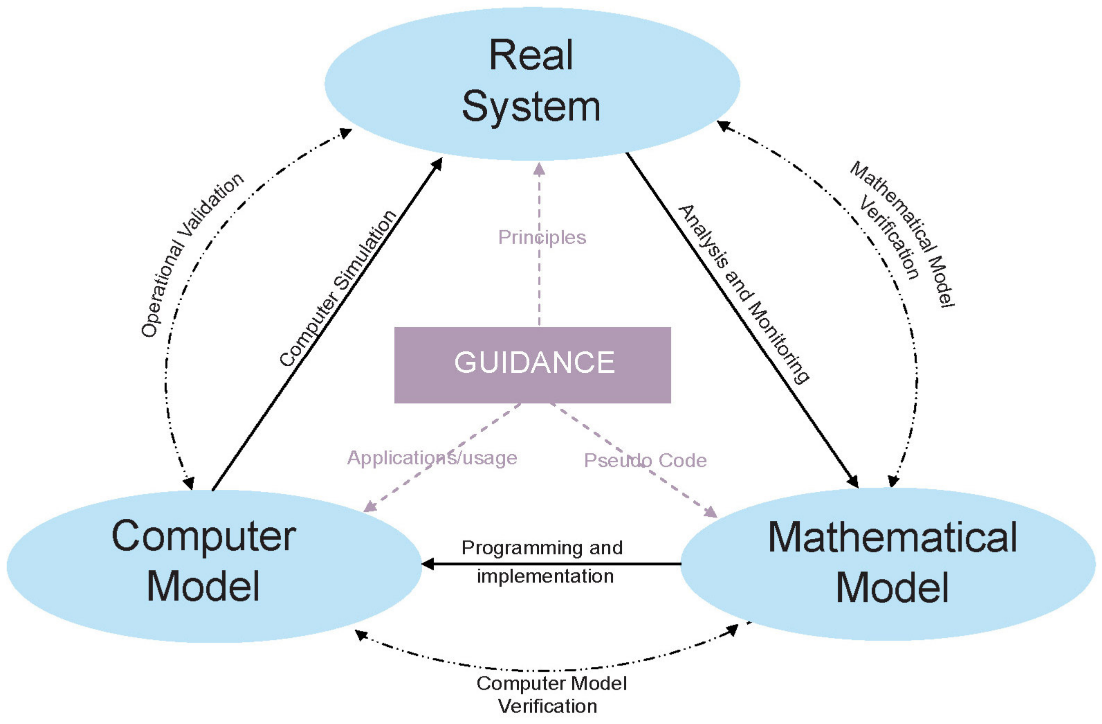
Flood Risk Management Research Software: Guidelines for Software Developers



Dr Jon Wicks	**Halcrow Group Ltd**
Dr Matt Horritt	**Halcrow Group Ltd**
Dr Matthew Roberts	**Capita Symonds Ltd**
Dr Hamish Harvey	**Newcastle University**

May 2008
FRMRC Research Report WP5.1
Project Website: www.floodrisk.org.uk

Document Details

Document History

Version	Date	Lead Authors	Institution	Joint Authors	Comments
001	5 Feb 2008	Jon Wicks	Halcrow		First draft - incomplete
004	26 Mar 2008	Jon Wicks	Halcrow	Matt Horritt	Draft
005	16 May 2008	Jon Wicks	Halcrow	Matt Horritt, Matthew Roberts, Hamish Harvey	Version addressing comments following review
006	23 May 2008	Jon Wicks	Halcrow	Matt Horritt, Matthew Roberts, Hamish Harvey	Response to internal review, final version

Statement of Use

This report is intended to be used by researchers working on flood risk management software development. It describes certain software development principles and activities that, if adhered to, will enhance the likelihood of uptake of the outputs of flood risk management research. The guidance is focussed on research in the UK (and it is expected that FRMRC2 researchers will comply with the guidance) although the findings are also of more general applicability.

Summary

The flood risk management research community have made and will continue to make significant contributions to the development of software to help inform flood risk management. However it is common for there to be barriers to the further development and/or uptake of research software due to, for example, blocks to collaboration (eg inadequate documentation, lack of access to modules and their source code), lack of evidence of suitability for particular purposes (eg lack of appropriate testing/benchmarking), and IT systems issues (eg being incompatible with users IT hardware/software).

This report draws together guidance which, when applied, will increase the likelihood that flood risk management research software will benefit the flood risk management process. Key elements of the guidance are:

1) Software should be designed and coded in accordance with the following principles:
    - Principle 1 – Develop modular software
    - Principle 2 – Write self-documenting source code
    - Principle 3 – Provide flexibility in computer hardware requirements
    - Principle 4 – Use standard development languages
    - Principle 5 – Use open data structures
    - Principle 6 – Use open interfaces for model linking

2) Testing is a critical activity and must be formally recorded as part of the project deliverables. Testing must cover both the software implementation and the underlying methods (eg new algorithms) and should also generate knowledge which can contribute to more general guidance on when to use the software.

