Software for Experiments

Tim Kovacs and Rob Egginton
Department of Computer Science
University of Bristol

July 31, 2010

Abstract

This is the handout for the half-day *workshop on software for experiments* held 27 July 2010. The content is very uneven: some parts are just the rough notes we prepared before the workshop while others are much more polished. We also include lists of ideas generated by participants.

Contents

1 Introduction

Writing software to run experiments is somewhat different from other applications. Since undergraduates are not taught specifically about software for experimentation, virtually all PhD students in computer science end up reinventing the wheel. We felt it might be useful to run a workshop on this subject based on what we learned during our PhDs (and since).

By "software for experiments" we mean software that generates and uses data. Typically lots of data, not just a few numbers. Usually the results depend on various parameter settings and often many experiments are run to explore the parameter space. Often the algorithm itself is experimental and modified over time. Sometimes the results are stochastic and must be averaged over multiple runs. Usually other post-processing is needed such as generating descriptive statistics, running statistical tests, and plotting graphs. Simply managing an archive of old results and keeping track of the parameters which generated them is an issue. We hoped not only to share our approaches but, since we don't have all the answers, to find out how participants handle these issues. We were really pleased with the contributions made by the participants and we definitely learned something from it.

Although we didn't discuss design patterns explicitly, to some extent the workshop could be thought of as a discussion of design patterns for experimentation. This includes design within an application, but also at the level of tool chains (combining applications).

So far we've referred to "software for experiments", which we feel is a little clearer at first glance than "experimental software", which could be about the design of programming languages. However, "software for experiments" is a bit of a mouthful so for the most part we refer to "Experimental Software" or ES in this document.

2 What is Software for Experiments?

2.1 Group work: What is experimental software?

Here we list the answers generated by the groups and during the following discussion with the whole workshop.

Group 1: What does good look like? List as many characteristics of good ES and good experiments as you can.

- Well documented
- Modular / flexible
- Repeatable
- Understandable, clear / transparent
- Portable (Operating System independent)
- Built-in self test
- Scalable as necessary
- Shouldn't be over-engineered
  - Focus is the experiments
- "Publishable" software
- Captured in a backed up repository
- Accuracy
- Concise output
- Stability

Group 2: What are the challenges of writing software for experiments?

- Lots of input parameters (usually in a mess)
- Lots of output data
- Preprocessing
- Logging parameters
- How accurate vs. complexity
- Very specific (no one can help)
- When outputs influence code; when to feedback parameters
- Large computation times (and experimentation times)

What can go wrong?

- It can be a waste of time (unassisted)
- Versions get mixed up (because comments missing!)
- Relics of old intentions

Group 3: What makes software for experiments different from what you write for other applications?

- ES may be more original
- Often individual (not a team)
- Written from scratch
- No “end user”, just self (or experts)
- Don’t know what end product will be
- Prototypes
- More data
- Never finished
- Configurable / parameters
- Goal: learn (vs. product). Product isn’t the software
- Many interfaces between components. Many languages. Combined in new ways
- “Alpha” quality: can crash sometimes. Performance sometimes not great
- Not portable
- More language choice
- Parallel

2.2 Discussion

This was prepared before the workshop and doesn’t cover all the issues raised by participants. To us the main difference with software for experimentation is the addition of science to a mainly engineering subject. Experiments must be replicable so we need to know all parameter settings and keep a copy of the exact version of the code used. Also, we often want experimental code to be incremental and interactive and hence scripting languages are popular. Other differences are engineering rather than scientific: planning for change and planning for reuse by ourselves and others These are requirements for most software but there are differences between ES and other software e.g. ES is likely to change more.

Types of science and types of ES. All science is exploratory but some areas more so than others. When it comes to writing code there are two extremes: at one we design a system by writing it whereas at the other we know fully in advance what we need the code to do and the exploration takes place outside the coding. (Nonetheless, although we may know *what* we want the code to do we may still explore *how* to implement it. However, the scientific exploration is external to the programming.) In the first case(design by writing code) we have greater requirements for interactive software and rapid prototyping. We tend to think of ’software for experiments’ with the first case in mind, i.e. the software is the experiment, not just a tool for processing data that comes out of an experiment. However, there’s a spectrum and this workshop should be relevant to some extent to any point on the spectrum.

