computer programs are about **algorithms**.

**An algorithm is unambiguous specification of a set of instructions that can be followed in a mechanical, unthinking, manner to achieve the desired result.**

Unlike the instructions we give to human beings algorithms do not require, or even allow for, the use of common sense or intelligence. The *agent* executing the algorithm has absolutely no choice in what steps to take. All the intelligence is in the algorithm.

The instructions we discussed at the beginning of this lecture are an example of an algorithm as they do not allow the application of individual intelligence or judgment when executing them.

## Another example of an algorithm

Suppose we are asked to write a computer program to calculate the wages

due to an hourly-paid employee given their hourly pay and a set of time-sheets saying how many hours they had worked each day. The specification might be:

*An employee's pay is the total number of hours they have worked multiplied by their hourly wage, with Sundays paying double time and Saturdays time and a half.*

Such a specification is not in itself an algorithm as it is not a sequence of instructions, but it is not hard to come up with one. A typical example might be:

- For each employee:
  1. Initially set the running-total (of the employee's pay) to zero.
  2. For each day in the time period under consideration:
    - i. Multiply the employee's hourly-wage by the number of hours they worked that day to obtain that day's pay.
    - ii. If the day is a Sunday multiply that day's pay by two.
    - iii. Otherwise, if the day is a Saturday multiply that day's pay by one-and-a-half.
    - iv. Add that day's pay to the running-total.
  3. At the end of this the running-total is now the final total of pay owing to that employee.

Given such a set of instructions a person of average intelligence could calculate the pay due to each employee.

Our algorithm above has several of the features and issues found in just about every computer program. All of these are important and all of which are simple to understand and deal with in a problem **of this size**. As our program grows in size some of these issues will remain simple and manageable. Others will not. Let's look at a few of these concepts in turn, as they illustrate the major issues we will be covering over the next few weeks.

## Data

The algorithm has terms such as "*hourly-wage*" which serve as "place holders", to be replaced by the actual value of that quantity when the algorithm is run, as well as others such as "*running-total*" which are calculated at the time. Furthermore the value of "*running-total*" is not a fixed quantity but is updated as the algorithm progresses.

In fact, if we look at the description of what we were trying to achieve we will see that it is defined in terms of data: what we want to know and how it is related to what we already know.

Growability factor: large.

## Sequences of instructions which are performed several times (loops)

"*For each employee ...*"    "*For each day ...*".

Almost every program we ever write will contain a number of these, and the C programming language contains several convenient ways of defining them.

Growability factor: small.

## If ... otherwise

"If the day is a Saturday ..." "otherwise ..."

(To save typing, programming languages tend to use the word "else" instead of "otherwise".)

These steps, often called as **conditionals** refer to steps ("multiply that day's pay by two.") which are only performed if a certain condition is met ("If the day is a Sunday").

You may have noticed that our two criteria ("It's Saturday" and "it's Sunday") are mutually exclusive so we could have missed out the word "otherwise" and simply written:

- ii. If the day is a Sunday multiply that day's pay by two.
- iii. If the day is a Saturday multiply that day's pay by one-and-a-half.

However, the two are not always equivalent, for example a menu application might say "If the diner is lactose-intolerant substitute non-dairy ice-cream, if the diner is gluten-intolerant substitute rice noodles...", in which case adding "otherwise" to the second and subsequent conditions would be a mistake. (In our actual Sunday/Saturday case having the word "otherwise" is clearer which is why we prefer it.)

Growability factor: medium.

## Requirements change

We can illustrate this by noticing that our specification does not account for Bank Holidays (also known as public holidays). This could easily change, either because somebody realised they had forgotten about them, or because of a genuine change in pay agreements after the original specification. **Both of these are very common** and should be borne in mind when writing our computer program. The revised specification might be:

*An employee's pay is the total number of hours they have worked multiplied by their hourly wage, with Sundays and Bank Holidays paying double time and non-Bank-Holiday Saturdays time and a half.*

Note the careful definition: had we said "Sundays and Bank Holidays paying double time and Saturdays time and a half" the question would have arisen: "Do Bank Holiday Saturdays pay time and a half, double time or triple time?". **Look out for ambiguities in specifications**, they make lawyers rich and programmers make mistakes.

The relevant part of our algorithm is now:

- ii. If the day is a Bank Holiday or a Sunday multiply that day's pay by two.
- iii. Otherwise, if the day is a Saturday multiply that day's pay by one-

and-a-half.

Notice that missing out the "otherwise" would now cause an error. **We will always need to manually "walk through" our algorithm to check for errors.**

A walk through will catch an error if it's a simple oversight. But suppose we implicitly assumed that all Bank Holidays are Mondays (as they tend to be in the UK), forgetting about Christmas and New Year's days. The walk-through wouldn't catch that as we would think it was correct. **Be even more careful about assumptions than you are about ambiguities.**

Growability factor: medium.

## It's part of a larger whole

Although we presented this as a standalone task it obviously isn't: in reality before any employee was actually paid we would need to calculate and deduct their tax. But that didn't really matter - we were able to present this as a separate task without thinking about the complete payroll package. This is a very important principle and directly impacts on the next issue.

Growability factor: large.

## It requires us to think about more than one thing at a time

Although our "directions" example is vulnerable to "law of averages" errors, it has one huge simplifying factor: we only have to think about one thing at a time, namely the next direction. Our wages example does not have this, whilst calculating the day's salary we have to think about two or three things at a time (hourly wage, hours worked and the day).

Growability factor: "Look out, I think it's coming this way!!".

### Exercise

In pairs or threes play multiplication ping-pong: one person says two one-digit numbers ("seven times two"), the other person says the answer and asks one back ("fourteen. Four times eight"). It's not a race but try to establish what the comfortable speed is. Then up it to a one-digit and a two-digit number ("four times twenty-seven"), after few of those go up to two two-digit numbers, etc.

How would you describe the shape of the graph of difficulty plotted against complexity? (I.e., how quickly does it get more difficult as the number of digits increases?)

**This is the single biggest issue we will face** and we shall be thinking about strategies to deal with this over the next few weeks.

## Other issues

One factor that's so obvious that you probably didn't notice it is that when we displayed the algorithm we automatically indented the levels of bullet points to reflect the structure of the algorithm. Bullet points are always

Did you notice that the directions we discussed at the start of the lecture took us the wrong way down a one-way street? Did you ask which exit we were leaving from?

These are typical "assumptions" problems: the instructions themselves are fine but we have made an assumption about exactly what we are trying to do (go by foot or by car) and precisely where we started from.

displayed this way of course, and so are computer programs and for the same reason: it makes them much easier to understand.

Finally, one major aspect of programming this example does not illustrate is the handling of errors ("I can't find Fred Smith's time sheets!"). We will start to address this at the start of the next lecture.

## To think about after the lecture

Over the next few weeks, in addition to learning the mechanics of the C programming language, we will start to address the following points:

- Whether our instructions are right or wrong, the computer will follow them with a level of mindless obedience that makes people who drive into rivers because their sat-nav told them to look like geniuses in comparison. We can't assume the computer will know what we mean!
- We can't rely on being able to write a large number of instructions without making a mistake. Instead we write a small amount at a time and test, test, test.
- The data our algorithm operates on is actually the most important part of our thinking.
- If the exercises we give we were to be part of a larger whole, how would that change the way we answer them?
- Plan for the specification of our program to change.
- We check the logic of our program by "walking through" the algorithm.
- We can only think about a few things at a time.

Thinking about these things now will make it easier when we come to cover them later on.

---