3) Adequate documentation is required in order to help disseminate (internally and/or externally) the new ideas that have been generated during the research. Advice on producing good documentation is provided including the use of 'pseudo code' to supplement equations.

4) It is recommended that a schedule of rights is drafted at the start of a project and is regularly updated during the project. This will help clarify intellectual property rights. A template for the schedule of rights is provided below.

Name and description of item	Extent of use in the Project	Background or foreground knowledge	Proprietary owner of IPR	Signed and dated by:	
				Research assistant	Supervisor

5) Software is rarely static; source code will evolve during and after a specific research project. In order to control different versions of source code, it is recommended that a version control system (such as Subversion) is implemented at the originating institution.

6) Consideration needs to be given to enabling access to the software (including documentation and source code). Some of the most successful 'research' software is directly and openly accessible from the web.

Table of Contents

1 Introduction

Technical software is a key tool used in the analysis phase of flood risk management and such software can be used for many functions, such as:

- retrieving, storing, visualising and quality controlling input data
- processing data to make predictions (eg flood modelling of climate change impacts)
- helping to select 'solutions' to better manage flood risk
- visualising and dissemination of outputs

The research community has made and will continue to make significant contributions to the development of improved software for flood risk management. However there are numerous examples of where such developments have failed to achieve their potential for improving the flood risk management process. There are various reasons for these failures, including those related to barriers to collaboration (eg inadequate documentation, lack of access to modules and their source code), lack of evidence of suitability for particular purposes (eg lack of appropriate testing/benchmarking), and IT systems issues (eg being incompatibility with users IT hardware/software or requiring access to unavailable third-party software). This report draws together guidance which, if applied, will increase the likelihood that flood risk management research which includes development of software will benefit the flood risk management process and thus help to reduce the negative impacts of flooding on people, the economy and the environment.

The report is structured as follows:

- Chapter 2 – describes key generic software development principles, such as modularity, prototyping and openness
- Chapter 3 – provides guidance on testing procedures, both for scientific accuracy and for software 'robustness'
- Chapter 4 – provides guidance on how to document software with a particular emphasis on fully describing new algorithms that may be embedded in research software
- Chapter 5 – contains guidance on how to manage different versions of source code, including controlling access and background and foreground intellectual property.
- Chapter 6 – conclusions and recommendations

This report draws on report FD2121 from the joint Defra/Environment Agency Flood and Coastal Erosion Risk Management R&D programme entitled "R&D Software Development Projects - Guidance for Research Contractors". The FD2121 report was produced to guide those involved in software development for the joint programme and is thus more strongly aligned with Environment Agency IT standards than the guidance contained in this report. Readers are encouraged to read the FD2121 report and it can be obtained by searching for FD2121 on the Defra website www.defra.gov.uk/environ/fcd/research.

The target audience for this report is flood risk management researchers whose primary background is scientific rather than software engineering. The language and terminology used in the report reflects this target audience.

2 Software Architecture

2.1 Introduction

Various types of software will be developed through flood risk management research projects:

1. Basic research where the focus is on the development of new methods that are likely to be 'far from market' - the software is primarily intended to help develop the method and it is acknowledged that a subsequent development phase will be required before the software can be used by others. Generally aligned with 'blue sky R&D'.
2. Research and development projects where new or enhanced methods are implemented in a software product and it is a secondary objective of the project that the software can be used by others (but, for example, the specific end users are not well specified). Generally aligned with 'proof of concept R&D'.
3. Research and development projects where the new or enhanced methods must be implemented in software products that can be used by specific users (for example on Environment Agency systems). Generally aligned with 'targeted R&D'.

This chapter contains guidance on some generic aspects of software architecture that are particularly relevant to flood risk management researchers who develop software in the above three categories. The guidance is primarily aimed at the first two categories - it is expected that the project specifications for the third category will be more prescriptive than this guidance in terms of IT requirements.

2.2 General principles

This section summarises a set of general software development principles which when combined with the testing, documentation and other recommendations contained in this report, would facilitate production of flood risk management research software with a life beyond the initial research project. Further information on the principles is provided in subsequent sections.

Principle 1 – Develop modular software (separation of specific tasks into modules which can be developed and updated independently and reused in other systems).

Principle 2 – Write self-documenting source code (use meaningful variable names and in-line comments within structured, modular code etc).

Principle 3 – Provide flexibility in computer hardware requirements (the software can run on a range of computer hardware and the user decides whether to run the application from one or across multiple computers)

Principle 4 – Use standard development languages (currently Java, C# / .NET, C++, Fortran and Visual Basic can be considered standard in the flood risk management research community).

Principle 5 – Use open data structures (eg XML)

Principle 6 – Use open interfaces for model linking (eg OpenMI)

2.3 Modularity – Logical Deployment Architecture

Principle 1 – Develop Modular software

Complexity of source code can be reduced by building clear and simple modules that undertake specific processing tasks. The modules can be developed and updated independently and reused in other systems.