To make it easy to explore the design of an evolving program you want to work with the simplest language that gives you the power you need and to work at a high (i.e. abstract) level. Three language features that make this easier are: rapid prototyping, interactivity and custom languages. We’ll look at each in turn but first we introduce script languages, which have these features.

Script languages Script (or scripting) languages originated as a way of gluing other programs together e.g. to call a program and pipe its output to another program. Script languages such as Python and Perl are now widely used as general-purpose languages and the definition of a script language is now a bit unclear. However, for our purposes the relevant points are that scripting languages are suitable for rapid prototyping, allow interactivity and are often a good choice for implementing custom languages. Matlabs built-in language is a script language. For more see `http://en.wikipedia.org/wiki/Scripting_language`

Rapid Prototyping Being able to rapidly create and change a working demonstration system is extremely useful for exploratory work. Rapid prototyping is supported by a number of language features. For example, strongly typed languages require you to define the type of all variables before using them. This allows the compile to catch many kinds of errors, but the programmer spends much of his time satisfying the type system. Script languages simplify matters by not requiring type declarations. This comes at a cost, however, since they cant do the type checking they otherwise could, which results in bugs and arguably makes them less suitable for developing large-scale applications.

Interactivity Interactivity means being able to alter the flow of control of your program as it is running. For example you might want to display or change the value of a variable at a particular point during the execution. There are several ways to achieve interactivity. One is to use an interpreted language in which case the flow may be defined entirely by the functions you call manually. This is the typical use of interactive environments such as matlab.

An alternative to using a script language is to use a script front end to any library for your domain as long as the script is vertically integrated. By this we mean not just a shell script which can only call up the library, but a script which lets you delve into the guts of the library and interact with it.

Another alternative is to use the unix prompt as the interactive environment i.e. split your code up into modules which you can then call and pipe data in and out of.

Custom languages A custom language is one which is specialised for a particular purpose. A relevant custom language supports rapid prototyping and hence exploration because much of the functionality needed has already been written for you. Many languages are both general-purpose and customised for certain purposes. Matlabs scripting language can be seen, among other things, as a custom language for working with matrices. Another example is string processing and Perl and Python are well-known for their powerful string manipulating features.

An example from Artificial Intelligence (AI) is the Prolog language which allows you to describe a problem declaratively (by stating a set of facts) and then asking Prolog questions. For example you can state that Barry is the father of Imi, who is the mother of Rose and so on. Then you can ask it questions like "How many grandchildren does John have?". Prolog knows how to answer questions like this (by doing search in a data structure) so you do not need to write an algorithm to answer the questions. Researchers have developed "AI languages" such as Prolog and Lisp and, interestingly, one of their keys feature is rapid prototyping. This is not surprising since one of the main ways of experimenting with AI has been by writing programs (in other words, design-by-writing). (We say "has been" since in recent years AI has become much more statistical and much less about design-by-writing.)

It' also possible to write your own custom language and this is a very powerful paradigm. At one extreme a custom language can be a general-purpose language but at the other it can be very simple and highly limited. For example, the "make" utility checks checks dependencies between files and triggers updates on any out of date files. (A major use is to control the compilation of C/C++ programs composed of multiple files. Make will recompile only the source files which have been edited since their compiled file was generated, and any source files which depend on these files.) Make reads in makefiles which use a simple custom language to specify dependencies between files. Make is a simple interpreter and the makefile is a simple program. Although make is not a general-purpose language its also moderately complex and much simpler examples exist. Configuration files can be thought of as very simple custom languages. For example, suppose a program reads a single number from a text file and sets one of its variables to that number. This is the simplest case we could think of, and it's stretching the idea of a custom language, but we can think of the text file as a (very) simple program and the program which reads it as an interpreter for that program. A less simple example is to create a file called "text" and to issue this command at the unix prompt:

```
cat text | banner
```

The program cat reads the file called "text" and prints it. The | tells unix to capture that text and pipe it as input to the banner program. Banner then prints out the text in very large ASCII graphics (suitable for printing on a banner). In this case banner is an interpreter and "text" is a simple program which specifies what the graphics will spell out. The way this is specified is particularly convenient: the text on the banner is the same as the text in the file. This illustrates how a custom language can be very convenient. Another approach would be to use a file to specify the coordinates of each ASCII character appearing in the banner. This could be made to produce the same output as banner but would be vastly more difficult to use. Yet another approach would be to write a program in a general-purpose language to print out a particular banner and to hard-code the contents of the banner in that general-purpose language. Again, this is vastly more difficult than creating a text file.