It is also recommended that software should be developed using an n-Tier approach so that user interface, business logic (program logic / analysis engine) and data management are decoupled (clearly separated). The advantages of this architecture are that it promotes a modular approach, allows changes to be made to any of the layers with minimum impact on other layers and facilitates the different layers being distributed across different hardware. Where it is not appropriate to use a formal n-Tier approach, then it is still recommended to keep separate the user interface, analysis and data management functionality.

Consideration should be given to a thin-client user interface running within a web browser. It such a user interface is unable to provide sufficient functionality then a rich client interface may need to be used.

An objective of the software design should be to develop modular, easily extensible and reusable software. The required level of 'granularity' of the modules is hard to define in general terms – the modules should encapsulate specific pieces of knowledge/process.

Where compiled modules are appropriate for reuse then the software component interface should be well-defined and formally documented and, preferably, publicly available. There are a range of approaches for passing data and control between such modules and the following are recommended:

- Reading/writing data to disk (eg using XML format files)
- OpenMI (www.openmi.org)
- Bespoke software component interface

Consideration should be given to the use of object-oriented design techniques and design patterns.

Principle 2 – Write self-documenting source code

Source code should be self-documenting and well structured. Human-readable names should be used (eg for functions, class, methods, variables etc). In-line comments should be used where the purpose of the source code is not clear from the names and code structure.

As an example of suitable names, the following Java code extract is a method that calculates direct economic flood damage for a property based on the Multi-coloured Manual (MCM) property code (MCMCode), water depth at the property (PropertyDepth), property threshold level adjustment (ThresholdAdjust) and plan area (Area) of the property.

```
dblPropertyDamage = dc.CalculateDamage(strMCMCode, dblPropertyDepth, dblThresholdAdjust, dblArea);
```

2.4 Physical Deployment Architecture

Principle 3 – Provide flexibility in computer hardware requirements

Physical deployment architecture refers to how the application is deployed on computer hardware. It covers both the 'types' of computer the software will run on (eg standard Windows/Intel box, Apple, Unix/Linux workstation), and whether the application can be distributed across multiple computer hardware (eg a so called 'n-tier' application could have the user interface, analysis engine and data all residing on separate hardware, with the analysis engine further distributed across many processors). It is recommended that, where possible, the system is designed so that end users can decide how they would like the system deployed.

2.5 Development Tools and Languages

Principle 4 – Use standard development languages

Standard development languages should be used wherever there is the possibility that the software source code may need to be viewed or modified by people other than the originator. The popularity of development languages can change rapidly and at the time of writing (2008) the following development languages can be considered to be in widespread use for implementation of flood risk management calculations and can thus be considered 'standard': Java, C# / .NET, C++, Fortran and Visual Basic. The Environment Agency has selected Java as its standard development language and where appropriate this is recommended for flood risk management research software. There are many further development languages which may be appropriate for particular uses, for example MATLAB for prototyping scientific methods, and languages specifically for user interface and web application development.

For 'basic research' projects (class 1 in Section 2.1) the emphasis should be on using development languages which are readily readable by humans and facilitate the trialling of ideas (rather optimisation of run times or compatibility with standards) and as such environments such as MATLAB may prove most appropriate for many researchers. It should, however, be understood by project stakeholders that a full rewrite is likely to be required before the software becomes 'operational'.

2.6 Data Management

Principle 5 – Use open data structures

Data form essential elements of all flood risk management research software and the potential sustainability of the software can be greatly enhanced through careful consideration of how the software interacts with the data. Where possible 'open' data structures should be used (here 'open' means that technically others have full access to the data, irrespective of whether data licencing is required). New 'closed' data structures should not be introduced.

The software developer will need to consider whether a formal database system is used (such as MySQL, Oracle or SQL Server) or whether non-database formats should be used (such as XML).

2.7 Open Systems

Principle 6 – Use open interfaces for model linking

Flood risk management analysis involves a complex series of processes and it is likely that a specific item of flood risk management research software will be designed to address only part of the series of processes. Thus the software item will need to take its inputs from precursor processes and provide its outputs to subsequent processes. Also, some software items may need to function in parallel with other processes sending and receiving information during the processing (eg there may be hydraulic feedback between a new surface flooding model and an existing subsurface drainage model).

To enhance the flexibility and potential sustainability of the flood risk management research software it is recommended that open interfaces are used for linking the new software to existing (or future) software items. The choice of open interface is the same as described under the section above on modular software and includes:

- Reading/writing data to disk (eg using XML format files)
- OpenMI (www.openmi.org) - an interface standard that provides a protocol to explicitly describe, define and transfer (numerical) data between model applications that run in parallel.
- Bespoke linking interface with formal (and publicly available) documentation on how to use the interface.

3 Testing

3.1 Introduction

All software requires testing throughout the development process, with the procedures required changing as the software moves through the development process. While this may at first appear to be a linear process, moving from software development, through verification, validation and testing, to distribution, an interactive model is more representative. For example, Figure 3.1 shows the three phases of numerical modelling and simulation. The mathematical model is created by analysis and observation of real physical systems and comprises the mathematical equations that describe the physical system with any modelling assumptions. The computer model implements the mathematical model and includes the code and the data input into the code, such as geometry data, initial and boundary conditions. Spanning these relationships is the guidance (documentation, training, test cases etc.), which should communicate the essence of the modelling process to the user. Communication of the testing undertaken as part of the model development process will serve to enhance confidence in the model or allow users to assess the reliability of model results.

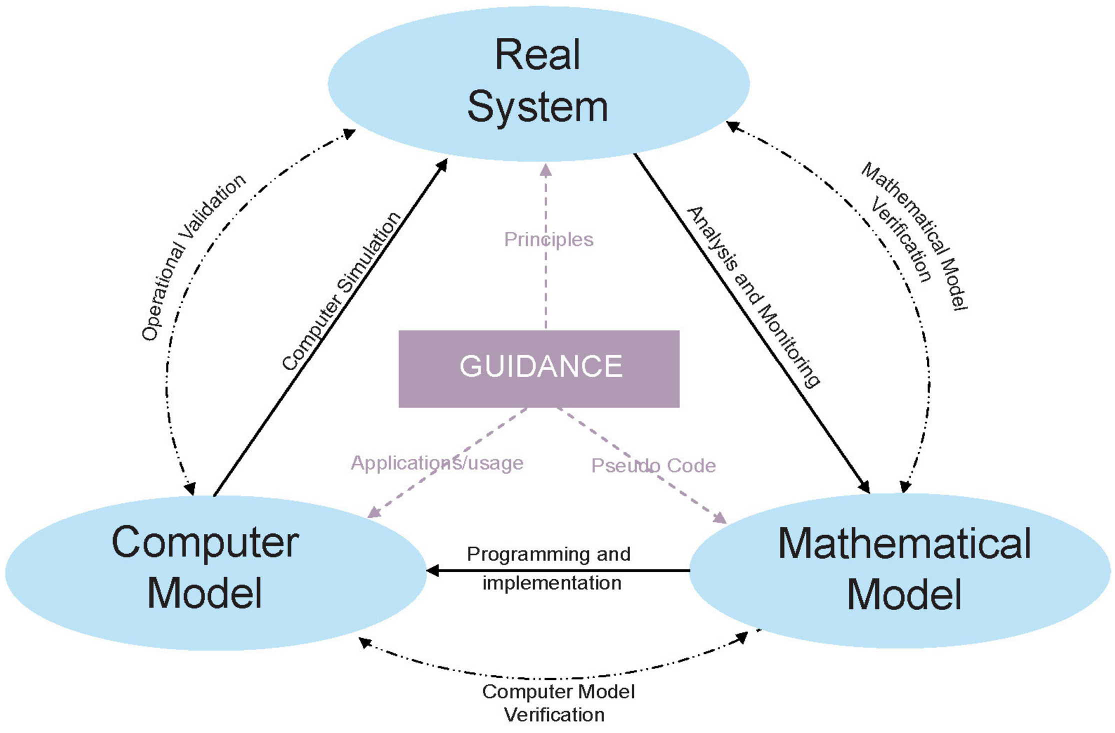


Figure 3.1: Schematic of the model development and testing process.

The credibility of any numerical simulation relies upon a thorough evaluation of the intermediary links between the three stage model strategy. This is primarily achieved through verification and validation. Verification involves testing the ability of the discrete computational scheme to provide an accurate solution of the underlying differential equations. Whether or not those equations and that solution bear any relation to the physical problem of interest is the subject of validation. In this report, software testing encompasses verification and validation, along with software testing:

(i) Verification: evaluating the assumptions implicit in transforming the mathematical model into an algorithm capable of deriving an approximate solution.

(ii) Validation: evaluating the effect of both the approximate nature of the solution to the governing equations, and the assumptions implicit in those governing equations as approximate representations of the real physical system. Validation aims to quantify the discrepancies between model predictions and the real physical outcomes we are trying to predict.

(iii) Software testing: this tests the suitability of the model as an operational tool for the investigation of real world systems. It thus encompasses many concepts such as reliability, ease of use and computational efficiency.

The procedures above ignore the issue of mathematical model verification (dashed line on the right of Figure 3.1), as in this report we are more concerned with the comparison of model results with the real physical system. This represents the errors or discrepancies between our model output and the real physical outcomes we are trying to represent.

The chapter is intentionally biased towards testing numerical hydraulic and hydrological modelling software, although much of the content is also applicable to other flood risk management software such as database methods, GIS and other analysis tools.

An important aspect of testing is to automate the tests where possible. The initial time taken to develop the test suites will be recouped many times over during the software development and testing process as errors are uncovered early and ‘automatically’. Creation of the automated tests will help formalise the expected inputs and outputs of each module of the code and thus contribute to the documentation. Test harnesses, such as JUnit for Java, are available to provide a unit testing framework.