2.3 Exploratory and Confirmatory Experiments

- create a system / gather data by observation
- Explore this system / data by iterated experimentation and visualisation

- ask a question based on what you've observed
- form a hypothesis
- test the hypothesis (on new data)
- analyze the results
- draw a conclusion
- communicate results

The second type is where we must consider our knowledge of statistical tests and pick the right one. The former is more possible in software than in physical experiments as a large number of tests can be run and analysed to look for areas of interest without normally great expense. Thats one advantage of a traffic simulator over observing and affecting real traffic.

2.4 Experimental design principles

Some experimental design principles are less important in a simulated environment than in practical engineering. For instance, randomisation is important to ensure that sources of error are equally spread, but most computer simulation lacks the attribute of error creep. Whereas a machine might wear down or break in, necessitating randomisation of sample points, this is just less likely, though obviously those interfacing with networks or external systems can come across these real-world issues.

Many principles of good scientific process apply just as well to experimental software as to physical or social settings. We'll quickly review some of these.

Replication

- If there is stochastic variation in your system, from whatever source, obviously you need replication in order to improve accuracy in terms of mean and standard deviation.

Variation sampling

- Pick your parameters and the levels you sample at based on the expected variation. If one area is expected to have a high level of noise, increase the sample points in that region. This is obviously part of iterated experimentation.

Replicability

- Different from replication, and an important principle, as if there are not enough details for an experiment to be replicated then the scientific aim of peer-review breaks down.

Factorial over OVAT approach

- Example: Experiment with 2 factors at 2 levels. One is varied, then the other.
- Q: With the OVAT approach, what is being missed?
- When factors and levels are too high, a fractional approach must be taken. Much written on how to choose experiment points in order to discover interactions between variables. Search for Plackett-Burman and Response surface designs.

A Good statistical source for engineers, including chapter 5 on design of experiments: NIST/SEMATECH e-Handbook of Statistical Methods `http://www.itl.nist.gov/div898/handbook/`

3 Developing Software for Experiments

Developing software for experiments involves both practical and social considerations. One of the first practical choices that has to be made when starting a new piece of software is what language to use. While this isn't absolutely specific to experimental software, it's an issue that always crops up. Let's briefly consider what we want from a language when we are about to start building a system.

Brainstorm:

- What features do you want of a language?
- How do you decide what language to use?

Expected answers:

Performance

- runtime speed/memory efficiency
- speed of writing
- interactive shells

Language Power

- structure (strong/weak typing)
- specific language features (e.g. functions as types)
- availability of libraries

Social

- what others use in our field
- what we're comfortable with
- what's available to us (commercially/what's installed/what works on system)
- ease of understandability (now and in the future)
- online support community

Note that speed of writing is initially very important. We can migrate to something stronger-typed later if need be.

3.1 The right tools for the job

You should always use the right language for the task at hand. There are many factors to consider in choosing a language, but assuming you want a rapid prototype, this is the language in which it's simplest to write the solution. "Simplest" involves minimising multiple criteria: time and the amount and complexity of code. (We assume you know the candidate languages already, otherwise you have to consider whether its worth learning a new one). Often you actually need more than one language. Experimental software usually has at least two parts: the experiment itself and running the experiment with different parameters. The two are distinct problems and the best language for running experiments may not be the best language for writing the experiment. Sometimes we also write several programs in the same language in order to achieve modularity/a pipeline/interactivity at the unix prompt and storage of intermediate values (so we don't need to rerun everything to return to any point).

Often we end up with a complex system involving multiple programs in multiple languages, each solving a particular aspect of the problem. Such a system really needs documentation about how it all fits together, simply for your own use, let alone to allow anyone else to use it.

3.2 Running example: a traffic simulator

In writing ES we often want both exploratory software development and scientific rigour (e.g. reproducibility). There's some tension between the two as rapid prototyping does not naturally lend itself to rigour. The balance between the two is non-stationary: we tend to be more exploratory at the outset and want more rigour later. Now we'll look at a running example which illustrates this shift.