3.2 Verification

This is the first phase of model testing, and tests whether the model does what the developers intended it to (the verification procedure is often summed up in the question “did we build it right?”). For hydraulic and hydrological models, software is often a means of solving a governing set of equations (e.g. the shallow water equations), in which case checking model output against other solutions of these equations is a valid approach, with additional checks that, for example, mass is conserved and the solutions behave as expected.

3.2.1 Analytical Solutions

The model should be tested against exact solutions of the governing equations where possible. The selection of an analytical test case for model comparison is not straightforward, as it requires a situation both simple enough for the equations to be solved “by hand”, but also interesting enough to be a stern test of model performance, with no important processes omitted.

There are many error measures that can be used to compare the model to the analytical solution, a minimum set of the mean, RMS error, and maximum value is recommended, along with an interpretation of the errors found.

3.2.2 Conserved quantity checks

There may be certain physical principles that a model solution should not violate, such as conservation of mass, and measurement of these quantities is a useful check of solution quality. The acceptable level of error should be related to physical quantities, such as: error in the boundary conditions, change in model predictions resulting from such errors, magnitude relative to other processes ignored by the model. How mass balance figures are calculated must also be reported clearly.

Energy conservation may also be a useful test, and it is useful in picking out solutions with unphysical (but still mass conservative) oscillations.

3.2.3 Qualitative tests

Model outputs can also be tested qualitatively, seeing if the solutions “look right” based on the experience of the developers. While not requiring an analytical solution, this still relies on some understanding of the way solutions to the governing equations behave.

Sensitivity analysis can be used to investigate a model’s response to changes in input parameters, as a model should reflect the real system’s response to uncertainty in the physical properties that the parameters represent. The model’s response to changes in numerical parameters such as timestep and grid size should also be tested for the expected behaviour.

3.3 Validation

Validation tests a models ability to represent the real world, rather than its abstraction as governing equations (summed up as “did we build the right thing?”).

3.3.1 Benchmarking

Benchmarking is the process of testing one model against another, and can be a useful method of validation if a solution from a known and trusted model is available and can be taken as “truth”. The benchmarking procedure is especially useful if the two models being compared use very different representations of physical processes (e.g. comparing 1D and 2D models), and allows many different scenarios to be investigated irrespective of the availability of ground truth or analytical solutions. It is thus well suited to complex or long return period situations.

The results of a benchmarking exercise must be interpreted carefully, as comparing two models with similar formulations or solving the same equations would be expected to give similar results. Benchmarking then reduces to a verification exercise.

The measures listed in section 3.2.1 are again recommended for comparison of model results, with care taken if interpolation is necessary.

3.3.2 Testing Against Observed Data

The ultimate test of a model is a comparison with real world data. Possible data sources that should be considered are:

- Time series data from flow gauges, soil moisture probes etc
- Point measurements of water elevation and extent from post flood event survey
- Flood extent from aerial photography and satellite imagery
- Historical records of flood levels and extent (these differ from post event survey in that they are not measured as part of a data collection exercise)

The following important issues must be addressed in comparing model output with real world observations of any type:

- The comparison must be quantitative, as qualitative comparisons (such as "the model fits the observed data well") are subjective and may be prone to bias toward certain models.
- The comparison must look at the difference between model and observations relative to the likely error in the observations.
- The comparison should measure the "skill" of the model, i.e. whether a much simpler approach gives equally good results.
- If the validation data are also used for calibration, the results should be treated with caution.

Again a number of measures can be used to compare model results with observations. Mean and RMS differences between modelled and observed water levels (preferable to depths with are more sensitive to topography) can be measured, and observed extents compared using measures of fit (such as the area predicted correctly as wet by the model relative to the area either predicted or observed as wet).

These issues also affect comparison with laboratory measurements, but with the added consideration that a scale model will only approximate the full scale system.

An important aspect of both verification and validation is to generate knowledge on the model behaviour – it is important that relevant parts of this knowledge are made available to users of the software through incorporation in guidance and other documentation (Chapter 4).

3.4 Software Testing

This phase tests both whether the software is actually implementing the model we think it is, and how easy it is to use. This is often the most difficult and time consuming stage in the development of a model, and is more generic to all software development projects, rather than the more flood specific issues described above.

The standard elements of IT testing include:

- Unit testing – to ensure that each unit or component of the software system performs reliably in isolation (where reliable means it meets detailed, predefined requirements). As mentioned in Section 3.1, extensive use of unit testing in an automated test harness will be of considerable benefit to the developer of the code and to anyone who wishes to use or modify the code subsequently.
- Integration testing – to ensure that when the components are assembled together then the links between components function correctly
- System testing (called 'factory acceptance testing') – testing of the full system away from the end user installation (ie in the 'factory' where the software is developed). Testing includes compliance with both functional and non-functional requirements. Non-functional requirements can include attributes such as run times and security, while functional requirements specify specific functions such as the result to expect when two data items are combined. (Also known as alpha testing.)
- Site acceptance testing – final testing of the software on the end user IT system. It is usual for this phase of the testing to be undertaken by end user representatives. (Also known as beta testing.)

The approach taken to software testing will depend on the requirements of the research project. Testing for basic research should focus on the verification and validation of the new method, and may not include much software testing. The product may be difficult to use, require "hacking" to use for each test case, and have little in terms of user interface. Type 3 software (ie that intended to be delivered to specific end users) must also cover the standard testing cycle of unit, integration, system and site acceptance.

A key objective of testing should be to generate knowledge which can then be communicated to potential users of the software to enable the users to assess whether the software is likely to be fit for particular purposes.

It is very important that the actual testing that has been undertaken is clearly documented and the documentation should also state what testing has not been undertaken. It is recommended that testing log sheets are used to record and communicate the testing activities, an example format is provided in Table 3.1.

Table 3.1 Example Format for Test Documentation Summary

Test Documentation		
Software item:		
Version:		
Tested by:		
Date tested:		
Hardware used:		
Operating system used:		
Summary of test outcomes:		
Tested functionality	**Pass / Fail / Not tested**	**Comment**

4 Documentation

4.1 Introduction

The primary output of much flood risk management research involving software is the development of ideas which can be expressed as algorithms. There is a danger that these ideas will be 'lost' in software which is inaccessible, eg due to formal copyright restrictions, lack of open access to source code, or because the source code is poorly structured or documented. Even when contractual conditions limit the access to the new knowledge, it is still imperative that the algorithms are fully documented so that colleagues working at the initiating research institute can access the knowledge. Thus formal documentation of flood risk management research software is vitally important. This chapter provides guidance on how to document the algorithms contained in flood risk management research software. In addition to the algorithm documentation, it is also recommended that 'user guide' type documentation is also provided to assist users applying the software (but guidance on such documentation is not provided in this document).

4.2 Principles

It is recommended that flood risk management researchers conform to the following key principles when producing algorithm documentation.

- The documentation should be sufficient a competent scientist/engineer to implement the algorithm, in any language and hardware platform.
- They should complement other research outputs (journal articles, reports). Some information on the supporting theory should be included, so that the document can stand alone as a description of the method. Lengthy derivations etc should, however, be avoided – these are best left to academic journals and full reports.
- The descriptions should be structured with background knowledge, foreground knowledge, implementation and pseudo code sections (see section 5.2 for a definition of background and foreground knowledge). Flow diagrams may be useful approaches for showing sequences of workflows, however, they are not sufficient to fully define methods.
- The description should aim to explain how the algorithm works, as well as how to implement it.
- The difference between background (pre current contract) and foreground (generated during the current contract) knowledge should be clear
- The implementation section should include features and rationale that may not be readily apparent from the pseudo code
- Psuedo code should be programming language independent (ie avoid language specific constructs and syntax where possible). Structuring features such as subprograms can be used as they help write less cluttered pseudo code.
- Information should be included on valid (or tested) ranges on input data – where automated test suites are used then the unit test data can be used to provide these ranges (See section 3.1).

4.3 Illustrative Example – LISFLOOD-FP Adaptive Timestep

The following illustrative example shows how foreground knowledge (enhancing LISFLOOD-FP by adding an adaptive timestep capability) can be fully documented in the context of background knowledge (ie the version of LISFLOOD-FP which existed before the new research was implemented).

Note that formal references to appropriate scientific literature should be included in the documentation (but have been removed from the following example).

<u>Background knowledge</u>

LISFLOOD-FP is a cellular floodplain flow model jointly developed by the University of Bristol and the EU Joint Research Centre. It was intended to be a conceptually simple and computationally economical way of calculating flood depths and extents.

Essential features:

- Floodplains discretised as a regular square grid
- Flow between cells calculated from depths and water surface slopes using Manning's equation
- Inertia and advection terms ignored
- Depths updated explicitly at each time step
- Solution conserves mass during wetting and drying
- Can be linked to 1D channel flow model

The objective of the new research was to enhance the LISFLOOD-FP method by developing an adaptive timestep capability which would enable improved stability and/or reduced run times (testing of the original code revealed stability problems, which increased for small grid cell spacings - these problems manifest themselves as "chequerboard" oscillations).

The usual way of assessing the stability of an explicit shallow water flow model is to use the Courant-Friedrichs-Levy (CFL) condition:

$$\frac{c\Delta t}{\Delta x} < 1 \qquad [1]$$

Where Δx is the grid cell spacing, Δt is the timestep, and c is a propagation velocity. This velocity, which measures the rate at which disturbances in the flow field can propagate, is based on either the shallow wave velocity, the fluid flow velocity, or both, depending on hydraulic conditions. This condition is necessary but not sufficient to ensure stability, since it relies on a linearised approximation to the shallow water equations. The inequality is used to determine a timestep by choosing some fraction of the right hand side of equation [1], typically 50% or 75%.