A traffic simulation Stu has been asked to research possible routes for ambulances travelling from hospital A to hospital B and to do so he writes a non-deterministic traffic simulation. To begin with he just wants to estimate the time needed to get from A to B although later he'll study how to optimise the route. The simulation could have very many parameters such as density of cars, density of pedestrians, traffic light control policy, and weather conditions and so on, and in a realistic simulation they would all vary over time. However, to begin he focuses on a subset of parameters. Each time he runs his system it will simulate one journey from A to B and return the time taken (a Monte Carlo approach). We now follow Stu through the stages of implementing his system, but we emphasise that his solutions are not the only ones nor necessarily better than others.

1. **Prototyping** Stu anticipates that implementing his simulation will be highly exploratory. In any simulation one must decide what level of detail to work at, and which parts of the original are relevant. Additionally, the domain to be simulated is complex and Stu wants to start simple and incrementally increase the complexity of his simulation but he isn't sure order to add features in. Stu decides to work in python which allows him to rapidly prototype his simulation and interact with it.

   During the initial prototyping stage the code is changing a lot and Stu is often not sure whether a feature should be a parameter to the program and whether it should be hard-coded. (He finds that often he starts by hard-coding something and later makes it into a parameter). At this stage he hard-codes everything which means that to change his algorithm at all he must edit the source code. Luckily since Python is interpreted he does not need to recompile each time he does this.

   Each run outputs the time taken from A to B and Stu inspects the output informally.

2. **Univariate experiments** Now the code is more stable and Stu knows what basic parameters he wants the simulation to have and he wants to experiment with them. He refactors his code so that instead of hard-coding certain things (like the density of cars) they become a parameter.

   Parameters can go either in a parameter file or be passed in from the prompt. Stu thinks the density of cars may be the most significant parameter and he evaluates 5 settings for it. As the simulation is non-deterministic he evaluates each setting 10 times, for a total of 50 runs. He loads the output of these runs into excel, averages the 5 sets and plots the car density vs. average trip-time.

3. **Multivariate experiments**

   a) **A script to start experiments**

   Scenario: Now Stu wants to evaluate 2 parameters with 10 possible levels each for a total of 100 distinct parameter settings. Stu realises that manually setting up each experiment will be tedious.

   Q: How can he avoid typing in these 100 commands into the command line?

   Stu's solution: Stu creates a script to call his simulation 100 times and to concatenate the output into a file.

   General note: This is a factorial experiment.

   b) **Tool-chain**

   Scenario: Manually loading the output of all these runs into excel is taking a long time, and Excel doesn't seem to have many options for plotting 3-dimensions of data.

   Q: What options do we have for automating this process?

   Stu's solution: Stu changes the output to print the average of the runs in a format that gnuplot understands. Each time the program is called the output is appended to the gnuplot input file. A separate command then calls gnuplot to create a 3d surface that plots the two parameters on two axes and the travel-time on the third.

   c) **Recoding parameter settings**

   Scenario: Unfortunately for Stu his algorithm has more than 2 parameters that need to be explored. Stu starts to run experiments varying 5 parameters and generates lots of plots, many of which look similar.

   Q: How can he ensure that each plot is associated with the parameters that it was run with?

Stu's solution: He adds the other 3 parameter settings to the name of each PDF file. [just 3 as the other 2 vary in the file]. He might also keep a paper or electronic note of his research results and notes on how these were made.

d) Storing raw data

Scenario: Stu now wants to evaluate a different route for the ambulance. He changes the simulation and adds a parameter to choose between routes. Stu thinks the change may be an improvement but it isn't obvious so he wants to use a t-test to compare them.

Q: What options are there for running the t-test?

Stu's Solution: Stu could use Excel, but this is hard to automate.
He could continue with Python, but he actually knows how to write this more easily in R, which is better for plotting and analysis than excel, and at least as good as Python.
However, there's a problem. For the t-test, both means and standard deviation are needed. Stu has changed the program to output means for gnuplot. Now he needs to edit the program, breaking the output that went to gnuplot.
At this point he realises the usefulness of splitting programs up into modules (with low coupling). By outputting all the relevant information to files, different post-processing scripts can either prepare the output for gnuplot, or for R. Another advantage of this approach is that once the data is generated the program does not have to be run again if another format is required – only a new post-processing script is needed.

Recap: current state of the system.