The CFL condition is not applicable to models like LISFLOOD-FP, whose behaviour is diffusive, rather than wave-like. To maximise the efficiency of LISFLOOD-FP, we therefore need a way of estimating the maximum stable timestep (which changes with changing hydraulic conditions), and then to use this method to varying the timestep automatically during a simulation.

Foreground knowledge

An equivalent of [1] for diffusion models can be derived from a von Neumann stability analysis, and is given by:

$$\Delta t \leq \frac{\Delta x^2}{4\alpha} \qquad [2]$$

Where α is the diffusion coefficient, which can be derived from an approximate form of the shallow water equations as:

$$\alpha = \frac{h_{flow}^{5/3}}{2n}\left|\frac{\partial h}{\partial x}\right|^{-1/2} \qquad [3]$$

Where h_{flow} is the flow depth, h is the water surface elevation, and n is Manning's coefficient of roughness. To ensure model stability everywhere in the model domain, we need to determine the maximum value of α, and use this to calculate the timestep.

One final difficulty is that α becomes infinitely large as ∂h/∂x goes to zero (ie for areas of flat water), causing the timestep to become zero, and for the model to become effectively stuck in time. We deal with these areas by defining a small height h_{lin} (typically 10cm), and for neighbouring cells whose water surface elevation differs by less than this, flow is calculated as a linear function of water surface slope, rather than by Manning's equation. The timestep condition for the linearised flow equations is:

$$\Delta t \leq \frac{\Delta x^2}{4}\left[\frac{n}{h_{flow}^{5/3}}\left(\frac{h_{lin}}{\Delta x}\right)^{1/2}\right] \qquad [4]$$

This prevents the timestep going to zero, and is only applicable to areas where the variation in surface elevation between neighbouring cells is less than h_{lin}. We thus have two stability criteria to check before coming up with a timestep that will guarantee stability throughout the domain. The inequalities are used to determine a timestep of 75% of the right hand side of equations [3] and [4], whichever is the smaller.

Implementation

- The main difference between the adaptive and non-adaptive version is the addition of loops to calculate the minimum timestep from the water surface elevations.
- Boundary conditions must now be interpolated “on the fly” as we don’t know at the start of the run what the timestep will be.
- Similarly, model output is made according to a list of times (e.g. every 300s) rather than every 100 iterations.
- The wetting and drying algorithm works by scaling any flows that would otherwise cause negative depths to appear at the next timestep.

Pseudo Code

The text box below provides an algorithmic summary of the new method generated by the research. When read in context of the above descriptions it contains sufficient detail for third parties to implement the method in their own code.

Example Pseudo Code

```
Time=0

Initialise array of water depths H e.g. to give uniform water surface
elevation

Initialise arrays of boundary condition from files

Set value of Hlin (typically 10cm)

Loop while time is less than end time
{
      Use Manning’s eqn to calculate flows in x and y directions Qx
      and Qy

      Interpolate boundary conditions to current time

      At boundary points, replace Qx Qy with values from boundary
      conditions, channel etc

      For each edge between cells, calculate minimum of:
      {
            If water surface elevation difference > Hlin then
                  Calculate timestep using eqns 2+3
            Else
                  Calculate timestep using eqn 4
      }

      For each cell
      {
            Sum Qx and Qy for all four edges

            If this will make cell dry with negative depth at next
            timestep, scale the four Q values so that the cell depth
            will go exactly to zero
      }

      For each cell
      {
            Sum Qx and Qy for all four edges, taking care to get signs
            right e.g. all four values are flows into the cell

            newH=H+(Qx1+Qx2+Qy1+Qy2)*timestep/cell_area
      }

      If time corresponds to one of the output times, save values of H

      Time=time+timestep
}
```

5 Ownership, Version Control and Access

5.1 Introduction

The previous chapters have provided recommendations covering software architecture, testing and documentation which will facilitate the production of flood risk management research software that has the potential to improve the flood risk management process. However, there are many other softer issues which influence likely success of the flood risk management research (in addition to the obvious one concerning delivery of new underlying methods that are 'better' than existing methods). This chapter provides guidance on the key softer issues of ownership, version control and access to source code and executables.

5.2 Ownership

The ownership of software developed in an academic or industrial establishment will be defined by the contracts (research contracts and terms of employment) under which the software is developed. It is very likely that the person(s) actually developing the software will have no ownership rights over the software - this is normal practice. For example, software developed under the Flood Risk Management Research Consortium funding is owned by the institution at which the software was developed (although certain rights are also provided to the FRMRC funders and others).

Contracts usually define ownership and other rights using the following terminology:

- IP - Intellectual Property such as methods, algorithms and other know how. IPR is the Intellectual Property Rights – and defines who has the rights to use (and/or licence) the IP.