- Compare the current state with the factors that we flagged up at the start:
  - reproducibility
  - ease of change
- The parameters, output measurements and code version are all experimental variables. In controlled experiment parlance, there are several types here:
  - Two types of dependent variables:
    - System variables - your set of experiments is likely to hold these constant, e.g. how many runs, type of experiment
    - Individual variables - these are measured in each test, e.g. trip-time, accuracy.
  - Also, independent variables, which you can consider control variables (though they don't have to be controllable if this varies but you do not directly control it, such as how busy the network is when you run a traffic-analysis experiment.

4. Better documentation

Scenario: Stu now has a number of scripts and intermediate files. Stu comes back from holiday and has no idea how it all works. One problem is there are now several entry points: what order do the script run in?

Q: What's the solution?

Stu's solution: Once he figures out how to use his system he writes up the documentation for his own use. See figure 1.

5. Collaborating

Scenario: Stu now has an MSc student working on a GUI, who needs to change the code at the same time as Stu.

Q: What options are there to avoid problems with altering the same code?

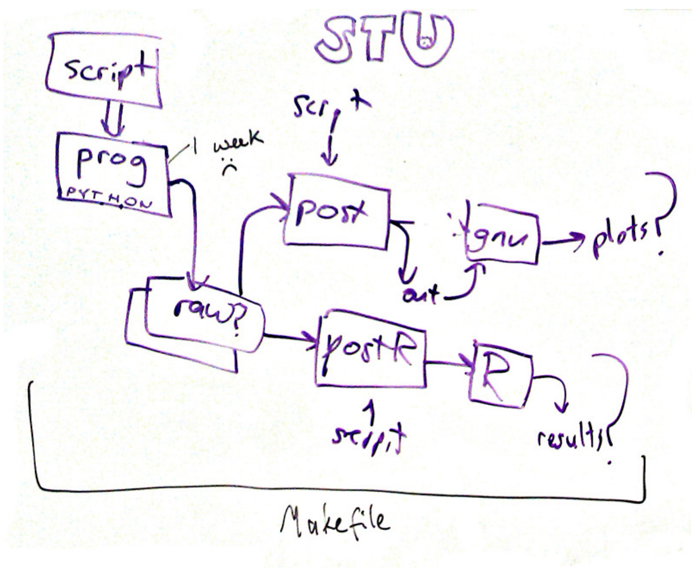


Figure 1: Stu's tool-chain after returning from holiday

Stu's solution: Better documentation, version control and use of collaborative tools.

Version control. In a version control system files are held in a central repository and each time they are updated they are given a new version number. A file can be checked out, the local copy edited, and then checked back in. If more than one person has edited a local copy of a file at the same time their changes are merged. In simpler cases this can be done automatically, in more complex ones it must be done manually. A quick example can be found here: `http://svnbook.red-bean.com/en/1.2/svn.intro.quickstart.html` Advantages of using a version control system include:

- Merging changes. It's easier for multiple people to work on the same code without breaking it.
- Incremental backup. You can return to an earlier version before you added a feature or refactored the code. With a server setup you can also be more sure that you can access your code anywhere and that you are less likely to lose it.
- Documentation. When checking a file in a short comment can be made on what was done to it. This is very useful for keeping track of how the system evolved.
- Keeping up to date. Does this folder have the latest version? Can you remember? Just check the version number.

Note that files can be of any type, not just source code. Although version control is undoubtedly useful for larger projects it's less clearly useful for smaller ones. Nonetheless some people routinely use version control even when they're working alone on a project.

SVN has replaced CVS as the standard tool for version control. GIT is getting very popular and apparently has a number of advantages. For one it separates the issues of version control and the storage location. However, we don't have any experience with it.

Collaborative tools. In recent years a lot of collaborative tools have appeared. Many are free and some are commercial. Many, such as Google calendar, are useful even when working alone but can also be used collaboratively. We have limited experience with them but do use several.

- Version control tools

- Wikis. Bristol's wiki is called confluence https://wikis.bris.ac.uk/. It allows external accounts.
- Bug tracking tools e.g. trac
- Project management tools e.g. MS project (includes Gantt charts) and trac
- Todo lists e.g. http://todoist.com
- Grid /cloud computing
- Google online tools: calendar, docs, wave (which aims to merge email, wikis, instant messaging and social networking for real-time collaboration) http://en.wikipedia.org/wiki/Google_wave
- Basecamp (commercial)
- Documentation tools e.g. doxygen
- Shared filespaces. Dropbox www.dropbox.com automatically mirrors a folder on a set of computers (e.g. your home and office) and on a cloud. Multiple users can share the filespace. This is more basic than a version control system but does allow multiple people to work on a set of files.
- Mendeley's "research management tool" http://www.mendeley.com/
- The *Science Online* conference in August 2010 looks relevant http://www.scienceonlinelondon.org/index.php

6. Publishing Code

Scenario: The code is now in a good state and Stu is publishing work based on it.