- Background knowledge - This is the knowledge (IP) that either existed before the current contract or is generated from other contracts but then used in the performance of the contracted services. Thus algorithms or source code that existed before the current contract but are reused in the current contract would be categorised as background knowledge. Also know as Prior Rights. It is usual for the prior rights to remain the property of the party introducing them to the project.

- Foreground knowledge - This is the knowledge (IP) generated as part of performing the contracted services. Thus new methods which are implemented in new source code would be categorised as foreground knowledge. Also know as Resulting Rights.

It is important that a schedule is kept of background and foreground knowledge used or developed by a flood risk management research project. Use of such a schedule will help protect the background rights of research assistants and supervisors – for example, individuals may be specifically appointed to a project because of their background knowledge and IPR and these rights can be formally recorded at the start of the project using the schedule. In addition to helping to define IP issues, the schedule can also help the researchers to document what value they have added. The schedule should be drafted at the start of the project and should be updated during the project (eg every three months). It should be a comprehensive, verifiable and current inventory of project inputs and outputs that attract IPR (these will include data and methods/source code). An example template schedule is shown in Table 5.1.

Table 5.1 Schedule of Background and Foreground Knowledge and Rights

Name and description of item	Extent of use in the Project	Background or foreground knowledge	Proprietary owner of IPR	Signed and dated by:	
				Research assistant	Supervisor

Even if the originator wishes to make the software freely available to everyone then it is still recommended that a schedule of rights is drafted and then, if the contractual conditions allow, the owner can then either put the software into the public domain uncopyrighted, or (preferably) can copyright the software then use a standard licence to enable others to use and change the software in a more managed way. A full list of accredited open source licences is provided at www.opensource.org/licenses - the GNU General Public Licence www.gnu.org/copyleft is an example of an open source licence in widespread use.

5.3 Version Control

It is recommended that a version control system is used to manage the evolving versions of source code that are usually generated as part of a flood risk management research software project. Version control systems are also know as revision control or source control systems and they provide a structured way to store different versions of source code (and other documents). With a well implemented version control system it is easy to track back to determine when specific changes have been carried out and who implemented them. It should also be easy to regenerate 'old' versions of the software from stored versions of the source code. There are both centralised and distributed version/revision control systems and examples include Subversion, SourceSafe and darcs. Where a team is working on the same software product it is important that standardised methods of working with the version control system are developed and implemented – training may be required.

5.4 Enabling Access to Source Code and Executables

Consideration needs to be given to how the software is to be made available to others. Options include:

- The originating institution organises storage of the software and provides either restricted access or public access to the software (executables, documentation, source code).
- Software is stored in a central flood risk management research site such as the FRMRC website (www.floodrisk.org.uk) or links provided on the site to an alternative storage site.
- The software is uploaded to a general public site such as sourceforge.net.

When software is made available to third parties it is important that licensing and liability issues are defined. Use of a standard licence, such as the GNU General Public Licence, will include relevant clauses covering licensing, liability and warranty. A further issue that needs to be clear when making software available to others is whether or not support and maintenance will be provided.

6 Conclusions and Recommendations

This report provides guidance for improving the process of software development for flood risk management research. Key recommendations are:

1) Software should be designed and coded in accordance with the following principles:

    - Principle 1 – Develop modular software (separation of specific tasks into specific modules which can be developed and updated independently and reused in other systems).

    - Principle 2 – Write self-documenting source code (use meaningful variable names and in-line comments within structured, modular code etc).

    - Principle 3 – Provide flexibility in computer hardware requirements (the software can run on a range of hardware including distributing the application across multiple computers where appropriate)

    - Principle 4 – Use standard development languages (currently Java, C# / .NET, C++, Fortran and Visual Basic can be considered standard in the flood risk management research community).

    - Principle 5 – Use open data structures (eg XML)

    - Principle 6 – Use open interfaces for model linking (eg OpenMI)

2) Testing is a critical activity and must be formally recorded as part of the project deliverables. Testing must cover both the software implementation and the underlying methods (eg new algorithms). It is also strongly recommended that guidance is produced to help others decide when use of the software is appropriate.

3) Adequate documentation is required in order to help disseminate (internally and/or externally) the new ideas that have been generated during the research. Chapter 4 provides advice on producing good documentation and recommends the use of 'pseudo code' to supplement traditional reporting of equations.

4) Those working on flood risk management research software must understand the ownership issues associated with their research. It is standard practice for the intellectual property rights generated during research projects to be owned either by the research institution or by the funding body (ie not by the individual). In order to safeguard any prior rights (including those of the individual) it is important that a schedule of rights is drafted at the start of a contract and is updated during the project. A template for the schedule of rights is provided in Table 5.1.

5) Software is rarely static; source code will evolve during and after a specific research project. In order to control different versions of source code, it is recommended that a version control system (such as Subversion) is implemented at the originating institution.

6) Consideration needs to be given to enabling access to the software (including documentation and source code). Some of the most successful 'research' software is directly accessible from the web.