Q: What information from his experiments needs to be made public? What's the best way to do this?

Stu's solution: Stu polishes the code and documentation and checks it in to ROSE http://rose.bris.ac.uk/ (Bristol's Repository of Scholarly Eprints) which provides a permanent URL and passes its contents on to various sites which aggregate such information.

He might make it open-source, allowing use of tools like SourceForge http://sourceforge.net/

Now that his code and parameters are public his work is fully reproducible and Stu feels that he has done good science.

Conclusion of Stu's example Stu went through many stages with his software, from a very basic and highly exploratory system to a stable, well-documented, public domain release. Obviously not all projects will reach the final stage that Stu's did, but equally, not all will start at the first one: often it's possible to start by building on top of an existing library.

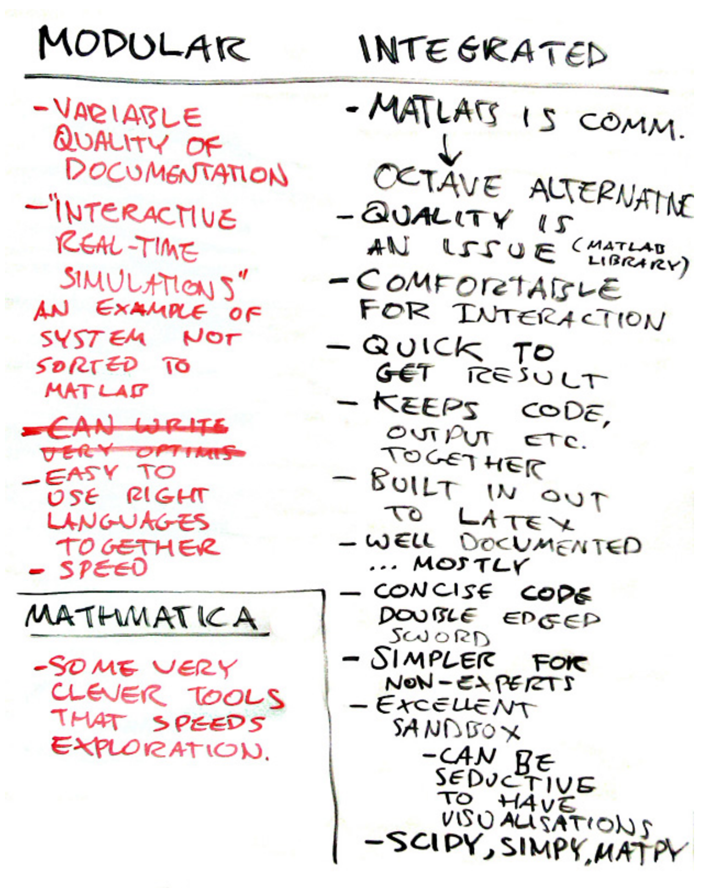
3.3 Modular tool-chains vs. integrated environments

Stu developed a modular tool-chain in unix composed of different languages and utilities. An obvious alternative is to use an integrated environment like Matlab or Python, both of which allow exploration and rapid prototyping, and include powerful plotting and statistics within the same environment.

Joint Discussion: advantages of integrated environments and of modular tool-chains. What are the advantages of each approach? *Before the workshop we came up with these lists.*

Advantages of modular toolchains:

- Parallel processing, we have clusters at the university, modular systems may be more easily parallelised.
- Easier to maintain and debug individual elements?
- Easier to integrate programs and libraries in different languages.



MODULAR

- VARIABLE QUALITY OF DOCUMENTATION
- "INTERACTIVE REAL-TIME SIMULATIONS" AN EXAMPLE OF SYSTEM NOT SORTED TO MATLAB
- ~~CAN WRITE VERY OPTIMIS~~
- EASY TO USE RIGHT LANGUAGES TOGETHER
- SPEED

MATHMATICA

- SOME VERY CLEVER TOOLS THAT SPEEDS EXPLORATION.

INTEGRATED

- MATLAB IS COMM.
  ↓
  OCTAVE ALTERNATIVE
- QUALITY IS AN ISSUE (MATLAB LIBRARY)
- COMFORTABLE FOR INTERACTION
- QUICK TO GET RESULT
- KEEPS CODE, OUTPUT ETC. TOGETHER
- BUILT IN OUT TO LATEX
- WELL DOCUMENTED ... MOSTLY
- CONCISE CODE DOUBLE EDGED SWORD
- SIMPLER FOR NON-EXPERTS
- EXCELLENT SANDBOX
  - CAN BE SEDUCTIVE TO HAVE VISUALISATIONS
- SCIPY, SIMPY, MATPY

Figure 2: Modular vs. integrated approaches. The bottom-right entry refers to python being a powerful integrated environment which includes the scipy, simpy and matpy libraries.

- Allows use of Makefiles to minimise code re-runs.

Advantages of integrated environments:

- Only one language needs to be known
- Do not need to transform internal representations to output and input text
- Can therefore be faster
- More easily packaged and shared with others

The results of discussion during the workshop are in figure 2.

3.4 Recent developments in software for experiments

Here are some recent interesting examples which we take as evidence that the need for good ES is being more widely recognised.

Experiment Databases for Machine Learning

> "An experiment database is a database designed to store learning experiments in full detail, aimed at providing a convenient platform for the study of learning algorithms.
>
> By submitting all details about the learning algorithms, datasets, experimental setup and results, experiments can be easily reproduced and reused in further studies.
>
> By querying and mining the database, it allows easy, thorough analysis of learning algorithms while providing all information to correctly interpret the results."

While this is an interesting and potentially very useful idea there is danger of research becoming too standardised, so that a community begins overfitting its solutions to what's been made easily available. The database can be found at http://expdb.cs.kuleuven.be/expdb/index.php.

The UK Software Sustainability Institute The SSI will "work in partnership with research communities to identify key software that needs to be sustained." The project started 1 June, 2010 with an EPSRC grant of approximately £5 million. See
http://gow.epsrc.ac.uk/ViewGrant.aspx?GrantRef=EP/H043160/1

JMLR Open Source Software Track The Journal of Machine Learning Research is highly regarded in its area. To support the development of open source software for machine learning it publishes short descriptive papers on such software and maintains a software archive http://jmlr.csail.mit.edu/mloss/.

"To support the open source software movement, JMLR MLOSS publishes contributions related to implementations of non-trivial machine learning algorithms, toolboxes or even languages for scientific computing."

4 Visualisation

The same approaches we use to analyse our data are useful for reporting information to others.

```
- Visualisation
- Visual data analysis
   interactive viewing, understanding and reasoning process
     - Conversion of numbers -> images
     - humans are generally poor at raw numerical data analysis
     - human visual reasoning allows robust analysis of visual stimuli
       -> convert numerical analysis into visual analysis
 - Combining multi-dimensional data into a single image so that it is easy to
    understand
 - Surfaces and volumes can show one or more series in 3 dimensions. EX
   - Extra dimensions can be shown by: EX
    - Animation
    - Colour
    - Size
 - high-dimensional data
     - Choice is to manually select dimensions for plotting, reduce the
       dimensions to those that are most interesting, or to perform a set of
       pair-wise comparisons. EX
    - Dimensionality reduction
    - This is useful when you either do not know which dimensions are the most
      important and need to find the factors that are most correlated with your
      dependent observations, or when you are attempting to show this.
    - Feature Selection - Supervised Learning
    - Feature Extraction - e.g. PCA
 - Networks -> Graphs
    - Much data can be represented as graphs. It is considered by some to be the
     first new science in a while, dealing as it does with connections and very
     large data-sets. Graphviz is a first-stop tool, though there are others
     out there that specialise in visualisation of large data sets.
 - Tools for plotting
    - JFreeChart, gnuplot, Matlab, R, VTK, Processing, Python with SciPy.
    - Advantage of the last is that it's free, and combined with the power of python.
```

5 Reading

- Paul R. Cohen. *Empirical Methods for Artificial Intelligence.* MIT Press, 1995. Very little of this book is specific to AI. http://www.amazon.com/Empirical-Methods-Artificial-Intelligence-Bradford/dp/0262032252

6 Acknowledgements

Thanks to all the participants and to Dave Cliff and Peter Flach for some references.